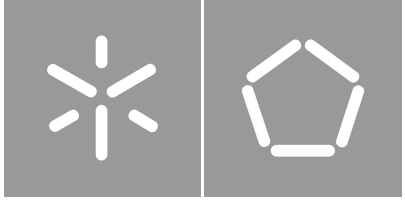
University of Minho
School of Engineering

Vitor Lelis Noronha Leite

OntoExpand

october 2025



University of Minho
School of Engineering

Vitor Lelis Noronha Leite

OntoExpand

Master's Dissertation in Informatics Engineering

Dissertation supervised by
José Carlos Leite Ramalho

october 2025

Copyright and Terms of Use for Third Party Work

License granted to users of this work:

Acknowledgements

With the completion of this thesis, I feel the need to take a moment to express my gratitude to the people who have supported and accompanied me throughout this journey. Their presence, encouragement, and guidance have made this achievement possible and meaningful.

First and foremost, I would like to express my gratitude to my supervisor, José Carlos Leite Ramalho, for his patience, guidance, and continuous support throughout this project. His knowledge and valuable insights have been instrumental in the completion of this thesis.

I am profoundly grateful to my family for always believing in me and providing the confidence and strength I needed, especially my mother, Ana Cecilia Lelis Noronha, whose support and encouragement have been invaluable. A very special acknowledgment goes to my partner, Mariana Brandão Nogueira Alves, the love of my life, whose motivation, love, and support have been my greatest source of inspiration and comfort whenever I needed it most.

Lastly, I would like to thank all my friends for the laughter, advice, and achievements we have shared along the way. Their companionship made this journey not only possible but also joyful and memorable.

Statement of Integrity

I hereby declare having conducted this academic work with integrity.

I confirm that I have not used plagiarism or any form of undue use of information or falsification of results along the process leading to its elaboration.

I further declare that I have fully acknowledged the Code of Ethical Conduct of the University of Minho.

University of Minho, Braga, october 2025

Assinado por: **Vitor Lelis Noronha Leite**
Num. de Identificação: PASBR-YE
Data: 2025.10.26 15:44:54 +0000

Vitor Lelis Noronha Leite

Abstract

The growing complexity and volume of OWL ontologies in the Semantic Web present significant challenges for reasoning and knowledge expansion. Traditional ontology reasoners often incur high computational costs and struggle to scale efficiently, limiting their suitability for real-time and large-scale applications. This thesis introduces **OntoExpand**, a novel approach that leverages **SPARQL CONSTRUCT** queries to replicate core reasoning tasks, offering an efficient and SPARQL-native alternative to conventional reasoning engines. By translating OWL axioms into executable queries, OntoExpand generates inferred triples that are seamlessly reintegrated into the ontology through **SPARQL INSERT** operations, enabling incremental and transparent knowledge enrichment.

The work was carried out in two phases. The first focused on the theoretical formulation of reasoning patterns in SPARQL, covering inference, property definitions, and consistency checking. The second phase extended this exploration into practice through the development of a web-based platform, built with modern web technologies and integrated with GraphDB. This platform allows users to inspect, expand, and validate ontologies dynamically, providing an accessible interface that abstracts the complexity of SPARQL while eliminating the need for external reasoners.

While limitations remain, particularly regarding SPARQL's expressiveness and reliance on GraphDB configuration, the results indicate that this approach offers a lightweight, transparent, and extensible mechanism for reasoning and ontology expansion. Beyond its immediate contributions, the project lays the groundwork for future enhancements, such as extended rule coverage, broader triple store compatibility, and potential adoption in educational and real-world Semantic Web applications.

Keywords OWL Ontologies, SPARQL, Reasoning, Knowledge Expansion, Inference, Semantic Web

Resumo

O crescente volume e a complexidade das ontologias OWL na Web Semântica apresentam desafios significativos para o *reasoning* e a expansão do conhecimento. Os *reasoners* tradicionais frequentemente acarretam altos custos computacionais e têm dificuldades em escalar de forma eficiente, limitando sua aplicabilidade em cenários de larga escala e em tempo real. Esta dissertação apresenta o **OntoExpand**, uma abordagem inovadora que utiliza *queries* **SPARQL CONSTRUCT** para replicar tarefas centrais de *reasoning*, oferecendo uma alternativa eficiente e nativa em SPARQL aos mecanismos convencionais. Por meio da tradução de axiomas OWL em *queries* executáveis, o OntoExpand gera triplas inferidas que são reintegradas incrementalmente à ontologia através de operações **SPARQL INSERT**, promovendo o enriquecimento transparente do conhecimento.

O trabalho foi conduzido em duas fases. A primeira concentrou-se na formulação teórica de padrões de *reasoning* em SPARQL, incluindo inferência, definição de propriedades e verificação de consistência. A segunda fase levou essa exploração à prática por meio do desenvolvimento de uma plataforma web, construída com tecnologias modernas e integrada ao GraphDB. Essa plataforma permite inspecionar, expandir e validar ontologias de forma dinâmica, oferecendo uma interface acessível que abstrai a complexidade do SPARQL e elimina a necessidade de *reasoners* externos.

Embora limitações permaneçam, em especial quanto ao poder de expressão do SPARQL e à dependência da configuração do GraphDB, os resultados indicam que esta abordagem oferece um mecanismo leve, transparente e extensível para *reasoning* e expansão de ontologias. Além de suas contribuições imediatas, o projeto estabelece as bases para aprimoramentos futuros, como a ampliação da cobertura de regras, maior compatibilidade com diferentes triple stores e potencial adoção em contextos educacionais e aplicações reais da Web Semântica.

Palavras-chave OWL Ontologies, SPARQL, *Reasoning*, Knowledge Expansion, Inference, Web Semântica

Contents

List of Figures

List of Tables

Listings

Chapter 1

Introduction

This chapter introduces the context and motivation for the OntoExpand project, highlighting the challenges associated with maintaining and expanding ontologies in dynamic, data-rich environments such as the Semantic Web. It discusses the limitations of traditional reasoning tools, particularly their computational cost and inefficiency when handling large or frequently updated datasets. The chapter also presents the main aims of the project, focusing on the development of a system that leverages SPARQL `CONSTRUCT` and `INSERT` queries to enable efficient, incremental ontology expansion. Additionally, it outlines the creation of a web application that facilitates real-time knowledge inference and ontology management, providing a user-friendly interface for interacting with evolving ontologies.

1.1 Context and Motivation

In knowledge representation, **ontologies** (Ontotext, 2025a) are defined to provide a formal structure for concepts and their relationships of a specific domain. These structured representations allow systems to model domain knowledge in a way that is both human-readable and machine-processable. Ontologies are particularly important in domains such as artificial intelligence, data integration, and the **Semantic Web** (Dhulekar & Devrankar, 2020), where data from different sources must be linked and interpreted consistently.

As the amount of available data continues to grow, particularly in the context of the Semantic Web, maintaining and expanding ontologies becomes an increasingly complex task. New knowledge is constantly being introduced, which can rapidly lead to ontologies becoming outdated or inconsistent. The task of maintaining these ontologies and ensuring that they remain current with new information is critical. However, the process of expanding an ontology traditionally requires the use of external **reasoners** (Mishra & Kumar, 2011) to infer new data based on existing axioms and relations. While reasoners are effective in drawing logical conclusions, they can become computationally expensive and inefficient,

particularly when handling large-scale datasets or when real-time updates are necessary.

These challenges are particularly prominent in dynamic, data-rich environments such as the Semantic Web, where ontologies need to be frequently updated and expanded as new data arrives. Traditional reasoners typically process ontologies as a whole, which can result in significant delays and high resource consumption. Consequently, there is a growing need for alternative solutions that are capable of efficiently managing the growth and expansion of ontologies without the drawbacks associated with traditional reasoning mechanisms.

To address these challenges, OntoExpand proposes an alternative approach that leverages **SPARQL (SPARQL Protocol and RDF Query Language) CONSTRUCT and INSERT queries** (W3C, 2013) to efficiently map ontological axioms and facilitate data inference. Unlike traditional reasoners, which typically process the entire ontology in a single step without checking, OntoExpand employs SPARQL CONSTRUCT queries to identify and generate new axioms, and subsequently uses INSERT queries to directly add the inferred triples into the knowledge base. This integration of reasoning and querying enables targeted, incremental ontology expansion, improving both scalability and efficiency while eliminating the need for external reasoning tools.

1.2 Main Aims

The main objective of this project is to develop an efficient engine for expanding ontologies using SPARQL `CONSTRUCT` queries. The initial phase of the project focuses on mapping ontological axioms into SPARQL queries, enabling the inference of new knowledge in a way that mimics traditional reasoning mechanisms but with improved performance. This mapping allows for real-time derivation of new facts and relationships, providing the flexibility needed to scale ontologies effectively.

To support this functionality, a web application was developed that automatically performs reasoning over ontologies using SPARQL `CONSTRUCT` queries. The application dynamically applies reasoning rules and uses SPARQL `INSERT` operations to update the ontology with newly inferred triples. Users can interact with the ontology through an intuitive interface that showcases these updates in real time, without requiring manual query writing.

Ultimately, the goal of OntoExpand is to provide a solution for ontology expansion that is efficient, scalable, and independent of external reasoners. By enabling real-time knowledge extraction and dynamic ontology management through a user-friendly web interface, OntoExpand supports a wide range of domains—particularly within the Semantic Web—where rapid updates and automated inference are essential

for maintaining evolving information systems.

1.3 Summary

The Introduction chapter presented the context and motivation behind the OntoExpand project, emphasizing the challenges of maintaining and expanding ontologies in dynamic, data-rich environments such as the Semantic Web. It highlighted the limitations of traditional reasoning tools, particularly regarding computational efficiency and scalability, and introduced the main objectives of the project. Specifically, OntoExpand leverages SPARQL `CONSTRUCT` and `INSERT` queries to enable incremental, real-time ontology expansion. The chapter also described the development of a web application that facilitates interaction with ontologies, enabling users to perform automated reasoning without relying on external reasoners. This foundation sets the stage for a deeper exploration of existing technologies and approaches, which is addressed in the next chapter.

Chapter 2

State of the Art

This chapter reviews the state of the art in ontology representation, querying, and reasoning. It introduces the fundamental concepts underpinning the Semantic Web, including ontologies, RDF, RDFS, OWL, SPARQL, and reasoners, and explains how these technologies interconnect to enable structured knowledge representation and inference. The chapter also examines current approaches for querying and reasoning over ontologies, highlighting SPARQL-based methods, recursive reasoning frameworks, and stream reasoning with C-SPARQL. Finally, it discusses the limitations of existing techniques in terms of performance, scalability, and real-time support, thereby motivating the need for OntoExpand's SPARQL-driven reasoning approach.

2.1 Background

The Semantic Web aims to provide a framework for sharing and linking data in a machine-readable way. Central to this vision are **ontologies, the Resource Description Framework (RDF), the Web Ontology Language (OWL), SPARQL, and reasoners,** which together enable the representation, querying, and inference of knowledge. This section explores these foundational technologies and their interconnections.

2.1.1 Ontologies

Ontologies are formal representations of knowledge within a specific domain. They're responsible to set concepts, relationships and rules of the information. Those representations enable a common ground of understanding between humans and machines, facilitating data processing and standardization. An ontology typically consists of:

- **Classes**: Representing abstract concepts or categories (e.g., `Person`, `Organization`, `Employee`,

`Manager`).

- **Attributes and Relations**: Defining connections between concepts (e.g., `hasAge`, `worksAt`, `knows`, `manages`).
- **Individuals**: Specific instances of classes (e.g., `Jim`, `Michael`, `Dunder Mifflin`).

By connecting those concepts, it is possible to create a graph-like structure like the following example:

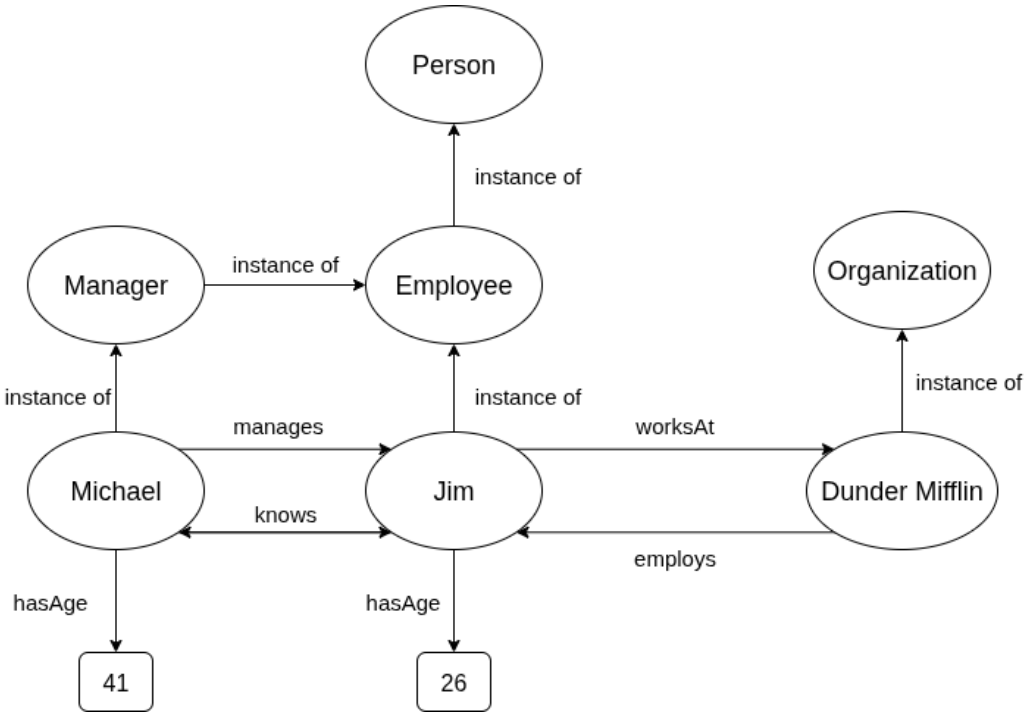


Figure 1: Ontology Structure Example

Ontologies play a key role in knowledge representation and reasoning, providing the vocabulary for describing and linking data in a structured manner. As illustrated in Figure 1, when properly specified, ontologies create a data model called **knowledge graph** (Ontotext, 2025b). A knowledge graph organizes information as a collection of entities (represented as nodes) and the relationships between them (represented as edges).

2.1.2 Resource Description Framework (RDF)

The Resource Description Framework (RDF) is the standard model for representing information on the web. RDF encodes data as **triples** `<subject, predicate, object>`, forming a directed labeled graph. This flexible structure allows for the representation of diverse types of data and relationships. For example:

```
<Jim> <hasAge> "26" .
<Jim> <worksAt> <Dunder Mifflin> .
```

Listing 2.1: RDF Example

In addition to triples, the RDF introduces the concept of **URI (Uniform Resource Identifier)**. URIs are used to uniquely identify resources, ensuring that every resource can be correctly referenced.

While RDF provides a great base model for representing data, RDF Schema (RDFS) (W3C, 2014) extends RDF by introducing a new vocabulary for defining classes, properties, and relationships among resources. RDFS enables more structured and meaningful data representation by adding the following key components:

- **Classes and Instances:** Resources can be categorized into classes using `rdfs:Class`. For example:

```
<Person> rdf:type rdfs:Class.
<Jim> rdf:type <Person>.
```

Listing 2.2: RDF Classes and Instances

Here, `Person` is declared as a class, and `Jim` is an instance of that class.

- **Properties and Domains/Ranges:** RDFS allows defining properties and specifying the types of subjects and objects they apply to using `rdfs:domain` and `rdfs:range`.

```
<hasAge> rdf:type rdf:Property.
<hasAge> rdfs:domain <Person>.
<hasAge> rdfs:range xsd:integer.
```

Listing 2.3: RDF Property Domain/Range

This means that `hasAge` is a property that applies to `Person` instances and expects an integer value.

- **Subclass and Subproperty Relationships:** Using `rdfs:subClassOf` and `rdfs:subPropertyOf`, RDFS enables hierarchy formation.

```
<Employee> rdfs:subClassOf <Person>.
<manages> rdfs:subPropertyOf <knows>.
```

Listing 2.4: RDF Subclass and Subproperty

This declares that an `Employee` is a specialized form of a `Person`, and `manages` is a more specific form of the `knows` relationship.

