An Ontology-Driven Approach for the Syntactic and Semantic Validation of SPARQL Queries

Juan Collio*
Escuela de Ingeniería Informática
Universidad de Valparaíso
Valparaíso 2340000, Chile
juan.colliob@postgrado.uv.cl

and

Ana Aguilera*
Escuela de Ingeniería Informática
MEDING Interdisciplinary Center
Universidad de Valparaíso
Valparaíso 2340000, Chile
ana.aguilera@uv.cl

and

Irvin Dongo
Electrical and Electronics Engineering Department
Universidad Católica San Pablo
Arequipa 04001, Peru
Univ. Bordeaux, ESTIA INSTITUTE OF TECHNOLOGY
Bidart 64100, France
ifdongo@ucsp.edu.pe

Abstract

The Semantic Web enables data exchange and knowledge exploitation through SPARQL queries; however, users often encounter difficulties when formulating such queries, leading to unexpected results and iterative trial-and-error processes. These issues are aggravated by incomplete knowledge of the application domain and the underlying ontological schema, frequently causing unnecessary query executions and increased computational load on RDF repositories. This work proposes a method for the prior syntactic and semantic validation of SPARQL queries, based on an ontology and independent of its instances. The method operates before query execution and detects inconsistencies arising from literal writing errors and violations of object property axioms, providing explicit feedback to users. The development of the proposed approach follows a classical methodological framework grounded in the software engineering process. Experimental results show that the additional computational overhead introduced by validation is minimal, while the benefits are significant in scenarios involving incorrect queries. In particular, prior validation prevents the execution of erroneous queries, reduces processing time, and maintains negligible overhead for correct queries, demonstrating a favorable trade-off between computational efficiency and reduced manual intervention. Overall, the results indicate that prior validation improves SPARQL query quality and enhances developer productivity in semantic data–driven applications.

*Corresponding author. This study corresponds to an extended version of the work published in [1]. The state of the art was expanded (Sec. 2), the proposed approach was presented from a methodological perspective evidenced in the research pathway (Sec. 3), new results were incorporated with their corresponding interpretation, and a discussion section was added (Sec. 5)

Keywords: SPARQL, SHACL, Semantic Web, OWL Ontology, Syntactic and Semantic Validation.

1 Introduction

The Semantic Web envisions an evolution of the traditional Web toward a network focused on the exchange and interpretation of data rather than documents. Its main objective is to enable computer systems to process and interpret data in a more meaningful way, thereby supporting more intelligent applications and services [2]. As a consequence of this paradigm, the amount of data represented using the Resource Description Framework (RDF) has increased substantially over time [3]. This rapid growth of RDF data has highlighted the need for appropriate mechanisms to store, access, and query such information efficiently. In response to this need, the World Wide Web Consortium (W3C) standardized SPARQL as the query language for RDF data [4, 5]. SPARQL provides a flexible way to retrieve and manipulate RDF data, particularly through selection queries that allow users to extract relevant information from large datasets [6]. At the same time, knowledge representation in the Semantic Web is grounded in the use of ontologies [7, 8], which define shared conceptualizations of a domain and are commonly expressed using the Web Ontology Language (OWL) [9]. Since OWL is built on top of RDF [10], both languages together constitute the foundational technologies that support the development and consolidation of the Semantic Web [11, 12].

SPARQL queries are usually syntactically simple and easy to learn; however, programmers often encounter situations where the query does not return the expected result [4]. This phenomenon is not only related to typing errors, but also to an insufficient understanding of the RDF graph and domain semantics, leading to empty, incomplete, or unexpected responses [13]. The correct formulation of a query requires knowledge of the ontological schema and the domain that structures the data [6], but in practice, this information is often partially unknown to application developers. As a result, queries are constructed through an iterative process of trial and error, which increases the complexity of development. This approach, in addition to generating a slow and unsystematic development process, produces an unnecessary computational load on RDF engines, since incorrect queries still require runtime for parsing the query, consuming server resources [14]. The repeated execution of erroneous queries affects system performance and, in web environments, impacts parameters such as response time, CPU utilization, memory consumption, and the rate of failed requests [15]. In short, a lack of knowledge of the schema and domain exacerbates the difficulty of formulating correct SPARQL queries, increases the number of unnecessary executions, and impairs the efficiency of semantic repository exploitation. These difficulties highlight the need for prior verification processes to identify inconsistencies in SPARQL queries.

In response to the problem identified, this work proposes a method for validating SPARQL queries prior to their execution in an RDF engine or SPARQL processor. The method is ontology-based and operates independently of ontology instances, with the objective of detecting inconsistencies in a query before it is processed. By doing so, the approach reduces the traditional trial-and-error cycle associated with query formulation and avoids unnecessary executions that may impose additional computational overhead on the system. The proposed method comprises two complementary levels of validation. The first level focuses on syntactic validation, aiming to detect writing errors related to literals. At this stage, the literals appearing in the query are analyzed by evaluating their conformity with the data types defined in the ontology, considering both their lexical form and associated datatype. The second level addresses semantic validation, whose goal is to identify inconsistencies arising from an incomplete or incorrect understanding of the ontological domain. This stage concentrates on object property axioms, which define the semantic constraints and relationships within the domain and enable the assessment of whether a query is consistent with the ontological structure. When a query fails to satisfy either the syntactic or semantic conditions, the method reports the detected errors, thereby contributing to the improvement of SPARQL query quality prior to execution.

For the development of the proposed method, this article follows a classical methodological approach grounded in the software engineering process described by Pressman [16]. Accordingly, the content is structured around the five fundamental software process activities—communication, planning, modeling, construction, and deployment—which enables a systematic presentation of the research process and facilitates comprehension of the proposed solution. In this context, the main contribution of this work lies in the definition of a five-phase method for the syntactic and semantic validation of SPARQL queries, together with a detailed description of the seven algorithms associated with these phases.

The rest of the article is organized as follows. Section 2 presents a review of the literature related to query validation. Section 3 describes the proposed method for syntactic and semantic validation. Section 4 presents the results and experimental evaluation. Section 5 presents a discussion on the importance of query formulation. Finally, Section 6 presents the conclusions of the work and future research directions.

2 Related work

In the context of SPARQL query engines, optimization refers to the process of identifying an efficient execution plan for a query, given that a single query pattern may admit multiple plans with significantly different computational costs despite producing the same results [17]. In contrast, validation focuses on the form of the query, understood as the process of analyzing how the query is formulated and verifying its syntactic correctness and semantic coherence, whether with respect to the RDF model, the underlying ontology, or the structuring of the query's triple patterns. Whereas optimization operates on valid queries to reduce execution time or improve the performance of the query engine, validation intervenes beforehand to identify structural or semantic errors that may lead to incorrect, empty, or inconsistent results, or require the relocation or modification of triple patterns.

Based on this distinction, related work can be categorized into two main lines: approaches aimed at validating SPARQL queries and those focused on query optimization. In addition, there are validation approaches based on *SHACL* (Shapes Constraint Language) for detecting inconsistencies in RDF graphs; however, these mechanisms operate at the data level, validating RDF instances against predefined constraints, rather than addressing the structural or semantic correctness of the query itself.

Guzel Kalayci et al. [17] propose an optimization methodology for SPARQL queries based on the Ant Colony Optimization (ACO) metaheuristic, whose goal is to improve the ordering of triple patterns in the Basic Graph Pattern in order to reduce query execution time. The approach considers different structural forms of queries (e.g., chain, star, cyclic, and chain–star) and demonstrates significant performance improvements compared to native processing in Jena ARQ and other existing reordering algorithms.

The work of Liu et al. [18] proposes an RDF query engine oriented toward SPARQL optimization through the use of inverted indexes and RDF statistics to estimate costs and select efficient execution plans. Their approach introduces a query graph model and a set of specialized operators that enable the transformation of SPARQL queries into optimal execution trees, drastically reducing the search space of join ordering.

The work of Saharan et al. [19] addresses SPARQL query optimization through the reordering of Basic Graph Pattern (BGP) triples, proposing a solution based on the Teaching Learning Based Optimization (TLBO) metaheuristic.

Almendros-Jiménez *et al.* [20, 21] present a tool for syntactic and semantic validation of SPARQL queries, based on an input ontology. The syntactic validation, performed with the OWL API, identifies errors such as incorrect use of vocabulary, resources, literals and filter conditions, especially in relation to domain and range. Although literals are validated, these are limited to integer and real number data types and certain facets, leaving out other data types [22] and more complete validations, such as *Lexical Space, Value Space and Facet Space*, which are proposed as an improvement. Semantic validation employs a reasoner (HermiT/Pellet) to verify the consistency of an ontology generated from the query pattern. However, reasoners have limitations and many have become obsolete [23, 24].

The authors Gashkov et al. [25] present a candidate SPARQL query validator, which distinguishes queries that return correct answers from those that would yield incorrect answers. Given a natural language NL query, Knowledge Graph Question Answering systems generate a list of candidate SPARQL queries. The main goal being to eliminate incorrect queries through comparison between the user's NL query and the SPARQL query verbalization. The validator determines if the SPARQL query correctly represents the original question.

Authors Ozger and Uslu [26] propose the discrete Artificial Bee Colony (dABCSPARQL) algorithm to reorder RDF triples of SPARQL queries using a heuristic approach. In this study, the ABC algorithm is applied to optimize SPARQL queries in different triple and graph sizes. It identifies the best options for query patterns and to determine long queries before executing them.

Guido [27] proposes a static analysis approach for SPARQL queries centered on two fundamental problems: query containment and query–update independence. Their work reformulates both problems through formal techniques based on modal logic, enabling the determination of structural relationships between queries without requiring execution over RDF datasets.

On the other hand, *SHACL* is a relatively new language [28], and studies have already explored SPARQL query validation using *SHACL*. Thapa and Giese [29] propose an optimization technique for SPARQL queries that preserves their results while improving efficiency. These optimizations are applied prior to query execution, provided that the RDF graph complies with certain *SHACL* shapes. The approach focuses on removing or restructuring redundant graph patterns in the query, guided by the constraints defined in *SHACL*. Corman *et al.* [30] present a tool called SHACL2SPARQL, which validates an RDF graph against recursive *SHACL* constraints and generates optimized SPARQL queries.

Oudshoorn et al. [31] propose a validation scheme for RDF data that integrates OWL ontologies into the *SHACL* validation process, addressing the semantic discrepancy between OWL's open-world assumption and SHACL's closed-world assumption. To this end, they introduce the notion of an austere canonical model

that incorporates minimal ontological inferences and enables the rewriting of *SHACL* constraints so that they can be executed by standard validators, thereby supporting more expressive validation in the presence of inferred knowledge.

In our work, we assume that the SPARQL syntax is correct, that is, that it does not present grammatical errors in keywords, delimiters, or operators. This level of basic syntactic validation is the responsibility of the SPARQL engine and lies outside the scope of this research.

Under this assumption, a query submitted to the RDF engine must necessarily undergo the parsing process, regardless of whether it produces results or not; consequently, optimization approaches do not avoid this computational load, as their focus lies on computational cost. In cases where a query is incorrectly formulated with respect to the domain or underlying ontological schema, these optimization approaches do not report the origin of the error. Meanwhile, validation methods may reorganize or modify triple patterns with the aim of improving query efficiency; however, if inconsistencies exist with respect to the ontological schema, the query will still be processed by the RDF engine and may return empty results without indicating the cause of the problem. In general terms, once a query has been validated, it is executed and subsequently undergoes the optimization process. Additionally, with the exception of the study by Almendros-Jiménez et al. [20], the other approaches reviewed in this section do not explicitly report ontology-related errors or inconsistencies. In contrast, the method proposed in this work provides an explicit list of detected semantic errors and inconsistencies, offering more informative feedback to the user.

Likewise, *SHACL* is a relatively recent language, standardized by the W3C in 2017 [28], and several studies have since explored its integration with SPARQL. However, validating SPARQL queries by means of *SHACL* typically relies on its shape-constraint language to ensure data integrity within RDF graphs [32], and therefore focuses primarily on data-level validation. In contrast, the approach adopted in this work employs *SHACL* to validate an RDF graph that is not derived from instance data, but is instead constructed from the structural patterns of the SPARQL query itself. Consequently, the validated graph does not correspond to a data graph in the strict sense, but rather to a query-induced representation, enabling the detection of inconsistencies prior to query execution.

In summary, the state of the art demonstrates relevant advances in query optimization and RDF data validation; however, a gap persists with respect to the syntactic and semantic validation of SPARQL queries based on the ontological schema, particularly regarding the early identification of errors prior to query execution. Moreover, existing approaches generally lack mechanisms for providing explicit and detailed feedback on the nature and origin of detected inconsistencies. This gap motivates the approach proposed in this work, which not only performs prior validation without executing the query, but also delivers a detailed report of each identified error, thereby supporting more informed query correction and improving the overall query formulation process.

3 SQV: Proposed Method for Syntactic and Semantic SPARQL Validation

This section provides an overview of the proposed SQV (SPARQL Query Validation) method for the syntactic and semantic validation of SPARQL queries. To facilitate the reader's understanding, the proposal is presented from a methodological perspective inspired by the classical software engineering process described by Pressman [16], adapted to the specific context of this research. The process is structured into five activities (Figure 1): (1) communication, (2) planning, (3) modeling, (4) construction, and (5) deployment. These activities provide a systematic framework for describing the algorithmic development cycle, from problem formulation to the experimental evaluation of the proposed method. Each activity explains a specific stage of the method's development, together constituting a comprehensive description of the overall process. Finally, this work represents the first increment of a broader research project, whose long-term objective is to integrate query validation mechanisms into a SPARQL endpoint capable of verifying queries before their execution.

The definitions and terminology used throughout this section follow those established by the W3C documentation for the Semantic Web[1] and are complemented by concepts introduced in a previously work [33]. The illustrative examples presented in this section are drawn from the film industry, with a particular focus on the movie Titanic.

