

MASTER-THESIS

in der Fachrichtung

Informatik

Structural Analysis of Unknown RDF Datasets via SPARQL Endpoints

Thomas Holst
Matrikelnummer: 4490780
thomas.holst@fu-berlin.de

Eingereicht am: 3. Mai 2013

Betreuer: Dr.-Ing. Edzard Höfig
Erstgutachterin: Prof. Dr.-Ing. Ina Schieferdecker
Zweitgutachter: Prof. Dr.-Ing. Robert Tolksdorf

Master-Thesis
von Thomas Holst
Berlin, 2013

Version vom 3. Mai 2013
Gesetzt mit L^AT_EX 2_ε

Eidesstattliche Erklärung

Ich erkläre hiermit an Eides Statt, dass ich die vorliegende Arbeit selbstständig und ohne Benutzung anderer als der angegebenen Hilfsmittel angefertigt habe; die aus fremden Quellen direkt oder indirekt übernommenen Gedanken sind als solche kenntlich gemacht.

Die Arbeit wurde bisher in gleicher oder ähnlicher Form keiner anderen Prüfungskommission vorgelegt und auch nicht veröffentlicht.

Berlin, 3. Mai 2013

Abstract

The World Wide Web is currently developing towards a Web of Data with explicit semantics. In this *Semantic Web* knowledge becomes machine-processable, enabling a more intelligent “Global Information Village”. Data for the Semantic Web is modeled using a generic triple-based format, called RDF. Due to RDF’s bottom-up characteristic and the resulting lack of explicit schema information, manual inspection is usually needed to understand the structure of unknown RDF datasets. Furthermore, these datasets are often located on remote servers which can only be accessed using the SPARQL query language. The goal of this thesis is to enable an automated structural analysis of RDF data based on SPARQL queries. We approach our goal in three steps. Based on a survey of related work and a detailed discussion of dataset understanding, we first identify a set of 18 measures for structural analysis. Secondly, we implement these measures in our software RDFSynopsis, following two alternative approaches called Specific Query Approach (SQA) and Triple Stream Approach (TSA). Finally, we demonstrate the structural analysis of RDF datasets along an example use case, and compare our SQA and TSA implementations with regard to query runtime performance; revealing that SQA is clearly superior to TSA, at least for the fuseki triple store used in our benchmark.

Contents

Acronyms	ix
List of Figures	xi
List of Tables	xiii
1. Introduction	1
1.1. Context and Motivation	1
1.2. Goal	1
1.3. Approach	2
1.4. Structure of Thesis	2
2. Background	5
2.1. Semantic Web – Vision and Reality	6
2.1.1. History and Future of the World Wide Web	6
2.1.2. A Web of Data	8
2.2. RDF – the Data Model	11
2.2.1. Triples form Graphs	11
2.2.2. URIs and Vocabularies	12
2.2.3. Serialization Formats	13
2.3. Ontologies – the Meta Models	15
2.3.1. Terms	16
2.3.2. Hierarchies	16
2.3.3. Domain and Range of Properties	17
2.3.4. Inverse and Transitive Properties	17
2.3.5. Expressing Equivalence	18
2.3.6. Advanced Meta-Modeling with OWL	20
2.4. Understanding RDF Data Modeling	21
2.4.1. Levels of Modeling	21
2.4.2. Open World Modeling	23
2.5. SPARQL – the query language	26
2.6. Triple Stores and SPARQL Endpoints	29
3. Structural Analysis of RDF Datasets	31
3.1. Use Case and Central Questions	32

3.2. Requirements and Definitions	33
3.2.1. RDF Datasets	33
3.2.2. Publisher and Consumer	33
3.2.3. Structure, not values	33
3.2.4. Access via SPARQL-Endpoints	34
3.2.5. Summary	34
3.3. Approaches and Related Work	35
3.3.1. Statistical Measures	35
3.3.2. Descriptive Subgraphs	42
3.3.3. Summary Graphs	44
3.4. Discussion and Choice of Measures	46
3.4.1. Entity Domain	46
3.4.2. Information Domain	49
3.4.3. Composition	51
3.4.4. Reusability	53
3.4.5. Chosen Measures	55
4. Design and Implementation of Structural Analysis	57
4.1. Structural Analysis via SPARQL	58
4.1.1. Specific Queries	58
4.1.2. Triple Stream	58
4.1.3. Measure Implementation	61
4.2. Architecture of RDFSynopsis	80
5. Evaluation	85
5.1. Example Use Case Study	85
5.2. Performance	89
5.2.1. Setup	89
5.2.2. Specific Query Approach	92
5.2.3. Triple Stream Approach	95
5.2.4. Comparison of Approaches	98
5.3. Accuracy of Partial Analysis & Random Sampling	99
6. Conclusion	103
6.1. Summary	103
6.2. Results	104
6.3. Future Work	105
A. Namespace Prefixes	107
B. Fuseki Endpoint Configuration	109
C. Command-Line Usage of RDFSynopsis	111
Bibliography	113

Acronyms

API Application Programming Interface.

AST the Austrian Ski Team dataset.

BC Bisimulation Contraction.

BSD Berkeley Software Distribution.

CBD Concise Bounded Description.

CKAN Comprehensive Knowledge Archive Network.

CMS Content Management Systems.

CPU Central Processing Unit.

CSP Coarsest Stable Partition.

CSS Cascading Style Sheets.

CURIE Compact URI.

FOAF Friend of a Friend.

FP Functional Property.

HTML Hypertext Markup Language.

HTTP Hypertext Transfer Protocol.

IFCBD Inverse Functional CBD.

IFP Inverse Functional Property.

JSON JavaScript Object Notation.

JSON-LD JSON for Linked Data.

MSG Minimum Self-contained Graph.

N3 Notation3.

OOP Object-Oriented Programming.

OS Operating System.

OWL Web Ontology Language.

RAM Random-Access Memory.

RDF Resource Description Framework.

RDFS RDF Schema.

SCBD Symmetric CBD.

SKOS Simple Knowledge Organization System.

SPARQL SPARQL Protocol and RDF Query Language.

SQA Specific Query Approach.

SQL Structured Query Language.

TSA Triple Stream Approach.

Turtle Terse RDF Triple Language.

URI Uniform Resource Identifier.

URL Uniform Resource Locator.

voID Vocabulary Of Interlinked Datasets.

W3C World Wide Web Consortium.

WWW World Wide Web.

XML Extensible Markup Language.

List of Figures

2.1. Classic Web and Web of Data	8
2.2. The Semantic Web Layer Cake	9
2.3. Merging of RDF Graphs	12
2.4. Example for RDF Graph Serialization	14
2.5. Different Modeling Levels in RDF	22
2.6. RDF Graph for SPARQL Example	27
3.1. Comparison of Summary Graphs	45
3.2. Example for Class-based Histograms	47
3.3. Implicit vs. Explicit Class Hierarchy	48
3.4. Example for Property-based Measures	50
3.5. Implicit vs. Explicit Property Hierarchy	51
4.1. Sequence Diagram for Specific Query Approach (SQA)	58
4.2. Sequence Diagram for Triple Stream Approach (TSA)	59
4.3. Creating a Triple Stream with SPARQL	59
4.4. Random Sampling TSA	60
4.5. Architecture of RDFSynopsis	80
5.1. Example Use Case: Implicit Class Hierarchy with Instances and Triples	86
5.2. Example Use Case: Property Usage	87
5.3. Example Use Case: Class Instances per Property & Common Properties	88
5.4. Example Use Case: Links to Other Namespaces	88
5.5. Total Runtime (SQA) Grouped By Dataset	93
5.6. Runtime (SQA) in Percent of Total Runtime	94
5.7. Runtime (SQA) Normalized to 1M Triples	94
5.8. Runtime of Single TSA Queries	96
5.9. Runtime of Single TSA Queries (Normalized to 1M Triples)	96
5.10. Runtime of Full TSA Queries	97
5.11. Accuracy of Sequential TSA	100
5.12. Accuracy of Random Sampling TSA	101

List of Tables

3.1. Survey of Statistical Measures	38
3.2. Measures Chosen for Structural Analysis	56
5.1. Datasets Used in Benchmark	90
5.2. System Specifications for Benchmark	91
5.3. Benchmark Results for SQA	92
5.4. Benchmark Results for TSA	95
5.5. Total Runtimes for SQA and TSA (ordered by subject)	98
A.1. Namespace Prefixes Used in this Thesis	107

1. Introduction

1.1. Context and Motivation

The term *Semantic Web* describes an idea for the next evolutionary stage of the World Wide Web (WWW). While the WWW today represents a web of interlinked documents targeted towards human consumers, supporters of the Semantic Web idea favor the development towards a *Web of Data* that makes Web content more *machine-processable*.

The Web of Data relies on a generic triple-based data model, which expresses information in terms of resource nodes and relationship edges. This data model, called Resource Description Framework (RDF), was designed to enable bottom-up modeling. Similar to the way humans express and share knowledge with natural language, RDF relies on consensual rather than centrally (top-down) defined meaning.

Due to the generic data model and bottom-up characteristic of RDF, an unknown RDF dataset may leave a potential consumer in the dark about its content and inner structure. Gaining an understanding of such a dataset can easily become a time-consuming hands-on process.

Furthermore, RDF data is often only available from a server that answers queries formulated in an RDF-specific query language. This language is called SPARQL Protocol and RDF Query Language (SPARQL). Web services that expose RDF data via SPARQL are called *SPARQL endpoints*.

1.2. Goal

The goal of the work presented in this thesis, is to enable an automated structural analysis of RDF data based on SPARQL queries. The ideal outcome of our work would be a software tool which consumers of RDF data could point at any SPARQL endpoint, and receive a concise structural analysis of the underlying dataset in a short time, if not in real-time.

1.3. Approach

We approach our goal in three steps.

1. Identification

We first need to understand, what a “structural analysis of RDF data” really means. In this thesis, we use the notion of a “measure”¹ to refer to a single aspect according to which a dataset could be analyzed. This may be a single value, like the number of triples, or more complex information, such as ratios or frequencies for different categories. The goal of the first step is to **identify** the set of measures that are needed to understand an RDF dataset.

2. Implementation

We need to study, how we can use SPARQL to analyze datasets which are only available through SPARQL endpoints. The goal of the second step is to **implement** the analysis according to the set of identified measures with SPARQL.

3. Evaluation

Finally, we need to investigate whether the measures chosen in the first step provide the desired insight into an unknown dataset, and whether our implementation is efficient enough for practical use. The goal of the third step is to **evaluate** whether we have reached the goal of this thesis.

1.4. Structure of Thesis

The remainder of this thesis is structured in five chapters.

Chapter 2 (*Background*) is dedicated to presenting the Semantic Web idea (section 2.1) and its core technologies. RDF is introduced in section 2.2. Information is expressed in RDF using terms from shared ontologies (section 2.3). Understanding the bottom-up characteristic of data modeling with RDF is important to understand the motivation and challenge behind a structural analysis of RDF datasets (section 2.4). The SPARQL Protocol and RDF Query Language (SPARQL) is introduced in section 2.5. RDF datasets are typically stored in specific databases called triple stores (section 2.6) and are often accessible through SPARQL endpoints.

¹We use the terms “measure” and “criterion” interchangeably throughout this thesis.

The chapters **3** to **5** represent the three steps of our approach: Identification, Implementation, and Evaluation.

In chapter **3** (*Structural Analysis of RDF Datasets*) we study different measures and approaches towards a structural understanding of RDF datasets. First, we define dataset understanding in terms of four central aspects (section 3.1). Then, we define the goal of our work in more detail (section 3.2). In section 3.3 we present related work from three areas, including a survey of 75 statistical measures (table 3.1) which are used in scientific work and practical applications. Based on our findings from related work, we separately discuss each central aspect of RDF dataset understanding, and identify suitable criteria, 18 in total (section 3.4).

In chapter **4** (*Design and Implementation of Structural Analysis*) we describe the implementation of the structural analysis with SPARQL. In section 4.1, we introduce two general approaches, the *Specific Query Approach* (Specific Query Approach (SQA)) and the *Triple Stream Approach* (Triple Stream Approach (TSA)). We present SQA and TSA implementations for all 18 criteria in section 4.1.3. Finally, we describe the architecture of our software RDFSynopsis which implements the structural analysis (section 4.2).

In chapter **5** (*Evaluation*) we evaluate our approaches with regard to three aspects. First, we employ our analytic criteria in an example use case, to demonstrate their suitability for structural analyses (section 5.1). Second, we compare SQA and TSA with regard to their query runtime performance (section 5.2). Third, we look into the accuracy achieved with partial and random TSA analyses (section 5.3).

The thesis ends with chapter **6** (*Conclusion*). In section 6.1 we provide a brief summary of this thesis. In section 6.2 we separately evaluate the three steps of our approach. We conclude with an outlook on potential future work in section 6.3.

2. Background

In this chapter, we describe the Semantic Web idea and its core technologies, thereby setting both context and foundation for the work presented in this thesis.

We first take a historical look at the WWW (section 2.1.1) and describe the vision of a “Web of Data” that underlies the Semantic Web idea (section 2.1.2). We then focus on core technologies developed for that vision, beginning with the abstract data model of RDF, that represents information as triples (section 2.2). To convey meaning, these triples are formulated with terms that are modeled in Ontologies (section 2.3). At first, it can be quite hard to understand RDF data modeling which can be characterized as *Open World* and *Bottom-Up* (section 2.4). RDF data is usually stored in triple stores (section 2.6) and queried via the SPARQL query language (section 2.5).

Many good introductory texts on the Semantic Web and its technologies exist. “Semantic Web for the Working Ontologist”[3], “A Semantic Web Primer”[5], “Programming the Semantic Web”[62], and “Linked Data: Evolving the Web into a Global Data Space”[49] are some that we found useful.

2.1. Semantic Web – Vision and Reality

The term Semantic Web describes an idea for the next evolutionary stage of the WWW. While the WWW today represents a web of interlinked documents targeted towards human consumers, supporters of the Semantic Web idea favor the development towards a *Web of Data* that makes Web content more *machine-processable*.

2.1.1. History and Future of the World Wide Web

In the beginning there were computers. And they were just a few, spread around the world; and all were used in isolation of each other. Then the *Internet* was invented and distant computers could now communicate. Their human users invented a lot of task-specific languages (protocols) for them and were happy to send emails. In the late 1980's Tim Berners-Lee invented the Hypertext Transfer Protocol (HTTP), the Hypertext Markup Language (HTML) and the Uniform Resource Identifier (URI) scheme to combine a general form of hypertext media and communicating computers.

I happened to come along [...] after hypertext and the Internet had come of age. The task left to me was to marry them together.

Tim Berners-Lee, 2000, *Weaving the Web* [15]

This “marriage” led to the World Wide Web as we have today, a vast and distributed collection of Web resources (HTML pages, pictures, videos and any other kind of file) uniquely identified by their Uniform Resource Locator (URL) and accessible via HTTP. The WWW is not only a single technological achievement but a huge success of the human community which has built its “Global Information Village”. The WWW has changed people’s everyday lives and improved access to information and education for many.

Since its birth the Web has matured, its technologies have been refined and become more sophisticated. Web developers use state of the art tools like Content Management Systems (CMS), relational databases and structured file formats like the Extensible Markup Language (XML) to build reliable and professional websites that users and customers would not want to miss.

Still, a group of Web experts and institutions like the World Wide Web Consortium (W3C) itself argue that the WWW as an “information web”[3] does not perform to its full potential. Antoniou and van Harmelen point out that “[k]eyword-based search engines such as [...] Google are the main tools for using today’s Web. It is clear

that the Web would not have become the huge success it is, were it not for search engines. However, there are serious problems associated with their use.”[5] Besides search engines’ difficulties to produce all (“low recall”) and only (“low precision”) the relevant results - without being too sensitive to vocabulary -, the authors mention the problem that results can only be single Web pages. “If we need information that is spread over various documents, we must initiate several queries to collect the relevant documents, and then we must manually extract the partial information and put it together.”[5]

The reason for these problems can be found in the basic fact that the WWW consists of interlinked documents that are made for human eyes. Almost all Web pages contain natural language that is logically structured for an optimized visual presentation in the user’s Web browser, using HTML and Cascading Style Sheets (CSS).

Tim Berners-Lee himself was among the first to wonder if computers could do *more* than just receiving and displaying Web pages, and what was needed to that end.

Some of the developments that [the WWW initiators] look forward to in the next few years include: [...] Evolution of objects from being principally human-readable documents to contain more machine-oriented semantic information, allowing more sophisticated processing.

Tim Berners-Lee et al., 1994, *the World Wide Web* [16]

The first step is putting data on the Web in a form that machines can naturally understand [...]. This creates what I call a *Semantic Web* - a web of data that can be processed [...] by machines.

Tim Berners-Lee, 2000, *Weaving the Web* [15]

Since Berners-Lee wrote this in 2000 many have picked up on the idea of a “web of data”. In their enthusiasm about what the Semantic Web could be and what benefits it would entail, and in order to promote the idea, the proponents used a lot of rather visionary examples. These spoke of “intelligent agents” that would enable “home automation” and self-scheduling medical treatment plans [17]. To some people this sounded more like a science fiction novel than any near-future technology, and thus lead to misconceptions regarding the severity of inherent technological challenges. Instead:

[T]he greatest current challenge is not scientific but rather one of technology adoption. G. Antoniou, F. van Harmelen, 2008, *A Semantic Web Primer*[5]

In the beginning, the World Wide Web had faced similar problems. Although the required technology was already there (HTTP, HTML and URL), people first had to be convinced to create simple HTML documents with links to other documents for their own use, before the WWW vision was realized [15]. In a similar fashion, instead of promoting the Semantic Web idea, people are now simply encouraged to publish their data as “linked data”[13].

[W]hile the Semantic Web, or Web of Data, is the goal or the end result of this process, Linked Data provides the means to reach that goal.

C. Bizer, T. Heath, T. Berners-Lee, 2009, *Linked-Data - The Story So Far*[19]

In a community effort of Semantic Web enthusiasts and researchers, and through the standardization process of W3C, a set of linked data technologies has been developed that could make the Semantic Web vision come true and are already in wide use today. In section 2.1.2 we will take a closer look at this technological side.

2.1.2. A Web of Data

The classic World Wide Web can be described as a web of documents. As visualized in figure 2.1a the Web is made of document *nodes* and directed link *edges* between these documents. Documents (or Web *resources*) can, in general, be any kind of file, and all documents on the Web have a unique identifier, their Uniform Resource Locator. The main kind of documents are Web pages that contain natural language

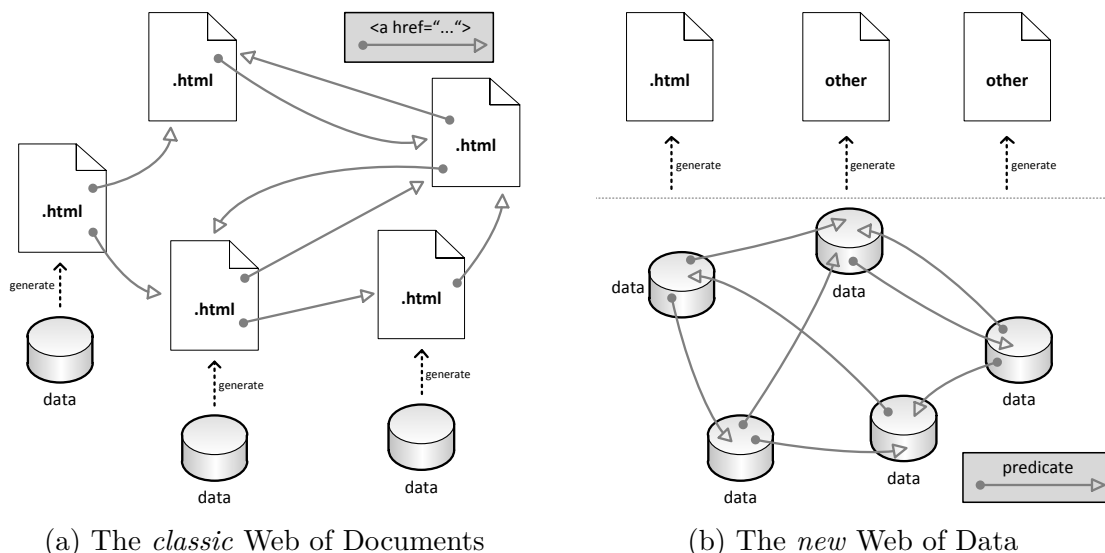


Figure 2.1.: Classic Web and Web of Data

text and are logically structured using the Hypertext Markup Language. In HTML, parts of the logical structure can be marked as referring to another document by using the ``-tag, thereby creating a *link*. Any document with a valid URL can be retrieved from the Internet using the Hypertext Transfer Protocol (HTTP). The user interface to this *web of documents* are Web browsers, a kind of software that is working on top of HTTP, used to retrieve and display documents, and to follow their links through the WWW.

Despite significant developments of Web technology that, for example, allow for the automatic generation of complex Websites from data (as shown in fig 2.1a) or client- and server-side dynamics, the basic WWW system of HTTP, HTML, and URL, has not changed.

In comparison to the WWW, the Semantic Web can be described as a Web of Data. As visualized in figure 2.1b the Web of Data is made of, as the name suggests, *data nodes* and directed link *edges* between them. A data node is usually referred to as a *resource*. Resources can, in general, represent any kind of real or imaginable “thing” - or abstract concept. Similar to the documents in the WWW, all resources have a unique identifier, a Uniform Resource Identifier. In contrast to the *general* links of the WWW, the Web of Data uses *typed links*, called *properties* (or predicates). Like resources, all property types have a unique URI. Consequently, links in the Web of Data are represented as *triples* of URIs. Furthermore, *all* information about *any* resource can be represented by using triples with appropriate properties. This basic data model of triples was specified by the W3C as the Resource Description Framework (RDF).

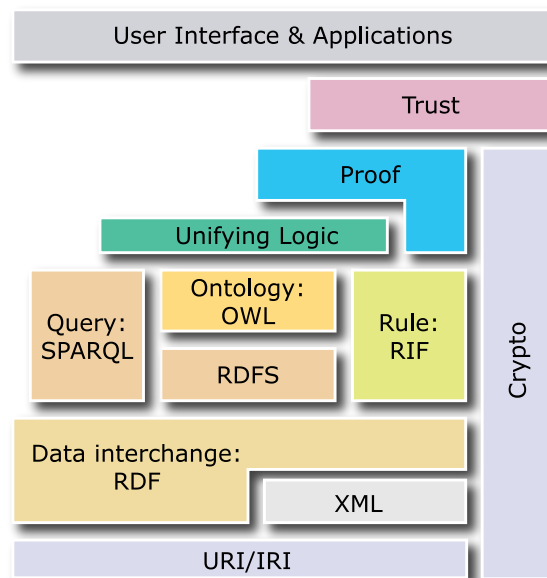


Figure 2.2.: The Semantic Web Layer Cake (source: [74])

Data for the Web of Data is usually published as RDF datasets (the “data” objects in figure 2.1b) containing a large number of triples, and stored in RDF databases, so called *triple stores*. Similar to the way the Structured Query Language (SQL) is used to query relational databases, the SPARQL Protocol and RDF Query Language (SPARQL) can be used to retrieve RDF data from triple stores. When resource and property URIs are *reused* across multiple datasets, the latter can easily be merged and treated as *one* dataset. These two principles, using common global URI identifiers for resources and properties, and using the common RDF data model, provide the “Web” in Web of Data.

The user interface will (in many cases) most probably remain to be the Web browser that displays HTML pages. But, in contrast to the classic WWW, these pages may not be generated from specific, self-contained databases, but from a combined view of many data sources, i.e., from the *Web of Data*.

The technologies that represent the foundation for the Web of Data, are often visualized in a stack, the so called “layer cake” (see figure 2.2).

2.2. RDF – the Data Model

RDF, short for Resource Description Framework, is a “framework for representing information in the Web” [26]. It features a simple, abstract data model that is based on triples; a set of triples is called an *RDF graph* (section 2.2.1). RDF was designed to enable information sharing, merging, and processing by machines; to this end RDF uses globally unique URIs and domain specific sets of terms, so called *vocabularies* (section 2.2.2). For the single abstract data model, different *serialization formats* (i.e., file formats) exist (section 2.2.3).

2.2.1. Triples form Graphs

The basic elements of RDF are *triples*. The same triple can be described in different ways.

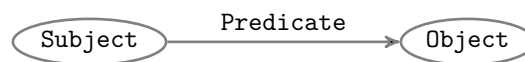
1. As consisting of *Subject*, *Predicate*, and *Object*.

Subject Predicate Object.

2. Predicate and object may be referred to as *Property* and *Value*, respectively.

Subject Property Value.

3. A triple can also be interpreted as part of an RDF graph, i.e., subject and object are *nodes* of an RDF graph, and the predicate is a directed *edge*.



In general, triples represent a relationship (denoted by the *predicate*) between two things (*subject* and *object*). Triples are sometimes referred to as *statements* or as *assertions* of relationships. A set of triples is called an RDF graph. This is due to the characteristic of RDF, that a set of triples with common subjects or objects can be merged into one connected component. For example, the following triples can be drawn as three independent graphs or as one merged graph (figure 2.3).

```

Sub1 Pred1 Obj1.
Sub1 Pred2 Obj2.
Sub2 Pred3 Obj1.
  
```

Subjects, predicates, and objects are URIs (explained below in section 2.2.2). Objects can also be literal values with optional datatype and language tag. The RDF standard [26] by W3C defines *blank nodes*, i.e., nodes without a global URI. They can be used as subjects or objects, but their use for *linked data* is discouraged [49] because they cannot be linked to from external documents and make merging of graphs harder.

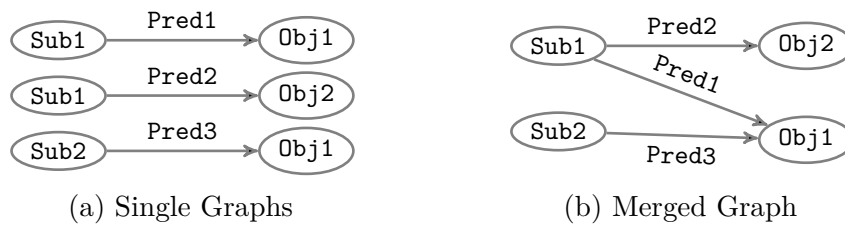


Figure 2.3.: Merging of RDF Graphs

2.2.2. URIs and Vocabularies

The basic idea of the Semantic Web is the use of data with *explicit* semantics, that is, meaning. Because things get their meaning only by convention, a data model with explicit semantics can only be established if it allows for the use of agreed-upon terms. Thus, RDF has to provide the means for people to use globally unambiguous terms and identifiers. For this purpose, RDF uses URIs, because they have two main advantages:

1. *Clear authority:*

The authority part of a URI [54] belongs to a registered owner, who has the sole authority to provide and manage URIs beginning with their authority part.

2. *Unambiguous identifiers:*

Trivially, there is only *one* way to write a specific URI. If two URIs are different then, by definition, they can be referred to independently of each other.

Using URIs, a triple might look like (wrapped around for space reasons):

```
<http://dbpedia.org/resource/Berlin>  
<http://www.w3.org/1999/02/22-rdf-syntax-ns#type>  
<http://dbpedia.org/class/yago/CapitalsInEurope> .
```

This triple asserts that the resource `<http://dbpedia.org/resource/Berlin>` has a `<http://www.w3.org/1999/02/22-rdf-syntax-ns#type>` relationship to the resource `<http://dbpedia.org/class/yago/CapitalsInEurope>`.

The resource `<http://dbpedia.org/resource/Berlin>` was defined by `dbpedia.org` to represent the German city of Berlin, among other cities like Wellington¹ or Nairobi². Apparently the people behind `dbpedia` have created all their resource URIs with the common prefix of `<http://dbpedia.org/resource/>`. A URI prefix like this is also called a *namespace* [22]. In RDF, an abbreviation scheme called Compact

¹`<http://dbpedia.org/resource/Wellington>`

²`<http://dbpedia.org/resource/Nairobi>`

URI (CURIE) [18], is used to make URIs more readable. For this representation, a short prefix is locally defined as equivalent to a namespace (e.g., `dbp` for `http://dbpedia.org/resource/`), and then used followed by a colon and the local reference (e.g., `dbp:Berlin`). A list of all prefixes used in this thesis can be found in the appendix (table A.1).

If we define the following prefixes,

```
@prefix dbp: <http://dbpedia.org/resource/>
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
@prefix yago: <http://dbpedia.org/class/yago/>
```

then the above example in CURIE syntax would read:

```
dbp:Berlin rdf:type yago:CapitalsInEurope .
```

Casting information about real-world entities into triples is, abstractly speaking, a *process of modeling* these entities. This is done according to a *meta-model*, that is made partly of the basic data model, RDF’s triple format, and partly of domain-specific conceptualizations, so called *vocabularies* (or *ontologies*¹). Basically, these vocabularies define the *terms*, i.e., the *classes* (or *concepts*) and *relationships*, that can be used to model a specific domain.

For example, a well-known vocabulary to describe people and their relationships is the Friend of a Friend (FOAF) vocabulary. It defines basic classes (concepts) like `foaf:Person` and properties (relationships) like `foaf:knows`. These terms could be used like this:

```
dbp:Angela_Merkel rdf:type foaf:Person .
dbp:Angela_Merkel foaf:knows dbp:Barack_Obama .
```

Summary. The URI scheme provides globally unambiguous identifiers. Vocabularies are fixed sets of URI terms, classes and properties, that can be used to describe a specific domain. Both, URIs and defined vocabularies, enable a common understanding of the world that is needed for machines to share and process information, without a layer of human interpretation in between.

