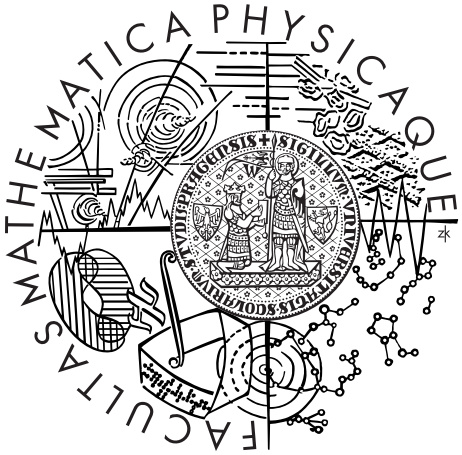
Charles University in Prague
Faculty of Mathematics and Physics

MASTER THESIS



Ondřej Hoferek

Ontological Reasoning with Taxonomies in RDF Database

Department of Software Engineering

Supervisor of the master thesis: Mgr. Martin Nečaský, Ph.D.

Study programme: Computer Science

Specialization: Software Systems

Prague 2012

First of all, I would like to thank my supervisor Martin Nečaský for his valuable suggestions, observations and help with writing and also my family and friends for their support in my life and studies.

I declare that I carried out this master thesis independently, and only with the cited sources, literature and other professional sources.

I understand that my work relates to the rights and obligations under the Act No. 121/2000 Coll., the Copyright Act, as amended, in particular the fact that the Charles University in Prague has the right to conclude a license agreement on the use of this work as a school work pursuant to Section 60 paragraph 1 of the Copyright Act.

In Prague, December 7, 2012 Ondřej Hoferek

Název práce: Ontological Reasoning with Taxonomies in RDF Database

Autor: Ondřej Hoferek

Katedra: Katedra softwarového inženýrství

Vedoucí diplomové práce: Mgr. Martin Nečaský, Ph.D.

Abstrakt: Vzhledem k tomu, že vývoj technologií vznikajících v rámci hnutí za realizaci idey tzv. Semantického webu v posledních letech nabírá na váze, je možné tyto technologie již používat pro řadu různých úkolů. Díky jejich zaměření na analyzování významu informací obsažených v datech je jejich použití obzvláště vhodné pro zvyšování relevance při vyhledávání dokumentů. V předkládané práci analyzujeme možnosti vymezení vhodné podmnožiny vyjadřovacích schopností dotazovacího jazyka SPARQL a vytváříme komponentu zapouzdřující technické detaily jejího použití.

Klíčová slova: RDF databáze, SPARQL, vyhledávání pomocí facetů, podobnostní vyhledávání, návrh API, taxonomie

Title: Ontological Reasoning with Taxonomies in RDF Database

Author: Ondřej Hoferek

Department: Department of Software Engineering

Supervisor: Mgr. Martin Nečaský, Ph.D.

Abstract: As the technologies for the realisation of the idea of the Semantic Web have evolved rapidly during past few years, it is possible to use them in variety of applications. As they are designed with the ability to process and analyze semantic information found in the data in mind, they are particularly suitable for the task of enhancing relevance of the document retrieval. In this work, we discuss the possibilities of identifying a suitable subset of the expressing capabilities of the SPARQL querying language and create a component that encapsulates the technical details of its usage.

Keywords: RDF databases, SPARQL, faceted browsing, similarity search, API design, taxonomies

Contents

Chapter 1

Introduction and motivation

In 2001, the public was presented with the concept of the Semantic Web[47]. The main purpose of this initiative was to enhance the way data was stored on the Web in order to make it machine understandable. That would mean that instead of searching for relevant information on their own, people could ask software agents to perform this task for them. The HTML standard widely-used for the WWW data had been developed for presentation purposes, so it was clear that new standards and technologies would have to be created in order to fulfill the idea behind the Semantic Web.

More then ten years later, we see that the original vision hasn't been accomplished yet. However, standards supporting the process have already been defined by W3C[41]. We are going to mention those of them which are the most important. RDF[23] is the essential one. It is a model which defines how to represent the data on the Semantic Web. It allows to identify entities universally and express their attributes and relations between them. In addition, several serialization formats of RDF are available, such as RDF/XML[27], Turtle[37] and RDFa[24]. In order to be able to determine the actual meaning of the data represented by RDF, standards such as RDFS [26], OWL[18] and SKOS[32] have been created. Moreover, their usage enables logical reasoning over the RDF data and inferring facts that are not explicitly expressed. Finally, SPARQL standard defines a language for querying and updating RDF data together with a protocol that should be used for the technical realization of the query processing.

There are two widespread methods for storing and exposing the RDF data. First of them relies on documents that contain the RDF data in one of the serialization formats. Another one is to store the data in a database and access it using the SPARQL protocol or some API, which is more flexible and efficient. Initially, existing relational databases coupled with RDF mapping extensions or middleware were employed for this task. However, there are already many databases providing native RDF data storage.

The Semantic Web standards together with the technologies already created for storing the data and reasoning over it form the so called Semantic Web Stack which can be used for building semantics-aware applications. One such possible application is to improve the search relevance for document retrieval by identi-

fying entities in the document and semantics of their occurrence. This technique would enable users to specify semantics of their queries as opposed to specifying only keywords using the full text search approaches, such as [48]. Moreover, the entities in RDF data can be organized in taxonomies and so users could enable the queries to find also the documents related to more specific/general entities than the one they specified. Important factor for this feature is the ability to reason over the transitive nature of the taxonomic relations. Finally, such application would benefit from exposing the information extracted in a standardized way and ability to reuse existing ontologies and taxonomies thanks to the universal identification of resources.

The application described above should be able to manage the extracted data efficiently within an RDF store. Currently, there are many stores available, they are evolving quickly, they offer different features on top of the basic standardized functionality and unfortunately, the interfaces for certain advanced features vary from one to another. Therefore, it is desirable to ensure the interchangeability of the stores within the application to the greatest possible extent. A solution for this task would be creation a data storage layer component which would encapsulate all the communication with the underlying store.

1.1 Goals of the thesis

The main goal of the thesis is to create a data storage component designed in order to be used in applications enabling semantics-driven document retrieval. The component should provide methods for identifying documents based on the entities or attributes they are related to. It should support such identification based either on the exact specification of entities/attributes or on specification of a particular document with which the resulting documents should have the relations to entities/attributes in common. Additionally, it should allow to specify whether any taxonomic relations between entities should be taken into account within the queries. Finally, it should be able to identify entities from available taxonomies whose string representation is encountered in given set of words.

Requirements placed on the component functionality should be analysed in order to discuss possibilities of their satisfaction. Existing RDF stores should be explored in order to find those which provide the features needed. The component should be implemented on top of selected stores and a sample set of internally generated queries should be used for their performance evaluation.

The component should be targeted to the Java platform as this is widely used in the Semantic Web domain and generally in projects with bigger scope.

1.2 Structure of the text

In Chapter 2, we describe selected standards of the Semantic Web. This is followed by survey of existing frameworks and APIs for working with RDF data in Chapter 2. The work related to the semantic information extraction from docu-

ments and its retrieval is present in Chapter 3. In Chapter **??**, we discuss the use cases that motivate the component creation and possible solutions. Chapter 5 describes the design of the component API. In Chapter 6, we analyze existing RDF stores and discuss the features they should provide. Chapter 7 gives a brief notice about the component imlpementation and evaluation. Finally, we summarize the results of the thesis and we provide ideas for future work in Chapter 8.

Chapter 2

The Semantic Web standards

We have already mentioned some of the Semantic Web standards briefly. In this chapter, we present more details about them.

2.1 URI (Uniform Resource Identifier)

URI[30] is not a Semantic Web specific standard. However, other standards use it extensively, so we include it in this overview. It is a format specification defining how to represent resources with universally understandable and unique identifier. URIs tend to be very long, so the XML Specification[39] defines a way to express URIs in a short form. So called QNames (qualified names) employ the definition of namespaces which stands for base URIs that form a common prefix for set of another URIs. The other URIs than can be represented by the namespace name and the rest of the URI following the common prefix. These identifiers are not universally unique though and therefore, they can only be used within small scope, such as documents or queries.

Several namespaces commonly used within Semantic Web are listed in Table 2.1. We use them through the whole thesis.

2.2 RDF (Resource Description Framework)

RDF defines an extremely flexible and powerful model which allows us to express not only the data itself, but also the classification of the data. Basically, RDF represents data as a collection of triple statements, each consisting of subject,

Prefix	URI
`rdf:`	`http://www.w3.org/1999/02/22-rdf-syntax-ns#`
`rdfs:`	`http://www.w3.org/2000/01/rdf-schema#`
`xsd:`	`http://www.w3.org/2001/XMLSchema#`
`owl:`	`http://www.w3.org/2002/07/owl#`
`skos:`	`http://www.w3.org/2004/02/skos/core#`

Table 2.1: Commonly used namespace prefixes on the Semantic Web

predicate and object. Each triple expresses a relation between the subject and the object. The type of the relation is determined by the predicate. Set of such statements is called an RDF Graph. The resources used in the statements are divided into three groups.

First group are URI references. These are meant to represent the real-world entities and relations. Thanks to using universal identifiers, it is possible to represent the same entities in different graphs with the same URI and after merging the information from both graphs, align the statements from both graphs to the entity represented by the URI.

Second type of resources are literals. These are, as the name suggests, literal string values. Literals can be tagged with a language tag denoting the language of the value. The tags conform to [29]. The literals may also be typed with types specified by URIs. RDF does not define literal types itself[1]. It relies on the XML Schema[40] built-in datatypes instead. However, any datatype represented by URI can be used.

The last group are blank nodes. They represent entities which are involved in relations between other entities/literals and which aren't identified. Their purpose is to provide means for existence of more complicated relations.

RDF also defines set of built-in resources that can be used in RDF graphs. These resources serve basic expression purposes, such as expressing the type-of relation (`rdf:type`), typing predicate resources (`rdf:Property`) or literals whose values are well-formed XML strings (`rdf:XMLLiteral`), identifying whole statements[2] (`rdf:Statement`, `rdf:subject`, `rdf:predicate` and `rdf:object`) and defining containers(`rdf:Seq`, `rdf:Bag`, `rdf:Alt`, `rdf:_1`, `rdf:_2`, etc.) or collections (`rdf:List`, `rdf:first`, `rdf:rest` and collection termination element `rdf:nil`). Using these resources, set of axiomatic triples, which are implicitly present in any RDF graph. is formed. Finally, the standard describes how to interpret the information represented by statements and graphs logically together with the axioms for entailment of one graph from another.

2.3 RDF serialization formats

Several formats were specified for serializing the RDF data model into documents. The original format which was created together with the RDF model uses a special XML syntax and it is called RDF/XML. It is a normative syntax for RDF [23]. Another format is Turtle. It doesn't rely on any underlying language and it tends to be more compact than RDF/XML which makes it the preferred format for writing RDF by hand. We use this format for examples in this work. Apart from formats which are intended for serializing RDF to its own documents, there are also formats which are meant for annotating existing documents of some other

[1] An exception is the literal type `rdf:XMLLiteral` mentioned later.

[2] The purpose of the identification of whole statements is the ability to state facts about them such as, who is the creator of the statement [23].

format usually used for presentation purposes. One such format is RDFa which can be used in order to include the semantic information into HTML documents.

2.4 RDFS (RDF Schema)

As the main purpose of RDF is to define a model and its interpration, it doesn't provide any means for classifying entities and their relations. Instead, RDFS, which is presented as a complement to RDF, defines resources needed for this task. It allows us to declare types of entities (`rdfs:Class`) and hierarchical relations between these types (`rdfs:subClassOf`) as well as to specify the types of entities which can take part in relation specified by given property (`rdfs:domain`, `rdfs:range`) and hierarchy of properties (`rdfs:subPropertyOf`). It also introduces common supertypes for containers, their membership properties and literals. Finally, RDFS adds several axiomatic triples to the set defined by RDF.

Exploiting the interpretaion and entailment framework described by RDF, RDFS adds rules for reasoning over vocabularies[3] and even the data constrained by given vocabulary. These rules involve the transitivness of `rdfs:subClassOf` and `rdfs:subPropertyOf` relations, inferring additional types for entities following the `rdfs:subClassOf` relation between classes or based on the appearance of the entity in a triple with given property (thanks to the specification of the domain/range of the property). It is also possible to infer a relation specified by some property between two resources if any triple defines relation between these nodes specified by any of its sub-properties.

RDFS entailement, as defined initially, was undecidable due to an infinite set of axiomatic triples but the standard was changed in order to solve this issue[64]. With this change, it is NPTIME-complete and and when rules involving generation of blank nodes are ommitted, it is even PTIME-complete [53].

2.5 OWL (Web Ontology Language)

OWL builds on top of RDF and RDFS. It adds new resources for describing vocabularies and ontologies and it describes the semantics of their interpretation. Three subsets of OWL can be used: OWL Full, OWL DL and OWL Lite.

