Journal on Data Semantics
https://doi.org/10.1007/s13740-021-00133-y

ORIGINAL ARTICLE

SPARQL Query Generator (SQG)

Yanji Chen[1] · Mieczyslaw M. Kokar[1] · Jakub J. Moskal[2]

Received: 25 August 2020 / Revised: 1 May 2021 / Accepted: 18 June 2021

Abstract
This paper describes a program—SPARQL Query Generator (SQG)—which takes as input an OWL ontology, a set of object descriptions in terms of this ontology and an OWL class as the context, and generates relatively large numbers of queries about various types of descriptions of objects expressed in RDF/OWL. The intent is to use SQG in evaluating data representation and retrieval systems from the perspective of OWL semantics coverage. While there are many benchmarks for assessing the efficiency of data retrieval systems, none of the existing solutions for SPARQL query generation focus on the coverage of the OWL semantics. Some are not scalable since manual work is needed for the generation process; some do not consider (or totally ignore) the OWL semantics in the ontology/instance data or rely on large numbers of real queries/datasets that are not readily available in our domain of interest. Our experimental results show that SQG performs reasonably well with generating large numbers of queries and guarantees a good coverage of OWL axioms included in the generated queries.

Keywords Synthetic SPARQL query generation · Ontology-based descriptions · Cognitive radios · OWL axioms · Jena ARQ · SPARQL query evaluation

1 Introduction

The work described in this paper resulted from our involvement in the development of applications for cognitive radio networks where individual radios (“RF devices,” or “nodes”) are equipped with various kinds of capabilities such as sensing, transmitting, receiving and computing. A node may provide and request services to and from other nodes in the network. Additionally, applications may request services on nodes in the network and the network then can match the radio capabilities against the requests. In all such scenarios, matching the requests against the device capabilities need to be performed in order to derive decisions on which devices should be used to satisfy a specific request.

The whole process can be viewed as a *represent - infer - query* cycle, which is executed by the so-called cognitive engines (CE) or reasoners.

While the majority of the software-defined radios in use today are configured and described in languages based on XML, vendors started to claim that their reasoners are based on the semantic languages, OWL in particular. Customers who want to choose a specific reasoner are then faced with the problem—which reasoner to choose?

Testing of such reasoners would need to include their capabilities to represent communication networks and derive facts that are not explicitly represented but are derivable via the inference rules of the OWL language. This kind of testing requires (1) generation of large collections of facts describing the networks (in OWL) and (2) generation of large and versatile collections of queries (in SPARQL) for retrieving information from such representations. In such a setting, both the generation and the retrieval processes must be based on the ontologies. Since the inference depends on the axioms of OWL, the testing must ensure that the whole power of OWL inference, i.e., all the types of the OWL axioms that appear in the ontology, is used in both descriptions and queries.

We have described a description generator in [9]. In this paper, we are focusing on the query generator. We are dealing with a more general use case that is not limited to cognitive

✉ Mieczyslaw M. Kokar
m.kokar@northeastern.edu

Yanji Chen
chen.yanj@northeastern.edu

Jakub J. Moskal
jmoskal@vistology.com

1 Department of Electrical and Computer Engineering, Northeastern University, Boston, MA, USA

2 VIStology Inc., Framingham, MA, USA

networks, but to any systems where reasoners are used to infer facts from OWL encoded descriptions, and then, SPARQL is used to query the fact base.

To ensure that the evaluation results carry a high level of credibility, it is required to use large numbers of requests of different types for matching against large pools of device descriptions.

The queries must have sufficient semantics so that the device capability matching system can precisely retrieve devices of a desired type. The collections of queries must be highly diversified to be representative of the whole tested space of the requests.

Theoretically, the problem could be resolved using real queries, or at least queries collected from real radio networks. Unfortunately, this kind of queries is not readily available due to various reasons. So the only practical solution is to use synthetic queries, instead. A wide literature search for software for this purpose has been performed. However, none of the tools and published approaches satisfy our requirements (stated in the next section). Consequently, a SPARQL query generator (SQG) program was developed taking into account the various lessons learned from the related work. The inputs to SQG are a domain ontology encoded in OWL and an OWL class selected from this ontology (we call it the "root class") that represents the type of objects of interest to the user. In our RF domain use case, the SDR ontology [32] and the *RFSystem* class were used in these roles.

In this paper, SQG is described and the evaluation of its features is presented. The generation approach focuses on OWL axioms coverage and diversity coverage such that the generated queries are highly diversified and aim at matching both explicit and implicit descriptions of objects of a specific type in a domain ontology. While our main objective is to use it for generating requests for services against descriptions of RF device capabilities based on an RF domain ontology, we believe that it is also applicable to other domain ontologies. To support this expectation, the results of using SQG for generating SPARQL queries against the datasets based on five other ontologies are also provided.

The rest of this paper is structured as follows. Section 2 reviews related work on SPARQL query generation. Section 3 formalizes basic concepts need to define the SPARQL query generation process. Section 4 presents an overview of SQG, followed by its implementation described in Sect. 5. Evaluation of SQG and query sets is presented in Sect. 6. Finally, conclusions and discussion are in Sect. 7.

2 Related Work

The literature search for methods of SPARQL query generation was guided by the following requirements that capture the needs of the use case described in the previous section.

1. **Query pattern satisfaction**: All of the generated SPARQL queries must be for retrieving objects of a specific type (root class) against RDF descriptions of objects.
2. **OWL axioms coverage**: The query generation process should have sufficient coverage of the OWL 2 axioms [28]. In other words, the generator must use OWL axioms in a given ontology in the generation process to ensure that the generated queries cover the semantics (explicit and implicit) in the ontology in which the dataset is represented.
3. **Space coverage**: The generated queries should have a good coverage of the following three characteristics: signature coverage, operator coverage and structure coverage. Signature coverage means including in the queries the concepts from the signatures of the ontologies given as input. Operator coverage means including the SPARQL operators({AND, UNION, OPTIONAL, MINUS, FILTER NOT EXISTS, FILTER EXISTS}). Structure coverage means the queries should have diverse structures.
4. **Scalability**: The method should be able to generate large numbers of queries.

Various approaches to query generation have been developed in the past years. We classified the approaches into seven groups described below.

2.1 Developing Queries by Hand

In this approach, a rather small number of carefully designed queries are provided by a testing system. LUBM [16] is a benchmark designed for testing the efficiency of OWL-based knowledge base systems (KBS). Fourteen test SPARQL queries about the university domain are provided. The queries are based on a specific ontology (Univ-Bench) [41]. The queries are realistic, and the selection criteria took into account five factors: input size, selectivity, complexity, assumed hierarchy information and assumed logical inference. Several works based on LUBM were described in [23,24,43]. LBBM [43] aims at testing the KBS against the knowledge base that commits to different benchmark ontologies. LBBM offers twelve test queries. UOBM [23], an extension of LUBM, provides fifteen test SPARQL queries for testing inference capabilities and scalability of ontology systems. EvoGen [24] is a synthetic Benchmark Suite for evolving data. Besides original LUBM queries, it provides custom queries that are commonly performed in the evolving data, such as retrieval of a diachronic dataset or a specific version, longitudinal queries across versions. The focus of this tool is queries about changes in the datasets.

Schmidt et al. [31] presented SP^{2}Bench, a language-specific benchmark designed to test SPARQL characteristics imposed on target SPARQL engines. It comprises seventeen meaningful SPARQL queries against benchmark datasets that mirror key characteristics and distributions of Digital Bibliography & Library Project (DBLP). The queries were carefully designed to have a good coverage of most common SPARQL constructs, operator constellations and a broad range of RDF data access patterns for query optimization.

Hasse et al. [17] defined a benchmark for the purpose of analyzing various design choices for federating distributed data sources. To this end, fourteen SPARQL queries were manually developed as the benchmark queries, where seven are for cross-domain benchmark datasets, and the rest are for benchmark datasets in the life-sciences domain. The queries represent real-life use cases. However, the authors did not aim at the completeness with respect to the features of the SPARQL language, but instead focused on the aspects that are relevant in the context of the query processing over multiple data resources [17].

Owens et al. [27] developed a configurable benchmark for measuring the performance of RDF stores at a low, diagnostic level. For this purpose, a number of SPARQL queries were meticulously designed for testing factors of the broader categories of assertion, deletion and query.

Kotsev et al. [22] presented the Semantic Publishing Benchmark (SPB) for performance assessment against Virtuoso [13] and GraphDB [15] RDF engines based on scenarios of the BBC media organization. The queries were elaborately designed to simulate a constant load generated by end-users, journalists, editors or automated engines. The set comprises 11 aggregation queries with good coverage of choke points (technical functionalities that systems must tackle) and three update queries with insert/update/delete operations.

Manual SPARQL query generation has some advantages—it is simple and configurable. Developers have full control over the queries for specific purposes without committing to fixed patterns. However, this method is not scalable since it is not feasible to develop a large number of queries by hand.

2.2 Generating Queries from Manually Developed Query Templates

The main idea of this approach is that SPARQL query expressions can be automatically constructed based on query templates by replacing placeholders within the templates with values in a knowledge base. Bizer et al. [7] introduced the approach, called BSBM, and applied it to an e-commerce use case for comparing the performance of four storage systems that expose SPARQL endpoints. Ten templates were designed that emulate the search and navigation patterns of a consumer looking for products. In the query instantiation process, the placeholders within the templates are replaced with values from the benchmark dataset.

