OntoPyLPG

A python-based interface to persist and reason OWL data in Neo4j

Yukta Rajapur IMT2021066
Ketaki Tamhanakar IMT2021017
Neha Tamhanakar IMT2021025

Project Overview

The OWL (Web Ontology Language) is widely used to represent ontologies in a machine-interpretable format and serves as a foundation of the Semantic Web, where knowledge is typically encoded in RDF (Resource Description Framework) triples. While tools like Protégé simplify ontology creation and reasoning, storing and querying OWL data at scale poses challenges, particularly due to limitations in traditional RDF triple stores when handling complex reasoning and flexible graph queries.

Given that OWL statements inherently express graph-like structures, we explored the idea of persisting ontological knowledge in a property graph database—specifically Neo4j. Neo4j is well-known for its flexible labeled property graph (LPG) model, rich query language (Cypher), and support for analytical queries and pattern matching at scale.

By translating OWL constructs into Neo4j's property graph model, we aim to bridge the gap between semantic reasoning and graph analytics—enabling scalable, expressive, and intuitive querying of ontological data. This integration supports advanced use cases such as pattern detection, semantic inference, and knowledge-driven graph analysis.

Related Work

Existing tools like Neosemantics (n10s), a Java-based Neo4j plugin, enable importing RDF vocabularies and performing basic inferencing within Neo4j. n10s focuses on loading RDF triples directly and supports limited OWL reasoning capabilities.

In contrast, our project takes a Python-centric approach by leveraging libraries such as Owlready2 for ontology parsing and reasoning. We designed a framework that not only parses OWL files but also performs logical inference and persists both asserted and inferred knowledge into Neo4j as a property graph. Users can then query the enriched graph using Cypher. This Python-native methodology enhances integration with modern machine learning and data science ecosystems and offers greater flexibility for extending reasoning mechanisms or customizing the OWL-to-LPG mappings.

Project Goals

1. Parse OWL (.owl) ontology files and map them into a labeled property graph (LPG) structure stored in Neo4j.

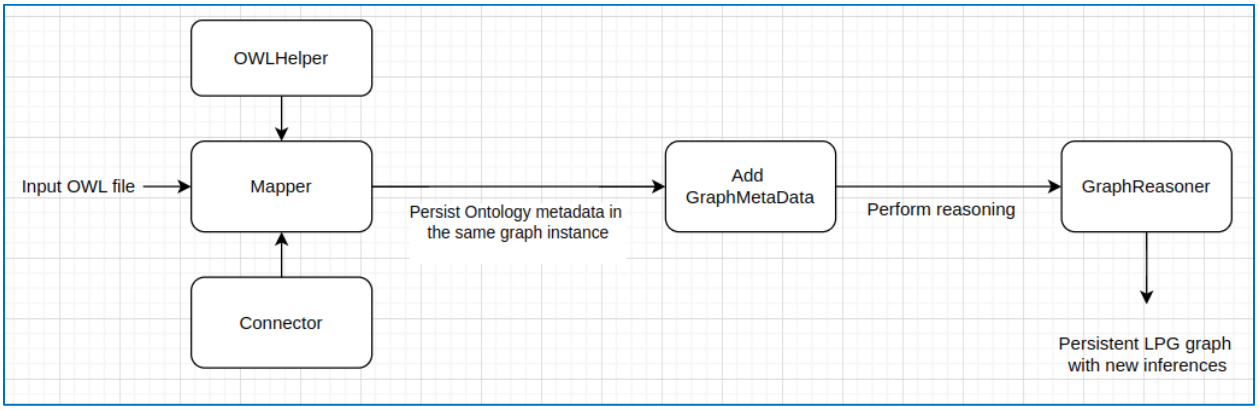
2. Build a custom Python-based reasoner that operates directly over the Neo4j LPG graph, without relying on external OWL reasoners.
3. Perform inference on the graph itself by identifying new relationships, class hierarchies, or properties based on predefined logical rules.
4. Explicitly enrich the Neo4j graph with newly inferred facts to improve semantic completeness and queryability.
5. Allow users to upload a .owl file through a simple frontend interface and trigger the graph generation and reasoning process.
6. Enable users to query the enriched Neo4j graph using Cypher after reasoning has been applied.
7. Focus on running all reasoning and processing using Neo4j as the sole database (no external RDF triple stores or external reasoners).
8. Demonstrate the feasibility and advantages of direct LPG-based reasoning for ontology data in a lightweight Python-based framework. It is designed to be lightweight, modular, and extensible for future expansions such as supporting more complex inference rules or integration with other graph analytics.

