UNIVERSITY OF TARTU
Institute of Computer Science
Computer Science Curriculum

Juan Carlos Javier Ramos Martínez

Quarser: a graph-aware JSON-LD parser

Master's Thesis (30 ECTS)

Supervisor(s): Riccardo Tommasini, PhD

Tartu 2021

Quarser: a graph-aware JSON-LD parser

Abstract:
The continuous growth of the Web of Data has fueled the interest of performing analytical operations over Knowledge Graphs (KGs). The challenge of handling large scale KGs foster the research on optimization and benchmarking of existing Semantic Web solutions. Most of them focus on query planning in the context of one-time queries. Nonetheless, the spreading of application domains like Internet of Things (IoT), Social Media Analytics, and News Analysis has focused attention on different kinds of queries that tend to be recurrent. Our focus is on the performance optimization of recurrent analytical SPARQL queries by leveraging the computation spent on the parsing process of data. The literature on this type of optimization in SQL workloads is being recently explored with positive results. To the best of our knowledge, in the Semantic Web landscape, the effort has been minimal. The current thesis presents a new JSON-LD parser called Quarser, that is particularly tailored to this class of applications. Quarser is aware of the RDF graph that the parser traverses and shares the same space to compute SPARQL variable bindings. Our results, tested over the LUBM Benchmark, show a reduction of 20% of the total time of query-answering.

Keywords: JSON-LD, RDF Graphs, SPARQL, Pushdown parsing

CERCS: P170 - Computer Science, numerical analysis, systems, control

Quarser: graafiteadlik JSON-LD parser

Lühikokkuvõte:
Üha suurenev veebiandmete maht on kaasa toonud suurenenud huvi Knowledge Graph (lüh. KG) operatiivsete analüüside jõustumise üle. Arvestades, et KG on suuremõõtmeline ja selle käsitlemine on keeruline, siis on oluline luua optimaalsemaid lahendusi otsinguteks ja teha võrdlusanalüüsi juba olemasoleva semantilise veebiga. Kuigi enamik lahendustest kasutavad ühekordseid päringuid, siis leidub ka selliseid, kus kasutatakse erinevaid. Sellisteks on mitmed programmi domeenid nagu Internet of Things, Social Media Analytics ja News Analytics, kusjuures tehtavad päringud on sagedased. Kasutades andmete avamiseks kulunud arvutusi, keskendume lahenduste leidmisel sellele, et optimeerida analüütilise SPARQL päringute jõustumist. Kirjandus on näidanud positiivseid tulemusi SQL töökoormuse optimeerimisel. Meie teadmiste kohaselt on semantilise veebi maastikul olnud pingutused minimaalsed. Käesolev dissertatsioon esitleb uut JSON-LD süntaksianalüsaatorit nimega Quarser. Tegemist on süntaksianalüsaatoriga, mida on kohandatud just sellist tüüpi päringutele. Quarse tugineb RDF-graafikule ja päringute tegemiseks

on kasutatud SPARQL. Saadud tulemusi on võrreldud LUBM-ga. Tulemused näitasid, et päringutele vastamiseks kulus 20% vähem aega kui muidu.

Keywords: JSON-LD, RKK graafid, SPARQL, Tagurpidi parsimine

CERCS: P170 - Arvutiteadus, arvutusmeetodid, süsteemid, juhtimine (automaatjuhtimisteooria)

CONTENTS

1 INTRODUCTION 1

2 BACKGROUND 2

2.1 The Semantic Web . . . 2

2.1.1 Linked Data . . . 2

2.2 Resource Description Framework . . . 3

2.2.1 IRI Node . . . 3

2.2.2 Literal node . . . 4

2.2.3 Blank node . . . 4

2.3 RDF Serialization . . . 5

2.3.1 Turtle Family . . . 6

2.3.2 RDF/XML . . . 7

2.3.3 JSON-LD . . . 8

2.4 SPARQL . . . 12

3 PROBLEM STATEMENT 15

3.1 Problem Context . . . 15

3.2 Research Questions . . . 16

3.2.1 From Macro to Meso . . . 17

3.2.2 From Meso to Micro . . . 19

4 DESIGN 20

4.1 Algorithm Design . . . 20

4.1.1 General Parser Design . . . 20

4.1.2 Quarser Design . . . 21

4.1.3 Data Structures . . . 21

4.1.4 Input Stream . . . 22

4.1.5 Tokenizer . . . 22

4.1.6 Graph population, BGP Matching and variable binding . . . 28

4.1.7 Post initial-binding phase . . . 29

4.2 Algorithm Implementation . . . 30

4.2.1 Conclusion . . . 31

5 EVALUATION 32

5.1 Benchmark . . . 32

5.1.1 Test Platform . . . 33

5.1.2 Scales . . . 33

5.1.3 Queries . . . 33

5.2 Experiments and Results . . . 34

5.2.1 Implementation . . . 34

5.2.2 Results . . . 36

5.2.3 Analysis and Discussion . . . 36

5.3 Conclusion . . . 39

6 CONCLUSION 40
A APPENDIX 42
A.1 Scalability of the query execution time for all 14 queries of LUBM 42
A.2 Execution times for experiments in Jena in-memory and Quarser 43
Bibliography 49