RDFUnit [20] is an open source tool for evaluating the quality of linked data based on the methodology for test driven quality assessment of linked data. It contains a pre-defined pattern library that consists of seventeen Data Quality Test Patterns (DQTP). Each pattern aims at evaluating one type of constraints satisfaction of datasets against ontologies. A pattern is further instantiated into concrete data quality test cases (SPARQL queries) by filling the placeholders with terms from an ontology.

This approach is also widely used in question answering (QA). The related work in this area provides some systems that translate natural language questions by users into SPARQL queries. Shekarpour et al. proposed an approach for constructing SPARQL queries from keywords. The method utilizes seventeen pre-defined basic graph pattern templates [34]. A filtered set of resource candidates in the ontologies obtained for two user-supplied keywords are injected into placeholders of suitable graph pattern templates. Since the graph pattern templates are pre-defined, which are agnostic to the underlying knowledge base and ontology schema, the quality of the generated queries is good only if the keywords and ontology are compatible. Atozori et al. presented QA^3 [5], a statistical question answering system over RDF data cubes. The system first tags a question with elements in a knowledge base belonging to the same dataset. Then, the question is tokenized with the tags. A sequence of tokens are then matched against an extensible set of regular expressions, each of which is associated with a SPARQL template. The chosen template is then filled with SPARQL fragments by using the tags and the structure of the dataset.

Pre-defined query templates are widely used for query generation. The approach is configurable and scalable. However, since pre-defined query templates can hardly cover sufficient axioms in an ontology, and queries generated from the same template have an identical structure, the generated queries do not sufficiently cover the semantics in the ontology, nor do they have diversified structures.

2.3 Generating Queries from Requests in Natural Language

This approach aims to translate natural language questions expressed as keywords or sentences into SPARQL queries. Note that some of the related work in this group is based on pre-defined graph pattern templates, as summarized in Sect. 2.2.

Shekarpour et al. proposed an approach to automatically generate query graphs from keyword queries or natural language queries [33] over federated linked data. The main idea is to first generate incomplete query graphs (IQGs) that contain relevant resources for the input query and then use an extension of the minimum spanning tree (MST) method to connect IQGs into the whole query graph that fully covers the relevant resources. The query graph is converted into a conjunctive federated SPARQL query.

QUICK [44] is a system to help users to construct SPARQL queries in pre-defined domain specific ontologies using keywords. Initially, all possible query graph templates are identified by the schema graph of a given ontology. Query graphs are instantiated by binding keywords to terms (literals, concepts and properties) in the templates. The query construction process is conducted by a query guide generated by the proposed algorithm, which directs the user to specify query intention until the query graph reflects the actual intention of the user. The query graph is transformed into SPARQL in the end.

Unger et al. [40] proposed an approach to generate SPARQL queries that capture the semantic structure of a user's requests in natural language. First, an input question is processed to create lexical entries for parsing and construct semantic representations of the question. The semantic representations are then translated into SPARQL templates. The slots of the templates are filled by the entities identified in the given RDF data, which produces a range of query candidates of the input questions. These queries are ranked based on the query score that combines a similarity score and a prominence score of the entities filled in the slots. The highest score query with a non-empty result is returned to the user.

The natural language approach is widely used in question answering (QA) due to its convenience. A user simply provides requests as keywords or sentences in natural language without the need to master the query language nor acquire specific details of the background knowledge. However, the approach does not fit our problem. It is semi-automated, which requires the user's input and may require the user to incrementally express the intent of queries during the process of constructing queries. It is not scalable—not feasible to generate a large number of queries. Due to the ambiguity of natural language and the limitations of the query translation capabilities, the constructed queries may not precisely reflect the intentions of the user.

2.4 Generating Queries Based on Given Queries

Morsey et al. [25] proposed the DBpedia SPARQL Benchmark (DBPSB) for evaluating the performance of four triple stores, where a generic SPARQL query generation methodology based on existing queries was described. First, the queries from a DBpedia SPARQL query-log are reduced based on query variations and frequency. Second, the remaining queries are clustered by computing query similarity. Thereafter, 25 representative queries that cover the SPARQL features of interest are selected based on the cluster ranking and the query frequency. Each selected query is then converted into a SPARQL query template by replacing a part of the query with placeholders. The actual SPARQL queries are generated by filling the placeholders with retrieved concrete values in the datasets. As an extension of this method, FEASIBLE [30] considers more query features (such as the number of join vertices and triple patterns selectivities) in the query selection process, which in turn produces better sample queries.

However, this approach is not applicable to our domain, as well as to other domains where relatively large numbers of real queries are not available.

2.5 Generating Queries from Datasets

Queries can be constructed by making use of graphical structure of RDF instance data. Qiao et al. proposed an application-specific benchmark named RBench [29], where a flexible query workload generation process from a given RDF dataset is proposed. The query generation process contains three steps: dataset preprocessing, dataset analysis and benchmark graph pattern generation. RBench preprocesses a given dataset to identify resource types, relationship triples and attribute triples, followed by the generation of relationship patterns and attribute patterns in the data analysis step. Graph pattern generation utilizes the results in dataset analysis to generate five types of queries (node queries, edge queries, star queries, cyclic queries and subgraph queries). The first four types of queries are generated by formulating triple patterns from selected relationship patterns and attribute patterns using resource types. Subgraph queries are generated by first selecting a subgraph from the given RDF graph, and then generating SPARQL queries from the subgraph. The queries are structurally diversified and have a good coverage of entities in the given datasets.

Görlitz et al. proposed SPLODGE [14], a systematic SPARQL benchmark for federated linked data. It provides a methodology for a systematic and scalable query generation. The methodology consists of three steps—query parameterization, query generation and query validation. The first step is to select and combine query parameters (query algebra, query structure and query cardinality) to fit desired evaluation scenarios. After that, it preprocesses the datasets and computes statistics. The query generation employs iterative combination of query patterns based on the statistical information. At last, the verification of generated queries is conducted by computing confidence value based on query selectivity. Queries are not accepted if their confidence values lie below a pre-defined threshold. However, this approach is only applicable to federated linked data scenarios. The implementation of SPLODGE is incomplete since only a few query features described in the paper are supported.

Apart from the limitations specific to each of the approaches described in those papers, some common limitations of the methods stand out: Since no background knowledge is involved, these systems cannot precisely retrieve matching results for the queries through logical inference. Moreover, since queries generated from small datasets cannot cover the whole test space of the application requests, the methods are inapplicable to our domain nor to other domains where very limited real RDF datasets are available or accessible.

2.6 Generating Queries from a Pre-defined Schema

Aluç et al. [1] developed the WatDiv system for stress-testing of RDF management systems against a wide spectrum of SPARQL queries with varying structural characteristics and selectivity classes. Its main component is a query (and template) generator of SPARQL queries against datasets. The benchmark queries are generated in two steps. First, a set of query templates are generated by performing a random walk on the data model represented in the WatDiv dataset description language. To this end, first a set of queries, referred to as basic graph patterns (BGP) with unbounded subjects and objects but bounded predicates, is created. For each BGP generation, a graph vertex for every entity type in the schema is created, followed by the connections between the vertices represented by the graph edges according to the associations of the corresponding entity types specified in the schema. Then, the query templates are randomly selected from the queries by replacing a number of randomly selected subjects/objects with placeholders. In the second step, the placeholders in each query template are instantiated dynamically from the datasets by the query generator.

gMark [6] is a domain and query language independent framework targeting highly tunable generation of both graph instances and graph query workloads based on user-defined schemas for the purpose of evaluating graph query processing engines. A schema is a configuration file in XML that allows users to specify query workload configurations, including query shape (chain, star, cycle, star-chain), the number of conjuncts/disjuncts, etc. It is the first benchmark for generating workloads exhibiting recursive path queries, which are central to graph querying [6].

The schema-driven approach is scalable and aims at generating diverse queries. However, since a pre-defined schema lacks semantics, matching to the queries does not facilitate logical inference.

2.7 Generating Queries from an Ontology

The main idea of this approach is to construct queries progressively based on concepts and their relations in an ontology. Dibowski et al. [11] presented a novel approach to modeling, representing, viewing, accessing and storing device descriptions with semantic web techniques for building automation devices. To save users from writing SPARQL queries manually, a generic search mask—a user-friendly graphical user interface (GUI) for generating SPARQL queries against device descriptions, is proposed. It is initialized according to a specific ontology view that lists available concepts of interest and their associated properties in an XML-based document. The display shows the knowledge specific for the view with tabs. In the attribute tab, users are allowed to edit data properties and define values of required devices. Concepts can be edited using the provided tabs, e.g., object properties of required devices. After that, a SPARQL query is dynamically generated by a query generation algorithm that combines all the device requirements together. In general, this approach is the closest to addressing the problem addressed in this paper. The queries satisfy the restrictions of the query pattern.

However, it does not fully satisfy the rest of the requirements of our problem. In particular, a pre-defined ontology view (XML) for an ontology does not sufficiently capture the semantics encoded in the ontology. Moreover, this approach is semi-automatic since the users have to configure concepts and properties manually by editing the search mask for each generated query. To address these limitations, SQG extracts and processes the OWL 2 axioms (explicit and implicit) in an input ontology using the OWL API [18], which in turn guarantees good coverage of semantics in the queries. SQG generates random queries automatically, where the randomness is controlled by a set of probability thresholds.

2.8 Summary of the Reviewed Literature

Table 1 summarizes the reviewed literature with respect to the satisfaction of the requirements. Each requirement is labeled as *Yes* only if the related benchmark fully satisfies the requirement. Otherwise, it is labeled as *No*. In summary, none of them fully satisfies our requirements. Some of the approaches do not generate queries that satisfy a specified query pattern. Handmade or semi-automated generation approaches are not acceptable when large numbers of queries are required. The queries generated from pre-defined query templates lack a sufficient coverage of the OWL semantics, and the structures of the queries are not diversified. Generating queries based on existing queries or datasets are not applicable to the domains such as the RF domain where large collections of real SPARQL queries or RDF datasets are not available. The queries generated from a pre-defined data model in a language that lacks a declarative semantics cannot sufficiently cover the OWL semantics and thus are not suitable for automatic inference.