3.1 Communication: Problem formulation

Understanding requirements is a fundamental activity in software engineering, as it serves as a bridge between problem definition and the subsequent design and construction phases. Requirements provide a structured framework for identifying needs, defining the scope of the problem, and specifying the characteristics of the

[1] `https://www.w3.org/2001/sw/wiki/Main_Page`

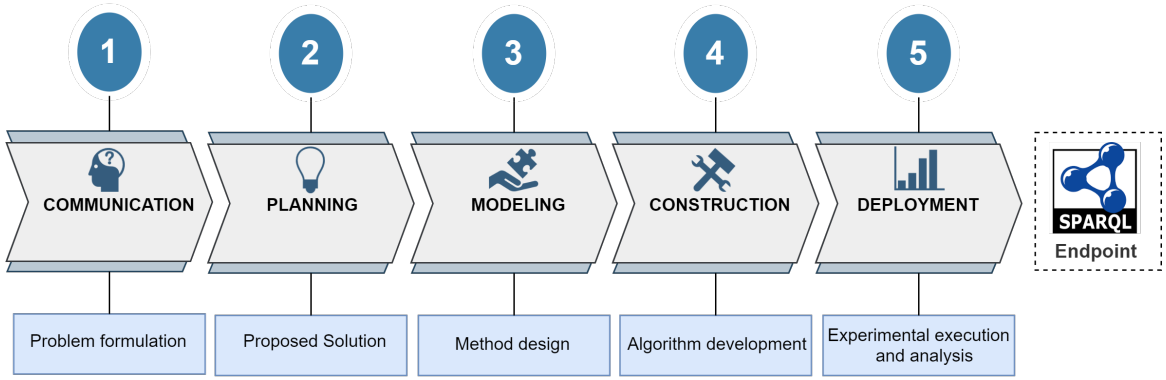

Figure 1: Methodology based on Pressman's software process model, adapted to algorithmic development for SPARQL query validation.

expected solution. In this context, this section presents a systematic formulation of the problem, progressing from the most general aspects to the most specific ones. The discussion is organized along four analytical dimensions: Domain of Study, Research Focus, Unit of Analysis, and Specific Context (Figure 2).

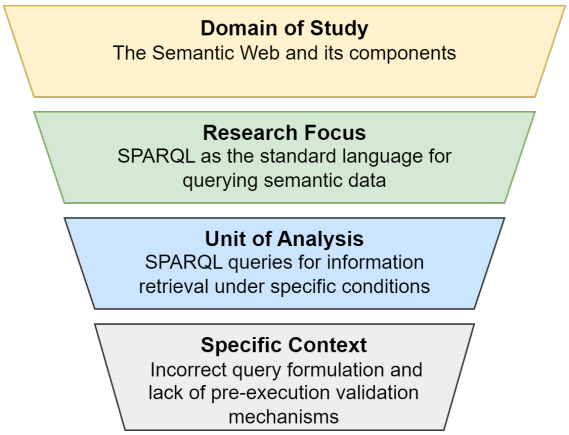

Figure 2: Aspects argued in the formulation of the problem

3.1.1 Domain of Study

The domain of study of this work is the Semantic Web, considered specifically from the perspective of its core representation and modeling components. Rather than reiterating its general objectives, this section focuses on the formal structures that enable semantic description, consistency, and automated processing, which are essential for the validation of SPARQL queries.

Within this context, knowledge is represented through ontologies [7, 8], which provide a formal specification of concepts and the relationships that exist among them within a particular domain [34]. Ontologies play a central role in Semantic Web systems by enabling shared understanding, semantic interoperability, and reasoning [35]. They are commonly serialized using the Web Ontology Language [9]. An ontology can be formalized as a tuple $O = (C, P, I, A)$ that specifies a domain through concepts C, properties P, instances I, and axioms A that describe their relationships [36].

OWL is based on RDF (Resource Description Framework) and RDFS (RDF Schema), but incorporates a more expressive vocabulary for describing classes and properties [10]. In RDF, knowledge is modeled by statements about resources in the form of triples[2] $t : <s, p, o>$. We consider well-formed triples as $(U \cup B) \times U \times (U \cup B \cup L)$, where U is the set of all valid IRIs[3], L is the set of literals, and B is the set of blank nodes; U, B, and L are disjoint sets. RDF uses IRIs as global identifiers to define the subjects, predicates, and objects involved in these statements.

[2]https://www.w3.org/TR/rdf12-concepts/#section-triples
[3]https://www.w3.org/TR/2014/REC-rdf11-concepts-20140225/#section-IRIs

3.1.2 Research Focus

The research focus of this work is on SPARQL.

SPARQL plays a fundamental role in data manipulation within the Semantic Web. As Tim Berners-Lee[4] points out, "*Trying to use the Semantic Web without SPARQL is like trying to use a relational database without SQL*", emphasizing its indispensability as a standard query language. Its syntax is similar to SQL [20], which facilitates its adoption, although its underlying logic is closely linked to the structure and semantics of RDF data. Since its standardization in 2008, it has become established as the standard language for information retrieval in the Semantic Web [4, 5].

3.1.3 Unit of Analysis

The unit of analysis in this work is the SPARQL query, understood as the formal mechanism used to retrieve specific information from RDF datasets under well-defined conditions. A SPARQL query expresses an information need by specifying constraints over a dataset, enabling the extraction of relevant data from a potentially large semantic graph [37]. This selection process is primarily governed by the WHERE clause, which defines the graph patterns that must be satisfied for data to be considered relevant, while the SELECT clause determines which elements of the matched data are ultimately returned in the query results.

In the example illustrated in Figure 3, the underlying semantic dataset comprises films produced in 1997; however, the objective of the query is more specific, namely to identify the leading actors in the film *Titanic*. To achieve this, the WHERE clause constrains the query evaluation by delimiting the subset of data that satisfies this condition, thereby ensuring that only the information relevant to the query intent is retrieved.

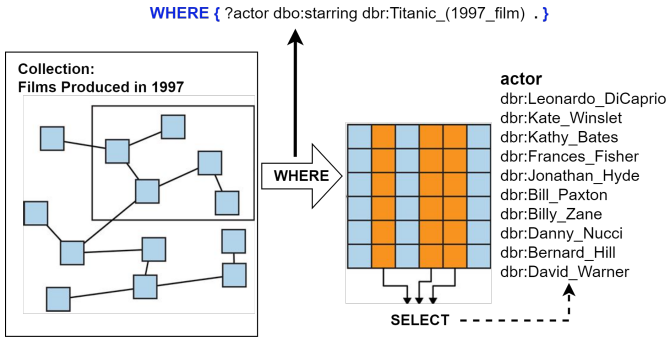

Figure 3: *WHERE* specifies the data to be extracted; *SELECT* selects the data to be displayed [37].

Following the formulation proposed in [6], a selection query enables the retrieval of knowledge from a semantic repository. Its formal definition is presented below.

An RDF graph G is a finite set of triples defined as: $G = \{t_1,t_2,...,t_n\}$ and each $t_i : <s, p, o>$, $i = 1...n$; an RDF graph is a set of facts belonging to the world (universe of discourse), whose elements are called Resources R. A *referential* resource is denoted by an IRI R_{ref} and a resource denoted by a literal is called a *literal value* R_{lit}. The relationship between resources is determined by a *predicate* P, a binary relationship expressed as *P(s, o)*. A SPARQL query Q is composed of a set of triple patterns that make up the Basic Graph Pattern (BGP), where the position of *subject*, *predicate*, and *object* can be variables V, which provides flexibility in the formulation of queries on RDF graphs.

Definition 1 (Selection Query) *Let $V = \{?x, ?y, \ldots\}$ be a set of variables. A Triple Pattern is an element of the set $(R_{\text{ref}} \cup V) \times (P \cup V) \times (R_{\text{ref}} \cup R_{\text{lit}} \cup V)$. A BGP is a finite set of triple patterns. A selection query is defined as $Q = (G_Q, V_Q)$, where G_Q is a BGP and $V_Q \subseteq V$ is the set of output variables appearing in G_Q. The evaluation of Q on an RDF graph G is denoted $Q(G)$.*

3.1.4 Specific Context

SPARQL queries are usually simple and the syntax is easy to learn. Despite their simplicity, however, programmers can make many mistakes, and correcting queries can be difficult. Incorrect queries not only produce empty responses, but can also generate erroneous or unexpected results [4]. In this context, programmers ask themselves: why doesn't the query return what I'm looking for? Errors in a query can be due to both typing mistakes and misconceptions arising from an inadequate understanding of the RDF

[4]https://www.w3.org/2007/12/sparql-pressrelease

graph, hence variables can bring an unexpected response, such as empty responses, few responses, too many responses, or unwanted results [13].

In this sense, the correct formulation of SPARQL queries requires a thorough understanding of the semantic dataset [6], which is the result of the instantiation of an ontology in a specific domain. Consequently, when performing a query, it is necessary to understand both the domain and the ontological schema that structures the data. However, ontologies are usually designed by ontology engineers in conjunction with domain experts [38], and in most cases, programmers who query these repositories do not have detailed knowledge of these aspects, leading them to formulate SPARQL queries using a trial-and-error approach.

This approach presents two main problems. First, users are generally unaware of the exact cause of the error in the query, which leads them to make multiple modifications and successive executions until they obtain an apparently correct result. Second, each execution of an incorrect query generates unnecessary computational load on the server, which directly impacts its performance. In web environments, server performance is affected by the load derived from user requests, hence it is essential to control and optimize the use of computational resources. This load influences various parameters for measuring server performance, including overall performance (number of requests served in a time interval), response time, CPU utilization, memory consumption, number of requests processed per second, failed request rate, and concurrency level, understood as the number of requests that the server can handle simultaneously [15]. From the point of view of internal processing in an RDF management system, also known as a triplestore [39], query processing involves several stages (Figure 4): query parsing, logical query plan selection, physical query plan selection, and query execution, each of which consumes computational resources [14]. Consequently, an incorrect SPARQL query generates unnecessary load in all these stages, affecting the performance of an RDF management system even before producing results. In this context, the repeated execution of incorrect SPARQL queries not only hinders the development process but also degrades the server's computing capacity.

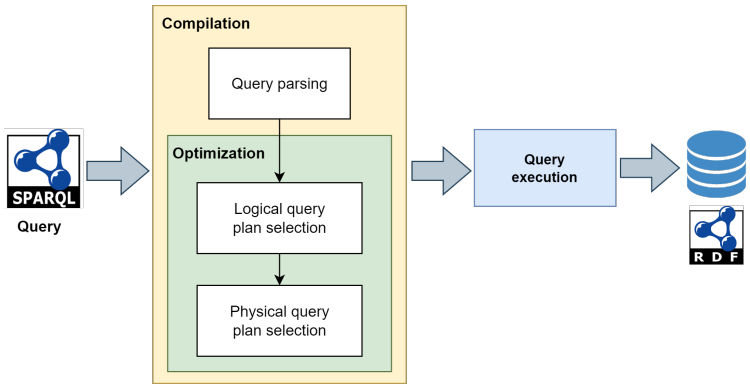

Figure 4: Overview of query processing [14].

The formulation and execution of SPARQL queries is critical to the effective use of semantic repositories, as syntactic errors and semantic inconsistencies not only affect the results obtained, but also generate unnecessary computational load that degrades the performance of an RDF management system. This problem is exacerbated by the sustained growth of linked data, which has increased the complexity of its processing and posed significant challenges in terms of optimization and efficiency. In this context, the efficient execution of SPARQL queries plays a critical role in data management in the Semantic Web [40], reinforcing the need to study and evaluate mechanisms that contribute to improving their performance. It is therefore desirable and necessary to investigate proposals aimed at the prior validation and optimization of SPARQL queries, with the aim of reducing errors, minimizing unnecessary executions, and optimizing the use of computational resources, thus promoting a more efficient exploitation of semantic data.

3.2 Planning: Proposed Solution

Once the problem has been identified and discussed, this section outlines the planned approach adopted to address it and describes the stages involved in the proposed solution. Specifically, it defines the validation method, its main components, and the overall operational flow that underpins the SPARQL query validation process. The section also specifies the inputs and outputs of the method, introduces the components responsible for syntactic and semantic validation, and defines the set of queries used as test cases. In addition, the experimental setup is described, including the dataset employed, the variables to be measured, and the metrics used to evaluate the performance of the proposed method.

The proposed solution focuses on the validation of SPARQL queries prior to their execution by an RDF

query engine or SPARQL processor. This validation is performed using an input ontology and does not require access to ontology instances. The approach incorporates two complementary levels of validation: syntactic validation, aimed at detecting errors in the formulation of SPARQL queries, and semantic validation, which focuses on the correct use of object properties according to the ontological model. When a query is identified as incorrect, the validator produces a detailed report of the detected errors, thereby supporting users in refining queries before execution.

Figure 5 presents a component diagram illustrating the flow of inputs, processes, and outputs for the syntactic and semantic validation of SPARQL queries.

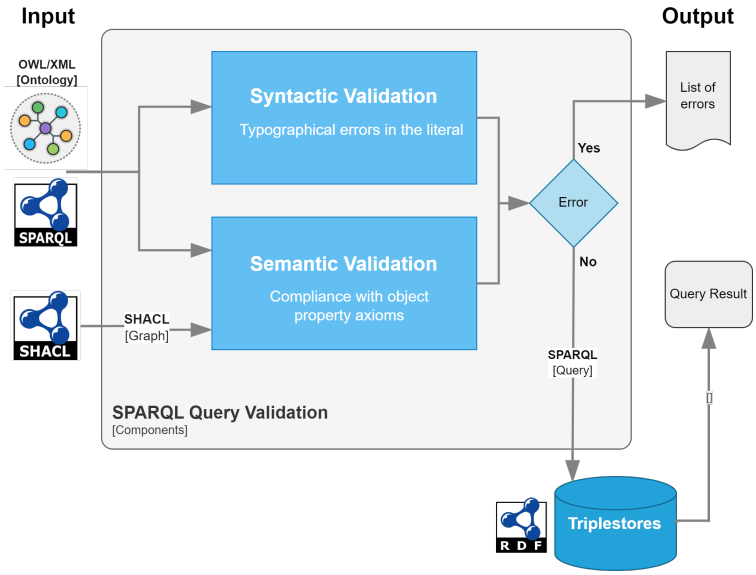

Figure 5: SPARQL query validation component diagram

As input, the method receives an ontology, a SPARQL query, and a shapes graph. These three elements must be present for the method to work correctly. As output, the method produces a list of identified errors, indicating for each one a brief explanation and the property involved when the query is incorrect. If the query is valid, the method returns the expected results. The method consists of two components: syntactic validation and semantic validation.