LIST OF FIGURES

Figure 1 Semantic Web Stack . . . 3
Figure 2 Example of a single triple . . . 4
Figure 3 Example of an RDF Graph . . . 5
Figure 4 RDF 1.1 group of formats recommended by the W3C. . . . 6
Figure 5 Example of RDF data and SPARQL query presented by [24] . . . 13
Figure 6 The shape of the Linked Open Data cloud . . . 15
Figure 7 Flow of execution of a query using JSON-LD as data input (in memory) in Apache Jena . . . 17
Figure 8 Percentage of time spent answering a query: Parsing and Query Processing . . . 18
Figure 9 Flow of a JSON input from a stream of bytes up to JSON objects . . . 20
Figure 10 Flow of a JSON-LD input from a stream of bytes up to an RDF Graph and Query execution, in Quarser . . . 21
Figure 11 Example of RDF Graph in construction during parsing . . . 28
Figure 12 Running example of parsing . . . 29
Figure 13 Data schema of the Lehigh University Benchmark (LUBM) studied in [21] . . . 32
Figure 14 Total time of parsing and query of Jena in-memory vs. Quarser in Flattened and Extended JSON-LD datasets for the scales 10, 25, 50, 75, and 100 . . . 37
Figure 15 Shape of the LUBM Query 2 . . . 38
Figure 16 Shape of the LUBM Query 6 . . . 38
Figure 17 Shape of the LUBM Query 4 . . . 39

1 INTRODUCTION

Computational systems today are more significant than ever. Whereas previously, one would architect their programs to run on a single system. It is common to design programs that share computation across multiple machines that communicate with each other in a coordinated fashion. Therefore, it is natural to ask why one might create from the latter perspective rather than the former. Big Data often comes in large graphs, where entities or nodes are interconnected with edges. Notable examples include social networks, the internet itself with hyperlinks forming edges, and biological networks such as protein-protein interaction networks.

This thesis examines the Semantic Web, which is a growing body of data. The Semantic Web has several standards. The Resource Description Framework (RDF) describes the data. RDF data consists of lists of triples, a subject, predicate, and object, where the subject and object are nodes, and the predicate is a labeled edge between them. JSON-LD is a modern, widely adopted standard to serialize RDF data on the Web, and SPARQL is the language for querying.

This thesis analyzes and evaluates a new graph-aware parsing algorithm called Quarser for accelerating the query-answering of SPARQL Queries. This is possible by pushing down the binding of variables into the parser of JSON-LD data. This idea is possible for query engines to improve the query answering by 1) skipping temporal variables during graph isomorphism and 2) selecting a near-optimal execution plan. The performance gain by Quarser (20% faster than Apache Jena) suggests new specialized parsers to be developed more often due to its time gain in query response, which ideal for analytical operations.

The remainder of this thesis is ordered as follows:

- In Chapter 2, we introduce the concepts of Linked Data, Semantic Web, RDF, SPARQL, JSON-LD to settle the ground to the contribution of the project.
- In Chapter 3, we describe the problem around parsing of serialized data, and we shed the light on our solution.
- In Chapter 4, we present the algorithm designed for optimizing SPARQL queries on the analytical domain.
- In Chapter 5, we present the most important results on query optimization, and analyze against current solutions from the literature.
- In Chapter 6, we conclude our work in this thesis, and we provide the perspective for the vision of the work.

2

BACKGROUND

In this chapter, we outline the preliminary information necessary to contextualize the remainder of this thesis for readers unfamiliar with the research of Semantic Web. Here, we start motivating the role of Linked Data, then we define the Resource Description Framework (RDF) and the query language SPARQL. To understand the terminology used in Chapters 3 and 4 we also define the most relevant RDF data serializations, and more importantly, the JSON-LD data serialization.

2.1 THE SEMANTIC WEB

The *Semantic Web* refers to a Web of data, readable and processable by machines. The main reason for its existence is to allow both humans and machines interact with data, and be able to infer meaning (and knowledge). To support this end, the working group that defines standards and recommendations is the *Web Wide Consortium* (W3C) .

The Semantic Web is represented as a stack [3] (Figure 1) that illustrates the basic core concepts and hierarchy of its components. The stack is composed of three main layers. The bottom layer contains well-known technologies that enable the basis for the Semantic Web i.e. Unicode, Internationalized Resource Identifier (IRI). The middle layer group technologies to build Semantic Web applications. i.e. Resource Description Framework (RDF), Web Ontology Language (OWL), SPARQL, JSON-LD, among others. Finally, the top layer constitute technologies that are not yet standardized. i.e. user interface, cryptography.

2.1.1 *Linked Data*

Linked Data is a term coined by Tim Bernes-Lee, inventor of the World Wide Web and director of W3C, in 2006. This term defines a set of principles for publishing and representing data in the Web so they can be interpreted in a meaningful way [4]. The principles defined for this end can be summarized as follows:

- Use Uniform Resource Identifiers (URIs) for naming things.
- Use Hypertext Transfer Protocol (HTTP) URIs so that people can look up those names.