3 Formalization of Basic Concepts

This section introduces some definitions needed to formalize the SPARQL query generation process. Note that all of the definitions originate from other papers. Definition 1 comes from [9]. Definitions 2 and 3 are based on [2] and [35], respectively. We adjusted these definitions to our objectives

Table 1 Requirement satisfaction by the reviewed systems

Related work	Query pattern	Axiom coverage	Space coverage	Scalability
LUBM	Yes	No	No	No
LBBM	Yes	No	No	No
UOBM	Yes	No	No	No
EvoGen	Yes	No	No	No
SP^{2}Bench	Yes	No	No	No
[17]	Yes	No	No	No
[27]	Yes	No	No	No
SPB	Yes	No	No	No
BSBM	Yes	No	No	Yes
RDFUnit	Yes	No	No	Yes
[34]	Yes	No	No	Yes
QA^3	Yes	No	No	Yes
[33]	No	Yes	No	No
QUICK	No	Yes	No	No
[40]	No	Yes	No	No
DBPSB	Yes	No	No	Yes
FEASIBLE	Yes	No	No	Yes
RBench	No	No	Yes	Yes
SPLODGE	No	No	No	Yes
WatDiv	No	No	Yes	Yes
gMark	No	No	Yes	Yes
[11]	Yes	No	No	No

of restricting the graph patterns in the generated queries. Definitions 4 and 5 are based on [2] and [8], respectively. Our ultimate goal is to define SPARQL queries for descriptions of objects of a specific type.

Definition 1 [*Knowledge base signature*] Given an ontology O, the knowledge base signature KBS of O is defined as $KBS = \{C, I, DP, OP\}$, where C denotes classes, I denotes individuals, DP denotes data properties, and OP denotes object properties.

Definition 2 [*Triple pattern and basic graph pattern*] A triple pattern is defined as $TP = (s, p, o) \in (I \cup V) \times (DP \cup OP \cup V) \times (C \cup I \cup L \cup V)$, where L denotes literals, V denotes variables, s is the subject, p is the predicate, and o is the object. A basic graph pattern BGP is a finite set of TP.

Definition 3 [*Weakly connected basic graph pattern*] A basic graph pattern BGP, viewed as a directed graph $G = (V, E)$ with vertex set V and edge set E, is a weakly connected graph basic graph pattern BGP^* if for all pairs of vertices $s, t \in V$, there exists a sequence of vertices $s = v_0, v_1, v_2, \ldots, v_k = t$ such that $(v_{i-1}, v_i) \in E$ or $(v_i, v_{i-1}) \in E$ for all $1 \leq i \leq k$.

Definition 4 [*Graph pattern with nested weakly connected basic graph patterns*]

1. A BGP^* is a GP.
2. If P_1 and P_2 are GP, then expressions (P_1 AND P_2), (P_1 OPTIONAL P_2), and (P_1 UNION P_2) are GP (conjunction graph pattern, optional graph pattern and union graph pattern, respectively).
3. If P is a graph pattern and $X \in I \cup V$, then (X GRAPH P) is a graph pattern.
4. If P is a GP and R is a SPARQL built-in condition, then the expression (P FILTER R) is a GP (a filter graph pattern).

A SPARQL built-in condition is constructed using elements of the set $I \cup L \cup V$ and constants, logical connectives ($\neg, \wedge, \vee$), ordering symbols ($<, \leq, \geq, >$), the equality symbol ($=$), unary predicates like bound, isBlank and isIRI, plus other features (see [37] for a complete list).

Definition 5 [*SPARQL query for objects of a specific type*] Given an ontology O and a class $Root \in O$, a SPARQL query q for objects of type R is a tuple $q = (QF, GP^*, SM)$, where $QF = \{\text{SELECT}\}$ is the query form; GP^* is a GP that includes the triple pattern (?v rdf:type R); R = Root or (R rdfs:subClassOf Root); $SM = \{\text{DISTINCT}\}$ is a set of solution modifiers.

Here is an explanation of the above definitions. First of all, SPARQL supports four query types—SELECT, CONSTRUCT, ASK and DESCRIBE. Their main differences are

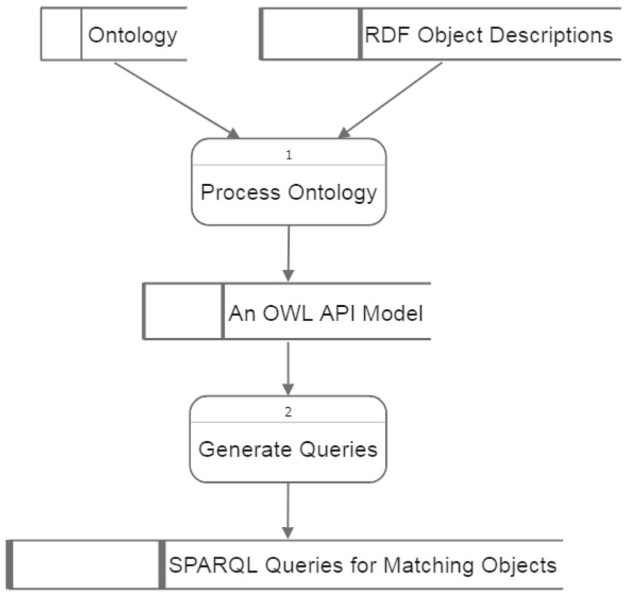


Fig. 1 SQG: top-level processes

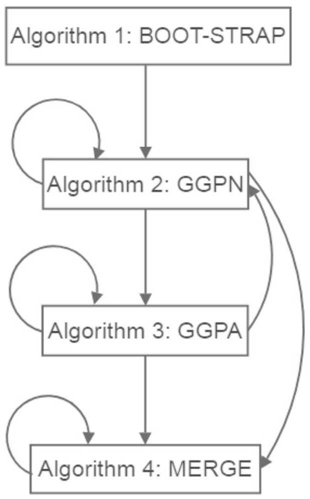


Fig. 2 SQG: algorithms

in the format of the query results. Each type defines query patterns in a WHERE clause and returns a multiset of variable bindings, an RDF graph or a Boolean value. DESCRIBE queries can also take a single URI and return an RDF graph. In our problem, all the requests are about finding objects of a specific type. Therefore, the only query type of interest to us is SELECT.

Second, DISTINCT, REDUCE, LIMIT, OFFSET and ORDER BY alter the result set which is returned by a query. The DISTINCT modifier is required to avoid duplicate matching results. The REDUCE modifier is not needed since it simply permits duplicate solutions to be removed if possible, but not mandatory. The rest of the modifiers are not required since no other operations are needed.

4 SQG Overview

SQG consists of two steps: *Process Ontology* and *Generate Queries* (see Fig. 1).

Process Ontology takes three kinds of input: an OWL ontology, an RDF graph of object descriptions and a root class that represents the type of objects of interest. It constructs a Java model using the OWL API [18]. The RDF dataset is generated by the RODG program [9].

Generate Queries takes the Java model as input and generates a number of SPARQL queries for matching objects. A typical query consists of five parts: query prologue, query form, result variable, query body and result modifiers. The queries are programmatically built using Jena ARQ [4]. The generation process involves two steps. First, generate a graph pattern as the query body. The graph pattern is progressively built starting from the root class. It recursively generates subgraph patterns that include triple patterns associated with the root class and then merges the sub graph patterns in the graph pattern recursively with randomly selected operators from {AND, UNION, OPTIONAL, MINUS, FILTER NOT EXISTS, FILTER EXISTS}. The query prologue is progressively built during the graph pattern generation process. At last, a SPARQL query that combines all the parts into a single query is built.

5 SPARQL Query Generation Algorithms

In this section, we describe only the algorithms for query body generation. The full implementation of SQG is available online [38].

The process is invoked by a boot strap procedure represented in Algorithm 1. It invokes Algorithm 2, which recursively calls the sub-algorithms, as shown in Fig. 2.

The algorithm descriptions use the following notations:

- M: Java model built by Process Ontology.
- $A(C_i)$: Anonymous class expressions of an OWL named class C_i in M, including anonymous super class expressions, anonymous equivalent class expressions and anonymous disjoint class expressions.
- $R(C_i)$: Relevant OWL named classes of an OWL named class C_i in M. It is defined as the union of C_i, subclasses of C_i, superclasses of C_i, equivalent classes with C_i and disjoint classes with C_i.
- $D_{P,R}(C_i)$: Datatype property key-value pairs of an OWL named class C_i; each pair (p,r) consists of a data property p for which C_i is its domain, and its data range r in M.
- $O_{P,R}(C_i)$: Object property key-value pairs of an OWL named class C_i; each pair (p,r) consists of an object

property p for which C_i is its domain, and its range r as an OWL class expression in M.
- $I(C_i)$: OWL named individuals of type C_i in M.
- $V(C_i)$: Variables that bind to OWL individuals of type C_i.
- $anonymous(C_{exp})$: *true* if class expression C_{exp} is anonymous, *false* otherwise.
- $type(C_{exp})$: Type of an OWL class expression C_{exp}; the types of the OWL 2 class expressions are listed in [28].
- $filler(C_{exp})$: Filler of a class expression C_{exp}. According to the OWL API, filler is: an OWL individual, if C_{exp} is of type ObjectHasValue; an OWL literal, if C_{exp} is of type DataHasValue; a class expression, if C_{exp} is of type ObjectSomeValuesFrom, ObjectAllValuesFrom, ObjectMinCardinality, ObjectMaxCardinality or ObjectExactCardinality; a data range, if C_{exp} is of type DataSomeValuesFrom, DataAllValuesFrom, DataMinCardinality, DataMaxCardinality or DataExactCardinality.
- $operands(C_{exp})$: Class expressions referenced by a class expression C_{exp} of type ObjectIntersectionOf, ObjectUnionOf, or ObjectComplementOf. In logical languages, these are called conjunction, disjunction, and negation, respectively [28].
- gp: A graph pattern.
- GP: A one-dimensional array of graph patterns.
- $length(GP)$: Function that returns the number of columns of GP.
- $comb(gp_l, gp_r, P)$: Function that combines two graph patterns using one of the operators from {AND, UNION, OPTIONAL, MINUS, FILTER NOT EXISTS, FILTER EXISTS} selected randomly using the probability thresholds P.
- $rand(S)$: Function that randomly selects a member of set S based on uniform distribution.
- $genVar()$: Function that generates a new variable.
- $genFilterExp(r, ?v_1)$: Function that generates a SPARQL filter expression about $?v_1$ based on data property range r.

