
An Alignment-Driven Approach for Constructing SPARQL Federated Queries

Moussa Abdenour Beldi^{1,*}, Sihem Klai²,
Karima Mecheri³ and Seif Eddine Bouziane⁴

¹*Department of Electronics, LABGED, Faculty of Technology, Badji Mokhtar-Annaba University, P.O. Box 12, Annaba, 23000, Algeria*

²*Department of Computer Science, LABGED, Badji Mokhtar-Annaba University, Annaba, Algeria*

³*Department of Computer Science, LISCO, Badji Mokhtar-Annaba University, Annaba, Algeria*

⁴*The National School of Artificial Intelligence, Algiers, Algeria*

E-mail: moussa-abdenour.beldi@univ-annaba.dz; sihem.klai@univ-annaba.dz; karima.mecheri@univ-annaba.dz; seifeddine.bouziane@ensia.edu.dz

**Corresponding Author*

Received 31 March 2026; Accepted 30 June 2026

Abstract

Integrating complementary information across distributed SPARQL endpoints remains a significant challenge. While query federation enables access across multiple endpoints, constructing federated queries requires both SPARQL expertise and knowledge of the underlying ontological schemas. Current query rewriting techniques primarily focus on vocabulary translation to retrieve equivalent data, often overlooking the integration of complementary information spread across sources. To address these limitations, we propose an alignment-driven approach that assists users in constructing federated queries by leveraging ontology alignments. Our approach

joins class paths – user-selected sequences of ontology classes and relationships – from source and target ontologies, guided by alignment-based filtering, into a unified path that serves as the foundation for generating federated SPARQL queries capturing complementary aspects of the user’s intent. An experimental evaluation using OAEI Conference track datasets demonstrates that the generated queries consistently match reference queries in both structure and data retrieval across simple and complex mapping types, confirming the approach’s effectiveness in integrating complementary data from heterogeneous sources.

Keywords: Query federation, Query generation, Ontology alignment, SPARQL.

1 Introduction

The Linked Open Data (LOD) initiative has driven a significant expansion of structured, interconnected data published on the Web, facilitating integration and interoperability across diverse domains and applications. However, the growing volume of distributed data sources introduces challenges in retrieving and combining complementary information from different endpoints. In many real-world scenarios, answering a single information need requires combining data scattered across multiple independent sources, each capturing different aspects of a shared domain. The SPARQL 1.1 specification addresses this through the notion of *federated queries* via the `SERVICE` keyword.¹ A federated SPARQL query retrieves and combines data from multiple distributed SPARQL endpoints in a single execution context. Rather than manually querying each endpoint and merging the results, a federated query specifies which portions should be evaluated at which remote endpoints and how the results should be joined based on shared identifiers or semantically aligned concepts. While federation enables the integration of complementary information from separate sources into a complete answer, constructing effective federated queries remains a challenging task. It requires proficiency in SPARQL and its federated capabilities, prior knowledge of the ontology structures involved, and a clear understanding of how concepts across different sources relate to one another. Most existing approaches address single-source querying with limited consideration for

¹www.w3.org/TR/sparql11-federated-query

federation, and none adequately support the construction of federated queries for retrieving complementary data from multiple sources. To overcome this limitation, we propose a novel approach that assists users in constructing SPARQL federated queries for retrieving complementary information from distributed ontologies. Our approach leverages ontology alignment between a source and a target ontology, supporting both simple and complex mappings. A federated query is generated by constructing and joining class paths from the source and target ontologies, using the alignment to form a unified class path that spans both ontologies. A class path is a user-defined sequence of ontology classes and their relationships, constructed by selecting source and target classes from the respective ontologies. The unified class path is then translated into a SPARQL federated query that retrieves and integrates data from both endpoints. Unlike query rewriting approaches, which translate a query from one vocabulary to another to retrieve equivalent data from an alternative source, our approach uses alignments as bridge conditions to combine complementary data distributed across heterogeneous endpoints.

The remainder of this paper is organized as follows: Section 2 presents an overview of related works. Section 3 provides the necessary background information. Section 4 describes the proposed approach and the techniques used for generating federated queries. Section 5 discusses the implementation and evaluation, and Section 6 concludes with a summary and directions for future work.

2 Related Work

Several studies have proposed different methods to facilitate SPARQL query construction. Template-based methods such as [1] bind input keywords to query templates computed from the graph schema in an incremental step process to generate SPARQL queries. Similarly, [2] maps resources to predefined graph pattern templates to generate the corresponding queries. [3] uses natural language techniques to map user keywords to ontology resources, and generates SPARQL queries by applying a minimum spanning tree algorithm in order to connect retrieved resources. [4] employs semantic similarity measures along with Word2Vec to map query terms to domain ontologies, utilizing the A^* shortest-path algorithm to construct the final SPARQL query structure. [5] extracts keywords from natural language questions and matches them to ontology classes to construct SPARQL queries. OWL-Path [6] provides a natural language interface that translates user questions into SPARQL queries. Furthermore, deep learning architectures, such as

Neural Machine Translation (NMT) models, have been employed to directly translate natural language into simplified SPARQL sequences [7]. Other approaches rely on visual query builders. [8] guides users in interactively building queries through faceted search, verbalizing them in natural language while covering most of SPARQL's expressivity. In [9], annotations obtained from the SPARQL endpoint are used to dynamically construct a query based on the relationships between user-specified input and output classes. [10] provides a graphical interface that allows users to add constraints and link classes from a predefined class hierarchy. To the best of our knowledge, limited work addresses automatic federated SPARQL query generation. [11] generates conjunctive federated SPARQL queries from natural language questions, mapping keywords to ontology resources via statistical information and connecting equivalents through `owl:sameAs` links. However, NLP-based approaches may misinterpret ambiguous queries. In the context of benchmarking, [12] applies heuristics and statistical information to predict federated queries, while [13] derives them by analyzing and combining real query logs from different datasets. Although both operate without user intervention, they rely on statistical patterns or existing logs that may not be available for new deployments. [14] derives federated queries by matching classes based on pairwise Jaccard similarity applied to their respective types, but does not leverage formal ontology alignments.

Other works focus on query rewriting and decomposition for query federation, each addressing a different level of schema mediation. [15] introduced a query plan that utilizes vocabulary mappings to translate queries from a global vocabulary to a local vocabulary, operating at the terminological level. [16] proposed a rewriting method that exploits complex correspondences between two ontologies, handling 1:n mappings at the schema level. [17] decomposes a global query into sub-queries using mappings between a domain ontology and local dataset vocabularies, targeting query decomposition for distributed execution. Despite these differences, all three approaches assume that a query already exists and transform it for execution across different sources, whereas our approach focuses on query *construction*. Recently, large language models (LLMs) have demonstrated promising capabilities in translating natural language questions into SPARQL queries [18–20]. However, in federated settings, LLMs are prone to hallucinating ontology terms and lack awareness of the schemas exposed by each endpoint, making it difficult to correctly distribute query fragments using the `SERVICE` keyword. Moreover, ensuring semantic correctness requires formal alignment with the target ontologies, which LLMs cannot guarantee.

Table 1 Comparative analysis of SPARQL query generation approaches

Approach	Input	SERVICE	Align.	Compl.	Task
Sparklis [8]	GUI	×	×	×	Construction
Yamaguchi et al. [9]	GUI	×	×	×	Construction
OWLPath [6]	NL	×	×	×	Construction
Zenz et al. [1]	KW	×	×	×	Construction
SPARQLing [10]	GUI	×	×	×	Construction
SINA [11]	NL	×	×	×	Construction
Yadav & Duhan [4]	NL	×	×	×	Construction
Lin & Lu [7]	NL	×	×	×	Generation
Moreau et al. [14]	Auto	×	×	×	Generation
Cunha & Lóscio [17]	×	×	✓	✓	Decomposition
SPLODGE [12]	Auto	✓	×	×	Generation
Cheng et al. [15]	×	×	✓	×	Rewriting
Thiéblin et al. [16]	×	×	✓	×	Rewriting
PFed [13]	Auto	✓	×	✓	Recommendation
Our approach	GUI	✓	✓	✓	Construction

KW = Keywords, NL = Natural Language, GUI = Graphical Interface, Auto = Automatic. SERVICE = SPARQL SERVICE keyword support. Align. = Alignment-driven. Compl. = Complementary data retrieval. ✓ = supported, × = not supported.