3.2.1 Syntactic Validation

Syntactic validation focuses primarily on the analysis of literals. A literal is a type of node that represents a value, through which individuals are described using data such as date of birth, email address, or age. For this purpose, OWL provides *Data Properties*, which connect individuals with data values[5]. The following example indicates that John's age is 42 (*<John><hasAge>"42"<xsd:integer>*).

The Semantic Web encourages the datatype in a *Datatype Property* to be implicitly included in the literal, as a description of the value [22]. RDF literals combine a string and an IRI that identifies a datatype. In this sense, the validation of the literal considers the *lexical form* and the *datatype IRI*.

Figure 6 presents the elements of the Literal. The Literal contains the *Lexical Form: string* ("42") and an explicit datatype (*xsd:integer*). The *Lexical Form* must be a valid string in the *Lexical Space* and must be contained in the set of all possible *Value Space* values for the datatype. Moreover, the datatype must be consistent with the *range* of the ontology's *Datatatype Property*. In addition, the ontology could have restrictions on the range of the *Datatype Property*, in this case the *"data value"* must be valid for the *constraining facet*.

In an ontology, the *range* of a *Datatype Property* comprises two components: (a) *Built-in Datatype*, which are the data types predefined by XML Schema Definition (`xsd:`), and (b) *Data Range Expression*, more complex expressions or conditions on the datatypes. For our study, the concept of *"range"* will be associated with the built-in data type (a), and range expressions for *"constraining facets"* (b).

[5]https://www.w3.org/TR/owl2-primer/#Datatypes

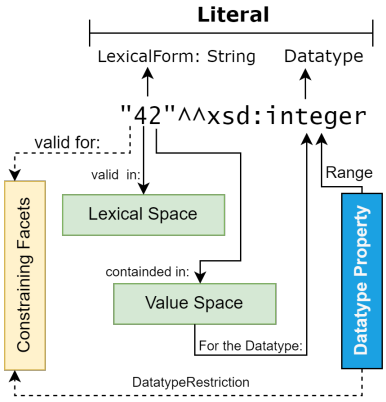

Figure 6: Components of an RDF Literal

3.2.2 Semantic Validation

Semantic validation focuses on the analysis of object property axioms. The user may have an incorrect understanding of the ontology and this may lead to a wrong interpretation of the RDF graph.

In OWL ontologies, axioms[6] constitute a fundamental component, as they define the statements that are considered true within a specific domain [41]. Semantic validation is performed under a set of specific conditions: the query is transformed into an RDF graph (*"data graph"*), which is validated using a document *SHACL* (*"shapes graph"*) that encodes the semantic constraints and rules of the domain.

Table 1 presents the object property characteristics considered in our proposal [33].

Table 1: Object property axioms

Object Property Axioms	Syntax	Semantics
Functional	Fun(R)	$(x,y) \in R^I \wedge (x,z) \in R^I \Rightarrow y = z$
Inverse-Functional	Ifu(R)	$(x,y) \in R^I \wedge (z,y) \in R^I \Rightarrow x = z$
Transitive	Tra(R)	$(x,y) \in R^I \wedge (y,z) \in R^I \Rightarrow (x,z) \in R^I$
Symmetric	Sym(R)	$(x,y) \in R^I \Rightarrow (y,x) \in R^I$
Asymmetric	Asy(R)	$(x,y) \in R^I \Rightarrow (y,x) \notin R^I$
Reflexive	Ref(R)	$(x,x) \in R^I$
Irreflexive	Irr(R)	$(x,x) \notin R^I$

3.2.3 Definition of evaluation queries

This section defines the set of SPARQL queries that will be used to evaluate the proposed method. Each query was designed to represent specific errors, described in following sections, in order to validate the proposal syntactically and semantically. Queries are used as test cases for experiments.

Following listing shows twelve SPARQL queries for syntactic validation.

```
Q1:   SELECT ?y WHERE {
P1:     ?x1 dbp:father ?y .
P2:     ?x2 dbp:father ?y .
P3:     ?y dbo:birthDate "1974-11-11"^^xsd:date .
F1:     FILTER (?x1 != ?x2)}

Q2:   SELECT ?x ?y ?z WHERE {
        ?x dbo:starring ?y .
        ?y dbo:birthDate ?z .
        FILTER ( ?z >= "1970-01-01"^^xsd:date &&
                 ?z < "1980-01-01"^^xsd:date )}

Q3:  SELECT ?x WHERE {
        ?x dbo:birthDate "1974-11-11"^^xsd:gYearMonth}

Q4:  SELECT ?x WHERE {
        ?x dbo:birthDate "1974-11-11"^^xsd:dateTime}

Q5:  SELECT ?x WHERE {
        ?x dbo:birthDate "11-11-1974"^^xsd:date}
```

[6] https://www.w3.org/TR/2012/REC-owl2-syntax-20121211/#Axioms

```
Q6:  SELECT ?x WHERE {
        ?x dbo:birthName "Leonardo DiCaprio"^^xsd:char}

Q7:  SELECT ?x WHERE {
        ?x dbo:birthDate "1897-06-01"^^xsd:date}

Q8:  SELECT ?x ?y ?z WHERE {
             ?x dbo:starring ?y .
             ?y dbo:birthDate ?z .
             FILTER (?z < "1900-01-01"^^xsd:date)}

Q9:  SELECT ?x WHERE {
             ?x dbo:birthPlace ?y .
             ?x dbo:birthPlace ?z .
             FILTER (?y != ?z)}

Q10: SELECT ?x ?y WHERE {
             ?x dbo:starring ?y}

Q11: SELECT ?x1 ?x2 ?y WHERE {
             ?x1 dbp:father ?y .
             ?x2 dbp:father ?y .
             ?y dbo:birthDate "11-11-1974"^^xsd:dateTime
             FILTER (?x1 != ?x2)}

Q12: SELECT ?x WHERE {
        ?x dbo:director "Titanic_(1997_film)"^^xsd:string}
```

Listing 1: SPARQL queries for syntactic validation

In Listing 2, SPARQL queries for semantic validation are shown.

Inverse-Functional *ns:p* = *dbp:father*, Rule: $x1$ *ns:p* y, $x2$ *ns:p* $y \rightarrow x1 = x2$

```
Q13: SELECT ?y WHERE {
             ?x1 dbp:father ?y .
             ?x2 dbp:father ?y .
             FILTER (?x1 != ?x2)}
    Note: The condition (?x1!=?x2) is in logical conflict.
```

Functional *ns:p* = *dbo:birthPlace*, Rule: x *ns:p* $y1$, x *ns:p* $y2 \rightarrow y1 = y2$

```
Q14: SELECT ?x WHERE {
             ?x  dbo:birthPlace ?y1 .
             ?x  dbo:birthPlace ?y2 .
             FILTER (?y1 != ?y2)}
    Note: The condition (?y1!=?y2) is in logical conflict.
```

Symmetric *ns:p* = *dbo:starring*, Rule: x *ns:p* $y \rightarrow y$ *ns:p* x

```
Q15: SELECT ?x ?y WHERE {
             ?x  dbo:starring ?y
             FILTER NOT EXISTS {?y dbo:starring ?x}}
    Note: The expression FILTER NOT EXISTS {p} indicates that p must not be fulfilled,
    which contradicts Symmetric.
```

Asymmetric *ns:p* = *dbp:children*, Rule: x *ns:p* $y \rightarrow y \neg$ *ns:p* x

```
Q16: SELECT ?x ?y WHERE {
             ?x dbp:children ?y .
             FILTER EXISTS {?y dbp:children ?x}}
    Note: The expression FILTER EXISTS {p} indicates that p must be fulfilled,
    this contradicts Asymmetric.
```

Functional *ns:p* = *dbo:country*, Rule: x *ns:p* $y1$, x *ns:p* $y2 \rightarrow y1 = y2$

```
Q17: SELECT ?x  WHERE {
    ?x dbo:country ?y1 .
    ?x dbo:country ?y2 .
    FILTER (?y1 != ?y2)}
Note: The condition (?y1!=?y2) is in logical conflict.
```

Listing 2: SPARQL queries for semantic validation

Listing 3 shows an excerpt from the ontology in OWL Functional Syntax format generated in our work [33]. In this excerpt, a restriction has been added for the *Datatype Property dbo:birthDate*.

```
Prefix(dbo:=<http://dbpedia.org/ontology/>)
Prefix(dbp:=<http://dbpedia.org/property/>)
Prefix(dbr:=<http://dbpedia.org/resource/>)
Prefix(xsd:=<http://www.w3.org/2001/XMLSchema#>)

InverseFunctionalObjectProperty(dbp:father)
FunctionalObjectProperty(dbo:birthPlace)
SymmetricObjectProperty(dbo:starring)
AsymmetricObjectProperty(dbp:children)
FunctionalObjectProperty(dbo:country)
```

```
DataPropertyRange(dbo:birthDate
                  DatatypeRestriction(xsd:date
                    xsd:minInclusive "1900-01-01"^^xsd:date))
DataPropertyRange(dbo:birthName xsd:string)
DataPropertyAssertion(dbo:birthName dbr:Leonardo_DiCaprio
                      "Leonardo DiCaprio"^^xsd:string)
DataPropertyAssertion(dbo:birthDate dbr:Leonardo_DiCaprio
                      "1974-11-11"^^xsd:date)
ObjectPropertyAssertion(dbp:father dbr:George_DiCaprio
                        dbr:Leonardo_DiCaprio)
ObjectPropertyAssertion(dbo:birthPlace dbr:Leonardo_DiCaprio
                        dbr:Los_Angeles,_California)
ObjectPropertyAssertion(dbo:starring dbr:Leonardo_DiCaprio
                        dbr:Titanic_(1997_film))
ObjectPropertyAssertion(dbo:starring dbr:Titanic_(1997_film)
                        dbr:Leonardo_DiCaprio)
ObjectPropertyAssertion(dbp:children dbr:Leo_Penn
                        dbr:Chris_Penn)
ObjectPropertyAssertion(dbo:country dbr:Los_Angeles,_California
                        dbr:United_States)
ObjectPropertyAssertion(dbo:director dbr:James_Cameron
                        dbr:Titanic_(1997_film )
```

Listing 3: Excerpt from ontology in the domain of the film industry

3.2.4 Experiment Setup

In a previous work [33], we have implemented a dataset of 16005 RDF triples, comprising 500 films produced in 1997. This dataset includes information on films, directors, music composers, production companies, actors, and attributes associated with the films, among other relevant elements from the film industry domain.

The experimental work will consist of measuring the processing time, expressed in milliseconds, on RDF datasets of different sizes for the same SPARQL query. However, in order to characterize the behavior of the method in different validation scenarios, we will consider the set of queries defined in Section 3.2.3, which we denote as $Q = \{Q_1, Q_2, \ldots, Q_m\}$. Each Q_j corresponds to a specific experimental event and can represent a correct or incorrect SPARQL query. Thus, for each query Q_j, the evolution of the processing time will be observed as the size of the RDF dataset increases.

The size of the RDF dataset will be defined as the independent variable, while the dependent variable will correspond to the processing time.

Therefore, the processing time of query Q_j for a set of n triples is defined as:

$$\overline{T}_{Q_j}(n) = \frac{1}{k} \sum_{i=1}^{k} T_{Q_j}^{(i)}(n), \quad j \in \{1, \ldots, m\},\ n \in \{1000, 2000, \ldots, 16005\} \tag{1}$$

Where:

- n represents the **number of RDF triples** considered in the experiment (independent variable), evaluated in increments of 1000 until reaching a maximum of 16005 triples.
- Q_j corresponds to an **experimental event**, understood as the execution of a specific SPARQL query, which remains constant during each individual evaluation, with $j \in \{1, \ldots, m\}$.
- k is the **number of iterations** performed for each value of n; in this work, $k = 10$ will be considered.
- $T_{Q_j}^{(i)}(n)$ denotes the **processing time** of the i-th iteration of query Q_j on a set of n triples, measured in milliseconds (ms).
- $\overline{T}_{Q_j}(n)$ corresponds to the **average processing time** of Q_j for the set of n triples, calculated as the arithmetic mean of the k iterations performed.

Finally, the data obtained will be analyzed and interpreted jointly, performing comparative analyses between the different results, in order to characterize their temporal behavior in relation to the growth in data volume.

The metrics used to evaluate the performance of the algorithms will be as follows:

- Average execution time, measured in milliseconds.
- Standard deviation, used as an indicator of system stability.
- Total processing time, considering all components involved in the execution of each process.

