OntoExpand: SPARQL-Based Ontology Expansion and Reasoning

Vitor Lelis N. Leite
University of Minho, Braga, Portugal

José Carlos L. Ramalho
University of Minho, Braga, Portugal

Abstract

The growing complexity and volume of OWL ontologies in the Semantic Web context present challenges for reasoning and knowledge expansion. Traditional ontology reasoners often face high computational costs, limiting their scalability and real-time applicability. Additionally there are other situations where reasonners crash and are not able to expand the ontology (for example, if your pattern includes property chains involving an irreflexive relation). This paper introduces OntoExpand, a new methodology for ontology expansion that uses SPARQL CONSTRUCT queries as an efficient alternative to conventional reasoning techniques. In OntoExpand we have created a mapping between OWL axioms and CONSTRUCT queries that are able to generate the new triples. These inferred triples are then reintegrated into the ontology using SPARQL INSERT queries, enabling incremental knowledge enrichment. This approach improves performance, reduces computational overhead, is pattern driven allowing a more granular expansion control and seamlessly integrates with SPARQL endpoints. OntoExpand is evaluated on multiple ontologies with varying sizes and complexities, demonstrating its scalability and practical utility in dynamic, ontology-driven environments. The results show that OntoExpand offers a viable and efficient mechanism for reasoning and expanding OWL ontologies, making it suitable for large-scale Semantic Web applications where traditional reasoning may be infeasible.

2012 ACM Subject Classification Software and its engineering → Search-based software engineering

Keywords and phrases OWL Ontologies, SPARQL, Reasoning, Knowledge Expansion, Inference

Digital Object Identifier 10.4230/OASIcs.SLATE.2026.12

1 Context and Problem Statement

1.1 Challenges in Ontology Expansion

Ontologies play a fundamental role in knowledge representation by providing a structured, machine-processable model of concepts and their relationships within a domain [3]. Their importance is amplified in domains such as Artificial Intelligence, data integration, and especially the Semantic Web [1], where consistent interpretation of heterogeneous data is crucial.

However, as knowledge domains evolve and new data becomes available, maintaining ontologies becomes increasingly complex. A key challenge lies in the expansion of ontologies to accommodate new facts without compromising consistency. Traditionally, this task relies on external *reasoners* [2], which apply logical inference to generate new knowledge from existing axioms. Although effective, these systems introduce significant computational overhead, particularly in large-scale or real-time environments. Most reasoners process the ontology holistically, resulting in performance bottlenecks, high memory usage, and latency.

Beyond performance, current reasoners often struggle with certain constructs, such as the combination of property chains and irreflexive properties, which are essential for modeling complex relations but are not well supported in classical reasoning engines. This can limit the expressiveness and practical applicability of traditional reasoning in dynamic, real-world applications.

1.2 Towards a Query-Based Reasoning Approach

To overcome these limitations, we propose **OntoExpand**, a reasoning and ontology expansion engine built upon SPARQL CONSTRUCT queries [6]. Unlike conventional reasoners, OntoExpand bypasses the need for external reasoning tools by translating ontological axioms into SPARQL constructs. This approach enables fine-grained, targeted inference directly over RDF data, improving both scalability and responsiveness.

SPARQL, being a W3C standard, provides a flexible and widely supported mechanism for querying RDF graphs. By leveraging CONSTRUCT queries, OntoExpand can infer new triples by applying rule-like transformations that mimic classical logical reasoning. This method is particularly well-suited for applications on the Semantic Web, where data is continuously evolving and systems must respond in real-time.

Furthermore, OntoExpand addresses specific reasoning gaps by explicitly supporting features such as property chains combined with irreflexivity constraints, cases that are problematic or unsupported in most existing reasoners. In doing so, it bridges the gap between expressiveness and performance, providing a reasoning model that is both practical and standards-compliant.

2 Methodology

2.1 Axiom-Driven Inference Using SPARQL

The core idea behind our approach is to replace traditional OWL reasoners with a query-based inference engine that leverages SPARQL CONSTRUCT queries to simulate logical reasoning over ontologies. To achieve this, we analyzed the OWL 2 specification [4] to identify key reasoning operations that could be directly mapped to SPARQL patterns. These reasoning operations are traditionally grounded in Description Logics and expressed through logical rules or axiomatic patterns.

Rather than relying on black-box reasoning tools, we manually translated selected OWL axioms into SPARQL query templates capable of deriving new RDF triples from existing assertions. Each SPARQL CONSTRUCT query acts as an application rule, injecting inferred knowledge back into the graph. This design allows us to model reasoning as an explicit, transparent, and tunable query process that can be selectively applied to a dataset.