By integrating RDF with RDFS, it is possible to enrich the semantics and inference capabilities to data, making it more machine-readable and interoperable.

2.1.3 Web Ontology Language (OWL)

Although RDF is both efficient and easy to comprehend, its structure is not always well-suited to representing complex use cases of the Semantic Web. To address these limitations, ontologies provide a more expressive alternative. The Web Ontology Language (OWL) was developed on top of the RDF model to effectively define and work with such ontologies (Antoniou & van Harmelen, 2003).

While RDF represents knowledge as triples and RDFS adds basic schema information, OWL extends these capabilities with richer semantics and formal reasoning mechanisms. OWL supports expressive class hierarchies, property characteristics, and logical axioms that enable automatic inference. Below are examples of OWL constructs illustrating those concepts:

- **Class Hierarchies and Disjoint Classes:** OWL allows the specification of subclasses and disjointness between classes.

```
<SubClassOf>
    <Class IRI="#Manager"/>
    <Class IRI="#Employee"/>
</SubClassOf>

<DisjointClasses>
    <Class IRI="#Person"/>
    <Class IRI="#Organization"/>
</DisjointClasses>
```

Listing 2.5: OWL Class Hierarchies Example

This declares that `Manager` as a subclass of `Employee`, while ensuring that no individual can be a `Person` and a `Organization` at the same time.

- **Object Properties and Data Properties:** OWL distinguishes between:
  - *Object Properties*, which link individuals to other individuals.
  - *Data Properties*, which link individuals to literal values (e.g., strings, numbers, dates).

```
<!-- Object Property: worksAt (inverse of employs) -->
<InverseObjectProperties>
    <ObjectProperty IRI="#worksFor"/>
    <ObjectProperty IRI="#employs"/>
</InverseObjectProperties>

<!-- Object Property: manages (transitive) -->
<TransitiveObjectProperty>
    <ObjectProperty IRI="#manages"/>
</TransitiveObjectProperty>

<!-- Data Property: hasAge -->
<DataProperty IRI="#hasAge">
    <Domain IRI="#Person"/>
    <Range IRI="&xsd;integer"/>
</DataProperty>
```

Listing 2.6: OWL Property Example

These declarations state that if an `Employee` `worksAt` a `Organization`, then the `Organization` `employs` the `Employee`; that the `manages` relationship is transitive; and that `hasAge` values specifies `Person` instances.

- **Logical Axioms:** OWL supports logical definitions that enable automatic inference, such as cardinality restrictions.

```
<!-- Define Manager as an Employee who manages at least one Employee -->
<EquivalentClasses>
    <Class IRI="#Manager"/>
    <ObjectIntersectionOf>
        <Class IRI="#Employee"/>
        <ObjectSomeValuesFrom>
            <ObjectProperty IRI="#manages"/>
            <Class IRI="#Employee"/>
        </ObjectSomeValuesFrom>
    </ObjectIntersectionOf>
</EquivalentClasses>

<!-- Enforce that every Person has exactly one birthdate -->
<SubClassOf>
    <Class IRI="#Person"/>
    <DataExactCardinality cardinality="1">
        <DataProperty IRI="#hasAge"/>
        <Datatype IRI="&xsd;integer"/>
    </DataExactCardinality>
</SubClassOf>
```

Listing 2.7: OWL Logical Axioms with Cardinality

The first axiom defines a `Manager` as an `Employee` who manages at least one other `Employee`. The second axiom enforces that every `Person` has exactly one `hasAge` value.

OWL employs an XML or RDF-based syntax for defining ontologies and leverages URIs to uniquely identify and link resources. This combination enables the creation of semantically rich, machine-interpretable knowledge models (W3C, 2012). Compared to RDF and RDFS, OWL provides greater expressiveness and supports automated reasoning, making it ideal for domains requiring complex classifications, rule-based inference, and data integration on the Semantic Web.

2.1.4 SPARQL Protocol and RDF Query Language (SPARQL)

SPARQL Protocol and RDF Query Language (SPARQL) is the W3C-standardized query language for retrieving and manipulating data stored in RDF format (Glimm, 2011). SPARQL operates directly on RDF graphs, allowing users to express complex graph-matching patterns and apply logical filters to extract, transform, and update data.

A typical SPARQL query can include several main components:

- **PREFIX**: Defines namespaces to abbreviate long URIs, improving readability.
- **SELECT**: Specifies variables to retrieve from the query, mainly for data extraction.
- **ASK**: Returns a boolean result indicating whether a given graph pattern exists.
- **CONSTRUCT**: Creates new RDF triples based on pattern-matching results.
- **INSERT/DELETE**: Adds or removes triples from the RDF dataset.
- **WHERE**: Describes the graph patterns (triple patterns) to match against the RDF dataset.
- **MODIFIERS**: Such as `LIMIT`, `OFFSET`, and `ORDER BY`, which refine the returned results.

SELECT Queries

The `SELECT` query form is used to retrieve specific data that matches certain patterns. For example, the query below finds all people and their ages in the dataset:

```
PREFIX ex: <http://example.org/>
SELECT ?person ?age
WHERE {
    ?person ex:hasAge ?age.
}
```

Listing 2.8: Basic SPARQL Select Query

To add conditions and constraints, SPARQL supports the use of `FILTER` expressions:

```
PREFIX ex: <http://example.org/>
SELECT ?person ?age
WHERE {
    ?person ex:hasAge ?age.
    FILTER (?age >= 40)
}
```

Listing 2.9: SPARQL Select with Filter

This query returns only those individuals whose age is 40 or older.

ASK Queries

The `ASK` query form checks whether a pattern exists in the dataset and returns a boolean value. This is useful for validation or constraint checking.

```
PREFIX ex: <http://example.org/>
ASK {
    ?person ex:hasAge ?age .
    FILTER (?age < 0)
}
```

Listing 2.10: SPARQL ASK Query Example

This query verifies whether there is any person with a negative age. The result is either `true` or `false`.

CONSTRUCT and INSERT Queries

The `CONSTRUCT` query form generates new RDF triples by pattern-matching and transforming existing data. The result of a `CONSTRUCT` query is a new RDF graph, which can be explored or reused.

```
PREFIX ex: <http://example.org/>
CONSTRUCT {
    ?person ex:aboveForty "true" .
}
WHERE {
    ?person ex:hasAge ?age .
    FILTER (?age > 40)
}
```

Listing 2.11: SPARQL CONSTRUCT Query Example

This query produces a graph that asserts each person aged 41 or older is an above forty years old. However, it does not change the original dataset.

To permanently add such inferred triples, SPARQL 1.1 provides `INSERT` queries (W3C, n.d.-d):

```
PREFIX ex: <http://example.org/>
INSERT {
    ?person ex:aboveForty "true" .
}
WHERE {
    ?person ex:hasAge ?age .
    FILTER (?age > 40)
}
```

Listing 2.12: SPARQL INSERT Query Example

Unlike `CONSTRUCT`, this query updates the dataset itself by adding the `ex:aboveForty` triples.

Property Paths

A Property Path is a sequence of predicates (properties) that connects two nodes in an RDF graph (W3C, n.d.-c). SPARQL supports property paths to concisely express navigation over chains, alternatives, or repetitions of properties. This makes it possible to query deep or complex graph structures without explicitly writing all intermediate steps.

Key operators in property paths include:

- **`path1/path2`**: Forwards path (path1 followed by path2)

- **^path1**: Backwards path (object to subject)
- **path1|path2**: Either path1 or path2
- **path1***: path1, repeated zero or more times
- **path1+**: path1, repeated one or more times
- **path1?**: path1, optionally
- **path1{m,n}**: At least m and no more than n occurrences of path1
- **path1{n}**: Exactly n occurrences of path1
- **path1{m,}**: At least m occurrences of path1
- **path1{,n}**: At most n occurrences of path1

Example — querying indirect social connections:

```
PREFIX ex: <http://example.org/>
SELECT ?manager
WHERE {
    ?manager ex:manages/ex:hasAge ?age .
    FILTER (?age <= 40)
}
```

Listing 2.13: SPARQL Query with Property Path

This query finds all managers that have employees with a age of 40 years old or younger, by using the path `manages` to find the employees and then `hasAge` to retrieve the age.

Another example — finding all employees under a management chain:

```
PREFIX ex: <http://example.org/>
SELECT ?manager ?employee
WHERE {
    ?manager ex:manages* ?employee .
    FILTER (?manager != ?employee)
}
```

Listing 2.14: SPARQL Property Path for Management Hierarchies

This query returns both direct and indirect employees managed by a person, excluding self-references. The use of * makes it possible to traverse arbitrarily deep hierarchies without manually specifying multiple joins.

SPARQL provides a powerful, expressive, and flexible interface for interacting with RDF data. `SELECT` and `ASK` queries enable data retrieval and validation, while `CONSTRUCT` and `INSERT` queries support transformation and persistence of new triples. Property paths further enrich querying capabilities by supporting advanced graph navigation, making SPARQL indispensable for reasoning, knowledge discovery, and ontology-driven applications.

2.1.5 Reasoning and Inference

Reasoning and inference are fundamental processes in semantic web technologies that allow for the derivation of implicit knowledge from explicitly stated data. These processes play a key role in enabling systems to perform intelligent tasks such as knowledge discovery, validation, and classification.

Inference refers to the process of generating new facts based on existing knowledge and logical rules. This mechanism enables systems to deduce conclusions not directly present in the data, thereby enriching the knowledge base.

Reasoning, in turn, is the computational realization of inference using formal logic. It involves analyzing ontologies to derive additional information, validate logical consistency, or classify entities according to their properties and relationships.

The tools specialized in reasoning are known as **reasoners**. These tools process ontologies to perform tasks such as:

- **Class Inference**: Determining class membership based on logical axioms (e.g., inferring that `John` is a `Mammal` if `John` is a `Person` and all `Persons` are `Mammals`).
- **Consistency Checking**: Ensuring that the ontology and data do not contain contradictions.
- **Property Reasoning**: Inferring relationships based on property characteristics, such as transitivity, symmetry, or inverse properties.

For example, given the statements `<A> <knows> <B>` and `<B> <knows> <C>`, a reasoner can infer that `<A> <knows> <C>` if the `knows` property is defined as transitive. Similarly, if a property such as `friendOf` is defined as symmetric, stating that `Alice` is a friend of `Bob` enables the inference that `Bob` is also a friend of `Alice`.

The reasoners can automatically compute the logical consequences of an ontology. The way ontological axioms interact can be subtle and complex, often making it difficult for humans to intuitively grasp the implications of certain modeling choices (W3C, n.d.-a).

Despite their utility, reasoners have limitations. Some, like HermiT (Glimm et al., 2014), support integration with SPARQL but do not use SPARQL to perform reasoning itself. Moreover, incomplete or inconsistent data can lead to incorrect inferences, and many reasoners struggle to scale efficiently for large ontologies or datasets. This can hinder their practicality in scenarios involving extensive or complex knowledge graphs.

A more subtle but important limitation lies in the interaction between property chain axioms and irreflexive properties. OWL reasoners are unable to infer relationships when the inferred property is explicitly declared as irreflexive, and its derivation depends on a property chain. For example, in our working ontology, the property `hasSibling` is defined as irreflexive and derived through a chain involving `hasParent` and `hasChild`. Most OWL reasoners do not support reasoning in such cases due to the conflict between the open-world assumption and the logical restrictions introduced by irreflexivity constraints on derived properties.

2.1.6 OWL Reasoner Operations

OWL (Web Ontology Language) reasoners enable automated inference over ontologies by applying logical rules grounded in OWL semantics. The OWL Web Ontology Language describes a language for ontologies, equipped with a formal semantics as specified in the OWL Web Ontology Semantics and Abstract Syntax (OWL S&AS) (University of Manchester, n.d.). Using these semantics, inferences about ontologies and individuals can be systematically derived. However, it is not always obvious why certain inferences occur. The explanation of the reasoning process remains a topic of ongoing research and has yet to reach a level of effectiveness that provides clear insight into the reasoning steps.

Below is a summary of key reasoning operations supported by OWL reasoners, presented in First-Order Logic (FOL) (Frade, 2009) notation for clarity and precision. RDF triples of the form `?subject ?predicate ?object` are represented in a function-style syntax as `?predicate(?subject, ?object)`.

Instance Checking

Instance checking determines whether a particular individual is a member of a specific class. This is typically derived from domain and range constraints in the ontology.

- **Domain Inference:**

$$\texttt{rdfs:domain}(P, C) \land P(s, o) \rightarrow \texttt{rdf:type}(s, C) \tag{2.1}$$

- **Range Inference:**

$$\texttt{rdfs:range}(P, C) \land P(s, o) \rightarrow \texttt{rdf:type}(o, C) \tag{2.2}$$

Hierarchy

Hierarchy reasoning infers subclass-superclass relationships, subproperties, and indirect instance typing.

- **Subclass Inference:**

$$\texttt{rdfs:subClassOf}(A, B) \land \texttt{rdfs:subClassOf}(B, C) \rightarrow \texttt{rdfs:subClassOf}(A, C) \tag{2.3}$$

- **Subproperty Inference:**

$$\texttt{rdfs:subPropertyOf}(P, Q) \land P(s, o) \rightarrow Q(s, o) \tag{2.4}$$

- **Instance Hierarchy:**

$$\texttt{rdfs:subClassOf}(A, B) \land \texttt{rdf:type}(s, A) \rightarrow \texttt{rdf:type}(s, B) \tag{2.5}$$

Inverse Property Inference

Inverse property reasoning allows us to infer new object property triples by recognizing inverse relationships defined in the ontology.

$$\texttt{owl:inverseOf}(P, Q) \Rightarrow (P(x, y) \rightarrow Q(y, x)) \tag{2.6}$$

$$\texttt{owl:inverseOf}(P, Q) \rightarrow \texttt{owl:inverseOf}(Q, P) \tag{2.7}$$

Transitive Property Reasoning

Transitive property reasoning infers indirect relationships by chaining repeated applications of a transitive property.

$$P \subseteq \texttt{owl:TransitiveProperty} \Rightarrow (P(x, y) \land P(y, z) \rightarrow P(x, z)) \tag{2.8}$$

Symmetric Property Reasoning

Symmetric property reasoning allows the inference of mirrored relationships.

$$P \subseteq \texttt{owl:SymmetricProperty} \Rightarrow (P(x,y) \rightarrow P(y,x)) \tag{2.9}$$

Restrictions

If an individual satisfies the property restrictions of an anonymous class that is defined as equivalent to a named class, it can be inferred to be an instance of that named class.

$$\texttt{rdf:type}(x, \exists P.C) \rightarrow \exists y.\ P(x,y) \wedge \texttt{rdf:type}(y, C) \tag{2.10}$$

Equivalence & Subsumption

Equivalence reasoning identifies when two classes can be considered logically equivalent based on their shared property restrictions.

$$\forall x.\ (\texttt{rdf:type}(x, \exists P.A) \leftrightarrow \texttt{rdf:type}(x, \exists P.B)) \Rightarrow \texttt{owl:equivalentClass}(A, B) \tag{2.11}$$

Property Chain

A property chain defines a new property as the composition of two or more existing properties, allowing the inference of indirect relationships.

$$R_1 \circ R_2 \rightarrow R_3 \Rightarrow (R_1(x,y) \wedge R_2(y,z) \rightarrow R_3(x,z)) \tag{2.12}$$

Consistency Checking

Consistency checking ensures that the ontology does not contain logical contradictions, either in its axioms or in its instance data.

$$\exists x.\ \varphi(x) \wedge \neg\varphi(x) \tag{2.13}$$

2.2 Related Works

This section provides an overview of existing approaches for querying and reasoning over ontologies, highlighting the strengths and weaknesses of **SPARQL-based** approaches and **C-SPARQL for stream reasoning**. While these methods offer valuable contributions, they still present scalability and efficiency challenges that OntoExpand seeks to overcome.

2.2.1 Querying OWL with SPARQL

Query answering in the presence of ontologies is known as ontology-based data access (OBDA), and has long been an important topic in applied and foundational research (Bischof et al., 2014). While OWL provides an expressive framework for knowledge representation, its direct querying capabilities are limited. SPARQL, as the standard query language for RDF, enables data retrieval but does not inherently support reasoning over ontological axioms.