2.2.3. Serialization Formats

RDF defines an abstract data model of triples that form RDF graphs. In order to save these graphs to files, their structure has to be serialized. For RDF, different *serialization formats* exist. The following example shows the graph in figure 2.4

¹“There is no clear division between what is referred to as ‘vocabularies’ and ‘ontologies’. The trend is to use the word ‘ontology’ for more complex, and possibly quite formal collection of terms, whereas ‘vocabulary’ is used when such strict formalism is not necessarily used or only in a very loose sense.”[75]

serialized in three widely used formats: *Terse RDF Triple Language (Turtle)*, *N-Triples* and *RDF/XML*. There is also a serialization format based on JavaScript Object Notation (JSON) [31] currently in the making, called JSON for Linked Data (JSON-LD) [64]. For brevity, all triple listings in this thesis use the Turtle notation.

■ Turtle [11]

```
@prefix foaf: <http://xmlns.com/foaf/0.1/>
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#>
@prefix owl: <http://www.w3.org/2002/07/owl#>

foaf:Person a1 rdfs:Class .
foaf:Person a owl:Class .
foaf:Person rdfs:subClassOf foaf:Agent .
foaf:Agent a owl:Class .
```

■ N-Triples [43] (*N-Triples* is a subset of Notation3 (N3) [14])

```
<http://xmlns.com/foaf/0.1/Person>
  <http://www.w3.org/1999/02/22-rdf-syntax-ns#type>
    <http://www.w3.org/2000/01/rdf-schema#Class> .
<http://xmlns.com/foaf/0.1/Person>
  <http://www.w3.org/1999/02/22-rdf-syntax-ns#type>
    <http://www.w3.org/2002/07/owl#Class> .
<http://xmlns.com/foaf/0.1/Person>
  <http://www.w3.org/2000/01/rdf-schema#subClassOf>
    <http://xmlns.com/foaf/0.1/Agent> .
<http://xmlns.com/foaf/0.1/Agent>
  <http://www.w3.org/1999/02/22-rdf-syntax-ns#type>
    <http://www.w3.org/2002/07/owl#Class> .
```

■ RDF/XML [12]

```
<rdfs:Class rdf:about="http://xmlns.com/foaf/0.1/Person">
  <rdf:type rdf:resource="http://www.w3.org/2002/07/owl#Class"/>
  <rdfs:subClassOf>
    <owl:Class rdf:about="http://xmlns.com/foaf/0.1/Agent"/>
  </rdfs:subClassOf>
</rdfs:Class>
```



Figure 2.4.: Example for RDF Graph Serialization

¹`a` is a Turtle shorthand for `rdf:type`

2.3. Ontologies – the Meta Models

Information is usually modeled in RDF using *terms* from domain-specific vocabularies (also called ontologies¹), as described in section 2.2.2. These ontologies define a set of terms, that are either classes or properties. In addition, ontologies can also define relationships between these terms, and thus represent a *meta-model* of a domain.

An ontology is a formal, explicit specification of a shared conceptualisation. A “conceptualisation” refers to an abstract model of some phenomenon in the world by having identified the relevant concepts of that phenomenon. “Explicit” means that the type of concepts used, and the constraints on their use are explicitly defined. [...] “Formal” refers to the fact that the ontology should be machine readable, which excludes natural language. “Shared” reflects the notion that an ontology captures consensual knowledge, that is, it is not private to some individual, but accepted by a group. R. Studer et al., *Knowledge Engineering: Principles and methods* [66]

Confusing at first, vocabularies *for* use in RDF models are also defined *in* RDF triples, with terms from the following three standard vocabularies

- RDF[26],
- RDF Schema (RDFS)[23], and
- Web Ontology Language (OWL)[55, 73].

These vocabularies are the result of a W3C standardization process, and they have formally defined semantics [27–29, 48]. For some terms, based on their semantics, new information (in the form of triples) can be deduced (or *inferred*). These *inference rules* can easily be expressed using SPARQL **CONSTRUCT** queries [3]. In the following subsections the basic language features of RDFS and OWL are explained; many of which are presented with their semantics expressed as a **CONSTRUCT** query and an inference example.

¹The term “ontology” is used in the area of knowledge representation in a narrow and technical way, but originally had a broader meaning: “The term *ontology* originates from philosophy. In that context, it is used as the name of a subfield of philosophy, namely, the study of the nature of existence (the literal translation of the Greek word *ὄντολογία*), the branch of metaphysics concerned with identifying, in the most general terms, the kinds of things that actually exist, and how to describe them.” [5, p. 10]

2.3.1. Terms

Basic terms from these standard vocabularies include `rdfs:Class` and `rdf:Property`; they are used to define the terms used in a vocabulary. For example, a vocabulary `music` could define three terms, two of them classes (piece of music and musician) and one property (performs).

```
@prefix music: <http://example.com/music/>
music:Piece rdf:type rdfs:Class .
music:Musician rdf:type rdfs:Class .
music:performs rdf:type rdf:Property .
```

This vocabulary could then be used as a *meta model* to assert information (i.e., create a *model*) about some musicians and their performance of music.

```
ex:OdeToJoy rdf:type music:Piece .
ex:Rihanna rdf:type music:Musician .
ex:Rihanna music:performs ex:OdeToJoy .
```

2.3.2. Hierarchies

Hierarchical relationships among terms of a vocabulary can be defined using the properties `rdfs:subClassOf` and `rdfs:subPropertyOf`. For example:

```
music:Singer rdfs:subClassOf music:Musician .
music:Song rdfs:subClassOf music:Piece .
music:sings rdfs:subPropertyOf music:performs .
```

Software developers used to the Object-Oriented Programming (OOP) paradigm and its typical features like classes, attributes, and inheritance, might find the semantics of these relationships [3] counterintuitive.

```
CONSTRUCT {?r rdf:type ?B}
WHERE {
  ?A rdfs:subClassOf ?B .
  ?r rdf:type ?A .
}
```

That is, no inheritance of attributes takes place, but all instances of a class **A** are automatically inferred to be instances of **A**'s super class **B** as well. The semantics of `rdfs:subPropertyOf` are analogous. So, for example, based on triples

```
ex:Sinatra rdf:type music:Singer .
ex:MyWay rdf:type music:Song .
ex:Sinatra music:sings ex:MyWay .
```

the following triples could be inferred.

```
ex:Sinatra rdf:type music:Musician .
ex:MyWay rdf:type music:Piece .
ex:Sinatra music:performs ex:MyWay .
```

2.3.3. Domain and Range of Properties

The classes of subjects and objects used with a property can be defined using `rdfs:domain` and `rdfs:range`.

```
music:sings rdfs:domain music:Singer .
music:sings rdfs:range music:Song .
```

The semantics for these RDFS language constructs are even more counterintuitive, compared to the OOP world. Instead of imposing a restriction on the use of a property (like method signatures constrain the types of objects that can be used with them, and allow for *type checking*), `rdfs:domain` and `rdfs:range` provide a mechanism to infer the classes of those resources used with the specified property. This becomes clearer from the semantics [3]:

```
CONSTRUCT {?x rdf:type ?D}
WHERE {
  ?P rdfs:domain ?D .
  ?x ?P ?y .
}
```

So, in the following example `ex:Rihanna` is used as a subject in a triple with property `music:sings`, although she was not defined to be of that property's `rdfs:domain` class `music:Singer`.

```
ex:Rihanna music:sings ex:OdeToJoy .
```

Instead of detecting an inconsistency in property usage, the following triples are inferred.

```
ex:Rihanna rdf:type music:Singer .
ex:OdeToJoy rdf:type music:Song .
```

As a convenience, the two classes `owl:DatatypeProperty` and `owl:ObjectProperty` (both are `rdfs:subClassOf rdf:Property`) can be used to distinguish properties that have literals, or respectively, URI objects.

2.3.4. Inverse and Transitive Properties

Often, when modeling information in RDF, it is not obvious in what direction a property should be defined. Take, for example, the ownership of things; it could be modeled from owner to thing (via a property `:owns`), or from thing to owner (`:belongsTo`). In many cases, it is useful to model both directions at the same time. The property `owl:inverseOf` can be used to make the relation between inverse properties (like `:owns` and `:belongsTo`) explicit. Some properties are `owl:inverseOf` themselves; a fact that can easily be expressed by defining the property to be of type `owl:SymmetricProperty`.

The semantics in SPARQL are:

```
CONSTRUCT {?y ?q ?x}
WHERE {
  ?p owl:inverseOf ?q .
  ?x ?p ?y .
}

CONSTRUCT {?p ?owl:inverseOf ?p}
WHERE {
  ?p rdf:type owl:SymmetricProperty .
}
```

So, for example, the following triples

```
ex:owns owl:inverseOf ex:belongsTo .
ex:marriedTo rdf:type owl:SymmetricProperty .
ex:Peter ex:owns ex:DisneyCastle .
ex:Peter ex:marriedTo ex:John .
```

could be used to infer the next triples.

```
ex:marriedTo owl:inverseOf ex:marriedTo .
ex:DisneyCastle ex:belongsTo ex:Peter .
ex:John ex:marriedTo ex:Peter .
```

Some properties are *transitive*, i.e., they apply to all resources that are indirectly related via that property. The notion of containment is a good example; if A contains B, and B contains C, then A also contains C. These kind of transitive properties can be defined of type `owl:TransitiveProperty` which has the following semantics:

```
CONSTRUCT {?a ?p ?c}
WHERE {
  ?p rdf:type owl:TransitiveProperty .
  ?a ?p ?b .
  ?b ?p ?c .
}
```

2.3.5. Expressing Equivalence

In the Semantic Web, things can be globally and uniquely identified by their URIs. This basic and very important feature must not be confused with a situation where each thing has *one* global identifier. The contrary is the case, and the *nonunique naming* concept has been identified as yet another crucial feature [3] of the Semantic Web. That is, when different data publishers talk about the same real world things, they will likely use *different* URIs for *same* things.

This poses a challenge, when data from different publishers need to be merged. A way of expressing equivalence between properties, classes, and their individual instances is needed. This demand is met by `owl:equivalentProperty`, `owl:equivalentClass`, and `owl:sameAs`, respectively. Informally, this means that if two properties, classes or individuals are asserted to be equivalent, these two can be used interchangeably.

The semantics for `owl:equivalentProperty` can be defined in SPARQL as:

```
CONSTRUCT {?x ?q ?y}
WHERE {
  ?p owl:equivalentProperty ?q .
  ?x ?p ?y .
}
```

The mirrored case (inferring P1 triples from P2 triples) is implicitly included, because `owl:equivalentProperty` is of `rdf:type owl:SymmetricProperty`. The semantics for `owl:equivalentClass` can be described analogously.

The property `owl:sameAs` has a more general meaning. Two resources, defined to be `owl:sameAs` each other, are used interchangeably disregarding their position in the triple (i.e., subject, predicate, or object). The semantics for `owl:sameAs` are shown for the subject case, and are analogous for the predicate and object cases.

```
CONSTRUCT {?y ?p ?o}
WHERE {
  ?x owl:sameAs ?y .
  ?x ?p ?o .
}
```

Sometimes, equivalence of two resources can be inferred from other triples. This is the case, when a property can be understood as a *function*, i.e., any subject can only have *one* value for that property. If a subject has two values for such a Functional Property (FP), they must be equivalent. OWL offers `owl:FunctionalProperty` and `owl:InverseFunctionalProperty` to define functional properties. An Inverse Functional Property (IFP) works in the other direction, i.e., it allows only one subject for a property value; if there are two subjects, they must be the same. The semantics can be expressed in SPARQL as

```
CONSTRUCT {?a owl:sameAs ?b}
WHERE {
  ?p rdf:type owl:FunctionalProperty .
  ?x ?p ?a .
  ?x ?p ?b .
}
```

For example, from the following triples

```
ex:hasBiologicalMother rdf:type owl:FunctionalProperty .
ex:Peter ex:hasBiologicalMother ex:Jane .
ex:Peter ex:hasBiologicalMother ex:Sarah .
```

it can be inferred that

```
ex:Jane owl:sameAs ex:Sarah .
```

The semantics of `owl:InverseFunctionalProperty` are analogous. A good example is the `foaf:mbox` property for email addresses; if two persons are asserted to have the same email address, then they must be the same person.

Another use case for the application of equivalence statements can be found in thesauri, i.e., actively administrated vocabularies that define semantic relations between terms. The Simple Knowledge Organization System (SKOS) [56] is an RDF vocabulary, and W3C Recommendation, that can be used to define thesauri and similar kinds of knowledge bases. SKOS offers the property `skos:exactMatch` to assert that two terms have the same meaning. There is a subtle difference between `owl:sameAs` and `skos:exactMatch`. While the former can be used to assert that two URI resources represent the *same thing*, the latter is used to claim that two URI resources represent two *different things* (natural language terms) that have the *same meaning* in natural language.¹

Sometimes, especially in more sophisticated ontologies built to enable automatic reasoning, resources have to be explicitly defined to be *not* equivalent, i.e., *different*. OWL offers the property `owl:differentFrom` to this end, and `owl:AllDifferent` to define lists of resources, all of which are different from one another. [3]

2.3.6. Advanced Meta-Modeling with OWL

The Web Ontology Language (OWL) offers many mechanisms for precise ontology definition, many of which have not been described so far.

One of the most important features is a mechanism to describe classes by restricting (`owl:Restriction`) the *values* allowed for certain properties (`owl:onProperty`). That is, the type of resources can be inferred based on their actual property values. For example, the following statements would define a class `music:MyWayInterprets` including all resources having the value `ex:MyWay` for the property `music:performs`.

```
music:MyWayInterprets owl:equivalentClass _:blankNode12 .
_:blankNode1 rdf:type owl:Restriction .
_:blankNode1 owl:onProperty music:performs .
_:blankNode1 owl:hasValue ex:MyWay .
```

Similar properties exist to define classes based on the *number of values* for some property (`owl:cardinality`, `owl:minCardinality`, and `owl:maxCardinality`). The *assumed open world* can be closed by using `owl:oneOf` to explicitly specify *all* members of a class. Further OWL properties allow for the definition of united, intersected, disjoint and complemented classes.

¹The official SKOS reference [56] states: “The property `skos:exactMatch` is used to link two concepts, indicating a high degree of confidence that the concepts can be used interchangeably across a wide range of information retrieval applications.”

²Nodes of the form `_:name` are RDF *blank nodes*, i.e., nodes in the RDF graph that have no URI and cannot be referenced from the outside of the RDF document.

2.4. Understanding RDF Data Modeling

To gain a general understanding of data modeling in RDF, we first differentiate between different levels of modeling (section 2.4.1), and then discuss the Open World nature of RDF (section 2.4.2).

2.4.1. Levels of Modeling

Now that we have seen what RDF is, and how ontologies can be defined in RDF, let us take a look at the different constituents of data modeling in RDF and how they relate to each other. These constituents are:

- The *abstract syntax* of RDF. All information is expressed in triples, as explained in section 2.2.
- The *standard vocabularies* RDF, RDFS, and OWL, just introduced above.
- Domain-specific *vocabularies* (or ontologies) that, in principle, can be defined by everybody.
- Real world *facts modeled in RDF triples*.

When a real world phenomenon is described in RDF triples, formally speaking, a **model** of the real world is created, that represents the phenomenon while it also omits many real world relations and characteristics, that is, every model is an **abstraction** of the real world.

For example, when the real world phenomenon of a specific male human, that lived in an area referred to as the “USA” and was known for specific art products referred to as “songs”, is modeled in RDF as

```
ex:Sinatra rdf:type music:Singer .
ex:MyWay rdf:type music:Song .
ex:Sinatra music:sings ex:MyWay .
```

then this model abstracts from a lot of things (e.g., biology, geography, cultural contexts of art). From the model, we only know that some real world *thing* (a `music:Singer` represented by the URI `ex:Sinatra`) has a specific relationship (represented by `music:sings`) to something (a `music:Song` represented by `ex:MyWay`).

The **model** of Frank Sinatra and his song "My Way" is expressed with terms (`sings`, `Singer`, `Song`) from the fictitious music vocabulary. This vocabulary can be understood

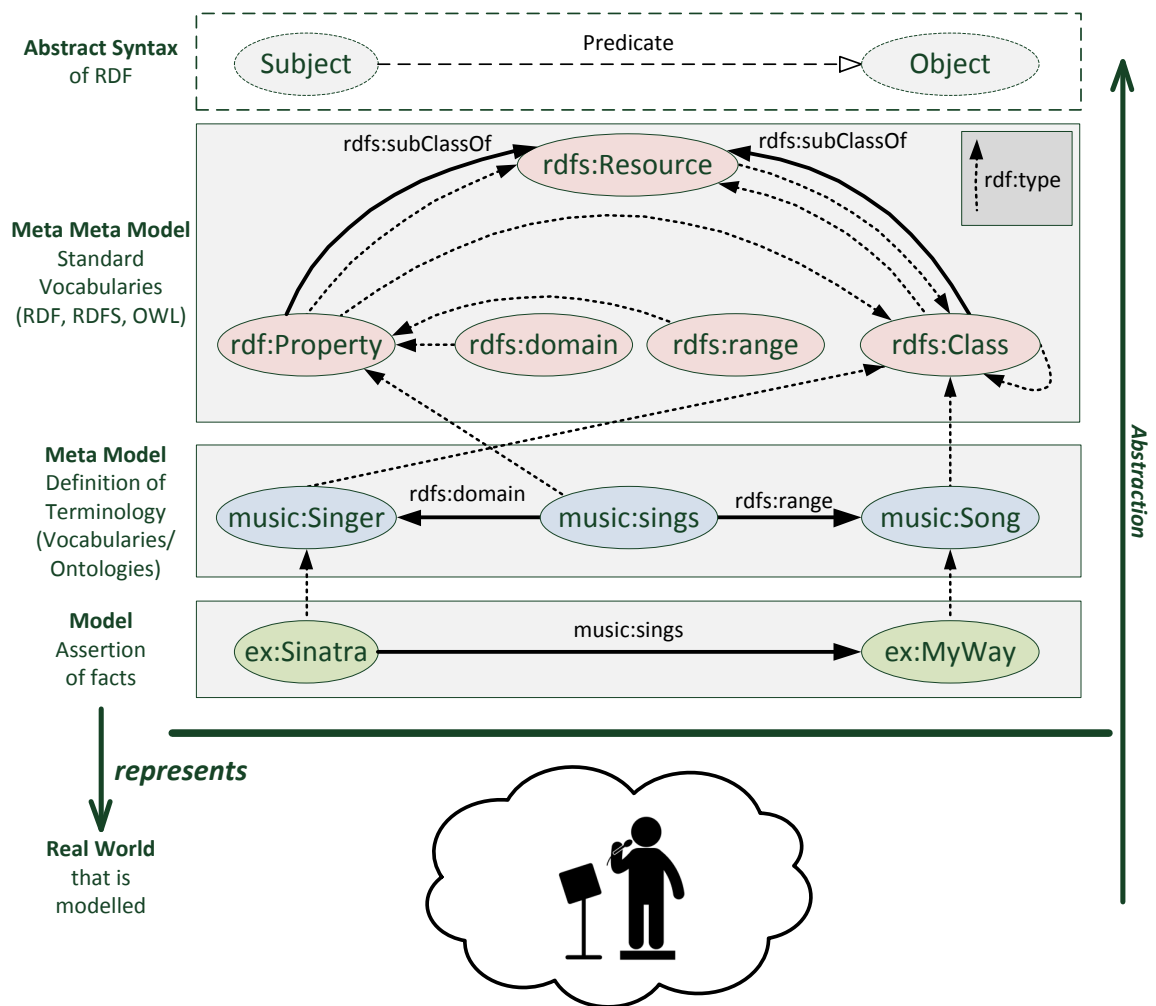


Figure 2.5.: *Different Modeling Levels in RDF* – The image shows how RDF is used on different modeling levels. In the example depicted, the fact that Frank Sinatra sings the song “My Way” is expressed in RDF. The fact itself is found on the **model** level, while it is expressed using terms (e.g., **music:sings**) defined on the **meta** level. The vocabularies (or ontologies) on this level are itself defined using the standard vocabularies RDF, RDFS, and OWL from the **meta meta** level. Underlying all these level is the **abstract syntax** of RDF, i.e., all information is expressed in triple form. The unlabeled, dashed arrows represent the **rdf:type** predicate, used to model "instance of" relations in RDF.

Note: The diagram is not complete regarding **rdf:type** and **rdfs:subClassOf** relations, because everything in RDF is of type **rdfs:Resource**, and all classes are subclasses of **rdfs:Resource**.

as a model that abstracts from individuals, and only relates the general concepts of singers and songs. Thus, the music vocabulary is a **meta model** for RDF models about singers and songs. Analogously, the standard vocabularies (RDF, RDFS, and OWL) constitute a model that abstracts from individual concepts; hence, they form a **meta meta model** for meta models about specific real world concepts.

This relation of the different levels of modeling in RDF is visualized in figure 2.5. The property `rdf:type` is a good indicator for a triple that relates two concepts on different model levels. It links specific individuals like `ex:Sinatra` on the **model level** to concepts like `music:Singer` on the **meta model level**, and specific concepts like these to abstract concepts like `rdfs:Class` on the **meta meta model level**. The standard vocabularies on the **meta meta model level** are defined in their own terms, e.g., `rdfs:Resource` is an instance of `rdfs:Class`, while the latter is a subclass of the former. These self-references terminate the model hierarchy. The **abstract syntax** is understood as standing outside of this hierarchy, although it is in fact a further abstraction.¹

2.4.2. Open World Modeling

The idea of a Semantic Web and its technical implications are usually hard to understand on first encounter. Some of the common misconceptions are obviously wrong, and can be clarified easily, while others are more *subtle*. Therefore, it is important to talk about the general characteristics of the *Web of Data* that is built using the technologies presented in this background chapter, RDF, URIs, SPARQL and so on.

To describe the characteristics of the Web of Data, we take two approaches. Firstly, we present in detail the similarities between the way humans speak about topics in the real world and the way data is published in the Web of Data. Secondly, we work out the differences between data modeling for the Web of Data and the way data models are usually created in software development, i.e., following the OOP paradigm.

Natural Language. When people talk about things, they compose sentences according to the syntax (or more general, grammar) of the natural language they are using. In general, people can say whatever they want, but to convey meaning in these sentences, they use words that, they expect, already have meaning for others, i.e., their words are chosen from commonly known vocabularies. These words acquire their

¹Formally, the abstract syntax of RDF is an abstraction of vocabularies; it defines a triple format consisting of URIs (plus literals and blank nodes). In our opinion there is no benefit in calling the abstract syntax a meta meta meta model, as it does not make sense to call the syntax of a programming language a meta model for programs, although it *is* correct, since every language's syntax is a meta model for its sentences.

meaning not through definition by an authority, but rather through an evolutionary process in society; the meaning of a word is shaped through the way it is used by people.

Of course contradictions and misunderstandings do occur. Just because something is said, it does not necessarily have to be true. Also, people have to expect new information at any time; just because something was not said, it does not necessarily have to be wrong. Sometimes people do not understand each other, because they use different words for the same meaning or the same word for different meanings.

The situation with RDF is similar. When people model information in RDF, they compose triples according to the syntax of RDF. Again in general, people can write any triple they want, but to convey meaning in these triples, they better use URIs for subject, predicate, and object, which already have some meaning for others, i.e., these URIs are chosen from commonly known vocabularies (or ontologies). Although these vocabularies are usually defined and published by an authority, there is no way in which that authority could enforce the envisioned meaning. Still, anybody can use the terms from a vocabulary in any way, and that is why the terms acquire their meaning through an evolutionary process in the Web of Data; the meaning of a term is shaped through the way it is used by data publishers.

Of course contradictions and misunderstandings do occur, as well. Triples do not have to be true, and the Web of Data is an *Open World*, i.e., no conclusions can be drawn from the nonexistence of triples. Different datasets about the same domain might not be able to cooperate, because they use different terms for the same meaning: the names for things are not unique. But, different from natural language, all names in the Web of Data (URIs) uniquely identify a thing.

OOP Data Modeling. When OOP developers design a data model for their software, they usually think about “classes” and “objects”, and work with meta models depicted in “class diagrams”. They typically use libraries provided by others together with their own classes. When the software is run, its behavior depends on the objects that are dynamically instantiated from the classes. That is, the definition of the meta model (classes) directly defines behavior and structure of the model (objects). The same is true for relational databases; the meta model (relational schema) defines how data is modeled (filled tables). Both examples entail a *top down* way of thinking about data modeling.

With data modeling in RDF the situation *seems* similar at first. When data engineers want to model some information in RDF, they create meta models called “vocabularies” (or “ontologies”) and think about “classes” and “properties”; terms that sound familiar. The terms from a vocabulary are then used in RDF triples to assert “facts about the world” on the model level. Different to OOP classes and relational schema, RDF

vocabularies do not define or constrain¹ the use of its terms, i.e., the meta model does not² define the model. Triples can be formed and combined like their authors wish, and meta models (ontologies) can only adopted by convention (bottom-up), not enforced by the system (top-down).

Summary. In their book “Semantic Web for the Working Ontologist” [3] Allemang and Hendler have identified the following characteristics of RDF and the Web of Data.

- **AAA Slogan:** Anyone can say **A**nthing about **A**ny topic.
- **Open World Assumption:** Because of the **AAA** slogan, there could always be new information that we are unaware of. This means that we cannot draw conclusions from the nonexistence of triples.
- **Nonunique Naming Assumption:** The same thing can be represented by different names (URIs).

Taking a data-modeling perspective, we can summarize these characteristics as *Bottom-Up Data Modeling*.

- **Bottom-Up Data Modeling:** Like the WWW, the Web of Data is created from the bottom up, by potentially all people and organizations. This enables the *network effect* that is needed for the realization of the Semantic Web vision. This characteristic can be counterintuitive for traditional *top-down* data modelers.

¹A good example for the non-constraining semantics of RDF are the properties `rdfs:domain` and `rdfs:range` as described in section 2.3.3.

²It can be argued that some advanced features of ontology modeling like `owl:oneOf` (see section 2.3.6) constrain or define the actual data model. Still this circumstance does not change the fact that the overall design of RDF follows a *bottom-up* style of modeling.

2.5. SPARQL – the query language

Data storage and data access are, in general, two sides of the same coin. Where databases are used, query languages typically play the access role. Relational databases and SQL are a well-known example. For RDF data, whether stored in local triple stores (section 2.6) or available remotely via SPARQL-Endpoints, the query language is SPARQL, short for SPARQL Protocol and RDF Query Language [61].

Users familiar with SQL will find similarities in SPARQL, like, for example, the “`SELECT...FROM...WHERE...`” structure of queries. But there are also important differences. Relational databases work with a set of relational tables, connected by primary and foreign keys; hence SQL is used to specify joins of tables and subsets of rows from these. As we have seen, RDF data forms a graph structure; hence SPARQL is used to specify graph patterns to select all data that matches these patterns.

The general pattern of a SPARQL `SELECT` query is as follows¹:

```
<PREFIX declarations...>

SELECT <variables...>
WHERE {
  <graph patterns with variables...>
}
ORDER BY <conditions...>
OFFSET <offset>
LIMIT <limit>
```

The following example (from [61], modified) describes three subjects, two of which have a `foaf:name` and `foaf:mbox`, while one (`ex:c`) only has a `foaf:mbox`. The RDF graph is shown in figure 2.6.

```
@prefix foaf: <http://xmlns.com/foaf/0.1/> .
@prefix ex: <http://example.com/> .

ex:a foaf:name "Johnny Lee Outlaw" .
ex:a foaf:mbox <mailto:jlow@example.com> .
ex:b foaf:name "Peter Goodguy" .
ex:b foaf:mbox <mailto:peter@example.org> .
ex:c foaf:mbox <mailto:carol@example.org> .
```

¹This general pattern of a SPARQL `SELECT` query is not complete. Many possible features and their according keywords have been omitted. These include: `UNION`, `FROM`, `SERVICE`, `FILTER`, `OPTIONAL`, `UNSAID` and more. For more details see the latest SPARQL 1.1 standard [46].

To obtain a list of names and corresponding email-addresses, a graph pattern is specified that connects `foaf:name` and `foaf:mbox` triples by their common subject resource and binds their values (or objects) to two variables `?name` and `?email`.

```
PREFIX foaf: <http://xmlns.com/foaf/0.1/>
SELECT ?name ?email
WHERE {
  ?subject foaf:name ?name .
  ?subject foaf:mbox ?email .
}
```

This will produce a list of binding results. Because the whole graph pattern has to be matched, the `foaf:mbox` triple with subject `ex:c` will not be matched, and, consequently, not be included in the results (they are shown grayed out in figure 2.6). The results for the example would look like:

name	email
"Johnny Lee Outlaw"	<mailto:jlow@example.com>
"Peter Goodguy"	<mailto:peter@example.org>

As can be seen from the general structure as well as from the example query, a prefix mechanism for URI namespaces (similar to the one in Turtle) is provided with the `PREFIX` keyword. The `SELECT` keyword indicates a select-query, the most common of four query types, and is followed by a list of variables or aggregates (e.g, `SUM(?amount)` `AS ?totalAmount`) of those. The `WHERE` keyword is followed by the graph pattern in braces. The results can be sorted (`ORDER BY` keyword) by any combination of variables, and an `OFFSET` can be specified into this list, i.e., the number of results that are skipped from the beginning. The total number of binding results (i.e., “rows”) can be limited via the `LIMIT` keyword.