2.2 Selected Inference Patterns

While OWL reasoning encompasses a wide variety of operations, in this paper we focus on two representative patterns that demonstrate the feasibility and benefits of our approach: **Instance Hierarchy Inference** and **Property Chain Inference**. The other patterns are only described.

2.2.1 Instance Hierarchy Inference

This pattern corresponds to the transitive propagation of class membership through subclass hierarchies.

$$\texttt{rdfs:subClassOf}(A, B) \land \texttt{rdf:type}(x, A) \rightarrow \texttt{rdf:type}(x, B) \tag{1}$$

This corresponds to the rule in Equation 1, inferring indirect class types for individuals by following subclass hierarchies.

Listing 1 Instance Hierarchy query.

```
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 .
    }
}
```

Explanation.

- For each individual `indiv` typed with class `class`, the query finds all superclasses `parent` via `rdfs:subClassOf*`.
- Adds new type assertions `rdf:type(indiv, parent)` unless already present.

2.2.2 Property Chain Inference

OWL allows defining new properties as compositions of existing ones using `owl:propertyChainAxiom`.

$$R_1 \circ R_2 \to R_3 \Rightarrow (R_1(x, y) \wedge R_2(y, z) \to R_3(x, z)) \tag{2}$$

This type of inference is not always supported by conventional reasoners, especially when paired with constraints like irreflexivity. By explicitly encoding this inference pattern as SPARQL queries, our system enables the use of expressive reasoning constructs within the query model.

Property Chain Identification

In order to correspond to the condition of Equation 2, this query identifies the property chains equivalent to the original property.

Listing 2 Property Chain SELECT query.

```
SELECT ?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
```

Explanation.

- Identifies each `owl:propertyChainAxiom` and its associated super-property.
- Builds the full chain of sub-properties using path concatenation.

Property Chain Inference

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

Listing 3 Property Chain CONSTRUCT query.

```
CONSTRUCT {
    ?x <superProperty> ?z .
}
WHERE {
    ?x <property1>/<property2>/.../<propertyN> ?z .

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

Explanation.

- Infers that `?x` is related to `?z` via the `<superProperty>` if there exists a matching chain of intermediate properties.
- The path expression `<property1>/<property2>/.../<propertyN>` represents the exact order defined in the chain.
- The `FILTER NOT EXISTS` clause ensures that the inference respects `owl:IrreflexiveProperty` semantics, i.e., it prevents assertions like `?x <superProperty> ?x` when such reflexivity is disallowed.
- This approach is particularly useful for modeling indirect or derived relationships (e.g., ancestry, delegation) while maintaining logical consistency with ontology constraints.

2.3 Mapping OWL to SPARQL

The methodology to map OWL patterns/axioms into SPARQL queries consists of three main steps:

1. **Axiom Pattern Identification**: We analyze OWL 2 constructs and define corresponding inference rules in logical form.
2. **SPARQL Template Design**: Each rule is translated into a reusable SPARQL CONSTRUCT query.
3. **Query Execution and Knowledge Expansion**: These queries are executed iteratively over an RDF graph, with inferred triples added dynamically.

By grounding inference in OWL semantics and expressing reasoning steps as explicit queries, this approach maintains alignment with Semantic Web standards while offering practical advantages in terms of scalability and flexibility.

It is important to note that the `CONSTRUCT` queries are used to generate inferred triples, but they do not modify the graph directly. To persist these results in the dataset, the inferred triples must be inserted using SPARQL `INSERT` queries [5]. This decouples inference from mutation, allowing better control over which inferences are materialized and when they are committed to the knowledge base.

3 Application example

The following example illustrates the methodology applied to a simple ontology.

3.1 Ontology Overview

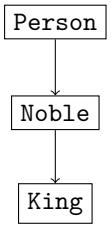
This ontology models a small fragment of the *House Targaryen* from the *Game of Thrones* universe. It was designed to demonstrate instance hierarchy and property chain inferences, two key reasoning patterns supported by OWL semantics.

3.1.1 Class Hierarchy

The ontology defines the following classes:

- **Person** – the most general class.
- **Noble** – a subclass of `Person`.
- **King** – a subclass of `Noble`.

The hierarchy can be visualized as:



3.1.2 Object Properties

The ontology defines several object properties:

- `hasParent` – links a child to a parent.
- `hasChild` – links a parent to a child.
- `hasSibling` – an `owl:IrreflexiveProperty` defined through a property chain:

$$\texttt{hasParent} \circ \texttt{hasChild} \Rightarrow \texttt{hasSibling}$$

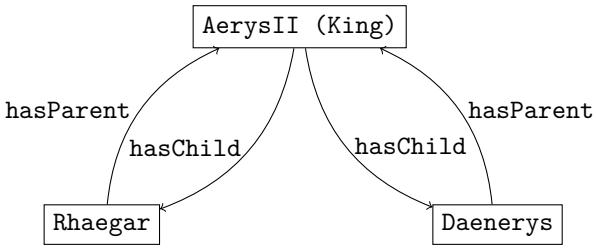
This chain means that two individuals are considered siblings if they share a parent.

3.1.3 Individuals and Relationships

Three key individuals are defined in the ontology:

- **Aerys II** is a `King`, and has two children: `Rhaegar` and `Daenerys`.
- **Rhaegar** is a `Person` and the child of `Aerys II`.
- **Daenerys** is a `Person` and also a child of `Aerys II`.

Their relationships can be represented as follows:



3.2 Inference Results

For the two selected patterns we present the new generated triples and the reasoning that led to them.

3.2.1 Instance Hierarchy Inference

In this ontology, the class hierarchy is structured as follows:

- `King` is a subclass of `Noble`
- `Noble` is a subclass of `Person`

The individual `AerysII` is explicitly typed as a `King`. Through subclass transitivity, it infers that `AerysII` is also a `Noble` and a `Person`.

These inferences are obtained using the query defined in Listing 1, which constructs new `rdf:type` statements for any superclass reachable via the subclass hierarchy.

- `AerysII rdf:type Noble`
- `AerysII rdf:type Person`

CONSTRUCT query results can then be added to the ontology by an INSERT query.

Listing 4 INSERT for inferred types.

