A SPARQL to Cypher Transpiler

Student Name: Lakshya A Agrawal, Nikunj Singhal
Roll Number: 2018242, 2018249

BTP report submitted in fulfillment of the requirements
for the Degree of B.Tech. in Computer Science & Applied Mathematics

on May 6, 2022

BTP Track: Research

BTP Advisor
Dr. Raghava Mutharaju

Indraprastha Institute of Information Technology
New Delhi

Student's Declaration

We hereby declare that the work presented in the report entitled **A SPARQL to Cypher Transpiler** submitted by us for the fulfillment of the requirements for the degree of *Bachelor of Technology* in *Computer Science & Applied Mathematics* at Indraprastha Institute of Information Technology, Delhi, is an authentic record of our work carried out under guidance of **Dr. Raghava Mutharaju**. Due acknowledgements have been given in the report to all material used. This work has not been submitted anywhere else for the reward of any other degree.

.... **Place & Date: ....**
Lakshya A Agrawal

.... **Place & Date: ....**
Nikunj Singhal

Certificate

This is to certify that the above statement made by the candidate is correct to the best of my knowledge.

.... **Place & Date: ....**
Dr. Raghava Mutharaju

Acknowledgments

We want to acknowledge our advisor Dr. Raghava Mutharaju for providing us with the opportunity to work on this project and shape this project with his guidance. We also express our gratitude towards Dr. Harsh Thakkar for collaborating with us on this project.

Work Distribution

The work presented in this report has been completed over the Winter 2022 Semester (January 2022 - May 2022). This is the third semester of BTP work for both. Weekly meetings with the Advisor were setup to track progress. The semester started with a review of the concepts used for dataset conversion from PG to RDF and RDF to PG. Further, time was spent studying SPARQL and Cypher query languages and understanding RDF and Property Graphs and the differences between them. A review of existing query converters and their working was then carried out. A gap in the form of lack of mapping languages was identified and a methodology developed. Both members of the team contributed equally to the project.

Contents

Chapter 1

Introduction

Knowledge Graphs are gaining popularity due to their widespread usage in multiple applications across several domains. The two most common representations for Knowledge Graphs are triples (RDF Graphs) and Property Graphs (PG). RDF and its corresponding query language, SPARQL, are W3C standards that support reasoning. Property Graphs, on the other hand, offer greater flexibility in representing the data in the form of graphs. Depending on the PG management system, there are several options to query PG, such as Gremlin, Cypher, and GQL, with Cypher being one of the popular query languages of choice.

Making a good choice between an RDF or a PG stack is complex and requires a balanced consideration of data modeling aspects, query language features, and their adequacy for current and future use cases. Furthermore, the choice has an impact on factors such as ease of data maintenance, augmentation, and interchange, software ecosystems (such as explorers, visualization, and application development tools), or simply developer expertise and preferences [1]. Many commercial and open-source applications built on either RDF or Property Graphs cannot leverage the data from one another due to the lack of interoperability. The ability to use SPARQL or Cypher to query Property Graphs or RDF Graphs respectively enables the applications to interoperate between the two Worlds easily. Existing performance and test benchmarks targeting RDF triplestores could be used with little modification to benchmark the wide variety of Property Graph tools. It will further empower people proficient in SPARQL to query PG databases without having to learn a new query language.

Currently, there are techniques to convert RDF Graphs to Property Graphs [2]. There are also tools that are trying to eliminate the problem of query language interoperability by proposing a new graph data model which supports both the query languages [1], this comes with a problem that it is difficult for existing users to port their large datasets with millions of entries to this new data model. There are no existing tools to convert SPARQL to Cypher directly. In order to fill this gap, we propose a transpiler (a source to source compiler) to convert SPARQL queries to Cypher queries without having to port your entire data. Along with that, we also discuss an extendable testbench that supports automatic testing of the SPARQL to Cypher converter.

We outlined the three-pass approach for SPARQL to Cypher Transpiler development and prototype for the first and second step in [3]. We demonstrated that the process of query conversion is dependent on the underlying dataset conversion scheme(RDF to PG), which posed a challenge in making a generalizable SPARQL to Cypher Transpiler. Existing query converters assume a default mapping from RDF to PG, which forms the basis for the query conversion process, and constrains the user to writing queries as determined by the mapping. This work presents a query conversion approach based on a mapping language that creates a virtual RDF graph from a given Property Graph. The virtual RDF created is then used to automatically convert the SPARQL to Cypher, hence making the conversion process independent of the graph conversion scheme. The contributions of this work are as follows:

1. Review of concepts in graph data representation, graph data conversion scheme, and existing query language converters.
2. Identification of generalizability gap in existing work in Graph Query Conversion.
3. Proposal for solution based on a mapping language from PG to RDF to address the generalizability gap.
4. Evaluation of requirements from a PG-RDF mapping language in the context of transpiler development
5. Understanding, assessing, and evaluating RML as the mapping language for PG to RDF conversion
6. SPARQL-Cypher Conversion Scheme based on mapping language

Chapter 2

Related Work

2.1 AWS Neptune

The work presented in [1] identifies the problem faced by new AWS Neptune (A cloud-based Graph Database Platform) users in deciding between an RDF based or PG based model, and the expense in converting from one to another, making it an almost irreversible decision once the project is underway. The solution proposed is based on the concept of "quad" graphs that can simultaneously represent both RDF and PG by storing named-triples. The key challenges identified in creating interoperability between RDF and PG are: (i) PGs support multiple same-labeled edges emanating from the same vertex, whereas the same is not possible in RDF. (ii) Different type systems in use in the domains of RDF and PG. (iii) Mapping PG elements including labels to RDF IRIs or Literals (iv) Lack of formal definitions for most PGs

The Quad graph-based approach, while relevant to providing interoperability between RDF and PG, has the disadvantage that existing users of a PG database cannot make use of the Quad model as it would require conversion of large datasets. Therefore, this validates the need for a mapping language to create a virtual RDF from PG. The other challenges identified by [1] should be addressed by the mapping language.

2.2 SPARQL to GREMLIN

An approach to converting SPARQL queries into GREMLIN is developed by Gremlinator[4], and demonstrates the capability to use SPARQL to query property graphs(PG). A mapping from SPARQL's BGP to Gremlin's SSTs(Single Step Traversal[5]) is defined. The abstract syntax tree of the SPARQL query is generated, and then each BGP(basic graph pattern) in the query is visited and mapped to the corresponding GREMLIN SST. Then, it generates a bytecode that can be used on multiple languages and platforms like Apache Tinker-pop Gremlin family. The current version of the paper is limited to SPARQL SELECT queries without support for REGEX in FILTER. The paper also does not take into consideration that there can be multiple ways in which the queries can be converted, which boil down to the graph data conversion. Hence the paper assumes a default mapping and does the conversion according to that.

2.3 SPARQL to Cypher

A programmatic mapping from SPARQL to Cypher using Neosemantics [6] as the underlying RDF to PG conversion scheme is described in [7]. GraphDB is used as the RDF store. PEG.js [8], a JavaScript-based Parser generator, is used to create a SPARQL parser, and semantic actions are added to node visitors that convert the SPARQL query to Cypher. The work does not describe the grammar used for SPARQL parsing and the semantic actions used to effect the conversion between queries. The supported SPARQL constructs are PREFIX, same subject or different subject triples, FILTER(only with basic mathematical and logical expressions), LIMIT and ORDER BY, ASK and DESCRIBE. Evaluation and validation of the translation are performed by executing the queries over an RDF dataset imported into GraphDB [9], and into Neo4j using Neosemantics [6]. The work does not provide a formal proof of correctness of the translation and does not mention any publicly available tool.