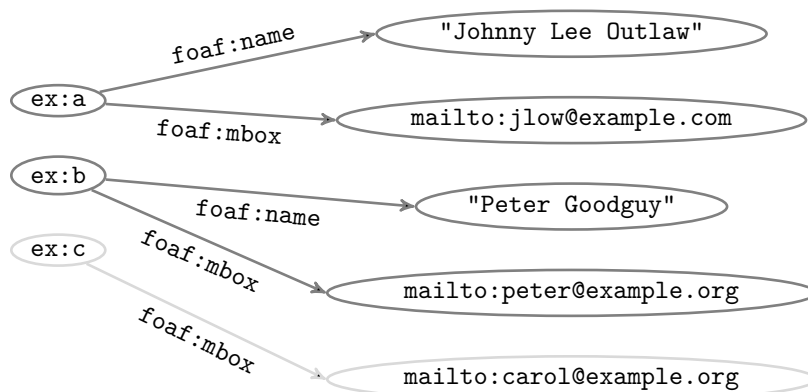


Figure 2.6.: RDF Graph for SPARQL Example – Gray lines indicate the part of the graph that is not matched by the graph pattern.

As noted before, there are four query types.

1. The **SELECT-query**, as seen above, is used to obtain a list of bindings for the variable in a graph pattern.
2. The **CONSTRUCT-query** returns an RDF graph. It can be used to create new triples based on a graph template and variable bindings from a matched graph pattern. For example, assuming that, if someone knows a person **Y**, then they also know all persons known by the **Y**, we can create according triples like this:

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

CONSTRUCT { ?x foaf:knows ?z }
WHERE {
    ?x foaf:knows ?y.
    ?y foaf:knows ?z.
}
```

3. The **DESCRIBE-query** returns an RDF graph that “describes” resources. The simplest example would be a description of a single resource:

```
DESCRIBE <http://dbpedia.org/resource/Berlin>
```

Scope and content of the response may vary. “The RDF returned is determined by the information publisher. It may be information the service deems relevant to the resources being described.” [46]

4. The **ASK-query** returns a single boolean result. It can be used to test whether a graph pattern has any match or none.

2.6. Triple Stores and SPARQL Endpoints

In general, data can be stored electronically in two ways: in simple files on hard disks or through specialized data storage systems (i.e., *databases*), which expose an interface to store and retrieve data. The situation with RDF is not different; RDF data can be stored

- in *files* using one of the defined serialization formats (see section 2.2.3), or
- in special databases, called *triple stores*.

Popular triple stores include Apache Jena¹, OpenLink Virtuoso², Sesame³, Allegro-Graph⁴, and Redland⁵. Basic criteria to compare triple stores include the performance in loading triples and processing queries, and the extent to which the triple stores support automatic inference and the full feature set of SPARQL 1.1 [46]. Apache Jena was used in the software developed for this thesis.

While many of these stores can be accessed via proprietary Application Programming Interfaces (APIs), the SPARQL Protocol and RDF Query Language provides a standardized way of querying triple stores. That is, triple stores are exposed over HTTP as a “SPARQL Protocol service”. They receive and answer SPARQL queries (see section 2.5 for query formulation) as HTTP requests and HTTP responses, respectively. The “URI at which a SPARQL Protocol service listens for requests from SPARQL Protocol clients” is called a *SPARQL endpoint*. [40]

¹<http://jena.apache.org/>

²<http://virtuoso.openlinksw.com/>

³<http://www.openrdf.org/>

⁴<http://www.franz.com/agraph/allegrograph/>

⁵<http://librdf.org/>

3. Structural Analysis of RDF Datasets

In this chapter we introduce our notion of a *structural analysis of RDF datasets* as a process that answers four central questions of dataset understanding, based on results from 18 statistical criteria.

First, we motivate the need for RDFSynopsis, a software tool that assists in understanding unknown RDF datasets, by describing a typical use case. We further establish four central questions representing four central aspects that can be understood about a dataset (section 3.1). We then explain, in more detail, the requirements that follow from the use case (section 3.2). In section 3.3, we present related work from three areas of RDF data analysis, namely statistical measures, descriptive subgraphs and summary graphs. Finally, in section 3.4, we explain what measures (18 in total) are chosen to answer each central question.

3.1. Use Case and Central Questions

Let us imagine the daily work of “data engineers”. As part of their job, they might be responsible for the maintenance, publication, use, and further development of data pools and data-intensive software. Let us further imagine these data engineers had to deal with a multitude of RDF datasets on a daily basis, and those were only available remotely, via SPARQL endpoints.

In order to decide, whether a concrete dataset could be useful and how it could be used, a data engineer would have to know the answers to several questions. The first two questions concern the domain of the information contained in a dataset. For a particular application, only a small subset of all possible domains will be of interest to the data engineer. We can distinguish between the kind of entities represented in a dataset (**Entity Domain**), and the kind of information a dataset contains about those entities (**Information Domain**). In order to access the information contained in a given dataset, a data engineer has to understand how the information is cast into triple structures (**Composition**). The last important question concerns the dataset’s suitability for reuse, potentially in combination with other datasets (**Reusability**).

We summarize these aspects as the following *central questions*:

- **Entity Domain:** What things is the dataset about?
- **Information Domain:** What information does the dataset contain about those things?
- **Composition:** How is the information expressed in RDF?
- **Reusability:** Can the dataset be used easily?

3.2. Requirements and Definitions

In this section we describe in more detail the basic requirements that follow from the use case, and define more strictly what we mean by a “structural analysis of unknown RDF datasets via SPARQL endpoints”.

3.2.1. RDF Datasets

We define an RDF dataset as the set of all RDF triples that can be retrieved from a single point of access, namely an RDF file in any serialization format or a SPARQL endpoint. The rationale behind this definition is that all triples available via a single point of access must appear to the consumer, being unaware of any underlying structures, as belonging to *one* meaningful collection of triples.

3.2.2. Publisher and Consumer

Different actors participate in handling RDF data. They can be broadly divided into two groups, namely *publishers* and *consumers*. [2]

On the side of *publishers*, we find all parties that produce, collect, refine and prepare data in order to make it available as RDF data. On the other side, the *consumer* side, are those actors that read or reuse the data for their own applications. In principle, these actors may be humans or - since it is the Semantic Web we are talking about - machines.

In our work, we focus on the **consumer** side, i.e., **we assume the absence of prior knowledge** regarding a dataset’s structure, the process that created RDF data from other forms of original data, or the original data itself.

3.2.3. Structure, not values

For the work presented in this thesis, we assume that a data engineer wants to know how a dataset is built and what kind of information is contained in it (**structure** in a broader sense), but has no interest to understand what real world facts are asserted in detail (**value**). Revisiting our example, we want to know that a dataset relates entities that are singers or songs. We do not want to know what the average length of Frank Sinatra’s song titles is.

In the world of top-down data modeling, the structure of a dataset is **explicitly**, and bindingly, defined in some kind of schema information (i.e., the meta model level).

In contrast, in the bottom-up world of RDF (see section 2.4.2), while some **explicit** structural definitions exist on the meta model level (e.g., class hierarchies), the **structure** of a dataset is **implicitly** contained in the asserted triples on the model level (section 2.4.1). Hence, to understand the structure of RDF datasets, we have to analyze the model level and meta model level.

Summary. For this thesis, we focus on obtaining structure (not value) information about a dataset. Due to the bottom-up character of RDF, this information is not (necessarily) explicitly, but rather implicitly contained in a dataset.

3.2.4. Access via SPARQL-Endpoints

In our scenario, the data engineer regularly has to determine the relevance of datasets that are available from remote locations, i.e., they are *on the Web*. We assume that, in many cases, a full download of the dataset in question is infeasible, either due to bandwidth or size limitations, or due to sheer unavailability of data dumps.

We therefore restrict our approaches to access datasets only through SPARQL endpoints. That means, all information we want to receive about a dataset have to be obtained as results of SPARQL queries. An advantage of this approach is that it can be used with locally available files, as well; they can be loaded into a local triple store and queried from there.

3.2.5. Summary

In this section, we have defined what we mean by a “structural analysis of unknown RDF datasets via SPARQL endpoints”.

“Structural” refers to the four aspects of understanding a dataset (Entity Domain, Information Domain, Composition, and Reusability), that are not concerned with value distributions on the fact level. “Analysis” refers to the fact that, in RDF, we cannot simply extract structure information from some kind of schema, but have to analyze a dataset, in order to find the implicitly contained structure. “Unknown” means that we assume a consumer with no prior knowledge about the dataset. By “RDF dataset” we mean a collection of all triples available from one point of access, either a file or a SPARQL endpoint. “Via SPARQL endpoints” reflects our requirement to find an analysis approach for remote datasets that can only be accessed using SPARQL queries.

3.3. Approaches and Related Work

In this section we explain how our work, presented in this thesis, relates to other scientific and practical works in the field.

We take a look at different approaches towards analyzing and understanding RDF data. First and foremost, we examine statistical approaches and conduct a survey on *statistical measures* (section 3.3.1). Then, we study approaches to extract *descriptive subgraphs* (section 3.3.2), that is, for a given subject, the subgraph that best describes it. Finally, we look into approaches to generate *summary graphs* (section 3.3.3) of RDF graphs.

3.3.1. Statistical Measures

In this section, we look into approaches to capture RDF datasets by means of collecting statistical data about them. We briefly describe each piece of related work, and compare it with our path chosen for RDFSynopsis, mainly focusing on its purpose, technology and data processing approach. In addition, we have extracted all statistical measures, that are referred to in these scientific works or used in practical applications, and compiled an overview (see table 3.1). The table contains one criterion per row. The central columns represent the different pieces of related work discussed in this section. “✓” indicates that the criterion was explicitly described (maybe by a different name) in the article or software documentation. “CR” indicates that we found the criterion during a code review of the respective software tool. For better comparison, this table also includes (in **bold**) the statistical criteria we have chosen for RDFSynopsis, as explained below in section 3.4.

■ *Scientific Work.* Several articles have been published on obtaining statistics about RDF data.

□ Langerger and Wöß describe *RDFStats* [52, 53]; a software tool for statistics generation and persistence. Like RDFSynopsis, RDFStats is based on Apache Jena and uses SPARQL queries to obtain statistical data. Unlike RDFSynopsis, RDFStats collects value histograms for each class and property; a generic approach, targeted not at human understanding but at supporting applications that process RDF data.

□ The Vocabulary Of Interlinked Datasets (void) [1, 2] represents a vocabulary to describe RDF datasets (*metadata*), including their access methods, licenses, “root resources”, subjects (topics), and general metadata like titles or publisher information. Several publications and applications use void to output their results in RDF. In void, statistics about a dataset are expressed by defining and describing “class- and property-based partitions”. For example the number of `foaf:Person` instances is

expressed by specifying the number of entities in the class-partition for `foaf:Person`, like so

```
ex:MyExampleDataset a void:Dataset .
ex:MyExampleDataset void:classPartition _:bn1 .
_:bn1 void:class foaf:Person .
_:bn1 void:entities 42 .
```

Similarly, the number of links between URI namespaces can be expressed by defining so called “linksets”. There is also a special `void:vocabulary` predicate to list the vocabularies used in a dataset.

□ For their software *voiDgen* [21], Böhm, Lorey, and Naumann use a distributed algorithm (Map-Reduce) to efficiently analyze large RDF graphs (directly in memory) and output statistical data in *voiD*. They propose to compute (and extend the *voiD* vocabulary to include) additional kinds of class-, property-, and link-based partitions, e.g., sets of resources connected via specific predicates. In contrast to *RDFSynopsis*, *voidGen* does not use SPARQL or analyzes remote datasets, but shows that distributed (“in the cloud”) analyses of large RDF graphs are feasible.

□ Auer et al. have developed *LODStats* [7, 8], a Python-based framework, to compute statistics for the RDF datasets on “The Data Hub” ([57], an instance of the Comprehensive Knowledge Archive Network (CKAN) [35] system). The results are output in RDF using *voiD* and *qb*¹. The authors employ a so-called “statement-stream” approach, i.e., an RDF dataset is not analyzed as a graph but as a sequence of triples. It is similar to the triple-stream approach (see section 4.1.2) of *RDFSynopsis*. Although they mention the approach, used by *RDFSynopsis*, to emulate the statement-stream with SPARQL, their focus lies with RDF graphs that are already available in a serialized form. In their paper, they also provide declarative definitions for 32 statistical criteria (each with statement filter rule, needed data structure and post-processing operations), including 8 of our 18 criteria. They point out that their approach performs significantly better than SPARQL-based tools like *RDFStats* and *make-void* (see below), while using a constant amount of Random-Access Memory (RAM).

□ In their article on “compact representations of large RDF data sets” [41] Fernández et al. introduce several statistical metrics. Among those are variations of “out-degree” metrics (e.g., the maximum number of triples per subject), and a metric.

□ *Swoogle* [36, 37, 45] is a Semantic Web search engine. To improve efficiency and effectiveness of the search, *Swoogle* collects, among other metadata, statistics about defined classes, properties and class instances. In their paper, the authors of *Swoogle* also introduce the *ontology-ratio* measure that is used by *RDFSynopsis*.

¹the Data Cube Vocabulary [32]

□ *Sindice* [58, 68, 70] is another Semantic Web Index. Their statistics [69] include instance and namespace counts for classes and properties, but are aggregated for all indexed datasets.

■ *Practical Work*. Several software tools collect and use statistics about RDF datasets.

□ *NX parser* [50] is a Java-based parser for RDF data in the N-Triples [43] serialization format (and n-tuple formats in general). A code review (marked “CR” in table 3.1) reveals, that it features basic class instance and property usage counts.

□ *make-void* [33] by Richard Cyganiak is another tool that computes statistics for RDF datasets and generates RDF output using the void vocabulary. It is based on Apache Jena and features more advanced criteria, such as the number of links between different URI namespaces or the number of distinct subjects per property (found in code review).

□ The *Apache Jena TDB Optimizer* [67] uses dataset statistics (collected with the *tdbstats* tool) to optimize query answering.

Table 3.1.: Survey of Statistical Measures

No.	Statistical Criterion	LODStats [8]	RDFStats [53]	make-void [33]	NX Parser [50]	voidGen [21]	tdbstats [67]	void vocab. [1, 2]	compact rep. [41]	Swoogle [45]	Description
	No. of ...										
1.	triples	✓		✓		✓		✓		✓	No. of (<i>?s ?p ?o</i>) triples
2.	<i>entities mentioned</i>	✓									No. of used URIs non-distinct
3.	<i>literals mentioned</i>	✓									No. of literals <i>?lit</i> for (<i>?s ?p ?lit</i>)
4.	<i>distinct resources</i>					✓					No. of distinct URI-subjects and -objects
5.	<i>distinct subjects</i>		CR		CR	✓		✓			No. of distinct <i>?s</i> for (<i>?s ?p ?o</i>)
6.	<i>distinct subjects (URI)</i>	CR	CR	✓	CR			✓			No. of distinct URI- <i>?s</i> for (<i>?s ?p ?o</i>)
7.	distinct blank subjects		CR		CR						No. of distinct blank- <i>?s</i> for (<i>?s ?p ?o</i>)
8.	<i>distinct objects (URI, blank, literal)</i>					✓		✓			No. of distinct <i>?o</i> for (<i>?s ?p ?o</i>)
9.	<i>blanks mentioned</i>	CR									No. of triples (<i>?bn ?p ?o</i>) or (<i>?s ?p ?bn</i>) with blank <i>?bn</i>
10.	<i>blanks mentioned as subj.</i>	✓									No. of triples (<i>?bn ?p ?o</i>) with blank <i>?bn</i>
11.	<i>blanks mentioned as obj.</i>	✓									No. of triples (<i>?s ?p ?bn</i>) with blank <i>?bn</i>
12.	isolated blanks										No. of distinct blank <i>?bn</i> for (<i>?bn ?p ?s</i>) only occurring in subject position
13.	<i>distinct classes (instantiated, URI)</i>			✓	CR			✓		✓	No. of distinct <i>?class</i> -URIs for (<i>?s a ?class</i>)
14.	<i>distinct properties (instantiated)</i>			✓	CR	✓		✓		✓	No. of distinct <i>?p</i> for (<i>?s ?p ?o</i>)
15.	<i>distinct classes (defined, URI)</i>									✓	No. of <i>?class</i> -URIs for triples (<i>?class a rdfs:Class</i>) and (<i>?class a owl:Class</i>)
16.	<i>distinct properties (defined)</i>									✓	No. of <i>?prop</i> -URIs for triples (<i>?prop a rdf:Property</i>)
17.	<i>distinct terms</i>									✓	No. of distinct defined and instantiated classes and properties
18.	<i>distinct non-term subjects</i>									✓	No. of distinct Subjects excl. classes and properties
19.	<i>typed subj.</i>	✓						✓			No. of triples (<i>?s a ?o</i>)
20.	<i>labeled subj.</i>	✓						✓			No. of triples (<i>?s rdfs:label ?o</i>)
21.	sameAs	✓						✓			No. of triples (<i>?s owl:sameAs ?o</i>)
22.	<i>subclass usage</i>	✓						✓			No. of triples (<i>?class1 rdfs:subClassOf ?class2</i>)
23.	<i>distinct contexts</i>				CR	✓					No. of distinct named graphs

Continued on next page. . .

Table 3.1.: Survey of Statistical Measures (... continued)

No.	Statistical Criterion	LODStats [8]	RDFStats [53]	make-void [33]	NX Parser [50]	voidGen [21]	tdbstats [67]	void vocab. [1, 2]	compact rep. [41]	Swoogle [45]	Description
Histogram (For each ... the No. of ...)											
24.	<i>class usage count</i>	✓	✓	✓	CR	✓		✓			For each <i>?class</i> the No. of triples (<i>?s a ?class</i>)
25.	<i>property usage</i>	✓		✓	CR		✓	✓			For each <i>?p</i> the No. of triples (<i>?s ?p ?o</i>)
26.	<i>class instances per property</i>										For each pair (<i>?p,?class</i>) the No. of distinct <i>?s</i> for (<i>?s ?p ?o. ?s a ?class</i>)
27.	<i>property usage per subj.</i>								✓		For each pair (<i>?s,?p</i>) the No. of triples (<i>?s ?p ?o</i>)
28.	<i>property usage per obj.</i>								✓		For each pair (<i>?p,?o</i>) the No. of triples (<i>?s ?p ?o</i>)
29.	<i>property usage distinct per subj.</i>	✓						✓			For each <i>?s</i> the No. of distinct <i>?p</i> -URLs for triples (<i>?s ?p ?o</i>)
30.	<i>property usage distinct per obj.</i>	✓							✓		For each <i>?o</i> the No. of distinct <i>?p</i> -URLs for triples (<i>?s ?p ?o</i>)
31.	<i>subject usage distinct per prop.</i>			CR				✓			For each <i>?p</i> the No. of distinct <i>?s</i> -URLs for triples (<i>?s ?p ?o</i>)
32.	<i>object usage distinct per prop.</i>			CR				✓			For each <i>?p</i> the No. of distinct <i>?o</i> -URLs for triples (<i>?s ?p ?o</i>)
33.	<i>property usage per subject class</i>										For each pair (<i>?p,?class</i>) the No. of bindings for (<i>?s ?p ?o. ?s a ?class</i>)
34.	<i>triples per subject class</i>										For each <i>?class</i> the No. of bindings (<i>?s ?p ?o. ?s a ?class</i>)
35.	<i>outdegree per subject</i>	✓				✓			✓		For each <i>?s</i> the No. of (<i>?s ?p ?o</i>)
36.	<i>indegree per object</i>	✓							✓		For each <i>?o</i> the No. of (<i>?s ?p ?o</i>)
37.	<i>literal usage</i>	CR									For each literal- <i>?val</i> the No. of (<i>?s ?p ?val</i>)
38.	<i>literal datatype usage</i>	✓									For each <i>xsd:</i> datatype the No. of literals
39.	<i>literal language usage</i>	✓									For each language the No. of literals
40.	<i>links per (ns1,ns2)</i>	✓						✓			For each pair (<i>NS1,NS2</i>) the No. of (<i>?s ?p ?o</i>), with (<i>?s</i> in <i>NS1</i> , <i>?o</i> in <i>NS2</i>)
41.	<i>namespace links</i>	✓		✓				✓			For each triple (<i>NS1,?p,NS2</i>) the No. of (<i>?s ?p ?o</i>), with (<i>?s</i> in <i>NS1</i> , <i>?o</i> in <i>NS2</i>)
42.	<i>subj. vocabularies</i>	✓									For each <i>s</i> -namespace the No. of (<i>?s ?p ?o</i>)
43.	<i>predicate vocabularies</i>	✓									For each <i>p</i> -namespace the No. of (<i>?s ?p ?o</i>)
44.	<i>obj. vocabularies</i>	✓									For each <i>o</i> -namespace the No. of (<i>?s ?p ?o</i>)
45.	<i>pred. value histogram per (subj.-class, pred., datatype)</i>		✓								For each triple (<i>?class,?p,?val-datatype</i>) the value distribution for triples (<i>?s ?p ?val. ?s a ?class</i>)

Continued on next page...

Table 3.1.: Survey of Statistical Measures (... continued)

No.	Statistical Criterion	LODStats [8]	RDFStats [53]	make-void [33]	NX Parser [50]	voidGen [21]	tdbstats [67]	void vocab. [1, 2]	compact rep. [41]	Swoogle [45]	Description
Sets of URIs											
46.	<i>used classes</i>	✓	✓								<i>Set of ?class-URIs for triples (?s a ?class)</i>
47.	<i>used properties</i>		CR								<i>Set of ?p-URIs for triples (?s ?p ?o)</i>
48.	<i>class instances</i>		✓								<i>For each ?class the set of ?ins-URIs for triples (?ins a ?class)</i>
49.	<i>classes defined</i>	✓								✓	<i>Set of ?class-URIs for triples (?class a rdfs:Class) or (?class a owl:Class)</i>
50.	<i>properties defined</i>									✓	<i>Set of ?prop-URIs for triples (?prop a rdf:Property)</i>
51.	<i>properties per subj.</i>	✓									<i>For each ?s the set of ?p-URIs for triples (?s ?p ?o)</i>
52.	<i>properties per obj.</i>	✓									<i>For each ?o the set of ?p-URIs for triples (?s ?p ?o)</i>
53.	<i>properties per subj. class</i>		✓								<i>For each subject-?class the set of ?p-URIs for triples (?s ?p ?o. ?s a ?class)</i>
54.	<i>common properties</i>										<i>For each subject-?class the set of ?p-URIs for which all ?s occur in triples (?s ?p ?o. ?s a ?class)</i>
55.	<i>properties per obj. class</i>										<i>For each object-?class the set of ?p-URIs for triples (?s ?p ?o. ?o a ?class)</i>
56.	<i>distinct entities</i>	✓									<i>Set of distinct used URIs</i>
57.	<i>vocabularies (prop. u. classes)</i>			CR				✓			<i>Set of ?p- and ?class-namespaces for (?s ?p ?o) resp. (?s a ?class)</i>
Graphs											
58.	<i>explicit class hierarchy</i>	✓									<i>Depth of (?class1 rdfs:subClassOf ?class2)-hierarchy</i>
59.	<i>implicit class hierarchy</i>										<i>Pairs (?class1,?class2) for which all instances of ?class1 are also instances of ?class2</i>
60.	<i>explicit property hierarchy</i>	✓									<i>Depth of (?prop1 rdfs:subPropertyOf ?prop2)-hierarchy</i>
61.	<i>implicit property hierarchy</i>										<i>Pairs (?prop1,?prop2) for which all triples with ?prop1 also exist with ?prop2</i>
Other											
62.	<i>Mean typed string length</i>	✓									<i>Mean length of all xsd:string-literals</i>
63.	<i>Mean untyped string length</i>	✓									<i>Mean length of all literals w/o datatype</i>
64.	<i>Min per property {int, float, time}</i>										<i>Min. value for each numerical ?p for (?s ?p ?numVal)</i>
65.	<i>Mean per property {int, float, time}</i>	✓									<i>Mean value for each numerical ?p for (?s ?p ?numVal)</i>
66.	<i>Max per property {int, float, time}</i>	✓									<i>Max. value for each numerical ?p for (?s ?p ?numVal)</i>

Continued on next page...

Table 3.1.: Survey of Statistical Measures (... continued)

No.	Statistical Criterion	LODStats [8]	RDFStats [53]	make-void [33]	NX Parser [50]	voidDgen [21]	tdbstats [67]	void vocab. [1, 2]	compact rep. [41]	Swoogle [45]	Description
67.	<i>outdegree (min,max,mean)</i>								✓		<i>Min., max. and mean No. of (?s ?p ?o) over all ?s</i>
68.	<i>partial outdegree (min,max,mean)</i>								✓		<i>Min., max. and mean No. of (?s ?p ?o) over all ?s,?p</i>
69.	<i>distinct prop. outdegree (min,max,mean)</i>								✓		<i>Min., max. and mean No. of distinct ?p for each ?s</i>
70.	<i>indegree (min,max,mean)</i>								✓		<i>Min., max. and mean No. of (?s ?p ?o) over all ?o</i>
71.	<i>partial indegree (min,max,mean)</i>								✓		<i>Min., max. and mean No. of (?s ?p ?o) over all ?p,?o</i>
72.	<i>distinct prop. indegree (min,max,mean)</i>								✓		<i>Min., max. and mean No. of distinct ?p for each ?o</i>
73.	<i>subject-object-ratio</i>								✓		<i>Ratio between No. of nodes that are subject <u>and</u> object, and No. of nodes that are subject <u>or</u> object</i>
74.	<i>ontology-ratio</i>									✓	<i>Ratio between No. of defined classes and properties, and No. of def. classes, properties and instances</i>
75.	<i>typed-subject-ratio</i>									✓	<i>Ratio between No. of typed subjects (?s a ?o), and No. of all subjects</i>

3.3.2. Descriptive Subgraphs

Closely related to the question for the kind of information an RDF dataset contains, is the question for the subset of triples that contains information about a specific subject, i.e., which *subgraph* best *describes* that subject. In this section we look into different approaches to extract these *descriptive subgraphs* from an RDF graph.

For RDFSynopsis, we follow the approach to understand datasets by a couple of statistical criteria, most similar to the approaches presented in section 3.3.1. Still, the work on descriptive subgraphs deserves mentioning, since it provides insight into the way information is expressed in RDF structures.

Also, one idea for RDFSynopsis is to find and present common (or repeating) structures within a dataset. This idea implies that a dataset can be decomposed into multiple parts, sharing a common structure. By studying what subgraph best describes a specific subject, we also study how an RDF graph is best decomposed by subjects. The criteria concerning commonality, namely *implicit class hierarchy*, *common properties*, and *implicit property hierarchy*, were influenced by the work presented in this section.

□ In their paper on provenance tracking for RDF graphs [38], Ding et al. define “RDF molecules”, the “smallest losslessly decomposable parts” of an RDF dataset. When an RDF graph containing blank nodes is split and merged again, information may be lost. This is due to the fact that blank nodes do not have globally unique identifiers, and hence it is impossible to determine whether two blank nodes were the same in the original graph. The term “lossless decomposition” refers to the idea of splitting an RDF graph in such a way that all information can be reconstructed when merging. Ding et al. achieve this goal by pairing all blank nodes with identifying triples, e.g., triples with an `owl:FunctionalProperty`. The idea of blank nodes that “lose” their identity for proper merging is reflected in one of the statistical criteria chosen for RDFSynopsis, namely *isolated blanks*.

□ The Concise Bounded Description (CBD) [65] of a specific subject is a subgraph which contains all triples with that subject, and - recursively - all triples connected via blank nodes (as subject). The article also mentions variations that include triples with blank nodes as objects, or limit the inclusion of triples with IFPs. These are called Symmetric CBD (SCBD) and Inverse Functional CBD (IFCBD), respectively.

□ Tummarello et al. introduce the notion of Minimum Self-contained Graphs (MSGs) [71]. An MSG is constructed based on an initial triple (i.e., an edge, not a node) by recursive extension, including all triples that are connected via blank nodes in already contained triples. A graph composed of all MSGs involving a specific resource, equals the CBD of that resource.

□ The SPARQL DESCRIBE query basically returns descriptive subgraphs for the resources matched in the WHERE clause. As explained in section 2.5, the decision about scope and content of the response is left to the “information publisher”, so it may vary. “The description is determined by the query service. [...] It may be information the service deems relevant to the resources being described. [P]ossible mechanisms for deciding what information to return include CBD. For a vocabulary such as FOAF, where the resources are typically blank nodes, returning sufficient information to identify a node such as the Inverse Functional Property (IFP) `foaf:mbox_sha1sum` as well as information like name and other details recorded would be appropriate.” [46]

□ An adaptable method of obtaining descriptive subgraphs is described by Simic in [63]. The article introduces a compact notation for the specification of predicate-based tree structures, so-called “predicate trees”. Based on these, descriptive subgraphs can be extracted from an RDF graph; the described subjects representing the root nodes. According to Simic, the notation matches SPARQL (1.0) graph patterns in their expressive power, while being significantly shorter. The author describes a Scala-based library *Scardf* that, among other features, parses the predicate tree notation and outputs SPARQL CONSTRUCT queries. Scardf can also be used to create application-specific SPARQL DESCRIBE handlers for Apache Jena, based on specified predicate trees.

□ In their paper on clustering of RDF resources [44] Grimnes, Edwards, and Preece, compare three approaches to extract descriptive subgraphs (calling it “instance extraction”). Besides CBD (i), they evaluate “immediate properties” (ii), i.e., triples with the described subject, and “Depth Limited Crawling” (iii). The latter refers to the approach of including all nodes within a specified distance of the subject node. The authors conclude that their results were largely domain-dependent, and remain interested in a hybrid approach, combining CBD and Depth Limited Crawling.

□ As part of their proposal of so-called “Semantic Sitemaps” [34], Cyganiak et al. also propose to specify a dataset’s “slicing method”. That is, the method by which a SPARQL endpoint constructs descriptive subgraphs for resources, in order to answer DESCRIBE queries. The desired values for the `sc:slicing` attribute include CBD, SCBD, MSG, and “immediate properties”.

□ In the paper on *voiDgen* [21], already mentioned in 3.3.1, the authors also describe different approaches to generate class-, property-, and link-based partitions. Among those are “connected datasets” (similar to predicate trees for one specific property), and “fuzzy linksets” (treating resources equally that are *k-similar*, i.e., having *k* property values in common).

3.3.3. Summary Graphs

Summary graphs are another approach to gain an understanding of an RDF dataset. That is, based on an analysis of the dataset’s full RDF graph, a summary is constructed that takes the form of a graph.

Similar to the previous section, the work presented in this section is not directly related to RDFSynopsis, but still important. It aims for a concise presentation of a dataset’s content and structure which integrates several otherwise separated statistical criteria.

□ Khatchadourian and Consens describe their software *ExpLOD* [30, 51], which produces summary graphs for one or more specific aspects of an RDF dataset, e.g., class instantiation or predicate usage. The approach taken by ExpLOD is rather complex and can be divided into several steps.

(i) The original RDF graph is transformed into a “labeled graph”. All nodes (subjects and objects) and all edges (predicates) of the original graph are represented as nodes in this labeled graph. The edges of this new graph are unlabeled, while all nodes have a hierarchical label. For example the predicate `foaf:knows` may be represented as a node with hierarchical label “P/foaf/knows” (for Predicates/ <Vocabulary>/ <identifier>) or “P/foaf” depending on the desired granularity.

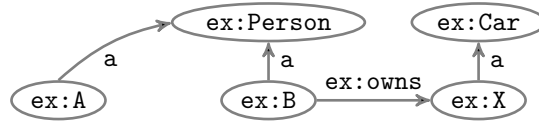
(ii) The *Bisimulation Contraction (BC)* [39] is now computed on the set of all nodes. The basic idea is to subdivide all nodes into disjoint subsets, in which all nodes are *bisimilar*, i.e., equivalent with regards to their links to nodes in other subsets. In other terms, if one node in subset A has an edge to a node in subset B, then all nodes in A have edges to nodes in B, otherwise it is no BC. Initially all nodes with the same label are in the same subset. The BC is then obtained by computing the *Coarsest Stable Partition (CSP)* [59]; details are out of scope of this thesis.

(iii) Nodes that were contained in equivalent structures in the original dataset, are now subsumed under one subset in the BC. These subsets and their interlinks are then visualized, together with the number of nodes in each subset.

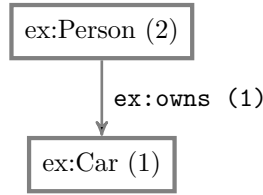
The advantage of ExpLOD’s approach is that its results show a dataset’s structure as homo- or heterogeneous as it may be. The level of detail (i.e., the granularity of the summary graph) can be controlled by changing the labels that are created for nodes. The big disadvantage is represented by the need for preprocessing the whole RDF graph to the “labeled graph”, a process that requires the materialization of the whole dataset for many of the investigated aspects.

□ Zhang, Tian, and Patel [76] describe an approach to construct summary graphs based on value analysis for a specific predicate, and to automatically find the most insightful summary. This involves the automatic categorization of numerical values and evaluation of the resulting graph’s “interestingness”. This aims at analyzing values of *one* chosen predicate (which all subjects have to provide). Hence, this work represents an approach for understanding the value distribution of a dataset whose structure is already known.

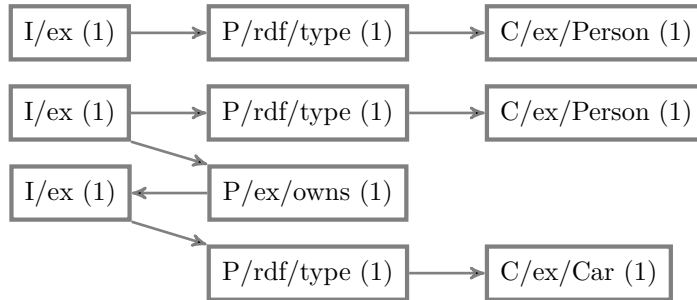
□ In order to assist users of *Sindice* (introduced above [58, 68, 70]) in query formulation, Campinas et al. have built a summary graph over all datasets on the platform [25]. Their approach is similar to ExpLOD in that they create a summary graph whose nodes each represent a subset of the original nodes (subjects and objects), based on their classes or used predicates. For example, a summary node may represent all subjects of type `foaf:Person`. Similar to ExpLOD a summary node is created for *each combination* of classes, i.e., two nodes, one of type A and one of types A *and* B, will end up in different disjoint classes. Unlike ExpLOD, the summary nodes are not further partitioned based on their interlinks. In the example (figure 3.1a), a dataset contains two entities of type `Person`. One `owns` an entity of type `Car`. Following the approach by Campinas et al. two summary nodes (`Person`, `Car`) would be created (3.1b), while ExpLOD would create several more (3.1c).



(a) Example Graph



(b) Results for Summary Graph [25]



(c) Possible Results for ExpLOD [30, 51]

Figure 3.1.: Comparison of Summary Graphs

3.4. Discussion and Choice of Measures

Let us get back to our use case of a *data engineer*, and their demand for information on RDF datasets which is reflected in the *central questions* described in section 3.1.

- **Entity Domain:** What things is the dataset about?
- **Information Domain:** What information does the dataset contain about those things?
- **Composition:** How is the information expressed in RDF?
- **Reusability:** Can the dataset be used easily?

Ideally, RDFSynopsis clearly answers these questions. To this end, the most meaningful measures have to be chosen, while the total number of measures should remain small.

In the following sections, we will argue which measures are best suited to answer each central question. Table 3.2 gives an overview of the chosen measures, and their relation to the four central questions. We also include several questions for each measure to describe the insight it offers.

3.4.1. Entity Domain

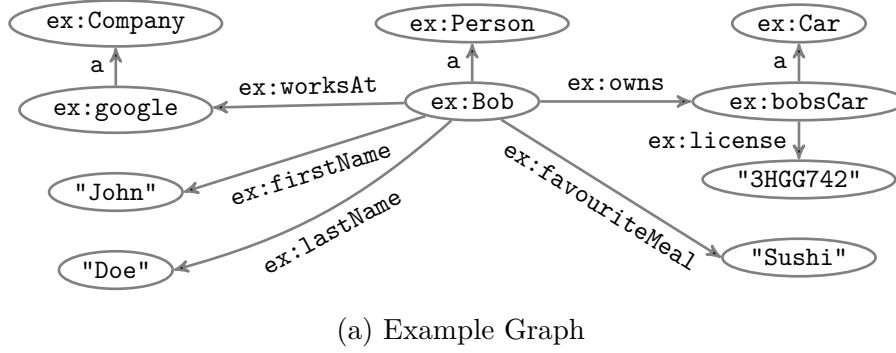
First and foremost, to give an overview of the kinds of things that the dataset is about, the obvious approach is naming the most frequently instantiated classes, i.e., the URIs that occur as objects in triples like