OWL brings its own `owl:Class`, `owl:Thing` and `owl:Individual` concepts. We can use it to specify that resources with different URI represent the same entity (`owl:sameAs`) or implicitly express that they are different from each other (`owl:differentFrom`, `owl:AllDifferent`). It allows us to further express the nature of relations using constructs such as `owl:FunctionalProperty`, `owl:TransitiveProperty`, `owl:SymmetricProperty` and a properties relation `owl:inverseOf`. It also allows us to restrict cardinality of domain or range of properties, construct special entity classes with set operations and even more ontological definitions.

[3] I.e. collection of types, properties and relations between them.

OWL complexity has its disadvantages. Reasoning is undecidable or computationally expensive due to the fact that unlike in RDF/RDFS where if-then semantics is mainly used, OWL rules are often defined with if-and-only-if semantics [53]. Therefore, entailment with OWL Lite is EXPTIME-complete, it is NEXPTIME-complete with OWL DL and it is even undecidable using OWL Full. This leads to creation of custom interpretation models which use a subset of OWL vocabulary with interpretation strictly using the if-then semantics put on top of RDFS. One such model was defined by Horst [53] and is offened reffered to as OWL-Horst. Entailment with this model is NPTIME-complete and when ommitting the blank nodes creation rules from RDFS, it is even PTIME-complete.

Limitations of OWL have been mostly addressed with the introduction of OWL 2[19]. It introduced new constructs such as reflexive or asymmetric properties and weakened some limitations to the OWL DL (this time OWL 2 DL) subset. Most importantly, it introduced profiles - subsets of language with their own semantics targeted to provide efficient reasoning capabilities. The profiles are OWL 2 EL targeted to efficient reasoning over the classification information, OWL 2 QL concentrating on providing SQL query rewriting capabilities[4] without the need of storing inferred data and OWL 2 RL meant as the most-complete entailment set keeping the computation relatively easy. All profiles provide PTIME-complete entailment.

2.6 SKOS (Simple Knowledge Organization System)

As its name suggests, SKOS is a standard that provides vocabulary/ontology for knowledge organization systems relying on RDF and OWL. Its main class, `skos:Concept`, is defined for representing concepts - basic units of knowledge. Concepts can be organized in concept schemes or collections. It also provides means for standardized concept labeling and mapping between concepts from different schemes. Important contribution of SKOS is the determination of relations that can be found between concepts. It defines `skos:broader` and `skos:narrower` (mutually inverse) properties in order to represent hierarchical relations in taxonomies. Another example could be `skos:related` for symmetric expression of an elementary (non-hierarchical) relation between two concepts.

SKOS is widely used in order to create an RDF representation of existing knowledge bases in projects such as DBpedia[46] or TheSoz[66].

2.7 SPARQL

In order to be able to specify queries over RDF data, SPARQL standard was created. It includes definition of the query language for querying and updating the data and the protocol that can be used to perform the queries over HTTP.

SPARQL queries are based on matching triple patterns where any part of the

[4]OWL ontologies aren't restricted to be used only with the RDF model.

pattern is either a variable or a resource. Specifying a resource (either with URI or as a literal value) is used in order to restrict triples that can match the pattern. Variables are used in order to obtain node values (either literals or URIs again). This variable than can be used in the result, for further restriction of the patterns matched (e.g. limiting possible values that the variable can be bound to) or for connecting with other patterns used in the query. When multiple patterns are specified, all of them have to be matched in order to satisfy the query. It is also possible to merge results from two groups of patterns using the UNION operation or specifying a group of patterns as optional which is often used to include additional non-required information in the result. Patterns can be limited to include only triples from particular graphs. Queries can be nested and aggregates over groups of results can also be computed.

There are four types of SPARQL queries: SELECT, ASK, CONSTRUCT and DESCRIBE. SELECT is used in order to obtain values of variables (or their aggregates) bound in the matching patterns. ASK can be used in order to find out whether any result satisfying given patterns exists. In CONSTRUCT query, obtained variables can be used in order to create a new RDF graph. Last of the query forms, DESCRIBE, can be used in order to return triples related to the nodes either specified as variables bound by the patterns or directly with their URI reference. However, the standard doesn't specify which triples should be included in this description so different SPARQL implementations can return different set of triples for the same data.

In addition to querying inserting and deleting triples to/from graphs based on the patterns matched or specified explicitely is also possible. Operation for changing any part of the triple is not supported.

SPARQL is aware of the fact the dataset queried can consist of more than one graph. It allows to specify particular graph in which a triple satisfying particular pattern should be located or to declare this graph as variable, making it possible to obtain the information about the location of the satisfying triple as a result of the query. If no graph restriction (either particular or variable) is specified, so called default graph should be used in order to find triples satisfying patterns. No further details on how this default graph is constructed are specified by SPARQL. In practice, RDF stores either contain special default graph in addition to the named graphs or construct the default graph by merging all named graphs. Additionally, SPARQL allows to specify named graphs from the dataset which should be used to form the default graph for the query by FORM clauses and restrict the set of named graphs that can be used for explicit pattern matching by the FROM NAMED clause.

Chapter 3

Related work

Infomation extraction from existing documents and its querying isn't an entirely new field. In this chapter, we present a survey of projects that deal with this problematics with the help of the Semantic Web Stack. As our component should basically offer an API to query RDF data in a specific way, we also investigate existing APIs and frameworks for working with RDF data.

3.1 Wikipedia related projects

Wikipedia[42] is a free collaboratively built encyclopedia consisting of more than four million English articles. Versions for other languages also exist but not all the articles are available in all languages. Articles are organized in categories which being further organized in a hierarchy serve for easier information retrieval. Articles can also contain semi-structured information in so called info boxes where common attributes of certain groups of article subjects are expressed.

Thanks to the opened nature of Wikipedia and the amount of semantic information stored there, it is used as a knowledge base for several semantic information retrieval projects. In [45], Athenikos and Lin showed that retrieval based on the information extracted from Wikipedia had better recall and precission results than traditional keyword search using a domain-specific (movies) resource base. Appart from research on domain-specific information extraction and retrieval, general/multi domain projects such as YAGO[63], WikiTaxonomy[61] and DBpedia [46] were created.

All of the projects use the Wikipedia categories in order to define taxonomies. However, they differ in the way they extract the information. YAGO uses only the lowest-level categories from the hierarchy and builds the taxonomy using relations between category labels as specified by WordNet[60]. The taxonomy is in turn expressed with help of RDFS classes and their subClassOf relation. Article topics are considered individuals and following a simple set of rules their presence in some category is converted to either instantiation of the entity represented by the article to the type defined by the category or some attribute the entity has (e.g. the year a person was born in). The dataset can be downloaded or accessed via a public SPARQL endpoint. WikiTaxonomy also expresses the resulting taxonomy as a RDFS class hierarchy. As opposed to YAGO, it exploits the category hier-

archy and it tries to recognize the subClassOf relations in this hierarchy. It doesn't work with the rest of the category relations though. The dataset can be downloaded. Finally, DBpedia uses the category hierarchy as it is and it simply projects it to `skos:broader` and `skos:narrower` or `skos:related` relations. It further extracts the information from info-boxes in order to express attributes of the entities represented by articles and defines its own classification of these entities. DBpedia also uses the YAGO classification and it links the resources to other data sets, such as GeoNames[8] or Music-brainz[13].

Appart from the dataset extraction and its availability via download or public SPARQL endpoint, several applications using the dataset have been developed. We mention applications which are related to the intended use of our component: DBpedia Spotlight[59] and Faceted Wikipedia Search[52].

DBpedia Spotlight is a system for automatic annotation of documents with resources from DBpedia dataset. It involves identification of possible resources, their disambiguation based on the context and scoring of the results. Identification of possible resources relies on the Aho-Corasick string matching algorithm[44]. The disambiguation phase exploits indices built by Lucene framework[12] over several different weights computed from contexts of occurence of the resources as a link in Wikipedia and Wikipedia disambiguation description. The resources available for matching can be restricted by the DBpedia type or even arbitrary SPARQL query over the DBpedia dataset.

Faceted Wikipedia Search aims to enhance the search possibilities over the set of Wikipedia articles (subjects) with the semantic information exposed via the DBpedia dataset. It provides combination of the full-text like search with the exact restriction choice provided by means of facets. Facets are basically certain attributes of subjects such as their types (common attribute of all subjects) or the river mouth (attribute specific to subjects of type river) with a set of existing values. When a keyword is entered to the full-text search, the most frequent facets are searched and the entered value is matched to some facet value. Subjects attributed by this value are then displayed as results. The set of results can be further restricted by specifying more facet values, either via the full-text search or exact restriction choice. Exact restriction choice is done via displaying the most frequent values of certain facets and possibility to choose one or more of these values. For numerical values, it is possible to select range of the values by specifying its extremes. If the type of subjects was already determined by a previous user search, facets are restricted to those specific for given type.

While the faceted search user interface is publicaly accessible, the implementation of both full-text search and exact restriction choice is proprietary.

3.2 KIM

While Wikipedia related projects work with a specific document structure and dataset, KIM platform[62] focuses on domain-independent information extraction and its retrieval from different types of documents. It allows to extract

occurences of entities from documents and even locate relations between entities stated in documents. In order to recognize entities, basic knowledge base (containing mainly geographical data, organizations, people) in form of RDF data together with describing ontology is supplied. Other domain ontologies, taxonomies and data can ba added. The framework also allows learning of new entities and relations between them. Extraction of the information is done with the help of customized GATE framework[7] and Lucene indexing capabilities are employed in order to index documents with the entities encountered. Finally, the information extracted is stored in the OWLIM[57] RDF store.

Once the information is extracted, KIM offers several ways of its retrieval. It enables the users to find out how many documents particular entity appeared in. Another use case is identification of entities that co-occur with other entities (that can be furter restricted, e.g. by their type, other attributes or full-text match of their string representation) in some documents and ordering appropriate entities per number of the co-occurences. In addition, entities and their relations can be found based on SPARQL-like pattern matching and attribute/string-representation restriction. Documents in which these patterns occur can be returned too. Different interfaces can be used for the retrieval: web browser based GUI, web services and Java RMI methods.

3.3 Pelorus and Spanner

Spanner[33] is another system for RDF and OWL data extraction from arbitrary documents. It extracts relations between documents and entities. It can use domain ontology and taxonomy in order to enhance extraction results. However, it is claimed to perform well even without providing these thanks to the built-in machine learning. Together with Pelorus[21], it also provides a browser based GUI for faceted search. Combination of full-text facet value search and facet values selection from list similar to the functionality of Faceted Wikipedia Search is employed. Both systems use Stardog[35] as their backend RDF store.

3.4 PoolParty

PoolParty is a system for taxonomy management and semantic document search. It consists of three main parts: Thesaurus Manager, Knowledge Discoverer and Semantic Search, all providing a web browser based GUI. Knowloedge Discoverer and Semantic Search also provide an HTTP based API. Using Thesaurus Manager, one can import existing taxonomies/concept schemes defined using SKOS or create new ones. Taxonomies can be enriched by creating new entities/concepts, relations between them, and linking them to entities from external data sources such as DBpedia. Knowledge Dicoverer serves for identifying entities from taxonomies in documents. Besides that, new candidate entities appearing in text are offered to user who can choose which of these entities can be submitted to taxonomy and select a subset of offered entities to tag the document. Finally, Semantic Search provides faceted search and similarity search over set of imported documents. Facets and their values correspond to taxonomies and their entities.

Their relation to document is the tag occurence.

Internally, RDF data is stored using Sesame[50]. Document tagging and recognition of taxonomy entities is based on Lucene indices.

3.5 RDF processing frameworks

In order to make manipulation with RDF data easier, several frameworks have been created. They usually support basic operations with the dataset such us adding and removing individual triples, obtaining and removing statements based on simple patterns (where only a part of the triple is specified), working with types and language tags of literals, loading statements from files in different RDF serialization formats or exporting the dataset the other way around. On the other hand, they differ in the way the statements included in the dataset can be organized and the additional functionality they offer, such as querying the dataset using more sophisticated query languages or working with ontologies and inferencing. The frameworks can usually be stacked on top of some RDF storage system. Thus, it is possible to abstract working with the storage layer. Apart from storage imlplementaions provided out of the box, third party implementations of storage also exist and include many existing stores in some cases.

Apache Jena[9] is a Java framework that consists of several components: Core, ARQ, LARQ, Fuseki and storage components SDB and TDB. Core component defines basic abrstractions for working with RDF data: simple Graph and more sophisticated Model, together with their in-memory implementations. It also offers Model extensions OntModel for working with RDFS schemas and OWL ontologies and InfModel which provides support for inference over RDF data. Several reasoner implementations are supplied. Models can be stacked on top of each other in order to provide additional funcitonality. An example of such usage would be stacking of the InfModel on top of a plain storage model. ARQ component adds support for querying graphs and models using SPARQL and representing RDF stores with the Dataset abstraction which allows organization of statements and their querying by means of named graphs. LARQ component adds basic full-text search capabilities to ARQ SPARQL queries using Lucene. SDB allows to store and query data represented by Jena models/datasets in relational databases. TDB is a native RDF storage implementation solution. Finally, Fuseki embedds all the other components together with Apache Jetty[3] in order to provide a standalone SPARQL server.

