LDBC

Cooperative Project

FP7 – 317548

D4.4.2 Benchmark Design for Reasoning

Coordinator: [Vassilis Papakonstantinou, Irini Fundulaki]

With contributions from: [Giorgos Flouris (FORTH), Vladimir Alexiev (ONTO)]

1st Quality Reviewer: Peter Boncz (UvA)
2nd Quality Reviewer: Thomas Neumann (TUM)

Deliverable nature:	Report (R)
Dissemination level: (Confidentiality)	Public (PU)
Contractual delivery date:	M24
Actual delivery date:	M24
Version:	1.0
Total number of pages:	92
Keywords:	Linked Open Data, RDF, RDFS, OWL reasoning

Abstract

Reasoning (mainly OWL reasoning) has received increasing attention by ontology designers for more accurately representing the domain at hand. To reflect this importance, one of LDBC's objectives is to identify a set of interesting use cases that consider OWL reasoning constructs (beyond the usual RDFS constructs) that can be used to challenge existing RDF engines or repositories. This Deliverable has two parts: in the first part, we present four different sets of queries that can be used to determine whether RDF query engines take into account OWL constructs during query plan construction or query execution; in the second part we consider how a repository or query engine incorporates and considers business rules, i.e., domain-specific rules that follow common templates, useful in practical applications.

EXECUTIVE SUMMARY

Reasoning (mainly OWL reasoning) has received increasing attention by ontology designers for more accurately representing the domain at hand. To reflect this importance, one of LDBC's objectives is to identify a set of interesting queries that consider OWL reasoning constructs (beyond the usual RDFS constructs) that can be used to test the extent to which these can be used by existing SPARQL query engines for query plan construction and query execution.

In the first part of this Deliverable we propose sets of tests that can be considered as the baseline for any reasoning-aware query engine as they check whether the engine correctly takes into account the reasoning constructs during query execution. Our main motivation behind this effort is the argument that query plans can be optimized using heuristics based on OWL reasoning constructs and constraints, and that such optimizations can be employed in a complementary fashion to traditional optimizations used by the query planners to further improve their performance. The four different sets of such queries are:

- *conformance tests* that examine whether an RDF engine correctly implements the semantics of OWL reasoning constructs; note that some of these tests have been included in LDBC Deliverable D4.4.1 [28] as well.
- *static tests* that can be employed to check whether the engine is able at compile time to take advantage of constraints such as class and property *disjointness* and *equivalence*, in addition to *functional* as well as *domain* and *range* property constraints. For instance, a query looking for an instance of two disjoint classes is certain to return no answers, so it should be answerable in constant time, without having the engine to evaluate the query (under the assumption that the dataset satisfies the schema).
- *selectivity tests* that determine how well the query planner can guess the selectivity of certain joins. For this task, query engines have traditionally used statistics or heuristics related to the form of the query that guide them in building the optimal query plans. In this Deliverable, we argue that such heuristics could be combined with schema information via reasoning; for instance, cardinality constraints may provide hints for the cardinality of certain relations. Such tests can be viewed also as suggestions for possible reasoning-based selectivity heuristics that a query planner could employ.
- *advanced reasoning tests* that comprise of various types of advanced tests that highlight cases where reasoning can help the query engine determine the optimal query plan using schema information. These tests are more sophisticated than the previous ones and focus on the incorporation of reasoning constructs for the creation of the query plans.

For the aforementioned tests we discuss how each proposed construct can be used by forward and backward reasoners since most benchmarks are suitable for both types; there is an explicit mention when this is not the case. All queries are presented in an abstract form, showing the pattern being tested; subsequently, they are grounded in the BBC ontology of the Semantic Publishing benchmark, so as to be directly applicable for use in the said benchmark.

In the second part of this Deliverable we also present several cases that are interesting for practical applications, but for which using standard OWL2 constructs are not sufficient because OWL2 lacks either expressive power (e.g. the ability to define conjunctive properties) or appropriate data structures. For such tasks we use domain-specific business rules or vendor-specific extensions. In some instances the same functionality can be captured with complex SPARQL queries, but most often this leads to very complex queries that do not have reasonable execution times.

DOCUMENT INFORMATION

IST Project Number	FP7 – 317548	Acronym	LDBC
Full Title	LDBC		
Project URL	http://www.ldbc.eu/		
Document URL	`http://wiki.ldbcouncil.org/display/PROJECT/Deliverables`		
EU Project Officer	Carola Carstens		

Deliverable	Number	D4.4.2	Title	Benchmark Design for Reasoning
Work Package	Number	WP4	Title	Semantic Choke Point Analysis

Date of Delivery	Contractual	M24	Actual	M24
Status	version 1.0		final ☒	
Nature	Report (R) ☒ Prototype (P) ☐ Demonstrator (D) ☐ Other (O) ☐			
Dissemination Level	Public (PU) ☒ Restricted to group (RE) ☐ Restricted to programme (PP) ☐ Consortium (CO) ☐			

Authors (Partner)	Irini Fundulaki, Giorgos Flouris (FORTH), Vladimir Alexiev (ONTO)			
Responsible Author	Name	Irini Fundulaki, Vassilis Papakonstantinou	E-mail	fundul@ics.forth.gr, papv@ics.forth.gr
	Partner	FORTH	Phone	+302810391725

Abstract (for dissemination)	Reasoning (mainly OWL reasoning) has received increasing attention by ontology designers for more accurately representing the domain at hand. To reflect this importance, one of LDBC's objectives is to identify a set of interesting use cases that consider OWL reasoning constructs (beyond the usual RDFS constructs) that can be used to challenge existing RDF engines or repositories. This Deliverable has two parts: in the first part, we present four different sets of queries that can be used to determine whether RDF query engines take into account OWL constructs during query plan construction or query execution; in the second part we consider how a repository or query engine incorporates and considers business rules, i.e., domain-specific rules that follow common templates, useful in practical applications.
Keywords	Linked Open Data, RDF, RDFS, OWL reasoning

Version Log			
Issue Date	Rev. No.	Author	Change
16/09/2014	0.1	Vassilis Papakonstantinou, Irini Fundulaki	First version
26/09/2014	0.2	Irini Fundulaki	Second version
30/09/2014	1.0	Irini Fundulaki	Final version

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

1 INTRODUCTION

Ontologies are increasingly being used by ontology designers for providing a more accurate description of the domain at hand. The de facto language for expressing rich ontologies is OWL [23], with its various "flavours", i.e., different fragments that take a different stance in the trade-off between high expressive power and low computational complexity. OWL is a language built on top of RDF [25] and RDFS [12].

OWL constructs support the modeling of implicit along with explicit information obtained through *reasoning* which is a very important compoment of ontologies; those constructs support the modelling of *schema information* such as *typing* (definition of classes, properties) as well as *constraints* at the schema and data level.

Despite the importance of reasoning, and the work performed at the theoretical level in defining languages for reasoning, query engines have generally not followed at the same pace, and, in fact, many of them do not fully support reasoning (see also LDBC Deliverable D4.4.1 [28]).

As a response to this fact, one of LDBC's objectives is to propose a set of tests/queries that measure the extent to which reasoning-aware SPARQL query engines consider reasoning constructs for *query answering*; this Deliverable is the first attempt towards this goal which contains four different sets of queries, testing different ways in which a *forward* or *backward* RDF reasoner can exploit these constructs.

Note that, unlike other benchmarks proposed in the literature, [11, 30, 40, 79], the objective of the queries presented in this work is not to stress the reasoner of the query engine into performing complex forms of reasoning with large amounts of data, i.e., our intention is not to provide standard workload benchmarks; instead, our objective is to see how schema information could be exploited for query answering.

More specifically, our objective is two-fold: first, we want to determine whether reasoning-aware query engines correctly consider the semantics of the various OWL constructs; second, we want to test whether the query engine uses schema information expressed through OWL constructs to perform interesting optimizations of possibly increasing complexity, in order to improve the query execution plans, and, consequently, the performance of the engine. Our main motivation behind this effort is the argument that query plans can be optimized using heuristics based on the schema, and the corresponding OWL reasoning rules, and that such optimizations can be employed in a complementary fashion to traditional optimizations used in query planners to further improve their performance.

Acknowledging the fact that forward and backward reasoners employ different strategies for performing reasoning, our sets of test queries have been designed to consider both types of reasoners. In fact, most of the queries are suitable for both types of reasoners. For the queries for which this is not the case, we explicitly mention this. All the proposed queries are presented in an abstract form, showing the pattern being tested using abstract URI constants.

The four different sets of queries that we propose are:

- *conformance tests* that examine whether an RDF engine correctly implements the semantics of OWL reasoning constructs; note that some of these tests are also used in LDBC Deliverable D4.4.1 [28] as well. These tests are necessary to verify that the reasoning engine is sound and complete with respect to reasoning and inference, and are considered to be the baseline test for any reasoning-aware query engine; for a query engine to pass this test, it should take into account both the explicit and the implicit information while running a query.
- *static tests* that can be employed to check whether the engine is able, at compile time, to take advantage of constraints such as class and property disjointness and equivalence, in addition to functional as well as domain and range property constraints. For instance, a query looking for an instance of two disjoint classes is certain to return no answers, so it should be answerable in constant time (under the assumption that the dataset satisfies the schema).
- *selectivity* tests that determine how well the query planner can guess the selectivity of certain joins. For this task, query engines have traditionally used statistics or heuristics related to the form of the query that guide them in building optimal query plans. In this Deliverable, we argue that such heuristics could be combined with schema information via reasoning; for instance, cardinality constraints may provide hints for the cardinality of certain relations. Such tests can be viewed also as suggestions for possible reasoning-based selectivity heuristics that a query planner could employ.

- *advanced* reasoning tests that comprise of various types of advanced tests that highlight cases where reasoning can help the query engine determine the optimal query plan using schema information. These tests are more sophisticated than the previous ones and determine whether the query engine incorporates advanced reasoning semantics while generating query execution plans.

Finally, in this Deliverable we consider how a repository or query engine incorporates and considers business rules, i.e., domain-specific rules that follow common templates, useful in practical applications. We present several cases that are interesting for practical applications, but for which using standard OWL2 [77] constructs are not sufficient. OWL2 reasoning often has high complexity, which is the reason for defining different profiles/dialects (Full, DL, RL, EL, QL) that trade off expressivity for tractability and speed of reasoning. Nevertheless, there are tasks for which OWL2 is not appropriate, because it lacks either expressive power (e.g. the ability to define conjunctive properties) or appropriate data structures. For such tasks we use domain-specific business rules or vendor-specific extensions. In some instances the same functionality can be captured with complex SPARQL queries, but most often this leads to very complex queries that do not execute in a reasonable time.

The cases that we consider include:

- *compressing* a complex network of relations by computing *search index* relations over them (so-called Fundamental Relations), using custom rules
- *extended property constructs* for more efficient property chains and transitive properties, using custom rules
- *exposing ambiguous entities*, *classifying* and *validating works*, using SPIN rules (similar to SPARQL Update)
- *faceted search*, by connecting to external faceting engine (Solr)
- *geospatial queries*, using custom extensions

Structure Chapter 2 provides an overview of the related work; in Chapter 3 we discuss the main concepts of the OWL language that we are going to use in this work. Chapters 4 discusses the *conformance tests*; *static tests* are presented in Chapter 5. The *selectivity* and *advanced tests* are given in Chapters 6 and 7 respectively. Chapter 8 discusses interesting use cases for which OWL2 constructs are not appropriate. The SPB specific queries for the conformance, static, selectivity and advanced tests are given in the Appendix.

2 BENCHMARKING RDF DATABASES

Benchmarking RDF systems presents different challenges than the ones posed in relational database engines. Consequently, existing relational benchmarks are not really suitable for RDF benchmarking. This is due to the intricacies of RDF data, which are expressed in the simple RDF data model that is based on the representation of information in the form of a triple *(subject, predicate, object)*. Nowadays, a large number of ontologies have been developed in order to represent information in various diverse domains of interest. The RDFS [12] and OWL [23] Semantic Web languages have been used to represent the concepts, relationships and constraints in the domain of study.

The recent increase in the number of real-world applications that use RDF data, is raising the need for more interesting benchmarks that should test all aspects of RDF query processing. As a result, in the last few years we have witnessed the development of a number of different RDF benchmarks. Existing, well-established, relational benchmarks have been used for testing the performance of RDF engines. TPC-H [67] can be translated into RDF, but the queries and especially the data still remain relational at heart. In this type of benchmarks, *schema information* expressed in terms of rich OWL constructs is not considered, and consequently OWL-based reasoning is basically absent. Reasoning is essential in benchmarking RDF query engines due to the increasing number of rich schemas that are used in the Linked Data Cloud.

In this Chapter we will make an overview of existing benchmarks for RDF engines focusing more on the aspect of the incorporation of OWL reasoning constructs for each of the benchmarks and how those have been used for performing query answering. A detailed presentation of the state of the art RDF benchmarks has been provided in LDBC Deliverable D1.1.1 [27].

A number of RDF benchmarks that use real datasets have been proposed over the last years [53, 59, 71, 64, 54]. DBPedia [63] extracts structured information from Wikipedia [69] to create a large dataset for benchmarking. The DBPedia ontology is a shallow ontology that contains core and domain specific concepts; the ontology contains also a small number of RDFS properties (`rdfs:subClassOf`, `rdfs:label`, `rdf:type`, `rdfs:comment`) and the OWL property `owl:sameAs` that links resources which refer to the same object but originate from different datasets. Although the DBPedia dataset is one of the reference datasets for Linked Open Data, there is no clear set of representative queries for it. The University of Leipzig [44] developed the DBPSB [64] (DBPedia SPARQL Benchmark) benchmark, which included a query workload derived from the DBPedia query logs. The produced query workload consists of mostly simple lookups and does not consider more complex OWL or RDFS constructs. UniProt KnowledgeBase (UniProtKB) [54] is a high-quality dataset expressed in RDF, describing protein sequences and related functional information. It uses an OWL ontology expressed in a sub-language of OWL-Lite (more expressive than OWL Horst), but still tractable. The UniProt queries are mainly lookup queries [68] but some are also used to test the reasoning capabilities of RDF databases (i.e., *taxonomic queries* along the RDFS *subclass* hierarchy). YAGO [71] is a vast knowledge base that follows the steps of DBPedia and integrates statements from Wikipedia, Wordnet, WordNet Domains, Universal WordNet and GeoNames [65]. The original dataset is not accompanied by a set of queries (as in the case of UniProtKB). However, Neumann et. al. provided eight queries for an earlier version of the YAGO ontology, for benchmarking the RDF-3X engine [50]. These queries are mostly lookup and join queries but none of those makes use of RDFS constructs.
In general, the aforementioned benchmarks do not take into account interesting reasoning constructs in the proposed query workload, hence query engines that use those benchmarks focus mostly on testing how well the engines perform for the proposed query mixes and for datasets of various sizes.

In addition to the previous benchmarks, a number of *synthetic* ones have been developed for testing RDF engines. The Lehigh University Benchmark (LUBM) [66] is intended to evaluate the performance of Semantic Web repositories over a *large data set* that adheres to a university domain ontology. The Univ-Bench ontology used in LUBM is a relatively small ontology consisting of 43 classes and 32 properties, and is expressed in OWL-Lite, the simplest language of OWL [43, 23]. In addition to the user-defined properties, the ontology includes OWL specific properties such as `owl:TransitiveProperty`, `owl:someValuesFrom` restrictions, and `owl:intersectionOf`.

The LUBM benchmark consists of *fourteen* mainly extensional lookup and join queries, that consider very simple *reasoning* focusing on `rdfs:subClassOf`, `rdfs:subPropertyOf` and `owl:sameAs` properties. As a standard benchmark, the LUBM itself has several limitations. First, it covers only part of the inference supported by OWL Lite and OWL DL, so it cannot exactly and completely evaluate an ontology system in terms of inference capability. Second, it only supports *extensional queries*, that is queries about instance data. Third, the ontologies used in LUBM are of *moderate* size, so they are not stressing the tested systems enough. The University Ontology Benchmark (UOBM) [41] extends LUBM in order to tackle complex inference, as well as scalability issues. In contrast with LUBM, UOBM uses both OWL Lite and OWL DL ontologies and covers most of the constructs of these two sublanguages of OWL (namely, `owl:sameAs` from OWL Lite, and `owl:disjointWith`, `owl:oneOf` and `owl:equivalentClass` from OWL DL). UOBM ontology contains more classes and properties than LUBM and can generate a larger number of instances. UOBM queries are designed based on two principles: (a) queries should consider *multiple lookups* and *complex joins*, and (b) each query should support *at least one different type of OWL inference*.
SP2Bench [56] and the Berlin SPARQL Benchmark (BSBM) [10, 60] are the benchmarks mostly used for testing the performance of RDF engines but none includes queries that test the performance of RDFS or OWL reasoning constructs. SP2Bench contains both a data generator and a set of queries. The data generator builds upon DBLP [62] bibliographic schema, which is a simple schema using 8 classes and 22 properties; The produced data is mostly relational-like and hence do not exhibit the intrinsic properties related to real-world RDF data. The workload consists of fourteen queries with different characteristics such as selectivity, query and output size, and different types of joins and employ the main SPARQL 1.0 operators. BSBM is built around an e-commerce use case where the schema models the relationships between products and product reviews. BSBM Version 3.1 [61] comes with a data generator and a test driver, as well as a set of queries that measure the performance of RDF engines for very large datasets, but do not test the ability of the RDF engines to perform complex reasoning tasks (i.e., RDFS and OWL inference).

A lot of work has been conducted on defining benchmarks for evaluating and comparing OWL reasoners; most of the benchmarks focus on testing the *performance*, *correctness* and *completeness* of systems.
Weithöner et. al. [79] formulated a set of general benchmarking requirements that will be helpful when designing OWL reasoners. These requirements mainly focused on the *changes* that happen to the *schema* (ABox) and are directly related to the performance of a reasoner in performing *inference*.

A framework for an automated comparison of OWL-DL reasoners was presented in [30] that has been developed for testing reasoners with real-life ontologies; this framework allows users to compare the ontologies as well as to check their *correctness* and *completeness*. Their system evaluated on top of four reasoners: FaCT++ [72], KAON2 [46], Pellet [58] and RacerPro [31]. The authors in [11] intend to provide guidance for choosing the appropriate reasoner for a given application scenario; they defined different sets of tests for OWL reasoners by first analyzing the ontology landscape and for each of the OWL ontology subfragments (RDFS, OWL Lite, OWL DLP and OWL DL) a representative ontology (VICODI[1], LUBM [66], SWRC[2], Wine[3]) is chosen. The authors reported on loading time (that also includes checking ontology consistency) as well as response times to sets of predefined tasks. The reasoners evaluated with the above tests were grouped into three categories according to their underlying reasoning techniques: tableau-based algorithms (HermiT [57], RacerPro and Pellet), datalog engines (KAON2), standard rule engines (Sesame [13] and OWLIM [37]). In [40] the authors report their experience and interesting findings in the course of selecting the optimal OWL reasoner when developing a specific ontology-based application. They employ a benchmark suite for large schemas, as well as a selection of small but difficult sets of instances and schemas. Through these tests they analyze the correctness of the results of FaCT++, Pellet, RacerPro, KAON2 and HermiT reasoners. The performance of several services and communication protocols compared in different computing environments leads to the conclusion that these largely underrated components may have a high impact on the overall performance of the systems.

[1] VICODI: `http://www.vicodi.org/about.htm`
[2] SWRC: `http://ontoware.org/swrc/`
[3] `http://www.w3.org/TR/owl-guide/wine.rdf`

It is obvious from the discussion above, that the existing benchmarks focus only on the OWL reasoners' *correctness*, *completeness* and *performance*. Moreover, the benchmarks that have been developed for testing the performance of RDF query engines so far consider a very limited, unexpressive set of OWL constructs mostly limited to testing the efficiency of query engines for query answering. The benchmark queries are not built for testing the capacity of the query engines regarding the incorporation of *schema information* expressed by means of OWL constructs and constraints into producing close to optimal plans.

Existing RDF native query engines [32, 8, 48, 49, 50, 78, 76, 24] have proposed *indexes* for RDF triples that are used during SPARQL query processing whereas SQL-based ones proposed different logical schemas (*triple* and *property* tables) and rely on optimization techiques of the underlying DBMS to evaluate SPARQL queries. Moreover, other works have discussed query graph models [33] and the use of algebraic rewritings for SPARQL queries [70]. SQL-based SPARQL query engines [2, 3, 14, 39] use large triple tables as well as standard indexes on the columns of the large triple table, or property tables. Query processing is done by translating the SPARQL query into its equivalent SQL which is subsequently evaluated by the underlying query engine.

Some native RDF as well as SQL-based query engines are based on *cardinality estimation techniques* [49, 47] for RDF data that can be used to enhance existing SQL optimizers for supporting efficient SPARQL processing. Tsialiamanis et. al [73] discuss *heuristics-based* query optimisation techniques for SPARQL query optimizers that explore the *syntactic* and the *structural* variations of the triple patterns in a SPARQL query in order to choose an execution plan without the need of any cost model. Currently though, existing engines focus on processing RDF queries by considering mainly statistics about the data and completely ignoring *schema information* that is expressed in terms of rich OWL ontologies that accompany, in an increasing number of cases, the RDF data. An interesting line of research is the use of this type of information for query plan construction. Towards this objective a first step is to understand how these constructs could be used by the query engines for devising better query plans; the second step is to write a set of tests (in the form of queries) that could help query optimisers into checking that the produced query plans take into consideration the aforementioned constructs.

3 PRELIMINARIES

The objective of the Semantic Web is to build an infrastructure of machine-readable semantics for data on the Web. The Resource Description Framework (RDF) [25] enables the encoding, exchange, and reuse of structured data, while providing the means for publishing both human-readable and machine-processable vocabularies.

The popularity of the RDF data model and RDF Schema language (RDFS) [12] is due to the flexible and extensible representation of information, independently of the existence or absence of a schema, under the form of *triples*. A triple is of the form *(subject, predicate, object)* where the *predicate* (also called property) denotes the *relationship* between *subject* and *object*. An RDF triple asserts the fact that *subject* is associated with *object* through *property*. An *RDF graph is a set of triples* that can be viewed as a *node* and *edge labeled directed graph* with subjects and objects of triples being the nodes of the graph and predicates the edges. RDF data do not necessarily come with a schema or semantics (expressed by constraints).

The RDF data model is *simple*, with a formal semantics and provable inference and with an extensible URI-based vocabulary which allows anyone to make statements about any resource. The RDF Schema (RDFS) language [12] provides a built-in vocabulary for asserting user-defined schemas in the RDF data model and is designed to introduce useful semantics to RDF triples. RDFS names such as `rdfs:Resource`, `rdfs:Class` and `rdf:Property` could be used as objects of triples describing *class* and *property* types. It also provides some useful relationships (properties) between resources, like *subsumption* or *instantiation*.

The OWL Web Ontology Language [23] is designed for use by applications that need to process the content of information instead of just presenting information to humans, and is used to (a) *create an ontology*, (b) *state facts* about a domain and (c) *reason about ontologies* to determine consequences of what was named and stated.

It provides a much richer set of constructs and semantics than RDFS that allows more complicated reasoning. It has three increasingly-expressive sublanguages, namely OWL Lite, OWL DL, and OWL Full. OWL, in addition to RDFS distinguishes between *object* and *datatype* properties, *complex class descriptions* through the *intersection*, *union*, *enumeration* and *property restrictions* of class descriptions. OWL contains constructs that allow one to combine class descriptions into *class axioms*. Namely *class subsumption* (defined in RDF Schema language), *equivalence* and *disjointness*. Property axioms define characteristics of properties. More specifically, these are property *subsumption*, *domain* and *range* (defined by RDFS), relationships to other properties such as *equivalence* and *inverse*, global cardinality constraints such as *functional* and *inverse functional* properties and finally *logical property characteristics* such as *symmetry* and *transitivity*. Last, OWL allows the specification of axioms for *individuals* or *instances*, such as *class membership*, *property values* as well as facts about the instance identity. More specifically, OWL allows one to specify that two instances refer to the *same* or to a *different* real world individual. Most of the OWL constructs were initially defined in 2004, in what is now known as OWL1 [43]. The semantics of some of the constructs were slightly refined in a subsequent version, OWL2 [77], introduced in 2012. Both OWL1 and OWL2 define several sublanguages that allow a different set of constructs, and thus adopt a different stance in the tradeoff between expressive power and reasoning complexity. Our presentation and analysis below focus on the constructs used in a specific sublanguage of OWL 2, namely OWL 2 RL [45], which is aimed at applications that require scalable reasoning without sacrificing too much expressive power.