```
?instance a ?class .
```

This measure can be found in table 3.1 by the name of *class usage count*.

Usually, the number of triples per subject varies significantly between classes of subjects and even individual subjects. For this reason, the number of class instances must not be confused with the amount of data (i.e., the number of triples) associated with instances of a class. To measure the latter, the number of *triples per subject class* can be counted.

The difference between *class usage count* and *triples per subject class* can be seen in the example depicted in figure 3.2. There are three class instances and nine triples, six of which describe the person instance. While the former instance distribution is best expressed using the *class usage count* measure, the actual data distribution is better explained with *triples per subject class*.



Class	<i>class usage count</i>	<i>triples per subject class</i>
ex:Person	1	6
ex:Car	1	2
ex:Company	1	1

(b) Results

Figure 3.2.: Example for Class-based Histograms

At a closer look, several factors can weaken the expressiveness of both class-based measures.

Untyped Resources. A dataset can contain untyped resources, i.e., resources without `rdf:type` statements. If datasets consist of such untyped resources to a large extent, then class frequencies are of small value. To enable an assessment of the expressiveness of class-based measures, a measure for the “type coverage” of datasets is useful. This coverage measure can be found in table 3.1 as *typed-subject-ratio* and is computed as:

$$\text{typed-subject-ratio} = \frac{\# \text{typed subjects}}{\# \text{all subjects}}$$

Multiple Types. Resources are frequently assigned to multiple types, i.e., they occur in more than one `rdf:type` statement. Figure 3.3 shows examples for three different cases of parallel class usage. An instance can be explicitly defined to belong to two classes (`ex:mother` and `ex:firefighter`) that have no hierarchical relation (3.3a). When all `ex:firefighters` are also explicitly of type `foaf:Person`, then we could call this an implicit hierarchical relation between those classes (3.3b). If this relation is made explicit via `rdfs:subClassOf`, then the `foaf:Person` type may only be implied by the `ex:firefighter` type (3.3c). In less obvious cases one could wrongly assume that two hierarchically related classes represented unrelated concepts. Therefore, information about the *explicit class hierarchy* and *implicit class hierarchy* should be integrated with the class histograms. This could partly be done in the sense of “*X* times `foaf:Person`, out of which *Y* times `ex:firefighter`”.

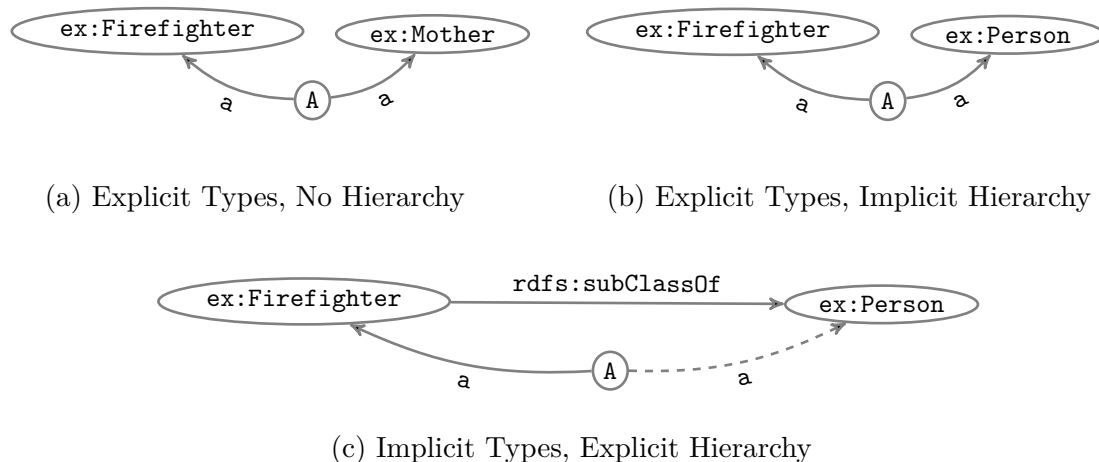


Figure 3.3.: Unrelated Classes, Implicit and Explicit Class Hierarchy

Mix of Modeling Levels. Class histograms mix the two levels of modeling, namely the *model level* and the *meta model level*, when they treat both triples

```
ex:Jane a ex:mother .
ex:mother a rdfs:Class .
```

equally. Although technically correct - as we have seen (in section 2.4.1) all levels are expressed in RDF triples -, this circumstance tends to represent a disadvantage; usually one of the levels will be dominant in the dataset and block out the other in frequency sorted class histograms. Class histograms should therefore distinguish between both levels. The dominance of the levels can be computed as

$$\text{ontology-ratio} = \frac{|\text{term instances}|}{|\text{all instances}|} = \frac{|CL| + |P| + |CO|}{|I|}$$

where CL , CO , and P are the sets of all class (`rdfs:Class` or `owl:Class`), concept (`skos:Concept`), and property (`rdf:Property`) instances, respectively. The set of all other instances is denoted by I . An ontology-ratio close to 1 indicates a dataset that mainly contains terminological statements (meta model level), while a value close to 0 indicates a majority of assertional statements (model level).

Summary. Two kinds of class-based histograms, *class usage count* and *triples per subject class*, are proposed to show what kind of objects are at the center of a given dataset. The first indicates how often a class is instantiated (i.e., used as value for `rdf:type`). The second histogram contains the number of triples a dataset contains with subjects of a specific class. These histograms should integrate the *explicit and implicit class hierarchies*, and distinguish between model and meta model levels. The *ontology-ratio* indicates, which level dominates the dataset. Class-based measures are only meaningful, if a large percentage of the dataset's subjects actually have a type. The percentage of typed subjects is computed as the *typed-subject-ratio*.

3.4.2. Information Domain

Information about resources is expressed in RDF using triples of the form

```
subject predicate object .
```

where the described resource appears as the subject (or object). What kind of information a triple expresses, is defined by the predicate. Therefore, the kind of information a dataset contains can be explained with a list of the most frequently used predicates (*property usage*).

In some cases, variations of the *property usage* measure may be more useful.

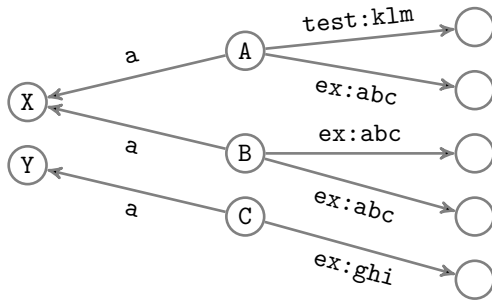
Predicate Vocabularies. Even medium-sized datasets can contain hundreds of different properties; a list of property frequencies is very likely to become too big to be of much help. Because predicates usually stem from external vocabularies, and those vocabularies typically group together predicates from the same domain or area of purpose, the *predicate vocabularies* used in a dataset present a much more concise, yet very expressive measure for the kind of information a dataset contains.

Relation of Class and Property Usage. If datasets contain instances of many different classes together with many different properties, the relation between the subjects' classes and predicates deserve a closer look. One approach is to subdivide the property usage measure by subject class (*property usage per subject class*), i.e. to count how many times a predicate is used together with subjects that are instances of a given class. This measure might reveal that some predicates are exclusively used by a subset of the classes in the dataset. A closely related, but different approach is to count the number of class instances that occur in triples with the given predicate, i.e., how many instances provide values for a specific property. This measure (*class instances per property*) can be used to compute the maximum number of different class instances a query may yield for the combination of specific classes and properties. An important subset of this measure are the *common properties* of a class, i.e., the set of properties *all* instances of a class provide values for. Queries for a class and its common properties do not filter out any class instance in the dataset. Figure 3.4 shows example results for these property-based measures. Note that the `rdf:type` property is not taken into account, since it provides the class information and is therefore implicitly included in the analysis.

Multiple Properties. The same subjects and objects can occur in triples with different predicates.

```
subject p1 object.  
subject p2 object.
```

Analogous to instances that have multiple classes (see previous section 3.4.1), this can either mean that the two statements are explicitly or implicitly contained in



(a) Example Graph

Class	<i>common properties</i>
X	{ex:abc}
Y	{ex:ghi}

(b) Common Properties

Vocabulary	<i>predicate vocabularies</i>
ex:	4
test:	1

(c) Predicate Vocabularies

Predicate	<i>property usage</i>
ex:abc	3
ex:ghi	1
test:klm	1

(d) Property Usage

Predicate	Class	<i>property usage per subject class</i>
ex:abc	X	3
	Y	0
ex:ghi	X	0
	Y	1
test:klm	X	1
	Y	0

(e) Property Usage per Subject Class

Class	Predicate	<i>class instances per property</i>
X	ex:abc	2
	ex:ghi	0
	test:klm	1
Y	ex:abc	0
	ex:ghi	1
	test:klm	0

(f) Class Instances per Property

Figure 3.4.: Example for Property-based Measures

the dataset. The implicit case refers to the existence of `rdfs:subPropertyOf` statements, that express a hierarchical relation between two properties. If a property is `rdfs:subPropertyOf` another property, then triples with the latter can be inferred from triples with the former property. Three cases of parallel property usage are shown in figure 3.5, including properties whose hierarchical relations are implicit (3.5b). The *explicit property hierarchy* and *implicit property hierarchy* reveal these relationships between properties.

Mix of Modeling Levels. It might be useful to clearly separate the model and meta model levels in property histograms, because these different levels are usually studied separately.

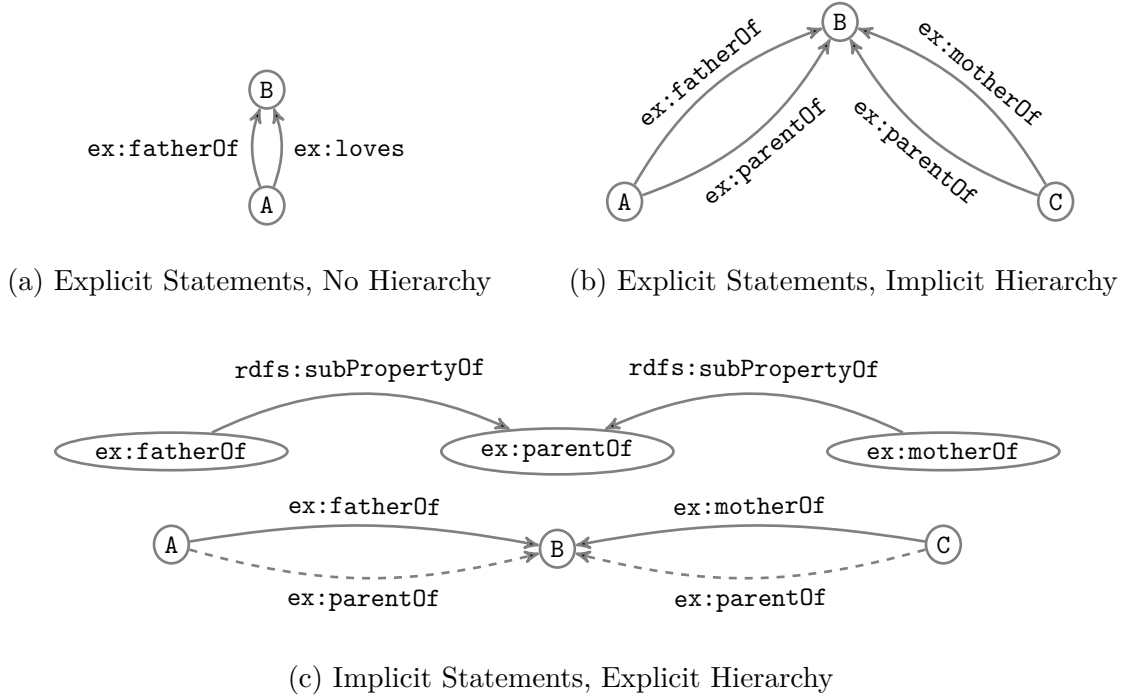


Figure 3.5.: Implicit vs. Explicit Property Hierarchy

Summary. The *property usage* measure, an overview of the predicates used in a dataset and their frequencies, provides insight into the “payload” of the dataset, i.e., the kinds of contained information. Because a high number of different predicates in a dataset may diminish the usefulness of that measure, an overview of the *predicate vocabularies* may be favorable. The classes of subjects may also be taken into account to provide better insight into information content; either as the distribution of predicates over classes (*property usage per subject class*) or as the number of class instances over predicates (*class instances per property*). The set of properties, that all instances of a specific class provide values for (*common properties*), can be used to explicitly query the dataset for properties without filtering out any class instance. Analogously to the previous central question, the property histograms should integrate the *explicit and implicit property hierarchies*, and should distinguish between model and meta model levels.

3.4.3. Composition

As different as the questions for kinds of entities and information (first and second central question), and composition (third central question) are, their answers have a lot in common, from a technical point of view. Thus, many of the criteria from the two previous sections can also help in understanding a dataset’s structure.

Hierarchical Relations. One of the most fundamental modeling decisions concerns hierarchical relations within the data. The same real world fact can be modeled on different levels of generality. To partly reiterate a previous example, the fact that

```
ex:Jane a ex:firefighter .  
ex:Jane ex:chiefOf ex:Tim .
```

may also be expressed in more general terms as

```
ex:Jane a foaf:Person .  
ex:Jane foaf:knows ex:Tim .
```

It is up to the creator of the dataset to decide, what level of generality is used. Often, different levels are contained in parallel, i.e., with respect to the example, all 4 triples may be included in the dataset. The appropriate measures reveal these implicit and explicit hierarchies of classes (*implicit class hierarchy* and *explicit class hierarchy*) and properties (*implicit property hierarchy* and *explicit property hierarchy*).

Implicit Schema. A dataset is usually created with a rather clear mental image of its purpose and contents. This usually means that it does not contain a random set of facts, but rather a set of similar information about similar entities. In the domain of top-down data modeling we could expect to find the same set of attributes for all entities of a given type. But, as we have seen in section 2.4.2, information modeling in RDF works bottom-up, i.e., we cannot rely on any kind of schema for entity descriptions. That is why it is very interesting to analyze how properties and classes are actually combined in the dataset. We can count how often a property is used per subject type (*property usage per subject class*) to see if a property is only included to describe specific kinds of subjects; and we can count the number of class instances per property (*class instances per property*) to distinguish between *common properties* that all instances provide, and those properties that are only used by some class instances. Finding the set of *common properties* can also be understood as extracting an implicit “minimum schema” for the given entity type.

Use of Blank Nodes. An important aspect of the way a dataset is structured concerns the use of blank nodes, i.e., nodes in an RDF graph that do not have a unique URI. They are typically used (but not necessarily needed) to model n-ary relationships (relating more than two nodes), or simply when external node references are not desired. The number of subjects without URIs is measured by *distinct blank subjects*. The *isolated blanks* measure focuses on the subset of blank nodes, that provide not even indirect means of identification (via `owl:InverseFunctionalProperty`), and hence remain completely isolated, i.e., they cannot be externally referenced or merged.

External Resource. In addition to its internal triple structure, an RDF dataset may also be part of an external structure of interlinked datasets, namely the Web of Data. RDF only fulfills its purpose of conveying data with explicit semantics, if terms from shared vocabularies are used, and externally defined resources are linked to. The vocabularies used in the dataset can be identified with the *class*

usage count and *predicate vocabularies* measures. Links to and between external resources are more concisely presented if all URIs with a common namespace are grouped together (*namespace links*). Explicit equivalence statements between local and external resources (*sameAs*) deserve special attention, because they ease merging.

Summary. The structure of an RDF dataset can be explored by looking at implicit and explicit hierarchies of classes (*implicit class hierarchy* and *explicit class hierarchy*) and properties (*implicit property hierarchy* and *explicit property hierarchy*). Because data modeling in RDF works bottom-up, we have to explicitly analyze the combinations of classes and properties (*property usage per subject class* and *class instances per property*). Finding the set of *common properties* can also be understood as extracting an implicit “minimum schema” for the given entity type. A high number of blank nodes (*distinct blank subjects*) may mean more complex structures. Other measures indicate, whether the dataset promotes (*namespace links*) or undermines (*isolated blanks*) the *linked data* idea.

3.4.4. Reusability

The reusability of an RDF dataset includes two aspects, the accessibility for data extraction and the suitability for merging with other datasets.

Size. The dataset’s size (*triples*) determines whether processing and retrieving of larger parts of the data is feasible.

Type Information. The basis for any kind of structured querying (and many of the criteria proposed in this thesis) are type information for the dataset’s resources. (*typed-subject-ratio*)

Detecting Ontologies. When looking for data about real world entities and relationships, one may want to treat ontologies separately (or not at all). RDF datasets that represent ontologies can be detected by a high percentage of term definitions (*ontology-ratio*).

Mergeability. Merging multiple RDF datasets is only useful, if the resulting graph can be treated as *one* dataset. This is only the case if the datasets use terms from the same vocabularies (*class usage count* and *predicate vocabularies*) or otherwise use the same (typically external) URIs for subjects or objects. The probability for two datasets to link to the same external resources can be estimated by analyzing the number of triples that link different namespaces (*namespace links*). Two different URIs can immediately be merged if their equivalence is explicitly asserted by `owl:sameAs` triples (*sameAs*). Blank nodes that cannot indirectly be identified (via `owl:InverseFunctionalProperty`), cannot be referenced from the outside or automatically merged with other datasets (*isolated blanks*).

Homogeneity. A dataset is most accessible if the desired parts can be retrieved with single straight-forward queries, i.e., by querying simply for the desired classes and properties without optional or conditional cases. A first important aspect to this end is the existence of explicitly defined class (*explicit class hierarchy*) and property (*explicit property hierarchy*) hierarchies, that provide for a more homogeneously structured dataset, and allow to formulate queries on the desired level of generality without missing relevant parts. The extent to which hierarchies are explicitly contained, can be determined by comparing explicit with implicit (*implicit class hierarchy* and *implicit property hierarchy*) hierarchies. The latter might reveal hierarchies that are contained but not explicitly modeled. A second aspect concerns the completeness of property usage by instances of a specific class. The set of *common properties* that all instances of a given class provide, can be used to query the dataset without missing an instance of that class. The set of common properties should not be too small compared to all properties used with that class.

Summary. Size (*triples*) and type (*ontology-ratio*) of a dataset are basic criteria to judge its reusability. Of equal importance are type information (*typed-subject-ratio*). The desired subset of a dataset is easier to retrieve if class and property hierarchies (*explicit class hierarchy* and *explicit property hierarchy*) are explicitly modeled, and the class instances share a large set of *common properties*. Explicit equivalence statements (*sameAs*), and links to external namespaces (*namespace links*), including terms from widely used vocabularies (*class usage count* and *predicate vocabularies*), are very important when merging multiple datasets.

3.4.5. Chosen Measures

We have chosen 18 statistical criteria which, in combination, help to answer our four *central questions*. For each criterion, the following table (3.2) lists the related central questions, and describes the given insight.

No.	Measure	CQ	Insight
1.	<i>class usage count</i>	1,4	<ul style="list-style-type: none"> · What classes are used? · What classes have the highest number of instances? · From which domain is the data?
2.	<i>triples per subject class</i>	1	<ul style="list-style-type: none"> · Instances of which classes does the dataset mainly describe? · What classes are used for triples' subjects?
3.	<i>explicit class hierarchy</i>	1,3,4	<ul style="list-style-type: none"> · Do the classes have hierarchical relations with each other? · What are the most general or most specific classes? · What triples can (will) be inferred?
4.	<i>implicit class hierarchy</i>	1,3,4	<ul style="list-style-type: none"> · How heterogeneous is the data structure? · How easy is it to query large parts of the data? · Do we have to take the different classes into account? · Instances of which classes will I get if I query for a specific class?
5.	<i>ontology-ratio</i>	1,4	<ul style="list-style-type: none"> · What are the proportions of data payload and terminological definitions? · Does the dataset represent an ontology?
6.	<i>typed-subject-ratio</i>	1,4	<ul style="list-style-type: none"> · To what extent are type information provided for triple subjects?
7.	<i>property usage</i>	2,3	<ul style="list-style-type: none"> · What are the most frequent properties used in the dataset?
8.	<i>predicate vocabularies</i>	2,3,4	<ul style="list-style-type: none"> · What vocabularies have been used for the properties? · From which domain is the data?
9.	<i>property usage per subject class</i>	2,3	<ul style="list-style-type: none"> · What properties are used with which classes? · What are the most frequent properties for specific classes? · What properties represent the payload?
10.	<i>class instances per property</i>	2,3,4	<ul style="list-style-type: none"> · How many instances of a specific class will I get if I query for a specific property? · How heterogeneous is the data structure? · How easy is it to query large parts of the data? · What information can I get for all subjects (of a given class) in the dataset?
11.	<i>explicit property hierarchy</i>	2,3,4	<ul style="list-style-type: none"> · Do the properties have explicit hierarchical relations with each other? · What are the most general or most specific properties? · What triples can (will) be inferred?

Continued on next page...

... continued

No.	Measure	CQ	Insight
12.	<i>implicit property hierarchy</i>	2,3,4	<ul style="list-style-type: none"> · Do the properties have implicit hierarchical relations with each other? · What are the most general or most specific properties? · How heterogeneous is the data structure? · How easy is it to query large parts of the data? · Do we actually have to take the different properties into account?
13.	<i>distinct blank subjects</i>	3	<ul style="list-style-type: none"> · Does the dataset contain blank nodes?
14.	<i>namespace links</i>	3, 4	<ul style="list-style-type: none"> · Does the dataset contain links to other namespaces, i.e, is it <i>linked data</i> ? · Which namespaces are linked in the dataset? · What properties are used to link namespaces?
15.	<i>isolated blanks</i>	3, 4	<ul style="list-style-type: none"> · How hard is it to merge the given dataset with others? · Can we link to the subjects of the dataset?
16.	<i>triples</i>	4	<ul style="list-style-type: none"> · How big is the dataset?
17.	<i>sameAs</i>	4	<ul style="list-style-type: none"> · How hard is it to merge the given dataset with others? · Does the dataset express equivalence of resources?
18.	<i>common properties</i>	2,3,4	<ul style="list-style-type: none"> · How heterogeneous is the data structure? · How easy is it to query large parts of the data? · What information can I get for all subjects (of a given class) in the dataset?

Table 3.2.: Overview: Measures chosen for structural analysis with related central questions (CQ). For each measure, one or more questions are given that can be answered based on the measure's information.

4. Design and Implementation of Structural Analysis

In this chapter we describe the design and implementation of the structural analysis of RDF datasets through SPARQL endpoints.

During our investigation of structure analysis in the last chapter, we identified four central questions to understand an RDF dataset. We finally chose 18 criteria, which help answering these questions. In this chapter, we show how SPARQL can be used to analyze a dataset with regard to these criteria.

We introduce two general approaches for the SPARQL-based structural analysis of RDF datasets; first, the Specific Query Approach (SQA) in section 4.1.1, and second, the Triple Stream Approach (TSA) in section 4.1.2. Thereafter, we present the implementation of each criterion in terms of a SPARQL query (for SQA), and a triple filter (for TSA). This self-contained *catalog of RDF analysis criteria* represents one of the main contributions of this thesis (section 4.1.3). To prove our concept, we have implemented all criteria and both approaches in a software called RDFSynopsis. We describe its basic architecture and usage in section 4.2.

4.1. Structural Analysis via SPARQL

In this section, we illustrate our approaches of using SPARQL to perform structural analyses of RDF datasets, in terms of the 18 measures we identified in chapter 3.

We begin by describing two alternative approaches, namely SQA (section 4.1.1) and TSA (section 4.1.2). In section 4.1.3 we present the implementation of both approaches for each measure.

4.1.1. Specific Queries

Our first approach to analyze a dataset that is available through a given SPARQL endpoint, is to formulate one SPARQL query per measure. We refer to this as the Specific Query Approach (SQA).

With SQA, we conduct a full analysis of a dataset by subsequently performing a set of queries, each of which is specifically tailored to one measure. This simple process is visualized in the sequence diagram in figure 4.1.

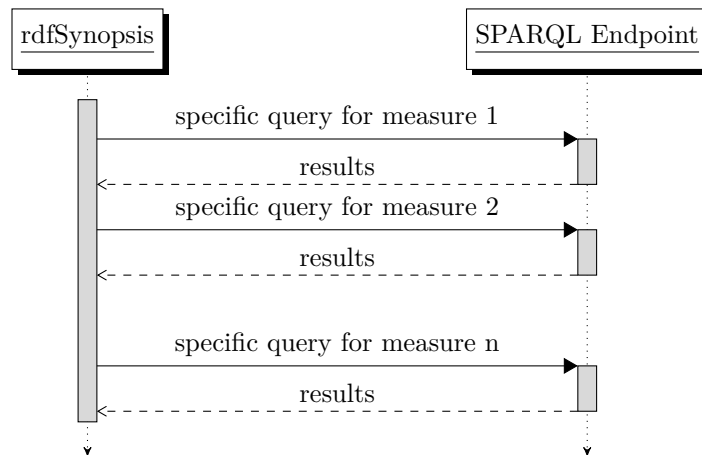


Figure 4.1.: Sequence Diagram for Specific Query Approach (SQA)

4.1.2. Triple Stream

Our second approach for a SPARQL-based dataset analysis, is to retrieve and analyze a sequence of triples. We refer to this as the Triple Stream Approach (TSA).

With TSA, we conduct a full analysis of a dataset by subsequently retrieving all triples of a dataset. The different measures are implemented in terms of triple filters;

each measure decides whether and how it takes a triple into account. The analysis is finished when all triples of a dataset have been filtered by each measure.

In the majority of cases it will be infeasible to retrieve the full dataset with one query. Triple stores usually constrain memory and time used per query, and these constraints are easily exceeded for larger datasets. We therefore perform several queries each retrieving only a part of the full dataset. This process is visualized in the sequence diagram in figure 4.2.

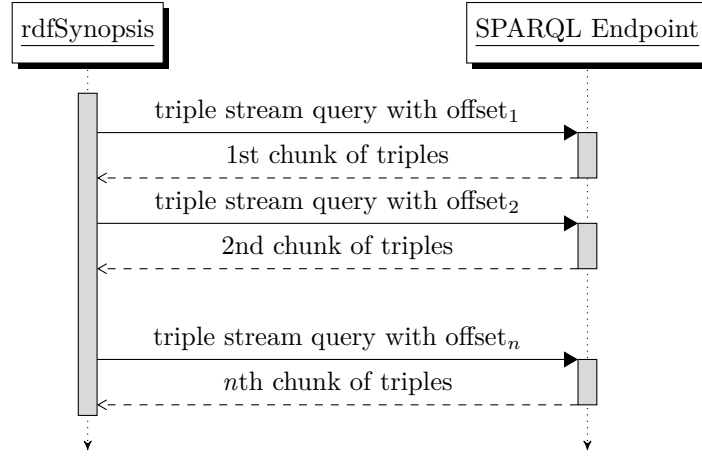


Figure 4.2.: Sequence Diagram for Triple Stream Approach (TSA)

The following SPARQL query can be used to subsequently retrieve all triples of a dataset.