Sesame[50] is another Java framework which can also be used as a standalone system. It abstracts RDF data as collections of statements organized with help of context URIs called repositories. Repositories are created on top of so called Sails which are interfaces whose appropriate implementation enables any RDF store to be used as a repository. Similarly to Jena models, Sails can be stacked. Sesame comes with three Sail storage implementations: in-memory, native and relational database one. Forward chaining RDFS reasoner Sail which can be used on top of in-memory and native storage is also included. In addition to basic RDF manipulation API, repositories can be queried using SPARQL and SeRQL[49]

languages. Sesame distribution comes with a standalone HTTP server which can be deployed in any Java Servlet container. This server operates on set of repositories that can be accessed either as SPARQL endpoints or via Sesame HTTP based protocol. Sesame HTTP clients include a console application, web browser GUI application and remote Repository implementation.

Last Java framework we mention is Oroboro[17]. It represents the RDF datasets by a simple Graph interface with no notion of context nor named graph concpets. In-memory implementation of graph is included. Oroboro-specific feature is the evaluation of a custom Datalog based language which can be used for querying, updating and inferencing over the graphs. Two reasoners, one implementing the OWL 2 RL profile and another using a custom set of basic OWL and RDFS rules are built-in.

dotNetRDF[6] is a .Net framework representing RDF datasets with ITripleStore interface which operate with named graphs abstracted by the IGraph interface. Triple store objects can be queried with SPARQL and API for working with ontologies is also included. Inference can be done using either RDFS and SKOS (taking into account transitivity and mutual inverseness of `skos:broader` and `skos:narrower`) built-in reasoners or a generic rule reasoner. Full-text search over literals is implemented using a .Net port of Lucene. Included triple store implementations are in-memory and persistent built on top of Talis Platform or Virtuoso[51] as a storage back-end. It is also possible to deploy it as a SPARQL endpoint via ASP.Net.

Another .Net framework is SemWeb[31]. It abstracts the RDF datasets with the Store interface operating on quads, thus taking into account named graphs. It supports executing SPARQL queries end backward-chaining reasoning with either RDFS reasoner or generic rule reasoner. In-memory and persistent (with relational database back-end) implementations of Store are provided. Again, it is possible to deploy included Store implementations as a SPARQL endpoint via ASP.Net. LinqToRdf library[11] which translates LINQ[10] queries to SPARQL is built on rop of SemWeb.

Applications written in C can use the Redland framework. It supports working with RDF abstracted as a Model structure supporting organization of statements by context. In-memory and Oracle Berkeley DB[15] or Virtuoso backed native together with several relational database based implmementations are provided. Querying with SPARQL and RDQL is done with help of accompanying Rasqal library. Redland is shipped with command line utilities for manipulating and querying data using provided Model implementations. Finally, it offers language bindings for Perl, PHP, Python and Ruby.

RDFLib[25] is a pure Python library. RDF data is manipulated through the Graph objects that are context-aware and can be queried using SPARQL. Interesting feature of Graph is the possibility of transitive closure evaluation for patterns where predicate and subject or object are specified without any reasoning enabled ressembling similar functionality using SPARQL property paths. Graphs

can be persisted using one of the supplied store implementations: In-memory, Oracle Berkeley DB, Zope Object Database[43] and several relational databases.

In addition to traditional RDF manipulation APIs consisting of fixed set of classes and methods, there are libraries providing an object - RDF mapping. Using these libraries, resources from RDF graphs are accessed as objects and predicates from statements where they appear as subjects can be used as their properties having values of corresponding object nodes. In Python, there are several such libraries based on RDFLib: ORDF[16], Sparta[34] and SuRF[36]. Another library based on object - RDF is ActiveRDF[1] for Ruby which can operate on any SPARQL endpoint.

Chapter 4

Analysis

After having presented so many existing frameworks for RDF data processing that could operate on some RDF storage system, it might look like the task of creating a component for encapsulating this functionality within the scenario of document retrieval application is a bit like carrying coals to Newcastle. Therefore, we are going to further discuss the advantages of doing so.

Even though the frameworks abstract the access to the storage layer, they usually only provide API for basic data manipulation and retrieval. This might be sufficient in case of updating the data as the expected usage in case of the document search scenario would be to take the graph resulting from the document information extraction and adding the statements included to the store. However, when it comes to retrieval, more sophisticated queries need to use one of the query languages (usually SPARQL). Again, this shouldn't be a problem since SPARQL is a wide-spread standard and its implementation is included in the majority of stores. However, SPARQL specification is evolving. Some stores use different syntax for particular functionality as they included it earlier than it was put into the standard. Good example could be syntax for property paths that can be used in Virtuoso SPARQL implementation. Such deviations are either propagated through the framework abstraction implementation or they aren't supported at all. Some functionality is not even part of the standard, such as the ability to include full-text search patterns in queries. Again, the consequence is the violation of the abstraction of the underlying implementation.

Another complication is the evolution of SPARQL itself. From time to time, the syntax or semantics of some constructs just changes. In order to preserve the original purpose of the queries, it is therefore needed to adjust all SPARQL queries affected by such change. In a complex domain-independent system by which our component is targeted to be used, it might turn to be very difficult to identify all such queries. By encapsulating the queries into well-defined API, it is only needed to change the implementation of the API which allows us to minimize the code affected by the changes.

Finally, not all the features currently defined by SPARQL are available for all the available storage systems. By analyzing the functionality that should be provided by the component, we can identify the features needed for its imple-

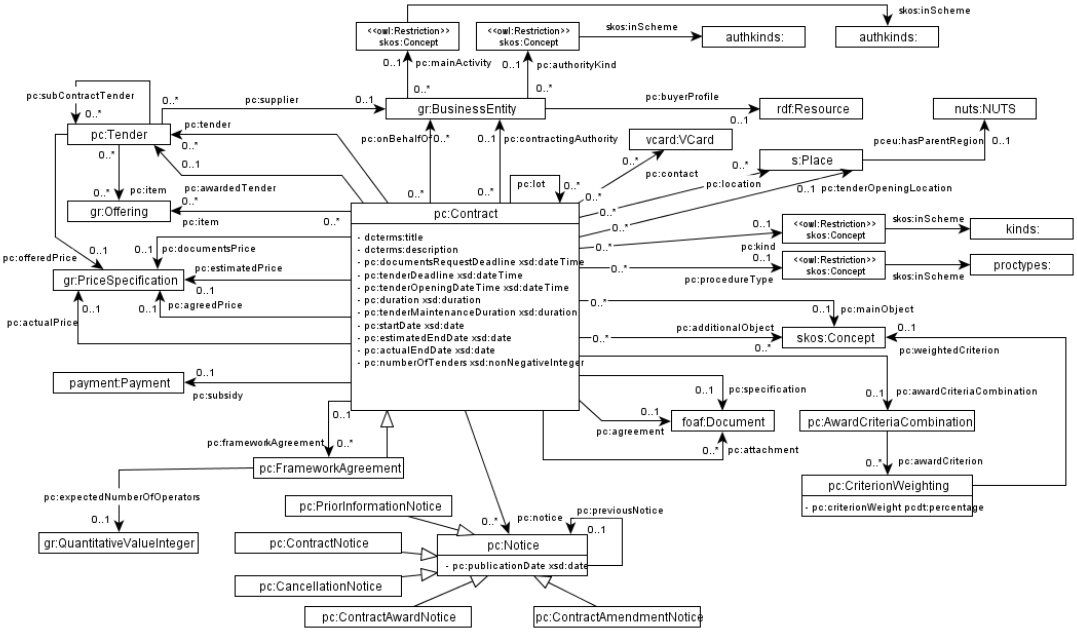


Figure 4.1: UML diagram of the Public Contracts Ontology

mentation which helps us in the process of selection of the underlying RDF store. In case more candidate stores are available, we can also use the knowledge about the features in order to either select appropriate existing evaluation framework or design a new evaluation schema in case none of the existing is covering the important features.

Now that we have stressed the importance of the component creation, we are going to discuss the motivations considered and decisions taken in the process leading to its creation. We start with presenting the Public Contracts Ontology[55] which describes the sample dataset we use in examples to support our argumentation and also for the component evaluation in later chapters. This is followed by analysis of the requirements placed on the component based on the research of possibilities offered by SPARQL and existing RDF stores. This chapter isn't intended to give a detailed survey of stores. This is provided in Chapter 6.

4.1 Public Contracts ontology

The Public Contracts Ontology describes the domain of data that can be extracted from notices about public contracts made by goverments and authorities of cities, regions and countries. It was chosen as an example of an ontology which is used in practice by ongoing projects. Moreover, it exploits other existing and widely accepted ontologies making it possible to link the dataset to publicly available taxonomies.

In order to ease understanding of the ontology, we include its UML diagram in Figure 4.1. The main class of the ontology is Contract which represents the contracts themselves. The contract has several attributes which give informa-

tion about the dates of its announcement and realization. Information about notices made about the contract is expressed by the `pc:notice` property. The originators of the contract are captured with help of `pc:onBehalfOf` and `pc:contractingAuthority`. Another information about the origin of the contract are the `pc:contact` and `pc:location` attributes. The later one can be linked to NUTS[14] which is the standardized European nomenclature of countries and regions. The ontology also allows to express the information about objects of the contract (`pc:mainObject` and `pc:additionalObject`) which are instances from the CPV product scheme taxonomy[5] in reality. Another important data about contracts are the competing tenders (`pc:tender`) and the applicants (`pc:supplier`) and offered prices (`pc:offeredPrice`) behind them. Finally, the information about the winning offer of the contract specified by (`pc:awardedTender`) and estimated (`pc:estimatedPrice`) and final (`pc:agreedPrice`) price can be expressed.

4.2 Identifying entities by their string representations

First of the main functionality aspects that should be provided by the component is the ability to identify entities present in taxonomies and even the document data itself by words that appear in their literal representation. We have seen that this task has an important role in the information extraction process where it allows to allign the information encountered in the document to the already known data. It may also serve for the simulation of the full-text like search in the faceted search solution over extracted data similar to those in presented projects. There, facet values (represented by entities from taxonomy) are searched based on the user input string and matched entities are used to constrain the search. Our solution should provide basic funcitonality of returning candidate matches for further analysis by upper layer of the application.

Looking at SPARQL string literal matching possibilities, we find out that it provides simple string equality testing on one hand and on the other hand several more sophisticated functions for string comparison. These functions serve for testing whether one string starts or ends with another one (`STRSTARTS` and `STRENDS`) or simply contains it (`CONTAINS`). It is also possible to ensure case-insensitive comparison thanks to the `UCASE` and `LCASE` normalizing functions. The most sophisticated function provided by SPARQL is `REGEX` which evaluates a string against a regular expression pattern. However, this function is usually claimed to be computationally expensive. In addition to SPARQL functionality, many RDF stores offer effective word-matching solution realized by creating a full-text index over the set of stored literals. Configuration possibilities of these indices are not that rich as with full-featured full-text search system. For example, additional processing and modification of the literals before building an index is not possible. However, some stores provide the options for accent-insensitive search and noise words ignoring. The querying features usually include abilities to test appearance of sequence of words, multiple words (using the AND operator) or at least one of given words (OR operator).

Once we are aware of the matching possibilities, we can take a look at the use case of identifying entities in text. We have several options of fulfilling its needs. First, we can search for exact occurence of the word representation of the entity or at least its part. However, the word order of the representation found in the text can be different from the one stored in the database. Therefore, performing only exact matches could lead to recognition misses. Another option is not take the order of the words as they appear in the document into account. Following this option, we would be able to recognize entity representations in more relaxed way. The disadvantage is that non-suitable entities can also be found. In the scope of our task, it seems reasonable to choose the second option in order to deliver more data to the upper layer delegating the decision about entity recognition validity.

We see that neither SPARQL nor the full-text search extensions provide the ability to search for occurence of the set of words which the literals consist of in provided text. They rather allow to search for occurence of words in the literal representation. However, there is no option to specify that all the words forming the literal should be matched by given set of words. Aware of this limitation, we propose a solution consisting of two phases. First, using either the full-text search matching of at least one of given words (using the OR operator) or SPARQL `CONTAINS` function together with built-in OR logical operator, we identify candidate literals that contain at least one of the words occuring in text. Then, the literals for which not all the words from their representation are encountered in the text can be filtered out manually. Such procedure could be applied also in the scenario in which not all the words forming the literal need to be matched. Then the filtering could simply be ignored.