The interested reader can find a detailed description of the OWL constructs in [23, 42] and a detailed description of its semantics in [52]. The OWL constructs along with a partial axiomatization in the form of first order implications that we use in this work are discussed below. Each construct is associated with specific semantics, which are formally encoded in the form of *if-then* rules. A rule means that if a dataset contains triples that match the triple pattern in the *"if"* part, then it should imply either (a) triples that match the triple pattern in the *"then"* part or (b) when the *"then"* part contains the keyword "FALSE", it means that an ontology containing the triples in the *"if"* part is inconsistent, i.e., it implies everything. The informal description of the involved OWL constructs, as well as the intuition behind their semantics is given in the subsections below.

3.1 Class and Property Subsumption

Class and *property subsumption* is the most basic, useful and frequent reasoning-intensive relationship that appears in semantic modeling. Subsumption is denoted using the RDFS constructs `rdfs:subClassOf` and `rdfs:subPropertyOf` for classes and properties respectively.

According to [12, 34] if a class c_1 is a subclass of c_2 (triple $(c_1, \texttt{rdfs:subClassOf}, c_2)$), then the instances of the former $(x, \texttt{rdf:type}, c_1)$ are also instances of the latter $(x, \texttt{rdf:type}, c_2)$. The same holds for subsumption between properties. Rules CAX-SCO and PRP-SPO1 in Table 3.1 describe these semantics.

According to the *Semantics of Schema Vocabulary* [45], class and property subsumption are *transitive* (see Rules SCM-SCO, SCM-SPO respectively in Table 3.1). More specifically, the existence of $(c_1, \texttt{rdfs:subClassOf}, c_2)$ and $(c_2, \texttt{rdfs:subClassOf}, c_3)$ in a dataset should cause the inference of $(c_1, \texttt{rdfs:subClassOf}, c_3)$.

	If	**Then**
CAX-SCO	$(?c_1, \texttt{rdfs:subClassOf}, ?c_2)$ $(?x, \texttt{rdf:type}, ?c1)$	$(?x, \texttt{rdf:type}, ?c2)$
PRP-SPO1	$(?p_1, \texttt{rdfs:subPropertyOf}, ?p_2)$ $(?x, ?p_1, ?y)$	$(?x, ?p_2, y)$
SCM-SCO	$(?c_1, \texttt{rdfs:subClassOf}, ?c_2)$ $(?c_2, \texttt{rdfs:subClassOf}, ?c_3)$	$(?c_1, \texttt{rdfs:subClassOf}, ?c_3)$
SCM-SPO	$(?p_1, \texttt{rdfs:subPropertyOf}, ?p_2)$ $(?p_2, \texttt{rdfs:subPropertyOf}, ?p_3)$	$(?p_1, \texttt{rdfs:subPropertyOf}, ?p_3)$

Table 3.1: Class and Property Subsumption

3.2 Property Domain and Range

The constructs `rdfs:domain` and `rdfs:range` are used to denote the *domain* and *range* of properties respectively. For example, $(p, \texttt{rdfs:domain}, c_1)$/$(p, \texttt{rdfs:range}, c_1)$ indicate that c_1 is the *domain*/*range* of property p. Rules SCM-RNG1/SCM-DOM1 shown in Table 3.2 state that if a property p has as *range/domain* a class c_1, then it has as *range/domain all superclasses* c_2 of c_1. Range and domain of properties is also inherited along the property subsumption hierarchy: rules SCM-RNG2/SCM-DOM2 (Table 3.2) state that if a property p_2 has as range/domain a class c, then its subproperty p_1 have also as range/domain class c. In addition whenever a subject s is connected via property p to some object o, s should be an instance of the domain of p, and o should be an instance of the range of p (rules PRP-DOM and PRP-RNG resp.).

3.3 Union and Intersection of Classes

The `owl:unionOf` construct is used to construct a new class, that is the union of two (or more) other classes. Dually, the `owl:intersectionOf` construct is used to construct a new class that is the intersection of two (or more) other classes. As with all OWL constructs, the semantics of `owl:unionOf` are intentional, i.e., all instances that are *known* to be instances of either of c_1, c_2 will be also instances of their union, and vice-versa, i.e., known instances of the union will be instances of either c_1 or c_2 (or both). According to rules SCM-UNI and SCM-INT shown in Table 3.3, a class c defined as *union* (respectively *intersection*) of a set of existing classes $c_1, c_2 \ldots c_n$, then c is inferred as their *superclass* (respectively *subclass*).

	If	Then
SCM-RNG1	$(?p, \texttt{rdfs:range}, ?c_1)$ $(?c_1, \texttt{rdfs:subClassOf}, ?c_2)$	$(?p, \texttt{rdfs:range}, ?c_2)$
SCM-RNG2	$(?p_2, \texttt{rdfs:range}, ?c)$ $(?p_1, \texttt{rdfs:subPropertyOf}, ?p_2)$	$(?p_1, \texttt{rdfs:range}, ?c)$
SCM-DOM1	$(?p, \texttt{rdfs:domain}, ?c_1)$ $(?c_1, \texttt{rdfs:subClassOf}, ?c_2)$	$(?p, \texttt{rdfs:domain}, ?c_2)$
SCM-DOM2	$(?p_2, \texttt{rdfs:domain}, ?c)$ $(?p_1, \texttt{rdfs:subPropertyOf}, ?p_2)$	$(?p_1, \texttt{rdfs:domain}, ?c)$
PRP-DOM	$(?p, \texttt{rdfs:domain}, ?c)$ $(?x, ?p, ?y)$	$(?x, \texttt{rdf:type}, ?c)$
PRP-RNG	$(?p, \texttt{rdfs:range}, ?c)$ $(?x, ?p, ?y)$	$(?y, \texttt{rdf:type}, c)$

Table 3.2: Property Domain and Range

SCM-INT	$(?c, \texttt{owl:intersectionOf}, ?x)$ LIST[$?x$, $?c_1$, ..., c_n]	$(?c, \texttt{rdfs:subClassOf}, ?c_1)$ $(?c, \texttt{rdfs:subClassOf}, ?c_2)$ ... $(?c, \texttt{rdfs:subClassOf}, ?c_n)$
SCM-UNI	$(?c, \texttt{owl:unionOf}, ?x)$ LIST[$?x$, $?c_1$, ..., c_n]	$(?c_1, \texttt{rdfs:subClassOf}, ?c)$ $(?c_1, \texttt{rdfs:subClassOf}, ?c)$ ... $(?c_n, \texttt{rdfs:subClassOf}, ?c)$

Table 3.3: Union and Intersection of Classes

3.4 Enumeration

The `owl:oneOf` construct is used to define a class via enumeration, i.e., by explicitly stating its instances. This implies that all such individuals are instances of the defined class, as shown in CLS-OO in Table 3.4.

	If	Then
CLS-OO	$(?c, \texttt{owl:oneOf}, ?x)$ LIST[$?x$, $?y_1$, ..., $?y_n$]	$(?y_1, \texttt{rdf:type}, ?c)$ ... $(?y_n, \texttt{rdf:type}, ?c)$

Table 3.4: Semantics of Enumerated Classes

3.5 Equality of Individuals

The uncontrolled nature of the Web of Data implies that there will be several cases where the *same resource* in the real world (e.g., a human being, an object or an idea) may be described using different URIs in different or even the same dataset. To address this problem, OWL2 proposes the use of the OWL construct `owl:sameAs`

to connect *instances* that represent the same real-world entity[1]. Hence OWL construct `owl:sameAs` denotes *equality*. The opposite of `owl:sameAs` is `owl:differentFrom`, which explicitly states that two individuals are different, i.e., they correspond to a different real-world entity. Table 3.5 presents all rules that hold for `owl:sameAs` and `owl:differentFrom` constructs.

	If	**Then**
EQ-REF	$(?s, ?p, ?o)$	$(?s, \texttt{owl:sameAs}, ?s)$ $(?p, \texttt{owl:sameAs}, ?p)$ $(?o, \texttt{owl:sameAs}, ?o)$
EQ-SYM	$(?x, \texttt{owl:sameAs}, ?y)$	$(?y, \texttt{owl:sameAs}, ?x)$
EQ-TRANS	$(?x, \texttt{owl:sameAs}, ?y)$ $(?y, \texttt{owl:sameAs}, ?z)$	$(?x, \texttt{owl:sameAs}, ?z)$
EQ-REP-S	$(?s, \texttt{owl:sameAs}, ?s')$ $(?s, ?p, ?o)$	$(?s', ?p, ?o)$
EQ-REP-P	$(?p, \texttt{owl:sameAs}, ?p')$ $(?s, ?p, ?o)$	$(?s, ?p', ?o)$
EQ-REP-O	$(?o, \texttt{owl:sameAs}, ?o')$ $(?s, ?p, ?o)$	$(?s, ?p, ?o')$
EQ-DIFF1	$(?x, \texttt{owl:sameAs}, ?y)$ $(?x, \texttt{owl:differentFrom}, ?y)$	FALSE

Table 3.5: Semantics of Equality

One observation is that whatever holds for one resource, holds for the other as well (rules EQ-REP-S, EQ-REP-P, EQ-REP-O). Obviously, a pair of individuals cannot be the same and different at the same time, thus the rule EQ-DIFF1. By its definition, `owl:sameAs` has the properties of equivalence relations, i.e., it is reflexive, symmetric and transitive. *Reflexivity* implies that $(x, \texttt{owl:sameAs}, x)$ for all resources x (rule EQ-REF). The relation being *symmetric* means that $(x, \texttt{owl:sameAs}, y)$ implies $(y, \texttt{owl:sameAs}, x)$ (rule EQ-SYM). Finally, *transitivity* implies that from $(x, \texttt{owl:sameAs}, y)$ and $(y, \texttt{owl:sameAs}, z)$ we should infer $(x, \texttt{owl:sameAs}, z)$ (rule EQ-TRANS).

3.6 Inverse of Properties

The inverse property construct (`owl:inverseOf`) allows one to define a property as the inverse of another. For example, the property has_parent is the inverse of has_child. More formally, if p_1 is the inverse of p_2 then a triple of the form (x, p_1, y) implies (y, p_2, x). Note that when p_1 is the inverse of p_2, then p_2 is the inverse of p_1, so the above implication holds both ways. Rows PRP-INV1, PRP-INV2 of Table 3.6 expresses these implications.

3.7 Constraints on Properties

Several OWL constructs are introduced to allow restricting the values that a property can have. In particular, there are constructs that restrict a property to be:

- *functional* (`owl:FunctionalProperty`)
- *inverse functional* (`owl:InverseFunctionalProperty`)
- *transitive* (`owl:TransitiveProperty`)

[1] Although `owl:sameAs` construct must be used only to denote that two URIs denote the same real world entity, it is sometimes used to express equality at the schema level.

	If	then
PRP-INV1	$(?p_1, \texttt{owl:inverseOf}, ?p_2)$ $(?x, ?p_1, ?y)$	$(?y, ?p_2, ?x)$
PRP-INV2	$(?p_1, \texttt{owl:inverseOf}, ?p_2)$ $(?x, ?p_2, ?y)$	$(?y, ?p_1, ?x)$

Table 3.6: Inverse Constraints

- *symmetric* (`owl:SymmetricProperty`)
- *asymmetric* (`owl:AsymmetricProperty`) and finally
- *irreflexive* (`owl:IrreflexiveProperty`)

The intuitive semantics of such constraints are given below. The formal semantics can be found at Table 3.7.

	If	then
PRP-FP	$(?p, \texttt{rdf:type}, \texttt{owl:FunctionalProperty})$ $(?x, ?p, ?y_1)$ $(?x, ?p, ?y_2)$	$(?y_1, \texttt{owl:sameAs}, ?y_2)$
PRP-IFP	$(?p, \texttt{rdf:type}, \texttt{owl:InverseFunctionalProperty})$ $(?x_1, ?p, ?y)$ $(?x_2, ?p, ?y)$	$(?x_1, \texttt{owl:sameAs}, ?x_2)$
PRP-TRP	$(?p, \texttt{rdf:type}, \texttt{owl:TransitiveProperty})$ $(?x, ?p, ?y)$ $(?y, ?p, ?z)$	$(?x, ?p, ?z)$
PRP-SYM	$(?p, \texttt{rdf:type}, \texttt{owl:SymmetricProperty})$ $(?x, ?p, ?y)$	$(?y, ?p, ?x)$
PRP-ASYP	$(?P, \texttt{rdf:type}, \texttt{owl:AsymmetricProperty})$ $(?x, ?p, ?y)$ $(?y, ?p, ?x)$	FALSE
PRP-IRP	$(?P, \texttt{rdf:type}, \texttt{owl:IrreflexiveProperty})$ $(?x, ?P, ?x)$	FALSE

Table 3.7: Constraints of Properties

Inverse functional and functional properties are useful to denote values that uniquely identify an entity. Note that, due to the fact that the semantics of OWL2 do not include the Unique Name Assumption (UNA), functional and inverse functional properties should not be viewed as integrity constraints, because they cannot directly (by themselves) lead to contradictions. Instead, they force us to assume (infer) that certain individuals are the same, as indicated by rules PRP-FP and PRP-IFP. If a property p is defined as transitive, then the existence of triples (x, p, y) and (y, p, z) should imply (x, p, z) (see also rule PRP-TRP). Transitive properties appear quite often in user-defined properties (e.g., *partOf*), but also in built-in properties (e.g., *subsumption*).

If a property p is defined as *asymmetric*, then whenever x is connected to y via p, then y cannot be connected to x via p. More formally, if p is asymmetric, then the existence of (x, p, y) and (y, p, x) violates the correctness of the database (rule PRP-ASYP). If a property p is *symmetric* then, an instance x is connected to itself through p (rule PRP-SYM). Finally, if a property p is defined as *irreflexive* then, no individual can be connected to itself via p, i.e., a triple (x, p, x) cannot exist in the dataset (cf. rule PRP-IRP).

3.8 Keys of Classes

The `owl:hasKey` construct is used to specify a property (or a set of properties) as being the key for a given class (in the sense of primary keys, as defined in relational tables). Thus, the values of said properties uniquely identify a resource that is an instance of the class. For example, if property p is the key for class c, then the triples $(x, \texttt{rdf:type}, c)$, $(y, \texttt{rdf:type}, c)$, (x, p, z) and (y, p, z) imply $(x, \texttt{owl:sameAs}, y)$. A more general form of this statement is shown by rule PRP-KEY of Table 3.8.

	If	then
PRP-KEY	$(?c, \texttt{owl:hasKey}, ?u)$ LIST[$?u$, $?P_1$, ..., $?P_2$] $(?x, \texttt{rdf:type}, ?c)$ $(?x, ?p_1, ?z_1)$ ... $(?x, ?p_n, ?z_n)$ $(?y, \texttt{rdf:type}, ?c)$ $(?y, ?p_1, ?z_1)$ ... $(?y, ?p_n, ?z_n)$	$(?x, \texttt{owl:sameAs}, ?y)$

Table 3.8: Keys

3.9 Property Chains

The construct `owl:propertyChainAxiom` allows one to define properties as a composition of others. As an example, the property $grandparent$ can be defined as the composition of $parent$ with itself. More formally, when a property p is defined as the composition of properties p_1 and p_2, then the triples (x, p_1, y), (y, p_2, z) imply (x, p, z) (rule PRP-SPO2, Table 3.9).

	If	then
PRP-SPO2	$(?p, \texttt{owl:propertyChainAxiom}, ?x)$ LIST[$?x$, $?p_1$, ..., p_n] $(?u_1, ?p_1, ?u_2)$ $(?u_2, ?p_2, ?u_3)$ ... $(?u_n, ?p_n, ?u_{n+1})$	$(?u_1, ?p, ?u_{n+1})$

Table 3.9: Property Chains

3.10 Disjoint Classes and Properties

Defining two classes c_1, c_2 as *disjoint* implies that they cannot share common instances. Disjointness is denoted using the `owl:disjointWith` construct. Disjointness between classes is generalized to multiple ones using the `owl:AllDisjointClasses` construct. The semantics of said constructs implemented by rules CAX-DW, CAX-ADC are shown in Table 3.10. Similar constructs `owl:propertyDisjointWith`, `owl:AllDisjointProperties` exist for specifying *disjoint properties*, i.e., properties that cannot share common instances. Rules PRP-ADP, PRP-PDW of Table 3.10 show some consequences of the semantics of the above constructs.

	If	Then
CAX-DW	$(?c_1, \texttt{owl:disjointWith}, ?c_2)$ $(?x, \texttt{rdf:type}, ?c_1)$ $(?x, \texttt{rdf:type}, ?c_2)$	FALSE
CAX-ADC	$(?x, \texttt{rdf:type}, \texttt{owl:AllDisjointClasses})$ $(?x, owl{:}members, ?y)$ LIST[$?y$, $?c_1$, ..., $?c_n$] $(?z, \texttt{rdf:type}, ?c_i)$ $(?z, \texttt{rdf:type}, ?c_j)$	FALSE
PRP-ADP	$(?x, \texttt{rdf:type}, \texttt{owl:AllDisjointProperties})$ $(?x, owl{:}members, ?y)$ LIST[$?y$, $?P_1$, $?P_2$, ... $?P_n$] $(?u, ?P_1, ?z)$ $(?u, ?P_2, ?z)$	FALSE
PRP-PDW	$(?P_1, \texttt{owl:propertyDisjointWith}, ?P_2)$ $(?x, ?P_1, ?y)$ $(?x, ?P_2, ?y)$	FALSE

Table 3.10: Disjoint Classes and Properties

3.11 Cardinalities

Cardinality constraints appear quite often in practice, and are used to allow a specific maximum or minimum number of values for any given property. Constraints `owl:maxCardinality` and `owl:minCardinality` link a restriction class to a data value belonging to the value space of the XML Schema datatype `xsd:nonNegativeInteger`.

The most common type of cardinality constraints are for values 0 and 1, which are simpler to handle; such cardinality constraints correspond to functional or required properties. Note that OWL 2 RL only supports cardinality constraints of this type (i.e., with values 0 or 1), so our analysis in this deliverable will focus on these types of cardinality constraints as well. Table 3.11 shows restrictions implied by `owl:maxCardinality`.

	If	Then
CLS-MAXC1	$(?x$,`owl:maxCardinality` "0"`xsd:nonNegativeInteger`$)$ $(?x, owl{:}onProperty, ?p)$ $(?u, \texttt{rdf:type}, ?x)$ $(?u, ?p, ?y)$	FALSE
CLS-MAXC2	$(?x$,`owl:maxCardinality` "1"`xsd:nonNegativeInteger`$)$ $(?x, owl{:}onProperty, ?p)$ $(?u, \texttt{rdf:type}, ?x)$ $(?u, ?p, ?y_1)$ $(?u, ?p, ?y_2)$	FALSE

Table 3.11: Cardinalities

4 REASONING BENCHMARK: CONFORMANCE TESTS

In this Chapter we discuss the *conformance tests* that determine whether a certain RDFS or OWL construct is supported by an RDF engine as specified by the semantics associated with the said construct [52, 45]. This is of crucial importance, because a limited number of reasoning constructs is supported by RDF query engines, and are not yet supported "by default" in all existing systems. A test is proposed per rule following the semantics specified in [45]. For each test we provide (a) the set of triples that should at least exist in the dataset (*preconditions*)) (b) the ASK SPARQL query that implements the construct's semantics and (c) the answer expected from the RDF engine. Recall, that a SPARQL ASK query checks whether a specific pattern is implied by an RDF graph.

4.1 Class and Property Subsumption

In this Section we propose the tests that can be used to check whether the semantics of *subclass* (`rdfs:subClassOf`) and *subproperty* (`rdfs:subPropertyOf`) constructs are supported by an RDF engine. Sections 4.1.1 and 4.1.2 are aimed at testing the ability of the RDF engine to compute correctly the *instances* of *classes* (`rdf:type`) and *properties* along the *subclass* (`rdfs:subClassOf`) and *subproperty* hierarchies (`rdfs:subPropertyOf`) respectively. Sections 4.1.3 and 4.1.4 describe tests used to check the ability of the systems to compute correctly the *transitivity* of *subclass* and *subproperty* relations.

4.1.1 Class Subsumption (CAX-SCO)

Preconditions

```
<C1> rdfs:subClassOf <C2> .
<x> rdf:type <C1> .
```

SPARQL Query

```
ASK { <x> rdf:type <C2> }
```

Expected behavior: The expected result is *true*, since according to the semantics of the CAX-SCO rule, all instances of class C are also instances of its super-classes (classes `C1` and `C2` in our example).

4.1.2 Property Subsumption (PRP-SPO1)

Preconditions

```
<P1> rdfs:subPropertyOf <P2> .
<P2> rdfs:subPropertyOf <P3> .
<x> <P1> <y> .
```

SPARQL Query

```
ASK { <x> <P1> <y> .
      FILTER NOT EXISTS { <x> <P3> <y> } }
```

Expected behavior: The expected result is *false*, since according to the semantics of the PRP-SPO1 rule, all instances of property P are also instances of its super-properties (properties `P1` and `P3` in our test case).

4.1.3 Class Subsumption (SCM-SCO)

Preconditions

```
<C1> rdfs:subClassOf <C2> .
<C2> rdfs:subClassOf <C3> .
```

SPARQL Query

```
ASK { <C1> rdfs:subClassOf <C3> }
```

Expected behavior: The expected result is *true*, as classes `C1` and `C3` are related with `C2` through the transitive relation `rdfs:subClassOf`.

4.1.4 Property Subsumption (SCM-SPO)

Preconditions

```
<P1> rdfs:subPropertyOf <P2> .
<P2> rdfs:subPropertyOf <P3> .
```

SPARQL Query

```
ASK { <P1> rdfs:subPropertyOf <P3> }
```

Expected behavior: The expected result is *true*, given that properties `P1` and `P3` are related with `P2` through the transitive relation `rdfs:subPropertyOf`.

4.2 Property Domain and Range

In this Section we discuss the tests that can be used to check whether the semantics of *domain* (`rdfs:domain`) and *range* (`rdfs:range`) constructs are correctly implemented. Similar to the class and property *subsumption* tests discussed in Section 4.1, the tests are ASK queries that return *true* if the semantics are correctly implemented, *false* otherwise.

4.2.1 Property Range (SCM-RNG1)

Preconditions

```
<P> rdfs:range <C1> .
<C1> rdfs:subClassOf <C2> .
<C2> rdfs:subClassOf <C3> .
```

SPARQL Query

```
ASK { <P> rdfs:range <C2> .
      FILTER NOT EXISTS { <P> rdfs:range <C3> } }
```

Expected behavior: According to the semantics of `rdfs:range` as specified in rule SCM-RNG1, if a property has *range* a class C then the property has a range C's superclasses. Hence, the expected result is *false*, since class `C3` is a superclass of `C1` (through transitivity), the latter being the range of property `P`.

4.2.2 Property Range (SCM-RNG2)

Preconditions

```
<P2> rdfs:range <C> .
<P1> rdfs:subPropertyOf <P2> .
```

SPARQL Query

```
ASK { <P1> rdfs:range <C> }
```

Expected behavior: According to the semantics of `rdfs:range` as specified in rule SCM-RNG2, if a property P has *range* a class C then its *subproperties* have the same range. Consequently, the expected result in this test is *true*, as the property `P1` is a subproperty of `P2`, and `P2`'s range is class `C`.

4.2.3 Property Domain (SCM-DOM1)

Preconditions

```
<P> rdfs:domain <C1> .
<C1> rdfs:subClassOf <C2> .
```

SPARQL Query

```
ASK { <P> rdfs:domain <C2> }
```

Expected behavior: According to the semantics of `rdfs:domain` as specified in rule SCM-DOM1, if a property has *domain* a class C then the property has domain C's superclasses. Hence, in this test the expected result is *true*, since class `C2` is a superclass of `C1`, the latter being the domain of property `P`.

4.2.4 Property Domain (SCM-DOM2)

Preconditions

```
<P2> rdfs:domain <C> .
<P1> rdfs:subPropertyOf <P2> .
```

SPARQL Query

```
ASK { <P1> rdfs:domain <C> }
```

Expected behavior: According to the semantics of `rdfs:domain` as given in rule SCM-DOM2, if a property P has *domain* a class C then its *subproperties* have the same domain. Consequently, the expected result in this test is *true*, as the property `P1` is a subproperty of `P2`, and `P2`'s range is class `C`.

4.2.5 Property Domain (PRP-DOM)

Preconditions

```
<x> <P> <y> .
<P> rdfs:domain <C> .
```

SPARQL Query

```
ASK { <x> rdf:type <C> }
```

Expected behavior: Rule PRP-DOM states that if a triple (x, P, y) exists in the dataset, and property P has as domain class C, then x must be an instance of C. The test proposed here must return $true$ since property `P` has domain class `C`.