Table 1 provides a comparative analysis of representative SPARQL query generation approaches. We compare approaches along five dimensions: input method (how users interact with the system), federation support (whether the approach generates queries with the SERVICE keyword), alignment awareness (whether ontology alignments are exploited), complementary data retrieval (whether federated sub-queries retrieve complementary information from different sources rather than decomposing or rewriting a single query), and the nature of the task (query construction, rewriting, or decomposition).

As shown in Table 1, most existing approaches lack SERVICE-based federation support [1, 6, 8–11, 14]. Among those that do, SPLODGE [12] automatically generates federated queries but without exploiting ontology alignments or retrieving complementary data, while PFed [13] recommends federated queries that retrieve complementary data from multiple endpoints but does not leverage ontology alignments. [17] also retrieves complementary data through query decomposition, but their approach assumes that a global query already exists and relies on custom schema mappings between a

domain ontology and local vocabularies. Alignment-aware approaches [15, 16] focus on query rewriting rather than construction. Our approach is the only one that combines all four capabilities: user-guided query construction, SERVICE-based federation, alignment awareness, and complementary data retrieval. This unique combination addresses the gap identified in the preceding analysis, enabling non-expert users to construct federated queries without requiring SPARQL expertise or manual inspection of ontology schemas.

3 Background

This section introduces the key concepts underlying our approach, including RDF graphs, class graph schemas, class paths, ontology alignment, and SPARQL query construction.

3.1 Knowledge representation

RDF (Resource Description Framework) expresses data as a set of *triples*, each of the form (s, p, o) , where s denotes the *subject*, p the *predicate*, and o the *object*. The subject identifies the resource being described, the predicate specifies a property or relation, and the object provides either another resource or a literal value. An RDF graph can be viewed as a directed labeled graph where subjects and objects are nodes and predicates are edge labels.

Formally, let I , B , and L be pairwise disjoint sets of IRIs (Internationalized Resource Identifiers), blank nodes, and literals, respectively. Literals represent data values such as strings, numbers, or dates.

Definition 1 (RDF Triple) An RDF triple is a tuple $(s, p, o) \in (I \cup B) \times I \times (I \cup B \cup L)$.

Definition 2 (RDF Graph) An RDF graph G is a finite set of RDF triples. Each triple $(s, p, o) \in G$ can be interpreted as a directed edge $s \xrightarrow{p} o$.

Definition 3 (RDF Dataset) An RDF dataset D is a collection of RDF graphs, where each graph contains a finite set of RDF triples. Formally, a dataset can be expressed as:

$$D = \{G_1, G_2, \dots, G_n\},$$

where each G_i is an RDF graph.

Definition 4 (Triple Pattern) Given a set of variables V disjoint from I , B , and L , a triple pattern is a tuple $(s, p, o) \in (I \cup B \cup V) \times (I \cup V) \times (I \cup B \cup L \cup V)$.

Our approach is designed to be domain-agnostic, enabling it to utilize a class graph schema associated with any ontology. In the context of OWL and RDFS, a *class* represents a category or type of resources. Classes are identified by IRIs and can be organized in hierarchies using `rdfs:subClassOf` relationships. A class graph schema represents the classes, properties, and relationships used to model the entities and their attributes. The following definition is built based on [1]:

Definition 5 (Class Graph Schema) Let R be an RDF graph, and let C be the set of all classes in R . A class graph schema $G_{R-c} = \langle V, E, u, r \rangle$ of R is a directed labeled multigraph defined as follows:

- V is a set of nodes with size $|C|$, where u is a function that maps each node $n \in V$ to a class IRI in C .
- E is a set of directed edges between the nodes of V , where r is a function that maps each edge $e \in E$ to a predicate IRI in R .
- a directed edge e from node n_1 to n_2 where $r(e) = p$, $u(n_1) = class_s$ and $u(n_2) = class_o$, is included in E if and only if p satisfies the following conditions:
 1. There exist triples “ p rdfs:domain $class_s$ ” and “ p rdfs:range $class_o$ ”, where both classes $class_s$ and $class_o$ are in C .
 2. There exists at least one triple “ $s p o$ ” in R , such that “ s rdf:type $class_s$ ” and “ o rdf:type $class_o$ ”.

Additionally, following the methodology introduced in [9], we introduce the concept of class path, which is a sequence of classes and properties that connect two or more classes in a graph schema. A class path can be represented as a set of triples, with each triple consisting of a subject, predicate, and object. In this context, the subject and object refer to the class names, while the predicate represents the relationship or connection between them.

Definition 6 (Class Path) Let G_{R-c} be a class graph schema. A class path C_p from a source class c_s to a target class c_t is a sequence $(n_1, e_1, n_2, e_2, \dots, n_m)$, where nodes n_i and edges e_i of G_{R-c} satisfy the

following conditions:

1. $u(n_1) = c_s$ and $u(n_m) = c_t$.
2. $u(n_i) \neq c_t$ for $1 \leq i < m$.
3. Edge e_i is a directed edge from n_i to n_{i+1} or from n_{i+1} to n_i .
4. If $u(n_i) = u(n_k)$, then $e_{i-1} \neq e_k$, $\forall i > k$.

An edge e_i that is directed from n_i to n_{i+1} is termed forward-directed, whereas an edge from n_{i+1} to n_i is termed reverse-directed.

To illustrate, consider our running example where we want to retrieve reviews of rejected papers. The class path in the Ekaw ontology would be: $\text{Rejected_Paper} \xrightarrow{\text{hasReview}} \text{Review}$. Here, Rejected_Paper is the source class (c_s), Review is the target class (c_t), and hasReview is the forward-directed edge connecting them. Condition 2 ensures we do not pass through the target class before reaching it. Condition 4 prevents cycles by ensuring that if we revisit a class, we must use a different edge than before.

3.2 Ontology Alignment

An *ontology* formally specifies a domain through classes, properties, and axioms. However, independent modeling of overlapping domains results in semantic heterogeneity. *Ontology alignment* addresses this challenge by establishing correspondences between entities of different ontologies. Given a source ontology O_s and a target ontology O_t , an alignment maps entities from O_s to semantically related entities in O_t . Such mappings range from simple one-to-one correspondences to complex expressions involving class constructors or restrictions. Beyond pairwise alignment, network-based approaches have also been proposed [28–30] to interconnect multiple ontologies.

Definition 7 (Alignment) *Given a source ontology O_s and a target ontology O_t , and a set of relations $R = \{\equiv, \sqsubseteq, \sqsupseteq\}$ representing equivalence, subsumption, and inverse subsumption respectively, an alignment $A_{s \rightarrow t}$ is a set of mappings $M = \{m_1, m_2, \dots, m_n\}$. Each mapping m_i is represented as a triple $\{e_s, r, e_t\}$ where:*

- e_s and e_t are entities (classes, properties, or instances) from the source and target ontologies respectively.
- $r \in R$ is a mapping relation between entities e_s and e_t .
- m_i is classified as a simple mapping if it represents a one-to-one (1:1) correspondence between e_s and e_t , such as class-to-class mapping.

- m_i is classified as a complex mapping if it represents a one-to-many (1:n) or many-to-many (n:n) correspondence, where an entity is mapped to a set of entities (such as the intersection or union of classes), or to an entity with logical constraints (such as property-based restrictions).

Alignments can be generated automatically, semi-automatically, or manually by domain experts. The generation of the alignment is out of scope for this paper. For further reading, we recommend [22–24]. An alignment can be expressed in various formats, including description logic, SWRL [27], and EDOAL.² In this work, the alignment dataset is expressed using EDOAL [25, 26] for its expressiveness. Figure 1 illustrates an example correspondence expressed in EDOAL.

```
<map>
  <Cell>
    <entity1>
      <edoal:Class rdf:about="http://conference#Topic"/>
    </entity1>
    <entity2>
      <edoal:Class rdf:about="http://edas#Topic"/>
    </entity2>
    <relation>=</relation>
    <measure rdf:datatype="xsd:float">1.0</measure>
  </Cell>
</map>
```

Figure 1 An example of alignment correspondence expressed in EDOAL.