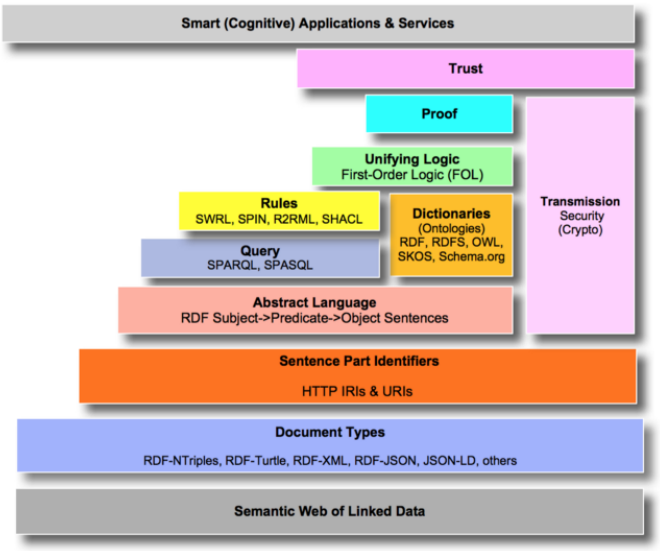


Figure 1: Semantic Web Stack

- When someone looks up a URI, provide useful information using standards: RDF, SPARQL, ...
- Include links to other URIs so that they can discover more things.

2.2 RESOURCE DESCRIPTION FRAMEWORK

The Resource Description Framework (RDF) is the standard data model to express structured information on the Web [13]. It describes resources by assigning properties and establishing relationships between them via URIs. This enables seamless communication across agents. RDF organizes information in triples (s, p, o), or statements, composed of a subject s, a predicate p, and an object o. A triple is the building block for defining resources of data. It postulates that the subject is the *thing* that the statement is about, the predicate refers to the property and the objects is its value.

The linking structure of RDF form a graph, where every node represent a resource, and edges represent the named relation between two resources.

An RDF resource can take one of the following forms:

2.2.1 *IRI Node*

IRI Node provides a global identifier for a resource. It is referenceable by others on the Web. An IRI can occur in any position in the triple (s, p, o).

Definition 2.1 (RDF Triple). *An RDF triple consists of three components:*

Subject $s \in I \cup B$

Predicate $p \in I$

Object $o \in I \cup L \cup B$

$$(s, p, o) \in (I \cup B) \times I \times (I \cup B \cup L).$$

where I is the set of IRIs. B is the set of blank nodes. L is the set of literals.

As a simple illustration [1], consider the following triple, made of IRI nodes, depicted in Figure 2:

```
<http://www.software-foo.org/home.html> <http://purl.org/dc/elements/1.1/creator> <http://www.software-foo.org/id/01>
```

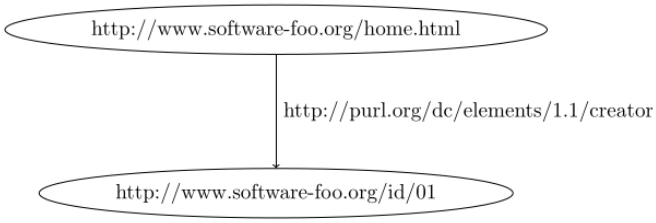


Figure 2: Example of a single triple

Definition 2.2 (RDF Graph)**.** *An RDF Graph is a finite set of RDF triples. If G is an RDF Graph, we use I(G), L(G), and B(G) to denote the set of IRIs, literals, and blank nodes. RDF Graphs are organized into datasets.*

2.2.2 *Literal node*

A *Literal node* represents values such as strings, numbers, and dates. It can be only part of an object (o) in a triple.

By default, a literal node is plain (untyped). However, it is possible to structure a literal assigning an additional information about the interpretation of a literal. This is possible by appending the information of the datatype to the literal value. For example, a plain literal node can be typed as an integer number when we add the suffix "`^^xsd:integer`".

2.2.3 *Blank node*

A *Blank node* is a unique identifier. Some serialization methods allow to create triples with the blank node identifier "_:". However, only the subjects and objects are allowed to be defined with blank node identifiers.

Definition 2.3 (Blank node)**.** *Blank nodes are disjoint from IRIs and literals. RDF makes no reference to any internal structure of blank nodes.*

[1] We follow the notation of a "full triple" with the angle brackets "<>"

Definition 2.4 (RDF Dataset). *An RDF Dataset is a collection of RDF graphs. It contains one RDF graph by default, which is either uniquely named or not. An RDF Dataset hols one or more named graphs:*

$$\{g_0, (u_1, g_1), (u_2, g_2), ...(u_n, g_n)\}$$

Figure 3 shows a graph representation of three RDF triple statements shown in Listing 1. Two objects are literals, and one is an IRI. As we observe, a graph is written as a set of triples that represent connection between subject, and objects through predicates [13].