```

1 SELECT ?s ?p ?o
2 WHERE {?s ?p ?o.}
3 ORDER BY ...
4 LIMIT ...
5 OFFSET ...

```

The graph pattern $\{?s \ ?p \ ?o.\}$ matches all triples (line 2). We use the **LIMIT** keyword to define the “chunk size”, i.e., an upper bound on the number of received triples (line 4). The **ORDER BY** clause is used to define a sequential order on the otherwise unordered RDF graph (line 3). We iterate over this sequence by subsequently using offsets incremented by the “chunk size” (line 5).

Figure 4.3 illustrates how the different SPARQL keywords define a sequence of triple “chunks”, that can be iteratively requested, to create the *triple stream*.

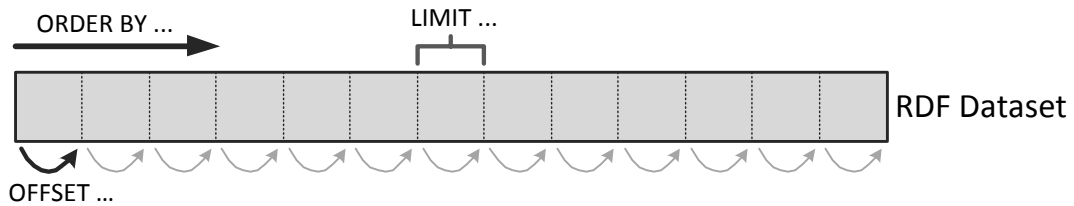


Figure 4.3.: Creating a Triple Stream with SPARQL

4.1.2.1. Partial Analysis & Random Sampling

An obvious disadvantage of the plain Triple Stream Approach is that it requires the transfer of the whole dataset from the SPARQL endpoint; a very expensive and time-consuming process. A potential solution to this problem is to refrain from a full analysis and only request a subset of all triples.

Because the `ORDER BY` clause imposes some sequence on the triples in the dataset, a partial analysis, that only takes the first k triples into consideration, is very unlikely to produce results that are representative for the full dataset.

In order to mitigate this adverse effect and to obtain a more representative sample of triples, we follow an approach to randomly select the parts we request from a dataset. This approach is visualized in figure 4.4.

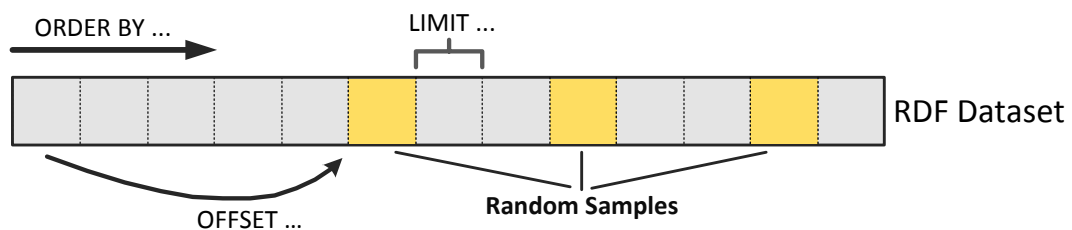


Figure 4.4.: Random Sampling TSA

4.1.3. Measure Implementation

This section demonstrates how all 18 identified measures can be implemented, both as a specific query and a triple filter.

For each measure, we give an informal description of its purpose, a specific SPARQL query (for SQA), and a triple filter definition including data structure and necessary post processing (for TSA). In the pseudocode descriptions of the triple filters, we use a `for each` construct that is meant to iterate over all pairs (k, v) of a $\text{Map}(K \rightarrow V)$.

4.1.3.1. Class Usage Count

Understand which classes are most frequently instantiated; that is, how many subjects have an `rdf:type` relationship to that class.

Due to RDF semantics, `rdfs:Resource` will always¹ be the most frequently instantiated class. The sets of instances are *not disjoint*, because one resource can have multiple types.

SQA

```

1 SELECT ?class (COUNT(DISTINCT ?I) AS ?numInstances)
2 WHERE {
3   ?I rdf:type ?class .
4 }
5 GROUP BY ?class

```

This query will return the number of instances per class.

TSA

```

1 // Data Structures
2 classUsageMap : Map(String → Integer)
3
4 function FILTERTRIPLE(s, p, o)
5   if p == rdf:type then
6     classUsageMap[o]++ // object == class
7
8 // Necessary Post-Processing
9   none

```

For every `rdf:type` triple, the instance count for the class (`object`) is incremented.

¹Always, unless SPARQL-Endpoint or underlying triple store do not perform proper RDF(S) inference.

4.1.3.2. Triples per Subject Class

Understand how triples are distributed over subject classes. Because subjects can have multiple types, triples are counted multiple times, once for each subject class. Hence, the sets of triples per class are *not disjoint*.

SQA

```
1 SELECT ?class (COUNT(*) as ?numTriplesPerClass)
2 WHERE {
3   ?s ?p ?o .
4   ?s a ?class .
5 }
6 GROUP BY ?class
```

The query will return, for each class, the number of triples whose subjects are an instance of that class.

TSA

```
1 // Data Structures
2 triplesPerClass : Map(String → Integer)
3 triplesPerSubject : Map(String → Integer)
4 classesPerSubject : Map(String → {String})
5
6 function FILTERTRIPLE(s, p, o)
7   triplesPerSubject[s]++
8   if p == rdf:type then
9     classesPerSubject[s].add(o) // object == class
10
11 // Necessary Post-Processing
12 for each (subject, subjectClasses) in classesPerSubject do
13   for each class in subjectClasses do
14     triplesPerClass[class] += triplesPerSubject[subject]
```

The number of triples and the classes per subject are counted separately. They are combined in post-processing to calculate the number of triples per class.

4.1.3.3. Explicit Class Hierarchy

Find out what class hierarchies are explicitly defined. Understand what subject types are inferred in the given dataset. Often, RDF resources are instances of multiple types. A usual reason is the existence of class hierarchies, i.e., one class is defined to be `rdfs:subClassOf` another class, and all instances of the former can be inferred automatically to be instances of the latter class, as well.

SQA

```

1 SELECT ?subclass ?superclass
2 WHERE {
3   ?subclass rdfs:subClassOf ?superclass .
4   FILTER (?subclass != ?superclass)
5   FILTER (?superclass != rdfs:Resource)
6 }
```

This query returns all subclass relationships, except for trivial and `rdfs:Resource` subclasses.

TSA

```

1 // Data Structures
2   subSuperClassMap : Map(String → {String})
3
4 function FILTERTRIPLE(s, p, o)
5   if p == rdfs:subClassOf && s != o && o != rdfs:Resource then
6     subSuperClassMap[s].add(o)
7
8 // Necessary Post-Processing
9   none
```

Record all `rdfs:subClassOf` triples, except for trivial and `rdfs:Resource` subclasses.

4.1.3.4. Implicit Class Hierarchy

Find out what class hierarchies are implicitly contained. In many cases all instances of one class are also defined to be instances of another class.

SQA

```

1 SELECT ?subClass ?superClass
2 WHERE {
3   ?s a ?subClass .
4   ?s a ?superClass .
5   FILTER (?subClass != ?superClass)
6   FILTER (?superClass != rdfs:Resource)
7   FILTER NOT EXISTS {
8     ?anyInstance a ?subClass .
9     FILTER NOT EXISTS {
10      ?anyInstance a ?superClass . }
11   }
12 }
13 GROUP BY ?subClass ?superClass

```

The query looks for pairs of classes (`?subClass`, `?superClass`) that are used in parallel for some subject (lines 3-4), and for which there is no instance (`?anyInstance`) that is instance of the first but not of the second class (lines 7-11). The query returns all class hierarchy relationships, including the explicit class hierarchy. We can read the results as: “The set of `subClass` instances is a *subset* of the set of `superClass` instances.”

TSA

```

1 // Data Structures
2   subSuperClassMap : Map(String → {String})
3   classInstances   : Map(String → {String})
4
5 function FILTERTRIPLE(s, p, o)
6   if p == rdf:type && o != rdfs:Resource then
7     classInstances[o].add(s)
8
9 // Necessary Post-Processing
10 for each (class, classInstances) in classInstances do
11   for each (superClass, superClassInstances) in classInstances do
12     if class != superClass && classInstances ⊆ superClassInstances then
13       subSuperClassMap[class].add(superClass)

```

During triple streaming all class instances are recorded. To find implicit class hierarchies in post-processing, the sets of class instances are searched for subset relations.

4.1.3.5. Ontology-Ratio

Determine whether a dataset represents an ontology, i.e., terms and their relationships, or rather a set of assertions about real world entities. The extent to which a dataset is made of terminological definitions may be computed as below. High (close to 1) and low (close to 0) ontology-ratios indicate terminological (ontology) or assertional datasets, respectively. Note that we define “terms” in a broader sense, to also include instances of type `skos:Concept`.

$$\text{ontology-ratio} = \frac{|\text{term instances}|}{|\text{all instances}|} = \frac{|CL| + |P| + |CO|}{|I|}$$

SQA

```

1 SELECT (COUNT(DISTINCT ?CL) AS ?numDefClasses)
2         (COUNT(DISTINCT ?CO) AS ?numDefConcepts)
3         (COUNT(DISTINCT ?P) AS ?numDefProperties)
4         (COUNT(DISTINCT ?I) AS ?numTypedResources)
5 WHERE {
6   { ?CL a rdfs:Class. }
7   UNION
8   { ?CL a owl:Class. }
9   UNION
10  { ?CO a skos:Concept. }
11  UNION
12  { ?P a rdf:Property. }
13  UNION
14  { ?I a ?class.
15    FILTER(?class != rdfs:Resource)}
16 }
```

This query will return the number of class, property and concept instances. It will also return the number of all instances, including term and non-term instances.

TSA

```

1 // Data Structures
2   defClasses      :   {String}
3   defProperties    :   {String}
4   defConcepts     :   {String}
5   typedResources  :   {String}
6
7 function FILTERTRIPLE(s, p, o)
8   if p == rdf:type then
9     if o != rdfs:Resource then
10      typedResources.add(s)
11     if o == rdfs:Class || o == owl:Class then
12      defClasses.add(s)
13     if o == skos:Concept then
14      defConcepts.add(s)
15     if o != rdf:Property then
16      defProperties.add(s)
17
18 // Necessary Post-Processing
19   none

```

Depending on their type subjects are added to different sets whose sizes can later be used to compute the ontology-ratio.

4.1.3.6. Typed-Subject-Ratio

Determine to what extent type information is provided for subjects in a dataset. This criterion is important to judge whether other class-based criteria, that rely on typed subjects, are expressive at all.

$$typed\text{-}subject\text{-}ratio = \frac{|typed\ subjects|}{|all\ subjects|}$$

SQA

```

1 SELECT (COUNT(DISTINCT ?TS) AS ?numTypedSubjects)
2        (COUNT(DISTINCT ?S) AS ?numSubjects)
3 WHERE {
4   { ?TS a ?class .
5     ?TS ?p ?o .
6     FILTER (?class != rdfs:Resource) }
7   UNION
8   { ?S ?p ?o . }
9   FILTER (?p != rdf:type)
10 }

```


The query will return the number of typed subjects and the number of total subjects. Note that we exclude `rdfs:Resource` from matched subject classes as, all subjects are implicitly of type `rdfs:Resource`, due to RDF semantics. For the same reason, we exclude `rdf:type` triples; we do not want to count resources (as subjects) that *only* appear in `rdf:type` triples.

TSA

```

1 // Data Structures
2   subjects      :   {String}
3   onlyTypedSubjects :   {String}
4   typedSubjects   :   {String}
5
6 function FILTERTRIPLE(s, p, o)
7   if p == rdf:type then                                // type triple
8     if o != rdfs:Resource then
9       if s ∈ subjects then
10        typedSubjects.add(s)
11      else
12        onlyTypedSubjects.add(s)
13   else                                                // non-type triple
14     if s ∈ onlyTypedSubjects then
15       typedSubjects.add(s)
16     subjects.add(s)
17
18 // Necessary Post-Processing
19   none

```

All subjects found in non-type triples are added to `subjects`. All subjects found in type triples are either immediately added to `typedSubjects` (if they are already in `subjects`), or are marked as `onlyTyped` (if they have not occurred in non-type triples so far), so they are latter added to `typedSubjects` when occurring in a non-type triple.

4.1.3.7. Property Usage

Find the most frequently used properties.

SQA

```
1 SELECT ?property (COUNT(*) as ?numUses)
2 WHERE {
3   ?s ?property ?o .
4 }
5 GROUP BY ?property
```

The query will return, for each property, the number of triples using that property.

TSA

```
1 // Data Structures
2   propertyUsageMap :   Map(String → Integer)
3
4   function FILTERTRIPLE(s, p, o)
5     propertyUsageMap[p]++
6
7 // Necessary Post-Processing
8   none
```

For each triple the property's usage count is incremented.

4.1.3.8. Predicate Vocabularies

Get an *overview* of property usage by subsuming properties under their vocabularies. Classes and properties in RDF are typically defined in datasets that only contain terminological definitions, so-called vocabularies. Usually, all terms within a vocabulary have a common prefix in their URI path, their “namespace”. Because vocabularies group together terms from a specific domain, the namespaces of used terms are usually enough to understand the dataset’s domain(s).

SQA

```

1 SELECT ?predVocab (COUNT(?p) AS ?numUses)
2 WHERE {
3   ?s ?p ?o .
4   FILTER(isIRI(?p))
5   BIND(REPLACE(str(?p), "[^/#]*$", "") AS ?predVocab)
6 }
7 GROUP BY ?predVocab

```

In the query, the namespace that identifies a vocabulary is obtained by truncating the fragment identifier (part after #) or, if none exists, the last path segment (lines 5-6). The query will return, for each property vocabulary, the number of triples with a property from that vocabulary.

TSA

```

1 // Data Structures
2 predicateVocabularyUsageMap : Map(String → Integer)
3
4 function FILTERTRIPLE(s, p, o)
5   predicateVocabularyUsageMap[namespace(p)]++
6
7 // Necessary Post-Processing
8   none

```

For each triple the property’s namespace is computed and its usage count is incremented.

4.1.3.9. Property Usage per Subject Class

Understand what are the most frequent combinations of subject class and property.

SQA

```
1 SELECT ?class ?property (COUNT(*) as ?numUses)
2 WHERE {
3   ?s ?property ?o .
4   OPTIONAL {?s a ?class}
5   FILTER(?property != rdf:type)
6 }
7 GROUP BY ?class ?property
```

The query will return, for each class and each property, the number of triples with the given property and a subject that is an instance of the given class.

TSA

```
1 // Data Structures
2 propPerSubjectClassMap : Map(String → Map(String → Integer))
3 propPerUntypedSubjectsMap : Map(String → Integer)
4 subjectClassMap : Map(String → {String})
5 propPerSubjectMap : Map(String → Map(String → Integer))
6
7 function FILTERTRIPLE(s, p, o)
8   if p == rdf:type then
9     subjectClassMap[s].add(o) // object == class
10  else
11    propPerSubjectMap[s][p]++
12
13 // Necessary Post-Processing
14 for each (subject, propUsageMap) in propPerSubjectMap do
15   subjectClasses ← subjectClassMap[subject]
16   for each (property, numUses) in propUsageMap do
17     if subjectClasses == {} then // subject has no type
18       propPerUntypedSubjectsMap[property] += numUses
19     else // subject is typed
20       for each class in subjectClasses do
21         propPerSubjectClassMap[class][property] += numUses
```

During triple streaming, we record the classes and the property usage of each subject. In post-processing we calculate the property usage per subject class by combining property usage for all instances of a class.

4.1.3.10. Class Instances per Property

Understand how many instances of a given class have a triple with a given property. This number can be understood as the *maximum*¹ number of class instances we may expect from a query for the given property.

SQA

```

1 SELECT ?class ?property (COUNT(DISTINCT ?I) as ?numInstances)
2 WHERE {
3   ?I ?property ?o .
4   ?I a ?class .
5   FILTER (?property != rdf:type)
6 }
7 GROUP BY ?class ?property

```

The query will return, for each class and property, the number of class instances that occur as subject in a triple with the property.

TSA

```

1 // Data Structures
2 classInstancesPerPropertyMap : Map(String → Map(String → Integer))
3 subjectClassMap              : Map(String → {String})
4 propSubjectMap               : Map(String → {String})
5
6 function FILTERTRIPLE(s, p, o)
7   if p == rdf:type then
8     subjectClassMap[s].add(o) // object == class
9   else
10    propSubjectMap[p].add(s)
11
12 // Necessary Post-Processing
13 for each (property, subjects) in propSubjectMap do
14   for each subject in subjects do
15     subjectClasses ← subjectClassMap[subject]
16     if subjectClasses != ∅ then
17       for each class in subjectClasses do
18         classInstancesPerPropertyMap[class][property]++

```

We separately record subjects' classes (line 8) and subjects per property (line 10). To compute the number of class instances that occur in triples with a given property, we increment the number of class instances (line 18) for each class (lines 15-17) of every subject per property (lines 13-14).

¹The actual number of retrieved instances may be significantly lower when multiple properties are combined in a query, i.e., an intersection is computed.

4.1.3.11. Explicit Property Hierarchy

Determine which property hierarchies are explicitly defined in a dataset. Understand which triples (with which properties) are inferred in the dataset. Often, a dataset contains many triples that only differ in their properties. This is usually due to the fact, that those properties have a hierarchical `rdfs:subPropertyOf` relation with each other. These hierarchy statements can be found with the following query.

SQA

```
1 SELECT ?subprop ?superprop
2 WHERE {
3   ?subprop rdfs:subPropertyOf ?superprop .
4   FILTER (?subprop != ?superprop)
5 }
```

The query will return pairs of properties, that have an explicit subproperty relationship. Triples with *superprop* are inferred based on triples with *subprop*.

TSA

```
1 // Data Structures
2   subSuperPropertyMap :   Map(String → {String})
3
4 function FILTERTRIPLE(s, p, o)
5   if p == rdfs:subPropertyOf && s != o then
6     subSuperPropertyMap[s].add(o)
7
8 // Necessary Post-Processing
9   none
```

Record all `rdfs:subPropertyOf` triples, except for trivial sub-properties.

4.1.3.12. Implicit Property Hierarchy

Understand what property hierarchies are implicitly contained in a dataset. Often multiple properties are used in the same subject-object combinations. If all triples of one property A also exist with another property B, then we call this an implicit property hierarchy, and A an implicit sub-property of B.

SQA

```

1 SELECT ?subProp ?superProp
2 WHERE {
3   ?s ?subProp ?o.
4   ?s ?superProp ?o.
5   FILTER (?subProp != ?superProp)
6   FILTER NOT EXISTS {
7     ?anyS ?subProp ?anyO.
8     FILTER NOT EXISTS {
9       ?anyS ?superProp ?anyO. }
10  }
11 }
12 GROUP BY ?subProp ?superProp

```

The query looks for pairs of properties (`subProp`, `superProp`) (lines 3-4), for which no triple with `subProp` exists (lines 6-7) that does not exist with `superProp` (lines 8-9).

TSA

```

1 // Data Structures
2 subSuperPropMap : Map(String → {String})
3 triplesPerProperty : Map(String → Map(String → {String}))
4
5 function FILTERTRIPLE(s, p, o)
6   triplesPerProperty[p][s].add(o)
7
8 // Necessary Post-Processing
9 for each (p,pSubObjMap) in triplesPerProperty do
10   for each (superP,superPSubObjMap) in triplesPerProperty do
11     if p != superP then
12       containsAll ← true
13       for each (s,objects) in pSubObjMap do
14         containsAll ← containsAll && objects ⊆ superPSubObjMap[s]
15       if containsAll then
16         subSuperPropMap[p].add(superP)

```

The `triplesPerProperty` data structure records all triples in a `property → subject → object` order (line 6). To find implicit property hierarchies in post-processing, all pairs of properties (lines 9-11) are checked for subset relations. An implicit property hierarchy is found (line 16) if all `subject,object`-pairs of one property are included in the pairs of the potential super-property (lines 12-15).

4.1.3.13. Distinct Blank Subjects

Determine the number of blank nodes in a dataset.

SQA

```
1 SELECT (COUNT(DISTINCT ?s) as ?numBlanks)
2 WHERE {
3   ?s ?p ?o
4   FILTER isBlank(?s)
5 }
```

The query yields the number of blank nodes that appear in subject position.

TSA

```
1 // Data Structures
2   blankSubjects :   {String}
3
4 function FILTERTRIPLE(s, p, o)
5   if isBlank(s) then
6     blankSubjects.add(s)
7
8 // Necessary Post-Processing
9   none
```

All blank nodes appearing in the subject position are collected, no post-processing is necessary.

4.1.3.14. Namespace Links

Assess to what extent a dataset links different namespaces, and what properties are used for linking. In order to reuse a dataset, and merge it with others, it is important to link resources defined within a dataset with resources defined in external namespaces, so that equivalence of resources among multiple datasets can be computed.

SQA

```

1 SELECT ?NS1 ?prop ?NS2 (COUNT(*) AS ?numLinks)
2 WHERE {
3   ?res1 ?prop ?res2 .
4   FILTER(isIRI(?res1))
5   FILTER(isIRI(?res2))
6   BIND(REPLACE(str(?res1), "[^/#]*$", "") AS ?NS1)
7   BIND(REPLACE(str(?res2), "[^/#]*$", "") AS ?NS2)
8 }
9 GROUP BY ?NS1 ?prop ?NS2

```

The query will return, for each subject and object namespace and each property, the number of triples that have subjects and objects from the given namespaces, and use the given property. The links with the `owl:sameAs` property may deserve special attention, as it is used to explicitly define equivalence.

TSA

```

1 // Data Structures
2 nsLinksMap : Map(String → Map(String → Map(String → Integer)))
3
4 function FILTERTRIPLE(s, p, o)
5   nsLinksMap[namespace(s)][namespace(o)][p]++
6
7 // Necessary Post-Processing
8   none

```

The filter computes the namespaces for subject (NS1) and object (NS2), and increments the counter for the (NS1,NS2,P)-triples.

4.1.3.15. Isolated Blanks

Determine whether a dataset contains *isolated* subjects that cannot be referenced from outside the dataset. If blank nodes in a dataset do not appear as an object in any triple, and if they have no triple with an `owl:InverseFunctionalProperty`, then the blank node cannot be referenced from the outside.

SQA

```
1 SELECT (COUNT(DISTINCT ?bnode) AS ?numIsolatedBlanks)
2 WHERE {
3   ?bnode ?p ?o .
4   FILTER isBlank(?bnode)
5   FILTER NOT EXISTS {
6     ?anySubject ?hasObject ?bnode . }
7   FILTER NOT EXISTS {
8     ?bnode ?ifp ?id .
9     ?ifp a owl:InverseFunctionalProperty . }
10 }
```

The query will return the number of isolated blank nodes in the dataset.

TSA

```
1 // Data Structures
2   ifps                :   {String}
3   isolatedBlanks      :   {String}
4   subjectBlankPropertyMap :   Map(String → {String})
5   objectBlanks        :   {String}
6
7 function FILTERTRIPLE(s, p, o)
8   if p == rdf:type && o == owl:InverseFunctionalProperty then
9     ifps.add(s)
10  if isBlank(s) then
11    subjectBlankPropertyMap[s].add(p)
12  if isBlank(o) then
13    objectBlanks.add(o)
14
15 // Necessary Post-Processing
16 for each o in objectBlanks do
17   subjectBlankPropertyMap.remove(o)
18 for each (s,properties) in subjectBlankPropertyMap do
19   if (properties ∩ ifps) == ∅ then
20     isolatedBlanks.add(s)
```

During triple streaming we separately collect all `owl:InverseFunctionalProperty`s, and blank nodes appearing as subject and object. To obtain the set of *isolated* blank nodes, we first remove all blank nodes that also appeared as objects (lines 16-17). From the resulting set of blank subjects we select those that did not occur with an inverse functional property (lines 18-20).

4.1.3.16. Triples

Assess the dataset's size.

SQA