When it comes to returning partial matches, question whether they can be ordered according to their relevance arises. The full-text implementations usually contain some variant of scoring the results of matching. However, this feature is mostly usable in case larger documents are indexed and frequency of the words appearing in them is taken into account. In contrary, the literal representations of entities tend to be short and the scoring function then usually just correspond to the number of words matched. Therefore, if the results of the underlying query would be post-processed by the component, better option might be to manually compute the ratio between matched words and total number of words that form the literal.

Another consideration for the matching functionality is the possibility to deal with errors in the entitty representations found in text/entered by user. Incorporating some kind of fuzzy matching (i.e. matching not only the exact words but also words that are close enough in terms of some distance metric such as [65]) would be a solution to this problem. However, SPARQL does not provide any such functionality and it is an exotic feature even when it comes to full-text capabilities of RDF stores with Stardog being the only one supporting it. Therefore, we propose not to provide any fuzzy matching solution in the feature set of the component.

So far, we considered the use case of identification of entity representation in

a set of words. There is yet another scenario where matching of entity representations may be helpful. It is the auto-complete functionality for text boxes where the end user enters the query during the faceted full-text like search. Instead of waiting for them entering the whole query and then matching entities in a set of whole words, part of the word already written can be assumed as a candidate for a representation of a constraining entity. Prefix matching, during which all representations that contain at least one word that starts with given characters are returned, suits this task. It is available both in SPARQL (by means of the `STRSTARTS` function) and as a usual full-text feature provided by RDF stores.

Appart from simply returning the entities whose representations are matched, taking some other information into account might be handy in some cases. For example, when processing some notice about public contract, thanks to its semi-structured nature, we can expect only certain type of entity, such as location, to be found in certain part of the document. Therefore, we propose that the component should allow restricting entities for matching by their type or by the property used for their string representation (such as label or description). Specifying multiple types or properties should be allowed. Any of the properties specified can be used to find suitable representations In case of types, semantics of `rdfs:domain` or `rdfs:range`, i.e. conformance of the entity to all the types specified should be followed. Additionally, taxonomies often provide the entity representations in several languages. These representations are usually marked by particular language tag. Thus, it should be also be possible to restrict the search by only searching literals marked with the language specified.

Putting it all together, we see that both the SPARQL included functionality and full-text extensions of RDF stores can be used for entity matching. However, considering the indexed nature of full-text search promissing better performance, we claim that full-text search functionality suits the component requirements better and should be considered a required feature of the store upon which the component can be implemented.

4.3 Retrieval of relevant documents

The main goal of the component is to provide a framewrok for retrieval of documents based on the RDF data extracted from their contents. The data contain particular relations between document and some entities, such as identification of some service or product as the main object of the public contract. Thanks to possessing the information about the type of relation, user can be allowed to use it when searching as opposed to full-text and tagging approaches where the relation between document and given term is unknown and thus, also entities representing additional objects of the contract may be taken into account. Moreover, with help of taxonomical information provided in RDF data, the query can be enriched in order to consider wider range of relevant entities satisfying given relation.

In this section, we take a look at how these relations between documents and entities or literal attributes and between entities themselves are represented in

the data. We also investigate the possibilities of representing and combining the search constraints expressed by users in queries in order to satisfy such enhanced retrieval.

4.3.1 Faceted search paradigm

The usual way of fulfilling the enhanced search capabilities is through faceted search. We have already mentioned this paradigm when describing some related projects. The idea behind it is that the user is aware of facets - different types of attributes that the documents can possess. These facets correspond to relations expressed either by individual properties or property paths in RDF graphs represented by multiple consequent properties. Example of such property path is the sequence of two properties `pc:awardedTender` and `pc:supplier` which identifies a relation between a public contract and awarder supplier going through an intermediate instance of `pc:Tender`. The instances of `gr:BusinessEntity` class which can be reached by following this path starting from `pc:Contract` instances form a range of values for the facet represented by this path. Not only entities can be facet values, but literals such as integers or dates can also have this role.

```
SELECT DISTINCT ?doc WHERE {
  ?doc pc:awardedTender|pc:supplier ?sup .
  FILTER (?sup IN (sample:SupplierA, sample:SupplierB))
}
```

Listing 4.1: Sample query representing the entity facet value restriction

Depending on the type of the facet range, its values can be restricted differently. If the range is formed by entities, user can usually pick one or more of these entities in order to specify that only documents which are related in terms of given facet to at least one of these entities should be returned. Such a restriction can be represented by simple SPARQL query with only one matching pattern consisting of document variable in place of a subject, property path representing the facet relations and variable representing the facet values as an object together with FILTER expression using the IN construct in order to test membership of the value in given set. Sample query representing this restriction for the "awarded supplier facet" is presented in Listing 4.1. For the sake of brevity, we omit the PREFIX declarations of namespaces in the query as well as in all the queries listed later. In case of a facet with literal range, restriction by one or more exact values is also possible. However, majority of literal types such numeric, date and even plain string types can be ordered. That means that it is possible to define a range of values acceptible for the facet. Such restriction would again be represented by a single matching pattern followed by a FILTER expression this time using standard inequality operators together with extremes of the range. It is also possible to set maximum and minimum of the range at the same time thanks to the AND ogical operator. This operator together with other logical operators: logial OR and negation can be used in order to create arbitrary combinations of restrictions.

```
SELECT DISTINCT ?doc WHERE {
  ?doc pc:agreedPrice ?spec .
    ?spec gr:hasCurrency ?curr .
      FILTER (?curr = "EUR") .
    ?spec gr:hasCurrencyValue ?value .
      FILTER ((?value >= 1000000) && (?value <= 2000000))
}
```

Listing 4.2: Query representing value restriction for facet with two dimensions

In some cases, a facet can have multiple dimensions that should be taken into account when constraining the search. An example of such facet would be relation between public contract and its agreed price. The price is not simply an amount of money. It is an instance of `gr:PriceSpecification` class which has several attributes of which the most important are `gr:hasCurrency` which specifies the currency of the price specification and `gr:hasCurrencyValue` which specifies the ammount of money in given currency. Obviously, some amount of Czech korunas has different value than the same amount of euro. Therefore, it is desirable to consider both of the attributes/dimensions when restricting the values of the facet. In SPARQL, we could do so by nesting property paths. The first one would relate the contract and the price specification entities and two other property paths, one for each of the facet dimensions, would relate the price specification entity to the currency and to the currency value. An example of such SPARQL query is presented in Listing 4.2 where we request documents whose agreed price was between one and two million euro.

An important aspect of the usability of facet restrictions is the ability to apply restrictions from multiple facets at the same time. In SPARQL, this can be represented by simply combining the matching patterns and FILTER expressions as we did in case of restricting the values of multiple dimensions.

```
SELECT DISTINCT ?doc ((?relevance1 + ?relevance2) AS ?relevance)
   WHERE {
  ?doc pc:awardedTender|pc:supplier ?sup .
  FILTER (?sup IN (sample:SupplierA, sample:SupplierB)) .

  OPTIONAL {
    ?doc pc:agreedPrice ?spec .
      ?spec gr:hasCurrency ?curr .
        FILTER (?curr = "EUR") .
      ?spec gr:hasCurrencyValue ?value .
        FILTER (?value >= 1000000) .
      BIND (20 AS ?relevance1)
  } .

  OPTIONAL {
    ?doc pc:mainObject ?object .
      FILTER (?object = sample:ServiceA) .
    BIND (30 AS ?relevance2)
  }
```

```
} ORDER BY DESC(?relevance)
```

Listing 4.3: Facets used for determining the relevance of the results

Another option of dealing with multiple facet restrictions is to treat some of them as optional, i.e. that non-fulfilling the constraints represented by the facet doesn't lead to exclusion of the document from the result but doing so makes the document more relevant. An example of such scenario would be to search for contracts that were agreed with a particualar supplier and increasing relevance of those of the satisfactory contracts whose agreed price was more than one million euro or that have a particular service as their main object. The later condition being more relevant than the first one. The relevance of results can be expressed by integer weights which we call significances. In SPARQL query, it is possible to create OPTIONAL patterns and bind the variables inside the OPTIONAL pattern which can be used for increasing the relevance. In the projection part of the query, we can calculate the total relevance with their help and order the results using the ORDER BY clause. Such representation is shown in Listing 4.3.

```
SELECT DISTINCT ?doc (MIN(?value) AS ?relevance) WHERE {
  ?doc pc:awardedTender|pc:supplier ?sup .
    FILTER (?sup IN (sample:SupplierA, sample:SupplierB)) .

  OPTIONAL {
    ?doc pc:agreedPrice ?spec .
      ?spec gr:hasCurrency ?curr .
        FILTER (?curr = "EUR") .
      ?spec gr:hasCurrencyValue ?value
  }
} GROUP BY ?doc ORDER BY DESC(?relevance)
```

Listing 4.4: Example of ordering results by values of the facet

When it comes to ordering the results, it is also possible to designate a facet whose values for individual documents will determine the ordering weight. Again, we give an example of contracts awarded to selected suppliers, this time ordered by their agreed price (if present) in euro in Listing 4.4. We realize that simple usage of the value for ordering isn't suitable for cases in which the facet can have multiple values for one document. Therefore, we need to use suitable aggregate.

4.3.2 Non-trivial facet constructs

So far, we have presented the usual facet types and their usage. In this section, we propose some non-trivial constructs that could be used to enrich the faceted search.

```
SELECT DISTINCT ?doc WHERE {
  ?doc pc:awardedTender|pc:supplier ?sup .
    FILTER (?sup IN (sample:SupplierA, sample:SupplierB)) .

  { SELECT ?doc WHERE {
      ?doc pc:lot ?sub
```

```
    } GROUP BY ?doc HAVING (COUNT(?sub) > 1)
  }
}
```

Listing 4.5: Demostration of facet defined by aggregate

The first facet extension idea lies in constructing the facet values by aggregatting. The range of facet would then usually be numeric (depending on the type of aggregate used) with all the consequences such as the possibility to specify range of allowed values. In our example, we present the facet represented by property `pc:lot` and COUNT aggregate. Its semantics is the count of sub-contracts of contract. We combine it with the agreed supplier facet in order to demonstrate that inner query must be used for its realization in multi-faceted search. We observe that the facet range restriction is realized with help of the HAVING clause. The resulting query is available as Listing 4.5.

```
SELECT DISTINCT ?doc WHERE {
  ?doc pc:estimatedPrice ?estimated .
    ?estimated gr:harCurrency ?curr .
    ?estimated gr:hasCurrencyValue ?estimatedValue .
  ?doc pc:agreedPrice ?agreed .
    ?agreed gr:harCurrency ?curr .
    ?agreed gr:hasCurrencyValue ?agreedValue .
      FILTER (?agreedValue > ?estimatedValue)
}
```

Listing 4.6: Facet composed from values of multiple facets

Another possibility is to construct a facet based on some relation between two other facets. We demonstrate the motivation behind this with an example of a boolean facet indicating whether the agreed price of the contract was higher than the one estimated. Another possibility would be to construct this facet as a numeric difference between the two prices. Obviously, we need to consider the fact that both price specifications should have the same currency. SPARQL query highlighting the usage of this facet restriction for document search is presented in Listing 4.6. Numeric difference between the two values can be realisable by adding a numeric operation on one side of the (in)equality. Unfortunately, for other types than numeric, SPARQL doesn't support the use of numeric operators. Therefore, computing duration between two date time values is not possible.

```
SELECT DISTINCT ?doc WHERE {
  { ?doc pc:mainObject <http://purl.org/weso/pscs/cpv/2008/resource
     /71300000> }
  UNION
  {
    ?doc pc:mainObject ?rel .
    ?rel skos:broader <http://purl.org/weso/pscs/cpv/2008/resource
       /71300000>
  }
}
```

Listing 4.7: Taxonomy relation usage for result enrichment

In order to exploit the potential lying in linking the data to established taxonomies, we consider use case in which the specification of values used for facet restriction is enriched by entities that are related to the entities from the original specification by some taxonomical relation such as `skos:broader`. With help of this facet enhancement, we allow user to specify more general entity representing the area he is interested in and obtain also documents related explicitely to more specific concepts. For example, he could specify "engineering services" to be the main subject of the contract and obtain also contracts having "subsurface surveying services" as their main object. We illustrate this example with appropriate SPARQL query in Listing 4.7. Note the usage of resource from the CPV taxonomy. The query is realized with two groups of matching patterns. The first group serves for matching the original entity and the second group is used for identifying related entities as facet values. Resulting combination of matches from both groups are concatenated with help of UNION SPARQL pattern. The relations between entities are often represented by transitive or symetric property or a pair of mutually inverse properties. In order to evaluate the complete hiearachy of the taxonomy, reasoning should be used. Reasoning capabilities are discussed in section 4.4.

```
SELECT DISTINCT ?doc WHERE {
  ?doc pc:notice ?notice .
  FILTER NOT EXISTS
  {
    ?doc pc:awardedTender|pc:supplier ?sup .
      FILTER (?sup IN (sample:SupplierA, sample:SupplierB))
  }
}
```

Listing 4.8: Negative facet restriction example

The last facet enhancement we mention is the possibility of negative application of facet values restriction, i.e. taking any of the facets and determining that the documents matched by its constraints should be excluded from the results. This construct could be represented with help of the FILTER NOT EXISTS pattern in SPARQL. It is important to remember that some positive statement matching documents is needed to be used together with negation as the negation used on its own doesn't lead to inclusion of any triples into results. We model the usage of negative facet by example in which all contracts about which any notice is taken and at the same time they weren't awarded to given set of suppliers are returned in Listing 4.8. Negation of simple entity restriction facet is used there.