2.4 SPARQL to SQL

Triplestores and Ontology-Based Data Access(OBDA) are two popular approaches to convert SPARQL queries to SQL. Triplestores provide a model to store any set of RDF triples. But if the data is in a different form(like relational database), then an intermediate ETL(extraction, Transform and Load) process is required to transform the data into RDF format which can then be queried easily. The ETL process can be expensive, especially if there are frequent updates in data sources. On the other hand, OBDA systems are set up over existing relational database sources and use the fact that their schemas are domain-specific. They, therefore, use ontologies and mappings to expose the relational database as a virtual RDF graph which can be queried using SPAQRL [10].

Ontop[10] describes an approach that allows SPAQRL to query relational databases by exposing relational databases as virtual RDF graphs. It does so by linking terms (class and properties) in the ontology to the data sources through mappings. Then these virtual RDF graphs can be queried using SPARQL by dynamically converting the SPARQL queries to SQL for the actual relational database, therefore, returning the required result. It also supports R2RML(RDB to RDF Mapping Language) mappings over general relational schemas. The authors conclude that they outperform other SPARQL to SQL convertors as well as triple stores by large margins[10]. R2RML is a W3C standard for expressing customized mappings from relational databases to RDF datasets.

A formal approach to converting SPARQL queries to SQL through R2RML mappings is described in [11]. The translation is divided into two steps: 1) Translating SPARQL queries to datalog(a declarative logic programming language) rules. 2) Generating relational algebra expressions from datalog programs and then converting to SQL queries using the standard translation of relational operators into corresponding SQL operators. [11] concludes that the generated datalog rules can be converted to any query language.

A translation function is proposed by [12] which takes two values as input: the query and two many-to-one mappings, which are a) a mapping between triples and tables and b) a mapping between pairs of the form (triple, pattern, position) and related attributes, with an assumption that the underlying relational DB is denormalized, and stores RDF terms.

2.5 SQL to SPARQL

A formal semantic is presented by [13] which preserves the translation from SQL to SPARQL. Only the schema and queries are mapped rather than converting the entire data. Schema mapping derives a domain-specific relational schema from RDF data. Query mapping transforms the SQL query over the schema into an equivalent SPARQL query, which in turn is executed against the RDF store.

2.6 RDF to PG

Neosemantics[6] is the official project for RDF handling in Neo4J. The data type properties in RDF are converted into node properties and object properties into relationships connecting nodes. Every node represents a resource and has a property with its IRI. $rdf : type$ statements are transformed into node labels. The IRIs identifying the elements in the RDF data (resources, properties, etc.) have their namespace part shortened to make them more readable and easier to query with Cypher. Prefixes for custom namespaces are assigned dynamically in sequence (ns0, ns1, etc.) as they appear in the imported RDF. There is an option for custom namespace prefixes to keep the complete IRIs in property names, labels, and relationships. In RDF, multiple values for the same property are signified by multiple triples. Neosemantics' default behavior is to keep only one value for literal properties, and it will be the last one read in the triples parsed. There are options to convert multiple values into a Neo4J array [14].

Chapter 3

Preliminaries

3.1 What is Knowledge? What is a Knowledge Graph?

We define knowledge as something that is known. Knowledge may be accumulated from external sources, or extracted from the knowledge graph itself [15]. Thus knowledge could pertain to the information that we are trying to store in our database in a structured manner, so that we can apply various operations on it as and when we need it.

A Knowledge graph is defined as a graph of data intended to accumulate and convey knowledge of the real world, whose nodes represent entities of interest and whose edges represent relations between these entities [15]. Therefore, a Knowledge Graph is a dataset that uses graphs to store and structure information(knowledge) so that it is connected, it makes it easier to query for information linked to one another. Resource Description Framework(RDF) and Property Graphs(PG) are the most common data models to represent Knowledge Graphs.

3.2 RDF: Resource Descriptive Framework

RDF is a standard model for data interchange on the Web [16]. An rdf graph is a set of triples each of the form (subject, predicate, object). RDF allows the following graph models to be expressed: Directed, Edge-Labelled and multi-graphs, but only if the multiple edges between the same nodes all have different labels.

Here, the subject is an IRI or a blank node, the predicate is an IRI, and the object is an IRI, blank node or literal.

IRI stands for Internationalized Resource Identifier. In RDF, an IRI is a UNICODE string that allows for the global identification of entities on the web[15].

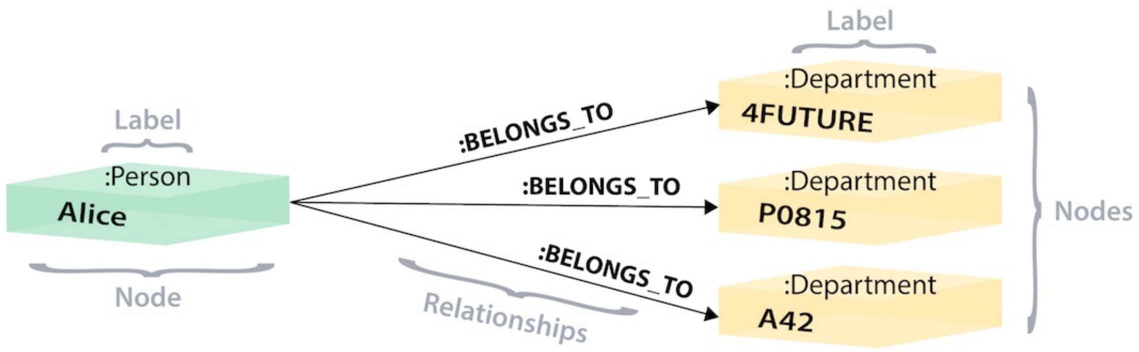


Figure 3.1: Property Graph

RDF Term

Let I be the set of all IRIs, RDF-L be the set of all RDF literals and RDF-B be the set of all RDF blank nodes, then set of RDF-Terms, RDF-T is: I ∪ RDF-L ∪ RDF-B.

Blank Nodes

RDF supports vertices that are not supported by an IRI and are called blank nodes. Intuitively, they are placeholders for some specific but unspecified node.

N-triples & Turtle

N-triples is a file format used to store RDF graphs. Turtle is an extension of N-triples with several convenient abbreviations.

3.3 Property graphs(PG)

A property graph is a type of graph model where relationships not only are connections but also carry a name (type) and some properties [17]. A property graph allows a set of property-value pairs to be associated with both nodes and edges and additionally a label associated with the edge [15].

3.4 SPARQL

SPARQL is a W3C standard query language and protocol for RDF databases. In addition to IRIs, literals, and blank nodes(in RDF), SPARQL also supports variables[17].

Variables

A variable is a string that begins with '?' followed by letters, numbers, and the symbol '-'.

Triple Pattern

A triple pattern is a triple of the form (S, P, O) where S and O can be an RDF terms or variables and P is an IRI or a variable. Example:

```
?album rdf:type :Album .
```

BGP: Basic Graph Patterns

Basic graph pattern is a set of triple patterns.

```
{
    ?album rdf:type :Album .
    ?album :artist ?artist .
}
```

Solution Mapping

A solution mapping μ, is a partial function μ: V $\rightarrow$ T. It is a mapping from set of variables to set of RDF terms.