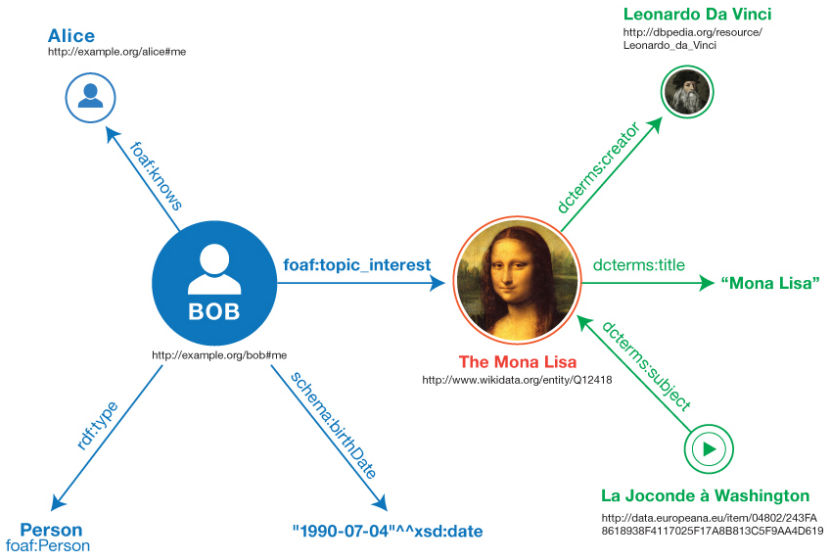


Figure 3: Example of an RDF Graph

```
<http://example.org/bob#me> <http://www.w3.org/1999/02/22-rdf-syntax-ns#type> <http://xmlns.com/foaf/0.1/Person> .
<http://example.org/bob#me> <http://xmlns.com/foaf/0.1/knows> <http://example.org/alice#me> .
<http://example.org/bob#me> <http://schema.org/birthDate> "1990-07-04"^^<http://www.w3.org/2001/XMLSchema#date> .
<http://example.org/bob#me> <http://xmlns.com/foaf/0.1/topic_interest> <http://www.wikidata.org/entity/Q12418> .
<http://www.wikidata.org/entity/Q12418> <http://purl.org/dc/terms/title> "Mona Lisa" .
<http://www.wikidata.org/entity/Q12418> <http://purl.org/dc/terms/creator> <http://dbpedia.org/resource/Leonardo_da_Vinci> .
<http://data.europeana.eu/item/04802/243FA8618938F4117025F17A8B813C5F9AA4D619> <http://purl.org/dc/terms/subject> <http://www.wikidata.org/entity/Q12418> .
```

Listing 1: Example of an RDF graph made of 7 triples

2.3 RDF SERIALIZATION

The W3C defines multiple syntaxes where triple statements can be serialized. An RDF Dataset or RDF Graph is encoded to a particular syntax to enable exchange and storage between systems. Every serialization should be logically equivalent, leading to the same compositional graph. However, they also introduce different complexities in their deserialization. We present a compacted list of RDF serialization formats in Table 1. where every format support particular

	RDF Dataset	RDF Graph	Triple	Allow blank node
Turtle and TriG	NO	YES	YES	YES
N-Triples/N-Quads	YES	YES	YES	YES
RDF/XML	YES	YES	YES	YES
RDFa	NO	NO	YES	NO
JSON-LD	YES	YES	YES	YES

Table 1: RDF Serialization formats

semantics, enable compatibility for RDF graphs, RDF datasets, support for blank nodes, and triple statements.

The semantics of RDF requires resources to be identified as IRIs to the RDF terms. For this, each syntax represents IRI definitions for non-literal nodes in an RDF graph. The example of figure 3 is shown in the listing 1 in turtle format. Notice that the terms `wd:`, `dcterms:`, `xsd:`, and `foaf:` compact the naming of resouces; these are known as prefixes. They avoid the verbosity of resource naming. The prefix notion is present mainly in document formats such as Turtle, RDF/XML, and JSON-LD.

A previous work by [35] presented a brief interplay of the most used RDF serialization formats (Figure 4), and discuss how the different representations impact the optimization of SPARQL queries. The analysis of this study is tied to the constraints of the Table 1, opening the chance to compare each RDF serialization and evaluate their strengths and weaknesses.

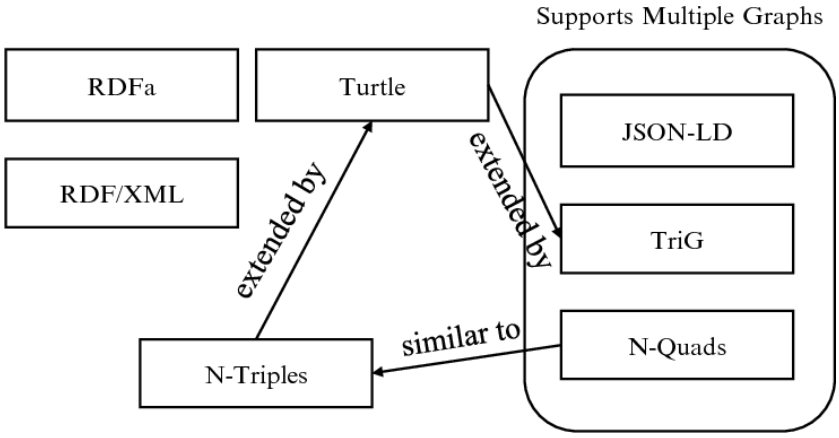


Figure 4: RDF 1.1 group of formats recommended by the W3C.