```
INSERT DATA {
    :AerysII a :Noble .
    :AerysII a :Person .
}
```

3.2.2 Property Chain Inference

The ontology defines the following property chain axiom:

$$\texttt{hasParent} \circ \texttt{hasChild} \Rightarrow \texttt{hasSibling}$$

This allows the reasoner to infer sibling relationships between individuals who share a common parent:

- `Rhaegar hasParent AerysII`
- `Daenerys hasParent AerysII`

And `AerysII` has both `Rhaegar` and `Daenerys` as children, we can infer:

- `Rhaegar hasSibling Daenerys`
- `Daenerys hasSibling Rhaegar`

These triples are generated using the construct pattern shown in Listing 3. The query includes a safeguard against asserting reflexive triples, considering the `owl:IrreflexiveProperty` type of `hasSibling`.

Listing 5 INSERT for inferred siblings.

```
INSERT DATA {
    :Rhaegar :hasSibling :Daenerys .
    :Daenerys:hasSibling :Rhaegar .
}
```

4 Other Axioms

In this section, we present additional OWL reasoning patterns that can be translated into SPARQL-based inference rules using the same methodology described previously. For each axiom type, we provide a concise description of its semantic meaning, the logical rule that characterizes the inference, and the corresponding SPARQL query used to materialize the inferred triples.

4.1 Instance Checking

Instance checking determines whether an individual belongs to a particular class. This type of inference can be derived from *domain* property and *range* constraints.

Domain Inference

If a property has a declared domain, any individual appearing as the subject of that property can be inferred to belong to the domain class.

$$\texttt{rdfs:domain}(P, C) \land P(s, o) \rightarrow \texttt{rdf:type}(s, C) \tag{3}$$

The following query implements the rule in Equation 3.

Listing 6 Domain Inference Query.

```
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 . }
}
```

Explanation.

- The query identifies which individuals appear as the subject of a property with a declared domain.
- It infers that the subject belongs to the domain class of that property.
- The `FILTER NOT EXISTS` clause prevents generating duplicate type assertions that are already present in the ontology.

Range Inference

Similarly, if a property has a declared range, any individual appearing as the object of that property can be inferred to belong to the range class.

$$\texttt{rdfs:range}(P, C) \land P(s, o) \rightarrow \texttt{rdf:type}(o, C) \tag{4}$$

The following query corresponds to the rule in Equation 4.

Listing 7 Range Inference Query.