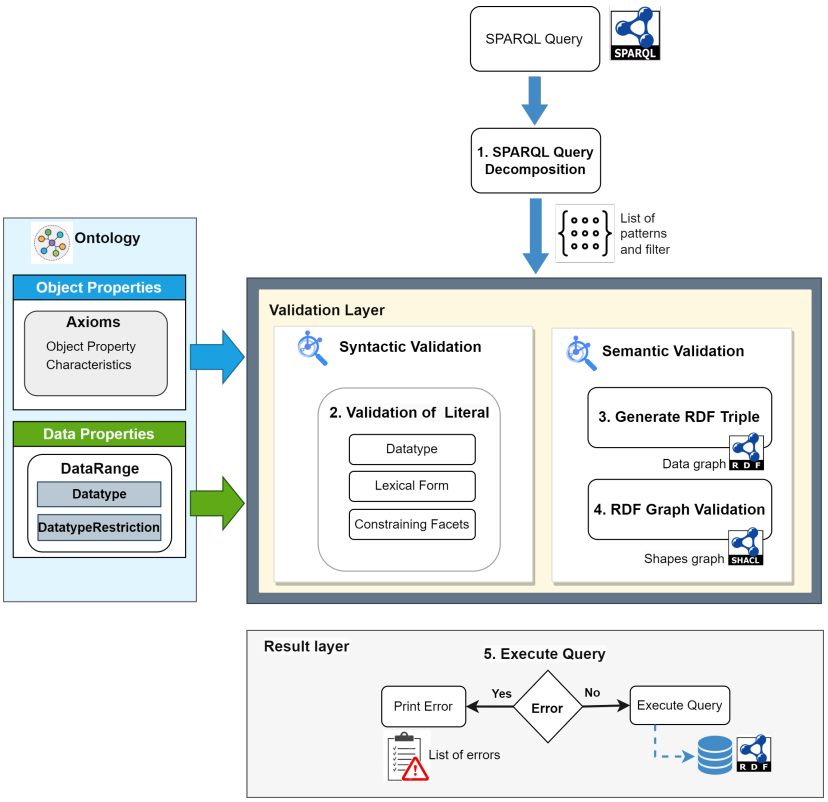

Figure 7: Phases for Syntactic and Semantic Validation

3.3 Modeling: Method design

The modeling activity develops the design of the proposed method by formalizing its internal components. This section introduces the functional diagrams and pseudocode corresponding to each validation stage, in order to clearly specify how the method works. The result of the modeling establishes the conceptual basis for the construction stage, in which the method will be implemented. Additional details supporting this modeling phase are provided in Collio et al. [1].

Figure 7 shows the proposed method for the Syntactic and Semantic Validation of SPARQL queries. This procedure is structured in 5 Phases. The Algorithm 1 *"SPARQL Query Validation" (SQV)* presents the main flow for SPARQL query validation.

1. SPARQL Query Decomposition: The query pattern is decomposed to generate separate triple patterns and FILTER expression.
2. Validation of Literal: The validation of the literal in the triple pattern is performed, in terms of: Datatype, Lexical Form and Constraining Facets. Constraining facet in FILTER expression is verified.
3. Generate RDF Triple: RDF triples and RDF graph generation are built.
4. RDF Graph Validation: The RDF Graph is validated based on a set of *SHACL* conditions.
5. Execute Query: The query is executed or the error list is printed, as appropriate.

Algorithm 1 SPARQL Query Validation (SQV)

Require: Q, O, D //Input: SPARQL Query Q and ontology O and SHACL document D
$list_{error} = TF_{syn} = TF_{sem} = G = EMPTY$
$M_{list}, prefix\_IRI \leftarrow SPARQLqueryDecomposition(Q)$
$list_{error}, TF_{syn} \leftarrow LiteralValidation(M_{list}, O)$
$G \leftarrow RdfTripleGeneration(M_{list}, prefix\_IRI)$
$list_{error}, TF_{sem} \leftarrow RDFGraphValidation(G, D)$
if $TF_{syn} = True$ and $TF_{sem} = True$ **then**
    Execute Query Q //SPARQL query result
else
    Print $list_{error}$ // Print list of error
end if

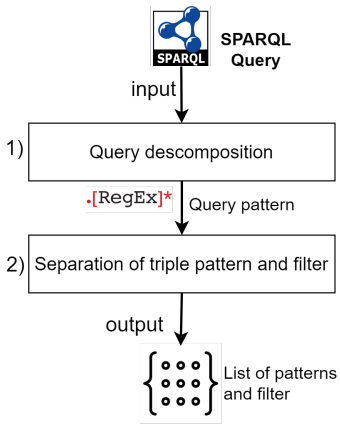

Figure 8: Steps for SPARQL Query Decomposition

3.3.1 SPARQL Query Decomposition

At this stage, the triple patterns and FILTER expression of the SPARQL query are extracted and a list is generated as input for the following stages. It is assumed that the SPARQL query is well formed. The "SPARQL Query Decomposition" process is performed in 2 Steps (Figure 8); Algorithm 2 shows our proposal for "SPARQL Query Decomposition".

Step 1, Query decomposition

Starting from an input SPARQL query, this is broken down, meaning that the query pattern delimited between curly brackets {} is extracted. This can be done with regular expressions (RegEx).

Step 2, Separation of triple pattern and filter

At this stage, the triple patterns and FILTER expression are identified, for which we create a 4-column matrix M with the form $\mathcal{M}_{m\times 4}(\mathbb{M})$, where m is the number of finite rows. Let $\mathbb{M}$ be the elements of the matrix of the form $\{m :< g, a, b, c >\}$, where g identifies whether the row is a triple pattern (P) or FILTER expression (F), a is the position of *subject*, b is the position of *predicate/property*, c is the position of *object* of the triple patterns. For the FILTER expression, the variable can be on the left or right side of the comparison operator, so b will be the comparator, a and c will be the variables or constant comparison values. In our proposal, we analyze that the FILTER expression contains comparison operators ($<$, $>$, $>=$, $<=$, $=$). In $\mathbb{M}$, the decomposed query pattern is entered, and the output is a list with the triple patterns and FILTER expression separately.

Algorithm 2 SPARQL Query Decomposition

Require: Q //Input: SPARQL Query
1: $M_{list} : \{m_i :< g_i, a_i, b_i, c_i >\}$
2: $triple\_pattern, filter, prefix\_IRI \leftarrow RegEx(Q)$
3: **for** tp **in** $triple\_pattern$ **do**
4: $s \leftarrow extractSubject(tp)$
5: $p \leftarrow extractPredicate(tp)$
6: $o \leftarrow extractObject(tp)$
7: $M_{list} \leftarrow ADD\ m_i :<$ "P", $s, p, o >$
8: **end for**
9: **for** f **in** $filter$ **do**
10: $a \leftarrow extractPosition1(f)$
11: $b \leftarrow extractPosition2(f)$
12: $c \leftarrow extractPosition3(f)$
13: $M_{list} \leftarrow ADD\ m_i :<$ "F", $a, b, c >$
14: **end for**
15: **return** $M_{list}, prefix\_IRI$ //Output: List with triple patterns and FILTER expressions (M_{list}) and namespace prefix bindings $prefix\_IRI$

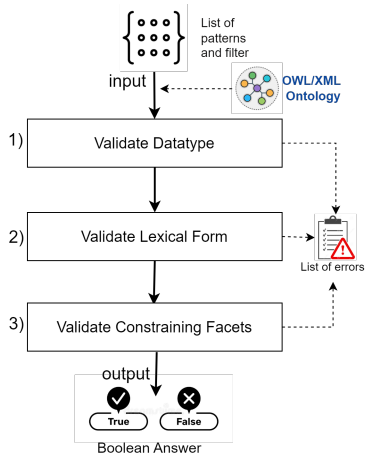

Figure 9: Steps for Literal Validation

3.3.2 Validation of Literal

The validation of literals is divided into three components: (1) Datatype: it is verified that the specified datatype belongs to the set of types recommended by the W3C and that it is also consistent with the *range* defined in the ontology for the *Datatype Property*; (2) Lexical Form: it is checked that the value is a valid string within the Lexical Space and that it is contained in the corresponding Value Space; and (3) Constraining Facets: it is validated that the value of the literal complies with the restrictions defined in the Facet Space established by the ontology. Simple and language-tagged Literals, such as "Hello"@en, are not considered.

To start the validation we have as input a list of triple patterns and FILTER expression, the validation of the Literal is performed when the *Property* corresponds to a *Datatype Property*. The triple pattern will be $\mathbb{M}(g)$=*"P"* and $\mathbb{M}(b)$=*DatatypeProperty*. For the FILTER expression $\mathbb{M}(g)$=*"F"*, we consider the variable in the FILTER expression to identify which one is the corresponding *DatatypeProperty*. The Validation of Literal process is performed in 3 Steps (Figure 9). Algorithm 3 describes the literal validation process. Algorithms 4 and 5 are sub-processes of Algorithm 3.

Algorithm 3 Literal Validation

Require: M_{list} , O // Input: List with triple patterns and FILTER expressions and ontology O

1: SDT : {"string","boolean",...,"unsignedByte"}// datatype W3C
2: $list_{lexSpace}$:{'xsd:string': 'Any character', 'xsd:date': 'CCYY-MM-DD',..., 'float':32-bit floating point type}
3: $list_{error}, TF_{syn} \leftarrow TriplePattern(M_{list}, O, SDT, list_{lexSpace})$
4: $list_{error}, TF_{syn} \leftarrow FILTERexpression(M_{list}, O)$
5: **return** $list_{error}, TF_{syn}$ //Output: list of errors and boolean expression.

Step 1, Validate Datatype

In this part we analyze the datatype defined in the Literal considering that:

- The datatype belongs to the data set recommended by W3C[7].
- The *range* of the *Datatype Property* of the ontology is equal to the datatype defined in the SPARQL query literal.

In $Q6$ the datatype *xsd:char* is not a valid datatype recommended by W3C, or there is a typo. In $Q4$ the datatype (*xsd:dateTime*) of the query is not equal to the *range* of the *Datatype Property dbo:birthDate* defined in the ontology.

[7]https://www.w3.org/TR/xmlschema-2/#built-in-datatypes

Algorithm 4 Triple Pattern

Require: M_{list} , O, SDT, $list_{lexSpace}$ //$list_{error}(L_e)$ document contains (type of error, description of the message(mes), affected property).

$M_{list} : \{m_i :< g_i, a_i, b_i, c_i >\}$
$TF_{syn} = True$, $range = Built\ in\ Datatype$
for ml **in** M_{list} **do**
  // Evaluation of the Literal in triple pattern
  if $ml.g =$ "P" and $DataProperty = O(ml.b)$ **then**
    if $ml.c$!= "?$" **then** //$ml.c$ is not a variable
      $lexForm, datatype \leftarrow\ ml.c$ // is literal
      if $datatype$ not in SDT **then**
        $TF_{syn} = False$, L_e += "datatype" + mes + $ml.b$
      end if
      // DataPropertyRange
      $range \leftarrow\ O(ml.b)$
      if $datatype$!= $range$ **then**
        $TF_{syn} = False$, L_e += "range" + mes + $ml.b$
      end if
      $lexSpace \leftarrow\ list_{lexSpace}[range]$
      if $lexForm \notin lexSpace \wedge ValueSpace[range]$ **then**
        $TF_{syn} = False$, L_e += "LexicalSpace" + mes + $ml.b$
      end if
      //DatatypeRestriction
      $patternFacet \leftarrow\ O(ml.b)$
      if $lexForm$ not in $patternFacet$ **then**
        $TF_{syn} = False$, L_e += "Facet" + mes + $ml.b$
      end if
    end if
  else
    $TF_{syn} = False$, L_e += "PropertyType" + mes + $ml.b$
  end if
end for
$list_{error} \leftarrow\ L_e$ // set of errors
return $list_{error}, TF_{syn}$

Algorithm 5 FILTER expression

Require: M_{list} , O

$M_{list} : \{m_i :< g_i, a_i, b_i, c_i >\}$
$TF_{syn} = True$
for ml **in** M_{list} **do**
  if $ml.g =$ "F" **then**
    $property \leftarrow\ m_i.b\ IN\ M_{list}\ WHERE\ m_i.g =$ "P"AND $O(ml.b)$=$DataProperty\ AND\ m_i.c$=$ml.a$
    if $property\ is\ not\ null$ **then**
      if $ml.b =$">" **then**
        $c \leftarrow ml.c(Next)$
      else if $ml.b =$"<" **then**
        $c \leftarrow ml.c(Back)$
      else
        $c \leftarrow ml.c$
      end if
      $filterFacet \leftarrow\ O(property)$ // DatatypeRestriction
      if c not in $filterFacet$ **then**
        $TF_{syn} = False$, $list_{error}$ += "Facet" + mes + $ml.b$
      end if
    end if
  end if
end for
return $list_{error}, TF_{syn}$

Step 2, Validate Lexical Form

This part validates that the *Lexical Form* is a valid string in the *Lexical Space*, also contained in the *Value Space*. It is verified that the *Lexical Space* corresponds for the *range* of the ontology's *Datatype Property*.

In the $Q5$ the *Lexical Form* cannot be converted to a valid value for the datatype, in this case it is an ill-typed Literal. The correct way is to invert the date of the form "1974-11-11", since for the datatype *xsd:date* its *Lexical Space* is *CCYYY-MM-DD*. In $Q3$ the *Lexical Form* cannot be converted to a month and year data value, since $gYearMonth$ represents a month within a specific year. For these cases the query response would be empty.

Step 3, Validate Constraining Facets

In an ontology we can restrict datatypes by using constraining facet, to define more specific ranges. The result is a subset of the *Value Space* for the datatype that applies. For example we can say that the working age will be between 18 and 65.

In this stage, *Facet Space* defined in the ontology by *DatatypeRestriction* is validated for the range of a *Datatype Property*. It is verified that *"data value"* (dv) of the Literal is in the VS' subset of the Values Space (VS), dv is a valid constrained value for the datatype. A Facet Space is a pair of the form (F,v) where F is an IRI called a constraining facet and v the constraining datatype value. Then $dv \in VS$', where $VS' = \{x \in VS \mid x\ satisfies\ the\ facet\ F\ with\ value\ v\}$.

For our study the scope of validation is for the constraining Facets[8]: $minInclusive$, $minExclusive$, $maxInclusive$, $maxExclusive$, for numeric and date data types in triple pattern and FILTER expression. For string data types ($length, minLength, maxLength$) in triple pattern.