Functionality

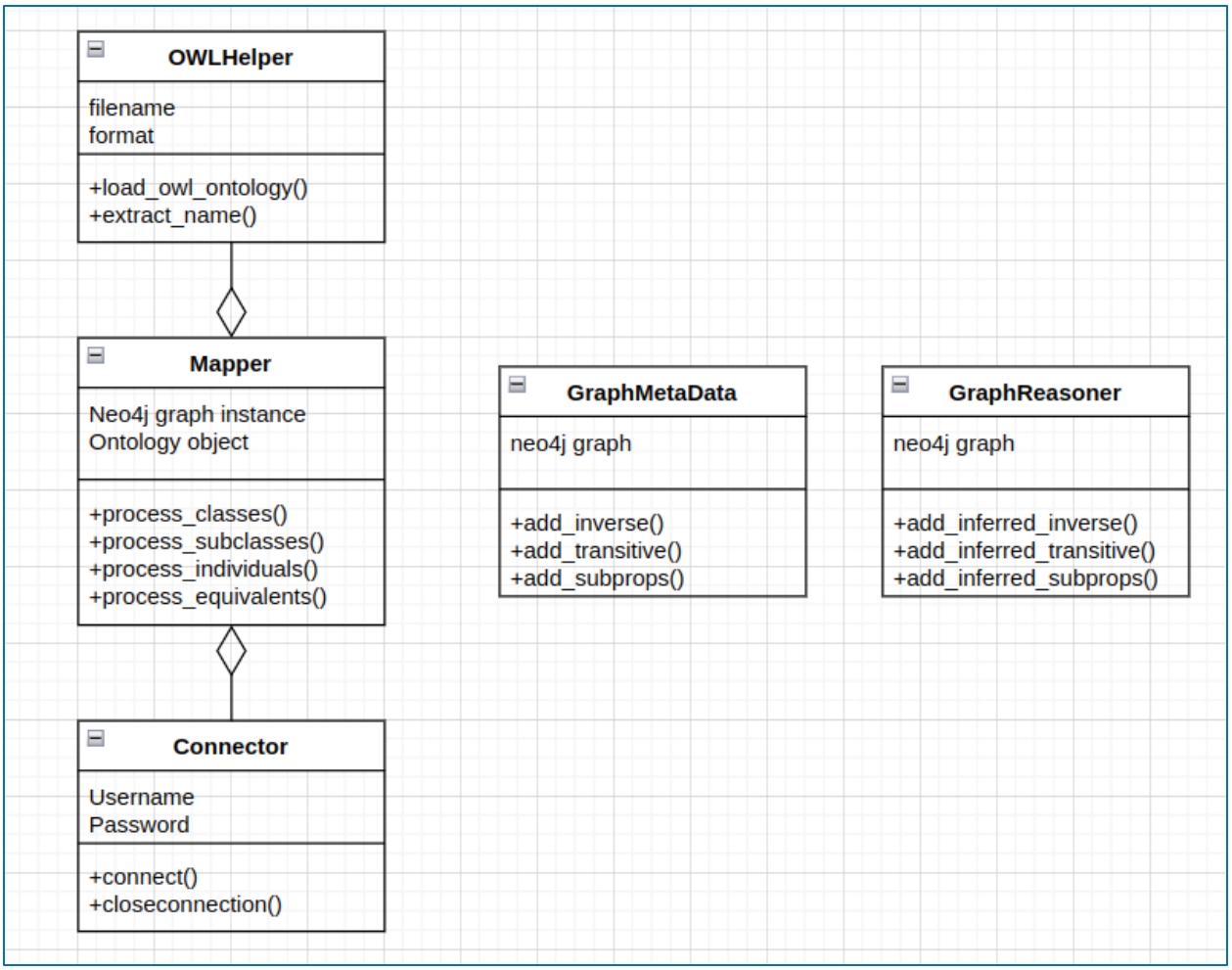
1. **OWL to LPG Mapping**
   The system parses a user-provided OWL (.owl) ontology file and maps its contents — classes, individuals, and object properties — into a labeled property graph (LPG) structure stored in a Neo4j database. Each OWL class is mapped as a label, individuals are mapped as nodes, and object properties are mapped as relationships between nodes.
2. **Custom Graph-Based Reasoner**
   A custom Python-based reasoner is developed to operate directly on the Neo4j graph. Alongside this, the inbuilt owlready2 is also available to switch to.
   This reasoner applies logical inference rules for the following.
   - Subclasses
   - Object Properties and its subproperty hierarchy
   - Cardinality data
   - Inverse properties
   - Transitive properties
   - Disjoint Classes
   - Equivalent Classes