4.3.3 Similarity search

In addition to the faceted search paradigm that can be used for document identification based on an exact set of constraints, we also propose a way to identify documents that are similar to some other document which is explicitely identified by its URI. The idea behind is that documents that share the values represented by some facets, or these values are somehow related, should be considered similar.

```
SELECT DISTINCT ?doc WHERE {
  sample:targetContract pc:tender|pc:supplier ?sup .
  ?doc pc:tender|pc:supplier ?sup
}
```

Listing 4.9: Basic similarity search

Such similarity searches can be realized in SPARQL thanks to its matching machinery which is designed in a way that the variables can be shared by multiple matching patterns. We simulate this ability on a simple example where such contracts are retrieved which were competed by at least one same supplier as the target contract. This case can be expressed by a query whose first pattern is used to identify the competing suppliers of the target contract and second pattern is used to obtain the similar contracts sharing the variable for suppliers following the same property path from the contract subject repersented by the result variable. The query is detailed in Listing 4.9. This similarity query has a few limitations. First, it does not express the growing relevance of the contracts that share more than one competing supplier with the target contract. Second, it cannot be used in combination with other similarity dimensions with such semantics that the more dimensions are matched, the more similar to the target the resulting contract is considered. In order to resolve these limitations, we must use some relevance measure and define how to express it in the SPARQL queries.

```
SELECT ?doc (SUM(?significance) AS ?relevance) WHERE {
  sample:targetContract pc:tender|pc:supplier ?sup .
  ?doc pc:tender|pc:supplier ?sup .
  BIND (20 AS ?significance)
} GROUP BY ?doc ORDER BY DESC(?relevance)
```

Listing 4.10: Similarity search aware of number of common attributes

We propose to use a simple relevance schema in which each dimension available for similarity evaluation is assigned an integer weight that we call significance. By extending the initial example by aggregating the results over the resulting contract entity and using the SUM of significance variable bindings in order the sort the results as shown in Listing 4.10, we solve the issue of expressing growing relevance of contracts with more similarity matches.

```
SELECT ?doc (SUM(?significance) AS ?relevance) WHERE {
  { SELECT ?doc ?significance WHERE {
      sample:targetContract pc:tender|pc:supplier ?sup .
      ?doc pc:tender|pc:supplier ?sup .
      BIND (20 AS ?significance)
    }
  }
  UNION
  { SELECT ?doc ?significance WHERE {
      sample:targetContract pc:additionalObject ?service .
      ?doc pc:additionalObject ?service .
      BIND (10 AS ?significance)
    }
  }
```

```
} GROUP BY ?doc ORDER BY DESC(?relevance)
```

Listing 4.11: Similarity search with multiple dimensions

Adding another dimension to the query can be done by simply concatenating the matching patterns to those of the first dimension, marking each group of matching patterns as OPTIONAL (in order to allow results where only one dimension is matched), binding another significance variable and counting the resulting relevance for a match as a SUM of significances for all groups. However, such solution faces a problem caused by the way SPARQL results are generated. As a result of matching two groups of patterns, SPARQL generates a SUM of significances for all combinations of matches for resulting contract. That means that in case the first dimension is matched once while the second dimension is matched twice, two result combinations of significances are generated by SPARQL and both of them count with significances from both dimensions. Therefore, in the resulting SUM, we lose the information about which of the dimensions was actually matched twice. Solution to this problem is to create a subquery for each of the dimensions and join these subqueries into union of the results. That way, significance for all dimension matches are counted just as many times as they appear. Resulting query is presented in Listing 4.11 where dimension for matching additional objects of contracts is added.

```
SELECT ?doc (SUM(?significance1 + ?significance2) AS ?relevance)
   WHERE {
  sample:targetContract pc:tender ?targetTender .
  ?doc pc:tender ?resultTender .

  ?targetTender pc:supplier ?sup .
  ?resultTender pc:supplier ?sup .
  BIND (20 AS ?significance1) .

  OPTIONAL {
    ?targetTender pc:offeredPrice ?targetPrice .
    ?resultTender pc:offeredPrice ?resultPrice .
      ?targetPrice gr:hasCurrency ?cur .
      ?resultPrice gr:hasCurrency ?cur .

      ?targetPrice gr:hasCurrencyValue ?value .
      ?resultPrice gr:hasCurrencyValue ?value .
      BIND (10 AS ?significance2)
  }
} GROUP BY ?doc ORDER BY DESC(?relevance)
```

Listing 4.12: Similarity search considering multiple aspects of a dimension

Similarly to facets having multiple dimensions in faceted-oriented matching, dimensions for similarity matching can have multiple aspects that should be matched together. This is desirable in cases where the combination of matches bears semantic information, such as when considering dimension of the suppliers that competed for contracts together with the price they offered. Obviously, matched together the two aspect rise the relevance of the result contract. On the other hand, the information about sharing the offered price without sharing the supplier who offered it doesn't seem to be suitable for increasing the relevance of the simi-

larity match. Therefore, we propose to combine the patterns for both aspects in one query sharing the intermediate tender entity variables and marking the offered price aspect as optional as shown in Listing 4.12. In this query, we also consider different aspect of the price itself: the currency and the value. The currency match doesn't have any significance on its own but it is needed in order to ensure comparability of the values. In general, we propose to allow multiple aspects of one dimension to be combined and their subset to be marked as optional.

```
SELECT ?doc (SUM(?significance1) AS ?relevance) WHERE {
  sample:targetContract pc:additionalObject ?service .
  ?doc ppc:additionalObject ?service .
  ?service skos:broader <http://purl.org/weso/pscs/cpv/2008/resource
     /71300000> .
  BIND (20 AS ?significance)
} GROUP BY ?doc ORDER BY DESC(?relevance)
```

Listing 4.13: Similarity search with restriction of values considered for similarity

In order to be able to take control over the values of attributes that can be considered for similarity evaluation, we propose to allow restriction of these values similar to restriction of simple (either literal or entity) facet values. This can be done easily by following the same patterns as in case of facet values restriction, this time applied to the shared variable representing the common value. We illustrate this case in Listing 4.13 where similar contracts are found based on the mutual additional objects which are limited to be `skos:narrower` than "engineering sevices".

```
SELECT ?doc (SUM(?significance) AS ?relevance) WHERE {
  sample:targetContract pc:agreedPrice ?targetPrice .
  ?doc pc:agreedPrice ?resultPrice .
    ?targetPrice gr:hasCurrency ?cur .
    ?resultPrice gr:hasCurrency ?cur .

    ?targetPrice gr:hasCurrencyValue ?targetValue .
    ?resultPrice gr:hasCurrencyValue ?resultValue .
    FILTER ((?resultValue >= 0.7 * ?targetValue) && (?resultValue <=
        1.3 * ?targetValue))
    BIND (10 AS ?significance)
  }
} GROUP BY ?doc ORDER BY DESC(?relevance)
```

Listing 4.14: Similarity search allowing range of literal matches

In all the previous examples of similarity search, we worked with exact matches only. This is a significant limitation for similarity evaluation. In case of literal values, such as integer price values, it is desirable to allow wider space of possible similar values that would be restricted by a range based on the value of the target attribute. Such a range can be defined within a FILTER expression combining restrictions defined with help of the target attribute value applied to the result attribute value. We give an example of using this approach in Listing 4.14 where we evaluate similarity based on the agreed price of the contract allowing the similar values to be in range between 0.7 and 1.3 multiplies of the target agreed

price. Again, we also need to include the currency limitation.

```
SELECT ?doc (SUM(?significance) AS ?relevance) WHERE {
  SELECT ?doc (MAX(?significance) AS ?significance) {
    { sample:targetContract pc:mainObject ?service .
      ?doc ppc:additionalObject ?service .
      BIND (40 AS ?significance)
    } UNION {
      sample:targetContract pc:mainObject ?service .
      ?doc ppc:additionalObject ?rel .
      ?rel skos:broaderTransitive ?service .
      BIND (30 AS ?significance)
    } UNION {
      sample:targetContract pc:mainObject ?service .
      ?doc ppc:additionalObject ?rel .
      ?rel skos:narrowerTransitive ?service .
      BIND (20 AS ?significance)
    }
  } GROUP BY ?doc ?service
} GROUP BY ?doc ORDER BY DESC(?relevance)
```

Listing 4.15: Similarity search with allowing similarity recognition through taxonomic relations

Similarly, we consider widening the similarity value space for entity attributes. In their case, we can work with the taxonomical relations between entities, e.g. do not have to consider only the exact matches for similarity but also the values that are somehow related to the target entity. Example of such scenario could be represented by working with the main object dimension for similarity and, besides similarities found as exact matches between main objects of different contracts, allow also matches that identify entities that are `skos:broader` or `skos:narrower` than the main object of the target contract. We realize that the query should also be able to distinguish between cases of exact match and related match by means of using different significance. In case more matches are enountered for one target entity following the relation based enrichment, only the result with the biggest significance should be used to compute the relevance. We propose to solve this task by employing the union of groups of patterns representing each of the matching variant together with binding of the significacne variable. Results from these unions should be aggregated over the resulting document entities and target entities matched for given dimension. Significance value for the dimension should be computed as a maximum of the significance values bound by the result matching patterns. In order to compute the overall relevance for each document matched, yet another aggregation should be performed and the significances summed. We present resulting query in Listing 4.15. The approach of combining multiple variants of enriching the value space considered for similarity recognition can be applied also in case of ranges of literal values.

4.4 Reasoning about taxonomy entities

We already discussed the standards that can be used in order to define ontologies and schemes with whose help it is possible to infer facts that are not explicitly

If graph contains	Where	Then infer and add to graph
u `rdfs:subClassOf` v v `rdfs:subClassOf` w		u `rdfs:subClassOf` w
u `rdfs:subClassOf` v e `rdf:type` u		e `rdf:type` v
p `rdfs:subPropertyOf` q q `rdfs:subPropertyOf` r		p `rdfs:subPropertyOf` r
p `rdfs:subPropertyOf` q e p f	q $\in$ U $\cup$ B	e q f
p `rdfs:domain` u e p f		e `rdf:type` u
p `rdfs:range` u e p f	f $\in$ U $\cup$ B	f `rdf:type` u

Table 4.1: Important RDFS inference rules

present in the data. Being aware of the fact that more sophisticated reasoning frameworks can be computationally expensive, we concentrate on the features important for working with relations between taxonomy entities and their types and attributes. Our goal is to find a reasonable subset of rules that should be provided by the underlying store. Another approach that can be taken when the infered statements should be considered is to pre-compute them using some standalone inference engine, such as [20]. However, in our work we concentrate on working with raw data, thus also testing the reasoning capabilities of the stores.

From the set of basic RDFS rules, we identify the six most important. They include transitivity of `rdfs:subClassOf` and `rdfs:subPropertyOf` relations, typing the instance with all super-classes of its types and inferring a relation between two nodes represented by a property whose sub-property already defines a relation between these two nodes. They also serve for inferring types of nodes based on their appearance in relation expressed by property with `rdfs:domain` / `rdfs:range` specified. A summary of these entailment rules is given in Table 4.1. In the left column we see patterns whose presence should imply inference of the triple in the right column. The statement in the middle column expresses a constraint that the particular node should either be a universally identified resource or a blank node.

We selected these rules because they help us to identify hierarchy of classes and properties which is important to consider in some cases, such as when restricting the type of the entity to be identified by its string representation. Together with specifying the specific six rules, we also note the last two rules for type resolution based on appearance of an entity in some relation aren't that important as the first four since in wide-spread taxonomies, entities are usually typed explicitely.