4.2.6 Property Range (PRP-RNG)

Preconditions

```
<x> <P> <y> .
<P> rdfs:range <C> .
```

SPARQL Query

```
ASK { <y> rdf:type <C> }
```

Expected behavior: Rule PRP-RNG states that if a triple (x, P, y) exists in the dataset, and property P has as range class C, then y must be an instance of C. The test proposed here must return $true$ since property `P` has *range* class `C` according to the stated preconditions.

4.3 Union and Intersection of Classes

In this Section we discuss tests that can be used to check whether *complex class definitions* through *union* and *intersection* are correctly implemented by RDF engines. The tests discussed here are also expressed in the form of ASK queries.

4.3.1 Union of Classes (SCM-UNI)

Preconditions

```
<C> owl:unionOf ( <C1> <C2> <C3> ) .
```

SPARQL Query

```
ASK { <C1> rdfs:subClassOf <C> .
      <C3> rdfs:subclassOf <C> .
      <C2> rdfs:subclassOf <C> }
```

Expected behavior: Rule SCM-UNI states that if a class C is defined as a *union* of a classes $C1$, $C2$, ... Ck, then the RDF engine should imply that each such class is a *subclass* of C. The expected result in our test case is *true* since class `C` is defined as a *union* of all classes `C1`, `C2` and `C3`.

4.3.2 Union of Classes (SCM-UNI, CAX-SCO)

Preconditions

```
<x> rdf:type <C2> .
<C> owl:unionOf ( <C1> <C2> <C3> ) .
```

SPARQL Query

```
ASK { <x> rdf:type <C> }
```

Expected behavior: The combination of rules SCM-UNI and CAX-SCO states that if a class C is defined as a *union* of a set of classes $C1$, $C2$, ..., Ck, then all instances of classes $C1$, $C2$, ..., Ck are also instances of class C. The result of this test is *true* given that class `C` is the union of classes `C1`, `C2` and `C3` implying that `C1`, `C2` and `C3` are all sub-classes of `C`. So, instances of class `C2` (resource `x`) are also instances of `C`.

4.3.3 Intersection of Classes (SCM-INT)

Preconditions

```
<C> owl:intersectionOf ( <C1> <C2> <C3> ) .
```

SPARQL Query

```
SELECT ?x
WHERE { <C> rdfs:subClassOf ?x }
```

Expected behavior Rule SCM-INT states that if a class C is defined as an *intersection* of classes $C1$, $C2$, ..., Ck, then the RDF engine should imply that class C is a *subclass* of each class. The expected result in our test case is the following set of bindings for variable `?x`:

$$\mu_1(?\texttt{x}) = \texttt{C1}, \mu_2(?\texttt{x}) = \texttt{C2}, \mu_3(?\texttt{x}) = \texttt{C3}$$

since class `C` is defined as an *intersection* of all classes `C1`, `C2` and `C3`.

4.3.4 Intersection of Classes (SCM-INT, CAX-SCO)

Preconditions

```
<x> rdf:type <C> .
<C> owl:intersectionOf ( <C1> <C2> <C3> ) .
```

SPARQL Query

```
ASK { ?x rdf:type <C1> .
      ?x rdf:type <C2>
      ?x rdf:type <C3> }
```

Expected behavior The combination of rules SCM-INT and CAX-SCO states that if a class C is defined as an *intersection* of a set of classes $C1$, $C2$, ..., Ck, then any instance of class C is also an instance of $C1$, $C2$, ..., Ck. The result of this test is *true*: given that class `C` is defined as the intersection of classes `C1`, `C2` and `C3` then the former is a subclass of `C1`, `C2` and `C3`. So, instances of class `C` (resource `x`) are also instances of `C1`, `C2` and `C3`.

4.4 Enumeration of Individuals

In this section we discuss tests regarding the definition of *complex classes* using *enumeration* through construct `owl:oneOf`. The test has the form of an ASK query that returns *true* if the requested pattern is implied by the RDF engine.

4.4.1 Enumeration of Individuals (CLS-OO)

Preconditions

```
<C> owl:oneOf ( <y1> <y2> ) .
```

SPARQL Query

```
ASK { <y1> rdf:type <C> .
      <y2> rdf:type <C> }
```

Expected behavior: According to the semantics of CLS-OO rule, if a class C is defined as an enumeration of a set of resources $x1, x2, \ldots, xn$, then the RDF engine should infer that each x_i is an instance of class C, i.e., infer triple $(x_i, \texttt{rdf:type}, C)$. The expected result is *true* if the RDF engine correctly implements the OWL semantics.

4.5 Equality

4.5.1 Equality (EQ-REF)

Preconditions

```
<s> <p> <o> .
```

SPARQL Query

```
ASK { <s> owl:sameAs <s> .
      <p> owl:sameAs <p> .
      ?o owl:sameAs ?o .
      <s> <p> ?o }
```

Expected behavior: According to the semantics of EQ-REF rule, every known term is inferred to have the `owl:sameAs` relation to itself. So, the expected result is *true* if the RDF engine correctly implements the OWL semantics.

4.5.2 Equality (EQ-SYM)

Preconditions

```
<x> owl:sameAs <y> .
```

SPARQL Query

```
ASK { <y> owl:sameAs <x> }
```

Expected behavior: The expected result is *true*, since according to the semantics of the EQ-SYM rule, the `owl:sameAs` relation is a symmetric one.

4.5.3 Equality (EQ-TRANS)

Preconditions

```
<x> owl:sameAs <y> .
<y> owl:sameAs <z>
```

SPARQL Query

```
SELECT ?z
WHERE { <x> owl:sameAs ?z }
```

Expected behavior: According to the semantics of EQ-TRANS rule, the `owl:sameAs` relation is a transitive relation. The test proposed here must return three bindings $\mu_1(?z) = y$, $\mu_2(?z) = z$, and $\mu_3(?z) = x$ since `x` is related to `z` and `x` through the transitive and symmetric relation `owl:sameAs`.

4.5.4 Equality (EQ-REP-S)

Preconditions

```
<s> owl:sameAs <s1> .
<s> <p> <o> .
```

SPARQL Query

```
ASK { <s1> <p> <o> }
```

Expected behavior: The expected result is *true*, since according to the semantics of the EQ-REP-S rule, every term can be replaced by its equivalent one through property `owl:sameAs`.

4.5.5 Equality (EQ-REP-P)

Preconditions

```
<p> owl:sameAs <p1> .
<s> <p> <o> .
```

SPARQL Query

```
ASK { <s> <p1> <o> }
```

Expected behavior: Similarly to Section 4.5.4, the expected result of above query is *true*.

4.5.6 Equality (EQ-REP-O)

Preconditions

```
<o> owl:sameAs <o1> .
<s> <p> <o> .
```

SPARQL Query

```
ASK { <s> <p> <o1> }
```

Expected behavior: Similarly to Section 4.5.4, the expected result is *true*.

4.6 Inverse of Properties

4.6.1 Inverse of Properties (PRP-INV1)

Preconditions

```
<P1> owl:inverseOf <P2> .
<x> <P1> <y> .
```

SPARQL Query

```
ASK { <y> <P2> <x> }
```

Expected behavior: The expected result should be *true*, since according to the semantics of the PRP-INV1 rule, given that property P1 is stated to be the inverse of a property P2 and x is related to y through P1, then y is related to x through P2.

4.6.2 Inverse of Properties (PRP-INV2)

Preconditions

```
<P1> owl:inverseOf <P2> .
<x> <P2> <y> .
```

SPARQL Query

```
ASK { <y> <P1> <x> }
```

Expected behavior Similarly to Section 4.6.1, the expected result should be *true*.

4.7 Constraints on Properties

4.7.1 Constraints on Properties (PRP-FP)

Preconditions

```
<P> rdf:type owl:FunctionalProperty .
<x> <P> <y1> .
<x> <P> <y2>  .
```

SPARQL Query

```
ASK { <y1> owl:sameAs <y2> . }
```

Expected behavior: The above query should return true, since P is a functional property, so there cannot exist two triples $(\texttt{x}, \texttt{P}, \texttt{y1})$ and $(\texttt{x}, \texttt{P}, \texttt{y2})$ without y1 and y2 being related with the owl:sameAs property.

4.7.2 Constraints on Properties (PRP-IFP)

Preconditions

```
<P> rdf:type owl:InverseFunctionalProperty .
<x1> <P> <y> .
<x2> <P> <y> .
```

SPARQL Query

```
ASK { <x1> owl:sameAs <x2> }
```

Expected behavior: The expected result is *true*, as according to the semantics of the PRP-IFP rule, given that there are two triples in the ontology of the form $(\texttt{x1}, \texttt{P}, \texttt{y})$ and $(\texttt{x2}, \texttt{P}, \texttt{y})$ for an inverse functional property P, then we can infer that x1 and x2 are the same individual.

4.7.3 Constraints on Properties (PRP-ASYP)

Preconditions

```
<P> rdf:type owl:AsymmetricProperty .
```

SPARQL Query

```
INSERT DATA { <x> <P> <y> .
              <y> <P> <x>  }
```

Expected behavior: If the engine supports checking the constraints on properties the above query should fail, since P is an asymmetric property, so for every pair of instances (x,y) which is an instance of property P, the pair (y,x) cannot be an instance of P.

4.7.4 Constraints on Properties (PRP-IRP)

Preconditions

```
<P> rdf:type owl:IrreflexiveProperty .
```

SPARQL Query

```
INSERT DATA { <x> <P> <x> }
```

Expected behavior: The above query should fail, as the property P is defined as an irreflexive property, so no pair (x,x) can be defined as an instance of this property.

4.7.5 Constraints on Properties (PRP-TRP)

Preconditions

```
<P> rdf:type owl:TransitiveProperty .
<x> <P> <y> .
<y> <P> <z> .
```

SPARQL Query

```
ASK { <x> <P> <z> }
```

Expected behavior: The expected result is *true*, as the property P is a transitive property and there is a path from x to z through P.

4.8 Class Keys

4.8.1 Class Keys (PRP-KEY)

Preconditions

```
<C> owl:hasKey ( <P1> <P2> ) .
<x> rdf:type <C> .
<x> <P1> <z1> .
<x> <P2> <z2> .
<y> rdf:type <C> .
<y> <P1> <z1> .
<y> <P2> <z2> .
```

SPARQL Query

```
ASK { <x> owl:sameAs <y> }
```

Expected behavior: The expected result is *true*, since according to the key constraints of PRP-KEY rule the two instances x and y are connected using an owl:sameAs link.

4.9 Property Chains

4.9.1 Property Chains (PRP-SPO2)

Preconditions

```
<P> owl:propertyChainAxiom ( <P1> <P2> ) .
<x> <P1> <y> .
<y> <P2> <z> .
```

SPARQL Query

```
ASK { <x> <P> <z> }
```

Expected behavior: The expected result is *true*, as according to the semantics of PRP-SPO2 rule, since property P is defined as the composition of properties P1 and P2, then the triples $(x, P1, y)$, $(y, P2, z)$ imply (x, P, z).

4.10 Disjoint Classes and Properties

4.10.1 Disjoint Classes and Properties (PRP-PDW)

Preconditions

```
<P1> owl:propertyDisjointWith <P2> .
```

SPARQL Query

```
INSERT DATA { <x> <P1> <y> .
              <x> <P2> <y> . }
```

Expected behavior: According to the semantics of PRP-PDW rule two disjoint properties cannot share common instances. So, if the engine supports checking the disjointness on properties the above query should fail.

4.10.2 Disjoint Classes and Properties (PRP-ADP)

Preconditions

```
_:bn1 rdf:type owl:AllDisjointProperties .
_:bn1 owl:members ( <P1> <P2> <P3> ) .
```

SPARQL Query

```
INSERT DATA { <x> <P1> <y> .
              <x> <P2> <y> .
              <x> <P3> <y> . }
```

Expected behavior: The above query should fail, since according to the semantics of PRP-ADP rule all disjoint properties cannot share common instances.

4.10.3 Disjoint Classes and Properties (CAX-DW)

Preconditions

```
<C1> owl:disjointWith <C2> .
```

SPARQL Query

```
INSERT DATA { <x> rdf:type <C1> .
              <x> rdf:type <C2> }
```

Expected behavior: The above query should fail, as according to the semantics of CAX-DW an individual cannot be an instance of two disjoint classes.

4.10.4 Disjoint Classes and Properties (CAX-ADC)

Preconditions

```
_:bn1  rdf:type owl:AllDisjointClasses .
_:bn1 owl:members ( <C1> <C2> <C3> ) .
```

SPARQL Query

```
INSERT DATA { <x> rdf:type <C1> .
              <x> rdf:type <C2> .
              <x> rdf:type <C3> }
```

Expected behavior: According to the semantics of CAX-ADC rule all disjoint classes cannot share common instances. So the above query should fail.

4.11 Cardinalities

4.11.1 Cardinalities (CLS-MAXC1)

Preconditions

```
<C> rdfs:subClassOf [
       rdf:type owl:Restriction ;
       owl:onProperty <P> ;
       owl:maxCardinality "0"^^xsd:NonNegativeInteger
    ] .
```

SPARQL Query

```
INSERT DATA { <x> rdf:type <C> .
              <x> <P> <y> }
```

Expected behavior:

If the engine performs consistency checking the above query should fail, as the property `P` is defined in such a way that cannot have any instances (triple $(\mathtt{x}, \mathtt{p}, \mathtt{y})$).

4.11.2 Cardinalities (CLS-MAXC2)

Preconditions

```
<C> rdfs:subClassOf [
       rdf:type owl:Restriction ;
       owl:onProperty <P> ;
       owl:maxCardinality "1"^^xsd:NonNegativeInteger
    ] .
```

SPARQL Query

```
INSERT DATA {  <x> rdf:type <C> .
               <x> <P> <y1> .
               <x> <P> <y2> }
```

Expected behavior

If the engine performs consistency checking the above query would either be rejected, or the knowledge that `y1` and `y2` are the same should be obtained (triple $(\mathtt{y1}, \mathtt{owl{:}sameAs}, \mathtt{y2})$).

5 REASONING: STATIC TESTS

In this Chapter we discuss a set of *static tests* that determine whether an underlying engine takes into account the schema information in order to answer more efficiently queries that refer to constraints defined in the schema. Such tests can also be defined in a similar manner for other constructs such as `owl:AllDisjointClasses`, `owl:AllDisjointProperties`, `owl:allValuesFrom` etc.

5.1 Equality of Classes (`owl:equivalentClass`)

Preconditions

```
<A> owl:equivalentClass <B> .
```

SPARQL Query

```
SELECT ?y
WHERE { ?y rdf:type <A> .
        ?y <P> ?y1 .
        ?y <P1> ?y2 .
        ?y <P2> ?y3 .
        ?y <P3> ?y4 .
        ?y <P4> ?y5 .
        FILTER NOT EXISTS { ?y rdf:type <B> } }
```

Expected behavior: This query requests instances of class `A` that are not instances of class `B`. Nevertheless, according to the semantics of `owl:equivalentClass` any instance of class `A` is also an instance of class `B` (and vice versa). Hence, a query engine should consider this information and return an empty answer in constant time instead of evaluating a complex query.

5.2 Disjointness of Classes (`owl:disjointWith`)

Preconditions

```
<A> owl:disjointWith <B> .
```

SPARQL Query

```
SELECT ?y1 ?y2 ?y3 ?y4 ?y5
WHERE { ?x rdf:type <A> .
        ?x rdf:type <B> .
        ?x <P> ?y1 .
        ?x <P1> ?y2 .
        ?x <P2> ?y3 .
        ?x <P3> ?y4 .
        ?x <P4> ?y5 }
```

Expected behavior: This test is similar to the one discussed in Section 5.1 for class equivalence. According to the semantics of `owl:disjointWith`, classes `A` and `B` cannot have common instances. Consequently, an optimizer that takes into account schema information, should return in constant time no answers instead of evaluating a complex query.

5.3 Equality of Properties (owl:equivalentProperty, owl:FunctionalProperty)

Preconditions

```
<P1> owl:equivalentProperty <P2> .
<P2> rdf:type owl:FunctionalProperty .
```

SPARQL Query

```
SELECT ?y
WHERE { ?y rdf:type <A> .
        ?y <P> ?y1 .
        ?y <P1> ?y2 .
        ?y <P2> ?y3 .
        ?y <P3> ?y4 .
        ?y <P4> ?y5 .
        ?y2 owl:differentFrom ?y3
}
```

Expected behavior: Given that properties P1 and P2 are *equivalent*, then for every triple $(x, P1, y)$ there should exist a triple $(x, P2, y)$. Since P2 is a *functional* property and the values bound to variables ?y2 and ?y3 are stated as *different* (i.e., triple pattern $(?y2, owl:differentFrom, ?y3)$) then the query should return no answers in constant time.

5.4 Range of Properties (rdfs:range, owl:disjointWith)

Preconditions

```
<P> rdfs:range <B> .
<A> owl:disjointWith <B> .
```

SPARQL Query

```
SELECT ?v
WHERE { ?v rdf:type <A> .
        ?u <P> ?v .
        ?u <P> ?v1 .
        ?u <P1> ?v2 .
        ?u <P2> ?v3 .
        ?u <P3> ?v4 .
        ?u <P4> ?v5 }
```

Expected behavior: The query above asks for all instances (variable ?v) of class A that are also values of property P whose range is class B. Note that we require that instances to which ?v is bound, to be also object values for triple pattern $(?u, P, ?v)$, hence instances of class B. An optimizer that takes into account schema information should return an empty result in constant time instead of devising or even evaluating the large star join.

5.5 Domain of Properties (rdfs:domain, owl:disjointWith)

```
<P> rdfs:domain <A> .
<A> owl:disjointWith <B> .
```

SPARQL Query

```
SELECT ?v
WHERE { ?u <P> ?v .
        ?u <P> ?v1 .
        ?u <P1> ?v2 .
        ?u <P2> ?v3 .
        ?u <P3> ?v4 .
        ?u <P4> ?v5 .
        ?u rdf:type <B> }
```

Expected behavior: This test is similar to the one given in Section 5.4 where we use the `rdfs:domain` instead of the `rdfs:range` property.

5.6 Uniqueness of Property Values (`owl:FunctionalProperty`)

Preconditions

```
<P> rdf:type owl:FunctionalProperty .
```

SPARQL Query

```
SELECT ?s
WHERE { ?s <P1> ?o1 .
        ?s <P2> ?o2 .
        ?s <P3> ?o3 .
        ?s <P4> ?o4 .
        ?s <P> ?o5 .
        ?s <P> ?o6.
        ?o5 owl:differentFrom ?o6 }
```

Expected behavior: The knowledge that one property is *functional* must be used from the query engine in order to answer some queries in constant time. Considering the above complex query and taking into account the schema information, (property `P` is functional) no answers should be returned at constant time, instead of evaluating the large star query.

6 REASONING: SELECTIVITY TESTS

In this Chapter we discuss the selectivity tests that determine whether an underlying engine takes advantage of OWL constructs, in its effort to find the optimal join ordering in a query plan. For each class of tests, we give (a) a set of triples that must exist in the dataset in order to be able to test if the semantics of the considered constructs are correctly implemented (b) and a set of SPARQL SELECT queries.

6.1 Cardinality

In this Section we discuss the use of *cardinality constraints* in queries that can be used in a benchmark to test if the query optimizer considers this specific *schema information* when selecting the order in which to perform the joins in a query plan. The query we propose here uses the following set of constructs:

- `owl:FunctionalProperty`
- `owl:maxCardinality`
- `owl:minCardinality`
- `owl:cardinality`

Preconditions

```
<P3> rdf:type owl:FunctionalProperty .
<A> rdfs:subClassOf [
       rdf:type owl:Restriction ;
       owl:onProperty <P4> ;
       owl:maxCardinality "2"^^xsd:NonNegativeInteger
    ] ;
    rdfs:subClassOf [
       rdf:type owl:Restriction ;
       owl:onProperty <P5> ;
       owl:minCardinality "3"^^xsd:NonNegativeInteger
       owl:maxCardinality "5"^^xsd:NonNegativeInteger
    ] ;
    rdfs:subClassOf [
       rdf:type owl:Restriction ;
       owl:onProperty <P2> ;
       owl:cardinality "7"^^xsd:NonNegativeInteger
    ] .
```

SPARQL Query

```
SELECT ?y7
WHERE { ?x rdf:type <A> .
        ?x <P1> ?y1 .
        ?x <P2> ?y2 .
        ?x <P3> ?y3 .
        ?x <P4> ?y4 .
        ?x <P5> ?y5 .
        ?x <P6> ?y6 .
        ?x <P7> ?y7 }
```

Expected behavior: As shown in the above query, *functional* properties (`owl:FunctionalProperty` construct) or restricted through **cardinality** constraints may appear in a large *star query*. In this case, the optimizer should recognize this schema information and apply the appropriate optimizations. This information could

take precedence over available *statistics* or other *heuristics* that operate on the *query form* (i.e., number of triple patterns, type of predicates used in the triple patterns).

More specifically, since property P3 is a functional property, it is most selective, so the triple pattern that involves P3 should be considered first in the query plan (in order to reduce drastically the size of intermediate results). In addition, since properties P2, P4 and P5 are restricted through cardinality constraints, their respective triple patterns should be evaluated in the correct order, meaning first the triple pattern that involves P4, then P5 and last P2.

6.2 Intersection of Classes (owl:intersectionOf)

OWL construct owl:intersectionOf is used to define a class by applying set theoretic intersection on the instances of user defined classes. Hence, if there is a triple pattern in a *join query* that requests instances of this complex class, in conjunction with triple patterns that request instances of (possibly both) intersected class(es), this should be considered first in the join plan, since this triple pattern will be more selective than the remaining ones.

Preconditions

```
<C> owl:intersectionOf ( <C1> <C2> ) .
```

SPARQL Query

```
SELECT ?x
WHERE { ?x rdf:type <C> .
        ?x <P1> ?y .
        ?y rdf:type <C1> .
        ?y <P2> ?z .
        ?z rdf:type <C2> }
```

Expected behavior: The triple pattern (?x, rdf:type, C) has a higher selectivity than the triple patterns involving variables ?y and ?z, as class C is the **intersection** of classes C1 and C2. Properties P1 and P2 have low selectivity (in the sense that they are common to many instances of the dataset). Hence, the query plan should be adapted accordingly. This should be understandable immediately by the optimizer, instead of having to resort to cost estimations.

6.3 Union of Classes (owl:unionOf)

OWL construct owl:unionOf is used to define complex classes by *unioning* the instances of user defined classes. The query that can be used to stress the optimizer is similar to the one that is specified for class intersection. Here, triple patterns that request for istances of the class defined by union are less selective than others hence, they should be the last considered in a query plan.

Preconditions

```
<C> owl:unionOf ( <C1> <C2> ) .
```

SPARQL Query

```
SELECT ?x
WHERE { ?x rdf:type <C> .
        ?x <P1> ?y .
        ?y rdf:type <C1> .
        ?y <P2> ?z .
        ?z rdf:type <C2> }
```

Expected behavior: The triple patterns involving ?y and ?z have a higher selectivity than the one involving variable ?x, as the class ?C is the *union* of classes C1 and C2; if properties P1 and P2 are less selective that rdf:type property, then the optimizer should build a query plan where these triple patterns are executed first.

6.4 Hierarchy of Classes (rdfs:subClassOf)

Preconditions

```
<A> rdfs:subClassOf <B> .
```

SPARQL Query

```
SELECT ?y1 ?y2 ?y3
WHERE { ?x rdf:type <A> .
        ?x rdf:type <B> .
        ?x <P> ?y1 .
        ?x <P1> ?y2 .
        ?x <P2> ?y3 }
```

Expected behavior The triple pattern involving class A has higher selectivity than the one involving class B, as A is a *subclass* of B and according to the semantics of the rdfs:subClassOf RDFS construct, the latter has less instances than B. Hence, if the optimizer considers this schema information should construct a plan that evaluates first the triple pattern (?x, rdf:type, A) and does not evaluate (eliminates) triple pattern (?x, rdf:type, B). The planner could consider this information instead of resorting simply to the use of statistics.

6.5 Hierarchy of Properties (rdfs:subPropertyOf)

Preconditions

```
<P1> rdfs:subPropertyOf <P2> .
```

SPARQL Query

```
SELECT ?y ?z
WHERE { ?x <P1> ?y .
        ?x <P2> ?z }
```

Expected behavior This test is similar to the rdfs:subClassOf test discussed in Section 6.4. Triple pattern (?x, P2, ?z) is less selective than (?x, P1, ?y) given the fact that P1 is a subproperty of P2. Hence, the query plan should be such that the former triple pattern should be evaluated before the latter.