The $Q7$ is incorrect since in the ontology the *Datatype Property dbo:birthDate* restricts the data value, in this case the dates must be greater than or equal to "1900-01-01". Similarly, in $Q8$ the FILTER expression indicates dates less than "1900-01-01". In both cases the query would be empty.

For the evaluation of the FILTER expression the variable ("?*") addresses the *Datatype Property*, the constant value of comparison c must be modified, when the comparison operator is greater than ($>$) move to the next value and when it is less than ($<$) go to the previous value.

3.3.3 Generate RDF Triple

This stage is the initial part for semantic validation. It starts with the generation of an RDF graph G. The goal is for each triple pattern to be converted to an RDF triple, G must be serialized in an RDF compatible format [42]. We have as input a list of triple patterns, where $\mathbb{M}(g)$=*"P"*, returns only the triple patterns of the query. Each variable, which is identified by the symbol *"?"* or *"$"* at the beginning, becomes a Blank Node, usually an RDF graph contains IRI, Literals and Blank Nodes. Each t_i must go with its corresponding Namespace (ns) IRIs. The separation of the *Prefix* of the query is done with regular expressions.

The output will be an RDF graph, the generated graph will never be that long even if the query is complex. The Algorithm 6 shows our approach to Generate RDF Triple.

Algorithm 6 RDF Triple Generation

Require: $M_{list}, prefix\_IRI$ //Input: List with triple patterns and FILTER expressions and namespace prefix bindings.

1: $triple = EMPTY$
2: $M_{list} : \{m_i :< g_i, a_i, b_i, c_i >\}$
3: $List_{triple} \leftarrow\ * IN\ M_{list},\ WHERE\ m_i.g =$ "P"
4: **for** t **in** $List_{triple}$ **do**
5: **if** $t.a, t.b, t.c\ is\ variable$ **then** // is variable ("?")
6: blank node ($s = t.a, p = t.b, o = t.c$)
7: **else**
8: ($s = t.a, p = t.b, o = t.c$)
9: **end if**
10: $triple$ += $triple + s + p + o +$ "."
11: **end for**
12: $G \leftarrow\ prefix\_IRI + triple$
13: **return** G //Output: RDF graph G serialized.

3.3.4 RDF Graph Validation

In this stage, the RDF graph is validated under a set of *SHACL* conditions. The formalization for the validation is based on that proposed by *Pareti and Konstantinidis* [43]. Validation requires 1) an RDF Graph G to be validated and 2) a document *SHACL* D that defines the validation conditions.

Formally, G is an RDF Graph and $D = \{s_1, s_2, ..., s_n\}$ a set of *SHACL* shapes, with $s_i = (t_i, c_i)$, where t_i is a target definition and c_i a constraint.

Figure 10 shows the 2 steps for the RDF Graph Validation.

Step 1, Generate SHACL Graph

The *Shapes Graph* generation process is based on the automatic extraction of the *object property characteristics* defined in the input ontology O, as described in Table 1. Each identified OWL axiom is transformed

[8] https://www.w3.org/TR/xmlschema-2/#rf-facets

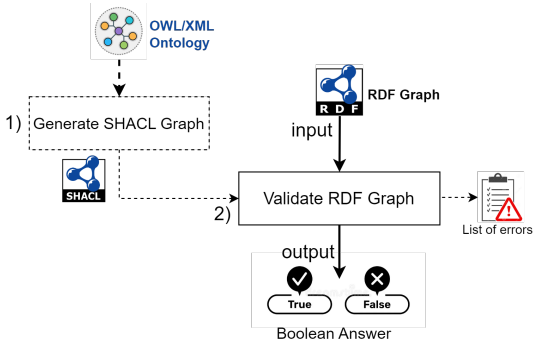

Figure 10: Steps for RDF Graph Validation

into an equivalent *SHACL* constraint using either *SHACL Core* or *SHACL-SPARQL* components, depending on the semantic complexity of the axiom. This process enables the construction of a domain-independent *SHACL* document, in which the syntactic structure of the *SHACL* constraints remains constant for each OWL axiom, varying only in the corresponding *Prefixes*, IRIs, and object property names involved in each generation process. In the case of error messages, these can be generated from predefined text templates concatenated with variables associated to the corresponding object properties. The generation of D employs the Turtle serialization format[9].

Table 2 presents the mapping of OWL object property characteristics into *SHACL* constraints.

Table 2: OWL to SHACL Mapping

Object Property Axiom	**SHACL Mapping**	**SHACL Type**
Functional	`sh:maxCount 1`	SHACL Core
Inverse-Functional	`sh:inversePath + sh:maxCount 1`	SHACL Core
Transitive	`sh:sparql` validating transitive closure	SHACL-Sparql
Symmetric	`sh:sparql` verifying reverse RDF triples	SHACL-Sparql
Asymmetric	`sh:sparql` detecting bidirectional relations	SHACL-Sparql
Reflexive	`sh:sparql` validating self-referential RDF triples	SHACL-Sparql
Irreflexive	`sh:sparql` detecting invalid self-relations	SHACL-Sparql

The following section illustrates three examples of *SHACL* constraints generated from the ontology and SPARQL queries described in Section 3.2.3. The examples, contextualized within the film industry domain, correspond to the following object properties: (1) *dbo:birthPlace* with the *Functional* axiom; (2) *dbp:father* with the *Inverse-Functional* axiom; and (3) *dbo:starring* with the *Symmetric* axiom.

Functional Object Property (*dbo:birthPlace*): This OWL axiom establishes that a subject may be associated with at most one object through a given property. To represent this restriction, a *NodeShape* is employed using *SHACL Core* components, specifically *sh:path* and *sh:maxCount*. This pattern enables the restriction of the maximum cardinality of the property to a single value. Listing 4 presents an example.

```
:FunctionalBirthPlaceShape
    a sh:NodeShape ;
    sh:targetSubjectsOf dbo:birthPlace ;
    sh:property [
        sh:path dbo:birthPlace ;
        sh:maxCount 1 ;
        sh:message "Incorrect query, the dbo:birthPlace property is functional" ; ] .
```

Listing 4: SHACL for functional property

In this example, an actor may have only one birthplace associated with the property *dbo:birthPlace*. For instance, Leonardo DiCaprio has only the value “Los Angeles, California” as his birthplace. Consequently, any SPARQL query constructed to return multiple values for this property will violate the semantic constraint defined in the *Shapes Graph*.

[9]https://www.w3.org/TR/turtle/

Inverse-Functional Object Property (*dbp:father*): This OWL axiom establishes that an object may be associated with at most one subject through a given property. To model this restriction, a *NodeShape* based on *SHACL Core* is employed, incorporating an *sh:inversePath* together with the cardinality restriction *sh:maxCount*. Listing 5 presents an example.

```
:InverseFunctionalFatherShape
    a sh:NodeShape ;
    sh:targetObjectsOf dbp:father ;
    sh:property [
        sh:path [ sh:inversePath dbp:father ] ;
        sh:maxCount 1 ;
        sh:message "Incorrect query, the dbp:father property is inverse-functional" ] .
```

Listing 5: SHACL for inverse-functional property

In this example, the property *dbp:father* is used to represent family relationships between actors and their fathers. For instance, George DiCaprio is the father of Leonardo DiCaprio. Due to the inverse-functional nature of this property, the same individual in the object position cannot be associated with multiple distinct subjects through the *dbp:father* relationship. Therefore, any SPARQL query constructed to return multiple values for this property will violate the semantic constraint defined in the *Shapes Graph*.

Symmetric Object Property (*dbo:starring*): This OWL axiom establishes that if a relationship exists between two RDF resources, then the inverse relationship using the same property must also exist. This type of restriction requires validations that cannot be represented solely through *SHACL Core*. For this reason, *SHACL-SPARQL* constraints are employed to verify the existence of RDF triples. Listing 6 presents an example.

```
:SymmetricStarringShape a sh:NodeShape ;
    sh:targetSubjectsOf dbo:starring ;
    sh:sparql [a sh:SPARQLConstraint ;
        sh:message "Incorrect query, the dbo:starring property is Symmetric.
        sh:select """SELECT ?this ?y WHERE {
                    ?this dbo:starring ?y .
                    FILTER NOT EXISTS { ?y dbo:starring ?this.}}""" ; ].
```

Listing 6: SHACL for symmetric property

In this case, if an RDF resource A is related to another RDF resource B through the property *dbo:starring*, then the inverse relationship between B and A must also exist. Consequently, any SPARQL query that omits the inverse RDF triple will violate the semantic constraint defined in the *Shapes Graph*.

Algorithm 7 illustrates the proposed approach for the automatic generation of the *Shapes Graph*. The function $RuleSHACL(property, axiom)$ encapsulates the OWL-to-SHACL transformation logic and automatically generates the validation components, constraints, and error messages associated with each OWL axiom identified in the ontology, using the *SHACL* templates defined in Table 3.

Algorithm 7 SHACL Graph Generation

Require: O // Input: Domain Ontology O
1: $docSHACL = EMPTY$
2: $prefixList \leftarrow ExtractPrefixes(O) +$ "$sh:$"
3: /*Generate the prefix declaration*/
4: $docSHACL \leftarrow GeneratePrefix(prefixList)$
5: $listAxiom \leftarrow$ Extract all pairs $(property, axiom)$ $FROM$ O $WHERE$ $ObjectPropertyAxiom$ **in** { Functional, Inverse-Functional, Transitive, Symmetric, Asymmetric, Reflexive, Irreflexive }
6: **for** $(property, axiom)$ **in** $listAxiom$ **do**
7: $\quad shape \leftarrow RuleSHACL(property, axiom)$
8: $\quad docSHACL \mathrel{+}= shape$
9: **end for**
10: **return** $docSHACL$ //Output: SHACL document D

The W3C provides the official *SHACL* documentation[10], including the complete specification and a summary of the syntactic rules (*Summary of SHACL Syntax Rules*)[11]. Additionally, several online tools are available for validation testing and experimentation with *SHACL* constraints, such as *SHACL Playground*[12].

[10] https://www.w3.org/TR/shacl/
[11] https://www.w3.org/TR/shacl/#syntax-rules
[12] https://shacl.org/playground/

Table 3: SHACL Templates for OWL Object Property Axioms

OWL Axiom	**SHACL Target**	**SHACL Constraint**
Functional	`sh:targetSubjectsOf ?property`	`sh:path ?property ; sh:maxCount 1 ; sh:message`
Inverse-Functional	`sh:targetObjectsOf ?property`	`sh:path [sh:inversePath ?property] ; sh:maxCount 1` ; sh:message
Transitive	`sh:targetSubjectsOf ?property`	sh:select + `SELECT ?x ?z WHERE {` `?x ?property ?y .` `?y ?property ?z .` `FILTER NOT EXISTS {` `?x ?property ?z . }` } ; sh:message
Symmetric	`sh:targetSubjectsOf ?property`	sh:select + `SELECT ?this ?y WHERE {` `?this ?property ?y .` `FILTER NOT EXISTS {` `?y ?property ?this . }` } ; sh:message
Asymmetric	`sh:targetSubjectsOf ?property`	sh:select + `SELECT ?this ?y WHERE {` `?this ?property ?y .` `?y ?property ?this .` } ; sh:message
Reflexive	`sh:targetSubjectsOf ?property`	sh:select + `SELECT ?this WHERE {` `FILTER NOT EXISTS {` `?this ?property ?this . }` } ; sh:message
Irreflexive	`sh:targetSubjectsOf ?property`	sh:select + `SELECT ?this WHERE {` `?this ?property ?this .` } ; sh:message

Note: The variable `?property` represents the object property extracted from the input ontology

Step 2, Validate RDF Graph

To perform RDF Graph G validation we define a function Validate(G, D) that will be validated according to the conditions set in the document *SHACL D*. The output will be $True$ if it meets the conditions *SHACL* otherwise $False$. If the SPARQL query does not satisfy the conditions *SHACL* will return a warning message, e.g.:

```
TYPE ERROR: PropertyAxiom
DESCRIPTION: "Incorrect query, the dbp:father property is inverse functional".
```

Algorithm 8 shows our proposal for RDF Graph Validation.

Algorithm 8 RDF Graph Validation

Require: G, D //Input: RDF graph G and SHACL document D
1: $conforms, outputMessage \leftarrow Validate(G, D)$
2: $TF_{sem} \leftarrow conforms$ // True or False
3: $list_{error}$ += "PropertyAxiom" + output message
4: **return** $list_{error}, TF_{sem}$ //Output: list of errors and boolean expression.

3.3.5 Execute Query

In this last stage it is defined if the query is executed. If *Validation of Literal* and *RDF Graph Validation* return $True$ the SPARQL query will be executed, if it returns $False$ the error list (Algorithm 1) is printed. The list of errors contains the accumulation of all possible errors found in the query, so that it serves the user as a solution guide. Figure 11 shows the conditional evaluation that determines the process flow, allowing either the execution of the query or the reporting of detected errors.

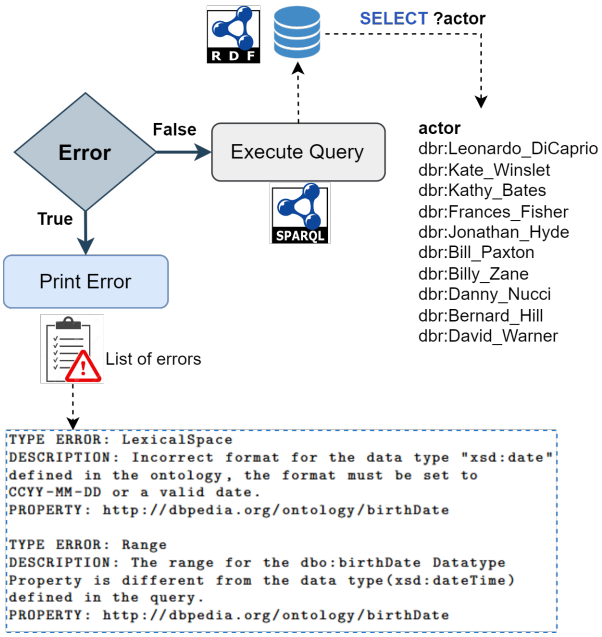

Figure 11: Validation-Based Query Execution Decision

3.4 Construction: Algorithm development

The construction activity corresponds to the implementation of the algorithms defined during the modeling stage. The implementation was carried out using Python 3.9.6, together with the RDFLib[13] library (version 7.0.0), an RDF framework accepted by the W3C, and the Owlready2 [44] module, which is oriented toward ontology-based programming. Development was conducted in the *Visual Studio Code* environment, enabling interactive execution of the syntactic and semantic validation processes, as well as the systematic collection of measurements required for experimental analysis. Figure 12 presents the algorithms developed during this stage.