Our approach restricts its focus to equivalence relations ($\equiv$) at the **class level**, as these enable direct entity substitution across ontologies during federated query construction. According to Gunaratna et al. [21], inter-ontology alignments can comprise property-level, class-level, and instance-level correspondences. We consider only class-level mappings, since our approach operates on class paths. Predicate mappings are excluded, as they do not necessarily imply correspondence between the associated domain and range classes. Instance-level mappings are also excluded, as our query construction relies on schema-level semantics rather than individual instances, distinguishing our work from entity linking (e.g., `owl:sameAs`). Consequently, we do not produce alignments, but leverage existing schema-level mappings as foundational input.

²<https://moex.gitlabpages.inria.fr/alignapi/edoal.html>

3.3 SPARQL Query Construction

SPARQL is the standard query language and protocol designed for querying and manipulating RDF data. A SPARQL query consists of a WHERE clause that defines a set of Basic Graph Patterns (BGPs). Each BGP consists of triple patterns with three elements: subject, predicate, and object. The SELECT clause specifies the variables that will be included in the query results. During query evaluation, the BGPs in the WHERE clause are matched against the RDF dataset graph to find matches and obtain a set of bindings for the variables identified in the SELECT clause. Note that a SPARQL query can be represented as a sub-graph within the data space.

In order to construct a SPARQL query, we translate a class path that aligns with the user's intention into a SPARQL query graph by capturing the relationships between a starting node and an ending node. Hence, we can obtain candidate SPARQL queries that match the user's requirements by identifying all possible class paths.

Definition 8 (Class Path Query) Assume a class path $(n_1, e_1, n_2, e_2, \dots, n_m)$ indicates a relationship between a start class and an end class. The corresponding SPARQL query is constructed as follows:

```
SELECT * WHERE {
  ?r1 p(e1) ?r2
  ?r1 rdf:type u(n1)
  ?r2 rdf:type u(n2)
  ...
  ?ri p(ei) ?ri+1
  ?ri rdf:type u(ni)
  ?ri+1 rdf:type u(ni+1)
}
```

Each variable $?r_i$ corresponds to a class node n_i in the path, and the `rdf:type` triple patterns restrict variables to instances of their respective classes. The property triple patterns connect consecutive variables using the edge predicates $p(e_i)$. If an edge e_i is reverse-directed, the subject and object positions are swapped in the corresponding triple pattern.

3.4 SPARQL Query Federation

SPARQL query federation involves processing a query over multiple distributed data sources via SPARQL remote endpoints. The SPARQL SERVICE keyword introduced in SPARQL 1.1 allows the specification of a SPARQL remote endpoint where the query will be evaluated by a SPARQL

engine. SPARQL endpoints are services that provide an interface for querying remote RDF data following the SPARQL protocol. An endpoint is defined by using the syntax *SERVICE* $\langle target \rangle$, where $\langle target \rangle$ is the URL that points to the SPARQL endpoint.

4 Query Federation Generation

In this section, we provide a high-level overview of our proposed system. Figure 2 illustrates a brief preview of our system architecture. The proposed system consists of two main processes: the Alignment Model Generation process and the query generation process.

An *alignment model* $M_{A_{s-t}}$ is a data structure that encodes the correspondences from an alignment A_{s-t} , enabling programmatic lookup of mappings between source and target entities. Given a source ontology, a target ontology, and an alignment dataset, the system parses the correspondences and generates the alignment model.

For each correspondence in the alignment dataset, the entities and their defining relations are extracted and parsed into structured entries within the alignment model. Through iterative processing, the model is progressively enriched, capturing the mappings and relationships between entities from the source and target ontologies. The query generation process follows a sequential workflow: first, the user selects a source class from a list of candidates retrieved from the model. The class path generator then queries

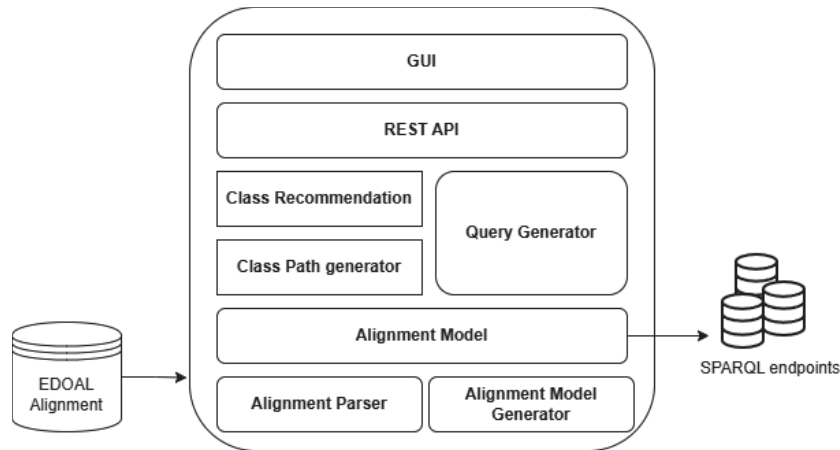


Figure 2 System architecture overview.

the corresponding ontology to extract all incoming and outgoing class paths – up to a predefined maximum depth – that originate from or terminate at the selected class. These resulting paths are then presented to the user, who selects the class path that best aligns with the intended information requirements for retrieval from the source ontology.

Algorithm 1 Class recommendation algorithm

Require: CP {Source Class Path}

Require: $M_{A_{s-t}}$ {Alignment Model}

```

1:  $targetClasses \leftarrow$  empty list
2: for each class  $c$  in  $CP$  do
3:   for  $c_t$  in  $QUERYMATCHINGCLASSES(c, M_{A_{s-t}})$  do
4:      $targetClasses.append(c_t)$ 
5:   end for
6: end for
7: return  $targetClasses$ 

```

As described in Algorithm 1, the class recommendation component iterates over each class in the selected source class path and invokes the function $QUERYMATCHINGCLASSES(c, M_{A_{s-t}})$, which queries the alignment model $M_{A_{s-t}}$ to retrieve all target classes mapped to a given source class c within the alignment dataset. Based on the collected results, the user is presented with a list of candidate classes from the target ontology. Following the user's selection, the class path generator extracts all possible class paths from the target ontology up to a predefined maximum depth. Once the target class path is selected, the query generator translates both paths into respective source and target queries. These are integrated via the mapping between source and target classes to construct a federated query capable of retrieving data from both ontologies.

4.1 Alignment Processing

In the context of query federation, knowledge of relevant data sources that can contribute to answering a query is essential. Ontology alignment plays a key role in this regard, as it identifies relationships between entities across different ontologies, thereby determining which data sources contain the required information. The alignment parser component processes the alignment dataset between the source and target ontologies, extracting each entity along with the corresponding relationships defined in the mapping. The alignment utilized in this work comprises both simple and

complex mappings, including 1:1 and 1:n relationships. The following subsections formally define each mapping type supported by the alignment parser.

4.1.1 Simple class mapping

Let C_s be a class from the source ontology O_s , and C_t be a class from the target ontology O_t . A simple class mapping occurs when a class from the source ontology is mapped to a class from the target ontology, denoted as:

$$C_s \equiv C_t$$

4.1.2 Class union mapping

Let C_s be a class from the source ontology O_s , and let C_t and C'_t be classes from the target ontology O_t . A class union mapping occurs when a class in the source ontology is mapped to a union of classes in the target ontology, denoted as:

$$C_s \equiv C_t \cup C'_t$$

4.1.3 Class intersection mapping

Let C_s be a class from ontology O_s , and let C_t and C'_t be classes from ontology O_t . A class intersection mapping occurs when a class in the source ontology is mapped to an intersection of multiple classes in the target ontology, denoted as:

$$C_s \equiv C_t \cap C'_t$$

4.1.4 Domain restriction mapping

Let C_s be a class from ontology O_s , C_t be a class from ontology O_t , R be a relation (object property), and $C_{t_{\text{range}}}$ be a class used to restrict the range of R . A Domain Restriction mapping occurs when a class in the source ontology is mapped to a class C_t in the target ontology, with the constraint that there exists at least one relation R whose range is restricted to instances of $C_{t_{\text{range}}}$, denoted as:

$$C_s \equiv C_t \cap \exists R.C_{t_{\text{range}}}$$

4.1.5 Value restriction mapping

Let C_s be a class from ontology O_s , C_t be a class from ontology O_t , P be a relation or property, and v be a value. A Value Restriction mapping occurs when a class in the source ontology is mapped to a class C_t in the target ontology, with the constraint that there exists a relation or property P that