7 REASONING: ADVANCED REASONING TESTS

In this Chapter we discuss tests that determine how an underlying engine could take advantage of schema information in order to make better choices on building the query plans. Each query addresses a certain class of challenges in situations in which the schema is present; note that constructing appropriate datasets that exhibit the necessary characteristics in order to prove the intended effect of the use of the constructs is a difficult and open problem.

7.1 Optimized Inference (rdfs:subClassOf, owl:allValuesFrom)

In this Section we consider a combination of `rdfs:subClassOf` and `owl:allValuesFrom` constructs and formulate a query that would test the optimizer's ability to use the appropriate schema information in order to find an interesting or non-trivial plan.

Preconditions

```
<B> rdfs:subClassOf <B1> .
<B1> rdfs:subClassOf <B2> .
<B2> rdfs:subClassOf <B3> .
<B3> rdfs:subClassOf <B4> .
<B4> rdfs:subClassOf <B5> .
<A> rdfs:subClassOf [
       rdf:type owl:Restriction ;
       owl:allValuesFrom <B5> ;
       owl:onProperty <P>
    ] .
```

SPARQL Query

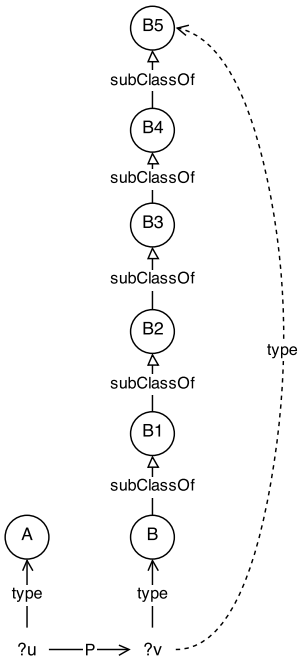
```
SELECT ?v
WHERE { ?u P ?v .
        ?u rdf:type <A> .
        ?v rdf:type <B5> .
        ?v rdf:type <B> }
```

Expected behavior: The above query requests the instances of class `B`, bound to variable `?v`. Note from the set of triples that exist in the dataset, that there are two ways to infer in a *backward reasoner* the instances of class `B`: one through the `owl:allValuesFrom` constraint and another through the transitivity of `rdfs:subClassOf`. The triples that we consider in this test case, define a class hierarchy with its root being class `B5`. We also define class `A` to be the set of instances, where all its instances, take all their values for property `P` from class `B5`.

As `B5` is sufficiently high in the hierarchy and `B` is sufficiently low, then it may be expensive to compute that `B` is a subclass of `B5`, and therefore the optimizer should decide to compute the inference via `owl:allValuesFrom`.

The opposite may be true if `B` and `B5` were sufficiently "close" in the hierarchy, especially given the fact that subsumption-related inference is probably the most optimized type (due to its widespread use).

Note here, that this test is only relevant for backward reasoning systems, where the triple $(?v, rdf:type, B5)$ does not exist in the dataset.



7.2 Redundant Triple Pattern Elimination (owl: intersectionOf)

In this Section we discuss an example of a query where the optimizer can take advantage the owl: intersectionOf OWL construct in order to eliminate unecessary triple patterns from the query. This optimization is rather useful since the optimizer will not perform uneccessary joins in order to evaluate a join query.

Preconditions

```
<C> owl:intersectionOf ( <C1> <C2> <C3> <C4> ).
```

SPARQL Query

```
SELECT ?x
WHERE { ?x rdf:type <C> .
        ?x rdf:type <C1> .
        ?x rdf:type <C2> .
        ?x rdf:type <C3> .
        ?x rdf:type <C4> }
```

Expected behavior: The query given above is a star join query that involves four triple patterns. The query actually asks for *all instances of class* C. A forward reasoner, that is one that materializes beforehand the closure of the dataset, should eliminate all triple patterns except (?x, rdf: type, C) in order to answer the query.

7.3 Search Space Pruning (rdfs: subClassOf)

Preconditions

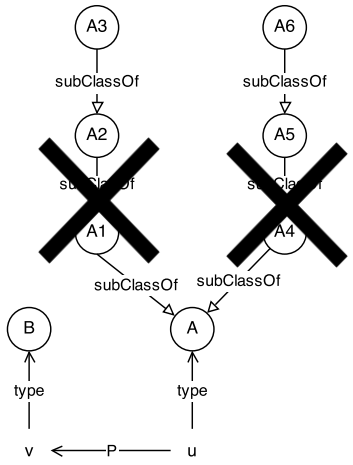
```
<A1> rdfs:subClassOf <A> .
<A2> rdfs:subClassOf <A1> .
<A3> rdfs:subClassOf <A2> .
<A4> rdfs:subClassOf <A> .
```

```
<A5> rdfs:subClassOf <A4> .
<A6> rdfs:subClassOf <A5> .
```

SPARQL Query

```
SELECT ?c
WHERE { ?x rdf:type ?c .
        ?x <P> ?y .
        FILTER NOT EXISTS { ?x rdf:type <A> } }
```

Expected behavior: The dataset considered contains a hierarchy of length 4, where A is the root of the class hierarchy. The query requests for all classes in the dataset that have some instance that is not an instance of class A. This test is mostly relevant for backward reasoning systems where the triples that potentially do not belong to the result set have to be computed at query time. An optimizer for every non-solution class should prune its subtree from the search space. So, as A is not a solution, the subtree of the class hierarchy rooted in A should be pruned. Queries over ontologies with many classes and deep hierarchies can gain the maximum advantage from this optimization.



7.4 Star Query Transformation owl:SymmetricProperty

Preconditions

```
<P5> rdf:type owl:SymmetricProperty .
```

SPARQL Query

```
SELECT ?y ?y1 ?y2 ?y3 ?y4
WHERE { ?x <P1> ?y .
        ?x <P2> ?y1 .
        ?x <P3> ?y2 .
        ?x <P4> ?y3 .
        ?y4 <P5> ?x }
```

Expected behavior: A smart optimizer should take advantage of schema information that property P5 is a symmetric one when constructing the query plan. Hence triple pattern (?x, P5, ?y4) should be considered instead of (?y4, P5, ?x). As a result, a star-query should be executed along with the optimizations of such a query.

This test is mostly relevant for systems that perform materialization (forward reasoning). Backward reasoning systems should also report some difference, but it depends on the reasoner's performance.

7.5 Intermediate Results Reduction: owl:sameAs

Preconditions

```
<x1> owl:sameAs <x2> .
<x1> owl:sameAs <x3> .
<x3> owl:sameAs <x4> .
<x2> owl:sameAs <x5> .
```

SPARQL Query

```
SELECT ?y
WHERE { ?x <P> ?y .
        ?x rdf:type <C> .
        ?y rdf:type ?y2 .
        ?y2 rdfs:subClassOf ?y3 .
        ?y3 rdfs:subClassOf ?y4 .
        ?y4 rdfs:subClassOf <B> }
```

Expected behavior: Suppose that x1, x2, x3, x4 and x5 are all related with the property P and their values (variable ?y), among with few others, are all in the result set of the above query.

A smart optimizer should take the advantage of owl:sameAs properties for individuals x1,. . .,x5. So, in the execution of the query, only one of the aforementioned instances should be considered.

This test, is mostly relevant for systems that perform materialization (forward reasoning). Backward reasoning systems should also report some difference, but it depends on the reasoner's performance.

7.6 Cardinalities Estimation: owl:TransitiveProperty

Preconditions

```
<P> rdf:type owl:TransitiveProperty .
```

SPARQL Query

```
SELECT ?y ?y1 ?y2 ?y3
WHERE { ?x rdf:type <C> .
        ?x <P> ?y .
        ?x <P1> ?y1 .
        ?x <P2> ?y2 .
        ?x <P3> ?y3 }
```

Expected behavior The query proposed here considers a transitive property `P`. Suppose that `P` is more selective than the properties `P1`, `P2`, and `P3`. If a backward reasoner, that makes use of statistics to produce a query plan, does not consider the information that `P` is a transitive property, then it might produce a query plan where the triple pattern containing property `P` is evaluated first. Given the fact that `P` is transitive, the result of backward reasoning will be a large number of triples that would make `P` eventually less selective than the other ones (since the property's selectivity would be cancelled out). Hence, it would be best (especially in graphs with large connected paths) to push the evaluation at the top of the query plan.

8 BENCHMARKS FOR REASONING WITH BUSINESS RULES

In this chapter we discuss that standard powerful OWL [23] constructs such as *class* and *individual* cannot model *conjunctive properties* [51] or other more complex *business rules* that are necessary in a number of application domains. These can be expressed with custom rules or vendor-specific extensions. Some times the same functionality can be captured with SPARQL queries, but most often this leads to very complex queries that have bad performance. Authors in [55] show conditions under which conjunctive properties can be added without increase in complexity. In particular, OWL2RL can be extended with role conjunctions without any restrictions or increase in complexity, and [35] proposes extending OWL with such capabilities in OWL3. But at present, such property constructs are not available in OWL.

The rest of the chapter is structured as follows: Section 8.1 provides a motivating example from the Cultural Heritage Domain; we discuss and compare several rule languages in Section 8.2. Last Section 8.3 presents several scenarios and the rules and queries that implement those.

8.1 Motivating Example: Complex Reasoning with a Cultural Heritage ontology

In this section we discuss an example from the cultural heritage (CH) domain that shows the use of rules that go beyond OWL. CIDOC CRM (ISO 21127:2006) [1] is an ontology for describing entities, properties and relationships appearing in CH documentation, art, history and archaeology. CIDOC CRM promotes shared understanding by providing an extensible semantic framework that any CH information can be mapped to.

CIDOC CRM finds increasing use in the CH domain, including applications to archaeology (CRM-EH), museum information (British Museum Ontology[1]), bibliographic data (FRBRoo), periodical publications (PRESSoo[2]), tracking the derivation of ancient sayings and wisdoms (SAWS), etc. Mappings to CIDOC CRM[3] have been developed for many metadata standards in these domains, including LIDO[4] for museums, EAD[5] for archives, Europeana's EDM[6] for aggregation among others.

CIDOC CRM has been used in CLAROS[7] which is an interdisciplinary research federation involving 401k objects (19M triples), the British Museum collection[8] that contains 2M objects (916M RDF triples), the Polish Digital Library[9] a national aggregation of museum and library objects using FRBRoo, and CRM that contains 3.1M objects (535M RDF triples). More information regarding the data statistics can be found in [7].

8.1.1 Fundamental Relations

An important question in the CH domain, is how a user can search through the complex node and edge labeled data graphs of CIDOC CRM, since the number of possible combinations is staggering. Authors in [74] describe an approach that "compresses" the semantic network by mapping many CIDOC CRM entity classes to a few *Fundamental Concepts* (FC), and mapping whole networks of CRM properties to fewer *Fundamental Relations* (FR); these fundamental concepts and relations can serve as a "search index" over the CIDOC CRM graphs and allow the user to use simpler queries to retrieve information from these graphs.

Table 8.1 shows a subset of the FRs (out of total 114 FRs defined over all combinations of FCs); the interested reader can find more information regarding fundamental concepts and relations in [75]. Example 1 discusses the Fundamental Relation "Thing from Place".

[1] http://semanticweb.com/tag/british-museum

[2] http://www.issn.org/the-centre-and-the-network/our-partners-and-projects/pressoo/

[3] http://cidoc-crm.org/crm_mappings.html

[4] http://network.icom.museum/cidoc/working-groups/data-harvesting-and-interchange/what-is-lido/

[5] http://www.loc.gov/ead/

[6] http://pro.europeana.eu/edm-documentation

[7] http://www.clarosnet.org/

[8] http://collection.britishmuseum.org

[9] http://dl.psnc.pl

Domain	Range (query parameter)				
	Thing	Actor	Place	Event	Time
Thing	2.is part of 3.is similar or the same with 4.has met 5.from 6.is origin of 8.refers to 9.is referred by	4.has met 5.from 8.refers to 9.is referred by	4.from 8.refers to 9.is referred to at	4.from 8.refers to	4.from
Actor	4.has met 6.is creator or provider of 8.refers to 9.is referred by	2.is member of 4.has met 5.has parent or founder 6.is parent or founder of 8.refers to 9.is referred by	4.has met 5.from 8.refers to 9.is referred to at	4.has met 8.refers to 6.has met	8.refers to 6.has met 4.from
Place	5.is origin of 8.refers to or is about 9.is referred by	5.is origin of 8.refers to or is about 9.is referred by	2.is part of 5.is origin of	9.is referred by 5.is origin of	7.at
Event	5.is origin of 9.is referred by 8.refers to or is about	4.from 9.is referred by 8.refers to or is about 6.has met	8.refers to or is about 7.at	8.refers to or is about 2.is part of	8.refers to or is about 7.at
Time	5.is origin of	5.is origin of	5.is origin of	5.is origin of	2.is part of

Table 8.1: A subset of CIDOC CRM Fundamental Concepts and Relations

Example 1 One of the considered FRs is "Thing from Place" used to capture how a Thing's *origin* can be related to Place. CIDOC CRM includes *part of* hierarchies for objects, events, actors (groups), etc. The "Thing from Place" FR is defined as follows (where "*" indicates recursion over the *part of* hierarchies):

Thing (part of another)* is considered to be from Place if it:

1. is formerly or currently located at Place (falling in another)*
2. **OR** was brought into existence (produced/created) by an Event (part of another)*
   (a) that happened at Place (falling in another)*
   (b) **OR** was carried out by an Actor (who is member of a Group)*
      i. who formerly or currently has residence at Place (falling in another)*
      ii. **OR** was brought into existence (born/formed) by an Event (part of another)* that happened at Place (falling in another)*
3. **OR** was moved to/from a Place (falling in another)*
4. **OR** changed ownership through an Acquisition (part of another)*
   (a) that happened at Place (falling in another)*

Using the CIDOC CRM ontology, the above definition is translated to a fundamental relation shown in Listing 8.1 where `Enn` are classes, `Pnn` are properties, and `FC70_Thing` is a custom class indicating a cataloged object. Figure 8.1 gives a graphical representation of the above Fundamental Relation. The SPARQL implementation of the previous fundamental relation in SPARQL is straightfoward and shown in Listing 8.2. One can easily see that this query is very complex

and expensive, especially when one needs to combine this FR with others to form more elaborate queries.

Listing 8.1: ”FR: Thing from Place”

```
FC70_Thing -(P46i_forms_part_of* | P106i_forms_part_of* |
             P148i_is_component_of*)-> FC70_Thing:
  {FC70_Thing -(P53_has_former_or_current_location |
                P54_has_current_permanent_location)-> E53_Place:
    {E53_Place -P89_falls_within*-> E53_Place}
  OR FC70_Thing -P92i_was_brought_into_existence_by-> E63_Beginning_of_Existence:
    {E63_Beginning_of_Existence -P9i_forms_part_of*-> E5_Event:
      {E5_Event -P7_took_place_at-> E53_Place:
        {E53_Place -P89_falls_within*-> E53_Place}
      OR E7_Activity -P14_carried_out_by-> E39_Actor:
        {E39_Actor -P107i_is_current_or_former_member_of* -> E39_Actor:
          {E39_Actor -P74_has_current_or_former_residence  -> E53_Place:
            {E53_Place -P89_falls_within*-> E53_Place}
          OR E39_Actor -P92i_was_brought_into_existence_by-> E63_Beginning_of_Existence:
            {E63_Beginning_of_Existence -P9i_forms_part_of*-> E5_Event:
              {E5_Event -P7_took_place_at-> E53_Place:
                {E53_Place -P89_falls_within* -> E53_Place}}}}}}}
  OR E19_Physical_Thing  -P25i_moved_by-> E9_Move:
    {E9_Move -(P26_moved_to | P27_moved_from)-> E53_Place:
      {E53_Place  -P89_falls_within*-> E53_Place}}
  OR E19_Physical_Object -P24i_changed_ownership_through-> E8_Acquisition:
    {E8_Acquisition -P9i_forms_part_of*-> E5_Event:
      {E5_Event -P7_took_place_at-> E53_Place:
        {E53_Place -P89_falls_within*-> E53_Place}}}}
```

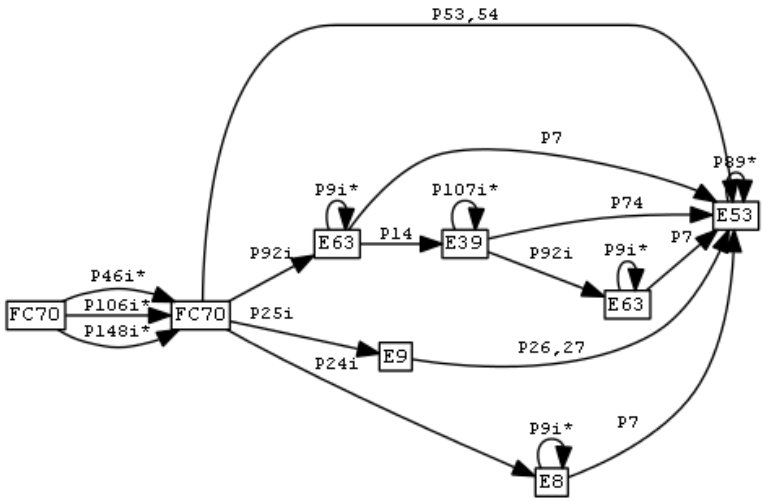


Figure 8.1: FR: Thing from Place

FRs can also be implemented as OWLIM Rules[10]; the details of the implementation are described in [4, 7]. and is based on the decomposition of the CIDOC CRM framework into growing fragments called *sub-FRs* that can be reused by multiple FRs, an approach which provides some performance benefits to the query engine as discussed in [7]. OWLIM Rules are used to overcome OWL2's inability to define conjunctive properties;

Out of all defined FRs, authors in [7] discuss implementations for 19 FRs of Thing. These FRs involve 44 CIDOC CRM properties and 26 sub-FRs. If we consider CIDOC CRM properties as input relations, sub-FRs as intermediate relations, and FRs as output relations, the intermediate relations constitute 29% of all relations. Authors in [7] present a dependency diagram of these relations, showing the level of reuse. The proposed implementation used 10 axioms and 86 rules; an improved implementation may refactor this to use fewer rules and a lot more axioms, which will improve maintainability and flexibility.

[10]OWLIM was recently renamed to Ontotext GraphDB

Listing 8.2: ”SPARQL representation for FR “Thing from Place”

```
SELECT ?t ?p2 {
?t a FC70_Thing.
  ?t (P46i_forms_part_of* | P106i_forms_part_of* | P148i_is_component_of*) ?t1.
  {?t1 (P53_has_former_or_current_location | P54_has_current_permanent_location) ?p1}
  UNION
  {?t1 P92i_was_brought_into_existence_by ?e1. ?e1 P9i_forms_part_of* ?e2.
      {?e2 P7_took_place_at ?p1}
      UNION
      {?e2 P14_carried_out_by ?a1.
        ?a1 P107i_is_current_or_former_member_of* ?a2.
          {?a2 P74_has_current_or_former_residence ?p1}
          UNION
          {?a2 P92i_was_brought_into_existence_by ?e3. ?e3 P9i_forms_part_of* ?e4.
           ?e4 P7_took_place_at ?p1}}}
  UNION
  {?t2 P25i_moved_by ?e5. ?e5 (P26_moved_to | P27_moved_from) ?p1}
  UNION
  {?t2 P24i_changed_ownership_through ?e6.
    ?e6 P9i_forms_part_of ?e7. ?e7 P7_took_place_at ?p1}.
?p1 P89_falls_within* ?p2}
```

Example 2

Figure 8.2 shows the graphical representation of FR "Thing created by Actor" (FR92i_created_by), which we define as "Thing (or part/inscription thereof) was created or modified/repaired by Actor (or a group it is a member of)". This FR involves the following CIDOC CRM source properties:

- `P46_is_composed_of`, `P106_is_composed_of`, `P148_has_component` that navigate the object part hierarchy;
- `P128_carries` that support the transition from object to the inscription that it carries;
- `P31i_was_modified_by` (includes `P108i_was_produced_by`), `P94i_was_created_by` that refer to the modification or production of some physical thing, the creation of conceptual thing as given by its inscription
- `P9_consists_of` that allows one to navigate the hierarchy of events
- `P14_carried_out_by`, `P107i_is_current_or_former_member_of` that refer to the agent that performs an event and the groups she is member of.

It uses sub-FR `FRT_46_106_148_128` (the first loop in Figure 8.2).

```
FRX92i_created := (FC70_Thing) FRT_46_106_148_128* / (P31i | P94i) / P9*
```

We first define a sub-FR `FRX92i_created` that extends to the second node `(Modification/Creation)` and includes the `P9` loop; it is used in the implementation of FR “Thing created by Actor”.

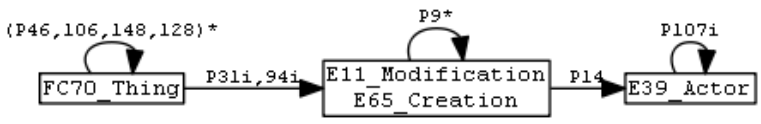


Figure 8.2: FR: Thing created by Actor

Listing 8.3 shows the implementation of sub-FR `FRX92i_created` used in the implementation of FR “Thing created by Actor” implemented with the 5 OWLIM Rules 5 shown in Listing 8.3.

Listing 8.3: sub-FR FRX92i_created

```
x <rdf:type> <rso:FC70_Thing>; x <crm:P31i_was_modified_by> y => x <rso:FRX92i_created> y
x <rdf:type> <rso:FC70_Thing>; x <crm:P94i_was_created_by> y => x <rso:FRX92i_created> y
x <rso:FRT_46_106_148_128> y; y <crm:P31i_was_modified_by> z => x <rso:FRX92i_created> z
x <rso:FRT_46_106_148_128> y; y <crm:P94i_was_created_by> z => x <rso:FRX92i_created> z
x <rso:FRX92i_created> y; y <crm:P9_consists_of> z => x <rso:FRX92i_created> z
```

8.2 Rule Languages

In this section we describe briefly some existing rule languages[11].

8.2.1 OWLIM Rules

OWLIM Rules [19] are the simplest variety of rules discussed in this Section. They use simple unification: a set of premise triple patterns is checked against the repository, and if it matches, a set of consequent triples is inferred and stored in the repository, where variables in the consequents are instantited from the premises (or emitted as blank nodes if not present in the premises). Nevertheless, OWLIM rules are powerful enough to implement OWL2 RL and OWL2 QL [9] (e.g., see Section 8.3.3 for the implementation of `owl:propertyChainAxiom`). Custom rules can be used for various purposes (see Section 8.1 for an example). The OWLIM Rules syntax is verbose (one line per premise/conclusion), so we use a simpler syntax (one line per rule, see Listing 8.3) that we expand to the verbose syntax with a simple script.

A benefit of OWLIM Rules is that they are "reversible": when a triple is deleted, all relevant rules are checked. If an inferred triple matches the consequences and there are no other triples to support it, the triple is retracted as well (see Section *Retraction of assertions* in [19] for details). This limited form of backtracking supports incremental deletion and is extremely important for use cases with high update rate. OWLIM Rules are simple by design, in order to support this feature. Some disadvantages of OWLIM Rules include:

- Rules are not very flexible: if a rule is changed, consequences are not recomputed automatically. In contrast, if the schema changes, then the consequences of rules are recomputed.
- These rules are proprietary to OWLIM and are not portable to other systems.
- These rules do not support arithmetics or real negation. A rule may pose variable inequalities (eg. that a rule variable is not bound to a specific class), but cannot check for lack of certain triples (eg. that a resource does belong to a specific class).

8.2.2 SPIN Rules

SPARQL Inferencing Notation (SPIN[12]) is an approach for representing SPARQL rules and constraints on Semantic Web models. SPIN also provides meta-modeling capabilities that allow users to define their own SPARQL functions and query templates. SPIN can be used to:

- calculate the value of a property based on other properties - for example, area of a geometric figure as a product of its height and width, age of a person as a difference between today's date and person's birthday, a display name as a concatenation of the first and last names
- define functions through SPARQL queries, which can be used in other queries, just like the standard SPARQL and XPath functions
- isolate a set of rules to be executed under certain conditions - for example, to support incremental reasoning, to initialize certain values when a resource is first created, or to drive interactive applications
- perform constraint checking with closed world semantics and automatically raise inconsistency flags when currently available information does not fit the specified integrity constraints

Rules are implemented with SPARQL CONSTRUCT or UPDATE queries, and constraints are implemented with ASK or CONSTRUCT queries. SPIN Templates also make it possible to define such rules and constraints in higher-level domain specific languages, so that rule designers do not need to work with SPARQL directly.