3. **Inference Persistence**
   Any knowledge inferred by the custom reasoner — such as new subclass relationships or new object properties between individuals — is explicitly inserted into the Neo4j graph. This ensures that the graph evolves into a richer and more semantically complete structure after reasoning.
4. **User Frontend Interface**
   A simple and intuitive frontend allows users to upload an OWL file, initiate the ontology-to-graph conversion, and trigger the reasoning process. After reasoning, users can run Cypher queries on the enriched graph and view the results.

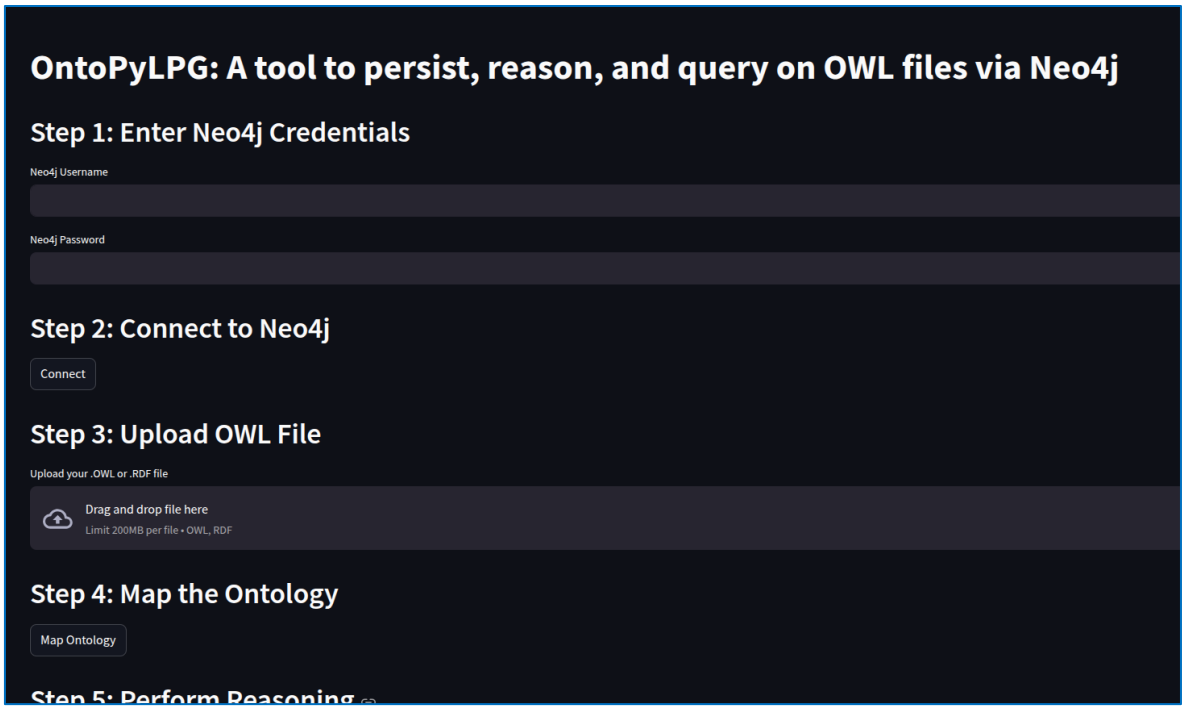
Project Design & Workflow



Flowchart Diagram



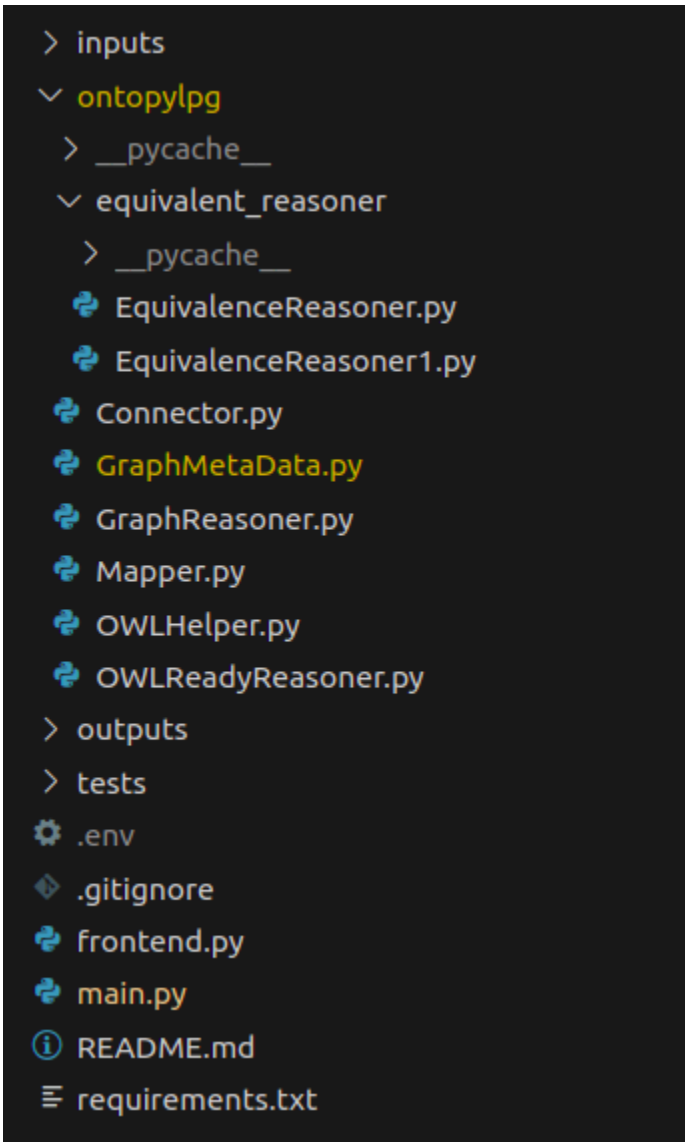
Class Diagram



Frontend Interface

Implementation Details

The current repository looks like this with the following files and folders



- Tests/: Contains test cases to validate functionality of the project modules.
- inputs/: Folder where OWL ontology input files are placed for processing.
- outputs/: Folder to store outputs like inferred ontology files or exported graph data.
- .gitignore: Specifies intentionally untracked files to ignore in the Git repository.
- ontopylpg/: Main Python package containing all ontology-to-graph functionality
    - Connector.py: Manages the connection to the Neo4j database
    - GraphMetaData.py: Extracts and stores ontology-level metadata

    - GraphReasoner.py: Core reasoning logic for subclass, object property, inverse property, etc.
    - Mapper.py: Converts OWL constructs into Neo4j property graph elements
    - OWLHelper.py: Helper functions for OWL term resolution and formatting
    - OWLReadyReasoner.py: Integrates Owlready2's built-in reasoner (if needed)
    - equivalent_reasoner/: Submodule for handling equivalence-specific reasoning
        - EquivalenceReasoner.py: Abstract class to implement equivalent reasoning functions
        - EquivalenceReasoner1.py: Implementation instance
- tests/: Contains test scripts (e.g., pytest-based unit tests)
- frontend.py: Streamlit-based GUI for user interaction (upload, connect, query)
- main.py: CLI entry point for running the full processing pipeline
- .env: Stores environment variables like Neo4j credentials
- .gitignore: Specifies files to ignore in version control
- README.md: Project documentation and usage instructions
- requirements.txt: List of Python package dependencies