2.3.1 *Turtle Family*

Turtle [2] is a textual syntax for RDF Graphs. It is caracterized by its compact representation of RDF triples. The formats N-Triples, Turtle, TriG, N-Quads belong to the Turtle family, where N-Triples is the standard line-based RDF

syntax where each triple is written without abbreviation such as prefix. Each IRI is written between angle brackets, and the support for blank nodes is done by prefixing "_:". An example of N-Triples is given in Listing 2.

```
<http://example/ntriple/subj> <http://example/pred1> "object"@en .
<http://example/ntriple/subj> <http://example/pred2> _:obj .
_:obj <http://example/property1> <http://example/something> .
_:obj <http://example/property2> "A short text here." .
```

Listing 2: Example of RDF in N-Triples

Turtle is an extension of N-Triples, where each statement is also a sequence of subject, predicate and object separated by spaces and ended with a dot character. However, turtle supports abbreviation via prefix to cope with long IRIs. An example is shown in Listing 3.

```
@prefix ex: <http://example.org/> .
@prefix xsd: <http://www.w3.org/2001/XMLSchema/> .
ex:book1 ex:authorName "Mr Doe" .
ex:book1 ex:title "Un livre"@fr .
ex:book1 ex:title "A book"@en .
ex:book1 ex:date "2016-09-15"^^xsd:date .
```

Listing 3: Example of RDF in Turtle

Turtle also supports compaction of triples when the subject is referenced by several predicates. This is achieved with the semicolon character ";", which indicates the start of a list expression. For instance, consider the example in Listing 4 that shortens the representation of the same subject in 3.

```
@prefix ex: <http://example.org/> .
@prefix xsd: <http://www.w3.org/2001/XMLSchema/> .
ex:book1 ex:authorName "Mr Doe" ;
         ex:date "2016-09-15"^^xsd:date ;
         ex:title "Un livre"@fr ,
"A book"@en .
```

Listing 4: Example of RDF in Turtle

2.3.2 *RDF/XML*

RDF/XML is the XML syntax for writing RDF graphs. It was the only official syntax supported when the standard came out, and currently, is the normative syntax for writing RDF [14]. An example in RDF/XML is given in Listing 5.

```
<?xmlversion="1.0"?>
<rdf:RDFxmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#" xmlns:ex="http://example.org/">
        <rdf:Descriptionrdf:about="http://www.example.org/book1">
                <ex:title>Abook</ex:title>
        </rdf:Description>
        <rdf:Descriptionrdf:about="http://www.example.org/book1">
                <ex:daterdf:datatype="http://www.w3.org/2001/XMLSchema#date">
                2016-09-15
                </ex:date>
        </rdf:Description>
        <rdf:Descriptionrdf:about="http://www.example.org/book1">
                <ex:authorNamerdf:resource="http://example.org/id/01"/>
        </rdf:Description>
</rdf:RDF>
```

Listing 5: Example of RDF in RDF/XML

2.3.3 *JSON-LD*

JSON-LD is a serialization format of RDF graphs and datasets in JSON [2]. The convenience of the syntax is that an original JSON document can be ported into a RDF dataset with minimal changes. JSON-LD is usually embedded within HTML documents. This is used as metadata for a resource when search engines crawl the web. JSON-LD serializes directed graphs, meaning that every property in the JSON document points from a node to another node or value [30]. JSON-LD enables existing JSON data to be interpreted as linked data with the use of a JSON-LD context. The JSON-LD serialization format is relatively new, and has been supported by big organizations over the years e.g. Google [8]. The current version, 1.1, extended the possibilities to write in different representations, with the use of standarized keywords, and processing algorithms. We cover these points after giving a definition of the JSON format.

JSON

JSON is a lightweight, highly popular text-based data format. It is widely used for its convenient schema-free data exchange. The popularity of JSON is mainly due to its simple, flexible, and expressive power that has engaged the interest for running analytical queries on JSON data [26]. In this regard, many data analytics engines natively support JSON data.