The benefits of SPIN include:

- an extensive library of common functions. These include SPIN Standard Modules Library (SP, SPIF, SPL), SPIN JavaScript Functions (SPINx), and SPIN Result Sets, Tables and Spreadsheets Vocabulary (SPR and SPRA).
- an open source implementation, the SPIN API[13]. It is built on top of the Jena API and provides *i)* converters between textual SPARQL syntax and the SPIN RDF Vocabulary *ii)* SPIN-based constraint checking

[11] Missing from this overview are SWRL and RIF, as we are not aware of practical examples using these languages.

[12] `http://spinrdf.org/`

[13] `http://topbraid.org/spin/api/`

engine *iii)* SPIN-based inferencing engine *iv)* user-defined SPIN functions using Jena/ARQ and *v)* user-defined SPIN templates.

SPIN underpins a significant part of TopQuadrant's technologies, including *(a)* SPARQLMotion (transformations) *(b)* SWP (Semantic Web pages for making UIs) and *(c)* SWA (Semantic Web applications, made from components and widgets). These are integrated and used extensively in TopBraid (Semantic Web IDE) and TopBraid Live (Server Platform). Therefore the SPIN functions and SPIN API are regularly updated and maintained. Possible disadvantages of SPIN rules are related to performance over large (disk-based) databases and include the following:

- *evaluation of SPIN functions*: when SPIN functions are evaluated, a lot of SPARQL queries are executed (not only the original queries and templates, but also all user functions defined with SPARQL). These can be executed efficiently in in-memory databases, but may be a problem for large disk-based databases. In particular, smart caching strategies and the locality of data needs to be exploited.
- *efficient updates*: SPIN API cannot incrementally delete inferred triples when a premise is removed. Authors in [38] explain *"The inferred triples usually go into a dedicated inferences graph that can have its own life cycle, eg. to reset and re-run inferences when new information comes in that contradicts prior knowledge"* (see 8.2.1 for an example of such capability).
- *recursion*: if the SPIN ruleset (or axioms) involve recursion (eg transitive closure and similar rules), the ruleset needs to be run repeatedly, until no new triples are inferred. But the SPIN API does not use RETE or a similar algorithm to exploit locality and incrementality of added triples [38]: *"We do not have a (RETE-like) rule chaining implemented but this could be achieved, at least for a subset of the SPIN expressivity"*. It also explains the reason why: *"Complex WHERE clauses with difficult patterns are likely not implementable [in a RETE like fashion] - that's why most other rule languages have a limited expressivity"*.

8.3 Scenarios of Business Rule

This section provides specific business scenarios, involving specific rule sets, queries, and in some cases vendor extensions.

8.3.1 Extended Property Constructs

As explained in the Introduction of Chapter 8, OWL2 has certain limitations when it comes to defining properties. In order to present this claim, we will give a short introduction (notations and semantics) for the property constructs that we consider in this work. Below and in Table 8.2, `pN` indicates premises, `q` a conclusion.

- *prop path* is a SPARQL 1.1 property path: `'^'` is *inverse*, `'|'` is *disjunction* (parallel composition), `'/'` is *chain* (sequential composition), `'+'` is recursion (Kleene closure).
- `:=` means *equivalentProperty*, `=>` means *subPropertyOf* (correspondingly `'<='` means *super-property*).

Table 8.2 provides illustrations of the property paths where the *conclusions* of the corresponding *prop path*'s are shown in red. The Turtle representation of these OWL constructs is shown in Listing 8.4. Note that the semantics thereof are given in Chapter 3.

Listing 8.4: Turtle representation of OWL constructs in Table 8.2

```
q a owl:SymmetricProperty.
q owl:inverseOf p.
p1 rdfs:subPropertyOf q. p2 rdfs:subPropertyOf q.
q owl:propertyChainAxiom (p1..pN).
p rdfs:subPropertyOf q. q a owl:TransitiveProperty.
```

OWL constructs `owl:FunctionalProperty` (respectively `owl:InverseFunctionalProperty`) are not used for defining new properties, but to restrict the cardinality of property instances with the same subject (respectively object).

Table 8.3 introduces some new extensions. Authors in [5] discuss additional potentially useful property constructs, illustrated with cases from the semantic representation of the British Museum collection and the Getty

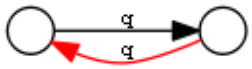
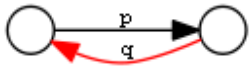
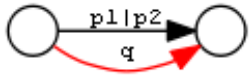
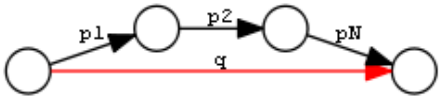
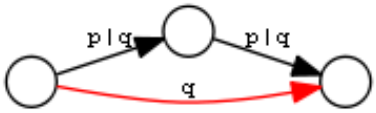
Construct	*prop path*	*illustration*
Symmetric	q := ^q	
Inverse	q := ^p	
Disjunction	q <= p1 | p2	
Chain	q <= p1/../pN	
Transitive	q <= p+	

Table 8.2: Standard OWL2 Property Constructs

Research Institute Vocabularies. [6] describes in detail the implementation of a few of these constructs, and their application to inferencing for the Getty Vocabularies.

- `r` indicates a *restriction property* (which is just another premise), `tN` is a type
- `p & r` is a *conjunction* (property restriction): holds between two nodes when both properties connect the same nodes
- `[t1] p [t2]` is *type restriction*: holds when the subject has type `t1` and the object has type `t2` (shown inside the node).

A possible representation of these constructs as Turtle axioms is shown in Listing 8.5. It uses the prefix `ptop:` that stands for Ontotext's PROTON ontology. `ptop:transitiveOver` has been part of PROTON since 2008, the other constructs are new; `x` stands for a node holding the structure together (a blank node can be used). Of course, the same data can be captured in other structures, so many other representations are possible.

Listing 8.5: Property Extensions

```
q ptop:transitiveOver p.
x a ptop:PropChain; ptop:premise1 p1; ptop:premise2 p2; ptop:conclusion q.
x a ptop:PropRestr; ptop:premise p; ptop:restricton r; ptop:conclusion q.
x a ptop:TypeRestr; ptop:premise p; ptop:type1 t1; ptop:type2 t2; ptop:conclusion q.
```

8.3.2 Implementing Extended Property Constructs

The implementation of most of the above constructs as OWLIM rules is shown in [6]. We provide below examples of these implementations. For instance, `PropChain` is implemented as shown in Listing 8.6. `transitiveOver` can be implemented using `owl:propertyChainAxiom`, `PropChain`, or a dedicated rule as shown in Listing 8.7. We prefer the `PropChain` implementation for the reasons that will be explained in Section 8.3.3. For illustration purposes only, we show the implementation of `PropChain` as a SPIN rule. It closely follows the OWLIM rule as given in Listing 8.8. The other constructs can be implemented with OWLIM-Rules or SPIN in a similar fashion.

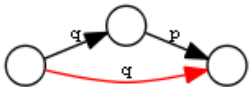
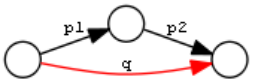
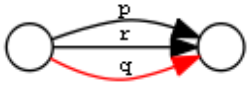
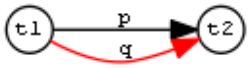
Construct	*prop path*	*illustration*
transitiveOver	q <= q / p	
PropChain	q <= p1 / p2	
PropRestr	q <= p & r	
TypeRestr	q <= [t1] p [t2]	

Table 8.3: Extended Property Constructs

Listing 8.6: Implementation of PropChain using OWLIM Rules

```
Id: ptop_PropChain
  t <ptop:premise1>   p1
  t <ptop:premise2>   p2
  t <ptop:conclusion> q
  t <rdf:type> <ptop:PropChain>
  x p1 y
  y p2 z
  ----------------
  x q z
```

Listing 8.7: `transitiveOver` Implementation

```
# q owl:propertyChainAxiom (q p).  # This is better:
[a ptop:PropChain; ptop:premise1 q; ptop:premise2 p; ptop:conclusion q].
```

Listing 8.8: Implementation of PropChain in SPIN

```
ptop:PropChain
  spin:rule [
    a sp:Construct;
    sp:text """
      CONSTRUCT {?x ?q ?z}
      WHERE {
         # "?this a ptop:PropChain" implied by the binding above
          ?this ptop:premise1   ?p1.
          ?this ptop:premise2   ?p2.
          ?this ptop:conclusion ?q.
          ?x ?p1 ?y.
          ?y ?p2 ?z}"""].
```

8.3.3 Two-Place (2-Place) Chains

In this Section we are going to discuss the implementation of chains of length 2 using 2-Place chain `PropChain`. Recall that `owl:propertyChainAxiom` allows one to define properties as a composition of others (see Section 3.9). The advantage of a 2-Place chain `PropChain` over the general `owl:propertyChainAxiom` is as follows:

1. a fixed-arity chain is easier to implement. A general chain is represented as an rdf:List, and the inferencer needs to unroll this list. A forward inferencer like OWLIM needs to make intermediate nodes and edges (see below)
2. most chains used in practice are two-place chains
3. any specific general chain can be implemented as a sequence of two-place chains. Eg `q <= p1/p2/p3` can be implemented as `q1 <= p1/p2. q <= q1/p3` as shown in Listing 8.9.

Listing 8.9: Implementing Three-Place Chain with Two-Place Chains

```
q owl:propertyChainAxiom (p1 p2 p3).
### Can be implemented as
[a ptop:PropChain; ptop:premise1 p1; ptop:premise2 p2; ptop:conclusion q1].
[a ptop:PropChain; ptop:premise1 q1; ptop:premise2 p3; ptop:conclusion q].
```

Regarding the first point above, the OWLIM rules implementing `owl:propertyChainAxiom` are as follows: The `[Context]` annotation matches, respectively inserts quads with a specific context (graph). The one used below is an internal graph that is not exposed to queries unless explicitly asked for.

```
prp_spo2_1
    p <owl:propertyChainAxiom> pc
    start pc last                   [Context <onto:_checkChain>]
    ----------------------------
    start p last

Id: prp_spo2_2
    pc <rdf:first> p
    pc <rdf:rest> t                 [Constraint t != <rdf:nil>]
    start p next
    next t last                     [Context <onto:_checkChain>]
    ----------------------------
    start pc last                   [Context <onto:_checkChain>]

Id: prp_spo2_3
    pc <rdf:first> p
    pc <rdf:rest> <rdf:nil>
    start p last
    ----------------------------
    start pc last                   [Context <onto:_checkChain>]
```

The correctness of this implementation is not obvious, so we give an example below. `Q <= P1/P2` is represented as a general chain in this way (the lists PC and PC1 are usually blank nodes):

```
Q owl:propertyChainAxiom PC.
PC a rdf:List; rdf:first P1; rdf:rest PC1.
PC1 a rdf:List; rdf:first P2; rdf:rest rdf:nil.
```

The rules fire in the following sequence (we show variable substitutions after the rule ID):

```
prp_spo2_3: pc=PC1, p=P2
  start1 P2  last1
  ----------------
  start1 PC1 last1 [Context <onto:_checkChain>]

prp_spo2_2: next=start1, t=PC1, last=last1, pc=PC, p=P1
  start P1 start1
  --------------
  start PC last1   [Context <onto:_checkChain>]

prp_spo2_1: start=start, pc=PC, last=last1, p=Q
  -------------
  start Q last1
```

So overall we get:

```
start  P1 start1
start1 P2 last1
---------------
start  Q last1
start  PC last1  [Context <onto:_checkChain>]
start1 PC1 last1 [Context <onto:_checkChain>]
```

The first conclusion is what we want, and the other two conclusions represent intermediate nodes and edges used for the inferencing. PC and PC1 are blank-node **properties**.

In contrast, a two-place chain is easy to implement and does not make such intermediate edges as shown below:

```
Id: ptop_PropChain
  t <ptop:premise1>   p1
  t <ptop:premise2>   p2
  t <ptop:conclusion> q
  t <rdf:type> <ptop:PropChain>
  x p1 y
  y p2 z
  ----------------
  x q z
```

Customized for BBC ontologies

Here we present an implementation of Two-Place Chains for the BBC use case scenario. For this purpose, we use the following axiom:

```
[a ptop:PropChain; ptop:premise1 cwork:thumbnail; ptop:premise2 cwork:altText;
                   ptop:conclusion ldbc:cworkThumbnailAltText].
```

Expected behavior

The new axiom should produce fewer triples, and/or run faster than the old one. Please note that we do not need to rewrite our axioms to use `PropChain`. If only two-place chains are used, `owl:propertyChainAxiom` can be converted to `PropChain` automatically with a simple rule like the following one:

```
Id: ptop_PropChainByPropertyChainAxiom
  p  <owl:propertyChainAxiom> l1
  l1 <rdf:first> p1
  l1 <rdf:rest>  l2
  l2 <rdf:first> p2
  l2 <rdf:rest>  <rdf:nil>
  ----------------
  t <rdf:type> <ptop:PropChain>
  t <ptop:premise1>   p1
  t <ptop:premise2>   p2
  t <ptop:conclusion> p
```

8.3.4 Better Transitive Properties

In this Section we discuss the `ptop:transitiveOver` construct that was introduced in Table 8.3; `q ptop:transitiveOver p` means that property `q` can be extended with another property `p` (on the right). For example, the inferencing of types along the class hierarchy can be expressed with a single axiom:

```
rdf:type ptop:transitiveOver rdfs:subClassOf.
```

Most often `owl:TransitiveProperty` is implemented as a self-chain using `owl:propertyChainAxiom`, following the rule PRP-TRP discussed in Table 3.7.

```
q owl:propertyChainAxiom (q q)
```

Therefore `transitiveOver` is a *generalization* of `owl:TransitiveProperty` as shown below:

```
q a owl:TransitiveProperty <=> q ptop:transitiveOver q
```

Note that this idea is analogous to `owl:inverseOf` being a *generalization* of `owl:SymmetricProperty` (which is a *self-inverse*):

```
q a owl:SymmetricProperty <=> q owl:inverseOf q
```

When implementing transitive closure, it is a good practice to keep the basic (*step*) property. As an example, SKOS [36] defines `skos:broader` as the *step*, and `skos:broaderTransitive` as its *closure*. As a counter-example, RDFS does *not* have step properties for `rdfs:subClassOf` and `rdfs:subPropertyOf`, both defined as reflexive and transitive properties. But it is often necessary to find the direct subclasses of a class, so one needs to use complex and expensive queries like the one listed below:

```
SELECT ?sub {
   ?sub rdfs:subClassOf pz:Pizza.
   FILTER NOT EXISTS {
      ?sub rdfs:subClassOf ?x .
      ?x rdfs:subClassOf pz:Pizza .
      FILTER (?x != ?sub && ?x != pz:Pizza)}}
```

For this reason, some frameworks support step properties for RDFS, such as `sesame:directSubClassOf`, `sesame:directSubPropertyOf` and `sesame:directType` in Sesame[14] if a `DirectTypeHierarchyInferencer` SAIL is deployed; those properties are also supported by OWLIM. These built-in predicates allow direct and more efficient querying in this case.

Not only `transitiveOver` is a *generalization* of `owl:TransitiveProperty`, but using it together with step properties potentially allows *more efficient* inferencing. Consider for instance the following example: the standard implementation of `broaderTransitive` in the SKOS ontology is:

```
skos:broader rdfs:subPropertyOf skos:broaderTransitive.
skos:broaderTransitive a owl:TransitiveProperty.
```

Now consider a chain of N `skos:broader` between nodes A and B. The same closure between A and B can be inferred from every split of the chain (there are $N-1$). So an inferencer may have to consider every split, although they all lead to the same conclusion.

The following axioms allow more efficient inferencing, since they seek to extend the chain only at the (right) end:

```
skos:broader rdfs:subPropertyOf skos:broaderTransitive.
skos:broaderTransitive ptop:transitiveOver skos:broader.
```

Customized for BBC ontologies

We illustrate here an implementation of `transitiveOver` for the BBC scenario. Consider the axiom

```
sport:directSubDisciplineOf rdfs:subPropertyOf sport:subDisciplineOf.
sport:subDisciplineOf ptop:transitiveOver sport:directSubDisciplineOf.
```

Expected behavior

Although the new axioms produce more triples (`sport:directSubDisciplineOf` is also repeated as `sport:subDisciplineOf`), they will potentially run faster than the ones shown in Table 3.7.

8.3.5 Interlinking Ambiguous Things in the Semantic Publishing Benchmark

The names of real-world things are often ambiguous. For example, the Wikipedia disambiguation page for Michael Jackson lists at least 30 people of that name, plus songs, movies, etc. In general, there are cases in the Semantic Publishing Domain which

- we want to interlink ambiguous things (with a property `ldbc:ambiguousWith`), to facilitate editorial actions to disambiguate them
- since the universe of *things* is very large, we should do this only for things that are tagged in a creative work.
- when a thing is disambiguated by an editor, he should check and eventually correct taggings of that thing in CreativeWorks, then add a `core:disambiguationHint`, at which point `ldbc:ambiguousWith` is deleted.

[14] Sesame User Guide: `http://openrdf.callimachus.net/sesame/2.7/docs/users.docbook?view`

This complex logic can be implemented as a mix of axioms and SPIN rules as shown below:

```
# Capture all labels of a thing. rdfs:label is also a good candidate to add.
core:shortLabel     rdfs:subPropertyOf ldbc:anyLabel.
core:preferredLabel rdfs:subPropertyOf ldbc:anyLabel.

# We want it both ways
ldbc:ambiguousWith a owl:SymmetricProperty.

core:Thing spin:rule
  [ # Interlink ambiguous things
    a sp:Construct;
    sp:text """
      CONSTRUCT {?this ldbc:ambiguousWith ?that}
      WHERE {
        FILTER EXISTS {?cwork cwork:tag ?this}
        ?this ldbc:anyLabel ?label.
        ?that ldbc:anyLabel ?label.
        FILTER NOT EXISTS {?this core:disambiguationHint ?hint}
    """],
  [ # Disambiguate interlinked ambiguous things
    a sp:Modify;
    sp:text """
      DELETE {?this ldbc:ambiguousWith ?that}
      WHERE {?this ldbc:ambiguousWith ?that.
        FILTER EXISTS {?this core:disambiguationHint ?hint}}
    """].
```

8.3.6 Classifying CreativeWorks in the Semantic Publishing Benchmark

A business-meaningful notion such as "International music video programme" can be computed based on attributes of creative works (see Figure A.1). We first define some shortcut properties as `owl:propertyChainAxiom` (or `PropChain`, see Section 8.3.3) as given below:

```
ldbc:primaryContentProduct owl:propertyChainAxiom (bbc:primaryContentOf bbc:product).
ldbc:primaryContentTopic   owl:propertyChainAxiom (bbc:primaryContentOf core:primaryTopic).
```

The above can be implemented in SPARQL as follows:

```
CONSTRUCT {?cwork a ldbc:InternationalMusicVideoProgramme}
WHERE {?cwork
  a cwork:Programme;
  cwork:audience cwork:InternationalAudience;
  cwork:primaryFormat cwork:VideoFormat;
  ldbc:primaryContentProduct bbc:Music.
}
```

Their implementation with OWL2 restrictions is the following:

```
ldbc:InternationalMusicVideoProgramme a owl:Class;
  rdfs:subClassOf cwork:CreativeWork;
  owl:intersectionOf (
    cwork:Programme
    [a owl:Restriction;
       owl:onProperty cwork:audience; owl:hasValue cwork:InternationalAudience]
    [a owl:Restriction;
       owl:onProperty cwork:primaryFormat; owl:hasValue cwork:VideoFormat]
    [a owl:Restriction;
       owl:onProperty ldbc:primaryContentProduct; owl:hasValue bbc:Music]
  )
```

8.3.7 Validating Creative Works in the Semantic Publishing Benchmark

A similar approach to the one presented in Section 8.3.6 could be used to check for valid combinations of characteristics. For example, if a creative work has a `Sports-Stats` `cms:locator` (i.e. a representation of the work can be found in BBC's **sports-stats** system) and `bbc:primaryContent`, then the content's `bbc:product` must be `Sports`.

This constraint can be implemented in SPIN, using a `constraint` that constructs a `ConstraintViolation` with a useful description of the problem. We can even define a suggested fix (which is to insert the missing `bbc:product` triple) with the SPIN rule below:

```
cwork:CreativeWork
  spin:constraint [a sp:Construct ;
     sp:text """
       CONSTRUCT {[a spin:ConstraintViolation ;
         spin:violationRoot ?this ;
         spin:violationPath [a sp:SeqPath;
                             sp:path1 bbc:primaryContentOf; sp:path2 bbc:product];
         rdfs:label "Sports-Stats locator implies that
                     primaryContentOf/product must be Sport";
         spin:fix [a sp:Modify; sp:insertPattern
            ([rdf:subject ?doc; rdf:predicate bbc:product; rdf:object bbc:Sport])]]}
       WHERE {
         ?this cms:locator/rdf:type cms:Sports-Stats;
           bbc:primaryContentOf ?doc.
         FILTER NOT EXISTS {?doc bbc:product bbc:Sport}}"""]
```

Here we do not use `ldbc:primaryContentProduct` because we need to check the existence of the intermediate node `?doc`.

8.3.8 Faceting for Co-occurrence

Faceted Search is a commonly used user interface paradigm for exploring a large dataset of entities:

- the user is presented with a set of *facets*, which are characteristics shared by the entities and can be type, model, country, price, etc.
- facets are sorted by popularity, i.e. by number of occurrences in the dataset in descending order displayed together with the number of occurrences.
- the user can select several facet values, upon which the occurrence counts of the other facet values are updated
- at the same time, the set of matching entities (or a part of it) is displayed as a list

The main benefit of faceted search is that the user can quickly explore the structure of an unfamiliar dataset and see how adding facet values (restrictions) decreases the size of the remaining subset. This avoids the common problem of formulating queries that either return no results, or return too many results to be useful.

The Ontotext KIM showcase "Latest News" [17] presents an interesting variant of the faceted search idea that is very relevant to the BBC dataset (see Figure 8.3): one uses faceting to explore the *co-occurrence* of semantic things in documents. More specifically:

- entities are news articles from several news feeds or crawled from the web. This corresponds to class `cwork:CreativeWork`.
- facets are specific people, organizations, locations mentioned in the document. This corresponds to tagging BBC creative works with real-world things i.e., instances of class `owl:Thing` using property `cwork:tag`.

In addition, Latest News facets also include any other Related Concepts (i.e. free keywords), and allow the user to select the semantic entities to be used as facets (see Figure 8.4). However, we limit the queries in this section to only fixed things (no free keywords). Facet values can be selected directly or using auto-completion; and can be deselected (removed). After facet values are selected, the matching set of documents is shown in a list below the facets. The user can drill down and explore interesting co-occurrences. Eg what is the relation between the Pixar animation company and Hogs (see Figure 8.5)? It turns out that a single document mentions "Toy Story maker Pixar" (a studio that was bought by Disney) and the movie "Wild Hogs 2" whose making was canceled.

Implementing facets in a repository is hard, because we need to calculate or store recursive counts of occurrences, and the queries touch a large fraction of the data.