Mapping

The Mapper class takes an **OWL ontology** file and converts its contents into a **Labeled Property Graph (LPG)** in **Neo4j** using **py2neo**. It reads classes, individuals, object properties, restrictions and inserts this structure as nodes and relationships in the graph.

Constructor

- Loads the ontology via an OWLHelper class.
- Stores reference to the Neo4j graph.
- Initializes a dictionary self.nodes to track added nodes (avoid duplicates).

Functions

1. **process_classes()**:
    - Iterates over all OWL classes.
    - Extracts class names.
    - Creates Class-labeled nodes in Neo4j.

2. **process_subclass_relationships():**
    - Handles *subClassOf* relationships between classes.

- Iterates through the parent classes of each class and adds SUBCLASS_OF edge for each pair.

3. **process_object_properties()**:
   - Iterates through all object properties and its parent properties in the ontology
   - Uses the domain and range of each property, if a property doesn't have a domain range, it assumes its child property's domain and range.
   - Creates relationships (edges) between domain and range class nodes using the property name.
   - Ignores abstract properties like ObjectProperty and TransitiveProperty.

4. **process_individuals():**
   - Adds individual instances as Individual-labeled nodes.
   - Adds object properties as edges between individuals.
   - Links them to their class types via INSTANCE_OF.
   - Adds data properties as attributes (properties) on nodes.

5. **process_subclass_restrictions():**
   - Adds the relations which are expressed via Restriction classes
   - Iterates through all the classes and checks for ancestors which are instances of Restriction and type *some*. This data is checked from the OWL file
   - For each of these nodes that satisfy the above condition, an edge is added to capture the property which is denoted by the Restriction class

6. **process_cardinality_data():**
   - Adds cardinality restrictions to class nodes
   - Iterates through all classes and checks for ancestors which are instances of type Restriction.
   - Captures the restriction type, cardinality and property name, and adds this information to the node.

7. **map_all():** Calls all the above processing functions in a specific order to build the full LPG from OWL:
   - Classes
   - Subclass relations
   - Individuals
   - Object properties

- Subclass Restrictions
- Cardinality Restrictions

Design Decisions

To implement this project the best we can and achieve our goals, we have made the following assumptions. The code will work completely **only if** these are upheld, else you may get incomplete results.

1. The input owl file must have domain and range specified for all the object properties
2. The ontology can fit into the *onto* object memory at all times, i.e. it is never too large.

Graph Metadata

Constructor

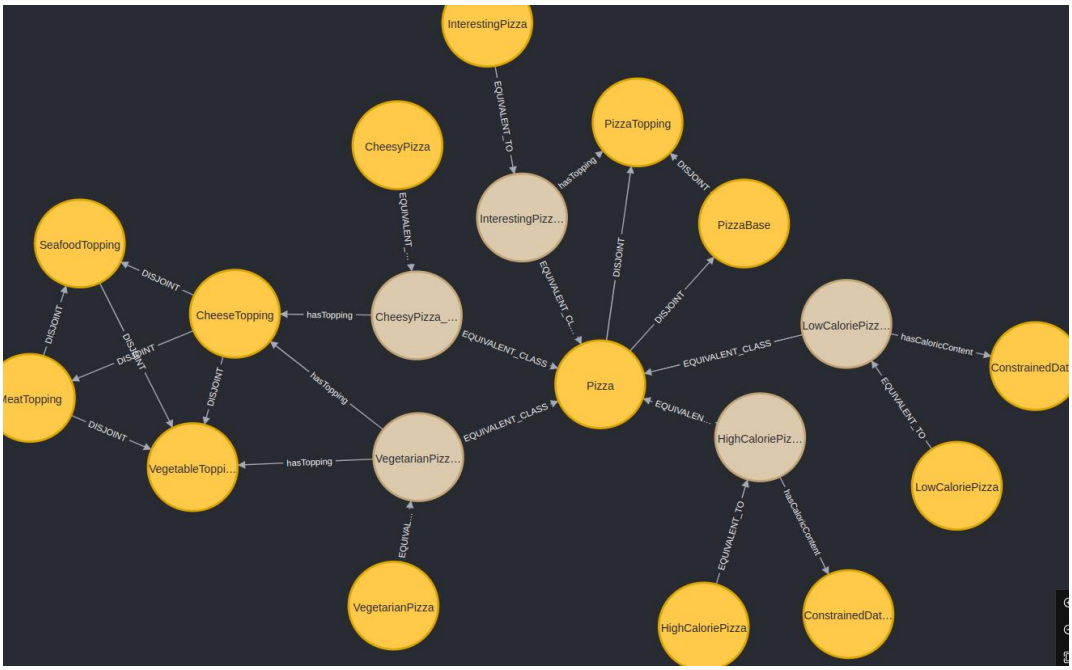
- Loads the ontology into memory
- Instantiates the instance of the graph to add data into
- Creates a mapping for property to edge names to use uniformly