When picking up suitable constructs and rules from OWL, we take the subset identified by Horst which is prooved to be workable with effectively as our guidance. This subset is highlighted in Table 4.2. Again, we note that the last four

If graph contains	Where	Then infer and add to graph
p rdf:type owl:FunctionalProperty u p v . u p w	v ∈ U ∪ B	v owl:sameAs w
p rdf:type owl:InverseFunctionalProperty u p w . v p w		u owl:sameAs v
p rdf:type owl:SymmetricProperty v p w	w ∈ U ∪ B	w p v
p rdf:type owl:TransitiveProperty u p v . v p w		u p w
v p w		v owl:sameAs v
v p w	w ∈ U ∪ B	w owl:sameAs w
v owl:sameAs w	w ∈ U ∪ B	w owl:sameAs v
u owl:sameAs v . v owl:sameAs w		u owl:sameAs w
p owl:inverseOf q . v p w	w, q ∈ U ∪ B	w q v
p owl:inverseOf q . v q w	w ∈ U ∪ B	w p v
v rdf:type rdfs:Class v owl:sameAs w		v rdfs:subClassOf w
p rdf:type rdf:Property p owl:sameAs q		p rdfs:subPropertyOf q
e p u e owl:sameAs f . u owl:sameAs v	u ∈ U ∪ B	f p v
v owl:equivalentClass w		v rdfs:subClassOf w
v owl:equivalentClass w	w ∈ U ∪ B	w rdfs:subClassOf v
v rdfs:subClassOf w w rdfs:subClassOf v		v owl:equivalentClass w
v owl:equivalentProperty w		v rdfs:subPropertyOf w
v owl:equivalentProperty w	w ∈ U ∪ B	w rdfs:subPropertyOf v
v rdfs:subPropertyOf w w rdfs:subPropertyOf v		v owl:equivalentProperty w
v owl:hasValue w v owl:onProperty p . u p w		u rdf:type v
v owl:hasValue w v owl:onProperty p u rdf:type v	p ∈ U ∪ B	u p w
v owl:someValuesFrom w v owl:onProperty p x rdf:type w . u p x		u rdf:type v
v owl:allValuesFrom w v owl:onProperty p u rdf:type v . u p x	x ∈ U ∪ B	x rdf:type w

Table 4.2: Rules using subset of OWL for effective entailment identified by Horst[53]

rules, which are used either for entailment of entity types or reasoning about value of property common for all instances of given class, can be considered as not so important for reasoning over taxonomies. On the other hand, the reasoning rules following `owl:TransitiveProperty`, `owl:SymmetricProperty` and also `owl:inverseOf` constructs are very important in order to help us to reason about taxonomical relations. For example, the widely used `skos:broader` and `skos:narrower` properties are sub-properties of `skos:broaderTransitive` and `skos:narrowerTransitive` which are transitive and inverse of each other at the same time. Another wide-spread property is `skos:related` which is symmetric. Basically, the properties describing some hierarchy are usually transitive and come in a mutually inverse pair. Properties describing "flat" relations are usually symmetric. Finally, `owl:sameAs` can play an important role when the taxonomies are merged from different datasets. As it is not usually used otherwise, the RDF stores capable of reasoning often offer to disable reasoning over this property as it might not be desired in context of given dataset.

When it comes to RDF storage engines, they usually implement some subset of original OWL, such as OWL Horst or some subsets of OWL-Lite. Some stores already support profiles from OWL 2, from which OWL 2 RL seems to be the one mostly used in case no simple rule-based reasoning is available. Finally, some stores don't support reasoning at all or support just the basic RDFS rules.

Chapter 5

Design of the component API

In previous chapter, we analyzed the scenarios of intended component usage and discussed possibilities of their fulfillment. We observed that a generic solution based on SPARQL query potentially enhanced with RDF store specific functionality is achievable in most cases. In this chapter, we build on the analysis and go on by designing the API of the component. We discuss the main interfaces and structures proposed and determine the behavior of methods,

5.1 Initial considerations

From use cases mentioned during the analysis, we see that the component should mainly be used for querying the RDF data, not manipulating them. The data have rather static character being extracted from documents once, stored and remaining unchanged. Most of RDF stores provide suitable utilities in order to load data from their textual representation.

We identified two main areas of functionality that should be provided by the component: entity identification based on their string representation and retrieval of document entities together with their attributes based on constraints defined in terms of faceted or similarity search. We isolated these areas into two services: `IEntityMatcher` and `IDocumentSearch`, each providing suitable methods for their tasks. We also include interface `ITaxonomySet` defining methods for answering basic queries over taxonomies. In order to isolate the work with configuration of the component, we also define the `IServiceFactory` whose instantiation is intended to accept any implementation-specific configuration and provide instances of the services which shouldn't be needed to instantiate by users themselves.

5.2 Entity Matcher

The tasks `IEntityMatcher` should be capable of performing are pretty straightforward. The analysis revealed two objectives: searching for string representations within given set of words on one hand and on the other hand, containment of given word prefix. The first task can be further divided into two variants, one identifying only representations for which all the words are encountered within

the set of words supplied and the other delivering also partial matches. We also proposed that the API should provide means to restrict the search by the type of the entity, relation determining the string representation and language of the representation.

Having the requirements in mind, we define three groups of overloaded methods for `IEntityMatcher`: `findEntireRepresentations` only retrieving full matches, `findRepresentations` retrieving also partial matches of entities and `matchWithPrefix` for returning candidate entities represented by given prefix. Each method appears with two signatures, one containing only the values for matching (collection of words in case of the first two methods and single prefix in case of the third) for non-restricted matching and another one adding the `MatchingConfiguration` object. `MatchingConfiguration` holds the information about the restrictions for the search: collection of strings representing URIs of restricting types, another collection of URIs of properties and string identifying the language of the representation. The implementation of restricted method calls should follow the semantics of type / property restriction as discussed earlier, i.e. the entity must be an instance of all types specified and any property can be used for locating its string representation.

Every method returns a set of some kind of `IEntityMatch` objects. These objects contain the information about matched entity URI, the representation property and matched literal. This is all the information returned with help of`IPrefixEntityMatch` by `matchWithPrefix`.

Objects being represented by instances of `IFullEntityMatch` returned by the `findEntireRepresentations` contain also collections of words that matched the individual words from the representation. It is important to note that one word from the string representation can be matched by multiple words depending on the underlying implementation which can be for example accent-insensitive. `findRepresentations` returns collection of `IFullEntityMatch` objects and also a list of `IPartialEntityMatch` which besides the matched words list contain also the list of the non-matched words and the match ratio which is used as a measure for sorting the partial matches.

5.3 Document Search

While the scenarios for Entity Matcher were relatively easy to turn into API having the set of matching parameters fixed, Document Search is intended to provide rich possibilities of quering the data based on variable combinations of restrictions that are not known in advance. As the interchangeable representation of the the queries is one of the main goals of the component, we consider it the main task of the API design.

5.3.1 Representing queries for constrained search

First, we have a look at the variety of queries that were used to represent facet search scenarios. Obivously, they are all document-oriented, i.e. they serve

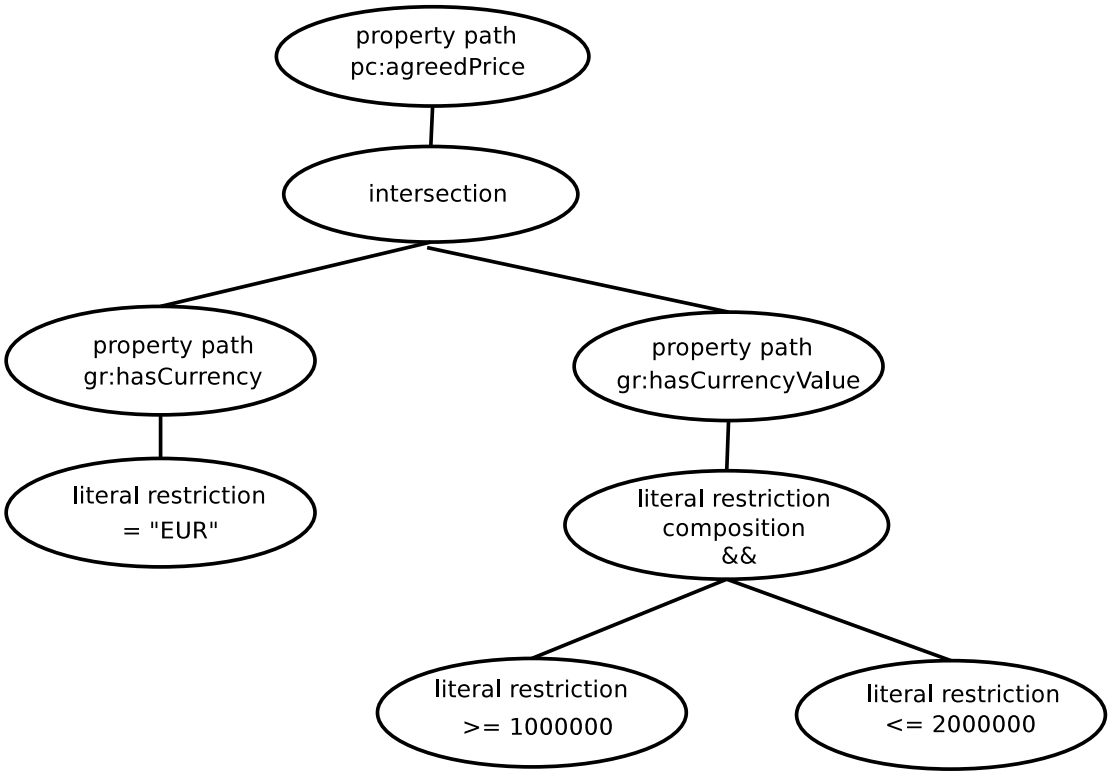


Figure 5.1: Tree representation of the query

for identifying documents by expressing relations to their attributes, restricting values of these attributes and combining multiple such attribute restrictions for different relations at the same time. Basic relation could be represented by a list of URIs forming its property path together with the restriction of the object value. If no restriction was used, the query would return all documents with any attribute value present. As for literals, we identified that the restriction can be represented either by a single condition expressed with a comparison operator and restricting value, condition enumerating the exact values the literal should have (using the IN operator) or a combination of these conditions created using traditional boolean operators AND, OR and negation. In case of resources, we are limited to the exact match either against one value or a set of values.

In case where multiple dimensions of some facet are considered, the property path leading from the document node to objects is branched in the intermediate node and the two branches are matched together. Similarly, in the use case of considering also entities that have some relation to the restricting entity as in Listing 4.7, the property paths are branched too but this time branches are separated in different pattern groups and result matches merged using union (not combining the results from different branches).

These observations lead to the idea of representing the query as a tree-like expression formed by nodes representing either the property path of the relation or combination of other nodes (with intersection or union semantics). Additionally, any property path node could be followed by another expression (or combination of expressions) in order to support branching or atrributed by either resource

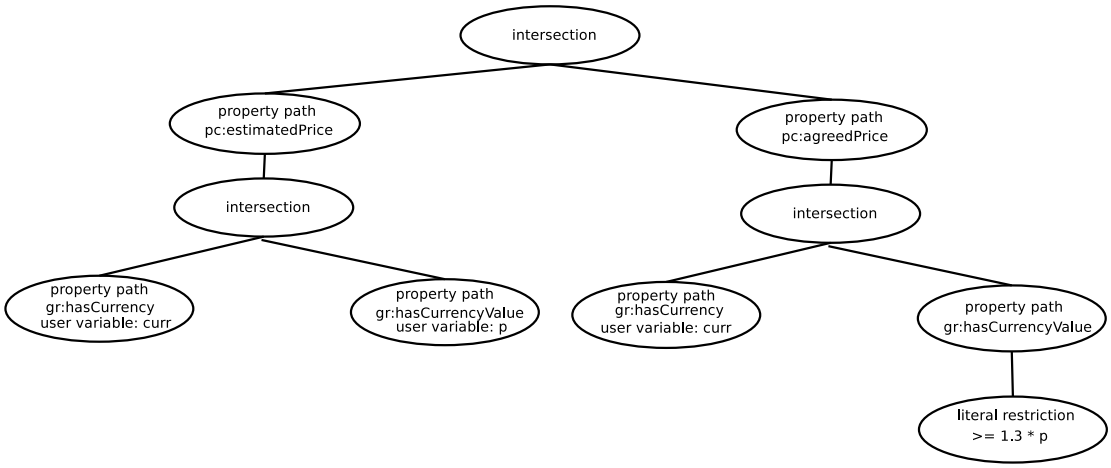


Figure 5.2: Tree representation of the query

or literal restriction applied to the objects of the relation. This tree expression is sketched in Figure 5.1 where it represents the query considering facet with two dimensions from Listing 4.2. Great advantage of this representation is that the user does not need to deal with the variables used in the query as the question of which variable to use in order to express certain relation is simply implied by the structure.

Nevertheless, we allow the assignment of user-defined variables for objects of relations expressed by property paths. This is the only place where new variables can appear (with the exception of the aggregate expression discussed later). For example, user could specify the variable used for the objects of the relation expressed by property `gr:hasCurrencyValue` in the previous example. This way, we allow the user to represent non-trivial relations such as facets composed from values of multiple facets. In order to enable this potential, we also need to allow usage of variables inside of the literal/resource restrictions together with their optional modification by some numeric operation. Combining these features, we can express the query returning contracts whose agreed price is bigger than 1.3 multiply of its estimated price and both prices are expressed using the same currency as presented in Figure 5.2.