Algorithm 1 takes three kinds of input—a model M, a root class R and a vector of probabilities P. It then invokes Algorithm 2 passing to it a class expression C_0 and a variable name. The algorithm returns graph pattern gp.

Example 1: As an example, a small fragment of an OWL API model of the SDR ontology (introduced in Sect. 6) is shown in Fig. 3. The model example is represented as an object diagram, where a rectangle represents an object of a specific class, and a link between two objects shows the corresponding Java model associations between the classes. *RFDevice:OWLClass* represents the root class. The association between this class and *USRPB200:OWLClass* is shown as *subClass/superClass*. *ObjectAllValuesFrom* represents a class expression that involves an *objectPropertyExpression* (an *ObjectProperty hasProducer*) and a *classExpression* (the *Producer* class). The right side of the graph can be interpreted in a similar way. In this example, *USRPB200* is selected as C_0 and passed to Algorithm 2.

Algorithm 1: BOOT-STRAP(M,R,P)

```
Input: $M$: model; $R$: root class; $P$: probability thresholds
Output: $gp$: graph pattern
1 $C_0 \leftarrow rand(\{subClass(R)\} \cup R)$
2 $?v_0 \leftarrow genVar()$
3 $gp \leftarrow GGPN(M, C_0, ?v_0, P)$
```

Algorithm 2: GGPN(M,C_0,$?v_0$,P)

```
Input: $M$: model; $C_0$: OWL named class; $?v_0$: variable; $P$:
       probability thresholds
Output: $gp$: graph pattern
1  $visited(C_0) \leftarrow true$
2  if $rand(0,1) > P[0]$ then
3      $GP[0] \leftarrow GP[0] \cup \{(?v_0, \text{rdf:type}, C_0)\}$
4  if $rand(0,1) > P[1]$ then
5      $e \leftarrow rand(A(C_0))$
6      $GP[1] \leftarrow$ GGPA($M$,$e$,$?v_0$,$P$)
7  if $rand(0,1) > P[2]$ then
8      $?v_1 \leftarrow genVar()$
9      $(p, r) \leftarrow rand(D_{P,R}(C_0))$
10     $GP[0] \leftarrow GP[0] \cup \{(?v_0, p, ?v_1)\}$
11     if $rand(0,1) > P[3]$ then
12         $exp \leftarrow genFilterExp(r, ?v_1)$
13         $GP[0] \leftarrow GP[0] \cup exp$
14 if $rand(0,1) > P[4]$ then
15     $(p, r) \leftarrow rand(O_{P,R}(C_0))$
16     if $anonymous(r)$ then
17         $?v_2 \leftarrow genVar()$
18         $GP[2] \leftarrow$ GGPA($M$,$r$,$?v_2$,$P$)
19         $t \leftarrow (?v_0, p, ?v_2)$
20     else if $rand(0,1) > P[5]$ then
21         $I_0 \leftarrow rand(I(r))$
22         $t \leftarrow (?v_0, p, I_0)$
23     else
24         $C_1 \leftarrow rand(R(r))$
25         if not $visited(C_1)$ then
26             $?v_2 \leftarrow genVar()$
27             $GP[2] \leftarrow$ GGPN($M$,$C_1$,$?v_2$,$P$)
28         else if $rand(0,1) > P[6]$ then
29             $?v_2 \leftarrow genVar()$
30         else
31             $?v_2 \leftarrow rand(V(C_1))$
32         $t \leftarrow (?v_0, p, ?v_2)$
33     $GP[0] \leftarrow GP[0] \cup \{t\}$
34 $gp \leftarrow$ MERGE($GP$,0,2,$P$)
```

Algorithm 2 (GGPN(M,C_0,$?v_0$,P)) implements the generation of graph patterns from the OWL named class C_0. Its structure is based on the depth-first search (DFS) algorithm.

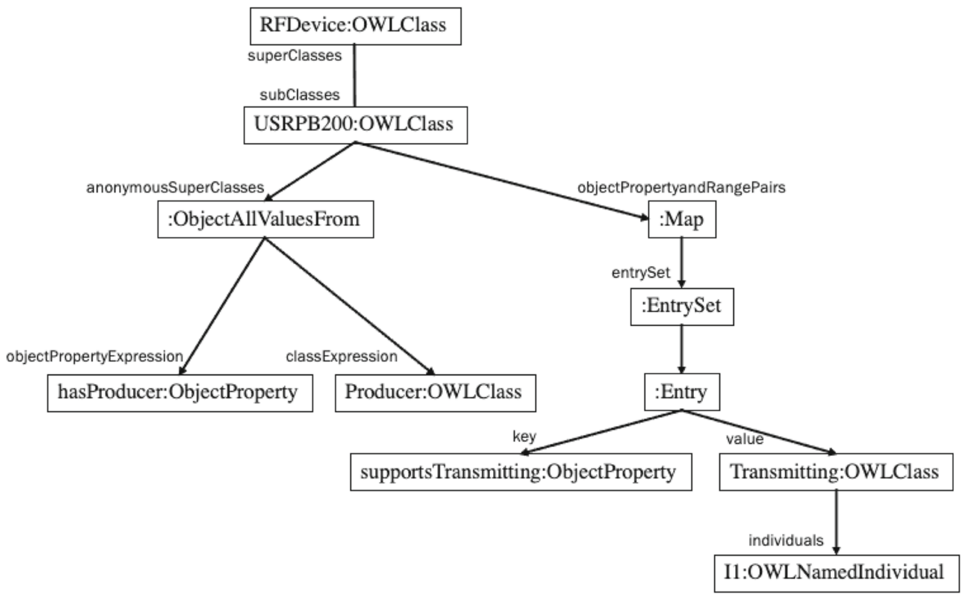
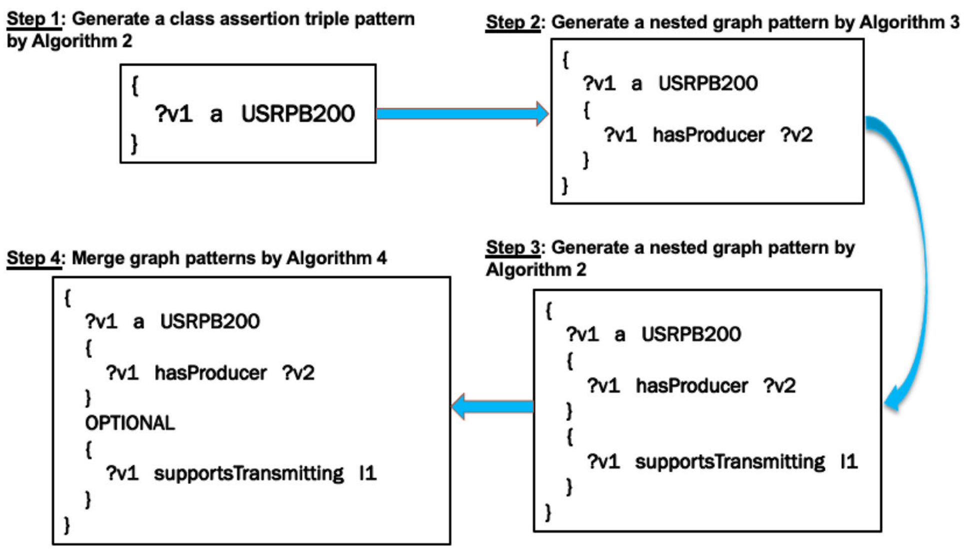


Fig. 3 An OWL API model example


Step 1: Generate a class assertion triple pattern by Algorithm 2
Step 2: Generate a nested graph pattern by Algorithm 3
Step 3: Generate a nested graph pattern by Algorithm 2
Step 4: Merge graph patterns by Algorithm 4

```
{
  ?v1  a  USRPB200
}
```

```
{
  ?v1  a  USRPB200
  {
    ?v1  hasProducer  ?v2
  }
}
```

```
{
  ?v1  a  USRPB200
  {
    ?v1  hasProducer  ?v2
  }
  {
    ?v1  supportsTransmitting  I1
  }
}
```

```
{
  ?v1  a  USRPB200
  {
    ?v1  hasProducer  ?v2
  }
  OPTIONAL
  {
    ?v1  supportsTransmitting  I1
  }
}
```

Fig. 4 SPARQL query body generation example

Graph patterns are initialized as nil and then progressively filled and combined by traversing the model as a directed graph starting from the class C_0. Initially, all OWL named classes are marked as unvisited. The algorithm first marks C_0 as visited and randomly generates a class assertion triple pattern (a class assertion whose subject is a variable). Then, it traverses the restrictions of C_0. The three types of restriction of C_0 are anonymous class expressions $A(C_0)$, key-value pairs $D_{P,R}(C_0)$ and $O_{P,R}(C_0)$. In the processing block of each type of restriction, the algorithm randomly picks exactly one restriction from class expressions/data ranges and then calls either the sub-algorithm GGPA(M,e,$?v_0$,P) or recursively invokes itself, with appropriate parameters. A subgraph pattern is recursively built in the process of selecting the restrictions. At last, the algorithm invokes MERGE(GP,0,2,P) to combine all the three subgraph patterns in GP into one.