Functions

1. **add_inverse_properties(self)**:
   - Adds inverse property relationships to the Neo4j graph. Iterates through all object properties in the ontology.
   - Checks if an object property has an inverse property. Creates nodes for the object property and its inverse.
   - Establishes an INVERSE_OF relationship between the two nodes and merges them into the graph.
2. **add_transitive_properties(self)**:
   - Adds transitive property relationships to the Neo4j graph. Iterates through all object properties in the ontology.
   - Checks if a property is a transitive property (owl.TransitiveProperty). Creates a node for the transitive property.
   - Establishes a self-relationship (TRANSITIVE) for the property and merges it into the graph.
3. **add_object_subproperties(**self):

- Adds subproperty relationships to the Neo4j graph. Iterates through all object properties in the ontology.
- Checks if a property has a parent property (rdfs:subPropertyOf). Creates nodes for the subproperty and its parent property.
- Establishes a SUBPROPERTY_OF relationship between the two nodes and merges them into the graph.

4. **add_disjoint_properties(self**):
    - Adds disjoint property relationships to the Neo4j graph.
    - Iterates through all classes in the ontology.
    - Identifies disjoint axioms and processes pairs of disjoint classes.
    - Creates nodes for the disjoint classes.
    - Establishes a DISJOINT relationship between the two nodes and merges them into the graph.
5. **add_equivalent_classes(self)**:
    - Iterates over all OWL classes.
    - Creates CLASS_PROPERTY nodes for each class.
    - Extracts their equivalentTo axioms.
    - Creates intermediate EQUI_COND nodes to model logical conditions.
    - Handles OWL logical constructs like:
        - **some, only, value** (PizzaOntology.hasTopping.some(PizzaOntology.CheeseTopping)
        - **Nested Or constructs** (PizzaOntology.hasTopping.only(PizzaOntology.CheeseTopping | PizzaOntology.VegetableTopping)
        - **Direct class equivalences** (PizzaOntology.Pizza)



In the image, the light brown nodes refer to the EQUI_COND nodes which are intermediate. All the edges that go out from here represent the axioms that define the equivalence.

6. **add_all(self)**:
    - A wrapper function that calls all the other functions (add_inverse_properties, add_transitive_properties, add_subproperty_relationships, and add_disjoint_properties) to add all metadata relationships to the Neo4j graph.

Graph Reasoner

This class performs **reasoning over a Neo4j Labeled Property Graph (LPG)** derived from an ontology. It infers additional relationships and flags logical inconsistencies such as violations of disjointness.

Constructor

- Instantiates the graph to work on
- Creates a dictionary object (called condition_handler) to map queries to their respective reasoning functions. This is needed for equivalent class reasoning.

Functions

1. **add_inferred_subclass_relationships():**
    - Iterates over all Class-labeled nodes.