SPARQL Algebra

A SPARQL Abstract Query is a tuple (E, DS, R) where E is a SPARQL algebra expression, DS is an RDF Dataset, R is a query form(SELECT, DESCRIBE, CONSTRUCT, ASK). For each symbol in a SPARQL abstract query, an operator for evaluation is defined which are used to evaluate SPARQL abstract query nodes [18]. This constitutes the SPARQL algebra. The SPARQL Algebra is a simplified AST for the SPARQL query which retains the semantic of the query, but removes information about query form.

3.5 Cypher

Cypher is one of the many popular query languages used for querying property graphs(Neo4j).

Node Patterns

Patterns for a node are described using parenthesis and are given a name[19]. It may optionally contain a variable name, list of labels and a set of properties. Example:

```
(a {name: 'Mike Tyson', sport: 'Boxer'})
```

Relationship Pattern

A pattern for a relationship between two nodes. It can be unidirectional or bidirectional. It may optionally contain a list of relationship types, a single label and a set of properties. The arrow tips indicate direction and absence of one means a bidirectional graphs[19]. Example:

```
(a) -[ r :REL_TYPE1 | REL_TYPE2 ]- (b) <- [ v {KEY:VAL} ] - (c)
```

Path Patterns

A series of connected nodes and relationships is a Path Pattern.

3.6 Boolean Satisfiability problem

Boolean expressions are built from boolean variables and constants using the operators AND, OR and NOT. Boolean Satisfiability problem, or SAT is the decision problem concerned with finding whether there is an assignment of values to the variables such that the expression evaluates to true.

3.7 Z3

Z3 is a Satisfiability Modulo Theories(SMT) Solver. SMT is a decision problem for logical formulas with respect to combinations of background theories over various domains including uninterpreted functions[20]. Z3 can be used to heuristically solve SAT problems.

Chapter 4

Approach

This work outlines steps towards building a SPARQL to Cypher Transpiler. There are two possibilities with regards to the origin of data to be queried by SPARQL after being converted to Cypher:

- Original Data modelled as RDF converted to PG : The original data that the user had was modelled in an RDF dataset, and the user had to convert it to a PG dataset. Now the user wants to convert all the SPARQL queries to corresponding cypher queries. Since we know that the data conversion can be performed in multiple ways as outlined in 4.1, and therefore the SPARQL to Cypher Transpiler should account for the underlying PG schema.
- Original Data modelled as PG directly : The original data that the user had was modelled in a PG database, however the user wanted to query it using SPARQL query language. Therefore the user provides a mapping from the original PG data model to an RDF data model and writes a SPARQL query according to that, the SPARQL to Cypher Transpiler accounts for the mapping and generates a valid Cypher query. Moreover, there are several PG semantics which do not have a direct equivalent in RDF, as described in 4.2, which adds the requirement of user provided schema being considered by the SPARQL to Cypher Transpiler.

Both of the above scenarios pose several challenges in the SPARQL to Cypher Transpilation process. SPARQL is built with RDF as a target and to provide the user flexibility to query Property Graphs using SPARQL, additional considerations need to be made.

4.1 Observations from Manual Query Conversion

- There can be multiple equivalent representations of the same fact as a property graph. Consider the fact "a knows John Doe" expressed in RDF and its conversions to PG: When

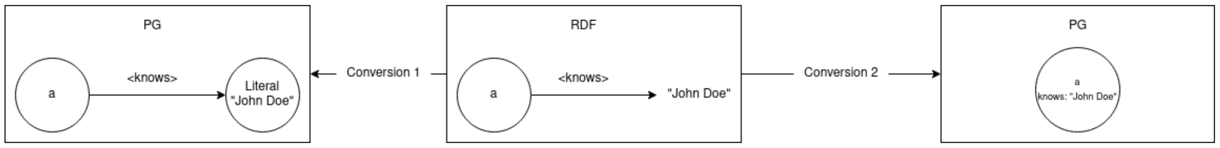


Figure 4.1: RDF Graph and corresponding conversion to PG

a user writes a query over the dataset, the SPARQL to Cypher Transpiler should provide the flexibility to address both the datasets. This leads to the second observation:

- The process of SPARQL to Cypher query conversion is dependent on the underlying PG schema.

  Consider the following SPARQL query to obtain the result "John Doe":

```
    SELECT ?b WHERE {
        <a> <knows> ?b
    }
```

  The corresponding Cypher query for Conversion-1 would be:

```
    MATCH ({name: "a"})-[:knows]->(b)
    RETURN b.lex
```

  And for Conversion-2:

```
    MATCH ({name: "a"})
    RETURN a.knows
```

  It can be seen that the way the queries have been converted is dependent on the underlying PG schema.

- There can be several syntactically different Cypher queries, that evaluate to the same result. This implies that there can be several equivalent query conversions for a given SPARQL query. Consider the following SPARQL query:

```
SELECT ?ee WHERE {
    ?ee a <Person>;
        <name> ?name .
    FILTER (?name = "John Doe")
}
```

  The following are two equivalent Cypher queries for the same:

```
MATCH (ee:Person)
WHERE ee.name = "John Doe"
RETURN ee
```

```
MATCH (ee:Person {name:"John Doe"})
RETURN ee
```

- The different SPARQL query forms(SELECT, ASK, DESCRIBE, CONSTRUCT) differ in the way the result is returned. The query patterns matched and computations performed however are the same. This was validated by the conversion of SPARQL to corresponding SPARQL Algebra using the ARQ [21] SPARQL processor. Hence, the conversion process should use the SPARQL Algebra for solution mapping, and the return statement could account for the query type.

4.2 Differences between RDF and PG

We consider the semantics allowed by the Property Graph definition in Neo4J:

- RDF nodes are of three types: IRI, Literal and blank node. An RDF node itself does not hold any data but rather acts as a placeholder for an entity in triples, which is the actual encoding of data. A node in a Property Graph, however, is a Data Structure with key-value pairs and labels that are encoding data. A SPARQL to Cypher Transpiler must provide mechanisms to expose the key and values within PG nodes.
- Nodes in PG can have (possibly zero) multiple labels, which can be used to distinguish them.
- PG edges have a single label associated with them. It is possible to have multiple connections of same type between same pair of nodes, however, in RDF, there can only be 0 or 1 edge of a given IRI between the same pair of nodes.
- PG edges can hold key-value pairs of data. RDF edges can just be IRIs. RDF allows for triples with other triples as the subject.
- Datatypes in PG systems is implementation-dependent, whereas RDF provides a mechanism of typed literals, under which a lexical representation of the value under predefined data types can be provided.

4.3 Mapping Language

Considering the above differences between the RDF and PG data model, a mapping language is required to map constructs in SPARQL query to constructs in Cypher. We do not need to actually convert the data, rather the transpiler needs to understand how the underlying virtual RDF would map to the Property Graph and hence convert the SPARQL query to Cypher accordingly.