Definition 2.5 (JSON Object)**.** *A JSON Object begins with a left brace ("{") and ends with a right brace ("}"), and it contains zero or more key/value pairs, separated by commas (","). Keys must be a string, and values must be a JSON data type.*

Definition 2.6 (JSON Array)**.** *The JSON Array is an ordered collection surrounded by square brackets ("[" and "]") that contain zero or more values, separated by commas. Array values must be a JSON data type.*

[2] https://www.json.org/json-en.html

Definition 2.7 (JSON Document). *A JSON Document is a collection of key and value pairs where each key identifies each value. A value can be:*

- *Atomic (string, number, boolean, null)*
- *An array: Ordered list of values*
- *Another JSON object*

The following is the recursive definition of the JSON format.

$$
\begin{aligned}
OBJECT &= \{STRING : VALUE, ..., STRING : VALUE\} \\
ARRAY &= [VALUE, ..., VALUE] \\
VALUE &= OBJECT | ARRAY | STRING | NUMBER | \texttt{true} | \texttt{false} | \texttt{null}
\end{aligned}
$$

A "STRING" is a sequence of zero or more characters wrapped in quotes (""). A "NUMBER" is a regular literal integer or decimal number. The null literal and the boolean literals true and false are represented in lowercase letters. A "VALUE" can be a string, a number, boolean (true, false), null, an object, or an array. The "key/value" pair is called a field of the object. Each key is followed by a single colon (":"), separating the key from its corresponding value. The standard RFC-7159 [7] restricts the uniqueness of the key in an object. In this thesis, we will take this assumption as well.

As mentioned earlier, JSON-LD uses JSON objects to describe resources and its relationships. Based on the RDF data model, JSON-LD allows a rich set of resources that are defined below.

JSON-LD Keywords

JSON-LD 1.1 defines a set of reserved keywords that are part of the serialization. Each keyword may appear within different JSON-LD structures [30]. The complete list of keywords supported in this version are the following: `@context`, `@list`, `@set`, `@language`, `@index`, `@id`, `@graph`, `@type`, `@base`, `@nest`, `@none`, `@prefix`, `@protected`, `@propagate`. We focus on the keywords that are relevant to the construction of the RDF graphs, which is the interest of the following sections.

JSON-LD Context

The JSON-LD Context is identified by the keyword "@context" in the first level of the document. The Context of a document is defined in the document as a JSON object and it is extensively well defined in [30]. The Context may also be an IRI that points to a remote context definition. It defines a mapping that is used to expand terms in the document to exact definitions with IRIs. In this way,

it avoids verbosity of IRI definitions. In addition, the specification of JSON-LD 1.1 declares a context only for the Compacted and Flattened representations. In this specification, JSON-LD also allows us to define multiple context objects, but in the scope of this thesis, we will assume that zero or one Context is defined in the JSON-LD document. The use of Context allows documents to use different names and be fully interoperable when the name of the property is expanded to the same IRI. Listing 6 shows how the same property value for two different property names can be represented. This allows the composition of JSON-LD to be programmatically friendly since it can take values from various sources.

```
{
  "@context": {
    "surname": "http://xmlns.com/foaf/0.1/name",
    "lastName": "http://xmlns.com/foaf/0.1/name"
  }
}
```

Listing 6: Example of a JSON-LD Context defining two keys for the same IRI

The "@id" keyword is used to identify objects in the document. Its value represents an IRI that can be absolute or relative to the document location. Objects that represent a node may or may not have an "@id" value. Its absence makes the node object have a blank node. The blank node identifiers are local to the document, and it is prefixed with the characters "_:".

The keyword "@value" contains the literal value related to the properties of a node object, in which case it is called "record".

"@type" is used to set the data type of a node object or a value object. Usually, the type of a node can be inferred based on the property names but specifying them helps process documents using the framing algorithm. It is also possible to identify multiple types for a single node object using the JSON array type. In any case, the value of the "@type" keyword should be an IRI.

The JSON-LD specification describes three forms that it is possible to apply for shaping the data without changing the meaning. These forms are called Expanded, Compacted, Flattened, and Framed. JSON-LD Framing is a particular form used to shape the data using an example frame to force the desired structure on a JSON-LD document to match the example. This form is not discussed in the scope of the thesis.

Definition 2.8 (JSON-LD Node Object)**.** *A Node Object is a JSON object representing zero or more properties of a node in the graph.*

JSON-LD Compacted Document Form

Compaction consists of mapping IRIs appearing in a JSON-LD context object to short terms, compacting their meaning. Compacted documents are smaller in

size and are also easier to read for humans. A compacted form of example 10, given a context defined in 7 would result in the example 8.

```
{
  "@context": {
    "name": "http://xmlns.com/foaf/0.1/name",
    "homepage": {
      "@id": "http://xmlns.com/foaf/0.1/homepage",
      "@type": "@id"
    }
  }
}
```

Listing 7: Example of a JSON-LD Context

```
{
  "@context": {
      "name": "http://xmlns.com/foaf/0.1/name",
      "homepage": {
        "@id": "http://xmlns.com/foaf/0.1/homepage",
        "@type": "@id"
      }
  },
  "name": "Manu Sporny",
  "homepage": "http://manu.sporny.org/"
}
```

Listing 8: Example of a Compacted JSON-LD

JSON-LD Flattened Document Form

An array of node objects represents the Flattened form without nesting. The flattening algorithm ensures that the shape of the output is deterministic. This algorithm replaces every property of a node that has as a value an IRI reference. It replaces all node properties in a JSON object and label all blank nodes with blank node identifiers. An example of a flattened JSON is shown in Listing 9.