corresponds to the value v , denoted as:

$$C_s \equiv C_t \cap \exists P : P = v$$

4.1.6 Property occurrence mapping

Let C_s be a class from ontology O_s , C_t be a class from ontology O_t , and P be a property defined in O_t . A Property Occurrence mapping occurs when a class C_s in the source ontology is mapped to a class C_t in the target ontology, with the constraint that only those instances of C_t for which property P has at least one value are considered, denoted as:

$$C_s \equiv C_t \cap \exists P. \top$$

The following examples (1)–(3) illustrate various mapping scenarios between entities from different ontologies using a set of datasets from the OAEI Conference track. These examples correspond to our running example of querying rejected papers and their co-authors. Example (1) presents a simple class mapping where the class `cmt:Paper` from the source ontology `Cmt` is mapped to the class `confOf:Contribution` in the target ontology `ConfOf`. Example (2) demonstrates a Domain Restriction Mapping, where the class `ekaw:Rejected_Paper` of the `Ekaw` ontology is mapped to the class `cmt:Paper` in the `Cmt` ontology if there exists an object property `cmt:hasDecision` with its range restricted to instances of the class `cmt:Rejection`. Example (3) introduces a value restriction mapping between the `Ekaw` and `ConfOf` ontologies, where the class `ekaw:Early-Registered_Participant` is mapped to the class `confOf:Participant`, with the restriction that the datatype property `confOf:earlyRegistration` equals `true`.

$$cmt:Paper \equiv confOf:Contribution. \quad (1)$$

$$ekaw:Rejected_Paper \equiv cmt:Paper \sqcap \exists hasDecision.Rejection \quad (2)$$

$$ekaw:Early-Registered_Participant \equiv confOf:Participant \sqcap (earlyRegistration = \text{true}) \quad (3)$$

4.2 Alignment Model Generation

The alignment model serves as a unified knowledge structure that integrates the source and target ontologies according to a predefined alignment

specification. The input alignment, expressed in EDOAL (Section 3.2), is transformed into a structured representation that enables efficient retrieval of mappings between entities. During the generation phase, distinct namespaces are assigned to each ontology to ensure unambiguous differentiation between source and target components.

Formally, an alignment model $M_{A_{s-t}}$ can be defined as a tuple (C_{so}, C_{to}, P_t, L) where:

- C_{so} : a set of classes from the source ontology.
- C_{to} : a set of classes from the target ontology.
- P_t : a set of object properties from the target ontology.
- L : a set of literals, if any are found.

Since only 1:n mapping is considered in our case, C_{so} represents a simple class expression from the source ontology. In contrast, C_{to} , P_t and L may represent various types of class expressions, ranging from simple to complex, depending on the provided alignment. Each entity in the mapping is parsed and stored as a structured entry in the alignment model $M_{A_{s-t}}$, preserving the classes, object properties, and literals involved in the correspondence. Formally, the generation process can be described as a function

$$\phi : A_{s-t} \longrightarrow M_{A_{s-t}}$$

where A_{s-t} denotes the alignment dataset between the source ontology O_s and the target ontology O_t , and $M_{A_{s-t}}$ is the resulting alignment model. Each correspondence in A_{s-t} , as defined in previous section, is parsed and stored as a structured entry within $M_{A_{s-t}}$.

4.3 Class Path Generation

Given an ontology O and a class C , the Class Path Generator component is responsible for generating all class paths associated with C , including both outgoing paths that start from C and incoming paths that end at C . To enable efficient property retrieval, two indexes are constructed for each loaded ontology. The *domain index* maps each class to the properties for which it serves as the domain along with their corresponding range classes, and is used for outgoing path generation. In contrast, the *range index* maps each class to the properties for which it serves as the range along with their corresponding domain classes, and is used for incoming path generation. Both indexes resolve all inherited property associations, including those derived from direct and inferred subclass relationships. Object properties lacking

predefined domains or ranges are excluded from both indexes. Algorithm 2 provides a pseudocode outline of the outgoing path generation method, where *domainIndex* represents the pre-computed domain index and *C* denotes the selected class from the given ontology.

Algorithm 2 Outgoing class paths generation

```

1: Input: Domain index domainIndex, Class C, Max depth maxDepth
2: Output: paths
3:
4: Function SearchOutgoingPaths(C)
5:   paths  $\leftarrow \emptyset$ 
6:   for each (p, range) in domainIndex[C] do
7:     path  $\leftarrow [C, p, range]$ 
8:     GeneratePaths(paths, path, 1)
9:   end for
10: Return paths
11:
12: Function GeneratePaths(paths, path, depth)
13:   paths  $\leftarrow paths \cup \{path\}$ 
14:   if depth  $\geq maxDepth$  then
15:     Return
16:   end if
17:   el  $\leftarrow lastElement(path)$ 
18:   for each (p, range) in domainIndex[el] do
19:     if range  $\notin path$  then
20:       newPath  $\leftarrow path + [p, range]$ 
21:       GeneratePaths(paths, newPath, depth + 1)
22:     end if
23:   end for

```

The algorithm begins by querying the domain index for the starting class *C* to retrieve all object properties for which *C* or any of its super classes serves as the domain. For each retrieved property, a level-1 class path is constructed, denoted as $CP_1 = (c_1, p_1, c_2)$, where the classes c_1 and c_2 are linked via the object property p_1 . Following the identification of level-1 class paths, the algorithm proceeds recursively to construct deeper paths. For each identified class path, the domain index is queried for the last class in the current path to retrieve its associated properties, and a new extended path is constructed for each valid property. For instance, a level-2 class path is denoted as $CP_2 = (c_1, p_1, c_2, p_2, c_3)$. Each path is added to the result set at every depth level, ensuring that paths of all lengths are collected. To guarantee termination, two conditions are enforced: a class that

already appears in the current path is not revisited, which prevents cycles in the ontology graph, and recursion stops once the predefined maximum depth $maxDepth$ is reached to avoid exploring excessively long paths. In contrast, when generating incoming paths, the range index is used instead of the domain index.

4.4 SPARQL Query Generation

Given a class path $CP = (c_1, p_1, c_2, p_2, \dots, c_n)$, the SPARQL Query Generator component is responsible for translating it into a SPARQL query by iterating over each element and generating the corresponding triple patterns.

Algorithm 3 Query generation algorithm

```

1: Input:  $CP$ 
2: Output:  $query$ 
3:  $query \leftarrow \text{empty}$ 
4:  $v \leftarrow \text{empty}$ 
5: for  $el$  in  $CP$  do
6:   if  $el$  is a class then
7:      $v \leftarrow var(el)$ 
8:      $query \leftarrow query + "v \text{ rdf:type } u(el) "$ 
9:   else if  $el$  is an objectProperty then
10:     $v' \leftarrow var(el)$ 
11:     $query \leftarrow query + "v \text{ r}(el) v' "$ 
12:     $v \leftarrow v'$ 
13:   else if  $el$  is a dataTypeProperty then
14:     $v'' \leftarrow var(el)$ 
15:     $query \leftarrow query + "v \text{ r}(el) v'' "$ 
16:     $query \leftarrow query + "FILTER(v'' = ValueOf(el)) "$ 
17:   end if
18: end for
19: return  $query$ 

```

Algorithm 3 provides a pseudocode outline of the query generation method, where CP denotes the input class path. The algorithm relies on three helper functions: $var(el)$ generates a unique SPARQL variable for an element, $u(el)$ returns the IRI of a class, and $r(el)$ returns the IRI of a property. The algorithm iteratively extracts each element in the class path CP . If the element is a class, a new variable v is generated, and an RDF type triple pattern " $var(el) \text{ rdf:type } u(c_1)$ " is added to the query. If the element is

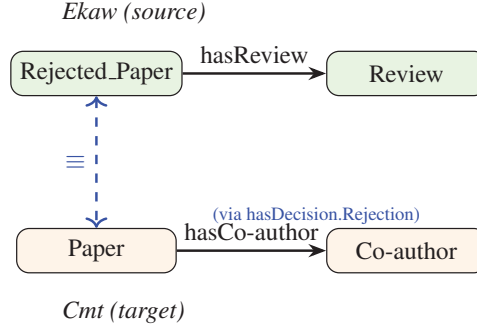


Figure 3 Running example: class paths from Ekaw and Cmt ontologies joined via alignment.

an object property, the algorithm assigns a new variable v' to the subsequent class in the path and adds a triple pattern “ $v \ r(el) \ v'$ ” that links v to v' through the object property, then updates $v \leftarrow v'$ for the next iteration. In the case of a datatype property, a new variable v'' is generated and added to the triple pattern “ $v \ r(el) \ v''$ ”, with the variable’s value constrained using a Filter operator “ $FILTER(v'' = ValueOf(el))$ ”. The resulting SPARQL query will be a SELECT query that projects all variables from the triple patterns specified in the WHERE clause.