- For each class, recursively adds transitive SUBCLASS_OF edges to capture multi-level hierarchies.
- Avoids self-loops and duplicates.

2. **process_subclasses(ancestor_node, current_node):**
    - Supports subclass inference by visiting each subclass recursively.
    - Adds SUBCLASS_OF edge from subclass to original ancestor.
    - Avoids redundant edges and cycles.

3. **find_subclasses(node) [helper func]:**
    - Finds all direct and indirect subclasses of a class.
    - Uses variable-length Cypher path: SUBCLASS_OF*.
    - Returns a flat list of subclass nodes.

4. **find_superclasses(node) [helper func]:**
    - Finds all superclasses of a class.
    - Uses Cypher path: SUBCLASS_OF*.
    - Applicable only to class nodes.

5. **add_inferred_inverse_properties():**
    - For every (a)-[:INVERSE_OF]->(b) pair of properties:
    - For each (x)-[:a]->(y), adds (y)-[:b]->(x).

6. **infer_disjoint_closure():**
    - Infers transitive closure of disjoint relationships.
    - For (a)-[:DISJOINT]->(b) and (b)-[:DISJOINT]->(c), adds (a)-[:DISJOINT]->(c) if not already present.
    - Operates on a node label CLASS_PROPERTY.

7. **mark_invalid_individuals(self)**:
    - Marks individuals that belong to two disjoint classes as invalid.
    - For each individual, checks if it is an instance of two disjoint classes.
    - If yes, sets an invalid property to True.
    - Relies on a separate CLASS_PROPERTY layer to track disjointness.

8. **process_disjoint_inference()**:
    - Runs both disjoint closure inference and individual validation.

- Calls infer_disjoint_closure() and mark_invalid_individuals().
- Helper wrapper for clarity and sequencing.

9. **add_inferred_transitive_relationships(self)**:
    - Adds inferred direct links for transitive object properties.
    - Identifies properties marked with TRANSITIVE relationship.
    - Finds chains of 2+ hops (*2..) and adds a direct shortcut edge.
    - Ensures no duplicate 1-hop edge already exists.
    - Only for object properties labeled as transitive. Handles both Class and Individual nodes (if label check is removed or generalized)
10. **propagate_restrictions_to_subclasses(self):**
    - Gets all class nodes with cardinality restriction properties
    - Finds all subclasses (indirect too) of this node
    - Get all keys from parent node that are cardinality restrictions, and set that property for the subclass node.
11. **propagate_subproperty_relationships(self):**
    - Gets all ObjectProperty nodes and their transitive ancestors via SUBPROPERTY_OF from the metadata
    - For each pair of class nodes connected by an existing relation, adds edges for all ancestors of that property

12. **apply_equivalence_reasoning(self):**
    - The GraphReasoner class contains a dictionary called condition_handler which maps each query to a reasoning class which processes the graph for the given condition.
    - Each query is written to find a certain pattern in the metadata describing a particular type of equivalence class definition.
    - The reasoning classes are all designed to be implementations of the **abstract class, EquivalenceReasoner** which contains one function – **evaluate_condition()**. This function contains the logic to infer new information from the condition provided.
    - Iterates over each condition (defined through a query) and applies logic using appropriate handler classes.

13. **perform_reasoning():**
    - Acts as the entry point to run all reasoning steps.

- Calls other methods in sequence to infer subclass, object property, and instance-level relationships.
- Assumes the LPG has already been populated from the OWL ontology.

Testing

We have used the **Pytest** framework to test our logic using assert statements for expected outputs after we perform the reasoning and get the output in the graph.

Setup Method:

- The setup_method() function is used to initialize the Neo4j graph connection and prepare the graph for testing. This ensures that the graph is accessible for all test cases.

Test Cases:

- Each test case runs a specific Cypher query to validate a particular aspect of the ontology mapping. The queries check for the existence of nodes and relationships in the graph that correspond to the ontology's structure.

Assertions:

- Each test case uses assertions to verify that the expected nodes and relationships exist in the graph. If the result of the query does not match the expected outcome, the test fails with a descriptive error message.