JSON-LD Expanded Document Form

The Expanded form is the outcome of taking a JSON-LD document and applying a "@context" such that all IRIs, types, and values are expanded to IRIs, blank nodes, or keywords so that the "@context" can be removed. The resulting shape of the expanded document is expressed in arrays. The JSON-LD Processing Algorithms and API specification define a method for expanding a JSON-LD document. Example 10 shows the result of the expansion of compacted data input. In the expanded version, the context object is removed, and every term is represented as a full IRI.

```
{
  "@context": {
    "name": "http://xmlns.com/foaf/0.1/name",
    "knows": "http://xmlns.com/foaf/0.1/knows"
  },
  "@graph": [{
    "@id": "http://me.markus-lanthaler.com/",
    "name": "Markus Lanthaler",
    "knows": [
      { "@id": "http://manu.sporny.org/about#manu" },
      { "@id": "_:b0" }
    ]
  }, {
    "@id": "http://manu.sporny.org/about#manu",
    "name": "Manu Sporny"
  }, {
    "@id": "_:b0",
    "name": "Dave Longley"
  }]
}
```

Listing 9: Example of a Flattened JSON-LD

```
[
  {
    "http://xmlns.com/foaf/0.1/name": [
      { "@value": "Manu Sporny" }
    ],
    "http://xmlns.com/foaf/0.1/homepage": [
      { "@id": "http://manu.sporny.org/" }
    ]
  }
]
```

Listing 10: Example of an Expanded JSON-LD

2.4 SPARQL

SPARQL is the standard query language for RDF data recommended by the W3C [3]. The syntax of SPARQL resembles SQL. It is built upon similar operators such as "SELECT," which specifies the variables that appear in the query result set; "FROM" specifies the RDF dataset. SPARQL 1.1 [12], which originated in 2013, is the latest stable version of the language. It features query language operations such as aggregates, subqueries, negation, property paths, and various new functions and operators. For a complete description and formalization of the SPARQL language, readers may refer to the SPARQL 1.1 specification document and [12] for an exhaustive formal definition of its semantics. A SPARQL

[3] https://www.w3.org/TR/sparql11-query/

query consists of RDF triple patterns modeled as a directed graph called Basic Graph Pattern (BGP). The pattern is matched against an RDF graph dataset, in which variables within patterns are bound to values to give a solution for the query. Figure 5 shows an example of a RDF graph, a SPARQL query, and the answer for the query.

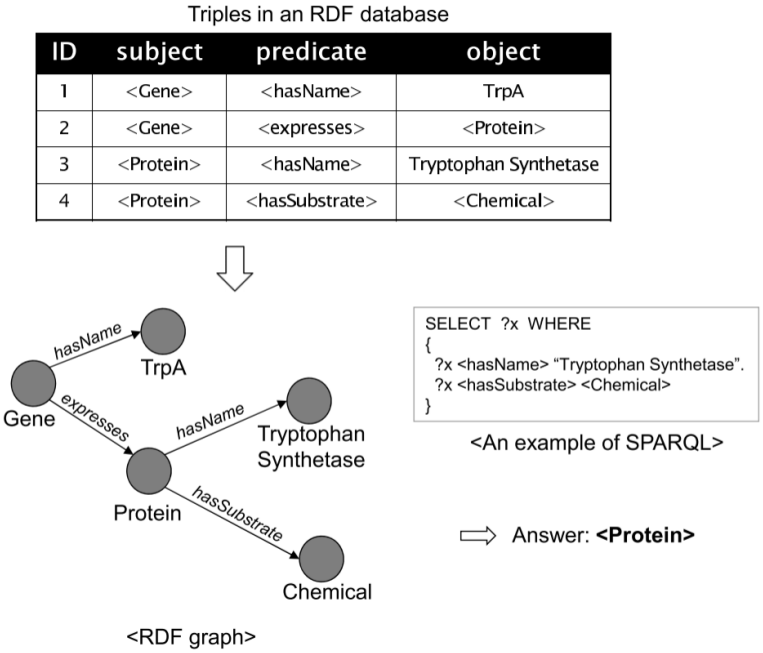


Figure 5: Example of RDF data and SPARQL query presented by [24]

.

Definition 2.9 (Query Variable)**.** *A query variable is a member of an infinite set disjoint from the set of RDF terms. The expression is given in the form "?" or "$," indicating that the variable is unbounded. These symbols are not part of the variable name (?x and $x refer to the same variable). We define V as the set of variable names.*

Definition 2.10 (Triple Pattern)**.** *This is similar to an RDF triple, except that elements of the triple pattern (s, p, o) can be bound or replaced by a Query Variable and appear in multiple triple patterns. It consists of three components.*

- *Subject* $s \in I \cup L \cup V$
- *Predicate* $p \in I \cup V$
- *Object* $o \in I \cup L \cup V$

Set of triple patterns are fundamental to SPARQL queries as they specify the access to RDF data.