4.3.1 Motivation for Mapping Language

Consider the following graph:

```
(:City {name:"Delhi"})-[:Flight {name: Indigo }]->(:City {name:"Mumbai"})
(:City {name:"Delhi"})-[:Flight {name: SpiceJet }]->(:City {name:"Mumbai"})
```

For this data to be queryable from SPARQL, the data needs to be mapped without loss into RDF. One lossless conversion is:

```
@prefix ex: <http://example.com/> .
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
_:Delhi rdf:a ex:City .
_:Delhi ex:name "Delhi" .
_:Mumbai rdf:a ex:City .
_:Mumbai ex:name "Mumbai" .
_:FlightIndigo rdf:a ex:Flight .
_:FlightIndigo ex:name "Indigo" .
_:FlightSpj rdf:a ex:Flight .
_:FlightSpj ex:name "SpiceJet" .
_:Delhi ex:edge_pre _:FlightIndigo .
_:FlightIndigo ex:edge_post _:Mumbai .
_:Delhi ex:edge_pre _:FlightSpj .
_:FlightSpj ex:edge_post _:Mumbai .
```

This conversion requires a way to map PG labels onto the ex: namespace(:City and :Flight in the given example), PG edges to be converted into 2 step edges(the :Flight edges), Information about the node not being an IRI but be created as a blank node.

With the above conversion, the following SPARQL query can be used to find the names of all flights between Delhi and Mumbai:

```
select ?flightname where {
   ?s rdf:a ex:City .
   ?s ex:name "Delhi" .
   ?t rdf:a ex:City .
   ?t ex:name "Mumbai" .
   ?s ex:edge_pre ?f .
   ?f ex:edge_post ?t .
   ?f rdf:a ex:Flight .
   ?f ex:name ?flightname .
}
```

Now, consider another conversion for the same PG to RDF, similar to the previous one, with the change that now nodes are given an IRI which is generated based on their values. This produces the relatively concise graph:

```
@prefix ex: <http://example.com/> .
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
_:FlightIndigo rdf:a ex:Flight .
_:FlightIndigo ex:name "Indigo" .
_:FlightSpj rdf:a ex:Flight .
_:FlightSpj ex:name "SpiceJet" .
_:FlightIndigo ex:edge_post <ex:City/Mumbai> .
_:FlightSpj ex:edge_post <ex:City/Mumbai> .
<ex:City/Delhi> ex:edge_pre _:FlightIndigo .
<ex:City/Delhi> ex:edge_pre _:FlightSpj .
```

The SPARQL query for the same result can now be expressed much more succintly:

```
select ?flightname where {
    ex:City/Delhi ex:edge_pre ?f .
    ?f ex:edge_post ex:City/Mumbai .
    ?f rdf:a ex:Flight .
    ?f ex:name ?flightname .
}
```

This can be reduced further by introducing IRI mappings for the edges:

```
select ?f where {
    ex:City/Delhi ex:edge_pre ?f .
    ?f ex:edge_post ex:City/Mumbai .
}
```

This reduction in query complexity is made possible by changing the mapping from PG to RDF, which is provided to the transpiler. Further, it enables the use of existing queries in SPARQL over RDF datasets, having a similar structure as the PG dataset, by creating an appropriate mapping.

4.3.2 Requirements from a Mapping Language

A Mapping Language from PG to RDF for the purpose of a SPARQL to Cypher Transpiler should address the following:

- Do the nodes in PG get mapped to an IRI or a blank node?
- If the nodes are mapped to IRIs, what is the template for IRI generation? Choices include using node labels, property keys and values along with RML like templates.
- Are the node labels exposed in the virtual RDF separately?
- Are the node properties exposed in the virtual RDF? If yes, what is the template for

property edge IRI generation?

- Conversion scheme for property value to RDF Literal. This also includes the datatype mapping from PG to RDF.
- Conversion of edge from PG to RDF. Choices: Create an intermediate node with source and destination connecting to this node, or directly create a single edge in RDF. In the second case, edge properties cannot be exposed.
- How is edge label exposed in virtual RDF?
- Depending on choice for conversion of PG edge, creation of IRIs for edges.

Multiple edges with the same label between same pairs of nodes can be handled by either creating an intermediate node, or by having the IRI of the edge in RDF be generated from properties of the labels. Finally, for the transpiler to be able to make inferences based on the mapping language, the mapping language should provide for a way to determine the inverse mapping from RDF to PG constructs, in order to map SPARQL constructs to Cypher. For example, if ”int” datatype in PG is mapped to xsd:int in RDF, then the transpiler must be able to identify the reverse mapping from xsd:int to ”int”. Similarly, mapping node properties to the IRI template ”ex:prop/@propname”, should make it feasible to infer that ”ex:prop/age” maps to the PG property ”age”.

4.3.3 Mapping Language in other Query converters

We found that the existing query conversion tools: [4], [1] and [6] assume a default mapping and do the conversion based on that. As was noted in the section 4.2, the various difference between RDF and PG models of data means that the query transpiler needs to assume a relation between constructs in RDF and constructs in PG, so as to convert the corresponding constructs in the SPARQL query to Property Graph. All the query converters reviewed so far have a fixed mapping and the queries are not converted dynamically based on any input mapping language. It is clear that the conversion does not depend on and cannot be changed based on any sort of mapping language.

4.3.4 Our Custom Mapping Language

In order to choose a mapping language from PG to RDF for the purpose of query transpilation, we evaluated the existing Mapping Languages: RML which is further based on R2RML[22]. RML is a generic mapping language defined to express customized mapping rules from heterogeneous data structures to RDF. It can be used to map data sources serialized as CSV, JSON and XML to generate RDF.

Since our usecase does not necessitate the actual conversion of data from a Property Graph to RDF, but rather to obtain a conceptual description of the mapping, we decided not to pursue

RML since the language has built abstractions for expressing rules over sources from CSV, JSON and XML. While it is theoretically possible for a Property Graph to be mapped to one of the above serializations without a loss of information, and therefore RML to describe the mapping, the added rules of serialization will prevent the user from making use of the flexibility being allowed by the Mapping Language. Further, due to the flexibility provided by RML in using any CSV, JSON and XML as data sources, the space of modeled data is not syntactically restricted to the Property Graph structure, therefore making it difficult to obtain required mapping information as outlined in 4.3.2.

Considering this, we decided to go ahead with building a custom Mapping Language, which conceptually maps Property Graphs directly to RDF. We reused concepts like iterators from RML to establish RDF triple generation rules.

Description of the mapping language

The mapping language follows a YAML based syntax. The data mapping are described in a declarative manner. The root level elements declare the entity for which conversion is described, followed by various templates for RDF generation. The following is an example of the mapping language with explanation of the various options. It is also the default recommendation considering the ease of usability as well as efficiency in capturing the user intent:

```yaml
# Describe the conversion of nodes
nodes:
  # This rule applies to nodes satisfying the following characteristics
  filter:
    labels: [] # Have the labels as per the list
    properties: {} # Have the key-value pairs as per the dictionary

  # Convert nodes to IRI according to the following template
  convertTo: "http://www.example.com/@prop[name]"

  # Convert Node Labels to IRI or Literal
  labelValConvertTo: IRI
  labels:
  - http://www.example.com/type # The edge from node to label
  - http://www.example.com/node/@LabName # Template for Label IRI

  # Convert Node property values to Literal or IRI
  propertyValConvertTo: Literal
  properties:
  # Template for edge from node to property value
  - http://www.example.com/node/propName/@propKey
  - '@propVal' # Template for literal generation
# Describe the conversion of edges
edges:
```

```
# Apply this conversion to edges satisfying the following characteristics
filter:
  properties: {} # Having the given properties
  # Source node with the following characteristics
  sourceFilter:
    labels: []
    properties: {}
  # Target node with the following characteristics
  targetFilter:
    labels: []
    properties: {}
  # Edge having the given type/label
  type: name
# Edges in PG are mapped to an edge in RDF according to the following template
convertTo: edge
# The edge template uses the edge label and one of its property values
edgeIRI: "http://www.example.com/edge/@LabName/@prop[name]"

# Convert Labels to Literal or IRI
labelValConvertTo: Literal
labels:
- http://www.example.com/edge/type # Edge leading to the Literal
- '@LabName' # Literal generation template

# Convert properties to literal or IRI
propertyValConvertTo: IRI
properties:
- http://www.example.com/edge/propName/@propKey # Edge template leading to the value
- http://www.example.com/edge/propVal/@propVal # Template for value IRI
```

4.4 Requirements for SPARQL to Cypher Transpiler

Based on a review of query translation (chapter 2) and observations from manual translation (section 4.1), the following requirements are determined for a SPARQL to Cypher Transpiler:

1. Flexibility: The SPARQL to Cypher Transpiler must be able to read a mapping in the Mapping Language and convert the queries accordingly.
2. Correctness: It must be demonstrated that the query conversion process retains the query semantics from the original language. There are two approaches to this:
   - Extensive testing: A testbench executing queries consisting of various language constructs over different datasets and demonstrating result identity across query conversion.