Several approaches have been proposed to bridge this gap by extending SPARQL with reasoning capabilities. These include query rewriting techniques and materialization-based approaches that allow SPARQL to work with inferred knowledge. However, these solutions often come with scalability limitations when handling large and dynamic datasets.

A notable contribution is the work by Jing et al.(Jing et al., 2009), who proposed a graph pattern rewriting approach to support OWL-DL reasoning without the need for materializing inference ontologies. Their method rewrites SPARQL queries at runtime based on predefined OWL inference rules, such as class hierarchies and property characteristics, to retrieve implicit knowledge directly from the base ontology. This approach significantly reduces storage and maintenance overhead compared to full materialization. Nonetheless, query expansion through UNION clauses can introduce performance trade-offs, particularly in low-selectivity query scenarios.

2.2.2 Recursive Reasoning with SPARQL CONSTRUCT

Another direction to enhance reasoning within SPARQL is through recursive CONSTRUCT queries, as demonstrated by the SiN3 framework(Arndt et al., 2023). SiN3 translates SPARQL CONSTRUCT queries into Notation3 (N3) rules, which can be executed recursively by N3-compliant reasoners like EYE. This enables modular, rule-based reasoning directly on RDF data using familiar SPARQL syntax.

Key advantages of SiN3 include:

- **Recursive reasoning**: Queries can act on the results of other queries, supporting complex inference chains.
- **Efficiency**: SiN3 demonstrated an order-of-magnitude performance gain over SPIN-based reasoning in their Zika virus case study.
- **Modularity and maintainability**: Reusable query modules reduce code duplication and enhance clarity.

While promising, SiN3 relies on external tooling for SPARQL-to-N3 translation and execution, which may introduce integration complexity. Moreover, full support for SPARQL features like solution modifiers or federated queries is still under development.

2.2.3 C-SPARQL and Stream Reasoning

Continuous SPARQL (C-SPARQL) is an extension of SPARQL whose distinguishing feature is the support of continuous queries, i.e., queries registered over RDF data streams and then continuously executed (Kramer & Augusto, 2017). Unlike traditional SPARQL, which operates over static RDF datasets, C-SPARQL enables reasoning over dynamic, time-dependent information, making it highly relevant for applications in the Internet of Things (IoT), smart cities, and financial analytics.

Key features of C-SPARQL include:

- Support for window-based queries, enabling real-time filtering of streaming RDF data.
- Integration of reasoning mechanisms to infer knowledge from continuous data sources.
- Application in dynamic environments requiring real-time decision-making.

The combination of static RDF data with streaming information leads to *stream reasoning*, an essential approach for enabling logical inference in real time over vast and often noisy data streams (Barbieri et al., 2010). This capability is crucial for supporting the decision-making processes of large numbers of concurrent users. With stream reasoning using languages such as C-SPARQL, continuous streams of raw RDF data can be processed for context-awareness. However, writing numerous context queries and rules in this manner can be error-prone and often involves significant boilerplate, presenting challenges in scalability and maintainability (Kramer & Augusto, 2017).

While C-SPARQL addresses some of the limitations of traditional SPARQL, it still relies on external reasoning engines to infer new knowledge. This dependency can introduce performance bottlenecks,

particularly when handling high-speed data streams, necessitating optimizations to enhance efficiency in large-scale applications.

2.2.4 Comparison and Limitations of Existing Approaches

Despite advancements in querying and reasoning over RDF and OWL data, existing approaches still have notable limitations that OntoExpand seeks to address. A comparison of key methods is summarized in Table1.

Approach	**Reasoning**	**Scalability**	**Real-time Support**
SPARQL	Limited	High	No
C-SPARQL	Yes (External)	Moderate	Yes
Graph Rewriting (Jing et al.)	Yes (Rewriting)	Moderate	No
SiN3 (Arndt et al.)	Yes (Recursive)	High	No
OntoExpand (Proposed)	Yes (SPARQL-based)	High	Yes

Table 1: Comparison of ontology reasoning approaches

From this comparison, the key gaps in existing approaches are:

- **Performance**: Most existing solutions rely on external reasoners or complex rule engines, leading to high computational costs.
- **Integration**: SPARQL-based reasoning is often disconnected from the ontology querying process, or requires transformation into other rule languages.
- **Scalability**: Large or streaming datasets introduce bottlenecks in traditional reasoning or rewriting methods.

These limitations form the basis for OntoExpand, which aims to address these issues by embedding reasoning directly into SPARQL queries, improving efficiency and scalability.

2.3 Summary

This chapter provided a detailed overview of the foundational technologies and current approaches for ontology representation, querying, and reasoning. It covered key Semantic Web concepts, including ontologies, RDF, RDFS, OWL, SPARQL, and reasoners, illustrating how these components support structured

knowledge representation and inference. The chapter also reviewed existing methods for integrating reasoning with querying, such as SPARQL-based approaches, recursive CONSTRUCT queries, and stream reasoning with C-SPARQL, while highlighting their limitations in performance, scalability, and real-time support. These insights underscore the motivation for OntoExpand, demonstrating the gaps in existing solutions that the project aims to address.

Chapter 3

The Problem and Its Challenges

This chapter presents the core problem and the specific challenges addressed by OntoExpand. It explains the limitations of traditional OWL reasoners, including high computational demands, poor scalability, and weak integration with SPARQL queries, which hinder real-time ontology expansion and reasoning. The chapter also identifies key challenges for efficient ontology management, such as improving computational efficiency, supporting large-scale datasets, enabling seamless query integration, and automating the conversion of ontology axioms into SPARQL queries. By defining these problems and challenges, this chapter establishes the technical context for the design and implementation of OntoExpand's SPARQL-driven reasoning framework.

3.1 The Problem

The expansion and maintenance of OWL ontologies pose significant challenges, especially when dealing with large, complex datasets. As the domain of ontologies grows, traditional ontology reasoners become more inadequate due to their high computational demands and slow processing times. These tools often process entire ontologies at once, which leads to inefficient handling and difficulty in scaling ontology-driven systems. In dynamic, real-time environments, this poor scaling can severely prejudice the system's ability to respond quickly to new queries or data updates.

Another critical issue surges from the lack of integration between existing reasoning tools and widely used query languages like SPARQL. Ontology reasoners are typically disconnected from the querying process, meaning that reasoning and querying are separate stages. This separation complicates the real-time extraction of knowledge and ontology expansion, making it difficult to integrate reasoning with dynamic data sources in a seamless way. Moreover, reasoners tend to lack flexibility in adapting to different use cases or handling ontology updates efficiently.

Given these limitations, there is a need for an alternative approach that improves both efficiency and

expressiveness, enabling the reasoning process to scale dynamically in large datasets while supporting real-time knowledge retrieval.

3.2 Challenges

To address the aforementioned problems, OntoExpand focuses on the following key challenges:

- **Efficiency**: Traditional reasoners require an substantial amount computational power, often resulting in slow performance when handling large or complex ontologies. This inefficiency makes real-time ontology expansion and reasoning more time consuming in many scenarios.
- **Scalability**: As ontologies grow in size and complexity, traditional reasoners struggle to keep up. This increases the demand for system resources, making it difficult to manage large-scale ontologies without substantial hardware or alternative resources such as cloud services.
- **Query Integration**: Existing reasoning tools do not integrate well with SPARQL. Without a proper integration, ontology-based reasoning cannot be properly combined within dynamic, query-driven systems, which limits its utility for real-time decision-making and data extraction.
- **Automation**: Converting OWL axioms into SPARQL queries manually is an error-prone process. Automated tools that can generate SPARQL queries directly from ontology axioms may be critical for improving efficiency and reducing human error in dynamic systems.

By overcoming these challenges, OntoExpand aims to enhance the scalability, efficiency, and automation of ontology-driven systems, making them more suitable for real-time, large-scale applications. The ultimate goal is to enable a seamless connection between reasoning and querying processes, allowing for faster, more efficient knowledge extraction and ontology management.

3.3 Summary

The Problem and Its Challenges chapter identified the key limitations of traditional ontology reasoners, such as their inefficiency in handling large datasets, limited scalability, and lack of seamless integration with SPARQL for real-time querying. It also emphasized the need for automation in translating OWL axioms into executable queries, a task often prone to errors when done manually. OntoExpand was presented as a response to these challenges, aiming to improve efficiency, scalability, and automation in ontology-driven

systems while enabling dynamic reasoning and querying. This problem definition laid the groundwork for the next chapter, which shifted from theoretical challenges to practical solutions by showing how core OWL reasoning operations can be simulated with SPARQL queries.

Chapter 4

Reasoner Operations with SPARQL

In earlier chapters, we explored the theoretical foundations of RDF, OWL semantics, and the role of reasoning in ontology-based systems. We also discussed SPARQL as the standard query language for RDF data manipulation on ontologies.

This chapter focuses on the practical aspect of simulating core OWL reasoning operations using SPARQL. Rather than relying on a reasoner, we demonstrate how these inferences can be expressed through SPARQL queries. To properly capture OWL semantics, we will rely on OWL axioms (W3C, n.d.-b) as the logical basis for formulating the queries.

4.1 Case study introduction

For clarity and consistency, we will use a simplified ontology from the *House Targaryen of Game of Thrones*, described in detail in Listing A.1, as the working example throughout this chapter. The ontology, which was developed by the author as a case study, captures a subset of concepts and relationships inspired by the Targaryen lineage and lore. It defines core classes such as `Person`, `Dragon`, `House`, and `Sword`, with more specific subclasses like `Noble`, `King`, and `Longsword`.

Some key object properties include familial relationships like `hasParent`, `hasChild`, and derived properties such as `hasGrandchild` and `hasSibling`, which are expressed using OWL property chains. The ontology also includes domain-specific relations like `hasDragon` (linking people to dragons), `isAllyOf` (a symmetric property between houses), and `belongsToHouse` (relating nobles to their houses) . Furthermore, it introduces the use of OWL reasoning patterns such as inverse, transitive, symmetric, irreflexive properties, and equivalence classes (e.g., `Dragonrider` defined as a person who owns at least one dragon).

To provide a structured overview of these modeling choices, the following tables present the ontology's classes, individuals, and properties.

Class	Parent Class	Restrictions
Dragon	-	-
Dragonlord	Person	Some value from hasDragon
Dragonrider	Person	Some value from hasDragon
House	-	-
King	Noble	-
Longsword	Sword	Some value from inches
Noble	Person	-
Person	-	-
Sword	-	-

Table 2: Classes from case study

Property	Domain	Range	Type
belongsToHouse	Noble	House	Object
hasAncestor	Person	Person	Transitive
hasChild	Person	Person	Object (Inverse of hasParent)
hasDragon	Person	Dragon	Object
hasFather	Person	Person	Object (Subproperty from hasParent)
hasGrandchild	Person	Person	Object (Property Chain)
hasGrandparent	Person	Person	Object (Property Chain)
hasMother	Person	Person	Object (Subproperty from hasParent)
hasParent	Person	Person	Object
hasSibling	Person	Person	Irreflexive (Property Chain)
isAllyOf	House	House	Symmetric
inches	Sword	integer	Datatype

Table 3: Properties from case study

Individual	Type
AegonTargaryen	Person
AegonV	King
AerysII	King
BethaBlackwood	Person
CassanaEstermont	Person
Daenerys	Person
DaeronTargaryen	Person
Drogon	Dragon
DuncanTargaryen	Person
EliaMartell	Person
Ice	Sword
JaehaerysII	King
JocelynBaratheon	Person
OrmundBaratheon	Person
RenlyBaratheon	Person
Rhaegal	Named Individual
Rhaegar	Person
Rhaella	Person
RhaelleTargaryen	Person
Rhaenys	Person
RobertBaratheon	King
Shaera	Person
StannisBaratheon	Person
SteffonBaratheon	Person
Viserion	Named Individual
Viserys	Person
houseBaratheon	House
houseTargaryen	House

Table 4: Individuals from case study

4.2 Operations

4.2.1 Instance Checking

Domain Inference

This query simulates the reasoning rule in Equation 2.1. It infers the class type of an individual based on the domain of a property used in a triple where the individual is the subject.

```
CONSTRUCT {
    ?indiv a ?class .
}
WHERE {
    ?indiv a owl:NamedIndividual .
    ?class a owl:Class .
    ?prop rdfs:domain ?class .
    ?indiv ?prop ?o .

    FILTER NOT EXISTS { ?indiv a ?class .}
}
```

Listing 4.1: Domain Inference Query

This approach targets individuals that are subjects of properties with a declared domain and infers that these individuals are instances of the property's domain class. To prevent duplicating existing type assertions, it employs a `FILTER NOT EXISTS` clause.

Existing Triples:
`targaryen:Rhaegar a owl:NamedIndividual`
`targaryen:Noble a owl:Class`
`targaryen:belongsToHouse rdfs:domain targaryen:Noble`
`targaryen:Rhaegar targaryen:belongsToHouse targaryen:houseTargaryen`
Result Triples:
`targaryen:Rhaegar a targaryen:Noble`

Table 5: Class inference by Domain example

Range Inference

This query corresponds to the rule in Equation 2.2. It infers the class type of an individual based on the range of a property used in a triple where the individual is the object.

```
CONSTRUCT {
    ?indiv a ?class .
}
WHERE {
    ?indiv a owl:NamedIndividual .
    ?class a owl:Class .
    ?prop rdfs:range ?class .
    ?s ?prop ?indiv .

    FILTER NOT EXISTS {?indiv a ?class .}
}
```

Listing 4.2: Range Inference Query

This approach targets individuals that are objects of properties with a declared range and infers that these individuals are instances of the property's range class. To prevent duplicating existing type assertions, it employs a `FILTER NOT EXISTS` clause.

Existing Triples:
`targaryen:Viserion a owl:NamedIndividual`
`targaryen:Dragon a owl:Class`
`targaryen:hasDragon rdfs:range targaryen:Dragon`
`targaryen:Daenerys targaryen:hasDragon targaryen:Viserion`
Result Triples:
`targaryen:Viserion a targaryen:Dragon`

Table 6: Class inference by Range example

4.2.2 Hierarchy

Subclass Inference

This query simulates the rule defined in Equation 2.3, retrieving inferred subclass relationships based on transitive subclass paths.

```
CONSTRUCT {
    ?sub rdfs:subClassOf ?super .
}
WHERE {
    ?sub a owl:Class .
    ?super a owl:Class .
    ?sub rdfs:subClassOf* ?super .

    FILTER(?sub != ?super)
    FILTER NOT EXISTS {?sub rdfs:subClassOf ?super .}
}
```

Listing 4.3: Subclass Inference Query

This approach identifies all subclass pairs where `?sub` is a direct or indirect subclass of `?super`. It utilizes the transitive closure operator `rdfs:subClassOf*`, filters out cases where the subclass and superclass are identical, and avoids duplicating existing subclass assertions by using `FILTER NOT EXISTS`.

Existing Triples:
`targaryen:King a owl:Class`
`targaryen:Person a owl:Class`
`targaryen:King rdfs:subClassOf* targaryen:Person .`
Result Triples:
`targaryen:King rdfs:subClassOf targaryen:Person`

Table 7: Subclass hierarchy inference example

Subproperty Inference

This query simulates the rule in Equation 2.4, inferring superproperty triples based on subproperty relationships.

```
CONSTRUCT {
    ?indiv ?superProp ?target .
}
WHERE {
    ?prop a owl:ObjectProperty ;
          rdfs:subPropertyOf+ ?superProp .
    ?indiv ?prop ?target .
    FILTER NOT EXISTS {?indiv ?superProp ?target .}
}
```

Listing 4.4: Subproperty Inference Query

For each triple with property `?prop`, this approach retrieves all superproperties `?superProp` using `rdfs:subPropertyOf+`. It then infers new triples `(?indiv, ?superProp, ?target)` to capture the subproperty hierarchy and avoids generating duplicate triples by employing `FILTER NOT EXISTS`.

Existing Triples:
`targaryen:hasMother rdfs:subPropertyOf targaryen:hasParent`
`targaryen:Daenerys targaryen:hasMother Rhaella`
Result Triples:
`targaryen:Daenerys targaryen:hasParent Rhaella`

Table 8: Subproperty hierarchy inference example

Instance Hierarchy

This corresponds to the rule in Equation 2.5, inferring indirect class types for individuals by following subclass hierarchies.

```
CONSTRUCT {
    ?indiv a ?parent .
}
WHERE {
    ?indiv a owl:NamedIndividual ;
           a ?class .
    ?class rdfs:subClassOf* ?parent .
    ?parent a owl:Class .

    FILTER NOT EXISTS {?indiv a ?parent .}
}
```

Listing 4.5: Instance Hierarchy Inference Query

For each individual `?indiv` with type `?class`, this approach finds all superclasses `?parent` using `rdfs:subClassOf*` and infers new type assertions `rdf:type(?indiv, ?parent)` if they are not already present.