During this phase, two types of tests are performed: (i) functional tests, designed to verify the expected behavior of each algorithm when faced with correct and incorrect SPARQL queries (according to the set of

[13]https://www.w3.org/2001/sw/wiki/RDFLib

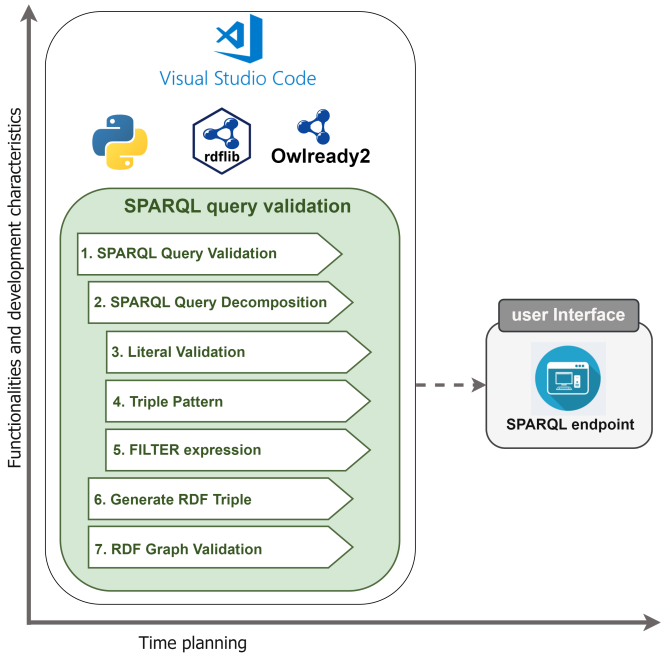

Figure 12: Algorithms developed in Python

queries defined in Section 3.2.3). For incorrect queries, it was verified that the errors reported were consistent with the errors classified in each case, while for correct queries, it was verified that the result obtained from the RDF dataset was as expected; and (ii) performance tests, aimed at evaluating the processing time associated with each query for RDF datasets of different sizes. In particular, it was verified that, as the volume of data increased, the behavior of the validator was consistent with what was expected; for example, that the execution with *SQV* validation remained stable, given that the proposal does not consider the instances of the dataset. These tests constitute the main input for the deployment stage, in which the experimental results are analyzed and interpreted.

For the performance tests, a computer with Windows 11 operating system, Intel(R) Core(TM) i5-1135G7 @ 2.40 GHz processor, 8 GB of main memory, and SSD was used.

3.5 Deployment: Experimental execution and analysis

The deployment activity corresponds to the experimental execution of the proposed method and the subsequent analysis of the results obtained. At this stage, the algorithms implemented during the construction phase are executed in order to assess their behavior and performance under different validation scenarios. For this purpose, a set of syntactically and semantically incorrect SPARQL queries, previously defined in Section 3.2.3, is used as test cases. The deployment activity and its outcomes are described in detail in the following section.

4 Experiments

In this section, time is considered the reference unit for evaluating the efficient use of resources in terms of performance. The processing time associated with each query for RDF sets of different sizes is analyzed, recording temporal variations and identifying possible trends associated with the growth in data volume. For each query, ten iterations were performed per variation in graph size, considering increments of 1000 triples up to a maximum of 16005 triples. Table 4 presents an extract corresponding to the iteration of the incorrect query $Q4$, where the validator *SQV* is applied.

Table 4: Execution time for syntactic validation of query $Q4$ on different datasets and 10 iterations

Iteration → Triples ↓	1	2	3	4	5	6	7	8	9	10	Mean
1000	0.3464	0.2854	0.2556	0.2396	0.2539	0.2534	0.2317	0.2382	0.2253	0.2224	0.2552
2000	0.246	0.2089	0.1731	0.1831	0.1824	0.1817	0.1898	0.1957	0.1733	0.1829	0.1917
3000	0.2975	0.2205	0.1979	0.1798	0.1867	0.1872	0.1845	0.1824	0.1774	0.1681	0.1982
4000	0.2773	0.2301	0.2182	0.1841	0.1729	0.1962	0.1819	0.1614	0.174	0.1762	0.1972
5000	0.2766	0.2027	0.2515	0.2162	0.2139	0.2012	0.2007	0.1812	0.1998	0.1609	0.2105
6000	0.3529	0.2277	0.2046	0.1934	0.1965	0.1805	0.1838	0.1912	0.1824	0.1962	0.2109
7000	0.3181	0.2038	0.2253	0.1986	0.2334	0.1984	0.1998	0.1719	0.1874	0.1788	0.2116
8000	0.2637	0.1976	0.1979	0.1731	0.1802	0.2291	0.1702	0.1981	0.19	0.1936	0.1993
9000	0.2987	0.2148	0.2232	0.1974	0.1962	0.1953	0.2029	0.2084	0.1843	0.1678	0.2089
10000	0.2711	0.2594	0.2494	0.2081	0.19	0.2246	0.1929	0.1688	0.1705	0.191	0.2126
11000	0.2437	0.2081	0.1841	0.1769	0.1719	0.1686	0.1647	0.1681	0.1993	0.1774	0.1863
12000	0.2892	0.2277	0.2029	0.1895	0.1698	0.1743	0.1786	0.1788	0.1931	0.1764	0.1980
13000	0.313	0.2646	0.2444	0.2191	0.212	0.2139	0.2053	0.2205	0.201	0.2103	0.2304
14000	0.3033	0.2282	0.196	0.2058	0.1957	0.1976	0.206	0.1979	0.2091	0.1798	0.2119
15000	0.3071	0.2394	0.2251	0.2203	0.212	0.1917	0.1752	0.1798	0.1824	0.1645	0.2098
16005	0.2978	0.2491	0.2172	0.231	0.2258	0.2379	0.2253	0.1764	0.1693	0.1724	0.2202

All times in milliseconds

Based on the set of queries defined above, five queries were selected for syntactic validation and five for semantic validation. The measurement compared the execution of each query with the *SQV* validator (Algorithm 1) and without prior validation. Tables 5 and 6 present the results obtained, showing the average and standard deviation of the time for *Query Validation* (with the *SQV* validator) and *Query Execution* (without validation). For reasons of space, only the values corresponding to 1000, 4000, 8000, 12000, and 16005 triples are reported.

The results derived from the deployment activity allow us to characterize the behavior of the method under different validation scenarios and provide empirical evidence regarding its performance.

In syntactic and semantic validation, only the SPARQL query and the ontology are used as input. Even if the ontology is complex, it is never too large; for *SQV* validation, ontology instances are not taken into account. On the other hand, an ontology is serialized in a specific format, such as OWL/XML, and stored in a plain text document, which is lightweight and easy to work with. Ontology instances are not necessarily included within the ontology itself, they may also reside in an RDF dataset or in RDF management systems [39].

4.1 Syntactic Validation

In this part, the validity of the Literal is checked. Examples and error analysis are presented below for: *Datatype, Lexical Form y Constraining Facets.*

In $Q4$, the *range* of the *Datatype Property dbo:birthDate* is verified. The *range* defined in the ontology is *xsd:date*, therefore the datatype *xsd:dateTime* is not a valid data type; thus, the datatype of the Literal is incompatible with the ontology and the generated error is of type "*range*". $Q5$ presents a *"LexicalSpace"* error, the string *Lexical Form* is not an appropriate value for the datatype *xsd:date*; the correct format is *CCYY-MM-DD*. $Q6$ has a *"Datatype"* error *xsd:char* is not a data type recommended by the W3C for use in ontologies. In $Q7$, *dbo:birthDate* is defined with a restriction: dates cannot be earlier than "1900-01-01", therefore a "$Facet$" type error occurs. Lastly, to validate a Literal, it must comply with a well-formed triple pattern, in the form $tp{:}< s, p, l >$, where s is the subject, p is a *Datatype Property*, and l is a Literal. In $Q12$, the property *dbo:director* is an Object Property, therefore it does not satisfy tp.

Table 5 presents the results for the syntactic validation. The column "Query Validation" shows the results of the experiment applying SQV, with the highest time being 0.4508 milliseconds (ms). On the other hand, the execution time of the query without validation ("Query Execution") shows higher times, ranging from 13.6 to 68.7 ms. These times include the parsing of the SPARQL query.

Table 5: Runtime for syntactic evaluation of SPARQL queries

Query	Datatype Property	Error Type	Triples	Query Validation*	Query Execution**
Q4	dbo:birthDate (total 1282)	Range	1000	0.2552±0.0370	36.4±21.3552
			4000	0.1972±0.0353	50.1±43.7250
			8000	0.1993±0.0280	23.6±9.1068
			16005	0.2202±0.0396	31.4±20.7000
Q5	dbo:birthDate (total 1282)	LexicalSpace	1000	0.2670±0.0310	25.3±8.0010
			4000	0.2587±0.0410	32.1±14.5789
			8000	0.2697±0.0348	27.9±9.8370
			16005	0.3522±0.0328	33.9±12.8621
Q6	dbo:birthName (total 912)	Datatype	1000	0.2194±0.0475	18.0±4.6667
			4000	0.2239±0.0349	25.2±11.6600
			8000	0.2113±0.0406	28.5±15.8272
			16005	0.2560±0.0638	25.7±14.3608
Q7	dbo:birthDate (total 1282)	Facet	1000	0.2255±0.0420	30.5±20.5278
			4000	0.2345±0.0448	22.7±10.9245
			8000	0.2323±0.0362	13.6±2.4129
			16005	0.2445±0.0320	36.6±10.3623
Q12	dbo:director (total 838)	PropertyType	1000	0.4182±0.1035	17.7±4.6916
			4000	0.3857±0.0704	20.2±12.3180
			8000	0.4508±0.0645	30.7±11.5282
			16005	0.4351±0.3680	68.7±16.6336

*Time (in Milliseconds) **Time Without Validation (in Milliseconds)

Figure 13 presents a comparison using a bar chart, where the X axis represents the label pairs Qx–Ty, with Qx denoting the SPARQL query ($Q4$, $Q5$, $Q6$, $Q7$, and $Q12$) and Ty the number of triples in the dataset over which the query was executed. The Y axis shows the execution time in milliseconds, using a logarithmic scale. "Query execution time" represents the average time it takes to execute the incorrect SPARQL query on the dataset. "Query validation time" represents the time taken by SQV to syntactically validate the query before execution. The black bar indicates the standard deviation of the measured time across 10 iterations of the experiment. The validation time is consistently low, much lower than the execution time, and remains very stable regardless of the dataset size. This is because syntactic validation is not affected by data volume, as it only analyzes the structure of the query and not its result over the data.

4.2 Semantic Validation

This section verifies that the SPARQL query is consistent with the object property axioms defined in the ontology.

Table 6 presents the results for semantic validation, where the semantic validation times are higher than those of syntactic validation. This is because validating object property axioms involves the use of a *SHACL* document, and therefore the time includes the overhead of processing this file. The time range when using the SQV validator is from 2.3504 to 6.2343 ms. In queries $Q14$ and $Q15$, the execution times without using

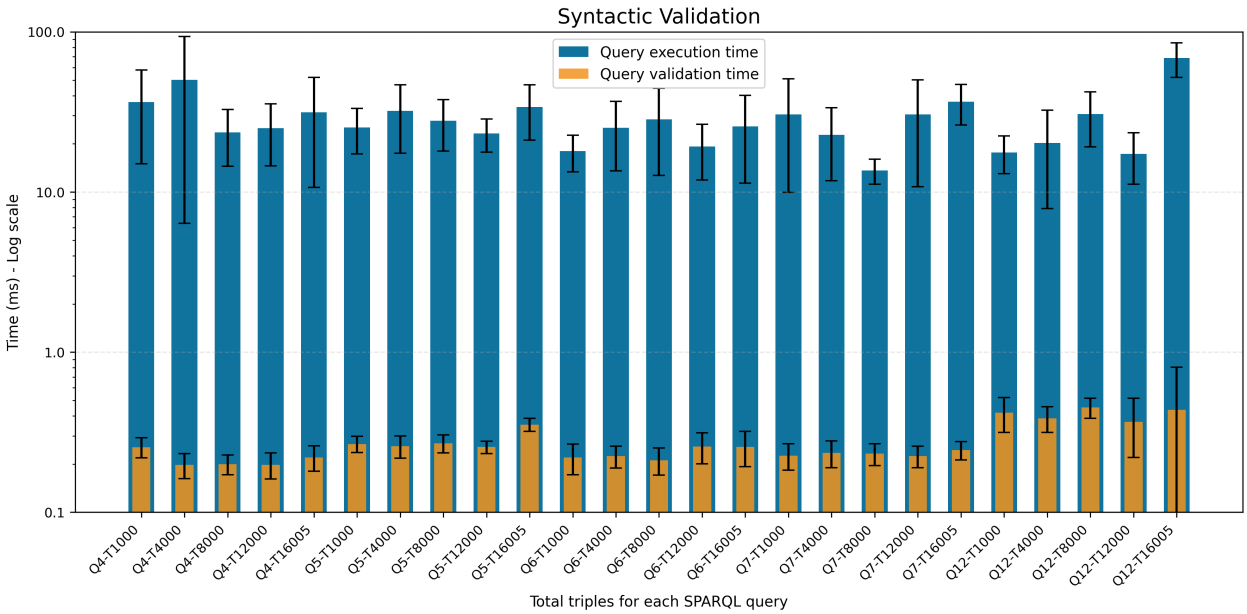

Figure 13: Average and standard deviation of 10 iterations per query

the validator are high due to the number of times the property appears in the dataset. For example, in $Q15$, the property *dbo:starring* appears 4032 times, unlike *dbp:children*, which appears only 28 times in the dataset ($Q16$). In this case, the execution time of the incorrect query is close to the times observed when applying validation.