- Formal/algebraic proof of correctness: An algebraic argument based on mathematically precise models of RDF, PG, SPARQL, and Cypher references.

3. Efficiency: There can be several equivalent conversions for the same query with different performance as shown in 4.1. The converted query should be optimal over the target dataset.

4. Accounts for underlying data conversion schema: As shown in 4.1, the transpiler must account for the underlying data conversion schema.

4.5 Approach for Transpiler Development

The following 3 step approach to Transpiler development was outlined in [3]:

1. Build the SPARQL to Cypher converter with respect to a custom defined RDF to PG conversion scheme. The design goal with the RDF to PG scheme would be to have easy query conversion.

2. Modify the SPARQL to Cypher transpiler in step 1 to work with existing RDF to PG conversion schemes used in PGBench[23], and Neosemantics [6].

3. Build a generic SPARQL to Cypher transpiler that accounts for the underlying conversion scheme as a parameter and hence independent of the graph conversion process.

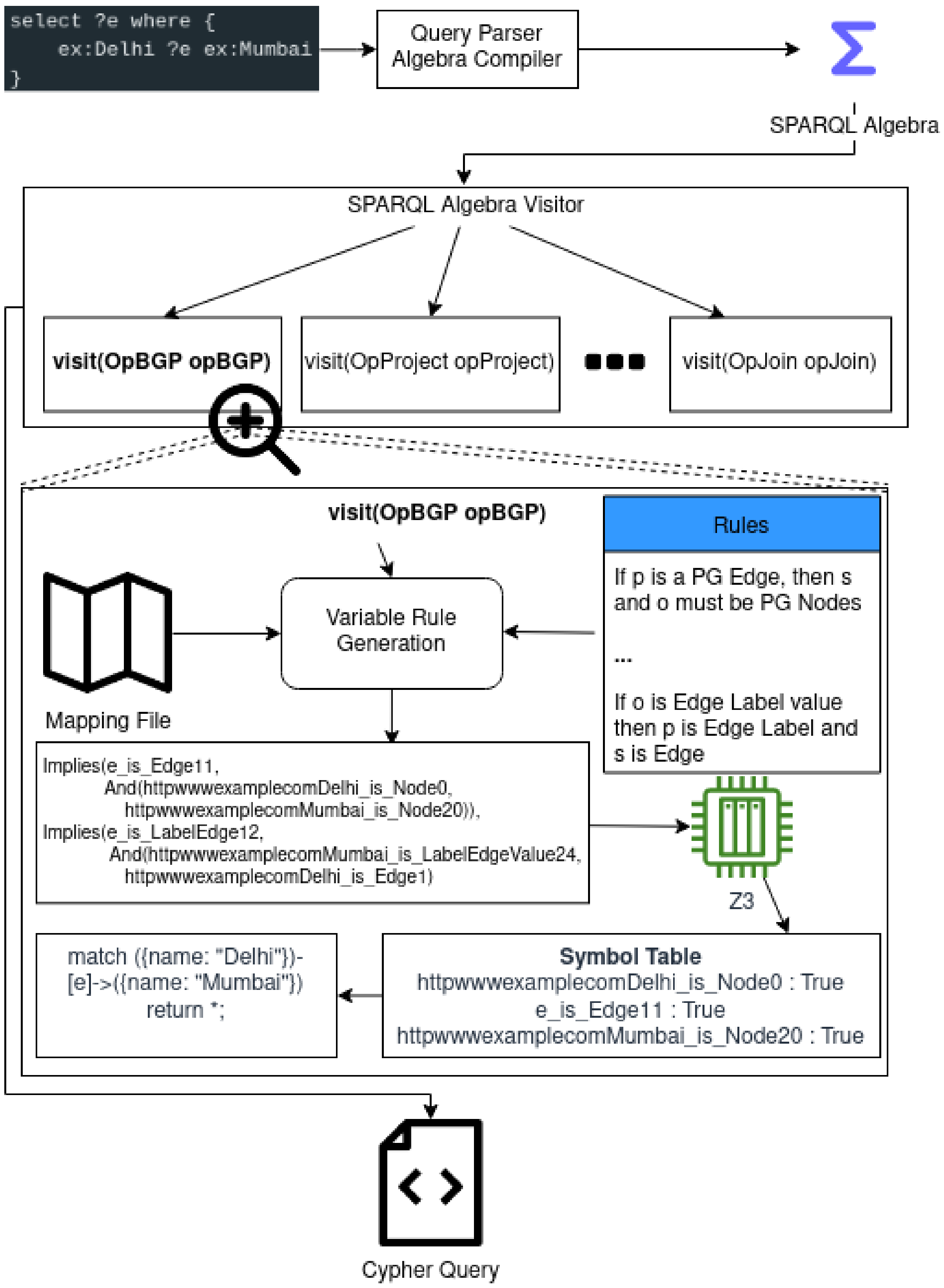
Following the demonstration of the first step in [3], there was a remodelling of the problem subsequent to the identification of a gap in the literature to target mapping-based query conversions. This work presents an approach towards building a SPARQL to Cypher Transpiler based on a mapping language satisfying the requirements as outlined in 4.3.

4.6 SPARQL to Cypher Transpiler

The mapping language-based SPARQL to Cypher Transpiler proceeds in the following phases:

4.6.1 SPARQL to ARQ Algebra conversion

The SPARQL to Cypher Transpiler internally uses RDFLib Algebra utilities to create a SPARQL algebra for the given SPARQL query. As was discussed under 4.1, the underlying algebra for several types of SPARQL queries is the same with just the return values changing. Further, conversion of SPARQL to SPARQL Algebra also removes elements of SPARQL introduced for syntactic ease, thus reducing the conversion complexity.



```
select ?e where {
    ex:Delhi ?e ex:Mumbai
}
```

```
Implies(e_is_Edge11,
        And(httpwwwexamplecomDelhi_is_Node0,
            httpwwwexamplecomMumbai_is_Node20)),
Implies(e_is_LabelEdge12,
        And(httpwwwexamplecomMumbai_is_LabelEdgeValue24,
            httpwwwexamplecomDelhi_is_Edge1)
```

```
httpwwwexamplecomDelhi_is_Node0 : True
e_is_Edge11 : True
httpwwwexamplecomMumbai_is_Node20 : True
```

```
match ({name: "Delhi"})-
[e]->({name: "Mumbai"})
return *;
```

Figure 4.2: SPARQL to Cypher Transpiler Architecture

4.6.2 Cypher Generation

The process of Cypher generation proceeds by visiting the SPARQL Algebra, which forms a tree data structure. Visitor methods are defined in the Transpiler for Algebra graph patterns, solution modifiers, property path and query types. The Visitor implements conversion for each node by recursively converting the child nodes, modifying the result, and converting the target node and then returning it.