In order to guarantee the diversity of the query patterns, random generation of triple patterns and selection of objects are used. This is implemented by setting the probability thresholds $P[]$ for various types of triple pattern generation or element selection. If a randomly generated value is greater than the threshold, the random selection/generation will take place. For instance, a class assertion triple pattern is included when a randomly generated value $0 \leq p \leq 1$ is greater than the class assertion probability threshold $P[0]$. A triple pattern may be randomly replaced with another triple pattern (we term it as a *relevant triple pattern* of the triple pattern) depending on the probability thresholds. For instance, if an object property assertion triple pattern ($?v_0$, p, $?v_1$) is considered, a relevant triple pattern ($?v_1$, p', $?v_0$), where p' is an

Algorithm 3: GGPA(M,C_{anon},$?v_0$,P)

```
Input: $M$: model; $C_{anon}$: anonymous class expression; $?v_0$:
       variable; $P$: probability thresholds
Output: $gp$: graph pattern
switch $type(C_{anon})$ do
    case ObjectIntersectionOf or ObjectUnionOf
        $S \leftarrow rand(2^{\{x|x \subseteq operands(C_{anon})\}})$
        $i \leftarrow 0$
        foreach $e \in S$ do
            if $anonymous(e)$ then
                $GP[i] \leftarrow$ GGPA($M$,$e$,$?v_0$,$P$)
            else
                $C_0 \leftarrow rand(R(e))$
                if $not\ visited(C_0)$ then
                    $GP[i] \leftarrow$ GGPN($M$,$C_0$,$?v_0$,$P$)
            $i \leftarrow i + 1$
        $gp \leftarrow$ MERGE($GP$,0,$length(GP) - 1$,$P$)
    case QuantifiedObjectRestriction
        if $rand(0, 1) > P[4]$ then
            $p \leftarrow getProperty(C_{anon})$
            $C_{exp} \leftarrow filler(C_{anon})$
            if $anonymous(C_{exp})$ then
                $?v_1 \leftarrow genVar()$
                $gp \leftarrow$ GGPA($M$,$C_{exp}$,$?v_1$,$P$)
                $t \leftarrow (?v_0, p, ?v_1)$
            else if $rand(0, 1) > P[5]$ then
                $I_0 \leftarrow rand(I(C_{exp}))$
                $t \leftarrow (?v_0, p, I_0)$
            else
                $C_1 \leftarrow rand(R(C_{exp}))$
                if $not\ visited(C_1)$ then
                    $?v_1 \leftarrow genVar()$
                    $gp \leftarrow$ GGPN($M$,$C_1$,$?v_1$,$P$)
                else if $rand(0, 1) > P[6]$ then
                    $?v_1 \leftarrow genVar()$
                else
                    $?v_1 \leftarrow rand(V(C_1))$
                $t \leftarrow (?v_0, p, ?v_1)$
            $gp \leftarrow gp \cup \{t\}$
    // Process other types of $C_{anon}$
```

Algorithm 4: MERGE(GP,i,j,P)

```
Input: $GP$: graph patterns; $i$: start index; $j$: end index; $P$:
       probability thresholds
Output: $gp$: graph pattern
if $i = j$ then
    $gp \leftarrow GP[i]$
else
    $k \leftarrow rand(\{i, \ldots, j - 1\})$
    $gp_l \leftarrow$ MERGE($GP$,$i$,$k$,$P$)
    $gp_r \leftarrow$ MERGE($GP$,$k + 1$,$j$,$P$)
    $gp \leftarrow comb(gp_l, gp_r, P)$
```

inverse property of p, may be the ultimate triple pattern to be included. The details of these procedures are not shown in the algorithms.

Example 2: Figure 4 shows the progression of SQG through the algorithms. For simplicity, the figure only includes the fragments that are traversed by the algorithms for the query body generation that was started in Example 1. Step 1 shows the generation of a class assertion triple pattern (*?v1 rdf:type USRPB200*) generated in line 3 of Algorithm 2. Then, Algorithm 2 invokes Algorithm 3 to generate a nested graph pattern (step 2), followed by processing key-value pairs $O_{P,R}(C_0)$ to generate a nested graph pattern that includes a triple pattern (*?v1 supportsTransmitting I1*) (step 3). At last, Algorithm 4 is invoked to merge the two graph patterns with keyword OPTIONAL (step 4).

Algorithm 3 (GGPA(M,C_{anon},$?v_0$,P)) implements subprocedures of generation of graph patterns. Graph patterns are computed by traversing the model starting from an anonymous class expression passed as C_{anon}. The result depends on the type of anonymous class expression. The algorithm covers all types of OWL 2 anonymous class expressions described in [28]. Due to the space limitation, only the processing of ObjectIntersectionOf, ObjectUnionOf and cardinality-based object property restriction (ObjectSomeValuesFrom, ObjectAllValuesFrom, ObjectMinCardinality, ObjectMaxCardinality or ObjectExactCardinality) is shown. Specifically, if C_{anon} is a ObjectIntersectionOf or ObjectUnionOf, the algorithm first initializes graph patterns GP, looks for the class expressions they operate on and then either recursively calls the algorithm itself or calls back to Algorithm 2 (GGPN(M,C_0,$?v_0$,P)), depending on whether they are anonymous or not. Elements of GP are progressively generated by the function calls. After all the elements are built, they are merged by calling Algorithm 4 (MERGE(GP,0,$length(GP) - 1$,P); if C_{anon} is a ObjectComplementOf, the algorithm gets the class expression it operates on and then either recursively calls the algorithm itself or calls back to Algorithm 2 (GGPN(M,C_0,$?v_0$,P)), depending on whether the class expression is anonymous or not; if C_{anon} is a ObjectOneOf, the algorithm terminates; if C_{anon} is a ObjectHasValue or DataHasValue, the algorithm gets the filler (an OWL individual or an OWL literal) and then generates a property assertion triple pattern accordingly; if C_{anon} is a ObjectHasSelf, the algorithm generates an object property assertion triple pattern with the same subject and object; if C_{anon} is a cardinality-based object property restriction, the algorithm gets the property p and the filler as OWL class expression C_{exp} referenced by C_{anon}. If C_{exp} is anonymous, the algorithm generates a variable $?v_1$ and calls itself. Otherwise, it randomly picks an OWL named individual $I_0 \in I(C_{exp})$ or generates a variable $?v_1$ and calls back to Algorithm 2 (GGPN(M,C_{exp},$?v_1$,P)) or simply generates a variable $?v_1$ or randomly picks a variable $?v_1 \in V(C_1)$, depending on the probability thresholds and the visit status of C_1. In any case, an object property assertion triple pattern $(?v_0, p, ?v_1)$ or $(?v_0, p, I_0)$ is generated and added

Table 2 Basic characteristics of the ontologies used in experiments

Ontology	Class	Object property	Data property	Class axiom	Object property axiom	Data property axiom
eDIANA	70	6	12	174	13	36
IoT	100	79	8	161	65	14
SAREF	112	63	31	251	89	51
SDR	321	104	21	548	203	37
Univ-Bench	43	25	7	42	47	4
WM30	351	81	20	561	88	23

into gp in the end; if C_{anon} is a cardinality-based data property restriction (DataSomeValuesFrom, DataAllValuesFrom, DataMinCardinality, DataMaxCardinality or DataExactCardinality), the algorithm gets the property p and the filler as data range r referenced by C_{anon} and generates a variable $?v_1$. Then, a data property assertion triple pattern ($?v_0$, p, $?v_1$) is generated and added into gp. The algorithm may also randomly generate a filter expression with the function $getFilterExp(r, ?v_1)$ and add it into gp in the end.

Algorithm 4 (MERGE(GP,i,j,P)) merges graph patterns indexed by m in GP, where $i \le m \le j$. The algorithm is a divide and conquer algorithm. Lines 1-2 test for the base case, where GP has just one column. Lines 3-7 handle the recursive case. First, an index k, where $i \le k \le j - 1$, is randomly selected, and then, the problem is divided into two sub-problems. The algorithm recursively resolves each sub-problem and gets the results as gp_l and gp_r. At last, gp_l and gp_r are combined using a randomly selected operator from {AND, UNION, OPTIONAL, MINUS, FILTER NOT EXISTS, FILTER EXISTS} based on the probability thresholds P.

The run time complexity of the $comb()$ operation is constant ($O(1)$). So the run time complexity of the MERGE algorithm is $O(n)$, where $n = j - i + 1$ is the number of columns in GP.

6 Evaluation

In this section, we present an evaluation of SQG with respect to a number of metrics. Since one of our objectives was to show that SQG can be used on different ontologies, we reused the same set of ontologies for evaluating RODG [9], i.e., the SDR ontology [32] developed by us, and five existing ontologies in different domains: the eDIANA ontology [12], the IoT ontology [19,21], the Smart Appliances REFerence (SAREF) ontology [10,36], the Univ-Bench ontology [41] and the WM30 ontology [39]. Table 2 summarizes the basic characteristics of each ontology, including the number of classes, object properties, data properties, class axioms, object property axioms and data property axioms.