Table 6: Runtime for semantic evaluation of SPARQL queries

Query	Object Property	Axiom	Triples	Query Validation*	Query Execution**
Q13	dbp:father (total 79)	inverse-functional	1000	2.5293±0.2970	4.2324±0.4373
			4000	2.3504±0.0336	4.7739±0.2645
			8000	2.4870±0.1855	5.7454±0.4713
			16005	2.5330±0.0541	7.9638±0.2893
Q14	dbo:birthPlace (total 1815)	functional	1000	2.5962±0.2294	9.3165±0.4323
			4000	2.6708±0.1374	27.7736±0.4124
			8000	2.7582±0.0912	58.5130±0.8071
			16005	2.7629±0.1149	142.1356±0.9016
Q15	dbo:starring (total 4032)	symmetric	1000	5.7305±0.1655	19.5504±0.4936
			4000	6.1236±0.5309	69.9447±1.4239
			8000	6.0315±0.0960	136.3465±0.9027
			16005	6.2343±0.4582	150.6631±1.8008
Q16	dbp:children (total 28)	asymmetric	1000	4.1629±0.7039	2.9953±0.5487
			4000	3.8813±0.1345	3.2359±0.4181
			8000	3.9311±0.2380	3.5219±0.1499
			16005	3.9358±0.1951	4.3246±0.2512
Q17	dbo:country (total 461)	functional	1000	2.5378±0.2664	5.5553±0.3918
			4000	2.5086±0.0522	5.8907±0.3302
			8000	2.5126±0.0919	13.9836±0.2469
			16005	2.8305±0.3127	25.6108±0.7899

*Time (in Milliseconds) **Time Without Validation (in Milliseconds)

Figure 14 presents a comparison using a bar chart. In general, the execution time of the incorrect query ("Query execution time") increases with the number of triples (from T1000 to T16005), which is particularly noticeable in Q13, Q14, Q15, and Q17. Queries Q14 and Q15, especially with larger dataset sizes, show very high execution peak, over 100 ms on a logarithmic scale, suggesting that these queries are more complex or involve more costly operations. In most cases, semantic validation ("Query validation time") represents only a small fraction of the total time; even in the most expensive cases, it does not exceed 10 ms on

a logarithmic scale. The semantic validation time of SQV, per query, is stable and significantly lower compared to the execution time of the incorrect query. The complexity of the query and the amount of data significantly influence execution times more than validation times. These results support the conclusion that prior semantic validation is efficient

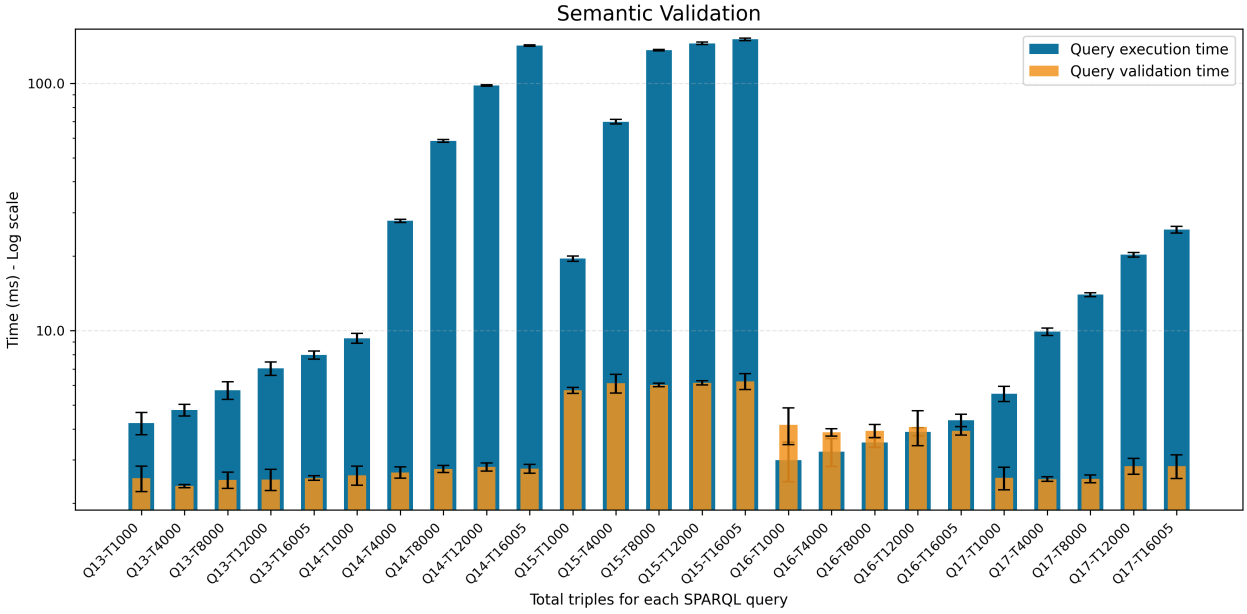

Figure 14: Average and standard deviation of 10 iterations per query

4.3 Comparative analysis

This section presents a comparative analysis of the results obtained by iterating query $Q1$ ten times under three experimental scenarios: (i) executing the incorrect query with the SQV validator (*SQV Time*); (ii) executing the incorrect query without validation (*Incorrect Query w/o SQV*); and (iii) executing the correct query (*Corrected Query w/o SQV*). In $Q1$, both the syntactic aspect is validated, due to the presence of a literal, and the semantic aspect, given that *dbp:father* is an Object Property (P_O) *Inverse-Functional* (Ifu). The incorrect part is manifested in the expression `FILTER` *(?x1!=?x2)*, therefore $Q1$ is considered an incorrect query. To evaluate its correct variant, the condition was adjusted to *(?x1=?x2)*, since if P_O = *Ifu*(*dbp:father*) and $x1$ *dbp:father* y and $x2$ *dbp:father* y are true, then $x1 = x2$. In summary, $Q1$ requires syntactic and semantic validation.

Table 7 shows the times measured for the three scenarios. Two additional columns are also included: *Saved Time* $(b - a)$, which corresponds to the saved time by not executing incorrect queries; and *Execution Penalty* $(a + c)$, which indicates the penalty cost of executing the correct query, after first passing through the validator. The time savings occur because an incorrect query is not executed: the SQV validator stops execution when it detects the error and, therefore, does not incur additional runtime costs during parsing. Conversely, in the case of correct queries, validation introduces additional time, albeit minimal.

As can be seen, the three times, validation, incorrect execution, and correct execution, increase progressively according to the number of triples, as do the metrics *Saved Time* and *Execution Penalty*.

In order to illustrate the behavior of the method, case $Q1$ with 16005 triples is analyzed. Figure 15 shows that $a = 2.5330$ ms corresponds to the SQV validation time, $b = 7.9638$ ms to the execution time without validation of the incorrect query, and $c = 27.4159$ ms to the execution time of the correct query. At this point, the time savings is 5.4308 ms $(b - a)$, while the execution penalty reaches 29.9489 ms $(a + c)$.

Although the use of SQV slightly increases the execution time for correct queries, it represents a favorable trade-off by reducing manual intervention during query processing, since the programmer will have to find the query problem through trial and error. This compromise between computational efficiency and man-hour savings allows tasks to be automated, thus improving the overall productivity of the process. Despite this, we believe it is better to check a query before execution. Furthermore, as the size of the dataset increases, the *Saved Time* increases, suggesting that the benefit of avoiding incorrect executions becomes more significant for larger data volumes. For its part, the Execution Penalty is minimal, since the validation process operates

Table 7: Comparison of executing time of successful and unsuccessful queries (Query Q1)

Triples	**SQV Time** (a)	**Incorrect Query w/o SQV** (b)	**Corrected Query w/o SQV** (c)	**Saved Time** ($b-a$)	**Execution Penalty** ($a+c$)
1000	2.5293	4.2324	5.0503	1.7031	7.5796
2000	2.4126	4.4804	6.7774	2.0678	9.1900
3000	2.4745	4.2942	6.9901	1.8197	9.4646
4000	2.3504	4.7739	8.5857	2.4235	10.9361
5000	2.5199	5.3660	9.3232	2.8461	11.8431
6000	2.3985	5.2269	9.9004	2.8284	12.2989
7000	2.7144	5.2108	10.5848	2.4964	13.2992
8000	2.4870	5.7460	12.4160	3.2590	14.9030
9000	2.3433	6.0869	15.5410	3.7436	17.8843
10000	2.4978	6.4550	15.7267	3.9572	18.2245
11000	2.5362	6.7438	16.1928	4.2076	18.7290
12000	2.4995	7.0255	21.7650	4.5260	24.2645
13000	2.5678	7.5658	21.3979	4.9980	23.9657
14000	2.5031	7.4052	22.9982	4.9021	25.5013
15000	2.7554	7.5242	23.3341	4.7688	26.0895
16005	2.5330	7.9638	27.4159	5.4308	29.9489

All times in milliseconds (ms)

exclusively on the ontology schema, and not on the RDF instances; therefore, the size of the dataset does not affect the application of the validation process.

Incorrect Query (Q1): $\begin{cases} \text{With SQV,} & a = 2.5330 \text{ ms.} \\ \text{Without SQV,} & b = 7.9638 \text{ ms.} \end{cases}$

Corrected Query (Q1'): $T_{SQV} = a + c, \quad T_{SQV} = 29.9489$ ms.

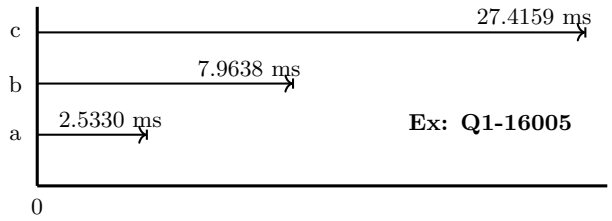

Figure 15: Comparative example of times:: Q1-16005

Figure 16 presents a comparison using a line graph, where the X axis represents the number of triples and the Y axis represents the execution time in milliseconds, using a logarithmic scale. On this type of scale, straight lines indicate a growth trend. The incorrect query shows an increase in time, but with a gentler slope. In contrast, the correct query has a steeper upward slope, suggesting that query time grows exponentially as the number of triples increases, due to the higher computational cost required. When applying validation with *SQV* ("Query validation"), the curve remains relatively flat, indicating low sensitivity to the size of the RDF graph, i.e., to the number of triples.

5 Discussion

From a conceptual perspective, data are representations of facts that, in the absence of context, lack meaning. When this data is interpreted within a domain, it acquires meaning and becomes information; and when that information is analyzed, integrated with experience, and used to recognize patterns, it is transformed into knowledge, forming a fundamental value chain for organizations [45]. In short, data generates information and information generates knowledge, enabling management and decision-making.

The knowledge derived from data underpins critical processes in public and private organizations, where database queries allow relevant information to be extracted for analysis and strategic decisions. In the field of semantic data, SPARQL queries fulfill this role; however, incorrect query formulation can lead to unexpected or erroneous results, directly affecting the quality of information and, therefore, decision-making. A recurring cause is that the programmer is unfamiliar with the ontological schema or domain, generating

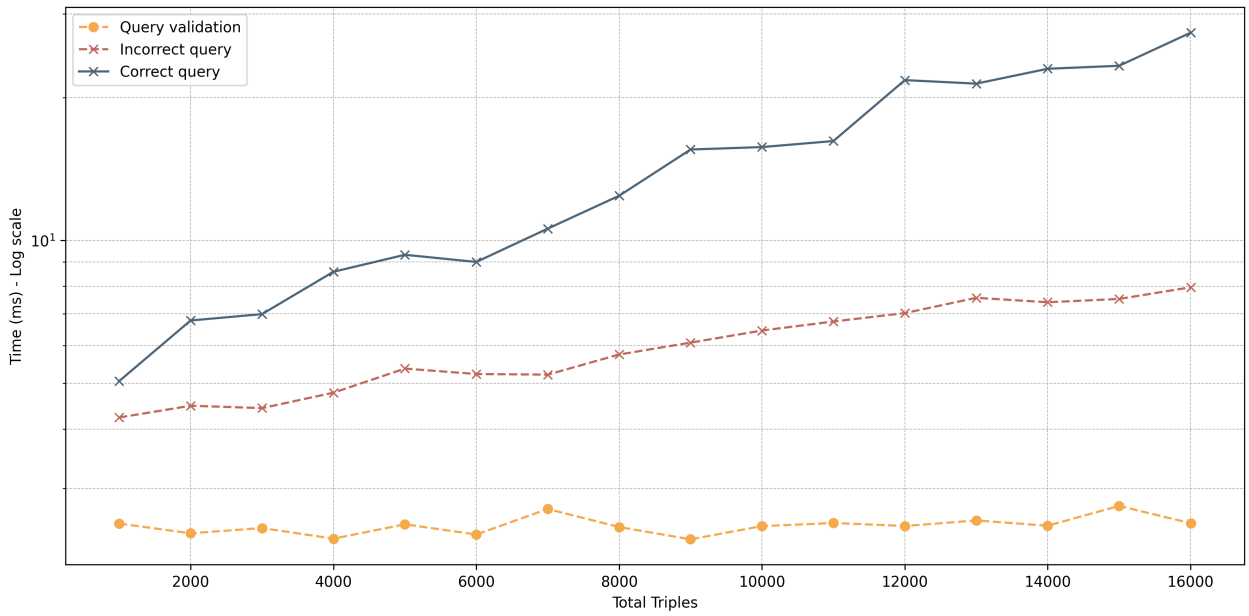

Figure 16: Comparison of correct and incorrect query

erroneous queries. Delivering incorrect information can have operational, financial, or strategic implications, as incorrect responses can harm management and planning processes.