Since the Mapping Language based inference is used to guide graph pattern match generation, the visitor for BGP is elaborate and discussed in the next section.

Remaining visitor implementations:

- Select: The select query is implemented as a Cypher "return" statement with the corresponding variable names.
- Project: Project solution modifiers are implemented as Cypher "with" statements similar to select queries.

Variable Inferencing in BGP

Based on the mapping language described in 4.3.4, all variables used in the SPARQL Algebra are classified as referring to one of:

- Node
- PG Node property predicate
- PG Node property value
- PG Node Label Predicate
- PG Node Label Value
- Edge
- PG Edge Property value predicate
- PG Edge Property value
- PG Edge label value
- PG Edge Label predicate edge

The aim of variable inferencing strategies is to classify each entity appearing in the BGP as one of the above. Such information will be utilized in later steps to build Cypher Graph patterns from the mapped SPARQL Basic Graph Patterns.

Variable inferencing can be performed by inferencing from mapping language. We propose a procedure based on solving SAT problems for the same. To evaluate our procedure, we use the Z3 SMT solver to solve SAT problems.

A Boolean variable is created for each entity in the BGP, for each possibility of PG Graph Concept. So, given a variable ”?s” in the BGP, a total of 10 boolean variables will be created.

A BGP consists of sets of triples, where each entity in the triple is either a variable, literal or an IRI. Based on the rules of construction for RDF, Property Graphs and constraints defined in our Mapping Language, we can impose constraints on the potential type that a SPARQL entity is mapped to. These constraints can be represented as boolean expressions over the variables defined for each entity. These constraints consist of rules like the following: ”Given a triple (s p o), If o is a PG Node, then p must be a PG edge”, which is then encoded as the following expression: ”Implies(oIsNode, pIsEdge)”

The complete list of constraints currently in use in the SPARQL to Cypher system is available at A.

Running the set of constraints obtained from the interaction of the SPARQL Query with the Mapping Language and in-built set of rules, results in the Constraint Solving system like Z3 to present either of the following 3 states:

- UNSAT: This occurs when there is a contradiction between the query elements and the rules. For example, if the mapping language maps all Node labels to RDF String literals, and the mapping language specifies the IRI ”http://example.com/NodeLabel” as the predicate for Node labels, and the user query has the following:

  ```
  select * where {
      ?s <http://example.com/NodeLabel> ?o .
      ?o ?q ?t .
  }
  ```

  Here, the variable ?o is expected to be a Literal, and therefore shouldn’t have any outgoing edges, which is being contradicted by the second triple.

- SAT with multiple solutions: We treat this scenario as when the query is underspecified for mapping to PG concepts. For example, given the following query:

  ```
  select * where {
      ?s ?p ?o .
  }
  ```

  The given query is expected to return all the triples in an RDF graph, but since the virtual RDF consists of concepts that do not exist in PG (like PG Node Label Predicate), such a query cannot be mapped to Cypher.

- SAT with unique solution: In this case, we proceed into the next step to generate Cypher corresponding to the BGP.

BGP to PG Graph Pattern Conversion

From the procedure described in 4.6.2, we obtain a mapping from entities in the SPARQL BGP to the corresponding concepts in PG, that these entities represent.

The Cypher Graph Matching is based on ASCII art description of the graph, consisting of variables, nodes, node labels, properties, edges, edge labels and edge properties.

In this step, we build a model PG from the obtained mapping from SPARQL entities to PG concepts.

We first obtain all the entities which refer to nodes in the PG. Next, we obtain all the entities which refer to edges in the PG and create a directed-graph data structure to represent the connections. Further, we obtain information about the labels, property keys and property values, and associate them with their corresponding nodes and edges. Each of these steps requires iterations over the triples, and matching against the various Mapping Language Templates to obtain the values.

Match queries generation

Having obtained the model PG corresponding to the input SPARQL BGP, we implement a procedure to generate the ASCII Art based PG Graph Pattern corresponding to the model PG.

4.7 Testbench

Following Test Driven Development [24] practices, a testbench for evaluating SPARQL to Cypher conversion over an implicitly assumed RDF-PG mapping was demonstrated in [3]. With a change in approach from fixed RDF-PG mapping to a conversion based on PG-RDF mapping, we propose a new testbench. The testbench is aimed to provide fast feedback and exhaustively test the transpiler over various datasets and language constructs. It is also made for easy extension of test cases over different datasets and queries.

4.7.1 Implementation

The testbench is implemented as a Junit 5 parametric test [25]. The parameters are generated from a test data directory, with each directory containing a serialized PG as a JSON file along with multiple pairs of SPARQL and Mapping language files. Each test case consists of a dataset from one of the PG files and a SPARQL-Mapping pair. The testbench initializes a PG model object corresponding to each PG JSON file, and calls into a function to convert the PG to RDF as per the conversion specified in the mapping, and loads the RDF in an in-memory Jena based database. Neo4j was chosen for PG since Cypher was originally created to be used in

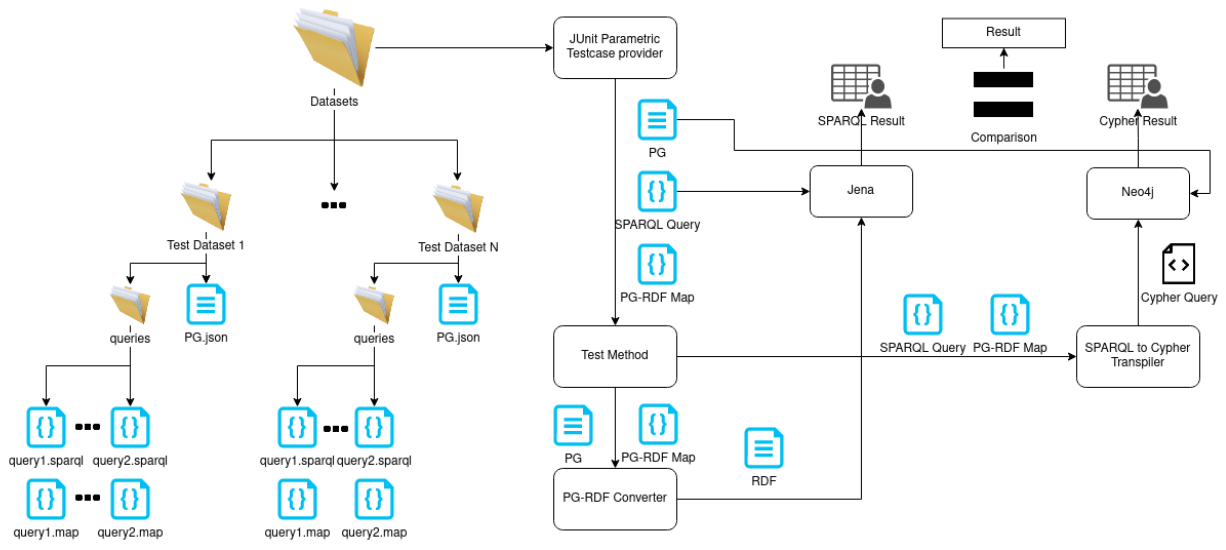


Figure 4.3: Testbench Architecture

Neo4j, and its Cypher implementation is continuously updated [26]. The testbench executes the SPARQLToCypherConverter on every query associated with the test case. The original SPARQL query is executed against the Jena model and the converted Cypher query against the Neo4j Database. The results from both the databases are then compared, accounting for ORDERBY operations.