Existing Triples:
`targaryen:AegonV a owl:NamedIndividual`
`targaryen:AegonV a targaryen:King`
`targaryen:King rdfs:subClassOf* targaryen:Person .`
`targaryen:Person a owl:Class`
Result Triples:
`targaryen:AegonV a targaryen:Person`

Table 9: Instance parent class inference example

4.2.3 Inverse Property Inference

Symmetric Inverse Declaration

Since `owl:inverseOf` is defined as a symmetric property in OWL, but ontologies may not always declare both directions, we must enforce this explicitly before applying inference. This query enforces Equation 2.7 by materializing the symmetric closure of `owl:inverseOf` relationships.

```
CONSTRUCT {
  ?inv owl:inverseOf ?prop .
}
WHERE {
  ?prop owl:inverseOf ?inv .
  FILTER NOT EXISTS { ?inv owl:inverseOf ?prop . }
}
```

Listing 4.6: Inverse Property Symmetry Enforcement

This approach ensures that if `?prop owl:inverseOf ?inv` exists, then `?inv owl:inverseOf ?prop` is also asserted. Maintaining this symmetry is essential for correct inverse property reasoning.

Inverse Inference

This query applies Equation 2.6 by inferring reverse triples through the inverse property relation. The inference can be fully made thanks to the results of the previous query Listing 4.6.

```
CONSTRUCT {
    ?b ?inv ?a .
}
WHERE {
    ?prop a owl:ObjectProperty .
    ?inv a owl:ObjectProperty .
    ?prop owl:inverseOf ?inv .
    ?a ?prop ?b .
}
```

Listing 4.7: Inverse Inference

For each triple `?a ?prop ?b` with a declared inverse `?inv`, this approach infers the reverse triple `?b ?inv ?a`, relying on the `owl:inverseOf` links established in the previous step.

Existing Triples:
`targaryen:hasChild a owl:ObjectProperty`
`targaryen:hasParent a owl:ObjectProperty`
`targaryen:hasChild owl:inverseOf targaryen:hasParent`
`targaryen:AerysII targaryen:hasChild targaryen:Daenerys`
Result Triples:
`targaryen:Daenerys targaryen:hasParent targaryen:AerysII`

Table 10: Inverse property inference example

4.2.4 Transitive Property Reasoning

Simple Transitive Inference

This basic query corresponds to a blunt definition of Equation 2.8 and attempts to infer one level of transitivity, capturing only two-step compositions:

```
CONSTRUCT {
    ?x ?prop ?z .
}
WHERE {
    ?prop a owl:ObjectProperty, owl:TransitiveProperty .
    ?x ?prop ?y .
    ?y ?prop ?z .
}
```

Listing 4.8: Simple Transitive Inference

This approach identifies direct property assertions where a transitive property holds between `?x` → `?y` → `?z` and adds a new triple `?x ?prop ?z` to represent a one-hop transitive inference. **Limitation:** it only works for chains of length 2 and cannot infer longer transitive paths.

Existing Triples:
`targaryen:hasAncestor a owl:ObjectProperty`
`targaryen:hasAncestor a owl:TransitiveProperty`
`targaryen:Daenerys targaryen:hasAncestor targaryen:AerysII`
`targaryen:AerysII targaryen:hasAncestor targaryen:JaeherysII`
Result Triples:
`targaryen:Daenerys targaryen:hasAncestor targaryen:JaeherysII`

Table 11: Simple transitive inference example

Transitive Property Identification

This query selects all properties declared as transitive in the ontology:

```
SELECT ?prop WHERE {
  ?prop a owl:ObjectProperty, owl:TransitiveProperty .
}
```

Listing 4.9: Retrieve Transitive Properties

This approach retrieves all Object Properties declared as Transitive Properties as well, which can then be used in subsequent transitive closure queries.

Deep Transitive Inference

For each property obtained from the selection query, the following is applied:

```
CONSTRUCT {
    ?x <{prop}> ?z .
}
WHERE {
    ?x <{prop}>+ ?z .
}
```

Listing 4.10: Deep Transitive Inference

This approach uses the + path operator to traverse any number of consecutive links via the transi-

tive property, efficiently inferring transitive chains of arbitrary length. It requires dynamically substituting <prop> with the actual property URI, as retrieved from query Listing 4.9.

Existing Triples:
`targaryen:hasAncestor a owl:ObjectProperty`
`targaryen:hasAncestor a owl:TransitiveProperty`
`targaryen:Daenerys targaryen:hasAncestor+ targaryen:AegonV`
Result Triples:
`targaryen:Daenerys targaryen:hasAncestor targaryen:AegonV`

Table 12: Deep Transitive inference example

Comparison

The **simple query** provides a direct but limited application of transitivity, handling only 2-step chains. In contrast, the **deep query** generalizes inference by capturing all indirect relations using path expressions, being a more robust and better aligns with the intended reasoning results.

4.2.5 Symmetric Property Reasoning

Symmetric Property Inference

This query corresponds to Equation 2.9.

```
CONSTRUCT {
  ?subj ?prop ?target .
}
WHERE {
  ?prop a owl:ObjectProperty, owl:SymmetricProperty .
  ?target ?prop ?subj .
  FILTER NOT EXISTS { ?subj ?prop ?target . }
}
```

Listing 4.11: Symmetric Property Inference

This approach selects all Symmetric Properties and for every triple `(?target ?prop ?subj)`, it infers the symmetric triple `(?subj ?prop ?target)` while avoiding duplicate triples by filtering.

Existing Triples:
`targaryen:isAllyOf a owl:ObjectProperty`
`targaryen:isAllyOf a owl:SymmetricProperty`
`targaryen:houseBaratheon targaryen:isAllyOf targaryen:houseTargaryen`
Result Triples:
`targaryen:houseTargaryen targaryen:isAllyOf targaryen:houseBaratheon`

Table 13: Symmetric inference example

4.2.6 Restrictions

Class Inference from Property Restrictions

This query enforces Equation 2.10.

```
CONSTRUCT {
  ?instance a ?inferClass .
}
WHERE {
  ?inferClass owl:equivalentClass/owl:intersectionOf ?restrictions .
  ?restrictions rdf:rest*/rdf:first ?part .

  OPTIONAL { ?part a owl:Restriction . }

  ?part owl:onProperty ?prop .
  ?part (owl:someValuesFrom | owl:allValuesFrom | owl:hasValue) ?val .

  { ?instance ?prop ?val .}
  UNION
  {
    ?instance ?prop ?anyValue .
    ?prop rdfs:range ?val .
  }
}
```

Listing 4.12: Class Inference from Restrictions

This approach handles cases where a class is defined as an intersection of restrictions by using the OWL axioms: `owl:equivalentClass` and `owl:intersectionOf`. It extracts restrictions from the RDF list and applies inference if instances satisfy any of the property constraints, supporting inference based on direct values or on range declarations.

Existing Triples:
`targaryen:Dragonlord owl:equivalentClass/owl:intersectionOf <Restrictions>`
`<Restrictions> rdf:rest*/rdf:first <Part>`
`<Part> owl:onProperty targaryen:hasDragon`
`<Part> owl:someValuesFrom targaryen:Dragon`
`targaryen:hasDragon rdfs:range targaryen:Dragon`
`targaryen:Daenerys targaryen:hasDragon targaryen:Drogon`
Result Triples:
`targaryen:Daenerys a targaryen:Dragonlord`

Table 14: Inference by restrictions example

4.2.7 Equivalence & Subsumption Reasoning

Equivalence Class Detection

To correspond to Equation 2.11 this CONSTRUCT query nested two SELECT queries.

```
SELECT ?class1
      (GROUP_CONCAT(CONCAT(STR(?p1), "->", STR(?v1)); SEPARATOR="|") AS ?rest1)
WHERE {
  ?class1 owl:equivalentClass/owl:intersectionOf/rdf:rest*/rdf:first ?r1 .
  ?r1 a owl:Restriction ;
      owl:onProperty ?p1 ;
      (owl:someValuesFrom | owl:allValuesFrom | owl:hasValue) ?v1 .
}
GROUP BY ?class1
```

Listing 4.13: Equivalence Class First Query

```
SELECT ?class2
      (GROUP_CONCAT(CONCAT(STR(?p2), "->", STR(?v2)); SEPARATOR="|") AS ?rest2)
WHERE {
  ?class2 owl:equivalentClass/owl:intersectionOf/rdf:rest*/rdf:first ?r2 .
  ?r2 a owl:Restriction ;
      owl:onProperty ?p2 ;
      (owl:someValuesFrom | owl:allValuesFrom | owl:hasValue) ?v2 .
}
GROUP BY ?class2
```

Listing 4.14: Equivalence Class Second Query

```
CONSTRUCT {
   ?class1 owl:equivalentClass ?class2 .
}
WHERE {
  {
    # First Query
  }

  {
    # Second Query
  }

  FILTER(?class1 != ?class2)
  FILTER(?rest1 = ?rest2)
}
```

Listing 4.15: Inference of Equivalent Classes

This query first extracts all property restrictions from class definitions involving `owl:Restriction` elements. It then computes a normalized signature string of these restrictions using `GROUP_CONCAT`. Two classes with identical restriction signatures are inferred to be equivalent via `owl:equivalentClass`, which is particularly useful for complex classes defined through `owl:intersectionOf` and property constraints.

Existing Triples:
targaryen:Dragonlord owl:equivalentClass/owl:intersectionOf <Restrictions>
targaryen:Dragonrider owl:equivalentClass/owl:intersectionOf <Restrictions>
<Restrictions> rdf:rest*/rdf:first <Part>
<Part> owl:onProperty targaryen:hasDragon
<Part> owl:someValuesFrom targaryen:Dragon
Result Triples:
targaryen:Daenerys owl:equivalentClass targaryen:Dragonrider

Table 15: Equivalence class inference example

4.2.8 Property Chain

Property Chain Identification

In order to correspond to the condition of Equation 2.12, this query identifies the property chains equivalent to the original property.

```
SELECT DISTINCT ?superProperty
    (GROUP_CONCAT(CONCAT("<", STR(?subProperty), ">"); SEPARATOR="/")
    AS ?propertyChain)
WHERE {
  ?superProperty owl:propertyChainAxiom ?chainList .
  ?chainList rdf:rest*/rdf:first ?subProperty .

  {?subProperty a owl:ObjectProperty .}
  UNION {
    ?listNode rdf:first ?subProperty ;
              rdf:rest rdf:nil .
    ?subProperty a owl:DatatypeProperty .
  }
}
GROUP BY ?superProperty ?chainList
```

Listing 4.16: Extract Property Chains

This approach identifies each `owl:propertyChainAxiom` along with its associated super-property and builds the full chain of sub-properties using path concatenation (e.g., `<p1>/<p2>/<p3>`).

Property Chain Inference

For each property chain discovered, the following inference is applied:

```
CONSTRUCT {
  ?x <{superProperty}> ?z .
}
WHERE {
  ?x <{propertyChain}> ?z .

  FILTER NOT EXISTS {
    ?p a owl:IrreflexiveProperty .
    FILTER(?p = <{superProperty}> && ?x = ?z)
  }
}
```

Listing 4.17: Infer Super-Property Relations via Property Chains

This approach infers that `?x` is related to `?z` via the `<superProperty>` if a valid property chain `<propertyChain>` exists between them. The `FILTER NOT EXISTS` clause ensures that if the super-property is declared as `owl:IrreflexiveProperty`, reflexive triples such as `?x <superProperty> ?x` are not produced.

Existing Triples:
`targaryen:hasSibling owl:propertyChainAxiom <Chain>`
`<Chain> rdf:first targaryen:hasParent`
`<Chain> rdf:rest*/rdf:first targaryen:hasChild`
`targaryen:Daenerys targaryen:hasParent targaryen:AerysII`
`targaryen:AerysII targaryen:hasChild targaryen:Rhaegar`
Result Triples:
`targaryen:Daenerys targaryen:hasSibling targaryen:Rhaegar`

Table 16: Inference by Property Chain axiom example

4.2.9 Consistency Checking

These queries aim to detect various forms of inconsistency, trying to avoid the contradiction of Equation 2.13. All examples assume hypothetical inconsistencies for illustration.

Unsatisfiable Class Detection

Rule: A class defined as the intersection of a class and its complement is unsatisfiable.

```
SELECT ?class WHERE {
  ?class a owl:Class .
  ?class owl:intersectionOf ?collection .

  ?collection rdf:rest*/rdf:first ?c1 .
  ?collection rdf:rest*/rdf:first ?c2 .

  ?c1 a owl:Class .
  ?c2 a owl:Class .
  ?c1 owl:complementOf ?c2 .

  FILTER(?c1 != ?c2)
}
```

Listing 4.18: Unsatiable Class Query

Hypothetical Triples:
`targaryen:Walker owl:intersectionOf (targaryen:Living, targaryen:Dead)`
`targaryen:Dead owl:complementOf targaryen:Living`
Result:
`?class = targaryen:Walker`

Table 17: Class defined as the intersection of a class and its complement

Contradictory Equivalence and Disjointness

Rule: Two classes cannot be declared both equivalent and disjoint.

```
SELECT ?class1 ?class2
WHERE {
  ?class1 owl:equivalentClass ?class2 .
  ?class1 owl:disjointWith ?class2 .
}
```

Listing 4.19: Contradictory Equivalence and Disjointness

Hypothetical Triples:
`targaryen:Dragon owl:equivalentClass targaryen:Wolf`
`targaryen:Dragon owl:disjointWith targaryen:Wolf`
Result:
`?class1 = targaryen:Dragon, ?class2 = targaryen:Wolf`

Table 18: Class declared both equivalent and disjoint

Disjoint Class Membership

Rule: No individual may belong to two disjoint classes.

```
SELECT ?individual ?class1 ?class2
WHERE {
  ?class1 owl:disjointWith ?class2 .
  ?individual a ?class1, ?class2 .
}
```

Listing 4.20: Disjoint Class Membership

Hypothetical Triples:
`targaryen:Dragon owl:disjointWith targaryen:Wolf`
`targaryen:DragonWolf a targaryen:Dragon, targaryen:Wolf`
Result:
`?individual = targaryen:DragonWolf,`
`?class1 = targaryen:Dragon, ?class2 = targaryen:Wolf`

Table 19: Individual declared as member of two disjoint classes

Functional Property Violation

Rule: An individual cannot have multiple distinct values for a functional property.

```
SELECT ?individual ?property (COUNT(DISTINCT ?value) AS ?valueCount)
WHERE {
  ?property a owl:FunctionalProperty .
  ?individual ?property ?value .
}
GROUP BY ?individual ?property
HAVING (COUNT(DISTINCT ?value) > 1)
```

Listing 4.21: Functional Property Violation

Hypothetical Triples:
targaryen:ownedBy a owl:FunctionalProperty
targaryen:Drogon targaryen:ownedBy targaryen:Daenerys
targaryen:Drogon targaryen:ownedBy targaryen:AegonV
Result:
?individual = targaryen:Drogon,
?property = targaryen:ownedBy, ?valueCount = 2

Table 20: Individual with multiple values for a functional property

Inverse Functional Property Violation

Rule: An object cannot have multiple distinct subjects via an inverse functional property.

```
SELECT ?object ?property (COUNT(DISTINCT ?subject) AS ?subjectCount)
WHERE {
  ?property a owl:InverseFunctionalProperty .
  ?subject ?property ?object .
}
GROUP BY ?object ?property
HAVING (COUNT(DISTINCT ?subject) > 1)
```

Listing 4.22: Inverse Functional Property Violation

Hypothetical Triples:
`targaryen:heirFrom a owl:InverseFunctionalProperty`
`targaryen:Rhaegar targaryen:heirFrom targaryen:AerysII`
`targaryen:Viserys targaryen:heirFrom targaryen:AerysII`
Result:
`?object = targaryen:AerysII, ?property = targaryen:heirFrom,`
`?subjectCount = 2`