For quantitative evaluation, we selected metrics related to: (1) Scalability—how does the query generation time increase with the the number of queries and object descriptions? (2) Coverage of OWL axioms—what is the percentage of OWL axiom types included in the queries? This aspect is the most important one since it addresses the main objective of developing SQG. (3) Coverage of SPARQL—what is the coverage of the SPARQL language in the generated queries?

Experimental Setup: All the experiments were run on the MacBook Pro 2016 computer. The system specification is given below:

- **Processor**: 2.6 GHz quad-core Intel Core i7, Turbo Boost up to 3.5 GHz, with 6MB shared L3 cache
- **Storage**: 256 GB PCIe-based onboard SSD
- **Memory**: 16 GB of 2133 MHz LPDDR3 onboard memory

6.1 Scalability Evaluation

This section presents the assessment of how the number of queries and the size of the datasets affect the performance. Though there are various existing approaches to generate SPARQL queries, as summarized in Sect. 2, to the best of our knowledge, none of the existing generators can be used for generating large scales of SPARQL queries against object descriptions with good coverage of SPARQL characteristics. Therefore, it is not really possible to make any reasonable comparisons of SQG with existing generators. In the experiments, all test cases are executed with the same probability thresholds. To make the evaluation results more intuitive, the query generation time does not contain ontology loading time, parsing time and extracting time since they are fixed nonrecurring expenses for the whole generation process regardless of the number of queries.

SQG underwent a comprehensive scalability/performance evaluation with exponentially increasing number of queries (20, 200, 2000, 20,000, 200,000, 2,000,000) against 3000 object descriptions for each of the six ontologies. The evaluation results are summarized in Fig. 5.

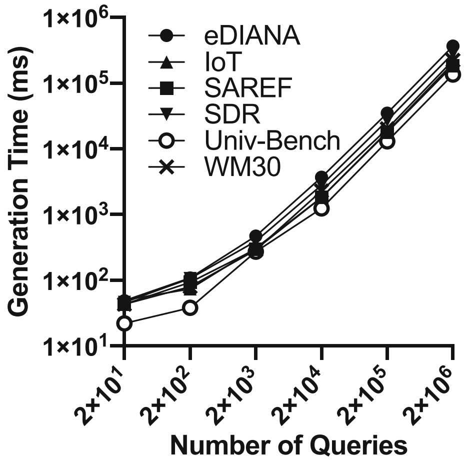


Fig. 5 Query generation time for the six ontologies with 3000 object descriptions

The testing results indicate relatively good performance and scalability of SQG. In general, the generation time grows linearly with the number of queries. The generation time for 20 queries in the worst case is 50 ms for IoT, whereas it does not exceed 7 mins (366,013 ms for eDIANA) for 2,000,000 queries.

To investigate the effects of the size of the datasets in the performance, we conducted another experiment to evaluate query generation time of 200,000 queries against increasing number of object descriptions (500, 1000, 1500, 2000, 2500, 3000) for each of the six ontologies. The evaluation results are shown in Fig. 6.

The testing results indicate that the size of the datasets does not affect the performance much. In general, the query generation time increases slightly with the number of object descriptions for eDIANA, SDR and Univ-Bench ontology and remains the same for the rest ontologies. According to the algorithms, the size of datasets may affect the performance of procedure $rand(I(C_i))$, i.e., randomly select an OWL named individual from OWL named individuals of type C_i for triple pattern generation. However, such processes are rarely executed by the algorithm. Since OWL individuals are distributed among the OWL classes, the number of individuals per class is relatively small.

6.2 Coverage of OWL Axioms

In this section, we present a comprehensive evaluation of the queries generated by SQG with respect to their coverage of the axioms of OWL. To achieve this, we focused on the coverage of the signatures of the ontologies. In our experiments, the metrics for signature coverage were collected for sets of queries of varying size (20, 200, 2,000, 20,000, 200,000, 2,000,000) against 3,000 object descriptions for each of the

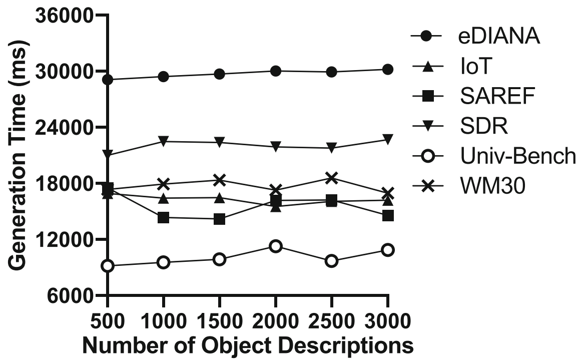


Fig. 6 Query generation time of 200,000 queries for the six ontologies with different numbers of object descriptions

Table 3 Summary of the focus signatures [9]

Ontology	$\|C\|$/total	$\|OP\|$/total	$\|DP\|$/total
eDIANA	70/70	6/6	12/12
IoT	100/100	71/79	8/8
SAREF	112/112	63/63	30/31
SDR	321/321	102/104	16/21
Univ-Bench	42/43	21/25	0/7
WM30	347/351	68/81	18/20

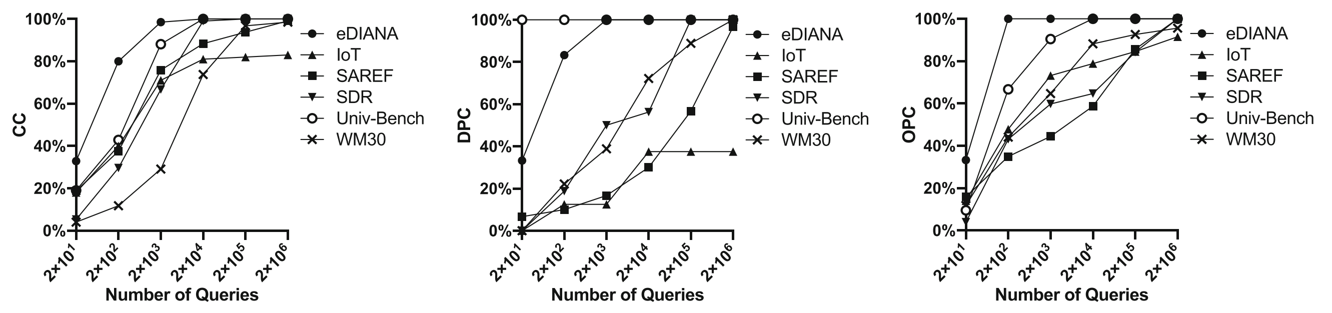
six ontologies. All the queries were generated with the same probability thresholds.

Given an ontology, SQG generates SPARQL queries for finding objects of specific types. In most cases, the ontologies were not designed just for this particular application scenario, and thus, some of the concepts in the signature of the ontology are not relevant to these kinds of query. Therefore, it is necessary to narrow down the scope of the signatures and focus only on the signatures, termed *focus signatures* [9], whose concepts are expected to be used for the generation of the queries. The number of focus signatures (class signatures $|C|$, object properties $|OP|$ and data properties $|DP|$) versus the total number of signatures for each of the six ontologies is listed in Table 3. Below we introduce the formalization of the three signature coverage metrics used. Then, we show the results and analysis.

Class Coverage (CC): It is defined as the ratio of the number of the focus classes whose individuals are bound by at least one variable in a query q to the total number of the focus classes.

$$CC = \frac{|\{C_i | \exists(s, rdf{:}type, C_i) \in q, s \in V\}|}{|C|} \tag{1}$$

Datatype Property Coverage (DPC): It is defined as the ratio of the number of the focus datatype properties that are placed at least once as predicate in a query q to the total



(a) CC **(b)** DPC **(c)** OPC

Fig. 7 Axiom coverage evaluation results

number of the focus data properties.

$$DPC = \frac{|\{DP_i | \exists (s, DP_i, o) \in q\}|}{|DP|} \tag{2}$$

Object Property Coverage (OPC): It is defined as the ratio of the number of the focus object properties that are placed at least once as predicate in a query q to the total number of the focus object properties.

$$OPC = \frac{|\{OP_i | \exists (s, OP_i, o) \in q\}|}{|OP|} \tag{3}$$

In our experiments, the three metrics were collected for the generated queries of varying size (20, 200, 2000, 20,000, 200,000, 2,000,000) for each of the six ontologies. The evaluation results are shown in Fig. 7. It can be observed that the metrics tend to increase with the number of queries. Assume the threshold of the metrics is set as 80%, for focus class/object property signatures, at least 200,000 queries are needed to cover all ontologies; for focus data properties, 2,000,000 queries are needed to cover all but IoT.

It can also be observed that the results are different among the ontologies. The main reasons are the structural differences of the ontologies and selection of the root class. If a focus signature is far away from the root class, it is less likely to be traversed by the algorithms for query generation. Additionally, based on the algorithms, in some cases, a focus signature may not be used for query generation by the algorithms. Hence, it is likely that the signature coverage never reaches 100%.

It is worth mentioning that other factors may also affect the metrics results, such as probability thresholds. Details of such discussion are not shown due to the space limitation.

6.3 Coverage of SPARQL Language

In order to guarantee a good coverage of OWL axioms, SQG needs to provide, and thus be evaluated on, the coverage of the constructs of SPARQL. In this section, we focus on the evaluation of the coverage of the types of SPARQL expressions. This is different from other approaches known in the literature in which the objective is to generate "difficult" loads of SPARQL queries. Our intent is to use SQG as an additional benchmark for SPARQL query generation when OWL inference is important and not as a replacement for other kinds of testing.