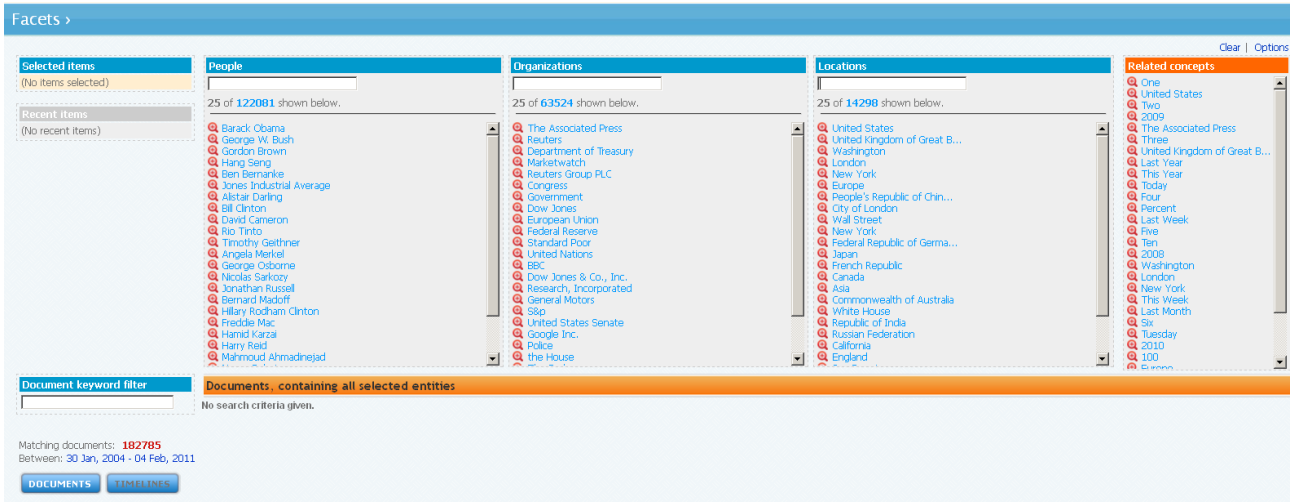


Figure 8.3: Ontotext KIM Showcase: Latest News Faceted Search

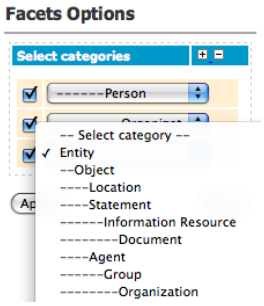


Figure 8.4: Customizing Facet Selections

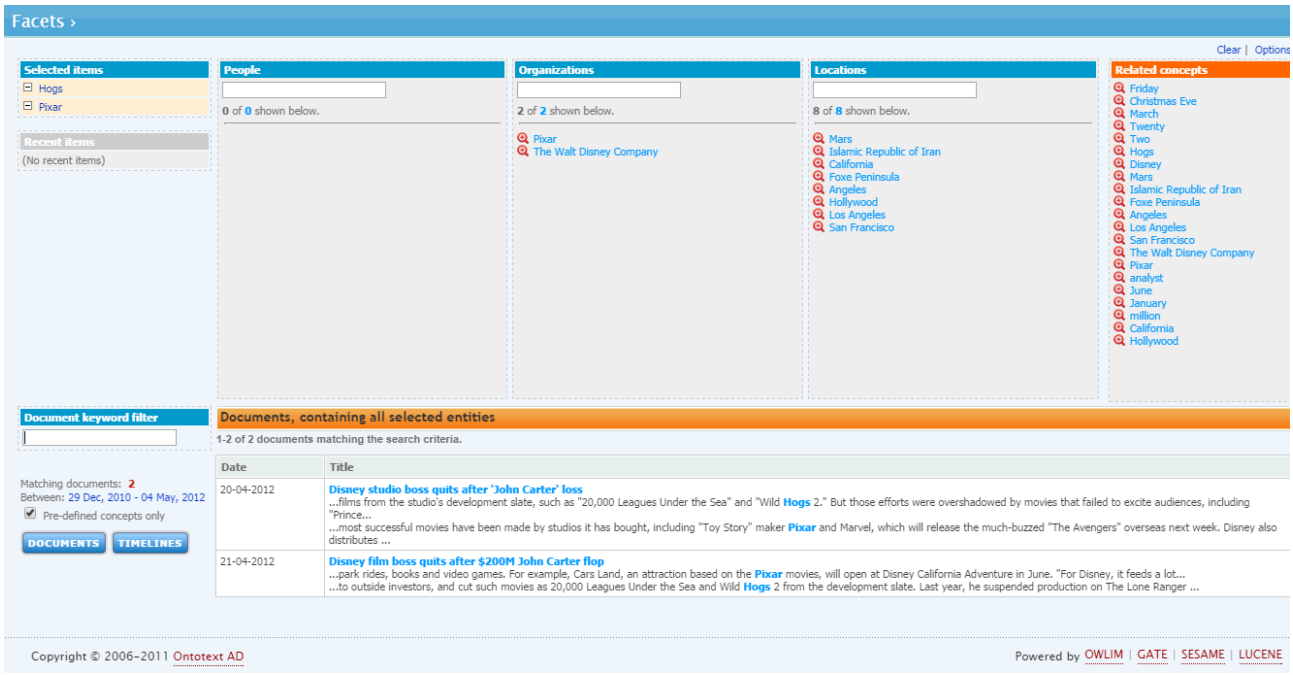


Figure 8.5: Narrowing Using Faceted Search: What's the Connection Between Pixar and Hogs?

Naive Implementation with Counting

We present in this Section a naive implementation approach that counts the occurrences of every facet value, and refreshes the counts when the facet selection changes.
Initially the facet selection is empty (there are no bound facets). Since the number of facet values may be very large (in the order of 10,000), we return only a limited number of facet values per type (for instance, Person, Organization, Place) *limited* to, for instance `10` (`limit 10`). This is reasonable even after some facets are bound, because the number of facet values co-occurring with a popular entity such as "The United Kingdom" may still be very large. The user may request further facets, which can be accomplished by specifying an *offset* (`offset 20`)[15]. Alternatively, the user may add facet values through auto-completion, which can be accomplished with appropriate full-text search. Below we show the SPARQL query that can be used for facet counting.

```
select ?type ?facet ?count where {
   filter (?type in (core:Organisation, core:Person, core:Place))
   {select ?type ?facet (count(*) as ?count)
    where {
      ?facet a ?type.
      ?work cwork:tag ?facet}
    group by ?facet
    order by desc(?count)
    limit 10
   }
}
```

When the user changes the selection to `t1...tn`, we need to re-count by filtering to only works tagged with the selected facets. Let `$TAGS` be the comma-separated concatenation of `t1...tn` represented as URLs, i.e. `<t1>,...,<tn>`. The query we need can be made by substituting `$TAGS` in the following template:

```
select ?type ?facet ?count where {
   filter (?type in (core:Organisation, core:Person, core:Place))
   {select ?type ?facet (count(*) as ?count)
    where {
      ?facet a ?type.
      filter (?facet not in ($TAGS))
      ?work cwork:tag ?facet, $TAGS}
    group by ?facet
    order by desc(?count)
    limit 10
   }
}
```

This re-count needs to happen on both addition and removal of facet values. So the first query is really a special case where the selection `$TAGS` is empty. Such approach is too slow to be practical because the repository needs to count all facet occurrences and sort the counts before picking the top 10. Below we describe an implementation using a custom extension.

Implementation with Lucene Faceting

For a long time the de-facto industry standard for implementing faceted search was Solr, an Apache open source project based on Lucene (since 2010, the two projects are developed together). It uses efficient data structures that are tuned to storing facets, and supports returning a small set of high-popularity facets quickly, facet pagination, etc. Since release 4 (Aug 2012), Lucene also has support for facets. Another widely used open-source product with strong faceting support is ElasticSearch, also based on Lucene.

The recently released Ontotext GraphDB 6 (the new name of the OWLIM repository) includes two features that facilitate faceting:

- **GraphDB Connectors** [20] provide extremely fast keyword search, hit highlighting and faceted search through integration with externally installed search servers (Solr, ElasticSearch, and Lucene are sup-

[15]To have reasonable paging, the `offset` is a multiple of the limit

ported). The search indexes stay automatically up-to-date with the GraphDB repository data. The rest of this section uses the Solr GraphDB connector [22], which provides sorting by facet popularity.

- **Lucene4 Plugin** [21] has provided similar functionality for a year, but is limited to Lucene and is now deprecated.

First we need to setup a Solr connector. This is done with an update query where the parameters are specified in a JSON string. We index only the facet fields (`cwork:tag`); GraphDB notices the values are URLs, so it maps them to an appropriate Solr type that is not passed through text analysis. We also "split" the property into "sub-properties" that are separated by the type of the connected Thing (a very similar example is provided in [22] section "Advanced entity filter example").

```
PREFIX : <http://www.ontotext.com/connectors/solr#>
PREFIX inst: <http://www.ontotext.com/connectors/solr/instance#>

INSERT DATA {
  inst:my_index :createConnector ''' {
    "solrUrl": "http://localhost:8983/solr",
    "types": ["http://www.bbc.co.uk/ontologies/creativework/CreativeWork"],
    "fields": [
       {"fieldName": "tagWithPerson",
         "propertyChain": ["http://www.bbc.co.uk/ontologies/creativework/tag"]},
       {"fieldName": "tagWithOrg",
         "propertyChain": ["http://www.bbc.co.uk/ontologies/creativework/tag"]},
       {"fieldName": "tagWithPlace",
         "propertyChain": ["http://www.bbc.co.uk/ontologies/creativework/tag"]}],
    "entityFilter":
      "?tagWithPerson type in (<http://www.bbc.co.uk/ontologies/coreconcepts/Person>) &&
       ?tagWithOrg type in (<http://www.bbc.co.uk/ontologies/coreconcepts/Organisation>) &&
       ?tagWithPlace  type in (<http://www.bbc.co.uk/ontologies/coreconcepts/Place>)"
    } ''' .
}
```

Now assume that the user has selected these facet values: `http://dbpedia.org/resource/Mozart` for Person and `http://dbpedia.org/resource/Salzburg` for Place.

First, we need to create a relatively complex Solr `$QUERY`. See [26] for a description of the query facet parameters. The query is a concatenation of the following:

- `wt=json`: return the result in JSON rather than the default XML format
- `&facet=true`: enable faceting
- `&facet.field=tagWithPerson&facet.field=tagWithOrg&facet.field=tagWithPlace`: enumerate the faceted Solr fields
- `&facet.mincount=1&facet.limit=10`: return only non-empty facet values, and 10 values per type
- `&facet.sort=count`: sort by descending count (this is the default for non-zero limit)
- `&q=tagWithPerson:http\:\/\/dbpedia.org\/resource\/Mozart`: query by first facet value. The ugly escaping of `:` and `/` is required by Lucene query syntax. This is explained in [21] section "Additional joins"
- `+AND+tagWithPlace:http\:\/\/dbpedia.org\/resource\/Salzburg`: add the second facet value using AND

Then we substitute `$QUERY` in this SPARQL query template.

```
PREFIX : <http://www.ontotext.com/connectors/solr#>
PREFIX inst: <http://www.ontotext.com/connectors/solr/instance#>

SELECT ?type ?facetValue ?facetCount ?facetLabel {
  ?search a inst:my_index ;
    :query "$QUERY";
    :facetFields "tagWithPerson,tagWithOrg,tagWithPlace" ;
    :facets [
     #:facetName  ?type; # instead, we use "a ?type" below
      :facetValue ?facetValue ;
      :facetCount ?facetCount ].
  ?facetValue a ?type; rdfs:label ?facetLabel
}
```

where *i)* `?type` corresponds to one of the `:facetFields`. The `:facetName` property returns it, but we prefer to get it with a normal triple pattern using `a` (i.e. `rdf:type`) *ii)* `?facetValue` is the facet URL, i.e. the Thing (resource) that a creative work is tagged with and *iii)* `?facetLabel` is the label of the facet resource.

8.3.9 GeoSpatial Queries

Geospatial queries are quite important for some Semantic Publishing scenarios. For example, the UK Press Association sells most of its assets through a combined temporal and geospatial search. A lot of its clients purchase journalistic content based on specific time and geographic locality.

Successful geospatial search depends on several factors:

1. Accurate geospatial tagging. The most important concern here is smart place name disambiguaton, since place names are highly ambiguous. The following can be used:
   (a) Various contextual hints (eg super-places mentioned in the text)
   (b) Place type information
   (c) Additional place features, such as population (we may assume that bigger cities occur more often than smaller cities)
2. Accurate geographic coordinates for semantic entities (`core:Place`).
3. Geospatial indexing and querying extensions in the repository.

Here we focus on points 2 and 3 above:

- Coordinates, place types and some additional features are provided by the additional datasets in SPB [15], which include GeoNames info.
- Geospatial extensions to RDF and SPARQL are standardized to a good degree, see [16]. But in this section we consider OWLIM's geospatial extensions (see [18]), which are quite simpler yet sufficient for the use cases (involving point geometries only).

For example, GeoNames contains the following information about Leicester (given in RDF Turtle format):

```
@prefix gn: <http://www.geonames.org/ontology#> .
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .
@prefix wgs: <http://www.w3.org/2003/01/geo/wgs#> .

<http://sws.geonames.org/2644668/> a gn:Feature ;
  rdfs:isDefinedBy <http://sws.geonames.org/2644668/about.rdf> ;
  gn:name "Leicester" ;
  gn:alternateName "Leicester"@en , "Lestera"@lv , "Lesteris"@lt...;
  gn:featureClass gn:P ;
  gn:featureCode gn:P.PPLA2 ;
  gn:countryCode "GB" ;
  gn:population "339239" ;
  wgs:lat "52.6386" ;
  wgs:long "-1.13169" ;
  gn:parentFeature <http://sws.geonames.org/3333165/> ;
  gn:parentCountry <http://sws.geonames.org/2635167/> ;
  gn:parentADM1 <http://sws.geonames.org/6269131/> ;
  gn:parentADM2 <http://sws.geonames.org/3333165/> ;
  gn:nearbyFeatures <http://sws.geonames.org/2644668/nearby.rdf> ;
  gn:locationMap <http://www.geonames.org/2644668/leicester.html> ;
  gn:wikipediaArticle <http://en.wikipedia.org/wiki/Leicester> ;
  rdfs:seeAlso <http://dbpedia.org/resource/Leicester> .
```

This RDF snippet contains the following[16]:

- useful properties: `gn:name`, `gn:alternateName` (name in many languages), `gn:featureClass` ("P" means "places") and `gn:featureCode` ("P.PPLA2" means "seat of a second-order administrative division"), `gn:population`.
- coordinate information by means of WGS84 point (centroid) (properties `wgs:lat` and `wgs:long`) and lists of nearby features through `gn:nearbyFeatures`.

[16]We assume that GeoNames is loaded in the repository, and CreativeWorks are tagged with GeoNames resources, or with resources coreferenced to GeoNames.

- parent features using properties `gn:parentCountry` (as resource) and `gn:countryCode` (as literal), first-level administrative region `gn:parentADM1` (England), second-level administrative region `gn:parentADM2` (City of Leicester region) and immediate parent `gn:parentFeature` (coincides with `gn:parentADM2`)
- useful links through `rdfs:isDefinedBy`, list of nearby places `gn:nearbyFeatures`, interactive location map (`gn:locationMap`) and corresponding Wikipedia page and DBPedia resource (coreferencing) through the `gn:wikipediaArticle` and `rdfs:seeAlso`.

To implement this functionality in OWLIM, a geospatial index must be initially created as follows:

```
prefix omgeo: <http://www.ontotext.com/owlim/geo#>
INSERT DATA {[] omgeo:createIndex []}
```

This puts all resources with fields `wgs:lat`, `wgs:long` (i.e. points or `wgs:SpatialThing`) in a spatial index. OWLIM provides several geospatial functions, of which we use the following ones:

- `?point omgeo:nearby(?lat ?long ?distance)` (predicate): finds points that are close to the given coordinates, i.e. within a given distance. The distance is measured in "km" by default, but a suffix "mi" (miles) can be provided
- `omgeo:distance(?lat1, ?long1, ?lat2, ?long2)` (function): calculates distance between two points, measured in "km"
- `?point omgeo:within(?lat1 ?long1 ?lat2 ?long2)` (predicate): finds points that fall within the given bounding box

While `omgeo:within` is a predicate and takes a `rdf:List` of arguments, `omgeo:distance` is a function and takes a comma-separated list of arguments. One may be tempted to rewrite `omgeo:within` to the query clause shown below:

```
?point wgs:lat ?lat; wgs:long ?long.
filter (?lat1 <= ?lat && ?lat <= ?lat2 && ?long1 <= ?long && ?long <= long2)
```

However, `omgeo:within` has the following advantages compared to a query that compares literals :

- each conjunction of the comparison query may return many results, since the literal index includes all kinds of numbers not only coordinates, and each conjunction limits the numbers only to one side. In contrast, `omgeo:within` uses a geospatial index which provides a fast 2D search structure
- the comparison query may be problematic if the bounding box spans the +-180 degree meridian

In the following sections we are going to consider three use case scenarios using geospatial functionality.

Local News

In this use case, we want to find works tagged with places nearby a given place (within 50 km) e.g., Leicester. We could either use the GeoNames URL of Leicester directly (as shown above), or retrieve it by name and featureCode (as in the query below)[17]:

```
prefix gn: <http://www.geonames.org/ontology#>
prefix wgs: <http://www.w3.org/2003/01/geo/wgs#>
prefix ontogeo: <http://www.ontotext.com/owlim/geo#>
prefix cwork: <http://www.bbc.co.uk/ontologies/creativework/>

select distinct ?work {
  [gn:name "Leicester"; gn:featureCode gn:P.PPLA2; wgs:lat ?lat; wgs:long ?long].
  ?work cwork:tag ?place.
  ?place omgeo:nearby(?lat ?long "50")
}
```

News About Colocated Places

We want to find works tagged with nearby places (within 10 km) of two specified types (eg Castles near Mountains). (Note: this query is inspired by a sample query in [6]). For this query, we need to know the featureCodes of:

[17] Note: don't get confused by appearances: the last line is a triple pattern where the object is an rdf:List.

- Castles: S.CSTL
- Mountains: there is a wider choice: T.MT (mountain), T.RDGE (ridge/s), T.MTS (mountains = a mountain range or a group of mountains or high ridges), T.PK (peak). We could even consider T.NTK (nunatak = a rock or mountain peak protruding through glacial ice; though there are hardly any castles located on ice), T.VLC (volcano), etc

It also helps to know the cardinalities: Castles are 10x fewer than Mountains (about 3.8k vs 380k). The SPARQL query below first finds Castles (the smaller cardinality), then finds `nearby` Mountains.

```
select * {
  ?work cwork:tag ?castle, ?mountain.
  ?castle gn:featureCode S.CSTL;
    wgs:lat ?castle_lat; wgs:long ?castle_long; gn:name ?castle_name.
  ?mountain omgeo:nearby(?castle_lat ?castle_long "10");
    wgs:lat ?mountain_lat; wgs:long ?mountain_long; wgs:alt ?mountain_alt;
    gn:name ?mountain_name;  gn:featureCode ?mountain_feat.
  filter (?mountain_feat in (gn:T.MT, gn:T.RDGE, gn:T.MTS, gn:T.PK))
  bind (omgeo:distance(?castle_lat, ?castle_long, ?mountain_lat, ?mountain_long) as ?dist)
}
```

In contrast to the previous query, we also return the names and coordinates of the matched places and the distance between them (using a geospatial *function*) in addition to the work.

Country Popularity

In this use case, we want to find the number of works tagged with places in each country. Naturally, this may count a work towards several countries. But we don't want to count a work several times if it mentions several places in a country. This is implemented with the SPARQL query below:

```
select ?country_name ?country (count(distinct ?work) as ?count) {
  {?work cwork:tag ?country}
  UNION
  {?work cwork:tag ?place. ?place gn:parentCountry ?country}
  ?country gn:name ?country_name ;
           gn:featureCode gn:A.PCLI.
# independent political entity
} group by ?country ?country_name
```

An interesting modification would be to calculate "popularity density" the ratio `?count/?population` (inverse).

9 CONCLUSIONS

In this deliverable we defined four different sets of queries, testing different ways in which a *forward* or *backward* reasoner can be exploited in a reasoning-aware RDF query engine. These different sets are: *conformance*, *static* tests, *selectivity* and *advanced reasoning* tests.

Note that, unlike other benchmarks proposed in the literature [11, 30, 40, 79], the objective of the benchmarks presented in this work is not to stress the reasoner of the query engine into performing complex forms of reasoning with large amounts of data, i.e., our intention is not to provide standard workload benchmarks; instead, our objective is to see how schema information could be exploited for query answering, i.e., whether it performs sound and complete reasoning, and, second, whether the query engine uses information provided by reasoning (schema) to perform interesting optimizations of increasing complexity, in order to improve the query execution plans, and, consequently, the performance of the engine. Our focus was mainly on the second point, where we showed that there is lots of room for optimizations and heuristics for the query planner, when the schema information is considered, and provided a series of tests that explores whether such optimizations are considered.

Appendices

A REASONING BENCHMARK: SPB TESTS

In this Chapter we present the tests discussed in each of the Chapters 4, 5, 6 and 7 using the SPB schema. For each test we provide (a) the set of triples that should at least exist in the dataset (*preconditions*) needed to run the test. There are cases in which additional triples should be added in the dataset to attain this objective (these triples are shown in bold); (b) the SPARQL query that implements the semantics of the examined construct using the SPB ontologies.

We also present the SPB ontologies as those have been provided by BBC, along with additions that we included to cover the OWL constructs that we study in our work.

A.1 Semantic Publishing Benchmark Ontologies

The Semantic Publishing Benchmark (SPB) uses seven *core* and three *domain* RDF ontologies provided by BBC. The former define the main entities and their properties, required to describe essential concepts of the benchmark namely, *creative works*, *persons*, *documents*, BBC *products* (news, music, sport, education, blogs), *annotations* (*tags*), *provenance* of resources and *content management system* information. The latter are used to express concepts from a *domain of interest* such as football, politics, entertainment among others. The employed ontologies have 74 classes, 88 and 28 *data type* and *object* properties respectively. They contain 60 `rdfs:subClassOf`, 17 `rdfs:subPropertyOf`, 105 `rdfs:domain` and 115 `rdfs:range` RDFS [12] properties. On the other hand the ontologies contain a limited number of OWL [43] constructs: they contain 8 `owl:oneOf` class axioms that allow one to define a class by enumeration of its instances and one `owl:TransitiveProperty` property. The ontologies consider few classes, properties and shallow class and property hierarchies. More specifically, the class hierarchy has a maximum depth of *3* whereas the property hierarchy has a depth of *1*. A detailed presentation of the ontologies employed by SPB can be found in [29].

In this section we discuss briefly a fragment of the *Creative Works* core ontology shown in Figure A.1. Ontologies are represented as *node* and *edge labeled directed* graphs where *classes* and *their instances* are depicted by an *oval*, and *properties* as *edges* between nodes, where the name of the property is the *label* of the edge.

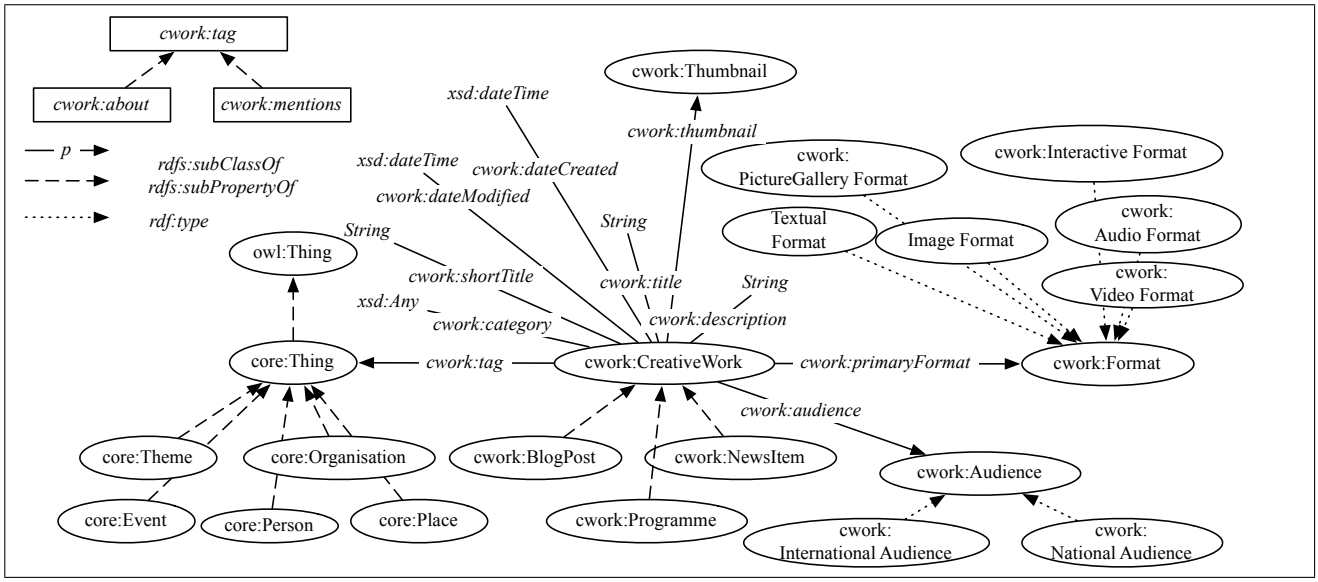


Figure A.1: BBC Creative Works Ontology

The main class is `cwork:CreativeWork` (shown in Figure A.1) that collects all RDF descriptions of creative works (also called *journalistic assets*) created by the publisher's editorial team. This class is defined

as a subclass of core:Thing (subclass of owl:Thing), allowing in this way the creation of complex information graphs. A creative work has a number of properties such as cwork:title, cwork:shortTitle, cwork:description, cwork:dateModified, cwork:dateCreated, cwork:audience, cwork:format and cwork:thumbnail among others; it has a category (property cwork:category) and can be tagged (property cwork:tag) with *anything* (i.e., instances of class owl:Thing). The latter property is further specialized (through the rdfs:subPropertyOf relation) to properties cwork:about and cwork:mentions that are heavily used during the creation of creative works. Creative works can be instances of classes cwork:NewsItem, cwork:Programme and cwork:BlogPost, all defined as subclasses of class cwork:CreativeWork. The BBC ontologies also use classes such as core:Place, core:Event, core:Organisation, core:Person, and core:Theme, all defined as subclasses of class core:Thing.