4.5 Federated query construction

To generate a federated query, our approach leverages the alignment mapping between a source class and a target class in order to join the corresponding class paths. The resulting combined path is then translated into a federated query. Figure 3 illustrates this process using our running example.

For example, let c_{s1} be a source class with a selected class path $CP_s = (c_{s1}, p_{s1}, c_{s2})$ from the source ontology, and let c_{t1} be a target class with a selected class path $CP_t = (c_{t1}, p_{t1}, c_{t2})$ from the target ontology. Depending on the mapping type between c_{s1} and c_{t1} , different forms of federated queries are generated.

4.5.1 Simple class mapping

Let c_{s1} be a class from the source class path mapped to a class c_{t1} from the target class path, where $c_{s1} \in CP_s$ and $c_{t1} \in CP_t$. A federated query is constructed by rewriting the variable binding $var(c_{t1})$ in the target query Q_t as $var(c_{s1})$, and by encapsulating both the source and target queries within SPARQL SERVICE closures.

```

SELECT * WHERE {
  SERVICE <SOURCE_ENDPOINT> {
    var(cs1) rdf:type u(cs1).
    var(cs2) rdf:type u(cs2).
    var(cs1) r(ps1) var(cs2).
  }
  SERVICE <TARGET_ENDPOINT> {
    var(cs1) rdf:type u(ct1).
    var(ct2) rdf:type u(ct2).
    var(cs1) r(pt1) var(ct2).
  }
}

```

4.5.2 Class union mapping

Let c_{s1} be a class from the source class path mapped to a class union of c_{t1} and c'_t where $c_{s1} \in CP_s$, $c_{t1} \in CP_t$ and $c'_t \in O_t$. A federated query is constructed by rewriting variable binding $var(c_{t1})$ in the target query Q_t as $var(c_{s1})$ and by adding the triple pattern “ $var(c_{s1})$ $rdf:type$ $u(c'_t)$ ” within the UNION pattern operator.

```

SELECT * WHERE {
  SERVICE <SOURCE_ENDPOINT> {
    var(cs1) rdf:type u(cs1).
    var(cs2) rdf:type u(cs2).
    var(cs1) r(ps1) var(cs2).
  }
  SERVICE <TARGET_ENDPOINT> {
    {
      var(cs1) rdf:type u(ct1).
    }
    UNION
    {
      var(cs1) rdf:type u(c'_t).
    }
    var(ct2) rdf:type u(ct2).
    var(cs1) r(pt1) var(ct2).
  }
}

```

4.5.3 Class intersection mapping

Let c_{s1} be a class from the source class path mapped to a class intersection of c_{t1} and c'_t where $c_{s1} \in CP_s$, $c_{t1} \in CP_t$ and $c'_t \in O_t$. A federated query is constructed by rewriting variable binding $var(c_{t1})$ in the target query Q_t as $var(c_{s1})$ and by adding the triple “ $var(c_{s1})$ $rdf:type$ $u(c'_t)$ ”.

```

SELECT * WHERE {
  SERVICE <SOURCE_ENDPOINT> {
    var(cs1) rdf:type u(cs1).
  }
}

```

```

        var(cs2) rdf:type u(cs2).
        var(cs1) r(ps1) var(cs2).
    }
    SERVICE <TARGET_ENDPOINT> {
        var(cs1) rdf:type u(ct1).
        var(cs1) rdf:type u(c't).
        var(ct2) rdf:type u(ct2).
        var(cs1) r(pt1) var(ct2).
    }
}

```

4.5.4 Domain restriction mapping

Let c_{s1} be a class from the source class path mapped to a domain restriction expression of the class c_{t1} where $c_{s1} \in CP_s$, $c_{t1} \in CP_t$ and $c'_t \in O_t$, if there exists at least a relation $p_{t'}$ restricted on its range to class type O'_t . A federated query is constructed by rewriting variable binding $var(c_{t1})$ in the target query Q_t to $var(c_{s1})$ and by adding the triples “ $var(c_{s1})$ $r(p_{t'})$ $var(c'_t)$ ” and “ $var(c'_t)$ rdf:type $u(c'_t)$ ”.

```

SELECT * WHERE {
    SERVICE <SOURCE_ENDPOINT> {
        var(cs1) rdf:type u(cs1).
        var(cs2) rdf:type u(cs2).
        var(cs1) r(ps1) var(cs2).
    }
    SERVICE <TARGET_ENDPOINT> {
        var(cs1) rdf:type u(ct1).
        var(cs1) r(pt') var(c't).
        var(c't) rdf:type u(c't).
        var(cs1) r(pt1) var(ct2).
        var(ct2) rdf:type u(ct2).
    }
}

```

4.5.5 Value Restriction Mapping

Let c_{s1} be a class from the source class path mapped to a value restriction expression of the class c_{t1} where $c_{s1} \in CP_s$, $c_{t1} \in CP_t$ and $c'_t \in O_t$ if there exists a property p'_t corresponding to an individual value. A federated query is constructed by rewriting variable binding $var(c_{t1})$ in the target query Q_t to $var(c_{s1})$ and by adding the triple “ $var(c_{s1})$ $p_{t'}$ $?v$ ” and a restriction “ $\text{FILTER}(?v = \text{value})$ ”.

```

SELECT * WHERE {
    SERVICE <SOURCE_ENDPOINT> {
        var(cs1) rdf:type u(cs1).
        var(cs2) rdf:type u(cs2).
        var(cs1) r(ps1) var(cs2).
    }
}

```

```

    }
    SERVICE <TARGET_ENDPOINT> {
        var(cs1) rdf:type u(ct1).
        var(cs1) r(pt') ?v.
        var(cs1) r(pt1) var(ct2).
        var(ct2) rdf:type u(ct2).
        FILTER(?v = value)
    }
}

```

4.5.6 Property occurrence mapping

Let c_{s1} be a class from the source class path mapped to a property occurrence expression of the class c_{t1} where $c_{s1} \in CP_s$, $c_{t1} \in CP_t$, if there exists a property $p_{t'}$ in O_t such that only instances for which $p_{t'}$ has at least one value are considered. A federated query is constructed by rewriting variable binding $var(c_{t1})$ in the target query Q_t to $var(c_{s1})$ and by adding the triple “ $var(c_{s1})$ $r(p_{t'})$?attr”.

```

SELECT * WHERE {
    SERVICE <SOURCE_ENDPOINT> {
        var(cs1) rdf:type u(cs1).
        var(cs2) rdf:type u(cs2).
        var(cs1) r(ps1) var(cs2).
    }
    SERVICE <TARGET_ENDPOINT> {
        var(cs1) rdf:type u(ct1).
        var(cs1) r(pt') ?attr.
        var(cs1) r(pt1) var(ct2).
        var(ct2) rdf:type u(ct2).
        FILTER(BOUND(?attr))
    }
}

```

5 Implementation and Evaluation

5.1 Implementation

Our system was developed as a RESTful application, with Flask serving as the backend framework in conjunction with the RDFlib library for processing RDF datasets. The user interface was built using the Angular framework as the frontend component. The application is deployed on a cloud-hosted container instance with 1 shared vCPU and 1 GB of RAM, and is publicly available at <https://fedpath.beldi.dev>. The live demo allows users to reproduce the experiments presented in this paper, including ontology selection, class path exploration, and federated query generation.