Computational costs depend on multiple factors, such as the size of the dataset instantiated in the RDF management system, the complexity of the query due to the processing associated with parsing, the technological architecture of the server, the network infrastructure, the programming language, the programming paradigm used, and the degree of code optimization. These factors, combined with repeated executions motivated by a lack of clarity regarding the source of the error, can significantly increase development time, resource consumption, and programmer fatigue. Prior query validation helps mitigate this scenario by reducing unnecessary executions and promoting more efficient use of computational and energy resources.

Several approaches in the literature address specific aspects of SPARQL query validation; however, most of them do not explicitly consider the role of the ontology as a structuring model of the domain. To the best of our knowledge, the work by Almendros-Jiménez et al. [20, 21] is the closest proposal to the approach presented in this paper. Nevertheless, their method relies on the use of a reasoner and requires modifications to the ontological schema, thereby introducing additional processing steps and external dependencies. In contrast, the method proposed in this work operates directly on the base ontology, which is treated as a lightweight plain-text representation, and does not depend on third-party reasoning tools. This design choice simplifies integration into existing systems and reduces overall operational complexity.

Although the proposed approach can be extended by incorporating additional types of ontological axioms, it constitutes an important first step toward a more comprehensive solution suitable for integration into a SPARQL endpoint. The availability of an explicit list of detected errors, together with a pre-validation process, provides substantial support for developers by reducing trial-and-error cycles, lowering development effort, and promoting more efficient interaction with semantic data. At present, most existing SPARQL endpoints are primarily limited to syntactic validation [20], such as checking the correct placement of delimiters or query structure, while largely ignoring the domain knowledge encoded in the ontology. Incorporating semantic-level verification therefore represents a promising direction for enhancing the robustness and reliability of Semantic Web–based systems.

6 Conclusions

This paper presents an approach for validating SPARQL queries from both a syntactic and semantic perspective, enabling early detection of errors before execution. Given that users can make mistakes when formulating queries, which can lead to unexpected responses without a clear indication of the source of the problem, the method constitutes a preventive solution that helps reduce the trial and error cycle and avoids unnecessary executions on RDF repositories. The proposed approach, structured in five phases, incorporates practical examples that facilitate its understanding and application.

The experimental evaluation showed that the method introduces minimal computational cost, which is justified by the benefit of avoiding the execution of incorrect queries or those with semantic inconsistencies. In particular, it was observed that the validator stops the processing of incorrect queries, avoiding the execution cost associated with the runtime for parsing the query, while in the case of correct queries the additional penalty is minimal. As the size of the dataset increases, the saved time also increases, suggesting that prior validation becomes more advantageous in scenarios with larger datasets, where execution times and operational efficiency become more relevant. Although the time increase for correct queries is minimal, it indicates a favorable trade-off between computational cost and reduction of manual tasks, which improves the productivity of the semantic data development and exploitation process.

Overall, the results confirm that prior query validation constitutes a key component for improving the quality, reliability, and operational efficiency of systems that rely on SPARQL queries over semantic datasets. By preventing the execution of incorrect or inconsistent queries, the proposed approach not only enhances system robustness but also has the potential to reduce unnecessary computational overhead. As a consequence, this reduction may contribute favorably to lowering the energy consumption demanded by current Semantic Web applications, aligning with sustainability-oriented computing practices. Furthermore, these findings open a promising line of research toward enriching the Semantic Web tool ecosystem with validation mechanisms that support both reliable and energy-efficient data processing.

Future work will expand the validation process by incorporating additional types of ontological axioms, including class expression axioms such as SubClassOf, EquivalentClasses, DisjointClasses, and DisjointUnion. Furthermore, future research will focus on integrating the validation process into a SPARQL endpoint capable of verifying the correctness of queries before execution, enabling seamless and efficient validation within real-world Semantic Web systems.

Acknowledgment

This study was funded by PROCIENCIA, Perú under grant agreement no. PE501083407-2023-PROCIENCIA and Regular Fondecyt No. 1250344, ANID, Chile.

References

[1] J. Collio, A. Aguilera, and I. Dongo, "Efficient method for validating sparql queries using semantic web ontologies," in *2025 LI Latin American Computer Conference (CLEI)*, 2025, pp. 1–10.

[2] A. Patel and S. Jain, "Present and future of semantic web technologies: A research statement," *International Journal of Computers and Applications*, vol. 43, no. 5, pp. 413–422, 2021.

[3] A. Loizou, R. Angles, and P. Groth, "On the formulation of performant SPARQL queries," *Journal of Web Semantics*, vol. 31, pp. 1–26, 2015.

[4] J. Almendros-Jiménez and A. Becerra-Terón, "Debugging of wrong and missing answers in SPARQL," in *18th International Symposium on Database Programming Languages*. NY, USA: ACM, 2021, p. 7–16.

[5] K. Kim, B. Moon, and H.-J. Kim, "R3F: RDF triple filtering method for efficient SPARQL query processing," *World Wide Web*, vol. 18, no. 2, pp. 317–357, Mar 2015.

[6] Y. Amsterdamer and Y. Callen, "Interactive SPARQL query formulation using provenance," *Knowledge and Information Systems*, vol. 66, no. 3, pp. 2165–2191, Mar 2024.

[7] S. Subhashree and P. S. Kumar, "Augmenting linked data ontologies with new object properties," *New Generation Computing*, vol. 38, no. 1, pp. 125–152, Mar 2020.

[8] B. Lakzaei and M. Shamsfard, "Ontology learning from relational databases," *Information Sciences*, vol. 577, pp. 280–297, 2021.

[9] H. Silvennoinen, A. Chadzynski, F. Farazi, A. Grišiūtė, Z. Shi, A. von Richthofen, S. Cairns, M. Kraft, M. Raubal, and P. Herthogs, "A semantic web approach to land use regulations in urban planning: The OntoZoning ontology of zones, land uses and programmes for Singapore," *Journal of Urban Management*, vol. 12, no. 2, pp. 151–167, 2023.

[10] T. Berners-Lee, J. Hendler, and O. Lassila, *"The Semantic Web: A New Form of Web Content that is Meaningful to Computers will Unleash a Revolution of New Possibilities"*, 1st ed. New York, NY, USA: Association for Computing Machinery, 2023, p. 91–103.

[11] O. Lassila and J. Hendler, "Embracing 'Web 3.0'," *IEEE Internet Computing*, vol. 11, no. 3, pp. 90–93, 2007.

[12] D. Wu, H.-T. Wang, and A. U. Tansel, "A survey for managing temporal data in RDF," *Information Systems*, vol. 122, p. 102368, 2024.

[13] L. Parkin, B. Chardin, S. Jean, and A. Hadjali, "Explaining unexpected answers of SPARQL queries," in *Web Information Systems Engineering*. Cham: Springer International Publishing, 2022, pp. 136–151.

[14] O. Curé and G. Blin, *"RDF Database Systems: Triples Storage and SPARQL Query Processing"'*, 1st ed. USA: Elsevier, 2015.

[15] O. H. Jader, S. Zeebaree, and R. R. Zebari, "A state of art survey for web server performance measurement and load balancing mechanisms," *International Journal of Scientific & Technology Research*, vol. 8, no. 12, pp. 535–543, 2019.

[16] R. Pressman, *"Ingeniería del software un enfoque práctico"*, 7th ed. McGRAW-HILL, 2010.

[17] E. Guzel Kalayci, T. E. Kalayci, and D. Birant, "An ant colony optimisation approach for optimising SPARQL queries by reordering triple patterns," *Information Systems*, vol. 50, pp. 51–68, 2015.

[18] C. Liu, H. Wang, Y. Yu, and L. Xu, "Towards efficient SPARQL query processing on RDF data," *Tsinghua Science & Technology*, vol. 15, no. 6, pp. 613–622, 2010.

[19] S. Saharan, J. Lather, and R. Radhakrishnan, "Optimisation using rearrangement of order of BGP and TLBO approach," *Procedia Computer Science*, vol. 125, pp. 282–289, 2018, the 6th International Conference on Smart Computing and Communications.

[20] J. M. Almendros-Jiménez, A. Becerra-Terón, and A. Cuzzocrea, "Syntactic and semantic validation of SPARQL queries," in *Symposium on Applied Computing*. NY, USA: ACM, 2017, p. 349–352.

[21] J. M. Almendros-Jiménez and A. Becerra-Terón, "Discovery and diagnosis of wrong SPARQL queries with ontology and constraint reasoning," *Expert Systems with Applications*, vol. 165, p. 113772, 2021.

[22] I. Dongo, Y. Cardinale, and R. Chbeir, "RDF-F: RDF datatype inferring framework," *Data Science and Engineering*, vol. 3, no. 2, pp. 115–135, Jun 2018.

[23] K. Abicht, "OWL reasoners still useable in 2023," 2023, arXiv:2309.06888.

[24] A. N. Lam, B. Elvesæter, and F. Martin-Recuerda, "A performance evaluation of OWL 2 DL reasoners using ORE 2015 and very large bio ontologies," in *1st International Workshop on Data Management for Knowledge Graphs*. RWTH Aachen University, 2023.

[25] A. Gashkov, A. Perevalov, M. Eltsova, and A. Both, "Improving question answering quality through language feature-based SPARQL query candidate validation," in *The Semantic Web*. Cham: Springer International Publishing, 2022, pp. 217–235.

[26] Z. B. Ozger and N. Y. Uslu, "An effective discrete artificial bee colony based SPARQL query path optimization by reordering triples," *Journal of Computer Science and Technology*, vol. 36, no. 2, pp. 445–462, 2021.

[27] N. Guido, "On the Static Analysis for SPARQL Queries Using Modal Logic," in *IJCAI 2015*, Buenos Aires, Argentina, Jul. 2015.

[28] K. Rabbani, M. Lissandrini, and K. Hose, "SHACL and ShEx in the wild: A community survey on validating shapes generation and adoption," in *Companion Proceedings of the Web Conference*. NY, USA: ACM, 2022, p. 260–263.

[29] R. B. Thapa and M. Giese, "Optimizing SPARQL queries with SHACL," in *The Semantic Web – ISWC 2023*. Cham: Springer Nature Switzerland, 2023, pp. 41–60.

[30] J. Corman, F. Florenzano, J. L. Reutter, and O. Savkovic, "Validating SHACL constraints over a SPARQL endpoint," in *The Semantic Web – ISWC 2019*. Springer, 2019, pp. 145–163.

[31] A. Oudshoorn, M. Ortiz, and M. Šimkus, "SHACL validation in the presence of ontologies: Semantics and rewriting techniques," *Artificial Intelligence*, vol. 352, p. 104483, 2026.

[32] C. Wang, L. Zhang, and W. Yan, "Enhancement and validation of ifcOWL ontology based on shapes constraint language (SHACL)," *Automation in Construction*, vol. 160, p. 105293, 2024.

[33] J. Collio, A. Aguilera, and I. Dongo, "Toward an enrichment of ontologies inferred from RDF documents," *IEEE Access*, vol. 12, pp. 148 037–148 056, 2024.

[34] Y. M. Saber, H. Abdel-Galil, and M. A. E. F. Belal, "Arabic ontology extraction model from unstructured text," *Journal of King Saud University-Computer and Information Sciences*, vol. 34, no. 8, B, pp. 6066–6076, Sep 2022.

[35] G. Stumme, A. Hotho, and B. Berendt, "Semantic web mining: State of the art and future directions," *Journal of Web Semantics*, vol. 4, no. 2, pp. 124–143, 2006, semantic Grid –The Convergence of Technologies.

[36] W. hao Chen, Y. Cai, H. fung Leung, and Q. Li, "Generating ontologies with basic level concepts from folksonomies," *Procedia Computer Science*, vol. 1, no. 1, pp. 573–581, 2010, iCCS 2010.

[37] B. DuCharme, *"Learning SPARQL: querying and updating with SPARQL 1.1"*, 2nd ed. USA: O'Reilly Media, 2013.

[38] B. Ben Mahria, I. Chaker, and A. Zahi, "A novel approach for learning ontology from relational database: From the construction to the evaluation," *Journal of Big Data*, vol. 8, no. 1, pp. 1–22, Dec. 2021.

[39] K. Alaoui, "A categorization of RDF triplestores," in *SCA 19: 4th International Conference on Smart City Applications*. NY, USA: ACM, 2019, pp. 1–7.

[40] V. Senthooran, "Review of querying RDF techniques with SPARQL in semantic web technologies," *International Journal of Scientific & Engineering Research*, vol. 3, no. 10, pp. 36–38, 2015.

[41] F. Saïs, C. Pruski, and M. Da Silveira, "Inferring the evolution of ontology axioms from RDF data dynamics," in *9th Knowledge Capture Conference*, ser. K-CAP '17. NY, USA: ACM, 2017.

[42] A.-C. N. Ngomo, S. Auer, J. Lehmann, and A. Zaveri, *"Introduction to Linked Data and Its Lifecycle on the Web"*. Cham: Springer International Publishing, 2014, pp. 1–99.

[43] P. Pareti and G. Konstantinidis, *"A Review of SHACL: From Data Validation to Schema Reasoning for RDF Graphs"'*. Cham: Springer International Publishing, 2022, pp. 115–144.

[44] L. Jean-Baptiste, *Accessing ontologies in Python*. Berkeley, CA: Apress, 2021, pp. 83–112.

[45] D. International, *"DAMA Guia Para o Corpo de Conhecimento em Gerenciamento de Dados"*, 1st ed. Sao Pablo-Brasil: Technics Publications, 2012.