Table 21: Object referred by multiple subjects using an inverse functional property

Disjoint Property Violation

Rule: Two disjoint properties cannot connect the same subject-object pair.

```
SELECT ?subject ?property1 ?property2 ?object
WHERE {
  ?property1 owl:propertyDisjointWith ?property2 .
  ?subject ?property1 ?object ;
          ?property2 ?object .
  FILTER(?property1 != ?property2)
}
```

Listing 4.23: Disjoint Property Violation

Hypothetical Triples:
`targaryen:hasSpouse owl:propertyDisjointWith targaryen:hasEnemy`
`targaryen:Rhaegar targaryen:hasSpouse targaryen:Lyanna`
`targaryen:Rhaegar targaryen:hasEnemy targaryen:Lyanna`
Result:
`?subject = targaryen:Rhaegar, ?property1 = targaryen:hasSpouse,`
`?property2 = targaryen:hasEnemy, ?object = targaryen:Lyanna`

Table 22: Disjoint properties linking the same subject-object pair

Domain-Class Disjoint Conflict

Rule: A property's domain must not be disjoint with any class of a subject using that property.

```
SELECT ?subject ?property ?domainClass ?conflictClass
WHERE {
  ?property rdfs:domain ?domainClass .
  ?domainClass owl:disjointWith ?conflictClass .
  ?subject a ?conflictClass .
  ?subject ?property ?object .
}
```

Listing 4.24: Domain-Class Disjoint Conflict

Hypothetical Triples:
`targaryen:Person owl:disjointWith targaryen:WhiteWalker`
`targaryen:NightKing a targaryen:WhiteWalker`
`targaryen:NightKing targaryen:belongsToHouse targaryen:houseTargaryen`
Result:
`?subject = targaryen:NightKing, ?property = targaryen:belongsToHouse,`
`?domainClass = targaryen:Person, ?conflictClass = targaryen:WhiteWalker`

Table 23: Subject class is disjoint with property's declared domain

Irreflexive Property Violation

Rule: An irreflexive property cannot relate an individual to itself.

```
SELECT ?prop ?subject
WHERE {
  ?prop rdf:type owl:IrreflexiveProperty .
  ?subject ?prop ?object .
  FILTER(?subject = ?object)
}
```

Listing 4.25: Irreflexive Property Violation

Hypothetical Triples:
`targaryen:hasSibling rdf:type owl:IrreflexiveProperty`
`targaryen:Daenerys targaryen:hasSibling targaryen:Daenerys`
Result:
`?prop = targaryen:hasSibling, ?subject = targaryen:Daenerys`

Table 24: Irreflexive property used reflexively on an individual

4.3 Summary

The Ontology Reasoning with SPARQL Queries chapter demonstrated how core reasoning tasks in OWL ontologies—such as inference and consistency checking—can be simulated using SPARQL `CONSTRUCT` and `INSERT` queries. It presented practical implementations of reasoning rules, showing how new knowledge can be derived and directly integrated into the ontology without relying on external reasoners. The chapter also illustrated examples of inconsistency detection and incremental ontology expansion, validating SPARQL as an effective alternative for reasoning tasks. This experimental foundation paved the way for the development of a full-fledged web platform, discussed in the next chapter, which integrates these reasoning processes into a user-friendly interface for dynamic ontology management.

Chapter 5

Ontology Expansion via Web Interface

This chapter introduces the web-based platform developed to automate ontology expansion through SPARQL reasoning. Building on the previous phase, where SPARQL `CONSTRUCT` queries were used to simulate core reasoning tasks such as inference and inconsistency detection, we now shift from a demonstration-based approach to a fully operational system.

The platform leverages these SPARQL queries to perform reasoning automatically, without requiring manual input. Inferred triples are inserted into the ontology using `INSERT DATA` operations, allowing real-time updates and knowledge enrichment. By integrating these processes into an interactive web interface, the system provides users with a practical tool for dynamic ontology management, eliminating the need for external reasoners and abstracting the complexity of SPARQL syntax.

5.1 Web Platform

The web platform developed in this project provides an interactive interface for reasoning and managing ontologies using SPARQL queries. It is designed to allow users to perform key reasoning operations, such as inference and consistency checking, through a clean and accessible user experience.

5.1.1 Overview

The platform is implemented using `Next.js` (Vercel, n.d.) and `TypeScript` (Microsoft, n.d.) for the frontend, offering a responsive and modern web interface. The backend functionality is handled through the `GraphDB` REST API (Ontotext, n.d.-b), connected to a local instance of GraphDB running inside a Docker (Docker Inc., n.d.) container. This instance serves both as the SPARQL endpoint and the ontology storage system.

The application consists of four main pages:

- **Home Page:** Presents an overview and showcases the queries.

- **Inspect Page:** Displays general information and statistics about the ontology.
- **Expand Page:** Provides access to inference rules implemented as SPARQL `CONSTRUCT` queries.
- **Check Page:** Offers consistency-checking queries to detect potential logical issues.

5.1.2 Interaction

The interaction between the user, the platform, and the ontology follows a simple workflow. Users interact with the web interface to select a reasoning or inspection operation. This triggers a predefined SPARQL query that is sent to the GraphDB SPARQL endpoint via the API. GraphDB processes the query and returns the results to the frontend for display.

In the case of inference queries (on the Expand Page), the user is also given the option to insert the inferred triples into the ontology using a SPARQL `INSERT DATA` operation, effectively expanding the ontology with newly derived knowledge.

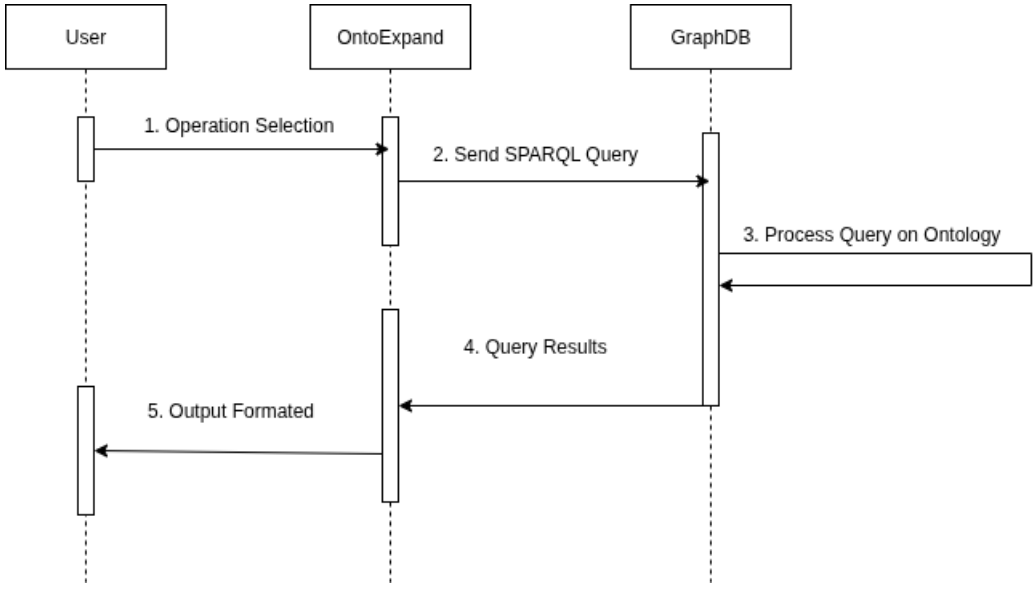


Figure 2: Sequence Diagram

5.1.3 Design Goals and features

The design of the web platform was guided by usability and clarity, with a strong emphasis on making ontology reasoning operations accessible to users without requiring in-depth knowledge of SPARQL or ontology structures. The interface was heavily inspired by the native GraphDB Workbench (Ontotext, n.d.-a) to maintain visual and functional cohesion with the underlying service.

Navigation throughout the platform is straightforward, focusing on clear task separation. Each reasoning operation or query type is grouped under its own dedicated page, allowing users to focus on one category of interaction at a time. The layout prioritizes readability and ease of use, presenting results in a structured format and providing direct controls for operations such as expanding the ontology with inferred triples.

Home Page

The Home Page serves as the entry point to the platform. It provides an overview of the application's purpose and capabilities, alongside a showcase of the available query types grouped by page. This acts as both an introduction and a quick-access menu, allowing users to explore the system's reasoning features and navigate directly to specific operations.

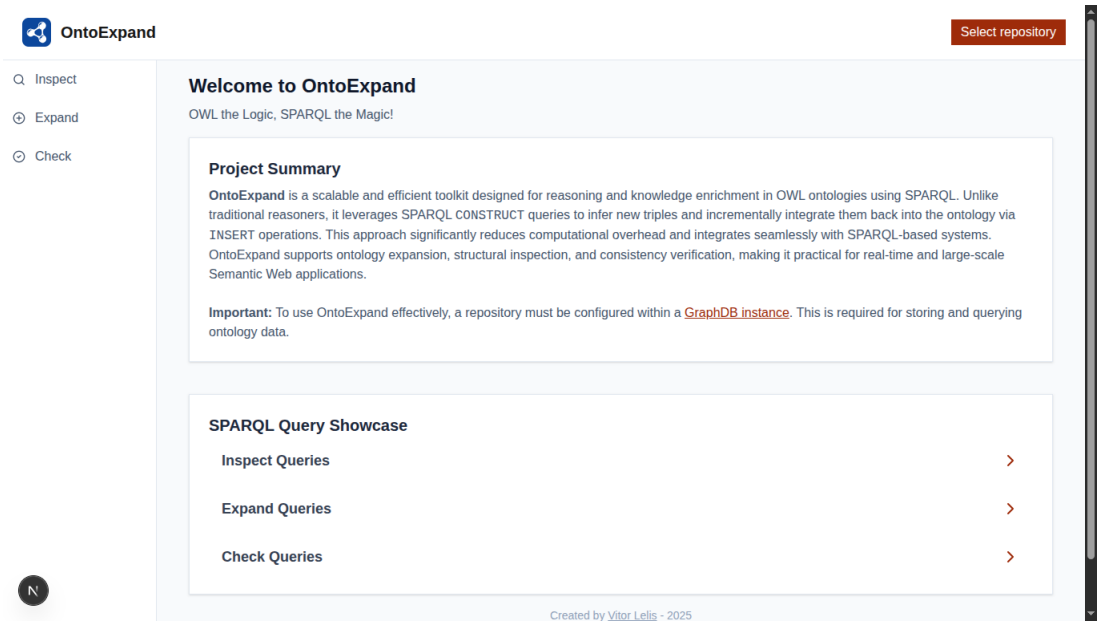


Figure 3: Home Page of the Web Platform

Inspect Page

The Inspect Page offers tools for exploring the existing contents of the ontology. It includes a set of predefined SELECT queries that summarize aspects such as classes, properties, and instances. Additionally, it provides an IRI search feature where users can input a specific resource IRI to retrieve its associated triples. This page allows users to gain insight into the structure and content of the ontology before performing reasoning tasks.

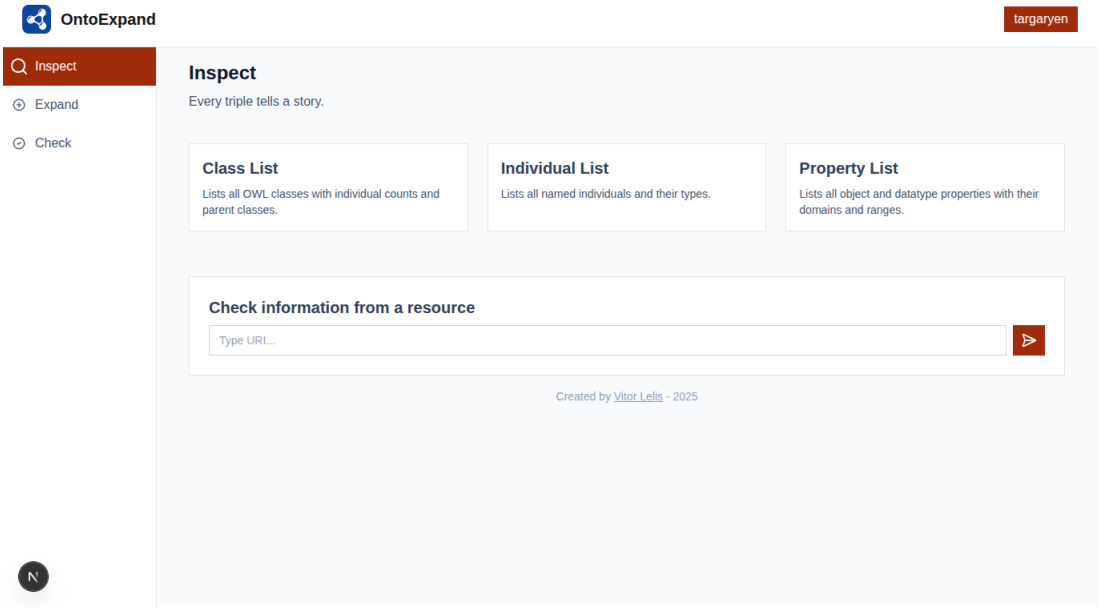


Figure 4: Inspect Page

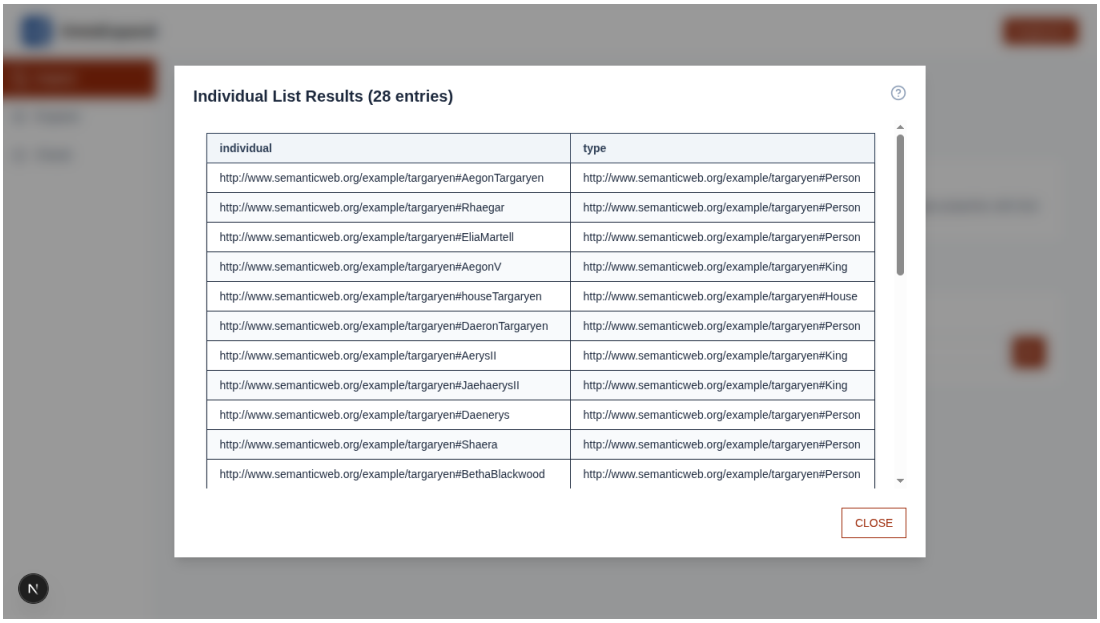


Figure 5: Results of the Individual List Query

Figure 6: Results of Resource information check

```
SELECT ?individual ?type
WHERE {
  ?individual a owl:NamedIndividual .
  OPTIONAL {
    ?individual a ?type .
    FILTER(?type != owl:NamedIndividual)
  }
}
```

Listing 5.1: Individual list with their Class

This query selects all individuals explicitly declared as `owl:NamedIndividual`. For each such individual, it optionally retrieves any additional types they may have, excluding the generic `owl:NamedIndividual` type itself. This allows the query to return both the individual and any more specific class assertions associated with it, providing a detailed view of the individual's classification without redundantly listing the base `owl:NamedIndividual` type.

```
SELECT ?class (COUNT(?individual) AS ?individualCount) ?parentClass
WHERE {
  ?class a owl:Class .
  FILTER(isIRI(?class))

  OPTIONAL {
    ?class rdfs:subClassOf ?parentClass .
    FILTER(isIRI(?parentClass))
  }

  OPTIONAL { ?individual a ?class }
}
GROUP BY ?class ?parentClass
ORDER BY ?class
```

Listing 5.2: Class list with individual count and parent class

This query retrieves all classes declared as `owl:Class`, ensuring that only IRI-identified classes are considered. For each class, it optionally identifies its parent class via `rdfs:subClassOf`, again filtering for IRIs, and counts the number of individuals that are instances of that class. The results are grouped by class and parent class, allowing the query to report both the hierarchical structure and the instance count for each class. Finally, the output is ordered by class for easier inspection.

```
# By a given <{id}>
SELECT ?p ?o
WHERE {
  <{id}> ?p ?o .
}
```

Listing 5.3: Resource Information query

This query retrieves all triples where a specific individual or resource, identified by `<id>`, appears as the subject. For each such triple, it returns the predicate `?p` and the corresponding object `?o`, effectively listing all properties and values associated with the given resource.

```
SELECT ?property ?type ?domain ?range
WHERE {
  { ?property a owl:ObjectProperty . } UNION { ?property a owl:DatatypeProperty
      . }
  ?property a ?type .

  OPTIONAL { ?property rdfs:domain ?domain . }
  OPTIONAL { ?property rdfs:range ?range . }
}
ORDER BY ?property
```

Listing 5.4: Property list with their type, domain and range

This query retrieves all properties declared as either `owl:ObjectProperty` or `owl:DatatypeProperty` and returns their specific type. Additionally, it optionally retrieves the `rdfs:domain` and `rdfs:range` of each property, providing information about the classes the property applies to and the types of values it can take. The results are ordered by property for easier inspection.