Chapter 5

Evaluation

5.1 Approach for Evaluation

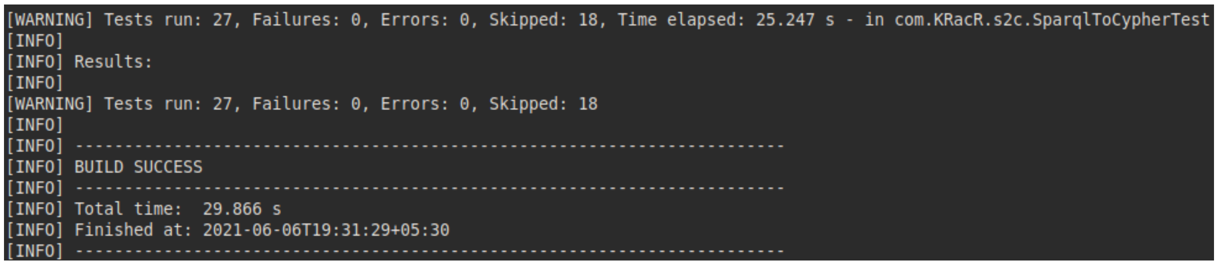
To evaluate the proposed approach, we obtain RDF based datasets and benchmarks from existing literature, and utilize the Neosemantics [6] system to convert the RDF to PG and insert into a live PG database. Further, we use the mapping language to map the benchmark SPARQL queries to the PG dataset.

5.2 Transpilation Test criteria and test case source

We evaluate the custom SPARQL to Cypher Transpiler built during the first pass of the transpiler development on the testbench described in section 4.7, as it can verify whether the results returned by the original and converted query are the same or not. The testbench is developed generically to make it easy to add test cases. Currently, the RDF data and query resources are:

- SP^2Bench [27]: The SP^2Bench is a SPARQL language-specific benchmark framework specifically designed to test the most common SPARQL constructs, operator constellations, and a broad range of RDF data access patterns. the queries implement meaningful requests on top of automatically generated data, covering a variety of SPARQL operator constellations and RDF access patterns [27]. An RDF graph generated using the SP^2Bench data generator consisting of 100 statements is used along with the queries outlined in the paper for our testing. There are 17 such queries in the testbench.

- W3C SPARQL Recommendation Document [18]: This specification defines the syntax and semantics of the SPARQL query language for RDF. It has been reviewed by W3C Members, software developers, and other W3C groups and interested parties and is endorsed by the Director as a W3C Recommendation. It is a stable document, and due to the standard nature of SPARQL, example RDF graphs from the document along with query examples were included in the testbench [18]. Currently, 10 such query files over 5 RDF graphs are



```
[WARNING] Tests run: 27, Failures: 0, Errors: 0, Skipped: 18, Time elapsed: 25.247 s - in com.KRacR.s2c.SparqlToCypherTest
[INFO]
[INFO] Results:
[INFO]
[WARNING] Tests run: 27, Failures: 0, Errors: 0, Skipped: 18
[INFO]
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------------------
[INFO] Total time:  29.866 s
[INFO] Finished at: 2021-06-06T19:31:29+05:30
[INFO] ------------------------------------------------------------------------
```

Figure 5.1: Test results for query transpilation

included in the testbench.

5.3 Results

The testbench can skip the test cases consisting of queries having unsupported constructs. The results are reported as the Number of tests passed, failed, and skipped by the testbench. The currently supported query types are described in 4.6 above, which include SELECT queries with Basic Graph Patterns and variable projections. Based on this, out of 27 test cases, 18 test cases are skipped due to containing unsupported constructs. The rest of the test cases over supported language constructs are successfully verified.

The converted queries for all supported query types return the same results as the corresponding SPARQL query, and hence pass the test. Further, the queries can be used to test the mapping based transpiler.

Chapter 6

Next Steps

A SPARQL to Cypher Transpiler capable of handling SELECT queries with Basic Graph Patterns over a custom-defined RDF to PG conversion scheme was demonstrated in [3]. Following the approach outlined in 4.5, an approach to a generalizable SPARQL to Cypher Transpiler based on a Mapping language has been demonstrated in this work. A working prototype for the same was developed as a part of the project work during the semester. Further, extension to the 4.6.2 phase can be affected by developing SPARQL algebra handlers for the remaining graph patterns and solution modifiers. Visitor design patterns will be helpful in exhaustively covering the language constructs.

It has been demonstrated in this work that there can be different equivalent queries with the same results. The transpiler testbench will be extended to support query performance benchmarking, which will further be used to work on query optimizations, and choosing the conversion with the best possible performance on the converted dataset.

The algebraic nature of queries, and mathematically precise definitions of the graph data model in RDF and Property Graphs will allow for formal arguments to prove the correctness of the query conversion process, which has not been seen in existing query conversion work for SPARQL and Cypher. This would require further study of algebraic formalisation of Cypher [28].

Property Graph standardization efforts are focusing on the GQL Language for PG queries, and it has been shown in the proposals that GQL is inspired by and similar to Cypher. This paves way for this work to be resused for a SPARQL to GQL Transpiler, which can be developed once the GQL Standard is ready.

The tool will be implemented as a publicly available web application along with Open Source release of the codebase. We will be continuing with the BTech Project next semester to make a fully implemented and functioning version of the work done so far.

Bibliography

[1] Ora Lassila et al. "Graph? Yes! Which one? Help!" In: *CoRR* abs/2110.13348 (2021). arXiv: `2110.13348`. URL: `https://arxiv.org/abs/2110.13348`.

[2] Olaf Hartig. "Reconciliation of RDF* and Property Graphs". In: *CoRR* abs/1409.3288 (2014). arXiv: `1409.3288`.

[3] Lakshya A Agrawal, Nikunj Singhal, and Raghava Mutharaju. "A SPARQL to Cypher transpiler: Proposal and Initial Results". In: *5th Joint International Conference on Data Science Management of Data (9th ACM IKDD CODS and 27th COMAD) (CODS-COMAD 2022), January 8–10, 2022, Bangalore, India.* 2022. ISBN: 978-1-4503-8582-4/22/01. DOI: `10.1145/3493700.3493757`.

[4] Harsh Thakkar et al. "A Stitch in Time Saves Nine – SPARQL querying of Property Graphs using Gremlin Traversals". In: (Jan. 2018).

[5] Marko A. Rodriguez. "The Gremlin graph traversal machine and language (invited talk)". In: *Proceedings of the 15th Symposium on Database Programming Languages* (Oct. 2015). DOI: `10.1145/2815072.2815073`. URL: `http://dx.doi.org/10.1145/2815072.2815073`.

[6] Jesús Barrasa and Adam Cowley. *neosemantics (n10s): Neo4j RDF Semantics toolkit - Neo4j Labs.* URL: `https://neo4j.com/labs/neosemantics/`.