6.3.1 Keyword Coverage

The objective of the keyword coverage evaluation is to assess the coverage of the SPARQL keywords in the generated queries. Given a set of queries, the coverage of a keyword is defined as the ratio of the queries that use the keyword at least once over the total number of the queries. In our experiments, we took into account keywords associated with SPARQL algebra operators that occur in the bodies of the 2,000,000 queries for each of the six ontologies.

As shown in Table 4, SQG has a good coverage of the keywords. Among the SPARQL algebra keywords, AND has been the most commonly occurring operator. UNION and OPTIONAL have been frequently used, too. FILTER is covered by all ontologies except for Univ-Bench and IoT. The reasons why Univ-Bench and IoT do not contain FILTER keyword can be explained as follows. The FILTER keyword is used within a filter expression, which is generated by Algorithm 2 function $genFilterExp(r, ?v_1)$ which takes a data property range r as an input. Univ-Bench does not have any focus data properties, so no filter expressions can be generated. The data property ranges xsd:anyURI and xsd:string of IoT are not supported in SQG.

Although queries are generated with the same probability thresholds for each ontology, keyword coverage varies. This is due to the structural differences of the ontologies. If an ontology on average has fewer types of class restrictions on a focus class, according to Algorithm 2, it is more likely to generate SPARQL queries that do not contain any keywords. The proportion of such queries (see Table 5) affects the rela-

Table 4 Percentage of queries using different keywords at least once

Keyword (%)	eDIANA (%)	IoT (%)	SAREF (%)	SDR (%)	Univ-Bench (%)	WM30 (%)
FILTER	8.81	0.00	0.27	2.02	0.00	0.03
AND	51.45	32.61	40.27	40.62	21.42	37.29
UNION	2.15	0.88	0.98	2.67	0.16	1.03
OPTIONAL	6.58	4.16	5.38	6.89	1.24	5.02
MINUS	2.27	1.41	1.82	2.45	0.41	1.70
NOT EXISTS	2.26	1.40	1.84	2.44	0.42	1.70
EXISTS	2.27	1.42	1.81	2.48	0.42	1.69

Table 5 Percentage of queries with different operator sets

Operator set (%)	eDIANA (%)	IoT (%)	SAREF (%)	SDR (%)	Univ-Bench (%)	WM30 (%)
None	47.57	66.18	57.35	57.20	78.58	60.92
A	36.22	28.91	36.22	33.41	20.05	33.19
F	0.00	0.00	0.00	0.00	0.00	0.00
F, A	7.85	0.00	0.20	0.73	0.00	0.01
CPF subtotal	91.64	95.09	93.77	91.35	98.62	94.12
U	0.00	0.00	0.00	0.00	0.00	0.00
O	0.99	1.21	2.38	2.17	0.00	1.79
F, U	0.00	0.00	0.00	0.00	0.00	0.00
F, O	0.00	0.00	0.00	0.00	0.00	0.00
A, U	1.58	0.75	0.82	1.34	0.14	0.85
A, O	4.53	2.82	2.82	3.27	1.22	3.05
U, O	0.00	0.00	0.00	0.00	0.00	0.00
F, A, U	0.20	0.00	0.02	0.42	0.00	0.01
F, U, O	0.00	0.00	0.00	0.00	0.00	0.00
F, A, O	0.70	0.00	0.05	0.54	0.00	0.01
A, U, O	0.32	0.13	0.14	0.59	0.02	0.16
F, A, U, O	0.04	0.00	0.00	0.33	0.00	0.00
Non-CPF subtotal	8.36	4.91	6.23	8.65	1.38	5.88

tive proportion of the queries with different combinations of the keywords.

6.3.2 Operator Distribution

The objective of operator distribution evaluation is to investigate how SPARQL operators are distributed in the queries. We can see in Table 4, operators $O = \{$FILTER, AND, UNION, OPTIONAL$\}$ are the most commonly used operators in the query bodies. Therefore, we investigated distribution of the queries whose bodies use constructs with a specific combination of these operators. Additionally, conjunctive patterns with filters (CPF) [8] are considered to be an important fragment of SPARQL patterns, because they are believed to appear often in practice [26,42]. The analysis of such queries is also included since the queries generated by SQG cover this pattern.

Definition 6 [*Conjunctive pattern with filters*] A conjunctive pattern with filters (CPF) is a graph pattern that only uses triples and the operators AND and FILTER.

Table 5 shows the proportion of the queries categorized by the combinations of the operators. (F, A, U, O are short for FILTER, AND, UNION and OPTIONAL, respectively.) The results demonstrate that most of the queries contain CPF patterns. Percentage of the queries with CPF patterns for eDIANA, IoT, SAREF, SDR, Univ-Bench and WM30 ontology are 91.64%, 95.09%, 93.77%, 91.35%, 98.62% and 94.12%, respectively. It can also be seen that the queries have a relatively good coverage of various combinations of the operators.

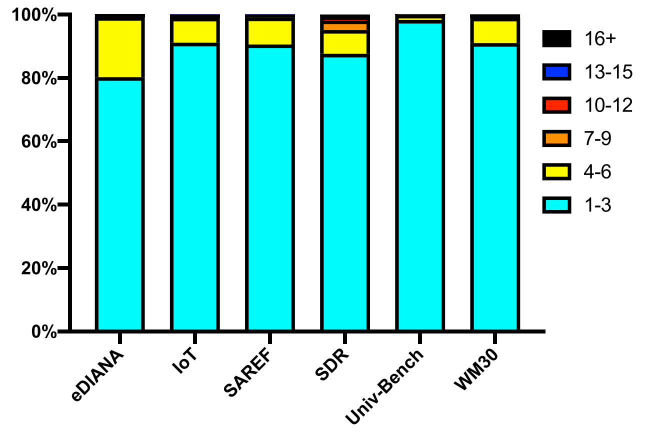


Fig. 8 Percentage of queries of different sizes

6.3.3 Number of Triple Patterns

The sizes of the queries are the counts of the number of triple patterns contained in each query. Figure 8 illustrates the results of the percentage of the 2,000,000 queries containing, respectively, triple patterns in the range of 1–3, 4–6, 7–9, 10–13, 14–16 and 16 or above for each of the six ontologies. A first observation is that the short queries (from 1 to 3) are the most frequent (80.20%, 91.09%, 90.52%, 87.61%, 98.26% and 90.98% for eDIANA, IoT, SAREF, SDR, Univ-Bench and WM30, respectively). Second, the proportion of the larger queries (with the number of triple patterns per query of 10 or above) is very small. The results are as expected since they are similar to the real-world SPARQL query sets, including the ones extracted from logs of the DBPedia and SWDF public endpoints [3,8].

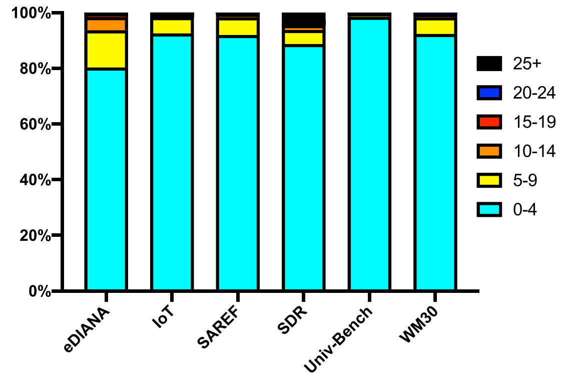
6.3.4 Structural Analysis

We have also performed a higher level analysis of the structure of the query expressions. In particular, we focused on the number of join operations [3] appearing in each query and the distribution of their types. According to Definition 2, a simple triple pattern consists of a triple where any of the subject, predicate or object may be bound to a variable. They can be combined into more complex patterns using join operations, which in turn leads to six types of joins depending on which positions the common variable appears in each pattern: Subject–Subject (SS), Predicate–Predicate (PP), Object–Object (OO), Subject–Predicate (SP), Subject–Object (SO) and Predicate–Object (PO) [3].

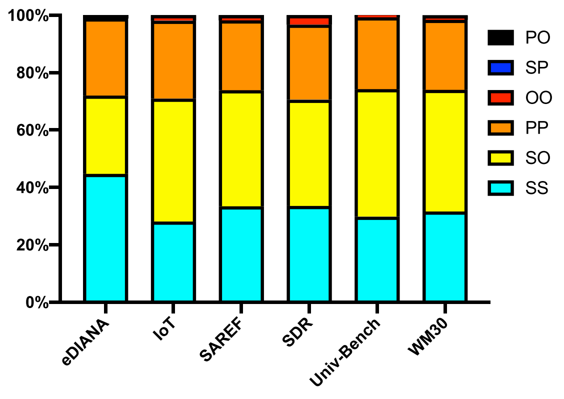
Definition 7 [*Join operation*] A join operation is a conjunction of two triple patterns, where both have at least one variable in common.

Figure 9a shows the distribution of the 2,000,000 queries with join patterns in the ranges of 0–4, 5–9, 10–14, 15–19, 20–24 and 25 or above for each of the six ontologies. Similar as the distribution of the number of triple patterns, the queries with a low number of joins (from 0 to 4) are most frequent (80.30%, 92.56%, 92.00%, 88.75%, 98.53% and 92.37% for eDIANA, IoT, SAREF, SDR, Univ-Bench and WM30, respectively). The proportion of the queries with higher numbers of joins is essentially negligible, in particular when the number of joins per query is 15 or above.