We define two groups of expressions. First group is formed by expressions which can be used as a root of the query. They include the property path, intersection and union expressions. We call this group standalone expressions. Other group of expressions are the literal/reference restrictions which can only be used for restricting the value of object reached by some property path. Another expressions that can be standalone are expression negations. These simply apply the `NOT EXISTS` filter on the supplied inner expression. Although being standalone, they should only be used together with intersection with some positive expressions. Otherwise, they simply generate no results.

Last type of expressions are the aggregate expressions. These need to be constructed with an inner standalone expression (in order to have some data to aggregate) and can specify restrictions on the aggregate values with help of

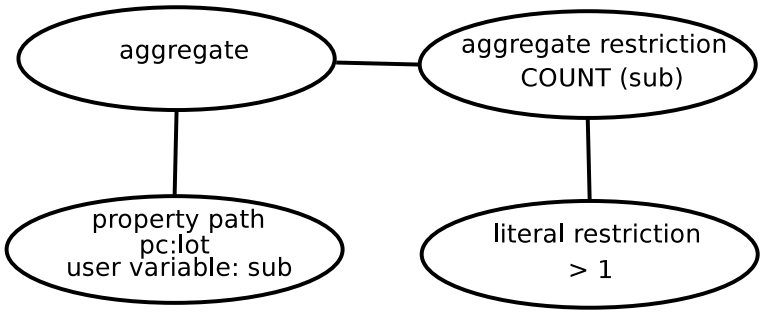


Figure 5.3: Tree representation of the query

literal restriction expressions or simply assigning some aggregate value a variable in order to be referenced in some other expression in the query. An example of this expression can be found in Figure 5.3 representing the query searching for contracts with more than one sub-contracts. A variable bound in the inner expression is needed in order to express either aggregate restriction or new variable formed by an aggregate. We should also note that it is always aggregated over objects of the closest enclosing property path expression. If no such expression exists, it is aggregated over entities representing the document.

5.3.2 Representing similarity search queries

While we provide relatively big freedom in specification of conditions for querying using exact set of restrictions, we cannot afford to do so with similarity matching. In its case, the component does more work in forming the query on its own. The only thing it allows to specify is the property path leading to possible similarity match together with its significance, optional restriction put upon its value and enriching relations widening the space of values which can be considered to mark the document related to such value as similar to the target document. Also, similarity matches can be put together and form so called similarity groups in order to express more aspects of some similarity dimension. Again, a property path is used to determine the common object of different aspects. In Figure 5.4, we present query from Listing 4.12 represented by means of similarity groups and matches. We can see that any similarity group or match appearing inside of other similarity group can be marked as optional in order to allow the rest of similarity aspects on the same level to be counted even without the co-occurence with the optional one.

There are four ways of restricting attribute values of the target object that should be considered for similarity matching. First two are to simply use a literal or reference restriction objects. Another two lie in associating the match object with a standalone expression. In one case, this standalone expression represents the attribute value and allows its restriction by the attribute-specific relations. Alternatively, the standalone expression represents the document object which is helpful if we have a variable bound to the similarity matched object and we restrict the value of this variable by some other attribute related to the document entity.

When analyzing the similarity search scenarios, we introduced the idea of enrich-

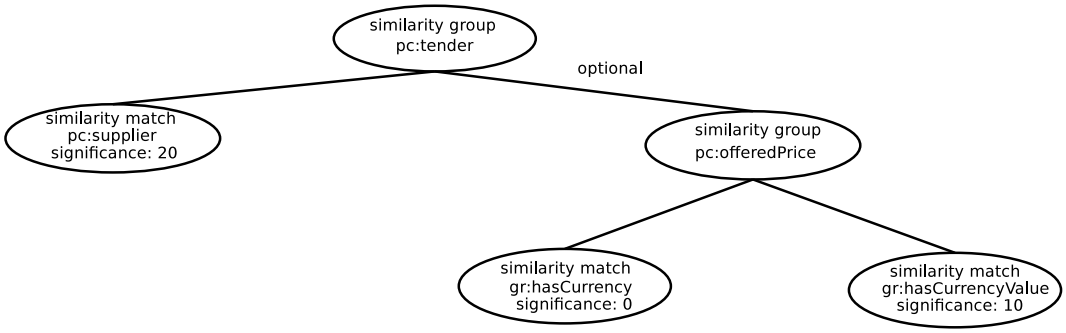


Figure 5.4: Tree representation of the query

ments which widen the space of values that can be considered similar to attribute values of the target object. We presented concept of the range enrichments for literals and relation enrichments for entities. In API, enrichments are represented as objects providing the significance of the similarity encountered thanks to applying it and the means of the enrichment itself. In case of relation enrichment, this is simply property URI of the relation. For range enrichments, it is a special variant of literal restriction which is then applied to the value comparing it implicitly to the corresponding attribute values of the target entities.

Important feature provided by both similarity groups and similarity matches is the ability to define different property paths relating targets and results to their attributes that should be considered for deciding about similarity. This makes it possible to compare two different sets of documents against each other.

5.3.3 The search interface

Once we identified tha abstraction used for representing the queries, we can go on with defining the search interface. Posessing two relatively different strategies, we concluded that it might be useful to be able to combine them. Therefore, we define one method meant for document retrieval: `search`. This method accepts a `SearchContraints` object which contains four collections providing required and optional conditions expressed by exact restrictions and required and optional similarity constraints. The conditions expressed by exact restrictions do so with help of `IStandaloneExpression` interfaces that define an abstraction of group of standalone expressions mentioned in Section 5.3.1. We create the whole variety of interfaces that determine the groups of expressions that can be used at different places in the expression tree. Implementations of these interfaces that can be used in order to build the expression tree are also supplied.

For similarity matching, there is only one `ISimilarityExpression` interface implemented by `SimilarityGroup`, `SimilarityMatch` and its special variants `LiteralSimilarityMatch`, `ReferenceSimilarityMatch` and `RootConstrainedSimilarityMatch` that allow the further restriction and/or enrichment of the matches. The rest of the objects are specific variants of enrichments. The `ISimilarityExpression` objects are wrapped by `Similarity` which also specifies the target URI for which similar documents are to be found.

The `search` method is declared to evaluate each of the included conditions independently combing their results at then and of processing. Therefore, it is possible to use arbitrary combinations of conditions. It is even possible to supply similarity search specifications with different targets in one search. The method also provides two overloading allowing specification of limit and offset of results.

5.4 Auxiliary taxonomy functionality

Last service represented by the `ITaxonomySet` interface provides methods for obtaining basic information about taxonomies such as returning relevant objects or subjects for given relation, instances of intersection of particular types, domains and ranges of properties, types of given entity and properties whose domain or range is subset of a particular collection of classes.

Chapter 6

Survey of available RDF stores

Having determined the exact functionality provided by the component, we can explore the RDF stores available, analyze the features they provide and select the most suitable of them for the implementation. First, we summarize the requirements that are placed on the stores. Then, we go on with describing them with regards to these requirements.

6.1 RDF store requirements

Due to the fact that the component should operate on a dataset formed by several taxonomies that can be updated from time to time, it is desirable to be able to store each of them in its individual graph in order to be able to remove all its triples when needed. Eventually, a new version of the taxonomy can be then uploaded. Therefore the underlying store should provided the ability to deal with named graphs or contexts in terms of which the triples are stored. Stores supporting named graphs are also called quad stores.

We have identified the subset of OWL/RDFS dialect which is suitable for usage with taxonomies. Above all, its features include the ability to reason about hierarchies of classes and properties, transitive, symmetric and inverse relations and identity of entities. Selected stores should support inference based at least on the rules these concepts appear in.

The SPARQL representation of the queries processed by the component heavily rely on new features from SPARQL 1.1 specifications such as aggregates, subqueries, negation, BOUND tests and additional functions. Therefore, the stores should support these features in order to be considered for usage.

There is one non-standard feature that is required: the capability of executing full-text search queries from within SPARQL. These queries should be able to deliver the information about the subject represented by the literal, relating property and the literal itself. Also, the full-text implementation provided by the store should be able to identify prefix matches and matches of at least one word from given set.

Finally, the store should be able to expose the dataset stored by a SPARQL endpoint allowing different clients to be able to access it. This feature is also

handy when experimenting with suitable queries for analyzing the data.

6.2 Investigation

6.2.1 4store

4store is, as its name suggests, RDF quad store which relies on memory caching of storage indices. It is distributed as an open-source standalone SPARQL server together with set of PHP, Phyton, Ruby and Java client libraries. It supports almost all features of SPARQL 1.1 (with exceptions such as the BIND functino) full-text search. It doesn't include any kind of reasoning.

6.2.2 AllegroGraph

AllegroGraph[2] is a native RDF quad store which relies on extensive use of memory caches. It supports a RDFS++ backward chaining inference which is able to deal with all the entailment required. The interfaces available include SPARQL endpoint, integration with Jena and Sesame and even Prolog and LISP clients. Named graphs together with SPARQL 1.1 features are also supported. Full-text indexinf is provided by custom Patricia trie based implementation. However, it is impossible to locate the literal matched as only the subject is returned when using the full-text search from within SPARQL. Such a functinality is only supported by the LISP client. Therefore, AllegroGraph fails to fulfill our requirements.

6.2.3 BigData

BigData[4] is a native store designed in order to be used within a cluster of multiple server nodes. It offers an open-source license, SPARQL endoint functionality, Sesame integration, SPARQL 1.1 functionality (with the exception of property paths), RDFS+ reasoning (working with all the important rules) and full-text indices based on Lucene. Unfortunately, it differentiates between triple and quad storage modes and the quad mode doesn't support reasoning.

6.2.4 Jena TDB and SDB

We already discussed Apache Jena as an RDF processing framework. Now, we concentrate on the possibility of using its data storage components TDB or SDB as RDF stores for our component. SDB is basically a data storage layer which operates on top of some relational database. Several database systems and storage layouts are supported. TDB is a native RDF storage storing its indexes mostly as B+Trees in the file-system with different memory caching approaches applied with 64-bit and 32-bit version respectively. Both TDB and SDB don't provide any additional features on their own. Instead, they rely on the Jena platform when it comes to support for reasoning and other features. As already discussed, Jena supports use of full-text search with the LARQ extension to its ARQ SPARQL querying component which in turn provides complete set of SPARQL 1.1 features.

Many reasoners are included with the system implementing variety of OWL

dialects and basic RDFS reasoning. Issue with using them is that they are mostly forward chaining, i.e. they pre-compute the infered triples when instantiated. Holding all of the infered triples in memory may lead to problems with bigger datasets. Also, though TDB and SDB support named graphs, the reasoner operates as a special graph on top of only one graph (or a couple of ontology and data graphs). Default graph as union of all the graphs (if it is configured so), can be passed to the reasoner in order to overcome the problem. However, once this is done all changes must be propagated through the reasoner layer in order to be taken into account while Jena is running. By doing so, the information about the named graphs is simply lost. There is also a pair of general rule based reasoning engines consisting of one forward chaining and one backward chaining reasoner. The backward chaining engine doesn't suffer from the issue described above because the updates don't need to be propagated through its representation.

Finally, SPARQL endpoint on top of these stores can be created with help of Fuseki which embeds functionality of all the Jena components.

6.2.5 Parliament

Parliament[56] is a traditional triple store which uses memory-mapped index files for efficient triple retrieval. It also employs B-tree indices of embedded BerkeleyDB[15] for storing the resource URI-to-ID mapping. It supports RDFS reasoning mixed with subset of OWL Horst which satisfy all required features. Its querying capabilities is realised via Jena and Sesame which can be usedin order to add support to additional functionality such as named graphs or full-text search. The last version is only supplied with rather obsolete versions of Jena/Sesame leaving out the important implementation of SPARQL 1.1.

6.2.6 Sesame

Sesame is mostly known as a library for working with RDF data. It also includes three implementations of its repository abstraction: memory store (simply dumping the data to the file system in order to persist it between separate runs), native store and RDBMS backend storage. SPARQL 1.1 functionality is supported and by deploying Sesame in any Servlet container, we can create a SPARQL endpoint. The full-text search is provided by the third-party uSeekM[38] repository that can be stacked on top of any of the included store implementations. uSeekM exploits PostgreSQL[22] full-text search capabilities. Nevertheless, only RDFS reasoning is provided for memory and native store while no reasoning is available for the RDBMS one.

6.2.7 OWLIM

OWLIM is an RDF store based on the Sesame platform. It offers efficient native storage together with many reasoning profiles such as RDFS, OWL Horst, OWL 2 RL, OWL 2 QL and rule based generic reasoner. The reasoners are forward chaining. Two variants of full-text search are included. One based on proprietary

implementation. The other one based on Lucene. It supports the concept of contexts in order to preserve the information about the origin of the triples. Finally, full support for SPARQL 1.1 id provided and thanks to the Sesame servlet component, it is possible to deploy OWLIM as a SPARQL endpoint.