Expand Page

The Expand Page presents a collection of most of the inference queries previously defined in section 4.2. Once a query is run, the resulting triples are displayed on screen. The user is then given the option to insert these inferred triples into the ontology using a corresponding `INSERT DATA` query, effectively expanding the ontology with newly derived knowledge.

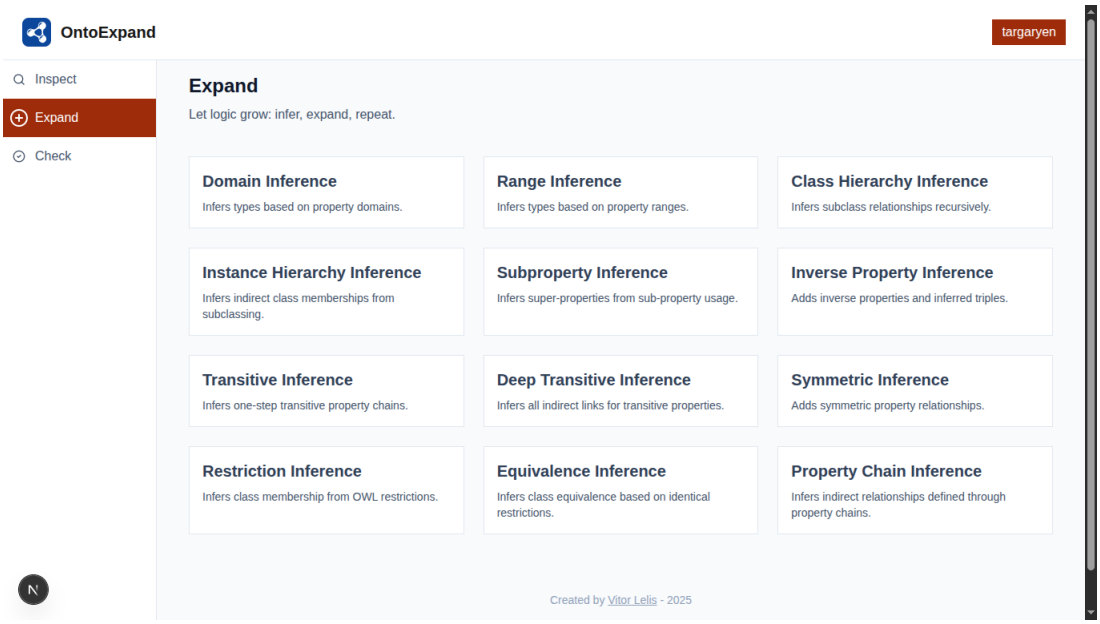


Figure 7: Expand Page

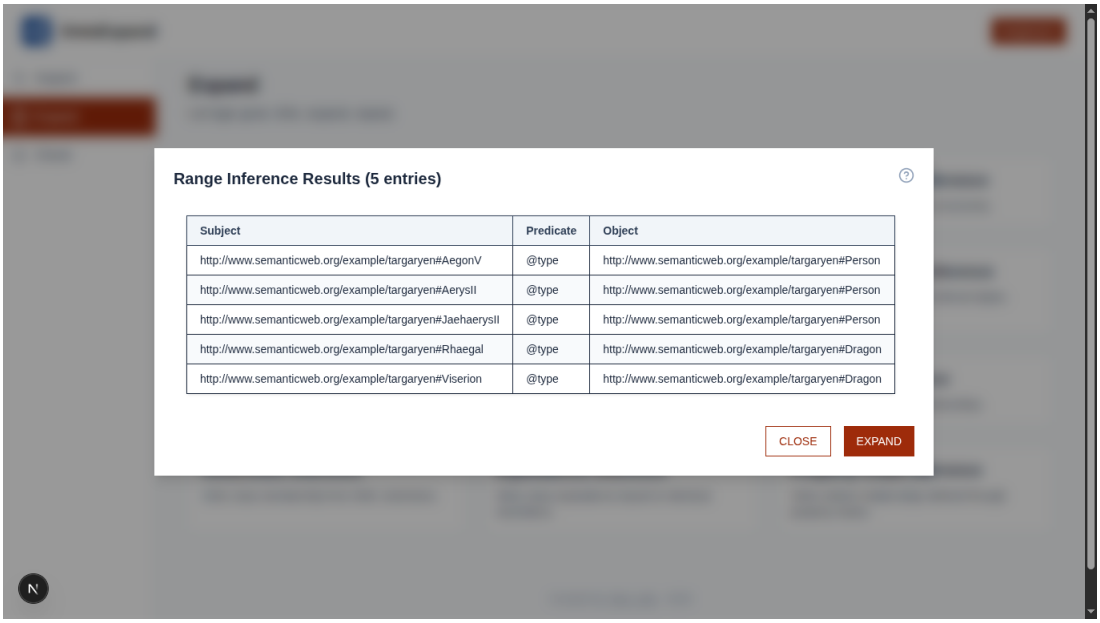


Figure 8: Result of Inference by Range

```
INSERT DATA {
  <http://www.semanticweb.org/example/targaryen#AegonV> a <http://www.
      semanticweb.org/example/targaryen#Person> .
  <http://www.semanticweb.org/example/targaryen#AerysII> a <http://www.
      semanticweb.org/example/targaryen#Person> .
  <http://www.semanticweb.org/example/targaryen#JaehaerysII> a <http://www.
      semanticweb.org/example/targaryen#Person> .
  <http://www.semanticweb.org/example/targaryen#Rhaegal> a <http://www.
      semanticweb.org/example/targaryen#Dragon> .
  <http://www.semanticweb.org/example/targaryen#Viserion> a <http://www.
      semanticweb.org/example/targaryen#Dragon> .
}
```

Listing 5.5: INSERT query with the results of CONSTRUCT

Check Page

The Check Page is dedicated to consistency checking. It features the list of queries from subsection 4.2.9 that help detect potential inconsistencies in the ontology, such as class disjointness violations or malformed relations. By identifying these logical errors, the platform helps users maintain the semantic integrity of their ontologies.

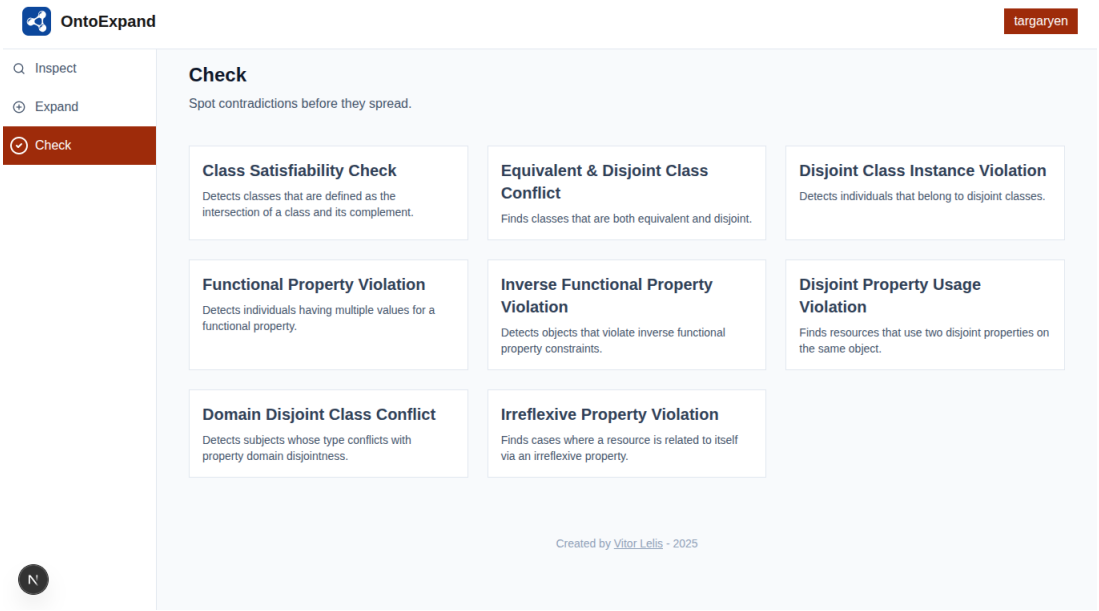


Figure 9: Check Page

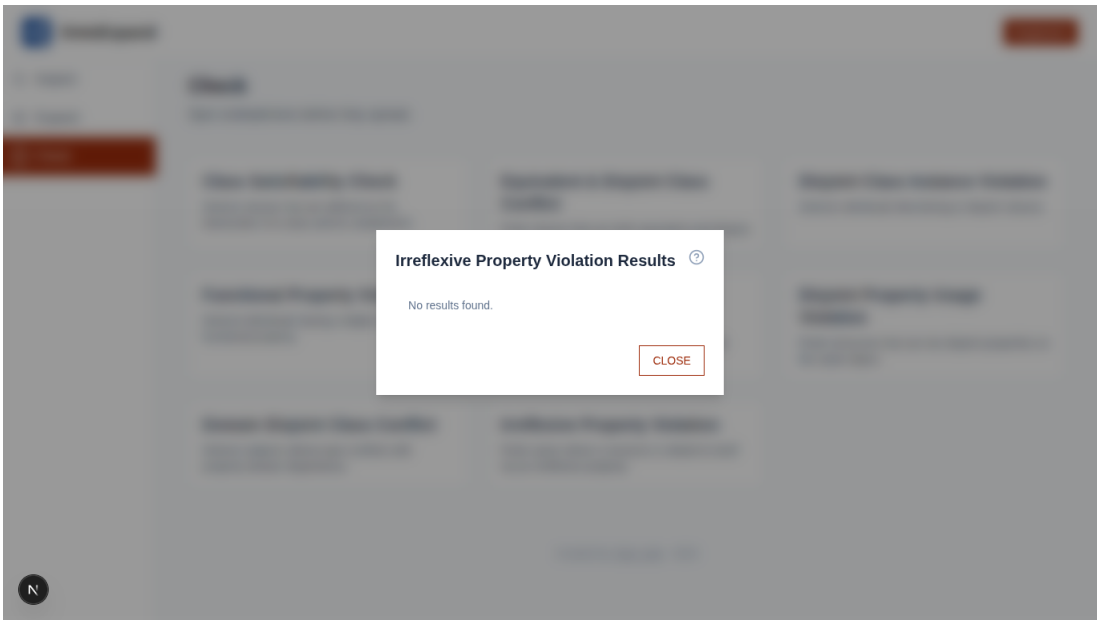


Figure 10: Result of Irreflexive Property Violation Check

5.2 Implementation Challenges and Limitations

While the web platform successfully demonstrates SPARQL-based reasoning over ontologies, several challenges and limitations were encountered during its development and use.

One significant issue involved handling the output of different types of SPARQL queries. Since the platform supports `SELECT`, `CONSTRUCT`, and `INSERT` queries, each produces structurally distinct results. To address this, dedicated formatting functions were developed to normalize and present query results consistently throughout the application, ensuring a uniform user experience regardless of query type.

Another key limitation lies in the platform's dependency on the underlying GraphDB configuration. The application itself does not include mechanisms to manage the GraphDB instance. For the platform to function properly, a repository must be created manually within GraphDB, and the target ontology must be imported in advance. This external setup step is required and may present a barrier for less technical users or those unfamiliar with GraphDB.

Finally, while the current implementation successfully covers core reasoning and validation operations, it remains a simplified prototype. Expanding the variety and complexity of supported queries and operations would enrich the platform's capabilities and make it a more versatile tool for ontology management. Future versions could benefit from broader rule support, user-defined query options, and tighter integration with the backend reasoning layer.

5.3 Summary

The Ontology Expansion via Web Interface chapter presented the development of a practical platform that integrates SPARQL-based reasoning into an interactive environment. Building on the earlier experimental demonstrations, the system allows users to perform inference, ontology expansion, and consistency checking directly through a web interface, backed by GraphDB. The chapter detailed the design choices, functionality, and challenges encountered, such as query result handling and repository dependencies. Overall, this prototype showcased how theoretical reasoning queries can be transformed into a usable tool for ontology management, setting the stage for the final reflections and directions for future work.

Chapter 6

Conclusions and Future Work

This final chapter concludes the thesis by reflecting on the contributions achieved and the limitations encountered throughout the project. It synthesizes the theoretical and practical findings, emphasizing how SPARQL can serve not only as a query language but also as a lightweight reasoning framework. Furthermore, the chapter discusses potential avenues for extending the approach, improving the platform, and fostering broader adoption, ultimately situating the work within the larger Semantic Web landscape.

6.1 Summary of Contributions

The primary objective of this work was to explore the use of SPARQL as a means to replicate and potentially replace traditional ontology reasoners. This approach aimed to deliver a more efficient, scalable, and SPARQL-native solution for performing core reasoning operations over ontologies.

The project was carried out in two main phases. The first phase focused on the theoretical development of SPARQL `CONSTRUCT` queries to simulate reasoning behaviors. Various reasoning tasks, such as inferring subclass relationships, properties definitions, and detecting inconsistencies, were implemented using SPARQL alone. This exploration demonstrated that key aspects of OWL reasoning could be encoded directly into SPARQL query logic. In some cases, due to SPARQL's expressiveness limitations, certain reasoning operations required a combination of two queries (subsection 4.2.3, subsection 4.2.4 and subsection 4.2.8) to fully capture the desired behavior.

The second phase involved the development of a web-based platform to showcase these reasoning operations in practice. The platform provides users with an interface to execute predefined reasoning queries, view results, and update the ontology with inferred knowledge. It successfully demonstrates how theoretical reasoning mechanisms can be made accessible, interactive, and actionable through modern web technologies.

Together, these phases represent both theoretical and practical contributions:

- **Theoretical:** A systematic exploration of SPARQL `CONSTRUCT` queries for simulating common reasoning tasks.
- **Practical:** A functional web application that integrates these queries into an interactive platform for ontology expansion and validation.

The strengths of this approach lie in its transparency, integration with existing SPARQL infrastructure, and avoidance of external reasoning engines. It offers a lightweight and understandable alternative for specific reasoning tasks.

However, both phases also encountered limitations. The first phase was bounded by SPARQL's expressive power, requiring workarounds for some operations. The second phase, in turn, was limited by its dependency on manual GraphDB configuration and a constrained query set.

6.2 Future Work

Several directions remain open for future improvement and expansion of this work:

- **Extended Query Coverage:** The platform could be enhanced by including additional queries that cover a broader set of reasoning operations, including more complex OWL constructs and custom rule patterns.
- **Web Tool Enhancements:** Features such as user-defined rule creation, result filtering, and history tracking could significantly improve usability and flexibility.
- **Reduced Dependency on GraphDB:** Making the system compatible with other triple stores or allowing automated repository setup and ontology loading would ease deployment and improve portability.
- **Broader Adoption Potential:** Packaging the platform for public deployment or integrating it into educational tools for ontology learning could increase its visibility and adoption within the Semantic Web community.
- **Benchmarking and Performance Evaluation:** Introducing a benchmarking component to measure and compare the execution time of each query would provide valuable insights into system performance.