```
1 SELECT (COUNT(*) AS ?numTriples)
2 WHERE {
3   ?s ?p ?o .
4 }
```

The query will return the number of triples in the dataset. This may also include triples that are *implicitly* contained in the dataset, due to inference.

TSA

```
1 // Data Structures
2   numTriples : Integer
3
4 function FILTERTRIPLE(s, p, o)
5   numTriples++
6
7 // Necessary Post-Processing
8   none
```

4.1.3.17. SameAs

Determine the number of explicit equivalence statements via the `owl:sameAs` property.

SQA

```
1 SELECT (COUNT(*) as ?sameAs)
2 WHERE {
3   ?s owl:sameAs ?o .
4 }
```

TSA

```
1 // Data Structures
2   numSameAs : Integer
3
4 function FILTERTRIPLE(s, p, o)
5   if p == owl:sameAs then
6     numSameAs++
7
8 // Necessary Post-Processing
9   none
```

4.1.3.18. Common Properties

Find the set of properties that all instances of a given class share. The *common properties* of a class can be used in queries without filtering out any instance of that class.

SQA

```
1 SELECT DISTINCT ?class ?property
2 WHERE {
3
4 {SELECT ?class ?property (COUNT(DISTINCT ?I) as ?numInstances)
5 WHERE {
6   ?I ?property ?o .
7   ?I a ?class .
8   FILTER (?property != rdf:type)
9 }
10 GROUP BY ?class ?property
11 }
12 {
13   SELECT ?class (COUNT(DISTINCT ?I) AS ?numI)
14   WHERE {?I a ?class . }
15   GROUP BY ?class
16 }
17 FILTER (?numInstances = ?numI)
18 }
```

The query will return, for each class, the properties that all instances of the class provide values for. The query uses two subqueries, one to compute the number of instances per class (lines 12-16), and the other to compute the number of instances of a class with a specific property (lines 4-11). The common properties are then selected by combining the results of both queries to filter those properties whose number of class instances (`numInstances`) equals the total number of instances (`numI`) of the given class (line 17).

TSA

```

1 // Data Structures
2 classCommonPropertyMap      : Map(String → {String})
3 classInstancesPerPropertyMap : Map(String → Map(String → Integer))
4 classInstanceMap            : Map(String → {String})
5 subjectClassMap             : Map(String → {String})
6 propSubjectMap              : Map(String → {String})
7
8 function FILTERTRIPLE(s, p, o)
9   if p == rdf:type then
10     subjectClassMap[s].add(o)
11     classInstanceMap[o].add(s)
12   else
13     propSubjectMap[p].add(s)
14
15 // Necessary Post-Processing
16 for each (property, subjects) in propSubjectMap do
17   for each subject in subjects do
18     subjectClasses ← subjectClassMap[subject]
19     if subjectClasses != ∅ then
20       for each class in subjectClasses do
21         classInstancesPerPropertyMap[class][property]++
22 for each (class, propInstanceMap) in classInstancesPerPropertyMap do
23   for each (p, numInstances) in propInstanceMap do
24     if numInstances == classInstanceMap[class].size() then
25       classCommonPropertyMap[class].add(property)

```

We separately record subjects' classes (line 10), class instances (line 11) and subjects per property (line 13). To compute the number of class instances that occur in triples with a given property, we increment the number of class instances (line 21) for each class (lines 18-20) of every subject per property (lines 16-17). We select the common properties of a class (line 25) by comparing the number of class instances with that property to the total number of classes instances (line 24), for each class and property (lines 22-23).

4.2. Architecture of RDFSynopsis

In this section we briefly describe the architecture of RDFSynopsis, the software that implements our two approaches (SQA and TSA) to a structural analysis of RDF datasets via SPARQL endpoints.

The software RDFSynopsis was written in Java. It uses the Apache Jena framework [6] to process RDF data and to query SPARQL endpoints. The goal for the design of RDFSynopsis was to write code that could be used both, as a stand-alone tool operated from the command-line, and as a framework of classes for reuse in other software. Figure 4.5 presents the class hierarchy and package arrangement.

Three concepts are at the center of RDFSynopsis's design: Datasets, Statistical Criteria, and Analyzers. The basic idea behind these concepts is to define a dataset analysis in three steps.

1. Define the dataset that shall be analyzed.
2. Define the statistical criteria according to which the dataset shall be analyzed.
3. Provide the dataset and the criteria to an analyzer, and let it do the work.

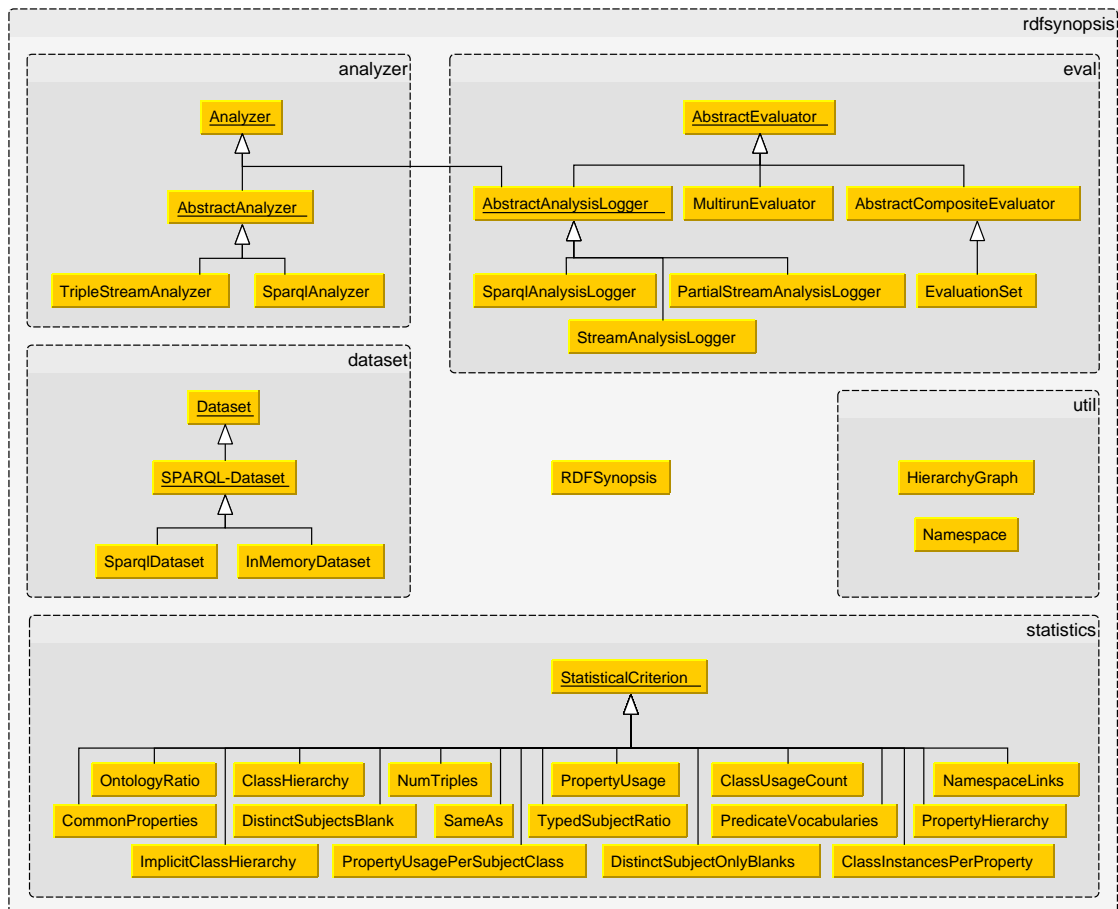


Figure 4.5.: Architecture of RDFSynopsis

A **dataset** represents any collection of RDF data. Our class `SparqlDataset` represents datasets that can be queried with SPARQL queries. The class offers a single method `query` that can be used to pose SPARQL queries against the dataset. Throughout the code, we only use this method for data access. The subclasses `SparqlEndpointDataset` and `InMemoryDataset` encapsulate access to remote datasets which are identified via an endpoint URI, and local datasets which are loaded into memory, respectively.

```

1 public abstract class SparqlDataset extends Dataset {
2     public abstract QueryExecution query(Query query);
3 }

```

Listing 4.1: `SparqlDataset.java`

A **statistical criterion** represents a single analytic measure. The abstract class `StatisticalCriterion` provides three public methods. The first, `flushLog`, prints the current state of measurements. The second, `considerTriple`, represents the `FILTERTRIPLE(s, p, o)` function (as seen in section 4.1), implementing the Triple Stream Approach. The third public method is `processSparqlDataset` which implements the Specific Query Approach. It uses the `query` method of the provided `SparqlDataset`, and delegates parsing of query results to its concrete subclasses. The specific SPARQL queries for each criterion are loaded from separate `.sparql`-files. This process is implemented in `StatisticalCriterion`, but has been omitted for brevity.

```

1 public abstract class StatisticalCriterion {
2     // print current measurements
3     public abstract void flushLog(PrintStream ps);
4
5     // filter triple (TSA)
6     public abstract
7     void considerTriple(Resource s, Property p, RDFNode o);
8
9     // process query results (SQA)
10    abstract void processQueryResults(ResultSet res);
11
12    // query dataset (SQA)
13    public void processSparqlDataset(SparqlDataset ds) {
14        ...
15        // execute query and obtain results
16        QueryExecution qe = ds.query(query);
17        ResultSet results = qe.execSelect();
18
19        // delegate result processing to subclass
20        processQueryResults(results);
21        ...
22    }
23 }

```

Listing 4.2: `StatisticalCriterion.java` (some parts omitted)

An **analyzer** represents a single analysis configuration. The `AbstractAnalyzer` class defines four public methods. The first two, `addCriterion` and `setDs`, are used to configure the analysis with regard to dataset and criteria. The third method, `performAnalysis`, actually performs the analysis and prints the results to the provided `PrintStream`. The fourth method is `equals`. `AbstractAnalyzer` and all subclasses of `StatisticalCriterion` re-implement this standard method, making analysis results comparable by calling `analyzer1.equals(analyzer2)`. The `performAnalysis` method is implemented differently for SQA and TSA. The subclass `SparqlAnalyzer` (SQA) calls `processSparqlDataset` for each criterion, while `TripleStreamAnalyzer` (TSA) creates a triple stream and calls `considerTriple` for each criterion and triple. The triple stream is created by subsequently querying the dataset with increasing `OFFSET` parameters (section 4.1.2). Random Sampling is implemented by calling `Collections.shuffle(offsets)` on a precomputed list of `offsets`.

```
1 public abstract class AbstractAnalyzer implements Analyzer {
2     // define criteria
3     public Analyzer addCriterion(StatisticalCriterion sc) { ... }
4
5     // define dataset
6     public Analyzer setDs(SparqlDataset ds) { ... }
7
8     // perform analysis and output results
9     public abstract void performAnalysis(PrintStream ps);
10
11    // compare analysis results
12    @Override
13    public boolean equals(Object obj) { ... }
14 }
```

Listing 4.3: `AbstractAnalyzer.java` (some parts omitted)

The classes in the `eval` package have been implemented for the evaluation of our work (chapter 5), and do not represent core components of `RDFSynopsis`. Hence, they are not further discussed.

The following examples demonstrate the use of RDFSynopsis's classes. We analyze the *number of triples* of a SPARQL endpoint using TSA (Example 1), and find the *common properties* of an in-memory dataset using SQA (Example 2).

```
1 TripleStreamAnalyzer tsa = new TripleStreamAnalyzer()
2   .addCriterion(new NumTriples())
3   .setDs(new SparqlEndpointDataset("http://..."))
4   .setRandomSampling(true)
5   .setTripleLimit(500);
6 tsa.performAnalysis();
```

Listing 4.4: Usage of RDFSynopsis Framework (Example 1)

```
1 SparqlAnalyzer sqa = new SparqlAnalyzer()
2   .addCriterion(new CommonProperties())
3   .setDs(new InMemoryDataset("file:../data/peel.rdf"));
4 sqa.performAnalysis();
```

Listing 4.5: Usage of RDFSynopsis Framework (Example 2)

RDFSynopsis also provides a command-line interface. The above examples can be invoked from the command-line. The `-c` parameter is used followed by a list of numbers to define the desired criteria; `-lc` prints the number-to-criteria mapping. Section C presents the command-line usage of RDFSynopsis in greater detail.