6.2.8 Stardog

Stardog is a relatively young implementation of RDF store which comes with rich set of features. All OWL 2 profiles together with basic RDFS inference are supported for reasoning purposes. Full-text functionality based on Lucene is also provided and support for all features of SPARQL 1.1 was added recently. Stardog is implemented in Java and it is deployed as a standalone server behaving as a SPARQL endpoint. Additionaly, communication based on proprietary SNARL protocol is available to programmers.

6.2.9 Virtuoso

Virtuoso is a compact RDBMS offering vast amount of various functionality. Among other features, it provides a RDF quad store which uses compressed bitmap indices together with their efficient in-memory buffering for native storage of RDF data as well as translational layer for accessing data stored in tables. However, this layer doesn't offer all the features of the native alternative. Virtuoso supports full-text search in SPARQL queries fulfilling all the needs of the component. HTTP server with SPARQL endpoint functionality available is included. Providers for Jena, Sesame and Redland frameworks are present it is also possible to exploit the ODBC and JDBC drivers for SPARQL queries.

Most of the features from SPARQL 1.1 are offered. However, not all of them are implemented such as BIND and VALUES. Virtuoso also has its own syntax for property paths and transitive closure computation. Finally, it supports reasoning based on backward chaining approach where infered triples are computed during the evaluation of query with help of rule set precomputed from the information stored in ontologies. All the desired reasoning capabilities are enabled. Virtuoso is offered in two editions: Virtuoso Universal Server and Virtuoso OpenSource which lacks the support for clustering and data replication.

Chapter 7

Implementation and evaluation

We implemented the component on top of two stores selected from those presented in the previous chapter. We chose Virtuoso and OWLIM as they seem to provide the most complete set of features, be widely-used even by enterprise applications and perform well in benchmarks. Stardog also seemed to be an appropriate choice. Nevertheless, its implementation of SPARQL 1.1 was completed too late for being considered for evaluation in this work. Another feature-complete systems: Jena TDB and SDB are reported to be outperformed by the stores we select.

7.1 Implementation considerations

The biggest challenge of the implementation was how to process the structure used to represent the semantic queries. After thorough analysis, we decided to process this structure with help of visitor pattern. The classes that form the structure define accepting methods for our query processing objects. The structure is than scanned in the depth-first-search manner.

Both stores are communicated with based on similar approach. Virtuoso is accesed via the JDBC protocol and OWLIM through the Sesame HTTP extension of SPARQL. We define methods and structures that encapsulate the reuslt set processing in order not to have to deal with the technical details of the individual solutions such as Exceptions direclty within the service objects.

Service objects are instantiated with help of factories in order to keep the configuration machinery isolated in one place.

7.2 Evaluation

We used a sample RDF dataset extracted from notices about public contracts in order to define suitable queries for the evaluation of the stores. This dataset is also linked to NUTS and CPV taxonomies allowing us to experiment with taxonmic queries and reasoning. For evaluation of full-text search, we used a text samples describing different enginnering fields and similar topics promissing bigger amount of matches in the CPV taxonomy from. OWLIM finished all the test without any problems. Virtuoso doesn't support the `BIND` pattern from

SPARQL 1.1 so specific type of expressions used in quieries such as enrichments or binding a variable to node modified by some function failed.

Chapter 8

Conclusion and future work

In this work, we have presented an approach of using modern Semantic Web technologies in order to enhance retrieval of relevant documents. Thanks to analyzing a real world dataset, we have identified basic use cases and expanded them in order to ensure general applicability. We discussed possibilities of their implementation using SPARQL querying language operating on set of RDF data extracted from the documents.

Thanks to identifying patterns occuring in the SPARQL queries, we were able to propose a component with original API for encapsulating the work with underlying data representation such as an RDF store. The API was designed in order to be used by apllications working within arbitrary domain and to allow convenient representation of complicated SPARQL queries without the need of dealing with unnecessary aspects such as intermediate variables which only connect different patterns within queries. The representation of queries is targeted to document-oriented approach based on facet driven selection of important attributes. The API also provides means to expressing search focusing on determining similarities between documents.

We implemented this copmponent on top of two widely-used RDF storage systems: Virtuoso and OWLIM. We also evaluated the stores by executing different queries on top of each other.

8.1 Further steps

In the scope of this document we focused on using existing solutions for the underlying data storage. Nevertheless, there is an ongoing progress in implementing efficient RDF storage solutions within the academic community. Interesting models of representing RDF data, such as RDFVector[58] built experimentally on top MonetDB[54], are proposed. Therefore, we suggest that the research initiated by this work is followed by exmperiments with custom solutions for the underlying storage.

The full-text features provided by existing RDF stores aren't sufficient in some cases. Therefore, it also might be helpful to employ some mature system specialized on full-text retrieval, such as Lucene, and incorporating it to be used side

by side with the RDF-oriented data storage.

Finally, we propose that some other aspects od the data extracted from the documents, such as the analytical information that could be obtained by queries inspired by OLAP solutions that use aggregations over different attributes of the documents, is explored and maybe added as an additional funcitonality to our component.

Bibliography

[1] ActiveRDF, http://activerdf.org/.

[2] AllegroGraph, http://www.franz.com/agraph/allegrograph.

[3] Apache Jetty, http://jetty.codehaus.org/jetty/.

[4] . Bigdata, http://www.systap.com/bigdata.htm.

[5] CPV Codes, http://simap.europa.eu/codes-and-nomenclatures/codes-cpv/.

[6] dotNetRDF, http://www.dotnetrdf.org/.

[7] GATE Framework, http://gate.ac.uk/.

[8] GeoNames, http://www.geonames.org/.

[9] Jena Framework, http://jena.apache.org/.

[10] LINQ, http://msdn.microsoft.com/en-us/library/vstudio/bb397926.aspx.

[11] LinqToRdf, http://code.google.com/p/linqtordf/.

[12] Lucene, http://lucene.apache.org/core/.

[13] MusicBrainz, http://musicbrainz.org/.

[14] NUTS: Nomenclature of territorial units for statistics, http://ec.europa.eu/eurostat/ramon/rdfdata/.

[15] Oracle Berkeley DB, http://www.oracle.com/technetwork/products/berkeleydb/overview/index.html.

[16] ORDF Python Library http://ordf.org/.

[17] Oroboro, http://code.google.com/p/oroboro/.

[18] OWL Standard, http://www.w3.org/TR/owl-features/.

[19] OWL 2 Standard, http://www.w3.org/TR/owl2-overview/.

[20] Pellet: OWL 2 Reasoner, http://clarkparsia.com/pellet/.

[21] Pelorus, http://clarkparsia.com/pelorus.

[22] PostgreSQL Database, http://www.postgresql.org/.

[23] RDF Standard, http://www.w3.org/RDF/.

[24] RDFa Standard, http://www.w3.org/TR/xhtml-rdfa-primer/.

[25] RDFLib Library, http://code.google.com/p/rdflib/.

[26] RDFS Standard, http://www.w3.org/TR/rdf-schema/.

[27] RDF/XML Standard, http://www.w3.org/TR/REC-rdf-syntax/.

[28] Redland Library, http://librdf.org/.

[29] RFC-3066: Tags for the Identification of Languages, http://www.ietf.org/rfc/rfc3066.txt.

[30] RFC-3986: Uniform Resource Identifier (URI): Generic Syntax, http://www.ietf.org/rfc/rfc3986.txt.

[31] SemWeb framework, http://razor.occams.info/code/semweb/.

[32] SKOS Standard, http://www.w3.org/2004/02/skos/.

[33] Spanner: From Documents to Linked Data Apps, http://weblog.clarkparsia.com/2010/12/14/introducing-spanner/.

[34] Sparta Framework, http://www.mnot.net/sw/sparta/.

[35] Stardog: The RDF Database, http://stardog.com/.

[36] SuRF, Object RDF Mapper, http://packages.python.org/SuRF/.

[37] Turtle - Terse RDF Triple Language, http://www.w3.org/TeamSubmission/turtle/.

[38] uSeekM, https://dev.opensahara.com/projects/useekm.

[39] XML Standard, http://www.w3.org/XML/.

[40] XML Schema Standard, http://www.w3.org/XML/Schema.html.

[41] World Wide Web Consortium, http://www.w3.org/.

[42] Wikipedia, the free encyklopedia, http://en.wikipedia.org/.

[43] Zope Object Database, http://www.zodb.org/.

[44] A. V. Aho, M. J. Corasick. Efficient string matching: an aid to bibliographic search. Magazine Communications of the ACM. Volume 18 Issue 6, June 1975 Pages 333-340.

[45] S. J. Athenikos, X. Lin. Enabling Type/Condition-Specified Entity/Fact Retrieval Using Semantic Knowledge Extracted from Wikipedia. SMER '11 Proceedings of the 1st international workshop on Search and mining entity-relationship data Pages 15-20. 2011.

[46] S. Auer C. Bizer, G. Kobilarov, J. Lehmann, R. Cyganiak, Z. Ives. DBpedia: A Nucleus for a Web of Open Data. 6th International Semantic Web Conference, 2nd Asian Semantic Web Conference, ISWC 2007 + ASWC 2007, Busan, Korea, November 11-15, 2007. Proceedings.

[47] T. Berners-Lee, J. Hendler, O. Lassila, Semantic Web. Scientific American 284, 5 (2001) 34–43.

[48] S. Blott, R. Weber. A simple vector-approximation file for similarity search in high-dimensional vector spaces. 1997.

[49] J. Broekstra, A. Kampman. SeRQL: a second generation RDF query language, 2003.

[50] J. Broekstra, A. Kampman, F. van Harmelen. Sesame: A Generic Architecture for Storing and Querying RDF and RDF Schema, First International Semantic Web Conference Sardinia, Italy, June 9–12, 2002 Proceedings.

[51] O. Erling, I. Mikhailov. RDF Support in the Virtuoso DBMS. Studies in Computational Intelligence, 2009, Volume 221/2009, pages 7-24.

[52] R. Hahn, C. Bizer, C. Sahnwaldt, C. Herta, S. Robinson, M. Bürgle, H. Düwiger, U. Scheel. Faceted Wikipedia Search. Business Information Systems Lecture Notes in Business Information Processing Volume 47, 2010, pp 1-11.

[53] H. J. Horst: Combining RDF and Part of OWL with Rules: Semantics, Decidability, Complexity. 4th International Semantic Web Conference, ISWC 2005, Galway, Ireland, November 6-10, 2005. Proceedings.

[54] S. Idreos, F. Groffen, N. Nes, S. Manegold, S. Mullender, M. Kersten. MonetDB: Two Decades of Research in Column-oriented

[55] J. Klímek, T. Knap, J. Mynarz, M. Nečaský, V. Svátek, Framework for Creating Linked Data in the Domain of Public Sector Contracts, 2012. Database Architectures. 2012.

[56] D. Kolas, I. Emmons, M. Dean. Efficient Linked-List RDF Indexing in Parliament. SSWS, 2009.

[57] A. Kyriakov, D. Ognyanov, D. Manov. OWLIM – A Pragmatic Semantic Repository for OWL. Lecture Notes in Computer Science Volume 3807, 2005, pp 182-192.

[58] J. P. McGlothlin. RDFVector: An Efficient and Scalable Schema for Semantic Web Knowledge Bases. Proceedings of the PhD Symposium ESWC, 2010.

[59] P. N. Mendes, M. Jakob, A. García-Silva, C. Bizer. DBpedia spotlight: shedding light on the web of documents. Proceedings of the 7th International Conference on Semantic Systems Pages 1-8. 2011.

[60] G. A. Miller, WordNet: A Lexical Database for English, 1995.

[61] S.P. Ponzetto, M. Strube. WikiTaxonomy: A large scale knowledge resource. Journal of Artificial Intelligence Research, 2007.

[62] B. Popov, A. Kiryakov, A. Kirilov, D. Manov, D. Ognyanoff, M. Goranov. KIM – Semantic Annotation Platform. In the proceedings of The Semantic Web - ISWC 2003.

[63] F. M. Suchanek, G. Kasneci, G. Weikum. YAGO: A Core of Semantic Knowledge Unifying WordNet and Wikipedia.

[64] J. Weaver, Redefining the RDFS Closure to be Decidable, 2009.

[65] L. Yujian, L. Li : A Normalized Levenshtein Distance Metric, 2007.

[66] B. Zapilko, J. Schaible, P. Mayr, B. Mathiak. TheSoz: A SKOS Representation of the Thesaurus for the Social Sciences. 2012.

Appendix A

Contents of the DVD-ROM enclosed

This thesis is accompanied by the DVD-ROM containing source code of the component prototype implementation and data used for the evaluation. The DVD-ROM is organized as follows:

The source code of the component API and its implementation is located in directory `/src`. Libraries needed in order to build it are available in directory `/lib`. Finally, directory `/data` contains the sample dataset we used for testing consisting of the public contracts representation and taxonomies of CPV codes and NUTS.