We extended the provided BBC ontologies with additional constructs such as property and class constraints shown in Figures A.2 and A.3.

In order to incorporate more OWL constructs, we extended the BBC ontologies as follows with concepts from DBPedia[1] and FOAF[2] ontologies. More specifically, we used the FOAF class foaf:Person, and the DBPedia classes dbpedia:Place, dbpedia:Event, dbpedia:Organisation, dbpedia:Sport all defined as *equivalent* to the classes with the same name in the BBC ontologies using the owl:equivalentClass property; all classes were defined as *subclasses* of core:Thing. We do not include all their properties as those are defined in the ontologies, but focus only on the ones that are useful in the context of the semantic publishing benchmark. Moreover, we used a set of properties from those classes and declared them as *equivalent* to properties with the same label defined in their equivalent DBPedia and FOAF classes; equivalence was defined through the owl:equivalentProperty property. We have also included classes from the Travel Ontology that defines travel-related entities[3], all defined as *subclasses* of BBC class core:Thing.

As mentioned above, we did not include all classes of the aforementioned ontologies but a subset thereof; in addition, we included only a subset of their properties. More specifically, for dbpedia:Event, we focused on properties rdfs:label, rdfs:comment, dbpedia–owl:country and dcterms:subject; for class dbpedia:Organisation we included data properties rdfs:label and rdfs:comment as well as the object properties dbpprop:manager, dbpprop:name, dbpprop:nickname and dbpprop:website. For class dbpedia:Sport we keep data properties rdfs:comment and dbpprop:caption, and object properties dbpprop:olympic, dbpprop:team and dbpprop:equipment. Last in the case of class dbpedia:Place we used data properties foaf:name, rdfs:comment and object properties dbpedia–owl:country and geo:geometry.

Regarding the FOAF ontology we focused our attention on the foaf:Person class; we considered its data type properties foaf:name, foaf:surname, foaf:givenName, dc:description, dbpedia–owl:birthDate, dbpedia–owl:deathDate and object properties dbpedia–owl:birthPlace and dbpedia–owl:deathPlace.

From the Travel ontology we included classes travel:AdministrativeDivision, travel:bodyOfLand, travel:City, travel:TierOneAdministrativeDivision, travel:Coastline, travel:Continent, travel:Country, travel:Island, travel:EuropeanIsland, travel:River to create a class hierarchy of length 3 with its root being class owl:Thing. Finally, we also considered classes travel:Recognised defined as a subclass of owl:Thing.

The enhanced SPB schema contains 31 classes, 38 and 98 *data type* and *object* properties respectively; it contains 83 rdfs:subClassOf, 19 rdfs:subPropertyOf, 134 rdfs:domain and 145 rdfs:range, 18 owl:equivalentProperty, 8 owl:equivalentClass, 3 owl:FunctionalProperty, 8 owl:disjointWith, 1 owl:AsymmetricProperty, 1 owl:IrreflexiveProperty, 1 owl:propertyChainAxiom, 1 owl:InverseFunctionalProperty, , 4 owl:intersectionOf, 1 owl:unionOf properties.

Figures A.4 and A.5 show a part of the triples considered from the DBPedia, FOAF and Travel ontologies and their relationship with the core BBC ontologies.

[1] DBPedia: dbpedia.org

[2] The Friend of a Friend (FOAF) project: http://www.foaf-project.org/

[3] Travel Ontology: http://swatproject.org/travelOntology.asp

```
ldbc:partOf
    rdf:type rdf:Property ;
    rdfs:domain bbc:WebDocument ;
    rdfs:range  bbc:WebDocument ;
    rdf:type owl:AsymmetricProperty ;
    rdf:type owl:TransitiveProperty ;
    rdf:type owl:IrreflexiveProperty .

ldbc:cworkThumbnailAltText
    rdf:type rdf:Property ;
    rdfs:domain cwork:CreativeWork ;
    rdfs:range xsd:string ;
    owl:propertyChainAxiom ( cwork:thumbnail cwork:altText ) .

news:Theme
    owl:unionOf ( ldbc:Sport ldbc:Politics ldbc:Music ldbc:Art ) .

news:Event
    owl:intersectionOf ( news:Person news:Organisation ) .

bbc:Platform
    owl:oneOf ( bbc:HighWeb bbc:Mobile ) .

bbc:primaryContentOf
    rdf:type owl:ObjectProperty ;
    rdfs:comment "Inverse of bbc:primaryContent"^^xsd:string ;
    rdfs:domain cwork:CreativeWork ;
    rdfs:isDefinedBy bbc: ;
    rdfs:range bbc:WebDocument ;
    owl:inverseOf bbc:primaryContent .

bbc:primaryContent
    rdf:type owl:FunctionalProperty .

core:disambiguationHint
        rdf:type owl:InverseFunctionalProperty ;
        rdf:type owl:FunctionalProperty .

core:facebook
    owl:propertyDisjointWith core:twitter .

core:twitter
    owl:propertyDisjointWith core:facebook .

_:node1 a owl:AllDisjointProperties ;
    owl:members ( cwork:tag cwork:audience cwork:primaryFormat
    cwork:thumbnail ) .

_:node2 a owl:AllDisjointClasses ;
    owl:members ( cwork:NewsItem cwork:BlogPost cwork:Programme ) .

ldbc:dateDestroyed
    rdf:type owl:DatatypeProperty ;
        rdfs:domain cwork:CreativeWork ;
        rdfs:range xsd:string .

ldbc:Event_Place_Theme
    owl:intersectionOf ( dbpedia-owl:Event dbpedia-owl:Place core:Theme ).
```

Figure A.2: Enhancements to the SPB ontologies with class and property constraints (a)

```
cwork:CreativeWork rdfs:subClassOf [
    rdf:type owl:Restriction ;
    owl:onProperty ldbc:dateDestroyed ;
    owl:maxCardinality "0"^^xsd:NonNegativeInteger
] .

cwork:CreativeWork rdfs:subClassOf [
    rdf:type owl:Restriction ;
    owl:onProperty cwork:thumbnail ;
    owl:cardinality "1"^^xsd:NonNegativeInteger
] .

cwork:NewsItem
     owl:disjointWith cwork:Programme, cwork:BlogPost .

cwork:CreativeWork rdfs:subClassOf [
    rdf:type owl:Restriction ;
    owl:onProperty cwork:audience ;
    owl:minCardinality "0"^^xsd:NonNegativeInteger ;
    owl:maxCardinality "2"^^xsd:NonNegativeInteger ;
] .

core:Thing
    owl:hasKey ( core:shortLabel core:preferredLabel
              core:disambiguationHint core:primaryTopicOf ) .

cwork:dateModified
    rdf:type owl:FunctionalProperty .
```

Figure A.3: Enhancements to the SPB ontologies with class and property constraints (b)

```
dbpedia-owl:Event
    rdf:type owl:Class ;
    owl:equivalentClass core:Event ;
    owl:disjointWith dbpedia-owl:Person .

dbpedia-owl:chairperson
     rdf:type owl:ObjectProperty ;
     rdfs:range dbpedia-owl:Person .

dbpedia-owl:hometown
    rdf:type owl:ObjectProperty;
    rdfs:domain dbpedia-owl:Person ;
    rdfs:label "hometown"^^xsd:string ;
    rdfs:range dbpedia-owl:Settlement .

dbpedia-owl:birthDate
    owl:equivalentProperty core:birthDate .
```

Figure A.4: Dbpedia schema triples used in SPB tests

```
travel:SaltLake
    rdfs:subClassOf travel:BodyOfSeaWater .
travel:BodyOfSeaWater
    rdfs:subClassOf travel:BodyOfWater .
travel:BodyOfWater
    rdfs:subClassOf travel:GeographicalFeature .
travel:GeographicalFeature
    rdfs:subClassOf travel:Natural_physical_thing .
travel:Natural_physical_thing
    rdfs:subClassOf travel:Natural_entity .
travel:Natural_entity
   rdfs:subClassOf travel:Physical_entity .

travel:nextTo
    rdf:type owl:SymmetricProperty, owl:ObjectProperty .

travel:hasMemberIsland
    owl:inverseOf travel:isMemberIslandOf .
```

Figure A.5: Travel schema triples used in SPB tests

A.2 Conformance Tests

A.2.1 Class and Property Subsumption

Class Subsumption (CAX-SCO)

Preconditions

```
cwork:BlogPost
     rdfs:subClassOf cwork:CreativeWork .

things:cw-cax-sco-1#id
     rdf:type cwork:BlogPost .
```

SPARQL Query

```
ASK { things:cw-cax-sco-1#id rdf:type cwork:CreativeWork }
```

Property Subsumption (PRP-SPO1)

Preconditions

```
cwork:about
     rdf:type owl:ObjectProperty ;
     rdfs:subPropertyOf cwork:tag .

things:pr-scm-spo-1#id
     rdf:type rdf:Property ;
     rdfs:subPropertyOf cwork:about .

things:cw-cax-sco-1#id
     things:pr-scm-spo-1#id tags:tag-cax-sco-spo-1#id .
```

SPARQL Query

```
ASK { things:cw-cax-sco-1#id cwork:about tags:tag-cax-sco-spo-1#id .
 FILTER NOT EXISTS{ things:cw-cax-sco-1#id cwork:tag tags:tag-cax-sco-spo-1#id}}
```

Class Subsumption (SCM-SCO)

Preconditions

```
cwork:BlogPost
     rdf:type owl:Class ;
     rdfs:subClassOf cwork:CreativeWork .

cwork:CreativeWork
     rdf:type owl:Class ;
     rdfs:subClassOf owl:Thing .
```

SPARQL Query

```
ASK { cwork:BlogPost rdfs:subClassOf owl:Thing }
```

Property Subsumption (SCM-SPO)

Preconditions

```
things:pr-scm-spo-1#id
     rdf:type rdf:Property ;
     rdfs:subPropertyOf cwork:about .

cwork:about
     rdf:type owl:ObjectProperty ;
     rdfs:subPropertyOf cwork:tag .
```

SPARQL Query

```
ASK { things:pr-scm-spo-1#id rdfs:subPropertyOf cwork:tag }
```

A.2.2 Property Domain and Range

Property Range (SCM-RNG1)

Preconditions

```
news:person
     rdf:type owl:ObjectProperty ;
     rdfs:range news:Person .

news:Person
     rdf:type owl:Class ;
     rdfs:subClassOf core:Person .

core:Person
     rdf:type owl:Class ;
     rdfs:subClassOf core:Thing .
```

SPARQL Query

```
ASK { news:person rdfs:range core:Person .
      FILTER NOT EXISTS { news:person rdfs:range core:Thing } }
```

Property Range (SCM-RNG2)

Preconditions

```
cwork:tag
     rdf:type owl:ObjectProperty ;
     rdfs:range core:Thing .

things:dr-scm-rng2-1#id
     rdf:type rdf:Property ;
     rdfs:subPropertyOf cwork:tag .
```

SPARQL Query

```
ASK { things:dr-scm-rng2-1#id rdfs:range core:Thing }
```

Property Domain (SCM-DOM1)

Preconditions

```
news:person
     rdf:type owl:ObjectProperty ;
     rdfs:domain news:Event .

news:Event
     rdf:type owl:Class ;
     rdfs:subClassOf core:Event .
```

SPARQL Query

```
ASK { news:person rdfs:domain core:Event }
```

Property Domain (SCM-DOM2)

Preconditions

```
cwork:about
     rdf:type owl:ObjectProperty ;
     rdfs:domain cwork:CreativeWork .

events:cw-prp-scm-dom2-1#id
     rdf:type rdf:Property ;
     rdfs:subPropertyOf cwork:about .
```

SPARQL Query

```
ASK { events:cw-prp-scm-dom2-1#id rdfs:domain cwork:CreativeWork }
```

Property Domain (PRP-DOM)

Preconditions

```
events:event-prp-dom-rng-1#id
     news:person org:org-prp-dom-rng-1#id .

news:person
     rdf:type owl:ObjectProperty ;
     rdfs:domain news:Event .
```

SPARQL Query

```
ASK { events:event-prp-dom-rng-1#id rdf:type news:Event .
      events:event-prp-dom-rng-1#id news:person ?org }
```

Property Range (PRP-RNG)

Preconditions

```
news:person
     rdf:type owl:ObjectProperty ;
     rdfs:range news:Person .

news:Person
     rdfs:subClassOf core:Person .

core:Person
     rdfs:subClassOf core:Thing .

core:Thing
     rdfs:subClassOf owl:Thing .

events:event-prp-dom-rng-1#id
     news:person org:org-prp-dom-rng-1#id .
```

SPARQL Query

```
ASK { org:org-prp-dom-rng-1#id rdf:type owl:Thing .
      ?event news:person org:org-prp-dom-rng-1#id }
```

A.2.3 Union and Intersection of Classes

Union of Classes (SCM-UNI)

Preconditions

```
news:Theme
   owl:unionOf ( ldbc:Sport ldbc:Politics ldbc:Music ldbc:Art ) .
```

SPARQL Query

```
ASK { ldbc:Sport rdfs:subClassOf news:Theme .
      ldbc:Politics rdfs:subClassOf news:Theme .
      ldbc:Music rdfs:subClassOf news:Theme .
      ldbc:Art rdfs:subClassOf news:Theme }
```

Union of Classes (SCM-UNI, CAX-SCO)

Preconditions

```
news:Theme
   owl:unionOf ( ldbc:Sport ldbc:Politics ldbc:Music ldbc:Art ) .

things:sport-scm-uni-1#id
     rdf:type ldbc:Sport .
```

SPARQL Query

```
ASK { things:sport-scm-uni-1#id rdf:type news:Theme}
```

Intersection of Classes (SCM-INT)

Preconditions

```
news:Event
   owl:intersectionOf ( news:Person news:Organisation ) .

things:sport-scm-int-1#id
     rdf:type news:Event .
```

SPARQL Query

```
SELECT ?x
WHERE { news:Event rdfs:subClassOf ?x }
```

Intersection of Classes (SCM-INT, CAX-SCO)

Preconditions

```
things:sport-scm-int-1#id
     rdf:type news:Event .

news:Event
   owl:intersectionOf ( news:Person news:Organisation ) .
```

SPARQL Query

```
ASK { things:sport-scm-int-1#id rdf:type news:Person .
      things:sport-scm-int-1#id rdf:type news:Organisation }
```

A.2.4 Enumeration of Individuals

Enumeration of Individuals (CLS-OO)

Preconditions

```
bbc:Platform
     owl:oneOf ( bbc:HighWeb bbc:Mobile ) ;
```

SPARQL Query

```
ASK { bbc:HighWeb rdf:type bbc:Platform .
      bbc:Mobile rdf:type bbc:Platform }
```

A.2.5 Equality

Equality (EQ-REF)

Preconditions

```
things:cw-eq-ref-1#id
     bbc:primaryContentOf things:webdoc-eq-ref-1#id .
```

SPARQL Query

```
ASK { things:cw-eq-ref-1#id owl:sameAs things:cw-eq-ref-1#id .
      bbc:primaryContentOf owl:sameAs bbc:primaryContentOf .
      ?o owl:sameAs ?o .
      things:cw-eq-ref-1#id bbc:primaryContentOf ?o }
```

Equality (EQ-SYM)

Preconditions

```
things:cw-eq-sym-1#id
     bbc:primaryContentOf things:webdoc-eq-sym-1#id ;
     owl:sameAs things:cw-eq-sym-2#id .
```

SPARQL Query

```
ASK { things:cw-eq-sym-2#id owl:sameAs things:cw-eq-sym-1#id }
```

Equality (EQ-TRANS)

Preconditions

```
things:cw-eq-trans-1#id
     owl:sameAs things:cw-eq-trans-2#id .

things:cw-eq-trans-2#id
     owl:sameAs things:cw-eq-trans-3#id .
```

SPARQL Query

```
SELECT ?z
WHERE { things:cw-eq-trans-1#id owl:sameAs ?z }
```

Equality (EQ-REP-S)

Preconditions

```
things:cw-eq-trans-1#id
     owl:sameAs things:cw-eq-trans-2#id .

things:cw-eq-trans-1#id
     bbc:primaryContentOf things:webdoc-eq-trans-1#id .
```

SPARQL Query

```
ASK { things:cw-eq-trans-2#id bbc:primaryContentOf things:webdoc-eq-trans-1#id }
```

Equality (EQ-REP-P)

Preconditions

```
ldbc:referTo
     owl:sameAs ldbc:refersTo .

things:cw-eq-trans-2#id
     ldbc:referTo things:bbc-product-eq-trans-1#id .
```

SPARQL Query

```
ASK { things:cw-eq-trans-2#id ldbc:refersTo things:bbc-product-eq-trans-1#id }
```

Equality (EQ-REP-O)

Preconditions

```
things:webdoc-eq-trans-1#id
     owl:sameAs things:webdoc-eq-trans-2#id .

things:cw-eq-trans-1#id
     bbc:primaryContentOf things:webdoc-eq-trans-1#id .
```

SPARQL Query

```
ASK { things:cw-eq-trans-1#id bbc:primaryContentOf things:webdoc-eq-trans-2#id }
```

A.2.6 Inverse of Properties

Inverse of Properties (PRP-INV1)

Preconditions

```
bbc:primaryContent
     owl:inverseOf bbc:primaryContentOf .

things:cw-prp-inv1-webdocument-1
     bbc:primaryContent things:cw-prp-inv1#id .
```

SPARQL Query

```
ASK {things:cw-prp-inv1#id bbc:primaryContentOf things:cw-prp-inv1-webdocument-1}
```

Inverse of Properties (PRP-INV2)

Preconditions

```
bbc:primaryContent
     owl:inverseOf bbc:primaryContentOf .

things:cw-prp-inv1#id
     bbc:primaryContentOf things:cw-prp-inv1-webdocument-1 .
```

SPARQL Query

```
ASK { things:cw-prp-inv1-webdocument-1 bbc:primaryContent things:cw-prp-inv1#id }
```

A.2.7 Constraints on Properties

Constraints on Properties (PRP-FP)

Preconditions

```
core:disambiguationHint
 rdf:type owl:InverseFunctionalProperty, owl:FunctionalProperty .

things:thing-prp-ifp-1#id
     core:disambiguationHint "hint for things:thing-prp-ifp-1-2#id" .

things:thing-prp-ifp-2#id
     core:disambiguationHint "hint for things:thing-prp-ifp-1-2#id".
```

SPARQL Query

```
ASK { "hint for things:thing-prp-ifp-1-2#id"
           owl:sameAs "hint for things:thing-prp-ifp-1-2#id" }
```

Constraints on Properties (PRP-IFP)

Preconditions

```
core:disambiguationHint
 rdf:type owl:InverseFunctionalProperty, owl:FunctionalProperty .

things:thing-prp-ifp-1#id
     core:disambiguationHint "hint for things:thing-prp-ifp-1-2#id" .

things:thing-prp-ifp-2#id
     core:disambiguationHint "hint for things:thing-prp-ifp-1-2#id".
```

SPARQL Query

```
ASK { things:thing-prp-ifp-1#id owl:sameAs things:thing-prp-ifp-2#id }
```

Constraints on Properties (PRP-ASYP)

Preconditions

```
ldbc:partOf
     rdf:type rdf:Property, owl:AsymmetricProperty .
```

SPARQL Query

```
INSERT DATA {
    things:cw-prp-asyp-constr-1#id
        bbc:primaryContentOf things:prp-asyp-webdocument-1#id .

    things:cw-prp-asyp-constr-2#id
        bbc:primaryContentOf things:prp-asyp-webdocument-2#id .

    things:prp-asyp-webdocument-1#id
        ldbc:partOf things:prp-asyp-webdocument-2#id .

    things:prp-asyp-webdocument#2
        ldbc:partOf things:prp-asyp-webdocument-1#id }
```

Constraints on Properties (PRP-IRP)

Preconditions

```
ldbc:partOf
     rdf:type rdf:Property, owl:IrreflexiveProperty .
```

SPARQL Query

```
INSERT DATA {
    things:cw-prp-irp-constr#id
        bbc:primaryContentOf things:cw-prp-irp-webdocument-1 .

    things:cw-prp-irp-webdocument-1
        ldbc:partOf things:cw-prp-irp-webdocument-1 }
```

Constraints on Properties (PRP-TRP)

Preconditions

```
sport:subDisciplineOf
     rdf:type owl:TransitiveProperty, owl:ObjectProperty ;

sports:sportsdiscipline-prp-trp-1#id
     sport:subDiscipline sports:sportsdiscipline-prp-trp-2#id .

sports:sportsdiscipline-prp-trp-2#id
     sport:subDiscipline sports:sportsdiscipline-prp-trp-3#id .
```

SPARQL Query

```
ASK { sports:sportsdiscipline-prp-trp-1#id
                sport:subDiscipline sports:sportsdiscipline-prp-trp-3#id }
```

A.2.8 Class Keys

Class Keys (PRP-KEY)

Preconditions

```
core:Thing
     owl:hasKey (core:shortLabel core:preferredLabel
                 core:disambiguationHint core:primaryTopicOf) .

cwork:BlogPost
     rdf:type owl:Class ;
     rdfs:subClassOf cwork:CreativeWork .

cwork:CreativeWork
     rdf:type owl:Class ;
     rdfs:subClassOf owl:Thing .

things:cw-prp-key-1-constr#id
     rdf:type cwork:BlogPost ;
     core:shortLabel "label1" ;
     core:primaryTopicOf things:cw-prp-key-webdocument-1 .

things:cw-prp-key-2-constr#id
     rdf:type cwork:CreativeWork;
     core:shortLabel "label1" ;
     core:primaryTopicOf things:cw-prp-key-webdocument-1 .
```

SPARQL Query

```
ASK { things:cw-prp-key-1-constr#id owl:sameAs things:cw-prp-key-2-constr#id }
```

A.2.9 Property Chains

Property Chains (PRP-SPO2)

Preconditions

```
ldbc:cworkThumbnailAltText
     rdf:type rdf:Property ;
     owl:propertyChainAxiom ( cwork:thumbnail cwork:altText ) .

things:cw-prp-spo2#id
     rdf:type cwork:CreativeWork ;
     cwork:thumbnail thumbnail:cw-prp-spo2-thumbnail .

thumbnail:cw-prp-spo2-thumbnail
     cwork:altText "AltText for CW: things:cw-prp-spo2#id" .
```

SPARQL Query

```
ASK {
things:cw-prp-spo2#id
    ldbc:cworkThumbnailAltText "AltText for CW: things:cw-prp-spo2#id" }
```

A.2.10 Disjoint Classes and Properties

Disjoint Classes and Properties (PRP-PDW)

Preconditions

```
core:facebook
     owl:propertyDisjointWith core:twitter .

core:twitter
     owl:propertyDisjointWith core:facebook .
```

SPARQL Query

```
INSERT DATA {
    things:cw-prp-pdw-constr#id
        cwork:title
            "Constraint Violation test for owl:propertyDisjointWith" ;
        rdf:type cwork:CreativeWork ;
        core:facebook things:cw-prp-pdw-webdocument-1 ;
        core:twitter things:cw-prp-pdw-webdocument-1 }
```

Disjoint Classes and Properties (PRP-ADP)

Preconditions

```
_:node1
    rdf:type owl:AllDisjointProperties ;
    owl:members ( cwork:tag cwork:audience
                  cwork:primaryFormat cwork:thumbnail ) .
```

SPARQL Query

```
INSERT DATA {
    things:cw-prp-adp-constr#id a cwork:NewsItem ;
        cwork:title
            "Constraint Violation test for owl:AllDisjointProperties" ;
        cwork:about things:value-1#id;
        cwork:primaryFormat things:value-1#id;
        cwork:audience things:value-1#id;
        cwork:thumbnail things:value-1#id }
```

Disjoint Classes and Properties (CAX-DW)

Preconditions

```
cwork:NewsItem owl:disjointWith cwork:Programme .
```

SPARQL Query

```
INSERT DATA {
    things:cw-cax-dw-const#id
         rdf:type cwork:NewsItem, cwork:Programme . }
```

Disjoint Classes and Properties (CAX-ADC)

Preconditions

```
_:node2
    rdf:type owl:AllDisjointClasses ;
    owl:members ( cwork:NewsItem cwork:BlogPost cwork:Programme ) .
```

SPARQL Query

```
INSERT DATA {
    things:cw-cax-adc-constr#id
        rdf:type cwork:NewsItem, cwork:BlogPost, cwork:Programme .}
```

A.2.11 Cardinalities

Cardinalities (CLS-MAXC1)

Preconditions

```
cwork:CreativeWork rdfs:subClassOf [
    rdf:type owl:Restriction ;
    owl:onProperty ldbc:dateDestroyed ;
    owl:maxCardinality "0"^^xsd:NonNegativeInteger
] .
```