```
1 // Example 1
2 java -jar rdfSynopsis.jar -tsa -rand -tl 500 -c 16 -ep http...
3 // Example 2
4 java -jar rdfSynopsis.jar -sqa -c 18 -f ../data/peel.rdf
```

Listing 4.6: Usage of RDFSynopsis (Command-Line)

5. Evaluation

In this chapter we evaluate our approaches to SPARQL-based analyses of RDF datasets.

First, we demonstrate our analytical measures along an example use case in section 5.1. Then, we compare the SQA and TSA approaches with regard to their query runtime performance (section 5.2). Finally, the chapter concludes with a look into the accuracy achieved with partial analyses of datasets (section 5.3).

5.1. Example Use Case Study

In this section, we illustrate how to employ the analytic measures we have chosen to investigate an RDF dataset. The structure of this section reflects the natural course of dataset exploration; for the measures presented in section 4.1.3, we explain the insight that is gained by its results.

We imagine a *Web developer* who wants to create a Website about winter sports, based on the Web of Data. That person would go hunting for appropriate RDF datasets that could be merged to form the Website's data base. In the course of this search, this person might stumble upon a SPARQL endpoint to the Austrian Ski Team dataset (AST) ¹. Although, one could guess from its name that the dataset behind that endpoint belongs to the domain of winter sports, one would not know

- what kind of information was actually contained (Entity and Information Domain),
- how that data was structured in RDF (Composition), and
- how easy this data could be merged and reused with other datasets (Reusability).

Before any kind of investigation starts, it is good to know how large a dataset one is dealing with. The AST dataset has 5874 triples; it is a fairly small dataset. (*triples*) We rely on subject type information for both, investigation and use of RDF datasets.

¹<http://vocabulary.semantic-web.at/PoolParty/sparql/AustrianSkiTeam>

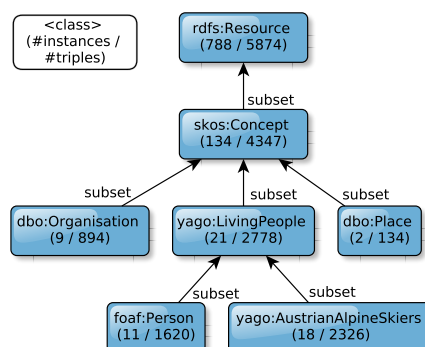


Figure 5.1.: Implicit Class Hierarchy with Instance and Triple Counts (some parts omitted)

Fortunately, we have a *typed-subject-ratio* of 99%; nearly all subjects have another type besides `rdfs:Resource`. According to the *ontology-ratio*, 93% of instances are terms. This is a significant percentage; the dataset seems to mainly represent a collection of term definitions, and contains only a small amount (7%) representations of real world entities.

In order to understand what kind of real world entities are represented in the dataset, we look at the number of instances per class. (*class usage count*) (see `#instances` in figure 5.1) Firstly, the four most frequently instantiated classes are `rdfs:Resource`, `skos:Concept`, `rdfs:Class`, and `rdf:Property` (partly omitted in figure). Secondly, and more valuable for our investigation, the dataset contains, among others, 21 instances of type `yago:LivingPeople` and 18 `yago:AustrianAlpineSkiers`. To understand if these entities, despite their little number, are “at the center” of the dataset, we study the distribution of *triples per subject class*. The results (see `#triples` in figure 5.1) reveal, that the dataset mainly contains data about living people and Austrian alpine skiers; AST could indeed prove useful for the winter sports Website.

But, besides the trivial results for `rdfs:Resource`, the most frequent subject class for triples is `skos:Concept`; 4347 of the total 5874 triples have a subject that is a `skos:Concept`. *Counterintuitively*, with 2778 triples for `yago:LivingPeople`, it follows that the two sets of concepts and living people cannot be disjoint in AST. At this point in the investigation, it is not clear how information is modeled in AST. For two sets of class instances not to be disjoint, a usual reason is the existence of hierarchical relationships between classes, such that all instances of one class are also instances of the other class. AST contains no *explicit class hierarchies* (in the form of `rdfs:subClassOf` relationships) except the standard subclass statements of RDF and RDFS. In contrast, investigating actual class instantiation in the dataset reveals *implicit class hierarchies* (as visualized in figure 5.1) For example, all instances of `yago:LivingPeople` are also instances of `skos:Concept`. Furthermore, `skos:Concept` seems to be used in a very general way for instances of classes from

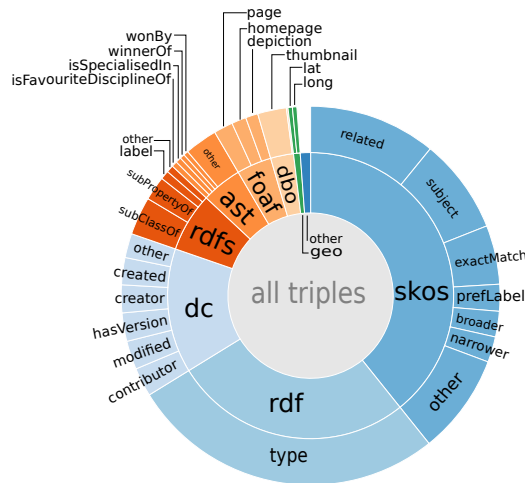


Figure 5.2.: Property Usage (inner circle shows property vocabularies, outer circle shows specific properties)

`foaf:Person` to `dbo:Place`.¹ The class `foaf:Person` seems to be used either heterogeneously or counterintuitively, because, contrary to real-world semantics, not all instances of `yago:LivingPeople` are also of type `foaf:Person`. We now know that our first impression - AST mainly representing an ontology - was wrong. In AST, all real world entities (living people, organisations, places) are also modeled as `skos:Concepts`.

After understanding what subject classes are at the center of the dataset, the next step is to investigate what information the dataset contains about those subjects, i.e., what properties are used. For a first overview we take a look at *predicate vocabularies* used in AST (see inner circle of figure 5.2). Besides standard (`rdf`, `rdfs`), term-related (`skos`) and metadata vocabularies (`dc`) which amount to more than 3/4 of triples, the dataset’s “payload” probably lies within properties from its own (`ast`) and the `foaf` (descriptions of people), `dbo` (dbpedia terms) and `geo` (geographic coordinates) vocabularies. When we take a look at the most frequently used properties (*property usage*) from these vocabularies (see outer circle in figure 5.2), it can be seen that type information accounts for 27% of triples, followed by several `skos`: properties that relate terms with each other. Thumbnail and web page URLs, and geographical coordinates are also among the most frequently used properties. The most frequent property from the dataset’s `ast`: namespace is `ast:isFavouriteDisciplineOf`. Until now, it remains unclear what kind of instances are described with the different properties.

The results (as visualized in figure 5.3) show the number of *class instances per property*. We see that, for example, 21 instances of `yago:LivingPeople` have a

¹Although it could be argued that this use of `skos:Concept` does not conform to the intentions of the SKOS standard, it is not *wrong* from a technical point of view.

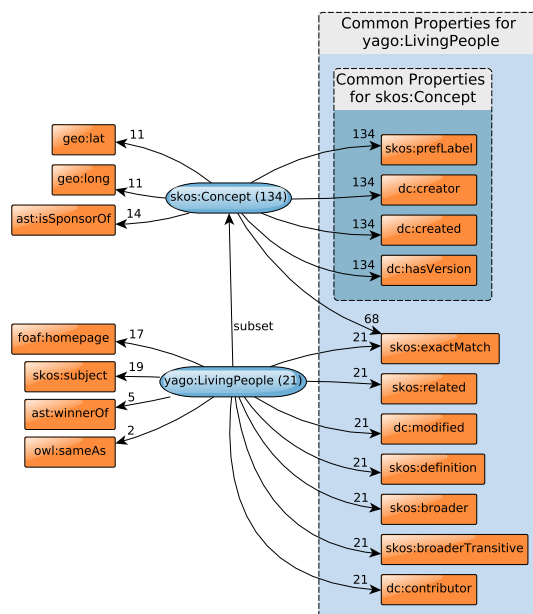


Figure 5.3.: Number of Class Instances per Property with Common Properties

`skos:prefLabel`, while 17 have a `foaf:homepage`. The set of *common properties* for `yago:LivingPeople` contains eleven properties; all instances have a triple for each of those eleven properties. Super classes (like `skos:Concept`) have a subset of those common properties. Looking at the *explicit property hierarchy*, the dataset defines all of its own properties (e.g., `ast:winnerOf`) to be `rdfs:subPropertyOf skos:related`, i.e., a `skos:related` triple will automatically be inferred from any of the `ast:` property triples.

Overall, a look at class and property hierarchies reveals that all locally defined real world entities (like people and places), and relationships (like `ast:winnerOf`) are actually modeled in general `skos` terms. All entities are instances of `skos:Concept`, and all `ast:` properties are `rdfs:subPropertyOf skos:related`. This use of `skos` may be questionable, since the vocabulary is meant to express thesauri-like term relations. The dataset is probably easy to reuse for the envisioned winter sports website, since nearly all subjects are typed (*typed-subject-ratio*), no *isolated blanks* are contained, and external namespaces like `dbpedia`, `freebase`, and `geonames` are frequently referenced. (*namespace links* in figure 5.4) In addition, meta-data (like `dc:modified`) and the `skos:exactMatch` property - expressing term equivalence - are *common properties* of all living people represented in AST.

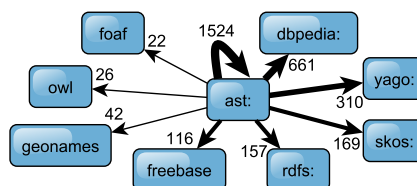


Figure 5.4.: Links to Other Namespaces

5.2. Performance

In this section we look into the performance of our SPARQL-based approaches to a structural analysis of RDF datasets.

Firstly, we describe our benchmark setup (section 5.2.1), and explain why we analyze performance in terms of *Query Runtime*. Secondly, we study both our approaches, SQA (section 5.2.2) and TSA (section 5.2.3) separately, before we compare them in section 5.2.4.

5.2.1. Setup

One of the main criteria to decide between our two SPARQL-based analysis approaches, SQA and TSA, is the computational performance. In this section we describe our performance evaluation approach and setup.

In general, the performance of an algorithm can be measured in terms of two computational resources, namely RAM usage and Central Processing Unit (CPU) runtime. In our case, we could measure these resources usages for the local (RDFSynopsis) and the remote side (SPARQL endpoint). For the performance evaluation presented in this thesis, we focus solely on runtime. The reason for this decision lies in our observation, that most SPARQL endpoints found in the web constrain query processing by specifying a *timeout*, i.e., a maximum time for query processing after which the query is canceled. To enable a SPARQL-based analysis all queries posed need to stay below these timeouts. We further restrict ourselves to the remote side and do not measure processing time for RDFSynopsis, because processing of the highly aggregated results for the Specific Query Approach is obviously not very complex. The situation for TSA looks different at first, because RDFSynopsis has to deal with quite large amounts of triples that are the results of triple stream queries. We can, however, understand our remote runtime results for SQA as an upper-bound for our local runtime results for TSA, because SQA can always be emulated on the local side by aggregating all streamed triples into one RDF graph and then running the SQA queries on that graph. In conclusion, we only measure the wall time¹ it takes until the endpoint has answered a query, including transmission of all results. We call this time the *query runtime*.

We measured query runtimes using the open-source “SPARQL Query Benchmark” [72], developed by YarcData (a division of Cray Inc.), and available under a Berkeley

¹We measure *wall time*, not *CPU time*, because we expect timeouts on the endpoint side will also be measured as wall time.

Software Distribution (BSD) 2.0 License. The software is based on Apache Jena. It can be used to repeatedly execute a set of SPARQL queries against a given endpoint, and collect runtime statistics. For our TSA measurements, we generated multiple sets of triple stream queries; one set of queries for each combination of dataset, triple limit, and order variable. The statistics we collected about query runtimes include the variance and standard deviation of query runtimes. These values are important to judge the significance of our results. Although we do not explicitly present the standard deviation for each measurement in the following sections, we only show results with a maximum standard deviation of 10% of the average absolute value.

We use three real-world datasets for the evaluation of RDFSynopsis. We refer to these as “dblp”, “Peel”, and “AM”. Table 5.1 contains a brief description.

Dataset	Description	
dblp	Filename	<code>dblp-publications-2012.rdf</code>
	Size	51 MB
	Triples	441058
	Content	Publication data for the year 2012 from the DBLP Computer Science Bibliography.
	Available from	<code>dblp.rkbexplorer.com</code> ¹
Peel	Filename	<code>peel.rdf</code>
	Size	22 MB
	Triples	271369
	Content	Information about the “Peel Sessions”, a regular BBC radio programme by John Peel.
	Available from	<code>dbtune.org/bbc/peel</code> / ²
AM	Filename	<code>am-data.ttl</code>
	Size	240 MB
	Triples	5700371
	Content	Collection data from the Amsterdam Museum, see [20].
	Available from	<code>semanticweb.cs.vu.nl/lod/am/data</code> ³

Table 5.1.: Datasets Used in Benchmark

¹<http://dblp.rkbexplorer.com/models/dblp-publications-2012.rdf>

²<http://moustaki.org/resources/peel.tar.gz>

³<http://eculture.cs.vu.nl/git/public/?p=econnect/metadata/AHM.git;a=blob;f=rdf/am-data.ttl>

For our evaluation, we set up a local SPARQL endpoint using Fuseki [42]. Fuseki is part of the Apache Jena Project [6] but can be downloaded as a stand-alone application. Fuseki can be used to load an RDF dataset and expose it as a SPARQL endpoint. We configured Fuseki to load our three example datasets to memory, and expose each with a separate endpoint. We disabled the inference engine, and assigned 8GB of RAM to Fuseki. See appendix section B for configuration details.

The technical system specifications for our evaluation setup are presented in table 5.2. SPARQL endpoints used in practice will probably be dedicated servers with better processing capabilities. Therefore, the absolute query times we measure for our local Fuseki endpoint are not representative for the performance seen in professional real world setups. Still, we can use the results to compare our different approaches.

System	Specification
Operating System (OS)	Windows 7 Professional SP1
CPU	Intel Core i5-2500K, 4x 3.30GHz
RAM	total 16 GB
	assigned to Fuseki 8 GB
	assigned to Benchmarker 3 GB

Table 5.2.: System Specifications for Benchmark

5.2.2. Specific Query Approach

Table 5.3 shows the runtime performance for the specific queries of all our structural measures. These runtimes represent the arithmetic average of 100 runs against the local fuseki instance (as explained in section 5.2.1). Each measure is assigned a unique color, that can be used to identify them in the following figures. All subsequent figures also adhere to the order of measures presented in table 5.3.


















Query	dblp	Peel	AM
 ClassHierarchy	0.008	0.008	0.008
 ClassInstancesPerProperty	0.549	0.363	6.760
 ClassUsageCount	0.107	0.086	0.786
 CommonPropertiesNew	0.628	0.408	7.563
 DistinctSubjectOnlyBlanks	0.188	0.112	5.474
 DistinctSubjectsBlank	0.190	0.112	2.823
 ImplicitClassHierarchy	0.212	0.141	1.782
 NamespaceLinks	5.129	2.583	52.049
 NumTriples	0.197	0.119	2.325
 OntologyRatioNew	1.111	0.716	9.312
 PredicateVocabularies	2.539	1.324	28.802
 PropertyHierarchy	0.008	0.008	0.008
 PropertyUsage	0.293	0.166	3.270
 PropertyUsagePerSubjectClass	1.454	0.875	21.004
 SameAs	0.029	0.009	0.008
 TriplesPerSubjectClass	0.364	0.245	4.042
 TypedSubjectRatio	3.275	1.777	43.846
Total	17.072	9.458	192.668

Table 5.3.: Benchmark Results for SQA
(in seconds, arithmetic average for 100 runs)

The results from table 5.3 are visualized in figure 5.5. Firstly, we can see that the dataset’s size clearly affects the query runtimes. The results for the biggest dataset (AM) are significantly higher than for smaller datasets (dblp, Peel). The average total runtime is more than three minutes for AM, and not even ten seconds for Peel. Secondly, we can also observe the same four measures are responsible for the longest running times in all three datasets. These measures are

1. namespace links
2. typed-subject-ratio
3. predicate vocabularies
4. property usage per subject class

These queries all operate on the full dataset; we can suspect that the SPARQL engine is not able to optimize query execution by omitting large amounts of triples. In addition, the runtimes of both, namespace links and predicate vocabularies, can be explained by their use of relatively expensive string operations based on regular expressions.

To analyze how the total runtime is distributed over the single queries, we express the query runtimes in percent of total runtime (see figure 5.6). We discover that the runtime distribution remains fairly stable among different datasets whose total runtimes differ significantly. That means, if the total runtime for a dataset grows, all single query runtimes grow proportionately.

To determine whether runtime is linearly dependent on the dataset's size (number of triples), we divide the results for each dataset by its number of triples. For better readability, we multiply all results by one million.

$$\text{runtime}_D^{1M} = \text{runtime}_D^{\text{Total}} * \frac{1000000}{\text{numTriples}_D}$$

Hence, figure 5.7 shows the runtimes normalized to one million triples. The optimal case would have been *constant* query times regardless of a dataset's size. In this case, there would have been a very significant difference between the biggest dataset (AM)

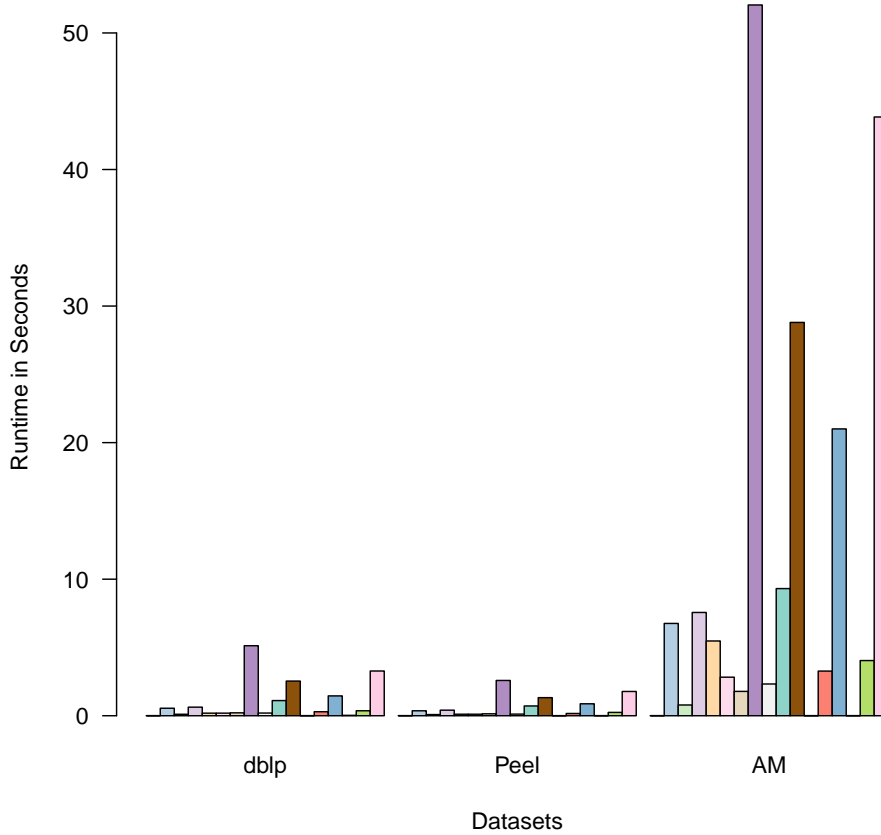


Figure 5.5.: Total Runtime (SQA) Grouped By Dataset

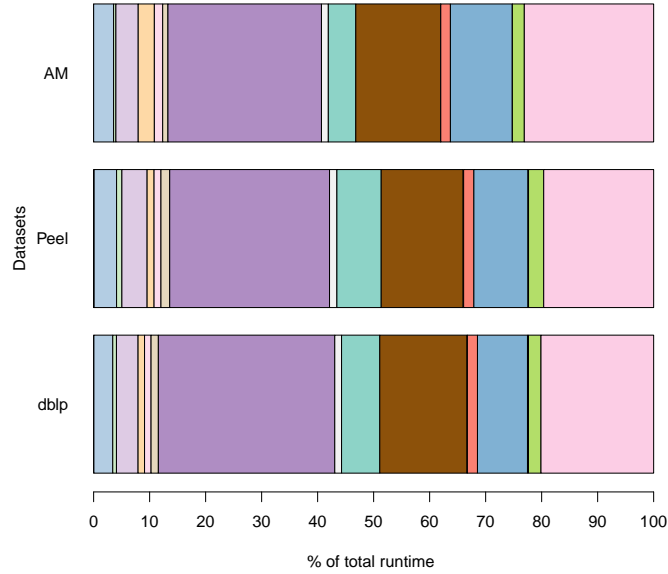


Figure 5.6.: Runtime (SQA) in Percent of Total Runtime

and the smallest dataset (Peel). In reality we find that, relative to their number of triples, AM and Peel show similar performance, while queries on the dblp dataset have taken slightly longer. Fortunately, we do not experience above-linear runtime growth.

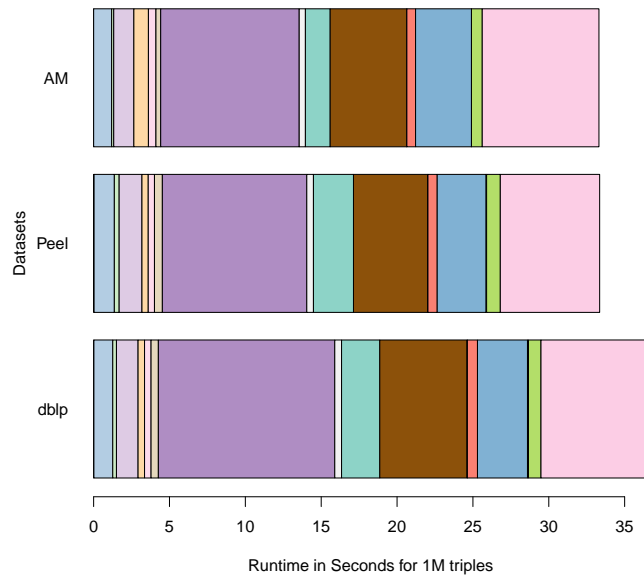


Figure 5.7.: Runtime (SQA) Normalized to 1M Triples

Summary. The results reveal that

- The same four measures (see above) are the most complex for all three datasets.
- The runtime linearly depends on the number of triples.
- The runtime distribution over specific queries is fairly stable for different datasets.

5.2.3. Triple Stream Approach

Table 5.4 shows query runtimes for the Triple Stream Approach. As we have seen in section 4.1.2, the triple stream query is parameterized with the number of requested triples (LIMIT keyword) and the order variable that defines the sequence (ORDER BY keyword).

LIMIT	ORDER BY	dblp		Peel		AM	
		Single	Full	Single	Full	Single	Full
10.000	subject	1.77	79.59	0.99	27.81	23.77	13570.49
	predicate	5.44	244.69	2.99	83.62	78.16	44627.40
	object	3.54	159.14	6.03	168.90	54.06	30869.26
50.000	subject	2.05	18.44	1.30	7.79	23.91	2749.80
	predicate	5.69	51.23	3.37	20.21	78.40	9016.22
	object	3.86	34.78	6.34	38.02	54.07	6217.73
200.000	subject	2.78	8.33	2.05	4.10	25.09	727.68
	predicate	6.49	19.46	4.09	8.17	79.62	2308.98
	object	4.57	13.72	7.12	14.23	55.32	1604.18

Table 5.4.: Benchmark Results for TSA
(in seconds, arithmetic average for 10 full runs)

The table shows results for three different triple limits (10k, 50k, and 200k) and all orders (by subject, predicate, and object). For each combination, we present the runtimes for a single query as well as the full runtime that is needed to completely stream the dataset. The runtimes are the average of 10 full runs with randomized sampling. The results for the sequential (non-random) approach have been omitted because they were almost identical.

The runtimes for single triple stream queries are visualized in figure 5.8 (note the log scale on the y-axis). The different datasets are differentiated by color and line-style; the orders are identified by their point-style. We observe that the lowest runtimes are obtained with a by-subject order for all datasets and all triple limits. We further find that the impact of triple limit on single query runtimes is low; the difference between 10k and 200k queries seems to be about 1 seconds for all cases. This has the consequence that full runtimes are inverse proportional to the triple limit, because the chosen limit directly implies the number of queries necessary for a full analysis, and all queries approximately take the same time.

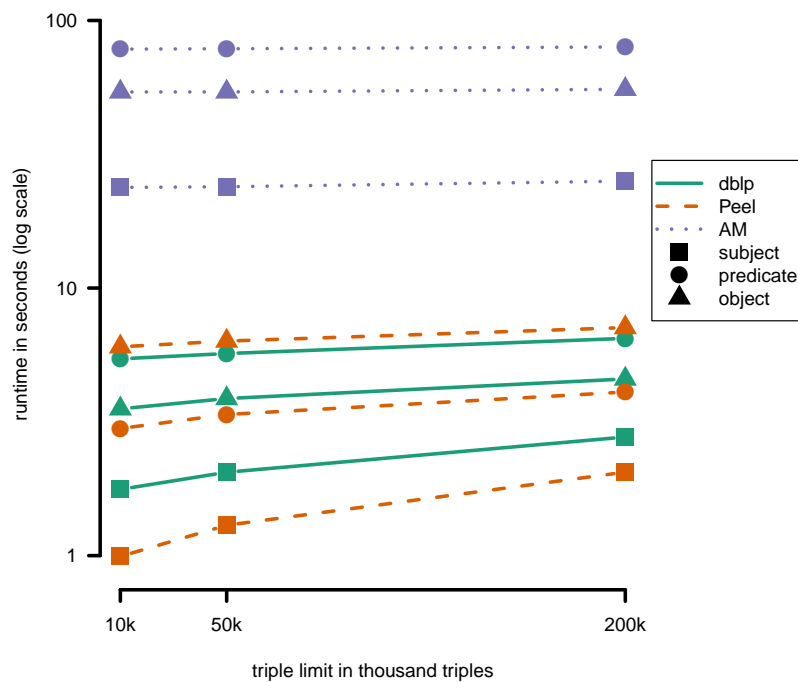


Figure 5.8.: Runtime of Single TSA Queries

Figure 5.9 shows the single query runtimes, normalized to one million triples (as in the previous section). We observe that the main factor for single query runtimes is the total size of the underlying dataset, and not the chosen triple limit.

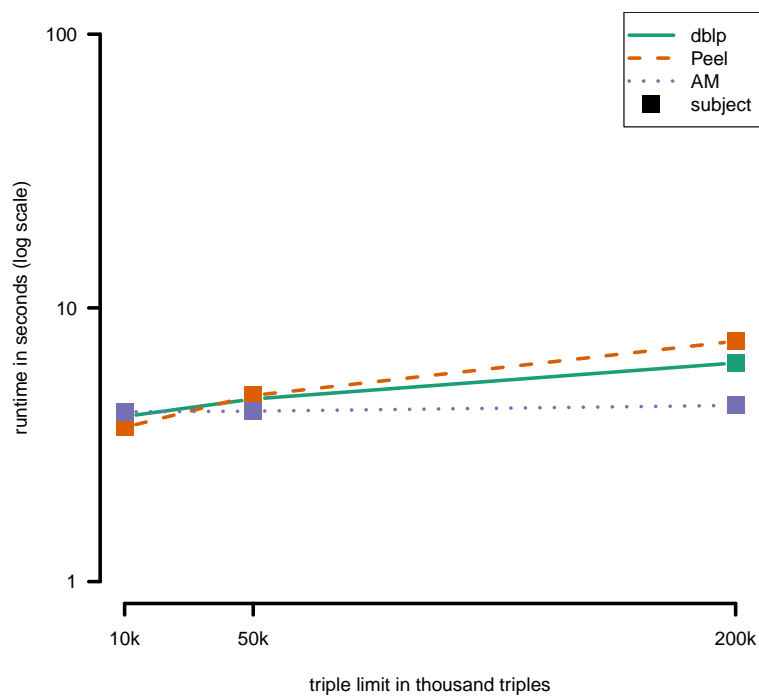


Figure 5.9.: Runtime of Single TSA Queries (Normalized to 1M Triples)

We present the results for full runtimes (by-subject order) in figure 5.10. For better comparison, we have again normalized these runtimes to one million triples. Supporting our previous argumentation - full runtimes being inverse proportional to the chosen triple limit -, we can see that an increase of the triple limit by a factor of five (from 10k to 50k) leads to a full runtime that is decreased to a fifth of the original value, approximately. Still, the runtimes for the three datasets differ significantly. This circumstance is explained with our observation that the runtime of a single query mostly depends on the total size of the queried dataset.

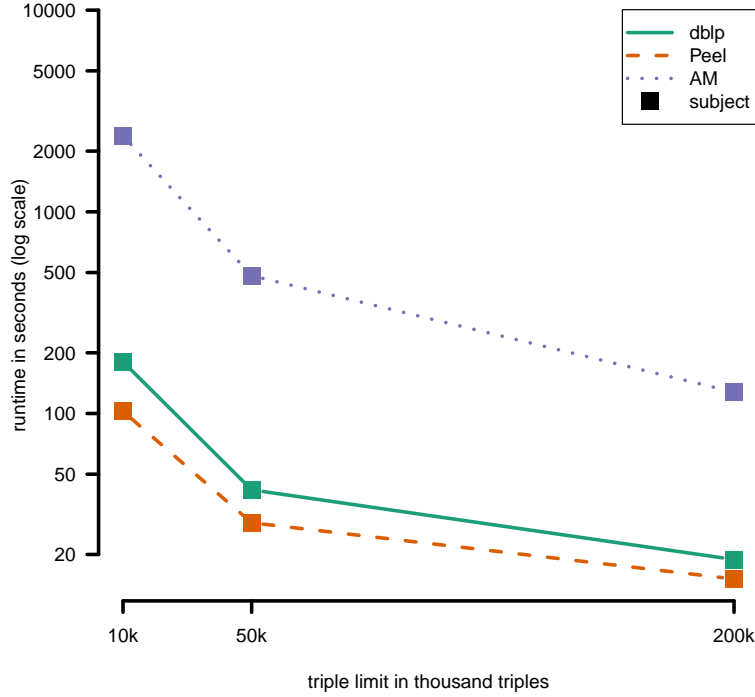


Figure 5.10.: Runtime of Full TSA Queries (Normalized to 1M Triples)

Summary. The results for TSA reveal that

- Triple streams ordered by subject have consistently yielded the lowest runtimes.
- The choice of triple limit has little effect on the runtime of a single query.
- The runtime of a single query linearly depends on the size of the queried dataset.
- Full runtimes are inverse proportional to the chosen triple limit.

5.2.4. Comparison of Approaches

In this section we compare our approaches to SPARQL-based analyses of RDF datasets with regard to their query runtime performance.

Table 5.5 presents the full runtimes using SQA and TSA (with different triple limits). Although TSA performs better for combinations of high triple limits and smaller datasets, we find SQA to be superior when scaling up in terms of dataset size. For example, TSA_{200k} runs 3.7 times longer for the AM Dataset than SQA.

This can be explained with our observations from the benchmarks. We found that the full runtime of a set of specific SPARQL queries grows linearly with the dataset size. The same is true for single triple stream queries, while the full runtime for TSA grows above-linearly, because for growing datasets the single query has to be run increasingly often.

Unfortunately, a single triple stream query that simply requests a “chunk” from a sequence of triples, seems to be very computationally expensive for the SPARQL endpoint under test. As long as no cheaper means of triple streaming are at hand, the Triple Stream Approach remains infeasible with regard to efficient computation.

	dblp	Peel	AM
SQA	17.07	9.46	192.67
TSA _{10k}	79.59	27.81	13570.49
TSA _{50k}	18.44	7.79	2749.80
TSA _{200k}	8.33	4.10	727.68

Table 5.5.: Total Runtimes for SQA and TSA (ordered by subject)

5.3. Accuracy of Partial Analysis & Random Sampling

In this section we look into the accuracy of partial analyses following the TSA and Random Sampling approaches. Conducting an analysis only on parts of a dataset implies the act of estimating results for the full dataset based on results for the partial dataset. Because we have no knowledge about the distribution of values and structures within the dataset, we assume a uniform distribution, or in other words, that a part is representative for the whole. We therefore use measurements that are relative to the number of analyzed triples, instead of absolute values; as in the following trivial formula:

$$measurement_{rel} = \frac{measurement_{abs}}{triples_{seen}}$$

Our analyses of RDF datasets comprise a set of 18 measures. Many of these do not represent single values, but histogram-style measurements. It is not trivial to establish a single accuracy measure for this set of measures. One could argue that two measures have different levels of importance for the overall accuracy of a partial analysis. Because of the complexity of weighing and combining the different measures towards an overall accuracy assessment, we deem a quantitative analysis of this topic to be out of scope of this thesis.

Instead, we try to give an intuitive insight into the problem and the accuracy that is obtained with partial analyses.

Figure 5.11 shows how the distribution of the *property usage* measure for the dblp dataset develops while increasing amounts of triples are analyzed. The same combinations of triple limits (10k, 50k, and 200k) and order variables (subject, predicate, and object) are presented, as in the previous section on TSA runtime performance. All analyses depicted in the figure were made following the sequential approach. Each color represents one property, and the vertical space occupied by the color represents the property’s number of occurrences relative to the number of analyzed triples. The color distribution at the right end represents the property distribution relative to all triples, i.e. the “true” property distribution of the dataset. Ideally the chart would show only horizontal lines, meaning a perfect value distribution from the beginning.

We observe that the different **ORDER BY** parameters affect how the distribution of properties over the analyzed triples develops. For example, the triple streams that are sorted by predicate and object show a more turbulent development than the triple stream that was sorted by subject. The latter shows a very representative distribution from the beginning, except for the fact that the property represented by the lavender color (above yellow) only occurs after more than 50% of all triples have been streamed.

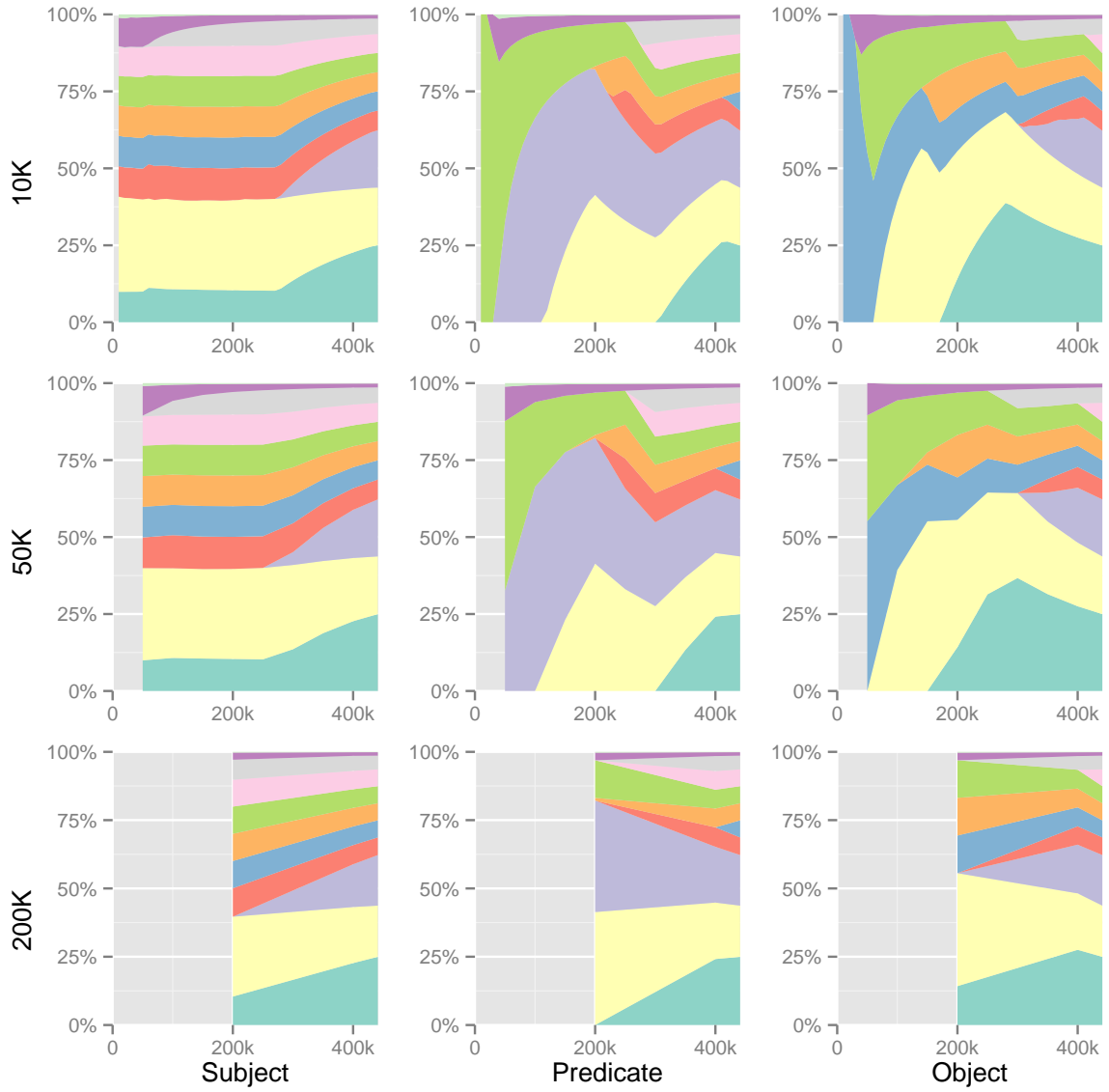


Figure 5.11.: Accuracy of Sequential TSA with different triple limits and order by clauses (*dblp*, *property usage*)

We also see that different triple limits generate the same distribution developments, while higher limits make the developments look more smooth.

We now look into the Random Sampling approach. We choose a triple limit of 10k and sort the dataset by subject, because these parameters have performed best in the sequential approach. Figure 5.12 presents nine random runs with these parameters. We observe that the randomization has helped further smoothen the distribution developments. In all runs a representative property usage distribution is reached after 10 to 20 random triple stream queries.

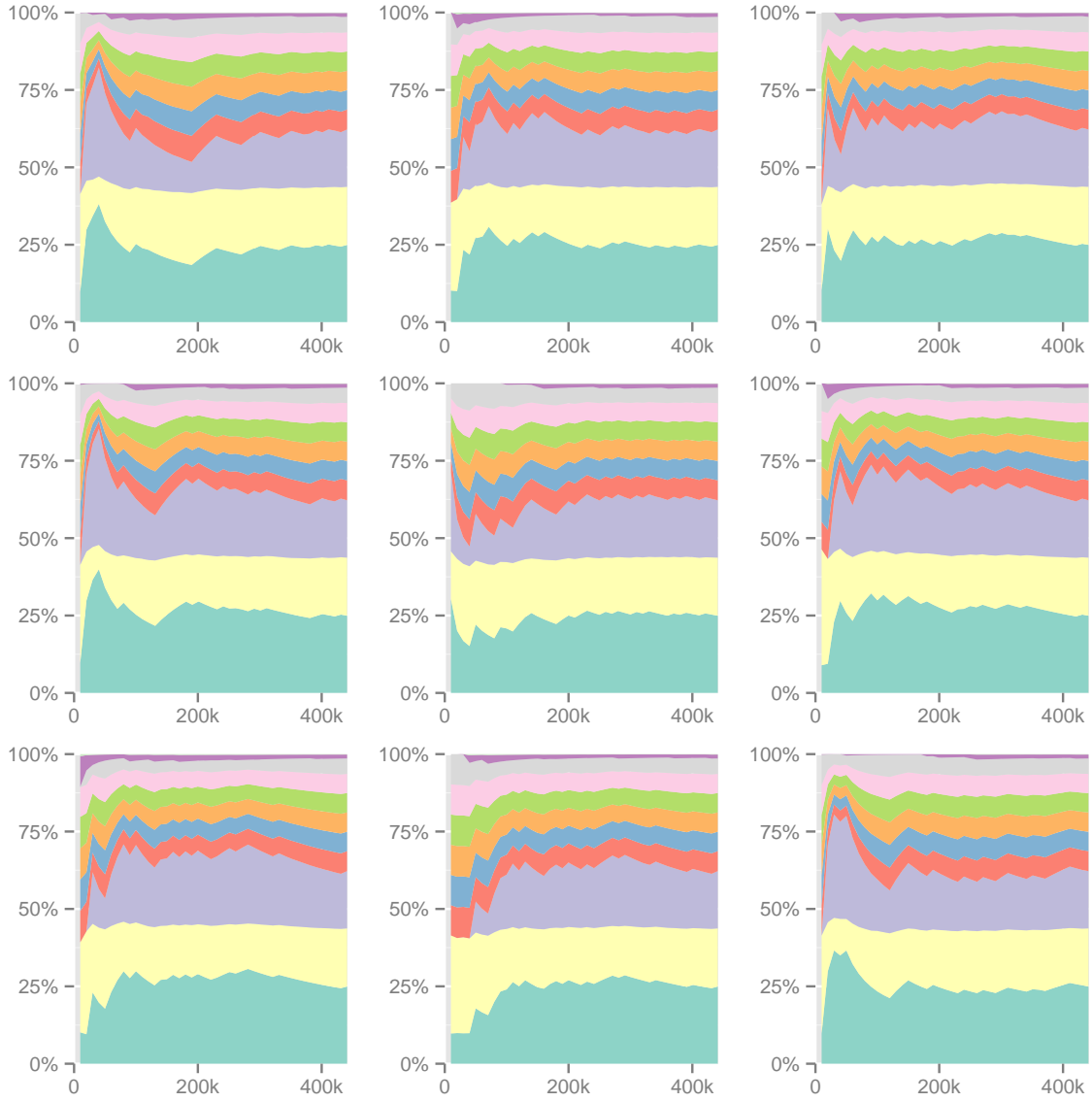


Figure 5.12.: Accuracy of Random Sampling TSA for 9 runs
(dblp, *property usage*, LIMIT=10k, ORDER BY=subject)

In this section we have seen that the ORDER BY variable affects how early (w.r.t. number of analyzed triples) a partial analysis reaches a value distribution that is representative for the full dataset. We have also observed that randomization of triple streaming can improve the accuracy of a partial analysis.

As we have pointed out above, our observations are only based on intuition. The findings for our example dataset and the property usage measure do not necessarily have to be true for other datasets or other measures.

6. Conclusion

In this final chapter, we first provide a summary (section 6.1) and an evaluation (section 6.2) of the work presented in this thesis. We conclude with an outlook on potential future work (section 6.3).

6.1. Summary

In this thesis, we investigate SPARQL-based approaches to a structural analysis of RDF datasets.

Our work is clearly located in the context of the Semantic Web idea, which aims to enable a machine-processable Web of Data. RDF, a generic triple-based data model, and SPARQL, a query language for RDF, are among the major technological building blocks of this Web of Data. Due to the bottom-up characteristic of RDF data modeling and the resulting lack of explicit schema information, gaining an understanding of a previously unknown RDF dataset can easily become a time-consuming hands-on process. Furthermore, RDF data is often only available remotely through a server that answers SPARQL queries, so-called SPARQL endpoints; hence the need for automated structural analyses based on SPARQL queries. The Semantic Web idea and its core technologies are presented in chapter 2 (*Background*).

In chapter 3 (*Structural Analysis of RDF Datasets*) we develop the notion of a structural analysis of RDF datasets. We characterize it as a process which sheds light on four central aspects of RDF dataset understanding: Entity Domain, Information Domain, Composition, and Reusability. We study related work from three areas, and compile a list of 75 statistical criteria (table 3.1) which are used for RDF data in scientific work and practical applications. Based on our findings from related work, we separately discuss each central aspect of RDF dataset understanding, and identify suitable criteria, 18 in total.

In chapter 4 (*Design and Implementation of Structural Analysis*) we identify two general approaches to analyze an RDF dataset with SPARQL queries. In the first *Specific Query Approach* (SQA), we formulate a specific query for each analytic criterion. The

second approach is called *Triple Stream Approach* (TSA), and subsequently retrieves all triples from a dataset. A criterion is implemented in terms of a triple filter, that only selects relevant triples. We also discuss the idea to only stream a random subset of triples (Random Sampling). We present SQA and TSA implementations for all 18 criteria. Finally, we describe the architecture of our software RDFSynopsis which implements all criteria with both approaches. The software can be used from the command-line, or as an analysis framework within other software projects.

In chapter 5 (*Evaluation*) we evaluate our approaches with regard to three aspects. First, we employ our analytic criteria in an example use case, to demonstrate their suitability for structural analyses. Second, we compare SQA and TSA with regard to their query runtime performance. Third, we study the accuracy obtained with partial and random TSA analyses. Due to the complexity of accuracy measurements, we limit the presentation to an intuitive example.

6.2. Results

In this section, we evaluate our work with respect to its goal, “to enable an automated structural analysis of RDF data based on SPARQL queries”. For the evaluation, we follow the three steps introduced in section 1.3.

Identification. The goal of the first step was to identify the set of measures that are needed to gain an understanding of an RDF dataset. Based on the four aspects of RDF dataset understanding and our study of related work, we identified a set of 18 criteria for structural analyses of RDF datasets (table 3.2). 7 of these are new criteria that have not been found in other work, including sophisticated criteria like *common properties* that look for homogeneous structures, explicitly taking the bottom-up nature of RDF into account. The result of the first step was a **choice of 18 measures for structural analysis**.

Implementation. The goal of the second step was to implement the analysis according to the set of identified measures with SPARQL. In the course of our work, we identified two approaches to analyze an RDF dataset with SPARQL queries, the Specific Query Approach (SQA) and the Triple Stream Approach (TSA). For the latter, we discussed a variation which creates a triple stream from random dataset samples. We then presented self-contained SQA and TSA implementations for all 18 chosen criteria. Finally, we implemented all criteria with both approaches in our software RDFSynopsis, which can be used from the command-line, or as an analysis framework within other software projects. The result of the second step was a **catalog of RDF analysis criteria** with implementations for **two different SPARQL-based approaches**. The overall concept was proven by our **software RDFSynopsis**.

Evaluation. The goal of the third step was to evaluate whether we had reached the goal of this thesis. Along an example use case, we demonstrated that our chosen measures provide insight into an unknown dataset’s structure. This is particularly true for combinations of measures, e.g., the combined view of *implicit class hierarchy* and *triples per subject class*. The use case also showed how important result *visualization* is. We investigated the performance of SQA and TSA in terms of query runtime. We observed that the runtimes for all queries grow linearly with the dataset’s size. Unfortunately, this was also true for the triple stream queries, independent of the chosen triple limit. For SQA, we found big differences in the runtimes of the different measures. The distribution of total runtime over the specific queries remained fairly stable among different example datasets. For TSA, we found that the `ORDER BY` parameter significantly affects query runtimes. We achieved the best results with triple streams ordered by subject. As noted above, triple stream query runtimes grew with dataset size, not with triple limit. This result is unfortunate, because the random sampling approach relies on the ability to quickly receive many small chunks of a dataset. In conclusion, TSA is clearly outperformed by SQA. Our analysis of the accuracy achieved with a randomly sampled triple stream revealed no absolute results. Instead we gave a rather intuitive insight into the accuracy improvement of random over sequential TSA. We must note, that all performance measurements have been obtained using a local fuseki instance, the SPARQL endpoint from the Apache Jena project. They are not necessarily representative for other SPARQL endpoints. The third step has revealed that our **analytic measures are suited for structural analyses**, and that **SQA performs clearly better than TSA**.

Overall, we have demonstrated that RDF datasets can be structurally analyzed using SPARQL. Within our limited benchmark setup, the approach to formulate a specific query for each analytic criterion has outperformed the SPARQL-based triple stream approach.

6.3. Future Work

In this section we look into potential future work relating to the structural analysis of RDF datasets and the work presented in this thesis.

Result Processing and Presentation. Our example case study in section 5.1 has demonstrated that our choice of analytic measures helps understanding an RDF dataset, if their results are combined and concisely presented. Hence, the processing and visualization of analysis results is one of the most important tasks for the future.

Real-World Benchmark. We have evaluated the runtime performance of our different SPARQL-based approaches and measure implementations by running several

benchmarks against a local fuseki instance, loaded with three real-world datasets. This is not enough to judge the feasibility of our approaches for real-world use. To evaluate RDFSynopsis at a broader scale, we should run benchmarks against a more diverse set of triple stores and datasets.

Advanced Triple Streaming. In principle, our work has shown that structural analyses of RDF datasets can be implemented with a Triple Stream Approach. However, during performance evaluation we found that the triple stream queries are very expensive with respect to query runtime. This fact renders TSA and its random sampling variation infeasible for practical use. Future work could include research on triple stream processing that goes beyond the capabilities of SPARQL and current triple stores. Related work includes [4, 9, 10, 24, 60].

Future Development of RDFSynopsis. We have implemented the structural analysis in RDFSynopsis, a command-line tool and software framework. Many additional features seem desirable, including but not limited to: different output formats for results, analysis result caching, automatic generation of result visualizations, and maybe even a web interface. Furthermore, due to many changes along the development process, the project contains some parts whose need for refactoring is obvious. The processing of analysis results should be made more generic.

Using Existing Knowledge. The goal of this thesis was to develop a general-purpose approach for RDF data analysis. As a result, none of our measures is tailored towards specific ontologies (besides RDF, RDFS, OWL and SKOS). Future work could include research on approaches to integrate existing ontologies in the process of analysis. Furthermore, some vocabularies are explicitly used to convey meta-information about a dataset, e.g., the *void* and *dublin core* vocabularies. Specific measures could be used in analysis which explicitly extract these meta-information from the dataset.

SPARQL Entailment Regimes. In our background section on Ontologies (section 2.3), we explained that the standard vocabularies (RDF, RDFS, and OWL) come with explicitly defined semantics, which can be expressed as *inference rules*. In the past, the extent to which inference could be used, varied among different triple stores and SPARQL engines. As a consequence, we implemented the structural analysis in way that both expects no inference of triples and still works with (basic RDFS) inference. A good example is the fact that many of our specific SPARQL queries exclude `rdfs:Resource`. A very recent W3C Recommendation now introduces *SPARQL Entailment Regimes* [47]. An Entailment Regime basically specifies what kind of inference (or “entailment”) a SPARQL query engine automatically, and transparently, performs when it answers a query. Future work should include an in-depth study of the consequences different Entailment Regimes have for the structural analysis of RDF datasets behind SPARQL endpoints. They might improve analysis capabilities as well as render our approaches partly infeasible.

A. Namespace Prefixes

Table A.1 lists all URI prefixes used in this thesis according to the CURIE [18] syntax. A URI like