6.3 Final Remarks

This project illustrates the practical potential of bridging theory and implementation within the context of the Semantic Web. By leveraging the power of SPARQL not only as a query language but as a reasoning engine, we have shown that core semantic tasks can be executed efficiently and transparently within existing infrastructure.

While traditional reasoners remain essential for full OWL reasoning, this work provides a compelling case for using SPARQL-based approaches in scenarios that prioritize integration, performance, and simplicity. The combination of theoretical modeling and real-world tooling presented here contributes to the growing set of lightweight, modular solutions in the Semantic Web ecosystem.

Bibliography

Antoniou, G., & van Harmelen, F. (2003). Web ontology language: OWL. In S. Staab & R. Studer (Eds.), *Handbook on ontologies* (pp. 91–110). Springer. https://doi.org/10.1007/978-3-540-92673-3_4

Arndt, D., Van Woensel, W., & Tomaszuk, D. (2023). SiN3: Scalable inferencing with SPARQL CONSTRUCT queries [Demo paper]. *Proceedings of the 22nd International Semantic Web Conference (ISWC 2023)*. https://ceur-ws.org/Vol-3513/

Barbieri, D. F., Braga, D., Ceri, S., Valle, E. D., & Grossniklaus, M. (2010). Querying RDF streams with C-SPARQL. *ACM SIGMOD Record*, *39*(1), 20. https://doi.org/10.1145/1860702.1860705

Bischof, S., Krötzsch, M., Polleres, A., & Rudolph, S. (2014). Schema-agnostic query rewriting in SPARQL 1.1. In P. Mika, T. Tudorache, A. Bernstein, C. Welty, C. Knoblock, D. Vrandečić, P. Groth, N. Noy, K. Janowicz, & C. Goble (Eds.), *The semantic web – ISWC 2014* (pp. 584–600). Springer.

Dhulekar, K., & Devrankar, M. (2020). A review on semantic web. *International Journal of Engineering Technologies and Management Research*, *6*, 22–28. https://doi.org/10.29121/ijetmr.v6.i12.2019.470

Docker Inc. (n.d.). *Docker overview*. Retrieved June 25, 2025, from https://docs.docker.com/get-started/docker-overview/

Frade, M. J. (2009). *Classical first-order logic [lecture slides]* [Course: Software Formal Verification, University of Minho]. Retrieved June 21, 2025, from https://www.di.uminho.pt/~mjf/pub/SFV-FOL-2up.pdf

Glimm, B. (2011). Using SPARQL with RDFS and OWL entailment. In A. Polleres, C. d'Amato, M. Arenas, S. Handschuh, P. Kröner, S. Ossowski, & P. Patel-Schneider (Eds.), *Reasoning web: Semantic technologies for the web of data* (pp. 137–201). Springer. https://doi.org/10.1007/978-3-642-23032-5_3

Glimm, B., Horrocks, I., Motik, B., Stoilos, G., & Wang, Z. (2014). HermiT: An OWL 2 reasoner. *Journal of Automated Reasoning*, *53*(3), 245–269. https://doi.org/10.1007/s10817-014-9305-1

Jing, Y., Jeong, D., & Baik, D.-K. (2009). SPARQL graph pattern rewriting for OWL-DL inference queries. *Knowledge and Information Systems*, *20*(2), 243–262. https://doi.org/10.1007/s10115-008-0169-8

Kramer, D., & Augusto, J. C. (2017). Supporting context-aware engineering based on stream reasoning. In *Lecture notes in computer science* (pp. 440–453). Springer. https://doi.org/10.1007/978-3-319-57837-8_37

Microsoft. (n.d.). *Typescript documentation*. Retrieved June 18, 2025, from https://www.typescriptlang.org/docs/

Mishra, R. B., & Kumar, S. (2011). Semantic web reasoners and languages. *Artificial Intelligence Review*, *35*(4), 339–368. https://doi.org/10.1007/s10462-010-9197-3

Ontotext. (n.d.-a). *Graphdb workbench (v10.1) documentation*. Retrieved June 19, 2025, from https://graphdb.ontotext.com/documentation/10.1/graphdb-workbench.html

Ontotext. (n.d.-b). *Using the GraphDB REST API (v11.0)*. Retrieved June 20, 2025, from https://graphdb.ontotext.com/documentation/11.0/using-the-graphdb-rest-api.html

Ontotext. (2025a). *What are ontologies?* Retrieved January 22, 2025, from https://www.ontotext.com/knowledgehub/fundamentals/what-are-ontologies/
Ontotext. (2025b). *What is a knowledge graph?* Retrieved January 22, 2025, from https://www.ontotext.com/knowledgehub/fundamentals/what-is-a-knowledge-graph/
University of Manchester. (n.d.). *Why OWL?* Retrieved March 19, 2025, from http://owl.man.ac.uk/2003/why/latest/
Vercel. (n.d.). *Next.js documentation*. Retrieved June 18, 2025, from https://nextjs.org/docs
W3C. (n.d.-a). *OWL 2 web ontology language primer (second edition)*. Retrieved May 6, 2025, from https://www.w3.org/TR/owl2-primer/
W3C. (n.d.-b). *OWL web ontology language reference*. Retrieved May 21, 2025, from https://www.w3.org/TR/owl-ref/
W3C. (n.d.-c). *SPARQL 1.1 property paths*. Retrieved June 2, 2025, from https://www.w3.org/TR/sparql11-property-paths/
W3C. (n.d.-d). *SPARQL 1.1 update*. Retrieved May 23, 2025, from https://www.w3.org/TR/sparql11-update/
W3C. (2012). *OWL 2 web ontology language document overview (second edition)*. Retrieved October 29, 2024, from https://www.w3.org/TR/2012/REC-owl2-overview-20121211/#Documentation_Roadmap
W3C. (2013). *SPARQL 1.1 query language*. Retrieved October 29, 2024, from https://www.w3.org/TR/sparql11-query/
W3C. (2014). *RDF schema 1.1*. Retrieved February 3, 2024, from https://www.w3.org/TR/rdf-schema/

Appendices

Appendix A
Listings

A.1 Targaryen Ontology RDF/XML

Listing A.1: Targaryen Ontology

```
<?xml version="1.0"?>
<rdf:RDF xmlns="http://www.semanticweb.org/example/targaryen#"
     xml:base="http://www.semanticweb.org/example/targaryen"
     xmlns:owl="http://www.w3.org/2002/07/owl#"
     xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
     xmlns:xml="http://www.w3.org/XML/1998/namespace"
     xmlns:xsd="http://www.w3.org/2001/XMLSchema#"
     xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#">
    <owl:Ontology rdf:about="http://www.semanticweb.org/example/targaryen"/>
    
    
    
    
    <!-- http://www.semanticweb.org/example/targaryen#belongsToHouse -->
    
    <owl:ObjectProperty rdf:about="http://www.semanticweb.org/example/targaryen#
        belongsToHouse">
        <rdfs:domain rdf:resource="http://www.semanticweb.org/example/targaryen#Noble"/
            >
        <rdfs:range rdf:resource="http://www.semanticweb.org/example/targaryen#House"/>
    </owl:ObjectProperty>
    
    
    
    <!-- http://www.semanticweb.org/example/targaryen#hasAncestor -->
    
    <owl:ObjectProperty rdf:about="http://www.semanticweb.org/example/targaryen#
        hasAncestor">
        <rdf:type rdf:resource="http://www.w3.org/2002/07/owl#TransitiveProperty"/>
        <rdfs:domain rdf:resource="http://www.semanticweb.org/example/targaryen#Person"
            />
        <rdfs:range rdf:resource="http://www.semanticweb.org/example/targaryen#Person"/
            >
    </owl:ObjectProperty>
    
    
    
    <!-- http://www.semanticweb.org/example/targaryen#hasChild -->
    
```

```
35      <owl:ObjectProperty rdf:about="http://www.semanticweb.org/example/targaryen#
            hasChild">
36          <owl:inverseOf rdf:resource="http://www.semanticweb.org/example/targaryen#
                hasParent"/>
37          <rdfs:domain rdf:resource="http://www.semanticweb.org/example/targaryen#Person"
                />
38          <rdfs:range rdf:resource="http://www.semanticweb.org/example/targaryen#Person"/
                >
39      </owl:ObjectProperty>
40
41
42
43      <!-- http://www.semanticweb.org/example/targaryen#hasDragon -->
44
45      <owl:ObjectProperty rdf:about="http://www.semanticweb.org/example/targaryen#
            hasDragon">
46          <rdfs:domain rdf:resource="http://www.semanticweb.org/example/targaryen#Person"
                />
47          <rdfs:range rdf:resource="http://www.semanticweb.org/example/targaryen#Dragon"/
                >
48      </owl:ObjectProperty>
49
50
51
52      <!-- http://www.semanticweb.org/example/targaryen#hasFather -->
53
54      <owl:ObjectProperty rdf:about="http://www.semanticweb.org/example/targaryen#
            hasFather">
55          <rdfs:subPropertyOf rdf:resource="http://www.semanticweb.org/example/targaryen#
                hasParent"/>
56          <rdfs:domain rdf:resource="http://www.semanticweb.org/example/targaryen#Person"
                />
57          <rdfs:range rdf:resource="http://www.semanticweb.org/example/targaryen#Person"/
                >
58          <owl:propertyDisjointWith rdf:resource="http://www.semanticweb.org/example/
                targaryen#hasMother"/>
59      </owl:ObjectProperty>
60
61
62
63      <!-- http://www.semanticweb.org/example/targaryen#hasGrandchild -->
64
65      <owl:ObjectProperty rdf:about="http://www.semanticweb.org/example/targaryen#
            hasGrandchild">
66          <rdfs:domain rdf:resource="http://www.semanticweb.org/example/targaryen#Person"
                />
67          <rdfs:range rdf:resource="http://www.semanticweb.org/example/targaryen#Person"/
                >
68          <owl:propertyChainAxiom rdf:parseType="Collection">
69              <rdf:Description rdf:about="http://www.semanticweb.org/example/targaryen#
                    hasChild"/>
70              <rdf:Description rdf:about="http://www.semanticweb.org/example/targaryen#
                    hasChild"/>
71          </owl:propertyChainAxiom>
72      </owl:ObjectProperty>
73
74
75
```

```
    <!-- http://www.semanticweb.org/example/targaryen#hasGrandparent -->

    <owl:ObjectProperty rdf:about="http://www.semanticweb.org/example/targaryen#
        hasGrandparent">
        <rdfs:domain rdf:resource="http://www.semanticweb.org/example/targaryen#Person"
            />
        <rdfs:range rdf:resource="http://www.semanticweb.org/example/targaryen#Person"/
            >
        <owl:propertyChainAxiom rdf:parseType="Collection">
            <rdf:Description rdf:about="http://www.semanticweb.org/example/targaryen#
                hasParent"/>
            <rdf:Description rdf:about="http://www.semanticweb.org/example/targaryen#
                hasParent"/>
        </owl:propertyChainAxiom>
    </owl:ObjectProperty>



    <!-- http://www.semanticweb.org/example/targaryen#hasMother -->

    <owl:ObjectProperty rdf:about="http://www.semanticweb.org/example/targaryen#
        hasMother">
        <rdfs:subPropertyOf rdf:resource="http://www.semanticweb.org/example/targaryen#
            hasParent"/>
        <rdfs:domain rdf:resource="http://www.semanticweb.org/example/targaryen#Person"
            />
        <rdfs:range rdf:resource="http://www.semanticweb.org/example/targaryen#Person"/
            >
    </owl:ObjectProperty>



    <!-- http://www.semanticweb.org/example/targaryen#hasParent -->

    <owl:ObjectProperty rdf:about="http://www.semanticweb.org/example/targaryen#
        hasParent">
        <rdfs:domain rdf:resource="http://www.semanticweb.org/example/targaryen#Person"
            />
    </owl:ObjectProperty>



    <!-- http://www.semanticweb.org/example/targaryen#hasSibling -->

    <owl:ObjectProperty rdf:about="http://www.semanticweb.org/example/targaryen#
        hasSibling">
        <rdf:type rdf:resource="http://www.w3.org/2002/07/owl#IrreflexiveProperty"/>
        <owl:propertyChainAxiom rdf:parseType="Collection">
            <rdf:Description rdf:about="http://www.semanticweb.org/example/targaryen#
                hasParent"/>
            <rdf:Description rdf:about="http://www.semanticweb.org/example/targaryen#
                hasChild"/>
        </owl:propertyChainAxiom>
    </owl:ObjectProperty>



    <!-- http://www.semanticweb.org/example/targaryen#isAllyOf -->
```

```
120
121     <owl:ObjectProperty rdf:about="http://www.semanticweb.org/example/targaryen#
            isAllyOf">
122         <rdf:type rdf:resource="http://www.w3.org/2002/07/owl#SymmetricProperty"/>
123         <rdfs:domain rdf:resource="http://www.semanticweb.org/example/targaryen#House"/
                >
124         <rdfs:range rdf:resource="http://www.semanticweb.org/example/targaryen#House"/>
125     </owl:ObjectProperty>
126
127
128
129
130     <!-- http://www.semanticweb.org/example/targaryen#inches -->
131
132     <owl:DatatypeProperty rdf:about="http://www.semanticweb.org/example/targaryen#
            inches">
133         <rdfs:domain rdf:resource="http://www.semanticweb.org/example/targaryen#Sword"/
                >
134         <rdfs:range rdf:resource="http://www.w3.org/2001/XMLSchema#integer"/>
135     </owl:DatatypeProperty>
136
137
138
139
140
141     <!-- http://www.semanticweb.org/example/targaryen#Dragon -->
142
143     <owl:Class rdf:about="http://www.semanticweb.org/example/targaryen#Dragon"/>
144
145
146
147     <!-- http://www.semanticweb.org/example/targaryen#Dragonlord -->
148
149     <owl:Class rdf:about="http://www.semanticweb.org/example/targaryen#Dragonlord">
150         <owl:equivalentClass>
151             <owl:Class>
152                 <owl:intersectionOf rdf:parseType="Collection">
153                     <rdf:Description rdf:about="http://www.semanticweb.org/example/
                            targaryen#Person"/>
154                     <owl:Restriction>
155                         <owl:onProperty rdf:resource="http://www.semanticweb.org/example/
                                targaryen#hasDragon"/>
156                         <owl:someValuesFrom rdf:resource="http://www.semanticweb.org/
                                example/targaryen#Dragon"/>
157                     </owl:Restriction>
158                 </owl:intersectionOf>
159             </owl:Class>
160         </owl:equivalentClass>
161     </owl:Class>
162
163
164
165     <!-- http://www.semanticweb.org/example/targaryen#Dragonrider -->
166
167     <owl:Class rdf:about="http://www.semanticweb.org/example/targaryen#Dragonrider">
168         <owl:equivalentClass>
169             <owl:Class>
170                 <owl:intersectionOf rdf:parseType="Collection">
```

```
                <rdf:Description rdf:about="http://www.semanticweb.org/example/
                    targaryen#Person"/>
                <owl:Restriction>
                    <owl:onProperty rdf:resource="http://www.semanticweb.org/example/
                        targaryen#hasDragon"/>
                    <owl:someValuesFrom rdf:resource="http://www.semanticweb.org/
                        example/targaryen#Dragon"/>
                </owl:Restriction>
            </owl:intersectionOf>
        </owl:Class>
    </owl:equivalentClass>
</owl:Class>



<!-- http://www.semanticweb.org/example/targaryen#House -->

<owl:Class rdf:about="http://www.semanticweb.org/example/targaryen#House"/>



<!-- http://www.semanticweb.org/example/targaryen#King -->

<owl:Class rdf:about="http://www.semanticweb.org/example/targaryen#King">
    <rdfs:subClassOf rdf:resource="http://www.semanticweb.org/example/targaryen#
        Noble"/>
</owl:Class>



<!-- http://www.semanticweb.org/example/targaryen#Longsword -->

<owl:Class rdf:about="http://www.semanticweb.org/example/targaryen#Longsword">
    <owl:equivalentClass>
        <owl:Class>
            <owl:intersectionOf rdf:parseType="Collection">
                <rdf:Description rdf:about="http://www.semanticweb.org/example/
                    targaryen#Sword"/>
                <owl:Restriction>
                    <owl:onProperty rdf:resource="http://www.semanticweb.org/example/
                        targaryen#inches"/>
                    <owl:someValuesFrom rdf:resource="http://www.w3.org/2001/XMLSchema
                        #integer"/>
                </owl:Restriction>
            </owl:intersectionOf>
        </owl:Class>
    </owl:equivalentClass>
</owl:Class>



<!-- http://www.semanticweb.org/example/targaryen#Noble -->

<owl:Class rdf:about="http://www.semanticweb.org/example/targaryen#Noble">
    <rdfs:subClassOf rdf:resource="http://www.semanticweb.org/example/targaryen#
        Person"/>
</owl:Class>