SPARQL Query

```
INSERT DATA {
    things:cw-cls-maxc1-constr#id
        rdf:type cwork:CreativeWork ;
        ldbc:dateDestroyed "1.1.1990" . }
```

Cardinalities (CLS-MAXC2)

Preconditions

```
cwork:CreativeWork rdfs:subClassOf [
    rdf:type owl:Restriction ;
    owl:onProperty cwork:thumbnail ;
    owl:cardinality "1"^^xsd:NonNegativeInteger
] .
```

SPARQL Query

```
INSERT DATA {
    things:cw-cls-maxc2-constr#cwork-id1
        rdf:type cwork:CreativeWork ;
        cwork:thumbnail things:cw-cls-maxc2-constr#thumbnail-id1 ;
        cwork:thumbnail things:cw-cls-maxc2-constr#thumbnail-id2 .}
```

A.3 Static Tests

A.3.1 Equality of Classes (owl:equivalentClass)

Preconditions

```
dbpedia-owl:Event owl:equivalentClass core:Event .
```

SPARQL Query

```
SELECT ?plc
WHERE {
    ?plc rdf:type dbpedia-owl:Place .
    ?plc foaf:name ?name .
    ?plc rdfs:comment ?comm .
    ?plc dbpedia-owl:country ?cntr .
    ?plc geo:geometry ?geo .
    FILTER NOT EXISTS { ?plc rdf:type core:Event } }
```

A.3.2 Disjointness of Classes (owl:disjointWith)

Preconditions

```
cwork:NewsItem
     owl:disjointWith cwork:BlogPost .
```

SPARQL Query

```
SELECT ?datec ?datem ?descr ?title
WHERE {
    ?ent rdf:type cwork:NewsItem .
    ?ent rdf:type cwork:BlogPost .
    ?ent cwork:dateCreated ?datec .
    ?ent cwork:dateModified ?datem .
    ?ent cwork:description ?descr .
    ?ent cwork:title ?title }
```

A.3.3 Equality of Properties (owl:equivalentProperty)

Preconditions

```
dbpedia-owl:birthDate owl:equivalentProperty core:birthDate .
core:birthDate rdf:type owl:FunctionalProperty .
```

SPARQL Query

```
SELECT ?mart
WHERE {
    ?mart rdf:type dbpedia-owl:musicalArtist .
    ?mart dbpprop:name ?name .
    ?mart dbpedia-owl:birthDate ?bdate1 .
    ?mart core:birthDate ?bdate2
    ?mart dbpedia-owl:birthPlace ?place .
    ?mart dbpedia-owl:genre ?genre .
    ?mart dbpedia-owl:hometown ?htown .
    ?bdate1 owl:differentFrom ?bdate2 }
```

A.3.4 Range of Properties (rdfs:range, owl:disjointWith)

Preconditions

```
dbpedia-owl:Event
     rdf:type owl:Class;
     owl:disjointWith dbpedia-owl:Person;

dbpedia-owl:chairperson
     rdf:type owl:ObjectProperty ;
     rdfs:range dbpedia-owl:Person .
```

SPARQL Query

```
SELECT ?ev
WHERE {
    ?ev rdf:type dbpedia-owl:Event .
    ?u dbpedia-owl:chairperson ?ev .
    ?u dbpprop:country ?cntr .
    ?u dcterms:subject ?sub .
    ?u dbpedia-owl:result ?res .
    ?u dbpedia-owl:place ?plc }
```

A.3.5 Domain of Properties (rdfs:domain, owl:disjointWith)

Preconditions

```
dbpedia-owl:Event
     rdf:type owl:Class ;
     owl:disjointWith dbpedia-owl:Person .

dbpedia-owl:hometown
     rdf:type owl:ObjectProperty ;
     rdfs:domain dbpedia-owl:Person .
```

SPARQL Query

```
SELECT ?ev
WHERE {
    ?ev rdf:type dbpedia-owl:Event .
    ?ev dbpedia-owl:hometown ?htown .
    ?ev dcterms:subject ?sub .
    ?ev dbpprop:country ?cntry .
    ?ev rdfs:label ?label .
    ?ev rdfs:comment ?comm }
```

A.3.6 Uniqueness of Property Values (owl:FunctionalProperty)

Preconditions

```
bbc:primaryContent
     rdf:type owl:FunctionalProperty .

things:contr-1#id
     rdf:type cwork:CreativeWork .

things:contr-2#id
     rdf:type cwork:CreativeWork ;
     owl:differentFrom things:contr-1#id .
```

SPARQL Query

```
SELECT ?webdoc
WHERE {
    ?webdoc rdf:type bbc:WebDocument .
    ?webdoc cwork:dateCreated ?dateC .
    ?webdoc cwork:dateModified ?dateM .
    ?webdoc cwork:description ?descr .
    ?webdoc bbc:primaryContent things:contr-1#id .
    ?webdoc bbc:primaryContent things:contr-2#id }
```

A.4 Selectivity Tests

A.4.1 Cardinality

Preconditions

```
cwork:dateModified
     rdf:type owl:FunctionalProperty .

cwork:CreativeWork rdfs:subClassOf [
    rdf:type owl:Restriction ;
    owl:onProperty cwork:thumbnail ;
    owl:cardinality "1"^^xsd:NonNegativeInteger
] .

cwork:CreativeWork rdfs:subClassOf [
    rdf:type owl:Restriction ;
    owl:onProperty cwork:audience ;
    owl:minCardinality "0"^^xsd:NonNegativeInteger ;
    owl:maxCardinality "2"^^xsd:NonNegativeInteger ;
] .
```

SPARQL Query

```
SELECT ?cw
WHERE {
    ?cw rdf:type cwork:CreativeWork .
    ?cw cwork:mentions ?ment .
    ?cw cwork:tag ?tag .
    ?cw cwork:thumbnail ?thumb .
    ?cw cwork:audience ?aud .
    ?cw cwork:dateModified ?dt }
```

A.4.2 Intersection of Classes (`owl:intersectionOf`)

Preconditions

```
news:Event
     owl:intersectionOf ( news:Person news:Organisation ) .
```

SPARQL Query

```
SELECT ?ev
WHERE {
    ?ev rdf:type news:Event .
    ?ev news:person ?per .
    ?per rdf:type news:Person .
    ?ev news:organisation ?org .
    ?org rdf:type news:Organisation }
```

A.4.3 Union of Classes (owl:unionOf)

Preconditions

```
news:Theme
     owl:unionOf ( ldbc:Sport ldbc:Politics ldbc:Music ldbc:Art ) .
```

SPARQL Query

```
SELECT ?theme
WHERE {
    ?theme rdf:type news:Theme .
    ?theme news:notablyAssociatedWith ?sport .
    ?sport rdf:type ldbc:Sport .
    ?sport news:notablyAssociatedWith ?mus .
    ?mus rdf:type ldbc:Music }
```

A.4.4 Hierarchy of Classes (rdfs:subClassOf)

Preconditions

```
news:Event
     rdfs:subClassOf core:Event .

core:Event
     rdfs:subClassOf core:Thing .
```

SPARQL Query

```
SELECT ?time ?place ?theme
WHERE {
    ?ev rdf:type news:Event .
    ?ev rdf:type core:Thing .
    ?ev news:time ?time .
    ?ev news:place ?place .
    ?ev news:theme ?theme }
```

A.4.5 Hierarchy of Properties (rdfs:subPropertyOf)

Preconditions

```
cwork:mentions
     rdfs:subPropertyOf cwork:tag .
```

SPARQL Query

```
SELECT ?ment
WHERE {
    ?cw rdf:type cwork:CreativeWork .
    ?cw cwork:mentions ?ment.
    ?cw cwork:tag ?ment }
```

A.5 Advanced Tests

A.5.1 Optimized Inference (rdfs:subClassOf owl:allValuesFrom)

Preconditions

```
@prefix travel: <http://www.co-ode.org/roberts/travel.owl#> .

travel:SaltLake
     rdfs:subClassOf travel:BodyOfSeaWater .

travel:BodyOfSeaWater
     rdfs:subClassOf travel:BodyOfWater .

travel:BodyOfWater
     rdfs:subClassOf travel:GeographicalFeature .

travel:GeographicalFeature
     rdfs:subClassOf travel:Natural_physical_thing .

travel:Natural_physical_thing
     rdfs:subClassOf travel:Natural_entity .

travel:Natural_entity
     rdfs:subClassOf travel:Physical_entity .

travel:Harbour rdfs:subClassOf [
     rdf:type owl:Restriction ;
     owl:allValuesFrom travel:Physical_entity ;
     owl:onProperty [
          travel:locatedIn rdf:type owl:DatatypeProperty ;
          rdfs:comment "A custom property defined for checking advanced test 1"
     ]
] .
```

SPARQL Query

```
SELECT ?harb
WHERE {
    ?harb travel:locatedIn ?slake .
    ?harb rdf:type travel:Harbour .
    ?slake rdf:type travel:Physical_entity .
    ?slake rdf:type travel:SaltLake }
```

A.5.2 Redundant Triple Pattern Elimination (owl:intersectionOf)

Preconditions

```
ldbc:Event_Place_Theme
     owl:intersectionOf ( dbpedia-owl:Event dbpedia-owl:Place core:Theme  ) .
```

SPARQL Query

```
SELECT ?ev
WHERE {
    ?ev rdf:type ldbc:Event_Place_Theme
    ?ev rdf:type dbpedia-owl:Event .
    ?ev rdf:type dbpedia-owl:Place .
    ?ev rdf:type core:Theme }
```

A.5.3 Star Query Transformation (owl:SymmetricProperty)

Preconditions

```
@prefix travel: <http://www.co-ode.org/roberts/travel.owl#> .

travel:nextTo
     rdf:type owl:SymmetricProperty, owl:ObjectProperty .
```

SPARQL Query

```
SELECT ?st ?rs ?do
WHERE {
    ?cntry rdf:type travel:Country .
    ?cntry travel:hasHeadOfState ?st .
    ?cntry travel:hasRecognitionStatus ?rs .
    ?cntry travel:dependencyOf ?do .
    ?cntry1 travel:nextTo ?cntry }
```

A.5.4 Intermediate Results Reduction (owl:sameAs)

Preconditions

```
bbc:sameAs
     owl:equivalentProperty owl:sameAs .

<http://www.bbc.co.uk/things/g3947cd18-59c0-45e7-aa00-4252b335a111#id>
     a news:Person ;
     bbc:preferredLabel "Diane Abbott" ;
     bbc:sameAs <http://dbpedia.org/resource/Diane_Abbott>,
          <http://www.wikidata.org/wiki/Q153454>,
          <http://rdf.freebase.com/ns/m.0kmws> .
```

SPARQL Query

```
SELECT ?bdate
WHERE {
    <http://www.bbc.co.uk/things/g3947cd18-59c0-45e7-aa00-4252b335a111#id>
       dbpedia-owl:birthDate ?bdate ;
```

```
      rdf:type ?class .
    ?y rdf:type xsd:date .
    ?class rdfs:subClassOf ?class2 .
    ?class2 rdfs:subClassOf core:Thing }
```

A.5.5 Cardinalities Estimation (owl:TransitiveProperty)

Preconditions

```
ldbc:partOf
     rdf:type rdf:Property, owl:TransitiveProperty .

things:webdocument-1#id
     ldbc:partOf things:webdocument-2#id .

things:webdocument-2id
     ldbc:partOf things:webdocument-3#id .

things:webdocument-3#id
     ldbc:partOf things:webdocument-4#id .

things:webdocument-4#id
     ldbc:partOf things:webdocument-5#id .

things:webdocument-5#id
     ldbc:partOf things:webdocument-6#id .

things:webdocument-6#id
     ldbc:partOf things:webdocument-7#id .

things:webdocument-7#id
     ldbc:partOf things:webdocument-8#id .
```

SPARQL Query

```
SELECT ?wd2 ?pcont ?prod ?plat
WHERE {
        ?wd rdf:type bbc:WebDocument .
        ?wd ldbc:partOf ?wd2 .
        ?wd bbc:primaryContent ?pcont .
        ?wd bbc:product ?prod .
        ?wd bbc:platform ?plat }
```

REFERENCES

[1] The CIDOC-CRM Conceptual Reference Model. http://www.cidoc-crm.org/.

[2] D. J. Abadi, A. Marcus, S. R. Madden, and K. Hollenbach. Scalable semantic web data management using vertical partitioning. In *VLDB*, 2007.

[3] D. J. Abadi, A. Marcus, S. R. Madden, and K. Hollenbach. SW-Store: A Vertically Partitioned DBMS for Semantic Web Data Management. *VLDB Journal*, 18(2), April 2009.

[4] V. Alexiev. Implementing CIDOC CRM search based on fundamental relations and OWLIM rules. In *SDA*, 2012. In conjunction with TPDL.

[5] V. Alexiev. Extending OWL2 Property Constructs with OWLIM Rules. Draft, http://vladimiralexiev.github.io/pres/extending-owl2/index.html, August 2014.

[6] V. Alexiev, J. Cobb, G. Garcia, and P. Harpring. Getty Vocabularies Linked Open Data: Semantic Representation. http://vocab.getty.edu/doc/, 2014.

[7] V. Alexiev, D. Manov, J. Parvanova, and S. Petrov. Large-scale Reasoning with a Complex Cultural Heritage Ontology (CIDOC CRM). In *Workshop on Practical Experiences with CIDOC CRM and its Extensions (CRMEX)*, 2013. In conjunction with TPDL.

[8] L. Baolin and H. Bo. HPRD: A High Performance RDF Database. In *Network and Parallel Computing*, pages 364–374, 2007.

[9] B. Bishop and S. Bojanov. Implementing OWL 2 RL and OWL 2 QL Rule-Sets for OWLIM. In M. Dumontier and M. Courtot, editors, *OWL: Experiences and Directions (OWLED)*, volume 796 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2011.

[10] C. Bizer and A. Schultz. The Berlin SPARQL Benchmark. *Int. J. Semantic Web and Inf. Sy.*, 5(2), 2009.

[11] J. Bock, P. Haase, Q. Ji, and R. Volz. Benchmarking owl reasoners. In *ARea2008 Workshop*, 2008.

[12] D. Brickley and R.V. Guha. RDF Vocabulary Description Language 1.0: RDF Schema. www.w3.org/TR/2004/REC-rdf-schema-20040210, 2004.

[13] J. Broekstra, A. Kampman, and F. Van Harmelen. Sesame: A generic architecture for storing and querying RDF and RDF Schema. In *ISWC*. Springer, 2002.

[14] E. I. Chong, S. Das, G. Eadon, and J. Srinivasan. An efficient SQL-based RDF querying scheme. In *VLDB*, 2005.

[15] LDBC Consortium. Semantic Publishing Benchmark: Additional Datasets. https://github.com/ldbc/ldbc_spb_optional_datasets, 2014.

[16] Open Geospatial Consortium. GeoSPARQL - A Geographic Query Language for RDF Data. http://www.opengeospatial.org/standards/geosparql, September 2012.

[17] Ontotext Corp. KIM Showcase: LatestNews Faceted Search. http://ln.ontotext.com/KIM/screen/CoreSearch.jsp, 2006-2011.

[18] Ontotext Corp. GraphDB-SE Geo-spatial Extensions. http://owlim.ontotext.com/display/GraphDB6/GraphDB-SE+Geo-spatial+Extensions, 2012.

[19] Ontotext Corp. OWLIM-SE Reasoner. http://owlim.ontotext.com/display/OWLIMv54/OWLIM-SE+Reasoner, 2013.

[20] Ontotext Corp. GraphDB Connectors. http://owlim.ontotext.com/display/GraphDB6/GraphDB+Connectors, 2014.

[21] Ontotext Corp. GraphDB Lucene4 Plug in (deprecated). http://owlim.ontotext.com/display/GraphDB6/Lucene4+Plug-in+%28deprecated%29, 2014.

[22] Ontotext Corp. Solr GraphDB Connector. http://owlim.ontotext.com/display/GraphDB6/Solr+GraphDB+Connector, 2014.

[23] M. Dean and G. Schreiber. OWL Web Ontology Language Reference. http://www.w3.org/TR/owl-ref, 2004.

[24] O. Erling and I. Mikhailov. RDF Support in Virtuoso DBMS. In *Networked Knowledge - Networked Media, SCI*, 2009.

[25] B. McBride F. Manola, E. Miller. RDF Primer. www.w3.org/TR/rdf-primer, February 2004.

[26] Apache Software Foundation. Solr Wiki: Simple Facet Parameters. https://wiki.apache.org/solr/SimpleFacetParameters, 2013.

[27] I. Fundulaki. D1.1.1: Overview and Analysis of Existing Benchmark Frameworks. LDBC Deliverable D1.1.1, 2013.

[28] I. Fundulaki. D4.4.1: Use case analysis and classification of choke points. LDBC Deliverable D4.4.1, 2013.

[29] I. Fundulaki, N. Martinez, R. Angles, B. Bishop, and V. Kotsev. D2.2.2 Data Generator. Technical report, Linked Data Benchmark Council, 2013. Available at http://ldbc.eu/results/deliverables.

[30] T. Gardiner, D. Tsarkov, and I. Horrocks. Framework for an automated comparison of description logic reasoners. In *ISWC*. Springer, 2006.

[31] V. Haarslev and R. Müller. Racer system description. In *Automated Reasoning*. Springer, 2001.

[32] A. Harth, J. Umbrich, A. Hogan, and S. Decker. YARS2: A Federated Repository for Querying Graph Structured Data from the Web. In *ISWC*, 2007.

[33] O. Hartig and R. Heese. The SPARQL Query Graph Model for Query Optimization. In *ESWC*, 2007.

[34] P. Hayes. RDF semantics. http://www.w3.org/TR/rdf-mt/, 2004. W3C Recommendation, 10 February 2004.

[35] P. Hitzler. Suggestions for OWL 3. In Rinke Hoekstra and Peter F. Patel-Schneider, editors, *OWL: Experiences and Directions (OWLED)*, volume 529 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2008.

[36] A. Isaac and E. Summers. SKOS Simple Knowledge Organization System Primer. W3C Working Group Note, urlhttp://www.w3.org/TR/skos-primer/, 2008.

[37] A. Kiryakov, D. Ognyanov, and D. Manov. OWLIM–a pragmatic semantic repository for OWL. In *WISE 2005 Workshops*. Springer, 2005.

[38] H. Knublauch. Google group "topbraid suite users". https://groups.google.com/forum/#!topic/topbraid-users/f0iUMQCPsDc, May 2014.

[39] J. Lu, F. Cao, L. Ma, Y. Yu, and Y. Pan. An Effective SPARQL Support over Relational Databases. In *SWDB-ODBIS*, 2007.

[40] M. Luther, T. Liebig, S. Böhm, and O. Noppens. Who the heck is the father of bob? In *The Semantic Web: Research and Applications*. Springer, 2009.

[41] L. Ma, Y. Yang, Z. Qiu, G, Xie, Y. Pan, and S. Liu. Towards a Complete OWL Ontology Benchmark. In *ESWC*, 2006.

[42] D. McGuiness and F. v. Harmelen. OWL Web Ontology Language Overview. `http://www.w3.org/TR/owl-features`, 2004.

[43] D. L. McGuinness and F. van Harmelen. OWL Web Ontology Language. `http://www.w3.org/TR/owl-features/`, 2004.

[44] M. Morsey, J. Lehmann, S. Auer, and A-C. N. Ngomo. DBpedia SPARQL Benchmark - Performance assessment with real queries on real data. In *ISWC*, 2011.

[45] B. Motik, B. Cuenca Grau, I. Horrocks, Z. Wu, A. Fokoue, and C. Lutz. OWL 2 Web Ontology Language Profiles (Second Edition). http://www.w3.org/TR/owl2-profiles/. W3C Recommendation 11 December 2012.

[46] B. Motik and R. Studer. KAON2–A scalable reasoning tool for the semantic web. In *ESWC*, 2005.

[47] T. Neumann and G. Moerkotte. Characteristic Sets: Accurate Cardinality Estimation for RDF Queries with Multiple Joins. In *ICDE*, 2011.

[48] T. Neumann and G. Weikum. RDF-3X: a RISC-style engine for RDF. *PVLDB*, 1(1), 2008.

[49] T. Neumann and G. Weikum. Scalable join processing on very large RDF graphs. In *SIGMOD*, June 2009.

[50] T. Neumann and G. Weikum. The RDF-3X engine for scalable management of RDF data. *The VLDB Journal*, 19(1), 2010.

[51] Answer on semanticweb.com. Property intersection (im)possible in OWL 2 Full? `http://answers.semanticweb.com/questions/11602/property-intersection-impossible-in-owl-2-full`, September 2011.

[52] P. F. Patel-Schneider, P. Hayes, and I. Horrocks. OWL Web Ontology Language Semantics and Abstract Syntax. `http://www.w3.org/TR/owl-semantics/`, 2004.

[53] H. Patni, C. Henson, and A. Sheth. Linked sensor data. In *CTS*, 2010.

[54] N. Redaschi and UniProt Consortium. UniProt in RDF: Tackling Data Integration and Distributed Annotation with the Semantic Web. In *Biocuration Conference*, 2009.

[55] S. Rudolph, M. Krötzsch, and P. Hitzler. Cheap Boolean Role Constructors for Description Logics. In *JELIA*. Springer, 2008.

[56] A. Schmidt, F. Waas, M. Kersten, M. Carey, I. Manolescu, and R. Busse. XMARK: A benchmark for xml data management. In *VLDB*, 2002.

[57] R. Shearer, B. Motik, and I. Horrocks. Hermit: A highly-efficient owl reasoner. In *OWLED*, volume 432, 2008.

[58] E. Sirin, B. Parsia, B. C. Grau, A. Kalyanpur, and Y. Katz. Pellet: A practical OWL-DL reasoner. *Web Semantics: science, services and agents on the World Wide Web*, 5(2):51–53, 2007.

[59] Barton Dataset. `http://simile.mit.edu/wiki/Dataset:_Barton`.

[60] Berlin SPARQL Benchmark (BSBM). http://wifo5-03.informatik.uni-mannheim.de/bizer/berlinsparqlbenchmark/.

[61] Berlin SPARQL Benchmark (BSBM) Specification - V3.1. http://wifo5-03.informatik.unimannheim.de/bizer/berlinsparqlbenchmark/spec/index.html.

[62] The DBLP Computer Science Bibliography. http://www.informatik.uni-trier.de/~ley/db/.

[63] DBPedia. http://dbpedia.org/sparql.

[64] DBPSB. http://aksw.org/Projects/DBPSB.

[65] Geonames. http://www.geonames.org/.

[66] LUBM. http://swat.cse.lehigh.edu/projects/lubm.

[67] TPC-H. http://www.tpc.org/tpch/default.asp.

[68] UniProtKB Queries. urlhttp://www.uniprot.org/help/query-fields.

[69] Wikipedia. The Free Encyclopedia. http://en.wikipedia.org/wiki/Wikipedia.

[70] M. Stocker, A. Seaborne, A. Bernstein, C. Kiefer, and D. Reynolds. SPARQL basic graph pattern optimization using selectivity estimation. In *WWW*, 2008.

[71] F. M. Suchanek, G. Kasneci, and G. Weikum. Yago: a core of semantic knowledge. In *WWW*, 2007.

[72] D. Tsarkov and I. Horrocks. Fact++ description logic reasoner: System description. In *Automated reasoning*, pages 292–297. Springer, 2006.

[73] P. Tsialiamanis, L. Sidirourgos, I. Fundulaki, P. Boncz, and V. Christophides. Heuristics-based Query Optimisation for SPARQL. In *EDBT*, 2012.

[74] K. Tzompanaki and M. Doerr. A New Framework for Querying Semantic Networks. Technical Report TR-419, ICS-FORTH, May 2011.

[75] K. Tzompanaki and M. Doerr. Fundamental Categories and Relationships for intuitive querying of CIDOC-CRM based repositories. Technical Report TR-429, ICS-FORTH, April 2012.

[76] M.-E. Vidal, E. Ruckhaus, T. Lampo, A. Martinez, J. Sierra, and A. Polleres. Efficiently Joining Group Patterns in SPARQL Queries. In *ESWC*, 2010.

[77] W3C OWL Working Group. OWL 2 Web Ontology Language. http://www.w3.org/TR/owl2-overview/, 2012.

[78] C. Weiss, P. Karras, and A. Bernstein. Hexastore: sextuple indexing for semantic web data management. *PVLDB*, 1(1), 2008.

[79] T. Weithöner, T. Liebig, M. Luther, S. Böhm, F. Von Henke, and O. Noppens. Real-world reasoning with owl. In *The Semantic Web: Research and Applications*, pages 296–310. Springer, 2007.