5.2 Evaluation Setup

Our evaluation aims to determine whether the proposed approach can generate federated queries that are correct in terms of semantic structure and answer retrieval compared to reference queries. To the best of our knowledge, there is currently no publicly available dataset that includes a pair of ontologies with semantically rich alignment. To address this gap, we built our evaluation around a subset of conference organization ontologies provided by the OAEI Conference track,³ focusing on the domain of conference organization. For our evaluation, we utilized several ontologies, including Ekaw, Cmt, Conference, Edas and ConfOf. These ontologies were independently developed within the OntoFarm dataset and describe the conference organization domain. As a result, they provide overlapping concepts with heterogeneous conceptualizations, offering a controlled setting that reflects common ontology integration issues while preserving reproducibility through the use of reference alignments. Table 2 summarizes their structural characteristics. An alignment dataset between each pair of ontologies was provided, satisfying the essential requirements for our experiment.

We evaluated our approach on five test cases designed to systematically cover most mapping types supported by our approach. Each test case requires combining complementary data from a source and a target ontology via a federated query. For each case, a reference query was manually constructed by a SPARQL expert to serve as a baseline for comparison. To support the experimental setup, each ontology was populated with synthetic instances satisfying the query requirements and loaded into a local RDF store, exposing a SPARQL endpoint for data access. As a complementary user-oriented assessment, six participants (three computer science students and three from non-technical backgrounds), none of whom had prior knowledge of SPARQL or the underlying ontologies, were asked to construct the test queries through

Table 2 Structural statistics of the ontologies used in the evaluation

Ontology	Classes	Predicates	Triples
Ekaw	86	16	557
ConfOf	49	15	539
Cmt	32	15	412
Conference	81	16	639
Edas	119	20	1720

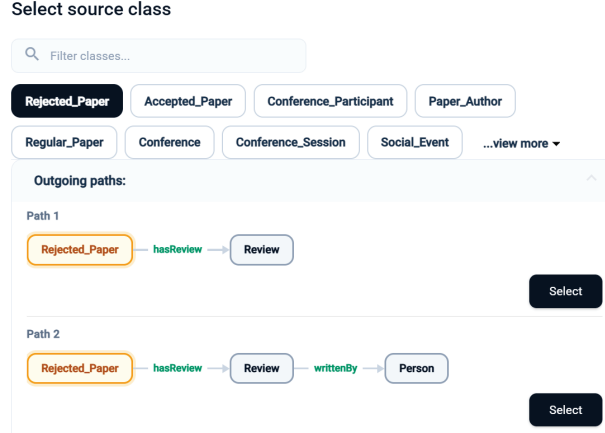
³<http://oaei.ontologymatching.org/2022/>

the graphical user interface. All participants successfully constructed correct federated queries for the five test cases, providing preliminary evidence that the proposed approach can help abstract the complexity of federated query formulation. We detail the following test cases below:

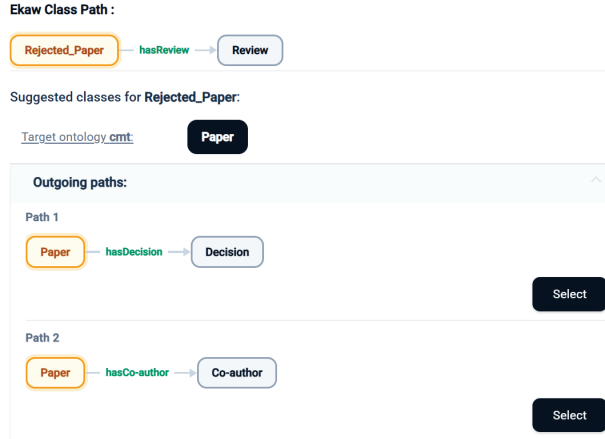
- q1)** Retrieve the decisions of papers along with their authors (Cmt to ConfOf).
- q2)** Retrieve reviewers along with the organizations they are employed by (Conference to ConfOf).
- q3)** Retrieve documents authored by an early-registered participant, along with the participant's country (Ekaw to ConfOf).
- q4)** Retrieve the programmes of conference events along with their locations (Edas to Ekaw).
- q5)** Retrieve the reviews of rejected papers and the co-authors of those papers (Ekaw to Cmt).

These queries reflect a progression from simple to complex mapping scenarios: class equivalence mapping (*q1*), class union mapping (*q2*), value restriction mapping (*q3*), property occurrence mapping (*q4*), and domain restriction mapping (*q5*). Class intersection mappings are supported by the proposed approach. However, no separate test case is provided for this mapping type, as no suitable class intersection correspondence was found in the reference alignments of the OAEI Conference track used in our evaluation.

As illustrated in Figure 4, the process involves two main steps: (a) selecting a source class path and (b) selecting a corresponding target class path. Consider query *q5*, which aims to retrieve reviews of rejected papers and their associated co-authors. The user is first presented with a list of classes from the source ontology Ekaw and selects the most relevant class. In this case, the class *Rejected_Paper* is selected as the source class. The system then retrieves the incoming and outgoing class paths associated with the selected class. As shown in Figure 4, class path 1, *Rejected_Paper* $\rightarrow$ *hasReview* $\rightarrow$ *Review*, addresses the first part of query *q5*: retrieving the reviews of rejected papers from Ekaw. Following the selection of this path, candidate target classes are generated by extracting the classes within the source path and retrieving their corresponding classes from the alignment model. As defined in Mapping Example 2 (Section 4), the class *Rejected_Paper* is mapped to class *Paper* in the target ontology (Cmt) via a domain restriction. Based on this mapping, *Paper* is retrieved and suggested as the target class. Following a similar procedure, after the selection of *Paper* as the target class, a list of class paths associated with the target class is retrieved from the target ontology.



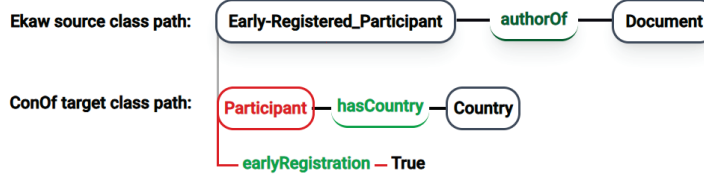
(a) Step 1: Source class path selection.



(b) Step 2: Target class path selection.

Figure 4 Overview of the user interface during the query construction process.

After reviewing the available target class paths, the class path 2, *Paper* → *hasCo-Author* → *Co-Author* addresses the second part of query *q5*: retrieving papers along with their co-authors from Cmt. The user then selects this path as the target class path. Finally, a SPARQL federated query addressing *q5* is constructed by transforming each selected class path into SPARQL triple patterns. These patterns are then combined using the mapping between *Rejected_Paper* and *Paper*, and encapsulated within a *SERVICE* clause to enable federation across the two endpoints.

Figure 5 Class path composition for $q3$.**Table 3** Evaluation results. R_{gen} and R_{ref} denote the result set sizes of the generated and reference queries, respectively

Query	Ontologies	Mapping Type	R_{gen}	R_{ref}	F1
$q1$	Cmt $\rightarrow$ ConfOf	Simple equivalence	8	8	1.0
$q2$	Conference $\rightarrow$ ConfOf	Class Union	5	5	1.0
$q3$	Ekaw $\rightarrow$ ConfOf	Value restriction	4	4	1.0
$q4$	Edas $\rightarrow$ Ekaw	Property Occurrence	3	3	1.0
$q5$	Ekaw $\rightarrow$ Cmt	Domain restriction	3	3	1.0
Average			–	–	1.0

For queries such as $q3$, which retrieves documents authored by early-registered participants along with the participants' countries, the same procedure is applied using the mapping defined in Example 3 (Section 4). In this mapping, the *Early-Registered_Participant* class from the source ontology Ekaw is mapped to class *Participant* in the target ontology ConfOf via a value restriction mapping, with the property *earlyRegistration* set to *true*. The source class path *Early-Registered_Participant* $\rightarrow$ *authorOf* $\rightarrow$ *Document* and the target class path *Participant* $\rightarrow$ *hasCountry* $\rightarrow$ *Country* are selected for this query. These class paths are shown in Figure 5.

5.3 Correctness Evaluation

To measure the correctness of the retrieved results, precision and recall are computed by comparing the result set of each generated query against that of the corresponding reference query. Table 3 summarizes these metrics. Across all five test cases, executing both the generated and reference queries against the populated endpoints yielded identical result sets, achieving an *F1* score of 1.0 in every case. The generation process is fully deterministic, producing an identical query for any given class path and alignment mapping, and correctness therefore depends entirely on the selected class path and the underlying alignment.