```

```
221
222
223     <!-- http://www.semanticweb.org/example/targaryen#Person -->
224
225     <owl:Class rdf:about="http://www.semanticweb.org/example/targaryen#Person"/>
226
227
228
229     <!-- http://www.semanticweb.org/example/targaryen#Sword -->
230
231     <owl:Class rdf:about="http://www.semanticweb.org/example/targaryen#Sword"/>
232
233
234
235     <!-- http://www.semanticweb.org/example/targaryen#AegonTargaryen -->
236
237     <owl:NamedIndividual rdf:about="http://www.semanticweb.org/example/targaryen#
            AegonTargaryen">
238         <rdf:type rdf:resource="http://www.semanticweb.org/example/targaryen#Person"/>
239         <hasFather rdf:resource="http://www.semanticweb.org/example/targaryen#Rhaegar"/
                >
240         <hasMother rdf:resource="http://www.semanticweb.org/example/targaryen#
                EliaMartell"/>
241     </owl:NamedIndividual>
242
243
244
245     <!-- http://www.semanticweb.org/example/targaryen#AegonV -->
246
247     <owl:NamedIndividual rdf:about="http://www.semanticweb.org/example/targaryen#AegonV
            ">
248         <rdf:type rdf:resource="http://www.semanticweb.org/example/targaryen#King"/>
249         <belongsToHouse rdf:resource="http://www.semanticweb.org/example/targaryen#
                houseTargaryen"/>
250         <hasChild rdf:resource="http://www.semanticweb.org/example/targaryen#
                DaeronTargaryen"/>
251     </owl:NamedIndividual>
252
253
254
255     <!-- http://www.semanticweb.org/example/targaryen#AerysII -->
256
257     <owl:NamedIndividual rdf:about="http://www.semanticweb.org/example/targaryen#
            AerysII">
258         <rdf:type rdf:resource="http://www.semanticweb.org/example/targaryen#King"/>
259         <belongsToHouse rdf:resource="http://www.semanticweb.org/example/targaryen#
                houseTargaryen"/>
260         <hasAncestor rdf:resource="http://www.semanticweb.org/example/targaryen#
                JaehaerysII"/>
261         <hasChild rdf:resource="http://www.semanticweb.org/example/targaryen#Daenerys"/
                >
262         <hasChild rdf:resource="http://www.semanticweb.org/example/targaryen#Rhaegar"/>
263         <hasFather rdf:resource="http://www.semanticweb.org/example/targaryen#
                JaehaerysII"/>
264         <hasMother rdf:resource="http://www.semanticweb.org/example/targaryen#Shaera"/>
265     </owl:NamedIndividual>
266
267
```

```
268
269     <!-- http://www.semanticweb.org/example/targaryen#BethaBlackwood -->
270
271     <owl:NamedIndividual rdf:about="http://www.semanticweb.org/example/targaryen#
            BethaBlackwood">
272         <rdf:type rdf:resource="http://www.semanticweb.org/example/targaryen#Person"/>
273     </owl:NamedIndividual>
274
275
276
277     <!-- http://www.semanticweb.org/example/targaryen#CassanaEstermont -->
278
279     <owl:NamedIndividual rdf:about="http://www.semanticweb.org/example/targaryen#
            CassanaEstermont">
280         <rdf:type rdf:resource="http://www.semanticweb.org/example/targaryen#Person"/>
281     </owl:NamedIndividual>
282
283
284
285     <!-- http://www.semanticweb.org/example/targaryen#Daenerys -->
286
287     <owl:NamedIndividual rdf:about="http://www.semanticweb.org/example/targaryen#
            Daenerys">
288         <rdf:type rdf:resource="http://www.semanticweb.org/example/targaryen#Person"/>
289         <belongsToHouse rdf:resource="http://www.semanticweb.org/example/targaryen#
                houseTargaryen"/>
290         <hasAncestor rdf:resource="http://www.semanticweb.org/example/targaryen#AerysII
                "/>
291         <hasDragon rdf:resource="http://www.semanticweb.org/example/targaryen#Drogon"/>
292         <hasDragon rdf:resource="http://www.semanticweb.org/example/targaryen#Rhaegal"/
                >
293         <hasDragon rdf:resource="http://www.semanticweb.org/example/targaryen#Viserion"
                />
294         <hasFather rdf:resource="http://www.semanticweb.org/example/targaryen#AerysII"/
                >
295         <hasMother rdf:resource="http://www.semanticweb.org/example/targaryen#Rhaella"/
                >
296         <hasParent rdf:resource="http://www.semanticweb.org/example/targaryen#AerysII"/
                >
297     </owl:NamedIndividual>
298
299
300
301     <!-- http://www.semanticweb.org/example/targaryen#DaeronTargaryen -->
302
303     <owl:NamedIndividual rdf:about="http://www.semanticweb.org/example/targaryen#
            DaeronTargaryen">
304         <rdf:type rdf:resource="http://www.semanticweb.org/example/targaryen#Person"/>
305         <hasMother rdf:resource="http://www.semanticweb.org/example/targaryen#
                BethaBlackwood"/>
306     </owl:NamedIndividual>
307
308
309
310     <!-- http://www.semanticweb.org/example/targaryen#Drogon -->
311
312     <owl:NamedIndividual rdf:about="http://www.semanticweb.org/example/targaryen#Drogon
            ">
```

```
        <rdf:type rdf:resource="http://www.semanticweb.org/example/targaryen#Dragon"/>
    </owl:NamedIndividual>



    <!-- http://www.semanticweb.org/example/targaryen#DuncanTargaryen -->

    <owl:NamedIndividual rdf:about="http://www.semanticweb.org/example/targaryen#
        DuncanTargaryen">
        <rdf:type rdf:resource="http://www.semanticweb.org/example/targaryen#Person"/>
        <hasFather rdf:resource="http://www.semanticweb.org/example/targaryen#AegonV"/>
        <hasMother rdf:resource="http://www.semanticweb.org/example/targaryen#
            BethaBlackwood"/>
    </owl:NamedIndividual>



    <!-- http://www.semanticweb.org/example/targaryen#EliaMartell -->

    <owl:NamedIndividual rdf:about="http://www.semanticweb.org/example/targaryen#
        EliaMartell">
        <rdf:type rdf:resource="http://www.semanticweb.org/example/targaryen#Person"/>
    </owl:NamedIndividual>



    <!-- http://www.semanticweb.org/example/targaryen#Ice -->

    <owl:NamedIndividual rdf:about="http://www.semanticweb.org/example/targaryen#Ice">
        <rdf:type rdf:resource="http://www.semanticweb.org/example/targaryen#Sword"/>
        <inches rdf:datatype="http://www.w3.org/2001/XMLSchema#integer">40</inches>
    </owl:NamedIndividual>



    <!-- http://www.semanticweb.org/example/targaryen#JaehaerysII -->

    <owl:NamedIndividual rdf:about="http://www.semanticweb.org/example/targaryen#
        JaehaerysII">
        <rdf:type rdf:resource="http://www.semanticweb.org/example/targaryen#King"/>
        <belongsToHouse rdf:resource="http://www.semanticweb.org/example/targaryen#
            houseTargaryen"/>
        <hasAncestor rdf:resource="http://www.semanticweb.org/example/targaryen#AegonV"
            />
        <hasFather rdf:resource="http://www.semanticweb.org/example/targaryen#AegonV"/>
        <hasMother rdf:resource="http://www.semanticweb.org/example/targaryen#
            BethaBlackwood"/>
    </owl:NamedIndividual>



    <!-- http://www.semanticweb.org/example/targaryen#JocelynBaratheon -->

    <owl:NamedIndividual rdf:about="http://www.semanticweb.org/example/targaryen#
        JocelynBaratheon">
        <rdf:type rdf:resource="http://www.semanticweb.org/example/targaryen#Person"/>
    </owl:NamedIndividual>

```

```
363
364
365     <!-- http://www.semanticweb.org/example/targaryen#OrmundBaratheon -->
366
367     <owl:NamedIndividual rdf:about="http://www.semanticweb.org/example/targaryen#
            OrmundBaratheon">
368         <rdf:type rdf:resource="http://www.semanticweb.org/example/targaryen#Person"/>
369     </owl:NamedIndividual>
370
371
372
373     <!-- http://www.semanticweb.org/example/targaryen#RenlyBaratheon -->
374
375     <owl:NamedIndividual rdf:about="http://www.semanticweb.org/example/targaryen#
            RenlyBaratheon">
376         <rdf:type rdf:resource="http://www.semanticweb.org/example/targaryen#Person"/>
377         <belongsToHouse rdf:resource="http://www.semanticweb.org/example/targaryen#
                houseBaratheon"/>
378         <hasFather rdf:resource="http://www.semanticweb.org/example/targaryen#
                SteffonBaratheon"/>
379         <hasMother rdf:resource="http://www.semanticweb.org/example/targaryen#
                CassanaEstermont"/>
380     </owl:NamedIndividual>
381
382
383
384     <!-- http://www.semanticweb.org/example/targaryen#Rhaegal -->
385
386     <owl:NamedIndividual rdf:about="http://www.semanticweb.org/example/targaryen#
            Rhaegal"/>
387
388
389
390     <!-- http://www.semanticweb.org/example/targaryen#Rhaegar -->
391
392     <owl:NamedIndividual rdf:about="http://www.semanticweb.org/example/targaryen#
            Rhaegar">
393         <rdf:type rdf:resource="http://www.semanticweb.org/example/targaryen#Person"/>
394         <belongsToHouse rdf:resource="http://www.semanticweb.org/example/targaryen#
                houseTargaryen"/>
395         <hasFather rdf:resource="http://www.semanticweb.org/example/targaryen#AerysII"/
                >
396         <hasMother rdf:resource="http://www.semanticweb.org/example/targaryen#Rhaella"/
                >
397         <hasParent rdf:resource="http://www.semanticweb.org/example/targaryen#AerysII"/
                >
398     </owl:NamedIndividual>
399
400
401
402     <!-- http://www.semanticweb.org/example/targaryen#Rhaella -->
403
404     <owl:NamedIndividual rdf:about="http://www.semanticweb.org/example/targaryen#
            Rhaella">
405         <rdf:type rdf:resource="http://www.semanticweb.org/example/targaryen#Person"/>
406         <hasFather rdf:resource="http://www.semanticweb.org/example/targaryen#
                JaehaerysII"/>
407         <hasMother rdf:resource="http://www.semanticweb.org/example/targaryen#Shaera"/>
```

```
    </owl:NamedIndividual>


    <!-- http://www.semanticweb.org/example/targaryen#RhaelleTargaryen -->

    <owl:NamedIndividual rdf:about="http://www.semanticweb.org/example/targaryen#
        RhaelleTargaryen">
        <rdf:type rdf:resource="http://www.semanticweb.org/example/targaryen#Person"/>
        <hasFather rdf:resource="http://www.semanticweb.org/example/targaryen#AegonV"/>
        <hasMother rdf:resource="http://www.semanticweb.org/example/targaryen#
            BethaBlackwood"/>
    </owl:NamedIndividual>


    <!-- http://www.semanticweb.org/example/targaryen#Rhaenys -->

    <owl:NamedIndividual rdf:about="http://www.semanticweb.org/example/targaryen#
        Rhaenys">
        <rdf:type rdf:resource="http://www.semanticweb.org/example/targaryen#Person"/>
        <hasFather rdf:resource="http://www.semanticweb.org/example/targaryen#Rhaegar"/
            >
        <hasMother rdf:resource="http://www.semanticweb.org/example/targaryen#
            EliaMartell"/>
    </owl:NamedIndividual>


    <!-- http://www.semanticweb.org/example/targaryen#RobertBaratheon -->

    <owl:NamedIndividual rdf:about="http://www.semanticweb.org/example/targaryen#
        RobertBaratheon">
        <rdf:type rdf:resource="http://www.semanticweb.org/example/targaryen#King"/>
        <belongsToHouse rdf:resource="http://www.semanticweb.org/example/targaryen#
            houseBaratheon"/>
        <hasFather rdf:resource="http://www.semanticweb.org/example/targaryen#
            SteffonBaratheon"/>
        <hasMother rdf:resource="http://www.semanticweb.org/example/targaryen#
            CassanaEstermont"/>
    </owl:NamedIndividual>


    <!-- http://www.semanticweb.org/example/targaryen#Shaera -->

    <owl:NamedIndividual rdf:about="http://www.semanticweb.org/example/targaryen#Shaera
        ">
        <rdf:type rdf:resource="http://www.semanticweb.org/example/targaryen#Person"/>
    </owl:NamedIndividual>


    <!-- http://www.semanticweb.org/example/targaryen#StannisBaratheon -->

    <owl:NamedIndividual rdf:about="http://www.semanticweb.org/example/targaryen#
        StannisBaratheon">
        <rdf:type rdf:resource="http://www.semanticweb.org/example/targaryen#Person"/>
```

```
        <belongsToHouse rdf:resource="http://www.semanticweb.org/example/targaryen#
            houseBaratheon"/>
        <hasFather rdf:resource="http://www.semanticweb.org/example/targaryen#
            SteffonBaratheon"/>
        <hasMother rdf:resource="http://www.semanticweb.org/example/targaryen#
            CassanaEstermont"/>
    </owl:NamedIndividual>



    <!-- http://www.semanticweb.org/example/targaryen#SteffonBaratheon -->

    <owl:NamedIndividual rdf:about="http://www.semanticweb.org/example/targaryen#
        SteffonBaratheon">
        <rdf:type rdf:resource="http://www.semanticweb.org/example/targaryen#Person"/>
        <hasFather rdf:resource="http://www.semanticweb.org/example/targaryen#
            OrmundBaratheon"/>
        <hasMother rdf:resource="http://www.semanticweb.org/example/targaryen#
            RhaelleTargaryen"/>
    </owl:NamedIndividual>



    <!-- http://www.semanticweb.org/example/targaryen#Viserion -->

    <owl:NamedIndividual rdf:about="http://www.semanticweb.org/example/targaryen#
        Viserion"/>



    <!-- http://www.semanticweb.org/example/targaryen#Viserys -->

    <owl:NamedIndividual rdf:about="http://www.semanticweb.org/example/targaryen#
        Viserys">
        <rdf:type rdf:resource="http://www.semanticweb.org/example/targaryen#Person"/>
        <belongsToHouse rdf:resource="http://www.semanticweb.org/example/targaryen#
            houseTargaryen"/>
        <hasFather rdf:resource="http://www.semanticweb.org/example/targaryen#AerysII"/
            >
        <hasMother rdf:resource="http://www.semanticweb.org/example/targaryen#Rhaella"/
            >
    </owl:NamedIndividual>



    <!-- http://www.semanticweb.org/example/targaryen#houseBaratheon -->

    <owl:NamedIndividual rdf:about="http://www.semanticweb.org/example/targaryen#
        houseBaratheon">
        <rdf:type rdf:resource="http://www.semanticweb.org/example/targaryen#House"/>
        <isAllyOf rdf:resource="http://www.semanticweb.org/example/targaryen#
            houseTargaryen"/>
    </owl:NamedIndividual>



    <!-- http://www.semanticweb.org/example/targaryen#houseTargaryen -->

```

```
500     <owl:NamedIndividual rdf:about="http://www.semanticweb.org/example/targaryen#
            houseTargaryen">
501         <rdf:type rdf:resource="http://www.semanticweb.org/example/targaryen#House"/>
502     </owl:NamedIndividual>
503 </rdf:RDF>
504
505
506
507 <!-- Generated by the OWL API (version 4.5.29.2024-05-13T12:11:03Z) https://github.com
        /owlcs/owlapi -->
```