```
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 . }
}
```

Explanation.

- The query identifies individuals that appear as the object of a property with a declared range.
- It infers that the object belongs to the range class of that property.
- The `FILTER NOT EXISTS` clause ensures that already existing type assertions are not duplicated.

4.2 Subclass and Subproperty Reasoning

Hierarchical relations between classes and properties allow the inference of more general knowledge from existing assertions. If a class is a subclass of another class, instances of the subclass can also be considered instances of the superclass. Similarly, if a property is defined as a subproperty of another property, triples using the subproperty imply corresponding triples using the superproperty.

$$\texttt{rdfs:subClassOf}(A, B) \land \texttt{rdfs:subClassOf}(B, C) \rightarrow \texttt{rdfs:subClassOf}(A, C) \tag{5}$$

$$\texttt{rdfs:subPropertyOf}(P, Q) \land P(x, y) \rightarrow Q(x, y) \tag{6}$$

Subclass Inference

The following query simulates the rule defined in Equation 5, retrieving inferred subclass relationships based on transitive subclass paths.

Listing 8 Subclass Inference Query.

```
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 . }
}
```

Explanation.

- The query retrieves all subclass relations using the transitive path operator `rdfs:subClassOf*`.
- It identifies cases where a class is indirectly related to a superclass through intermediate classes.
- The `FILTER(?sub != ?super)` condition avoids reflexive subclass relations.
- The `FILTER NOT EXISTS` clause prevents generating subclass triples that already exist.

Subproperty Inference

The following query implements the rule in Equation 6, inferring superproperty triples based on subproperty relationships.

Listing 9 Subproperty Inference Query.

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

Explanation.

- The query identifies properties that are part of a property hierarchy using `rdfs:subPropertyOf+`.
- For each triple `(?indiv ?prop ?target)`, it infers a new triple using the corresponding superproperty.
- The `FILTER NOT EXISTS` clause ensures that duplicate triples are not produced.

4.3 Inverse Property Inference

Inverse properties enable inference of new relationships by reversing existing triples according to inverse property declarations. If a property is defined as the inverse of another, then the existence of one relationship implies the corresponding reverse relationship.

$$\texttt{owl:inverseOf}(P, Q) \Rightarrow (P(x, y) \rightarrow Q(y, x)) \tag{7}$$

Additionally, the `owl:inverseOf` relation is defined as symmetric in OWL.

$$\texttt{owl:inverseOf}(P, Q) \rightarrow \texttt{owl:inverseOf}(Q, P) \tag{8}$$

Symmetric Inverse Declaration

Since ontologies may declare only one direction of an inverse property relation, the symmetric counterpart must be explicitly materialized before applying inverse inference. The following query enforces the rule in Equation 8.

Listing 10 Inverse Property Symmetry Enforcement.

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

Explanation.

- The query identifies properties connected through an `owl:inverseOf` relation.
- If the symmetric declaration does not exist, it constructs the reverse statement.
- This ensures that both directions of the inverse relation are explicitly represented in the ontology.

Inverse Inference

Once inverse relationships are fully declared, new triples can be inferred by reversing the direction of existing property assertions. The following query applies the rule in Equation 7.

Listing 11 Inverse Property Inference Query.

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

Explanation.

- The query identifies object properties that are declared as inverses.
- For each triple `?a ?prop ?b`, it infers the corresponding inverse triple `?b ?inv ?a`.
- This inference relies on the symmetric inverse relations established by the previous query.

4.4 Transitive Property Reasoning

Transitive properties allow the inference of indirect relationships through repeated applications of the same property. If a property is declared as transitive, the existence of two consecutive relationships implies a third relationship connecting the first and last individuals.

$$P \subseteq \texttt{owl:TransitiveProperty} \Rightarrow (P(x,y) \land P(y,z) \rightarrow P(x,z)) \tag{9}$$

Transitive Property Identification

The first step consists of identifying all properties declared as transitive in the ontology.

Listing 12 Retrieve Transitive Properties.

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

Explanation.

- The query retrieves all object properties that are also declared as `owl:TransitiveProperty`.
- The resulting properties are then used to apply transitive inference over the dataset.

Transitive Inference

For each property obtained from the previous query, the following inference is applied.

Listing 13 Deep Transitive Inference.

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

Explanation.

- The query searches for paths of one or more consecutive links using the transitive property.
- The `+` path operator enables traversal of chains of arbitrary length.
- For each discovered path between `?x` and `?z`, a direct triple is inferred.
- The placeholder `<prop>` is dynamically replaced with the property URI retrieved from Listing 12.

4.5 Symmetric Property Reasoning

Symmetric properties allow the inference of mirrored relationships between individuals. If a property is declared as symmetric, whenever a relationship exists in one direction, the corresponding reverse relationship can also be inferred.

$$P \subseteq \texttt{owl:SymmetricProperty} \Rightarrow (P(x,y) \rightarrow P(y,x)) \tag{10}$$

Symmetric Property Inference

The following query implements the rule in Equation 10.

Listing 14 Symmetric Property Inference.

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

Explanation.

- The query identifies all object properties declared as `owl:SymmetricProperty`.
- For each triple `(?target ?prop ?subj)`, it infers the corresponding symmetric triple `(?subj ?prop ?target)`.
- The `FILTER NOT EXISTS` clause prevents generating duplicate triples that may already exist in the ontology.

4.6 Restrictions

OWL class restrictions allow the definition of classes based on constraints over properties. If an individual satisfies the conditions expressed by a restriction that is part of a class definition, it can be inferred to belong to that class. These restrictions are commonly expressed using constructs such as `owl:someValuesFrom`, `owl:allValuesFrom`, or `owl:hasValue`.

$$\texttt{rdf:type}(x, \exists P.C) \rightarrow \exists y.\ P(x, y) \wedge \texttt{rdf:type}(y, C) \tag{11}$$

Class Inference from Property Restrictions

The following query implements the rule in Equation 11.

Listing 15 Class Inference from Restrictions.