SPARQL Federated Query q_5

```

PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX ekaw: <http://ekaw#>
PREFIX cmt: <http://cmt#>
SELECT * WHERE {
  SERVICE <http://ekaw/sparql_endpoint> {
    ?b rdf:type ekaw:Rejected_Paper.
    ?b ekaw:hasReview ?c.
    ?c rdf:type ekaw:Review.
  }
  OPTIONAL {
    SERVICE <http://cmt/sparql_endpoint> {
      ?b cmt:hasDecision ?d.
      ?d rdf:type ?e.
      ?b cmt:hasCo-author ?f.
      ?f rdf:type cmt:Co-author.
      FILTER(?e = cmt:Rejection)
    }
  }
}

```

Figure 6 Generated federated query for q_5 .

Notably, these results confirm correctness across all five mapping types, from simple class equivalence (q_1) to structurally complex domain restrictions (q_5). Each requires a fundamentally different transformation strategy when generating SPARQL triple patterns, demonstrating that the class-path-based formulation generalizes across the full range of alignment types represented in the OAEI Conference track, rather than being tailored to any single mapping pattern.

To illustrate the structural differences across mapping types, the generated federated queries for q_5 and q_3 are presented in Figures 6 and 7. In both queries, the target SERVICE clause is enclosed within an OPTIONAL block, implementing left-join semantics that preserve source-side results while including target-side bindings only when matched through the shared join variable.

5.4 Performance and Scalability

To improve the efficiency of the path generation component, a materialized indexing strategy was adopted that pre-computes all class-property associations during a preprocessing phase. In the baseline implementation, a SPARQL query is issued for each class at query time, requiring repeated graph traversals proportional to the number of classes visited during path

SPARQL Federated Query q_3

```

PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX ekaw: <http://ekaw#>
PREFIX confOf: <http://confOf#>
SELECT * WHERE {
  SERVICE <ekaw_endpoint> {
    ?b rdf:type ekaw:Early-Registered_Participant .
    ?c rdf:type ekaw:Document .
    ?b ekaw:authorOf ?c .
  }
  OPTIONAL {
    SERVICE <confOf_endpoint> {
      ?b confOf:hasCountry ?e .
      ?e rdf:type confOf:Country .
      ?b confOf:earlyRegistration ?d .
      FILTER(?d = true) .
    }
  }
}

```

Figure 7 Generated federated query for q_3 .

Table 4 Materialized index metrics for each ontology

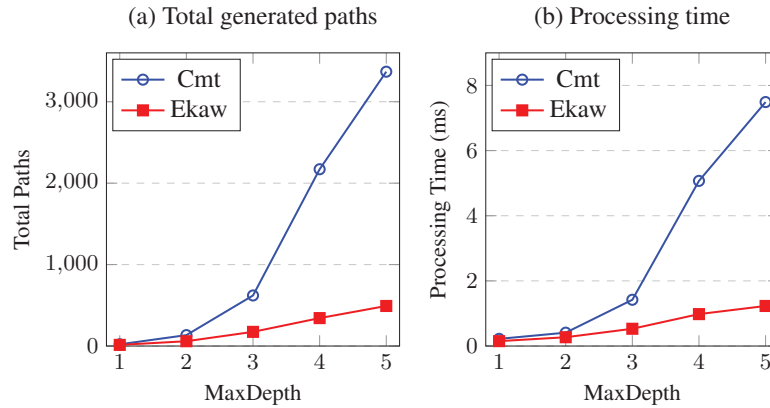
Ontology	Domain Index Entries	Range Index Entries	Memory (KB)	Build Time (ms)
Cmt	107	97	44.11	135.52
ConfOf	77	38	27.95	154.74
Conference	137	134	62.18	134.86
Edas	295	298	132.26	187.30
Ekaw	397	389	160.47	495.11
Total	1013	956	426.97	1107.53

exploration. The materialized index eliminates this overhead by resolving all class-property associations, including inherited properties propagated through the transitive closure of the subclass hierarchy, into hash-map structures indexed by domain and range, enabling constant-time lookups at query time. Table 4 reports the memory footprint and construction time of the materialized index for each ontology.

As summarized in Table 4, total index construction for all five ontologies requires approximately 1.1s with a combined memory footprint of 427 KB. The domain and range entries represent the total materialized class-property associations; for example, Ekaw yields 397 domain and 389 range entries, reflecting the high density of property associations propagated through its

Table 5 Processing time and total generated paths for class `Paper` in the Cmt and Ekaw ontologies

MaxDepth	Cmt (32 classes)		Ekaw (86 classes)	
	Total Paths	Time (ms)	Total Paths	Time (ms)
1	21	0.22	14	0.15
2	134	0.41	59	0.27
3	621	1.42	174	0.53
4	2171	5.07	343	0.98
5	3369	7.49	492	1.23

**Figure 8** Comparison of total generated paths and processing time for class `Paper` in the Cmt and Ekaw ontologies across increasing MaxDepth values.

hierarchy. Notably, Ekaw exhibits the longest build time (495.11 ms) despite containing only 557 triples. This overhead is a consequence of schema complexity, specifically the depth of the class hierarchy and the density of OWL axioms, rather than graph size. In contrast, Edas, which contains roughly three times as many triples (1,720), exhibits a significantly lower build time (187.30 ms) due to its comparatively flatter hierarchy. These results confirm that index construction costs are driven primarily by structural complexity rather than raw RDF graph size.

Beyond the indexing strategy, the processing time and scalability of the path generation component were evaluated by comparing the class `Paper` across Cmt (32 classes) and Ekaw (86 classes). Table 5 and Figure 8 summarize these results. The comparison indicates that scalability is primarily driven by *structural connectivity* rather than ontology size. Despite being

significantly smaller, Cmt generates 3,369 paths at depth 5 compared to only 492 in Ekaw, as Paper in Cmt functions as a highly connected hub class with numerous incoming and outgoing properties, which triggers approximately exponential path growth at each depth increment. In contrast, Paper in Ekaw has fewer direct relationships, resulting in near-linear growth across all depths. Since the materialized index replaces per-request SPARQL evaluation with in-memory dictionary lookups, the cost of each path extension step is reduced from a network round-trip to a constant-time hash-map access. For densely connected classes such as Paper in Cmt, the system generates over 3,000 paths at depth 5 in only 7.49 milliseconds, while at depths 2 and 3 response times remain below 1.5 milliseconds. These findings confirm that the proposed approach maintains interactive performance across all practical configurations.

6 Conclusion and Future Work

In this paper, we presented an approach for generating SPARQL federated queries by exploiting ontology alignments to bridge heterogeneous endpoints. Our approach enables users to construct federated queries by exploring class paths that express relationships between classes across aligned ontologies, allowing the retrieval of complementary information from multiple sources. By incorporating alignment mappings, class paths from the source and target ontologies are joined and translated into a federated query. We evaluated our approach using conference ontologies from the OAEI Conference track, across a range of simple and complex mapping types. The results show that the generated federated queries can reproduce the manually constructed reference queries and retrieve complementary data from heterogeneous sources. In future work, we plan to extend our framework to support networks of aligned ontologies. Furthermore, we intend to investigate the integration of LLMs to assist users in translating natural language intent into structured class paths. In this hybrid architecture, the LLM would serve as a natural language interface, while our framework ensures schema-level correctness and robust query federation.

References

- [1] Zenz G, Zhou X, Minack E, Siberski W, Nejdl W. From keywords to semantic queries – Incremental query construction on the Semantic Web. 2009 Jul 18. <http://dx.doi.org/10.2139/ssrn.3199425>.