Test Coverage:

- test_transitive(): Verifies that transitive relationships (e.g., ISSPICIERTHAN) are correctly mapped.
- test_object_property(): Validates that object properties (e.g., HASTOPPING) are correctly represented in the graph.
- test_subclass(): Ensures that subclass relationships (e.g., SUBCLASS_OF) are correctly mapped.
- test_inverse(): Checks that inverse relationships (e.g., ISTOPPINGOF) are correctly represented.

Execution:

- Each test case runs independently, querying the graph and asserting the results. If all assertions pass, the test suite is successful; otherwise, it reports the failed test(s) with the corresponding error message.

Examples

Query: MATCH (parent:Class {name: "Pizza"})<-[:SUBCLASS_OF*]-(sub:Class)

RETURN sub.name

Explanation: Finds all direct and indirect subclasses of the class Pizza and returns their names.

Query: MATCH (ind:Individual {name: "MargheritaPizza1"})-[:INSTANCE_OF]->(cls:Class)

RETURN cls.name

Explanation: Returns the class (e.g., MargheritaPizza) that the individual MargheritaPizza1 is an instance of.

Query: MATCH (pizza)-[:HASTOPPING]->(topping:Class)

WITH pizza, collect(topping.name) AS toppings, count(topping) AS toppingCount

WHERE toppingCount = 2

RETURN DISTINCT pizza.name AS Pizza, toppings AS Toppings

Explanation: Finds all pizzas that have **exactly two toppings**, then returns their names and the list of those toppings.

Query: MATCH (:Class {name: "SohoPizza"})-[:HASTOPPING]->(topping:Class)

RETURN DISTINCT topping.name

Explanation: Get all the toppings of Soho Pizza

Query (inverse): MATCH (topping:Class)-[:ISTOPPINGOF]->(pizza:Class {name: "SohoPizza"})

RETURN DISTINCT pizza.name

Explanation: Finds which toppings lead to SohoPizza

Query: MATCH (c:Class) WHERE c.has_cardinality_hasTopping = "min:1" RETURN c.name AS class_name

Explanation: Get classes that have a minimum cardinality of 1 for hasTopping

Query: MATCH (a {name: "JalapenoPepperTopping"})-[:ISSPICIERTHAN]->(b)

RETURN a.name AS SourceNode, b.name AS LessSpicyNode

Explanation: This returns all nodes b that "JalapenoPepperTopping" points to via ISSPICIERTHAN.

Query: MATCH (i:Individual)-[:SUBCLASS_OF]-(c:Class {name: "Nothing"})

RETURN i,c

Explanation: This query finds all individuals that are invalid and there come under the Nothing class. The output should contain “CheeseTopping1”, which is invalid because it is an INSTANCE_OF two disjoint classes.

Query: MATCH (i:Individual {name: 'VegetarianPizza1'})-[:INSTANCE_OF]->(c:Class)

WHERE c.name IN ['VegetarianPizza', 'CheesyPizza']

RETURN collect(c.name) AS classes

Explanation: If VegetarianPizza1 is inferred to be an INSTANCE_OF VegetarianPizza or CheesyPizza, then the query returns the names of these inferred classes.

In the ontology, VegetarianPizza1 is defined as an Individual which is an INSTANCE_OF Pizza and hasTopping CheeseTopping and VegetableTopping. This satisfies the condition

to be inferred as both a VegetarianPizza and a CheesyPizza, by the equivalent class definitions.

Future Work

The following gaps are still there in achieving the maximum translation and semantic capturing in neo4j graph.

- Top/Bottom object/data properties
- Union, Intersection
- Property chaining
- Symmetric/Asymmetric property

These features are yet to be added. Along with these, we would also like to setup automatic build tools that test any commits using tools like Jenkins and deploy it using docker

Challenges Faced

These are some of the challenges we faced

- Finding a good solution to implement in this niche area and architecting it to be scalable and easily extendible
- Managing a huge codebase with logic running into multiple lines of code and doing version control frequently
- Since we are dealing with two distinct data models, the decision on where to compromise like expressing semantics at the tradeoff of storing multiple edges for everything was tricky.