The distribution of the six join types of the queries is shown in Fig. 9b. It can be observed from the results that SS, SO and PP are the most commonly join types in the queries. The OO join type is relatively rare. The PP join type occurs mostly in the cases when two triple patterns share the common resource rdf:type as the predicate. SP and PO do not occur in any of the test cases, but they can be easily supported by extending the query generation algorithms if needed.



(a) Percentage of queries with different numbers of joins



(b) Distribution of join types

Fig. 9 Query structure evaluation results

7 Conclusion

This paper describes SQG—a generic SPARQL query generator, which is able to generate relatively large numbers of random SPARQL queries for retrieving descriptions of objects of a specific type from RDF/OWL datasets. The intent behind the development of SQG was to provide a benchmark for testing systems that rely on the inference based on the OWL semantics. Apart from applying SQG to our specific use case of generating requests for RF devices in SPARQL, we believe that it may be applicable to the application scenarios where large synthetic SPARQL queries with similar query patterns and characteristics are needed for testing systems that implement Semantic Web solutions. SQG and the generated benchmark query sets were evaluated on six ontologies with benchmark datasets in terms of the scalability/performance and the coverage of the OWL axioms and query characteristics. The evaluation results demonstrate that SQG is generic, scalable, and the generated queries are of high diversity—covering both the axioms of OWL and features of the SPaRQL language.

To the best of our knowledge, SQG is the first SPARQL query generator that is able to automatically generate random SPARQL queries for requesting descriptions of matching objects while taking into account the OWL semantics in the query formulations. However, we are not claiming that SQG supersedes other SPARQL query generation benchmarks that focus on generating loads that are known to be difficult to handle by the query engines. We submit that SQG provides features that are complementary to such benchmarks.

Acknowledgements This research was developed with partial funding from the Defense Advanced Research Projects Agency (DARPA). The views, opinions and/or findings expressed are those of the authors and should not be interpreted as representing the official views or policies of the Department of Defense or the US Government.

Declarations

Conflict of interest The authors declare that they have no conflict of interest.

References

1. Aluç G, Hartig O, Özsu MT, Daudjee K (2014) Diversified stress testing of RDF data management systems. In: ISWC. Springer, Riva del Garda, pp 197–212
2. Arenas M, Gutierrez C, Pérez J (2010) On the semantics of SPARQL. In: Semantic web information management. Springer, pp 281–307
3. Arias M, Fernández JD, Martínez-Prieto MA, de la Fuente P (2011) An empirical study of real-world SPARQL queries. In: USEWOD. Hyderabad
4. ARQ-A SPARQL Processor for Jena (2011). http://jena.apache.org/documentation/query/index.html. Accessed: 22 Feb 2020
5. Atzori M, Mazzeo GM, Zaniolo C (2019) QA3: a natural language approach to question answering over RDF data cubes. Semant Web 10(3):587–604
6. Bagan G, Bonifati A, Ciucanu R, Fletcher GH, Lemay A, Advokaat N (2017) gMark: schema-driven generation of graphs and queries. IEEE Trans Knowl Data Eng 29(4):856–869
7. Bizer C, Schultz A (2008) Benchmarking the performance of storage systems that expose SPARQL endpoints. In: Proceedings of the 4th international workshop on scalable semantic web knowledge base systems (SSWS), Karlsruhe, Germany
8. Bonifati A, Martens W, Timm T (2017) An analytical study of large SPARQL query logs. Proc VLDB Endow 11(2):149–161
9. Chen Y, Kokar MM, Moskal JJ (2020) RDF object description generator (RODG). Int J Web Eng Technol 15(2):140–169
10. Daniele L, den Hartog F, Roes J (2015) Created in close interaction with the industry: the smart appliances reference (SAREF) ontology. In: International workshop formal ontologies meet industries. Springer, Berlin, pp 100–112
11. Dibowski H, Kabitzsch K (2011) Ontology-based device descriptions and device repository for building automation devices. EURASIP J Embed Syst 2011(1):1–17
12. eDIANA Ontology. https://www.smartappliancesproject/ontologies/ediana.owl. Accessed: 22, Feb 2020
13. Erling O, Mikhailov I (2009) RDF support in the virtuoso DBMS. In: Networked knowledge-networked media. Springer, pp 7–24
14. Görlitz O, Thimm M, Staab S (2012) SPLODGE: systematic generation of SPARQL benchmark queries for linked open data. In: ISWC. Springer, Boston, pp 116–132
15. GraphDB. https://www.ontotext.com/products/graphdb/. Accessed: 30 Mar 2021
16. Guo Y, Pan Z, Heflin J (2005) LUBM: a benchmark for OWL knowledge base systems. J Web Seman 3(2):158–182
17. Haase P, Mathäß T, Ziller M (2010) An evaluation of approaches to federated query processing over linked data. In: Proceedings of the 6th international conference on semantic systems. ACM, Graz
18. Horridge M, Bechhofer S (2011) The OWL API: a Java API for OWL ontologies. Seman Web 2(1):11–21
19. IoT Ontology (2010). http://i-lab.aegean.gr/kotis/Ontologies/IoT/IoT-ontology-v2.1.owl. Accessed: 22 Feb 2020
20. Kontokostas D, Westphal P, Auer S, Hellmann S, Lehmann J, Cornelissen R, Zaveri A (2014) Test-driven evaluation of linked data quality. In: Proceedings of the 23rd international conference on World Wide Web. ACM, Seoul, pp 747–758
21. Kotis K, Katasonov A (2012) An IoT-ontology for the representation of interconnected. Clustered and Aligned Smart Entities, Semantic Web
22. Kotsev V, Minadakis N, Papakonstantinou V, Erling O, Fundulaki I, Kiryakov A (2016) Benchmarking RDF query engines: the LDBC semantic publishing benchmark. In: Proceedings of the workshop on benchmarking linked data (BLINK 2016), Kobe

23. Ma L, Yang Y, Qiu Z, Xie G, Pan Y, Liu S (2006) Towards a complete OWL ontology benchmark. In: The semantic web: research and applications. Springer, Berlin, Heidelberg, pp 125–139
24. Meimaris M, Papastefanatos G (2016) The EvoGen benchmark suite for evolving RDF data. In: 2nd Workshop on managing the evolution and preservation of the data web (MEPDaW 2016). Heraklion, Crete, pp 20–35
25. Morsey M, Lehmann J, Auer S, Ngomo ACN (2011) DBpedia SPARQL benchmark—performance assessment with real queries on real data. In: ISWC. Springer, Bonn, pp 454–469
26. Neumann T, Weikum G (2010) The RDF-3X engine for scalable management of RDF data. VLDB J 19(1):91–113
27. Owens A, Gibbins N, Schraefel M (2008) Effective benchmarking for RDF stores using synthetic data. In: ISWC. Washington, DC, pp 94–109
28. OWL 2 (2012) Web Ontology Language Structural Specification and Functional-Style Syntax, 2nd edn. https://www.w3.org/TR/owl2-syntax/. Accessed: 22 Feb 2020
29. Qiao S, Özsoyoğlu ZM (2015) RBench: application-specific RDF benchmarking. In: SIGMOD. ACM, Melbourne, pp 1825–1838
30. Saleem M, Mehmood Q, Ngomo ACN (2015) Feasible: a feature-based SPARQL benchmark generation framework. In: ISWC. Springer, Bethlehem, pp 52–69
31. Schmidt M, Hornung T, Lausen G, Pinkel C (2009) SP2Bench: a SPARQL performance benchmark. In: International conference on data engineering. IEEE, Shanghai, pp 222–233
32. SDR Ontology (2018). https://SDROntology/SDR.owl. Accessed: 22 Feb 2020
33. Shekarpour S, Marx E, Ngomo ACN, Auer S (2015) Sina: semantic interpretation of user queries for question answering on interlinked data. J Web Seman 30:39–51
34. Shekarpour S, Auer S, Ngomo ACN, Gerber D, Hellmann S, Stadler C (2011 Keyword-driven SPARQL query generation leveraging background knowledge. In: International conferences on web intelligence. IEEE, Lyon, pp 203–210
35. Skiena SS (2008) The algorithm design manual, 2nd edn. Springer, Berlin, Heidelberg
36. Smart Appliances REFerence Ontology. http://ontology.tno.nl/saref/ (2015). Accessed: 22 Feb 2020
37. SPARQL 1.1 (2013) Query Language. https://www.w3.org/TR/sparql11-query/. Accessed: 22 Feb 2020
38. SPARQL Query Generator (2019). https://github.com/YankeeChen/sparqlquerygenerator/. Accessed: 10 Mar 2020
39. SSN Ontology Example: Wind sensor (2010). http://purl.oclc.org/NET/ssnx/meteo/WM30. Accessed: 22 Feb 2020
40. Unger C, Bühmann L, Lehmann J, Ngonga Ngomo AC, Gerber D, Cimiano P (2012) Template-based question answering over RDF data. In: Proceedings of the 21st international conference on World Wide Web. ACM, Lyon, pp 639–648
41. Univ-Bench Ontology. http://swat.cse.lehigh.edu/onto/univ-bench.owl (2004). Accessed: 22 Feb 2020
42. Vidal ME, Ruckhaus E, Lampo T, Martínez A, Sierra J, Polleres A (2010) Efficiently joining group patterns in SPARQL queries. In: Extended semantic web conference (ESWC). Springer, Heraklion, pp 228–242
43. Wang SY, Guo Y, Qasem A, Heflin J (2005) Rapid benchmarking for semantic web knowledge base systems. In: ISWC. Galway, pp 758–772
44. Zenz G, Zhou X, Minack E, Siberski W, Nejdl W (2009) From keywords to semantic queries-incremental query construction on the semantic web. J Web Seman 7(3):166–176