- [2] Shekarpour S, Auer S, Ngomo A-CN, Gerber D, Hellmann S, Stadler C. Keyword-driven SPARQL query generation leveraging background knowledge. In: Proc IEEE/WIC/ACM Int Conf Web Intell Intell Agent Technol, Lyon, France, 2011, pp. 203–210. doi: 10.1109/WI-IAT.2011.70.
- [3] Zhou Q, Wang C, Xiong M, Wang H, Yu Y. SPARK: Adapting keyword query to semantic search. In: The Semantic Web: ISWC ASWC 2007, vol. 4825 of Lecture Notes in Computer Science. Berlin, Germany: Springer; 2007.
- [4] Yadav U, Duhan N. Efficient Retrieval of Data Using Semantic Search Engine Based on NLP and RDF. *Journal of Web Engineering*. 2021;20(8):2285–2318. doi: 10.13052/jwe1540-9589.2084.
- [5] Peres L, Coelho da Silva L, Macedo T, Araujo D. Ontology-schema based query by example. In: Laender A, Pernici B, Lim EP, de Oliveira J, editors. Conceptual Modeling: ER 2019, vol. 11788 of Lecture Notes in Computer Science. Cham, Switzerland: Springer; 2019.
- [6] Valencia-García R, Garcia-Sanchez F, Nieves D, Fernandez-Breis J. OWLPath: An OWL ontology-guided query editor. *IEEE Trans Syst Man Cybern Part A: Syst Humans* 2011; 41: 121–136. doi: 10.1109/TSMCA.2010.2048029.
- [7] Lin JH, Lu EJL. SPARQL Generation with an NMT-based Approach. *Journal of Web Engineering*. 2022;21(5):1471–1490. doi: 10.13052/jwe1540-9589.2155.
- [8] Hyvönen E, Ferré S. Sparklis: An expressive query builder for SPARQL endpoints with guidance in natural language. *Semant Web* 2017; 8: 405–418. <https://doi.org/10.3233/SW-150208>.
- [9] Yamaguchi A, Kozaki K, Lenz K, Wu H, Kobayashi N. An intelligent SPARQL query builder for exploration of various life-science databases. In: Proc 3rd Int Conf Intell Exploration Semantic Data (IESD'14), Aachen, Germany, 2014, pp. 83–94.
- [10] Dastgheib S, McSkimming DI, Natarajan K, Kochut KJ. SPARQLing: A graphical interface for SPARQL. In: ISWC (Posters & Demos), CEUR Workshop Proceedings, vol. 1486, 2015.
- [11] Shekarpour S, Marx E, Ngomo A-CN, Auer S. SINA: Semantic interpretation of user queries for question answering on interlinked data. 2015 Jan. <https://ssrn.com/abstract=3199174> or <http://dx.doi.org/10.2139/ssrn.3199174>.
- [12] Görlitz O, Thimm M, Staab S. SPODGE: Systematic generation of SPARQL benchmark queries for Linked Open Data. In: The Semantic

- Web – ISWC 2012, vol. 7649 of Lecture Notes in Computer Science. Berlin, Germany: Springer; 2012.
- [13] Hacques F, Skaf-Molli H, Molli P, Hassad SEL. PFed: Recommending plausible federated SPARQL queries. In: Hartmann S, Küng J, Chakravarthy S, Anderst-Kotsis G, Tjoa A, Khalil I, editors. Database and Expert Systems Applications: DEXA 2019, vol. 11707 of Lecture Notes in Computer Science. Cham, Switzerland: Springer; 2019.
 - [14] Moreau B, Kieffer M, Serrano-Alvarado P. Deducing Federated SPARQL queries from RDF mappings. In: 20th International Semantic Web Conference (ISWC), Oct 2021, Online, United States. hal-03407978.
 - [15] Cheng S, Ferrada S, Hartig O. Considering vocabulary mappings in query plans for federations of RDF data sources. In: Proc. 2023; 1–10. https://doi.org/10.1007/978-3-031-46846-9_2.
 - [16] Thiéblin É, Amarger F, Haemmerlé O, Hernandez N, Trojahn dos Santos C. Rewriting SELECT SPARQL queries from 1:n complex correspondences. In: OM@ISWC, CEUR Workshop Proceedings, vol. 1766, 2016, pp. 49–60.
 - [17] Cunha, D. R. B., and Lóscio, B. F. (2016). An Approach for Query Decomposition on Federated SPARQL Query Systems. *Journal of Information and Data Management*, 6(2), 106. <https://doi.org/10.5753/jidm.2015.1562>.
 - [18] Meyer LP, Frey J, Brei F, Arndt N. Assessing SPARQL Capabilities of Large Language Models. In: Proceedings of the NLP4KGc Workshop at SEMANTICS 2024. CEUR Workshop Proceedings, Vol. 3874; 2024. pp. 35–53.
 - [19] Frey J, Meyer LP, Brei F, Gründer-Fahrer S, Martin M. Assessing the Evolution of LLM Capabilities for Knowledge Graph Engineering in 2023. In: The Semantic Web: ESWC 2024 Satellite Events. Lecture Notes in Computer Science, vol. 15344. Cham: Springer; 2025. pp. 51–60.
 - [20] Giuliani A, Manca MM, Piano L, et al. Are LLMs Adequate SPARQL Query Generators? Investigating Zero-Shot NL-to-SPARQL Translation. *Neural Comput Appl*. 2026; 38: 33. doi: 10.1007/s00521-025-11799-x.
 - [21] Gunaratna K, Lalithsena S, Sheth A. Alignment and dataset identification of linked data in Semantic Web. *Wiley Interdiscip Rev Data Min Knowl Discov* 2014; 4: Art. no. 10.1002/widm.1121.

- [22] Qin H, Dou D, LePendu P. Discovering executable semantic mappings between ontologies. In: Meersman R, Tari Z, editors. *On the Move to Meaningful Internet Systems 2007: CoopIS, DOA, ODBASE, GADA, and IS*, vol. 4803 of *Lecture Notes in Computer Science*. Berlin, Germany: Springer; 2007, pp. 708–719. https://doi.org/10.1007/978-3-540-76848-7_56.
- [23] Parundekar R, Knoblock CA, Ambite JL. Discovering concept coverings in ontologies of linked data sources. In: Cudré-Mauroux P, et al., editors. *The Semantic Web – ISWC 2012*, vol. 7649 of *Lecture Notes in Computer Science*. Berlin, Germany: Springer; 2012. https://doi.org/10.1007/978-3-642-35176-1_27.
- [24] Portisch J, Hladik M, Paulheim H. Background knowledge in ontology matching: A survey. *Semantic Web*. 2024;15(6):2639–2693. doi: 10.3233/SW-223085.
- [25] Euzenat J, Scharffe F, Zimmermann A. Expressive alignment language and implementation. Tech Rep, INRIA; 2007. <http://hal.inria.fr/hal-00822892/>.
- [26] David J, Euzenat J, Scharffe F, Trojahn C. The Alignment API 4.0. *Semantic Web*. 2011; 2(1): 3–10.
- [27] Horrocks I, et al. SWRL: A Semantic Web Rule Language Combining OWL and RuleML. W3C Member Submission; 2004. <http://www.w3.org/Submission/SWRL/>.
- [28] Klai S, Zimmermann A, Khadir MT. Multi-Level Networked Knowledge Base: DDL Reasoning. In: Bellatreche L, Pastor O, Almendros Jiménez J, Aït-Ameur Y, editors. *Model and Data Engineering (MEDI 2016)*. Cham, Switzerland: Springer; 2016. pp. 118–131.
- [29] Klai S, Zimmermann A, Khadir MT. Multi-Level Networked Knowledge: Representation and DL Reasoning. *Int J Metadata Semantics Ontol*. 2016; 11(1): 1–15. doi: 10.1504/IJMSO.2016.078101.
- [30] Klai S, Zimmermann A, Khadir MT. Networked Ontologies with Contextual Alignments. *Comput. Informatics*. 2019; 38: 115–150. <https://api.semanticscholar.org/CorpusID:146806892>.

Biographies



Moussa Abdenour Beldi is a PhD candidate at Badji Mokhtar Annaba University and Senior Software Engineer. He received his Master's degree in Computer Science from the University Badji Mokhtar Annaba. His research interests include Semantic Web technologies and artificial intelligence.



Sihem Klai is an Associate Professor at Computer Science Department and member of the LABGED laboratory at the University of Badji Mokhtar Annaba, Algeria. Her research interests include Semantic Web, knowledge representation and reasoning with multi-contextual information on the Web.



Karima Mecheri is an Associate Professor at the Department of Computer Science, University of Badji Mokhtar Annaba, and a member of the LISCO laboratory. Her research interests include information systems cooperation, service-oriented architectures, semantic Web services, and deep learning-based recommendation.



Seif Eddine Bouziane is a faculty member at the National School of Artificial Intelligence in Algeria, a position he has held since 2022. He received his PhD in Information Science and Technology from the University of Annaba in 2022. His research interests encompass machine learning and deep learning, with a particular focus on computer vision and time series forecasting.