[7] Ezequiel José Veloso Ferreira Moreira and José Carlos Ramalho. "SPARQLing Neo4J (Short Paper)". In: *9th Symposium on Languages, Applications and Technologies (SLATE 2020).* Ed. by Alberto Simões, Pedro Rangel Henriques, and Ricardo Queirós. Vol. 83. OpenAccess Series in Informatics (OASIcs). Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum für Informatik, 2020, 17:1–17:10. ISBN: 978-3-95977-165-8. DOI: `10.4230/OASIcs.SLATE.2020.17`. URL: `https://drops.dagstuhl.de/opus/volltexte/2020/13030`.

[8] Futago-za Ryuu. *PEG.js – Parser Generator for JavaScript.* URL: `https://pegjs.org/` (visited on 06/07/2021).

[9] *GraphDB.* URL: `https://graphdb.ontotext.com` (visited on 06/07/2021).

[10] Diego Calvanese et al. "Ontop: Answering SPARQL Queries over Relational Databases". In: *Semantic Web J.* 8.3 (2017). Semantic Web Journal outstanding paper award for 2016, pp. 471–487. DOI: `10.3233/SW-160217`.

[11] Mariano Rodríguez-Muro and Martin Rezk. *Efficient SPARQL-to-SQL with R2RML Mappings*. June 2018. URL: https://papers.ssrn.com/sol3/papers.cfm?abstract_id=3199192.

[12] Artem Chebotko, Shiyong Lu, and Farshad Fotouhi. "Semantics preserving SPARQL-to-SQL translation". In: *Data & Knowledge Engineering* 68.10 (Oct. 2009), pp. 973–1000. DOI: 10.1016/j.datak.2009.04.001. URL: https://doi.org/10.1016/j.datak.2009.04.001.

[13] Jyothsna Rachapalli et al. "RETRO: A framework for semantics preserving SQL-to-SPARQL translation". In: *ISWC 2011*. 2011.

[14] *Chapter 4. Importing RDF data*. URL: https://neo4j.com/docs/labs/nsmntx/current/import/#actual-rdf-import.

[15] Aidan Hogan et al. *Knowledge Graphs*. Jan. 2021. URL: https://arxiv.org/abs/2003.02320.

[16] *RDF*. Feb. 2014. URL: https://www.w3.org/RDF/.

[17] Michelle Knight. *What Is a Property Graph?* Apr. 2021. URL: https://www.dataversity.net/what%5C-is%5C-a%5C-property%5C-graph/.

[18] *SPARQL Query Language for RDF*. June 6, 2021. URL: https://www.w3.org/TR/rdf-sparql-query/.

[19] *Patterns - Neo4j Cypher Manual*. URL: https://neo4j.com/docs/cypher-manual/current/syntax/patterns/.

[20] Nikolaj Bjørner et al. *Programming Z3 - Stanford CS theory*. URL: https://theory.stanford.edu/~nikolaj/programmingz3.html.

[21] *ARQ - SPARQL Algebra*. URL: https://jena.apache.org/documentation/query/algebra.html.

[22] *RDF mapping language (RML)*. URL: https://rml.io/specs/rml/.

[23] Meenakshi Maindola and Raghava Mutharaju. "PGBench: A Property Graph Benchmark for Knowledge Graphs". MA thesis. 2020.

[24] Kent Beck. *Test Driven Development: By Example*. USA: Addison-Wesley Longman Publishing Co., Inc., 2002. ISBN: 0321146530.

[25] Sam Brannen et al. URL: https://junit.org/junit5/docs/current/user-guide/.

[26] Nadime Francis et al. "Cypher: An Evolving Query Language for Property Graphs". In: *Proceedings of the 2018 International Conference on Management of Data*. SIGMOD '18. Houston, TX, USA: Association for Computing Machinery, 2018, pp. 1433–1445. ISBN: 9781450347037. DOI: 10.1145/3183713.3190657. URL: https://doi.org/10.1145/3183713.3190657.

[27] Michael Schmidt et al. "SP2Bench: A SPARQL performance benchmark". In: May 2009, pp. 222–233. DOI: 10.1109/ICDE.2009.28.

[28] József Marton, Gábor Szárnyas, and Dániel Varró. “Formalising openCypher Graph Queries in Relational Algebra”. In: *Advances in Databases and Information Systems.* Ed. by Mārıte Kirikova, Kjetil Nørvåg, and George A. Papadopoulos. Cham: Springer International Publishing, 2017, pp. 182–196. ISBN: 978-3-319-66917-5.

Appendix A

List of rules for Variable Inferencing

- Any entity x can only be one of the possible entity types at a time and hence only one of the Bools must be true.
- Given a triple (s p o), s can only be either a PG Node or PG edge
- Given a triple (s p o), p can only be PG Edge, Edge label, Edge property, Node label or Node property
- Given a triple (s p o), o can only be Edge Label Value, Edge property value, Node Label value, Node property value or PG Node
- Given a triple (s p o), o can never be node property, edge property, node label, edge label, PG Edge
- Given a triple (s p o), If o is a PG Node, then p must be a PG edge
- Given a triple (s p o), If p is a PG Edge, then s and o must be PG Nodes
- Given a triple (s p o), If p is Edge Label then o is Edge Label value and s is Edge
- Given a triple (s p o), If p is Edge Property then o is Edge Property value and s is Edge
- Given a triple (s p o), If p is Node Label then o is Node Label value and s is Node
- Given a triple (s p o), If p is Node Property then o is Node Property value and s is Node
- Given a triple (s p o), If o is Edge Label value then p is Edge Label and s is Edge
- Given a triple (s p o), If o is Edge Property value then p is Edge Property and s is Edge
- Given a triple (s p o), If o is Node Label value then p is Node Label and s is Node
- Given a triple (s p o), If o is Node Property value then p is Node Property and s is Node
- Given a triple (s p o), If (s) satisfy the m["nodes"]["convertTo"] template then it is Node

- Given a triple (s p o), If (s) satisfy the m[”edges”][”edgeIRI”] template then it is Edge, p is edge property or edge label, o is edge property value or edge label value
- Given a triple (s p o), if p satisfy the m[”nodes”][”labels”] template, then it is LabelNode, o is node label value, s is node label
- Given a triple (s p o), if p satisfy the m[”nodes”][”properties”] template, then it is PropertyNode, O propertyval thus literal, s is node
- Given a triple (s p o), if p satisfy the m[”edges”][”labels”] template, then it is LabelEdge, o is edge label value, s is edge
- Given a triple (s p o), if p satisfy the m[”edges”][”properties”] template, then it is PropertyEdge, O propertyval thus edge property value, s is edge
- Given a triple (s p o), if p satisfy m[”edges”][”edgeIRI”] template, then it is Edge, s is node, o is node
- Given a triple (s p o), if o satisfy the m[”nodes”][”convertTo”] template then it is Node, p has to be Edge as it cannot be any label or property, s has to be Node
- Given a triple (s p o), if o satisfy the m[”nodes”][”properties”][1] i.e IRI template for propert value then it is Node property value, p has to be Node property, s has to be Node
- Given a triple (s p o), if o satisfy the m[”nodes”][”labels”][1] i.e IRI template for label value then it is Node label value, p has to be Node label, s has to be Node
- Given a triple (s p o), if o satisfy the m[”edges”][”properties”][1] i.e IRI template for propert value then it is Edge property value, p has to be Edge property, s has to be Edge
- Given a triple (s p o), if o satisfy the m[”edges”][”labels”][1] i.e IRI template for label value then it is Edge label value, p has to be Edge label, s has to be Edge
- Given a triple (s p o), If o is a Literal, then it could be a property value or a Label value, Correspondingly, p must be a property edge or a label edge or property node or label node, s must be either Node or Edge