```
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 .
  }
}
```

Explanation.

- The query identifies classes defined through `owl:equivalentClass` combined with `owl:intersectionOf`.
- Each element of the intersection is inspected to detect property restrictions.
- If an individual satisfies the required property constraint, the query infers that the individual belongs to the corresponding class.
- Restrictions using `owl:someValuesFrom`, `owl:allValuesFrom`, or `owl:hasValue` are supported.
- The query also allows inference when the property range implies the restricted class.

4.7 Equivalence & Subsumption Reasoning

Equivalence reasoning identifies when two classes can be considered logically equivalent based on the restrictions that define them. If two classes share the same property constraints, individuals satisfying those constraints can belong to either class, implying that the classes are semantically equivalent.

$$\forall x.\ (\texttt{rdf:type}(x, \exists P.A) \leftrightarrow \texttt{rdf:type}(x, \exists P.B)) \Rightarrow \texttt{owl:equivalentClass}(A, B) \qquad (12)$$

Equivalence Class Detection

To correspond to Equation 12, this approach compares the property restrictions used in the definitions of different classes.

Listing 16 Equivalence Class First Query.

```
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 17 Equivalence Class Second 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 18 Inference of Equivalent Classes.

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

  {
    # Second Query
  }

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

Explanation.

- The first two queries extract the property restrictions that define each class.
- These restrictions are normalized into a signature string using `GROUP_CONCAT`.
- If two classes produce identical restriction signatures, they are inferred to be equivalent.
- The final query constructs an `owl:equivalentClass` relation between classes with matching definitions.

5 Conclusion

This paper presented a practical approach to simulating OWL reasoning operations using SPARQL queries, with a focus on instance hierarchy inference and property chain inference. By aligning OWL axioms with SPARQL `CONSTRUCT` queries, we demonstrated how implicit semantic relationships can be made explicit and persisted through `INSERT` operations.

Using a simplified ontology, we show how hierarchical class relations and property chain axioms can lead to meaningful inferences – such as superclass types and inferred sibling relationships – without relying on traditional OWL reasoners. The results highlight how SPARQL can be used to implement reasoning patterns in a transparent and explainable manner.

Although only two reasoning operations were presented in this paper, a broader set of OWL inference patterns were specified and described. Current work is focused on automating the generation of SPARQL queries for these operations with the goal of expanding coverage and improving scalability in more complex ontologies. This line of work contributes toward bridging the gap between declarative ontology semantics and practical query-driven inference over RDF data. We are also creating a web workbench where it is possible to select which patterns to apply and decide to expand or not the initial ontology.

References

1 Komal Dhulekar and Madhuri Devrankar. A review on semantic web. *International Journal of Engineering Technologies and Management Research*, 6:22–28, 2020. doi:10.29121/ijetmr.v6.i12.2019.470.

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

3 Ontotext. What are ontologies?, 2025. URL: https://www.ontotext.com/knowledgehub/fundamentals/what-are-ontologies/.
4 W3C. OWL web ontology language reference. URL: https://www.w3.org/TR/owl-ref/.
5 W3C. SPARQL 1.1 update. URL: https://www.w3.org/TR/sparql11-update/.
6 W3C. SPARQL 1.1 query language, 2013. URL: https://www.w3.org/TR/sparql11-query/.