```
<http://this.namespace.com/specificTerm>
```

with the namespace prefix definition

```
@prefix ns: <http://this.namespace.com/>
```

becomes the following.

```
ns:specificTerm
```

<i>Prefix</i>	<i>Namespace</i>
rdf	http://www.w3.org/1999/02/22-rdf-syntax-ns#
rdfs	http://www.w3.org/2000/01/rdf-schema#
owl	http://www.w3.org/2002/07/owl#
skos	http://www.w3.org/2004/02/skos/core#
foaf	http://xmlns.com/foaf/0.1/
void	http://rdfs.org/ns/void#
dbp	http://dbpedia.org/resource/
yago	http://dbpedia.org/class/yago/
ex	http://www.example.com/
test	http://www.example.com/test/
dbo	http://dbpedia.org/ontology/
ast	http://vocabulary.semantic-web.at/AustrianSkiTeam/
geo	http://www.w3.org/2003/01/geo/wgs84_pos#
geonames	http://www.geonames.org/ontology#
dc	http://purl.org/dc/elements/1.1/
freebase	http://rdf.freebase.com/ns/

Table A.1.: Namespace Prefixes Used in this Thesis

B. Fuseki Endpoint Configuration

We set up our local endpoints using Fuseki [42] from the Apache Jena Project [6]. We started Fuseki with the following line to assign 8GB of RAM and to use the config file presented below.

```
1 java -Xmx8G -jar fuseki-server.jar --config=config-TH.ttl
```

Listing B.1: Starting Fuseki

```
1 #####
2 ## Prefixes ##
3 #####
4
5 @prefix :      <#> .
6 @prefix fuseki: <http://jena.apache.org/fuseki#> .
7 @prefix rdf:    <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
8
9 @prefix rdfs:   <http://www.w3.org/2000/01/rdf-schema#> .
10 @prefix tdb:    <http://jena.hpl.hp.com/2008/tdb#> .
11 @prefix ja:     <http://jena.hpl.hp.com/2005/11/Assembler#> .
12
13 #####
14 ## Fuseki Server with 3 Endpoints ##
15 #####
16
17 [] rdf:type fuseki:Server ;
18     fuseki:services (
19         <#service_DBLP>
20         <#service_AM>
21         <#service_PEEL>
22     ) .
23
24 #####
25 ## dblp Dataset ##
26 #####
27
28 <#service_DBLP> rdf:type fuseki:Service ;
29     fuseki:name "dblp" ;
30     fuseki:serviceQuery "query" ;
31     fuseki:serviceReadGraphStore "get" ;
32     fuseki:dataset <#dblp> .
33
34 <#dblp> rdf:type ja:RDFDataset ;
35     rdfs:label "DBLP 2012 publications" ;
```

```

36   ja:defaultGraph [
37     rdfs:label "dblp-publications-2012.rdf" ;
38     a ja:MemoryModel ;
39     ja:content [
40       ja:externalContent <file:data/dblp-publications-2012.rdf>
41     ] ;
42   ] .
43
44   #####
45   ## AM Dataset ##
46   #####
47
48   <#service_AM> rdf:type fuseki:Service ;
49     fuseki:name           "AM" ;
50     fuseki:serviceQuery   "query" ;
51     fuseki:serviceReadGraphStore "get" ;
52     fuseki:dataset        <#museum> .
53
54   <#museum>      rdf:type ja:RDFDataset ;
55     rdfs:label "Amsterdam Museum" ;
56     ja:defaultGraph [
57       rdfs:label "am-data.ttl" ;
58       a ja:MemoryModel ;
59       ja:content [
60         ja:externalContent <file:data/am-data.ttl>
61       ] ;
62     ] .
63
64   #####
65   ## Peel Dataset ##
66   #####
67
68   <#service_PEEL> rdf:type fuseki:Service ;
69     fuseki:name           "Peel" ;
70     fuseki:serviceQuery   "query" ;
71     fuseki:serviceReadGraphStore "get" ;
72     fuseki:dataset        <#peel> .
73
74   <#peel>      rdf:type ja:RDFDataset ;
75     rdfs:label "Peel" ;
76     ja:defaultGraph [
77       rdfs:label "peel.rdf" ;
78       a ja:MemoryModel ;
79       ja:content [
80         ja:externalContent <file:data/peel.rdf>
81       ] ;
82     ] .

```

Listing B.2: config-TH.ttl

C. Command-Line Usage of RDFSynopsis

RDFSynopsis provides a command-line interface. The `-h` (or `--help`) parameter prints the following usage.

```
1 Usage: rdfSynopsis [options]
2   Options:
3     -all, --allCriteria
4       Use all available criteria for analysis.
5       Default: false
6     -c, --criteria
7       A space-separated list of criteria to use for analysis, e.g.,
8         "-c 3 5 7"
9     -ep, --endpoint
10        The SPARQL endpoint URL that shall be analyzed.
11    -f, --file
12        The RDF dataset file that shall be analyzed.
13    -h, --help
14        Print this usage information.
15        Default: false
16    -lc, --listCriteria
17        Print list of analytical criteria.
18        Default: false
19    -mnq, --maximumNumberQueries
20        The maximum number of queries to perform a partial analysis;
21        "-1" means "infinite". (TSA only, NA)
22        Default: -1
23    -ob, --orderBy
24        One of the following variables used to define an order in the
25        triple stream: subject, predicate, object (TSA only)
26        Default: subject
27    -o, --outFile
28        The filename used to store analysis results. (NA)
29    -rand, --randomSampling
30        Use a "random sampled" triple stream. (TSA only)
31        Default: true
32    -rf, --resultFormat
33        One of the following result output formats: text,... (NA)
34        Default: text
35    -sqa, --specificQuery
36        Use one specific SPARQL query per criterion. (SQA)
37        Default: false
```

```

35  -tl, --tripleLimit
36      The maximum number of triples requested per query. (TSA only)
37      Default: 50000
38  -tsa, --tripleStream
39      Use generic SPARQL queries to create a triple stream. (TSA)
40      Default: false

```

Listing C.1: rdfsSynopsis --help

The criteria which shall be used for analysis can be specified with the `-c` (`--criteria`) command, followed by a list of space separated criteria identifiers. The `-lc` (or `--listCriteria`) command prints a list of all available criteria and their identifiers.

```

1  [id]      criterion
2  -----
3  [1]       class usage count
4  [2]       triples per subject class
5  [3]       explicit class hierarchy
6  [4]       implicit class hierarchy
7  [5]       ontology-ratio
8  [6]       typed-subject-ratio
9  [7]       property usage
10 [8]       predicate vocabularies
11 [9]       property usage per subject class
12 [10]      class instances per property
13 [11]      explicit property hierarchy
14 [12]      implicit property hierarchy
15 [13]      distinct blank subjects
16 [14]      namespace links
17 [15]      distinct subject-only blanks
18 [16]      triples
19 [17]      sameAs
20 [18]      common properties

```

Listing C.2: rdfsSynopsis --listCriteria

Bibliography

- [1] Keith Alexander et al. *Describing Linked Datasets with the VoID Vocabulary*. W3C Interest Group Note. Mar. 3, 2011. URL: <http://www.w3.org/TR/2011/NOTE-void-20110303/> (cit. on pp. 35, 38–41).
- [2] K. Alexander et al. “Describing linked datasets”. In: *Proceedings of the 2nd Workshop on Linked Data on the Web (LDOW2009)*. 2009 (cit. on pp. 33, 35, 38–41).
- [3] D. Allemang and J. Hendler. *Semantic Web for the Working Ontologist: Effective Modeling in RDFS and OWL*. Morgan Kaufmann, 2011 (cit. on pp. 5, 6, 15–18, 20, 25).
- [4] Darko Anicic et al. “EP-SPARQL: a unified language for event processing and stream reasoning”. In: *Proceedings of the 20th international conference on World wide web*. ACM, 2011, pp. 635–644. URL: <http://doi.acm.org/10.1145/1963405.1963495> (cit. on p. 106).
- [5] G. Antoniou and F. Van Harmelen. *A semantic web primer*. 2nd. MIT Press, 2008 (cit. on pp. 5, 7, 15).
- [6] *Apache Jena*. The Apache Software Foundation. URL: <http://jena.apache.org/index.html> (cit. on pp. 80, 91, 109).
- [7] Sören Auer and Jens Lehmann. *LODStats*. URL: <http://aksw.org/Projects/LODStats.html> (cit. on p. 36).
- [8] Sören Auer et al. “LODStats - An Extensible Framework for High-Performance Dataset Analytics”. In: *EKAW*. 2012, pp. 353–362 (cit. on pp. 36, 38–41).
- [9] Davide Francesco Barbieri et al. “C-SPARQL: SPARQL for continuous querying”. In: *Proceedings of the 18th international conference on World wide web*. WWW ’09. Madrid, Spain: ACM, 2009, pp. 1061–1062. URL: <http://doi.acm.org/10.1145/1526709.1526856> (cit. on p. 106).
- [10] D. Barbieri et al. “C-SPARQL: A Continuous Query Language For RDF Data Streams”. In: *International Journal of Semantic Computing* (2010), pp. 3–25. URL: <http://www.worldscientific.com/doi/abs/10.1142/S1793351X10000936> (cit. on p. 106).
- [11] D. Beckett and T. Berners-Lee. *Turtle - Terse RDF Triple Language*. W3C Team Submission 28 March 2011. URL: <http://www.w3.org/TeamSubmission/2011/SUBM-turtle-20110328/> (cit. on p. 14).

- [12] D. Beckett and B. McBride. *RDF/XML syntax specification (revised)*. W3C Recommendation 10 February 2004. URL: <http://www.w3.org/TR/REC-rdf-syntax/> (cit. on p. 14).
- [13] Tim Berners-Lee. *Linked Data - Design Issues*. W3C. 2006. URL: <http://www.w3.org/DesignIssues/LinkedData.html> (cit. on p. 8).
- [14] Tim Berners-Lee and Dan Connolly. *Notation3 (N3): A readable RDF syntax*. W3C Team Submission 14 January 2008. URL: <http://www.w3.org/TeamSubmission/2008/SUBM-n3-20080114/> (cit. on p. 14).
- [15] Tim Berners-Lee and Mark Fischetti. *Weaving the web - the original design and ultimate destiny of the World Wide Web by its inventor*. HarperBusiness, 2000, pp. I–IX, 1–246 (cit. on pp. 6–8).
- [16] Tim Berners-Lee et al. “The World-Wide Web”. In: *Commun. ACM* 37.8 (Aug. 1994), pp. 76–82. URL: <http://doi.acm.org/10.1145/179606.179671> (cit. on p. 7).
- [17] T. Berners-Lee, J. Hendler, O. Lassila, et al. “The semantic web”. In: *Scientific american* 284.5 (2001), pp. 28–37 (cit. on p. 7).
- [18] M. Birbeck and S. McCarron. *CURIE Syntax 1.0 A syntax for expressing Compact URIs* W3C Candidate Recommendation 16 January 2009. 2009. URL: <http://www.w3.org/TR/2009/CR-curie-20090116/> (cit. on pp. 13, 107).
- [19] C. Bizer, T. Heath, and T. Berners-Lee. “Linked data-the story so far”. In: *International Journal on Semantic Web and Information Systems (IJSWIS)* 5.3 (2009), pp. 1–22 (cit. on p. 8).
- [20] Victor de Boer et al. “Supporting Linked Data Production for Cultural Heritage Institutes: The Amsterdam Museum Case Study”. In: *The Semantic Web: Research and Applications*. Springer, 2012, pp. 733–747 (cit. on p. 90).
- [21] C. Böhm, J. Lorey, and F. Naumann. “Creating void descriptions for Web-scale data”. In: *Web Semantics: Science, Services and Agents on the World Wide Web* 9.3 (2011), pp. 339–345 (cit. on pp. 36, 38–41, 43).
- [22] T. Bray, D. Hollander, and A. Layman. *Namespaces in XML*. 1999. URL: <http://www.w3.org/TR/1999/REC-xml-names-19990114/> (cit. on p. 12).
- [23] Dan Brickley and R.V. Guha. *RDF Vocabulary Description Language 1.0: RDF Schema*. W3C Recommendation 10 February 2004. Ed. by Brian McBride. URL: <http://www.w3.org/TR/2004/REC-rdf-schema-20040210/> (cit. on p. 15).
- [24] Jean-Paul Calbimonte, Oscar Corcho, and Alasdair JG Gray. “Enabling ontology-based access to streaming data sources”. In: *The Semantic Web–ISWC 2010*. Springer, 2010, pp. 96–111 (cit. on p. 106).
- [25] S. Campinas et al. “Introducing RDF Graph Summary with application to Assisted SPARQL Formulation”. In: *Proceedings of the 11th International Workshop on Web Semantics and Information Processing.(to appear)*, Vienna, Austria. 2012 (cit. on p. 45).

-
- [26] Jeremy J. Carroll and Graham Klyne. *Resource Description Framework (RDF): Concepts and Abstract Syntax*. W3C Recommendation 10 February 2004. URL: <http://www.w3.org/TR/2004/REC-rdf-concepts-20040210/> (cit. on pp. 11, 15).
- [27] Jeremy Carroll, Ivan Herman, and Peter F. Patel-Schneider. *OWL 2 Web Ontology Language RDF-Based Semantics (Second Edition)*. W3C Recommendation 11 December 2012. Ed. by Michael Schneider. URL: <http://www.w3.org/TR/2012/REC-owl2-rdf-based-semantics-20121211/> (cit. on p. 15).
- [28] Wolfram Conen and Reinhold Klapsing. “A Logical Interpretation of RDF”. In: *Journal of Electronic Transactions on Artificial Intelligence (ETAI), Area: The Semantic Web (SEWEB)* 5 (2000). Note: Not conforming to current standard of RDF semantics. URL: http://nestroy.wi-inf.uni-essen.de/rdf/logical_interpretation/ (cit. on p. 15).
- [29] Wolfram Conen and Reinhold Klapsing. *Logical Interpretations of RDFS - A Compatibility Guide*. 2001. URL: http://nestroy.wi-inf.uni-essen.de/rdf/new_interpretation/ (cit. on p. 15).
- [30] Mariano P Consens and Shahan Khatchadourian. *ExpLOD: Exploring Interlinking and RDF Usage in the Linked Open Data Cloud*. Tech. rep. University of Toronto, 2009 (cit. on pp. 44, 45).
- [31] D. Crockford. *The application/json Media Type for JavaScript Object Notation (JSON)*. Internet RFC 4627. 2006. URL: <http://www.ietf.org/rfc/rfc4627.txt> (cit. on p. 14).
- [32] R. Cyganiak, D. Reynolds, and J. Tennison. *The RDF Data Cube Vocabulary*. W3C Working Draft 05 April 2012. Working Draft. W3C. URL: <http://www.w3.org/TR/2012/WD-vocab-data-cube-20120405/> (cit. on p. 36).
- [33] Richard Cyganiak. *make-void*. Oct. 2010. URL: <https://github.com/cygri/make-void> (cit. on pp. 37–41).
- [34] Richard Cyganiak et al. “Semantic Sitemaps: Efficient and Flexible Access to Datasets on the Semantic Web”. In: *The Semantic Web: Research and Applications*. Ed. by Sean Bechhofer et al. Vol. 5021. Lecture Notes in Computer Science. Springer Berlin / Heidelberg, 2008, pp. 690–704. URL: http://dx.doi.org/10.1007/978-3-540-68234-9_50 (cit. on p. 43).
- [35] Daniel Dietrich and Rufus Pollock. “CKAN: apt-get for the Debian of Data”. In: *26th Chaos Communication Congress, Berlin, Germany, 27–30 December 2009*. URL: <http://events.ccc.de/congress/2009/Fahrplan/events/3647.en.html> (cit. on p. 36).
- [36] Li Ding et al. *Swoogle*. URL: <http://swoogle.umbc.edu/> (cit. on p. 36).

- [37] Li Ding et al. “Swoogle: a search and metadata engine for the semantic web”. In: *Proceedings of the thirteenth ACM international conference on Information and knowledge management*. CIKM '04. Washington, D.C., USA: ACM, 2004, pp. 652–659. URL: <http://doi.acm.org/10.1145/1031171.1031289> (cit. on p. 36).
- [38] L. Ding et al. “Tracking RDF Graph Provenance using RDF Molecules”. In: *Proc. of the 4th International Semantic Web Conference (Poster)*. 2005 (cit. on p. 42).
- [39] Agostino Dovier, Carla Piazza, and Alberto Policriti. “An efficient algorithm for computing bisimulation equivalence”. In: *Theor. Comput. Sci.* 311.1-3 (2004), pp. 221–256 (cit. on p. 44).
- [40] Lee Feigenbaum et al. *SPARQL 1.1 Protocol. W3C Recommendation 21 March 2013*. URL: <http://www.w3.org/TR/2013/REC-sparql11-protocol-20130321/> (cit. on p. 29).
- [41] Javier Fernández, Miguel Martínez-Prieto, and Claudio Gutierrez. “Compact Representation of Large RDF Data Sets for Publishing and Exchange”. In: *The Semantic Web – ISWC 2010*. Ed. by Peter Patel-Schneider et al. Vol. 6496. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2010, pp. 193–208. URL: http://dx.doi.org/10.1007/978-3-642-17746-0_13 (cit. on pp. 36, 38–41).
- [42] *Fuseki*. Apache Jena Project. URL: http://jena.apache.org/documentation/serving_data/index.html (cit. on pp. 91, 109).
- [43] Jan Grant and Dave Beckett. *RDF Test Cases – N-Triples. W3C Recommendation*. Ed. by Brian McBride. W3C, 2004. URL: <http://www.w3.org/TR/rdf-testcases/#ntriples> (cit. on pp. 14, 37).
- [44] Gunnar Aastrand Grimnes, Peter Edwards, and Alun D. Preece. “Instance Based Clustering of Semantic Web Resources”. In: *ESWC*. 2008, pp. 303–317 (cit. on p. 43).
- [45] L. Han et al. *Swoogle’s Metadata about the Semantic Web*. 2006. URL: <http://www.csee.umbc.edu/courses/graduate/691/spring13/01/papers/tr-swoogle-metadata0525.pdf> (cit. on pp. 36, 38–41).
- [46] Steve Harris and Andy Seaborne. *SPARQL 1.1 Query Language. W3C Recommendation 21 March 2013*. URL: <http://www.w3.org/TR/2013/REC-sparql11-query-20130321/> (cit. on pp. 26, 28, 29, 43).
- [47] Sandro Hawke et al. *SPARQL 1.1 Entailment Regimes. W3C Recommendation 21 March 2013*. Ed. by Birte Glimm and Chimezie Ogbuji (cit. on p. 106).
- [48] Patrick Hayes. *RDF Semantics. W3C Recommendation 10 February 2004*. Ed. by Brian McBride. URL: <http://www.w3.org/TR/2004/REC-rdf-mt-20040210/> (cit. on p. 15).

-
- [49] Tom Heath and Christian Bizer. *Linked Data: Evolving the Web into a Global Data Space*. Synthesis Lectures on the Semantic Web. Morgan & Claypool Publishers, 2011 (cit. on pp. 5, 11).
- [50] Aidan Hogan, Andreas Harth, and Tobias Kaefer. *NxParser*. URL: <http://code.google.com/p/nxparser/> (cit. on pp. 37–41).
- [51] Shahan Khatchadourian and Mariano P. Consens. “ExpLOD: Summary-Based Exploration of Interlinking and RDF Usage in the Linked Open Data Cloud”. In: *The Semantic Web: Research and Applications*. Ed. by Lora Aroyo et al. Vol. 6089. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2010, pp. 272–287. URL: http://dx.doi.org/10.1007/978-3-642-13489-0_19 (cit. on pp. 44, 45).
- [52] Andreas Langeegger and Wolfram Wöß. *RDFStats*. URL: <http://rdfstats.sourceforge.net/> (cit. on p. 35).
- [53] Andreas Langeegger and Wolfram Wöß. “RDFStats - An Extensible RDF Statistics Generator and Library”. In: *DEXA Workshops*. 2009, pp. 79–83 (cit. on pp. 35, 38–41).
- [54] L. Masinter, T. Berners-Lee, and R.T. Fielding. *Uniform resource identifier (URI): Generic syntax*. 2005. URL: <http://tools.ietf.org/html/RFC3986> (cit. on p. 12).
- [55] Deborah L. McGuinness and Frank van Harmelen. *OWL Web Ontology Language Overview. W3C Recommendation 10 February 2004*. URL: <http://www.w3.org/TR/2004/REC-owl-features-20040210/> (cit. on p. 15).
- [56] Alistair Miles and Sean Bechhofer. *SKOS Simple Knowledge Organization System Reference. W3C Recommendation 18 August 2009*. URL: <http://www.w3.org/TR/2009/REC-skos-reference-20090818/> (cit. on p. 20).
- [57] Open Knowledge Foundation. *The Data Hub*. URL: <http://datahub.io/> (cit. on p. 36).
- [58] E. Oren et al. “Sindice. com: a document-oriented lookup index for open linked data”. In: *International Journal of Metadata, Semantics and Ontologies* 3.1 (2008), pp. 37–52 (cit. on pp. 37, 45).
- [59] Robert Paige and Robert E Tarjan. “Three partition refinement algorithms”. In: *SIAM Journal on Computing* 16.6 (1987), pp. 973–989 (cit. on p. 44).
- [60] Danh Le Phuoc, Josiane Xavier Parreira, and Manfred Hauswirth. *A Native And Adaptive Approach For Unified Processing Of Linked Streams And Linked Data*. Tech. rep. DERI, July 2011 (cit. on p. 106).
- [61] E. Prud’Hommeaux, A. Seaborne, et al. *SPARQL query language for RDF. W3C Recommendation 15 January 2008*. URL: <http://www.w3.org/TR/2008/REC-rdf-sparql-query-20080115/> (cit. on p. 26).
- [62] T. Segaran, C. Evans, and J. Taylor. *Programming the semantic web*. O’Reilly Media, 2009 (cit. on p. 5).

- [63] Hrvoje Simic. “Predicate trees: a tool for descriptive subgraph extraction”. In: *Proceedings of the 2nd International Conference on Web Intelligence, Mining and Semantics*. WIMS '12. Craiova, Romania: ACM, 2012, 24:1–24:9. URL: <http://doi.acm.org/10.1145/2254129.2254160> (cit. on p. 43).
- [64] Manu Sporny et al. *JSON-LD Syntax 1.0. A Context-based JSON Serialization for Linking Data*. W3C Working Draft 12 July 2012. URL: <http://www.w3.org/TR/2012/WD-json-ld-syntax-20120712/> (cit. on p. 14).
- [65] Patrick Stickler. *Concise Bounded Description*. June 3, 2005. URL: <http://www.w3.org/Submission/CBD/> (cit. on p. 42).
- [66] Rudi Studer, V. Richard Benjamins, and Dieter Fensel. “Knowledge engineering: Principles and methods”. In: *Data & Knowledge Engineering* 25.1–2 (1998), pp. 161–197. URL: <http://www.sciencedirect.com/science/article/pii/S0169023X97000566> (cit. on p. 15).
- [67] The Apache Jena project team. *TDB Optimizer*. Apache Jena, The Apache Software Foundation. 2012. URL: <http://jena.apache.org/documentation/tdb/optimizer.html> (cit. on pp. 37–41).
- [68] Giovanni et al. Tummarello. *Sindice - The semantic web index*. URL: <http://sindice.com/> (cit. on pp. 37, 45).
- [69] Giovanni et al. Tummarello. *Sindice Web Stats*. URL: <http://sindice.com/stats/> (cit. on p. 37).
- [70] Giovanni Tummarello, Renaud Delbru, and Eyal Oren. “Sindice.com: Weaving the Open Linked Data”. In: *The Semantic Web*. Ed. by Karl Aberer et al. Vol. 4825. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2007, pp. 552–565. URL: http://dx.doi.org/10.1007/978-3-540-76298-0_40 (cit. on pp. 37, 45).
- [71] Giovanni Tummarello et al. “Signing individual fragments of an RDF graph”. In: *WWW (Special interest tracks and posters)*. 2005, pp. 1020–1021 (cit. on p. 42).
- [72] Robert Vesse and Gail Alverson. *SPARQL Query Benchmark*. YarcData, a division of Cray Inc. URL: <http://sourceforge.net/projects/sparql-query-bm/> (cit. on p. 89).
- [73] W3C OWL Working Group. *OWL 2 Web Ontology Language Document Overview (Second Edition)*. W3C Recommendation 11 December 2012. URL: <http://www.w3.org/TR/2012/REC-owl2-overview-20121211/> (cit. on p. 15).
- [74] *W3C Semantic Web Activity*. URL: <http://www.w3.org/2001/sw/> (cit. on p. 9).
- [75] World Wide Web Consortium W3C. *Vocabularies*. Accessed on Feb 6, 2013. URL: <http://www.w3.org/standards/semanticweb/ontology> (cit. on p. 13).

- [76] N. Zhang, Y. Tian, and J.M. Patel. “Discovery-driven graph summarization”. In: *Data Engineering (ICDE), 2010 IEEE 26th International Conference on*. IEEE. 2010, pp. 880–891 (cit. on p. 44).