Definition 2.11 (Basic Graph Pattern)**.** *A Basic Graph Pattern (BGP) is a set of Triple Patterns. The empty graph pattern is a BGP which is the empty set.*

BGPs can include compound patterns defined by algebraic operators [5]. In addition, SPARQL includes the forms "SELECT" that returns all, or a subset of, the variables bound in a query pattern match. "CONSTRUCT" returns an RDF graph constructed by substituting variables in a set of triple templates. "ASK" returns a boolean that indicates whether or not a query pattern has at least one solution. Finally, "DESCRIBE" returns an RDF graph that describes resources found. In the context of this thesis, we will only work with "SELECT" queries.

Definition 2.12 (Group Graph Pattern)**.** *A Group Graph Pattern is a set of graph patterns.*

Definition 2.13. *A solution mapping μ is a partial function.*

$$\mu : V \to I \cup B \cup L$$

A solution mapping μ is defined over a domain $dom(\mu) \subseteq V$, and $\mu(x)$ indicates the application of μ to a variable x.

Let Ω be a multiset of solution mappings. A SPARQL query solution can be represented as a set of solution mappings. Given an RDF graph, each RDF triple in the graph is said to be bind to a variable. We follow the definitions developed by [27] to show the semantics of graph pattern expressions.

A graph pattern expression takes an RDF Dataset D as input and returns a set of solution mappings Ω. The evaluation function $[\![t]\!]_D$ of a Dataset D of triple pattern t and a graph pattern $P_i \times P_j$ is defined, as in [5], as shown in the following formula:

$$[\![t]\!]_D = \{\mu | dom(\mu) = var(t) \in D\}$$
$$[\![P_1 \ AND \ P_2]\!] = \Omega_1 \bowtie \Omega_2 = \{\mu_1 \cup \mu_2 | \mu_1 \in \Omega_1 \wedge \mu_2 \in \Omega_2 \mu_1 \backsim \mu_2\}$$
$$[\![P_1 \ UNION \ P_2]\!] = \Omega_1 \cup \Omega_2 = \{\mu_1 \cup \mu_2 | \mu_1 \in \Omega_1 \vee \mu_2 \in \Omega_2 \mu_1 \backsim \mu_2\}$$
$$[\![P_1 \ OPTIONAL \ P_2]\!] = \Omega_1 ⟕ \Omega_2 = (\Omega_1 \bowtie \Omega_2) \cup (\Omega_1 / \ \Omega_2)$$

$$\Omega_1 / \ \Omega_2 = \{\mu | \mu \in \Omega_1 \wedge \nexists \mu' \in \Omega_2 \mu \backsim \mu'\}$$

3

PROBLEM STATEMENT

In this chapter, we formulate the problem that this thesis addresses. For its fulfillment, we contextualize the problem and define the research questions for the Macro, Meso, and Micro levels.

3.1 PROBLEM CONTEXT

Knowledge Graphs (KGs) have become the prefered technology for representing, sharing, and using knowledge in applications [34]. For example, the Linked Open Data cloud (LOD) [1], depicted in Figure 6, is now (as of 2021–05–01) over 50 billion edges [19]. LOD provides a loosely coupled collection of data, information, and knowledge accessible by any machine or human, backed by the abstraction layer provided by the Web. LOD allows both basic and sophisticated lookup-oriented access using query languages like SPARQL Query Language and SQL.

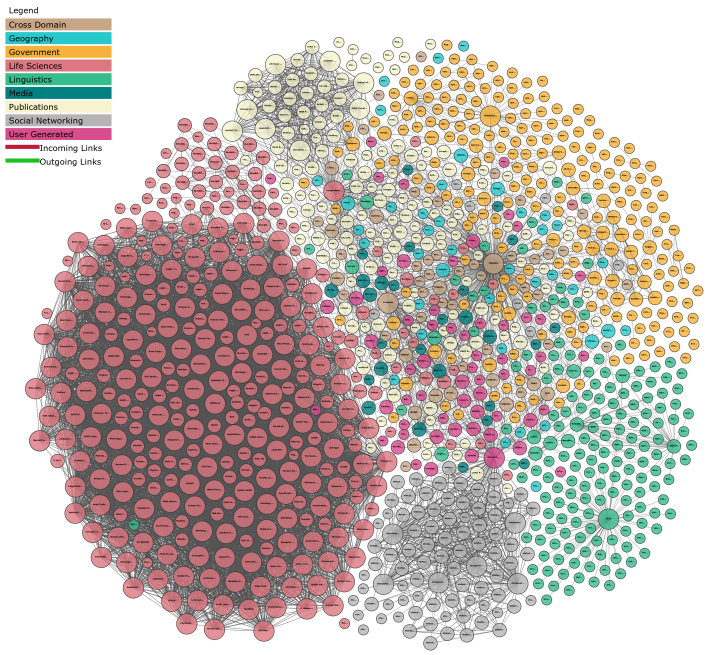


Figure 6: The shape of the Linked Open Data cloud

KGs are relevant for several application domains, e.g. healthcare, cultural heritage, chatbots [33]. They enable a number of analytical tasks, e.g., IoT

[1] http://lod-cloud.net/

and news analytics. Despite this relevance, analysis is still a challenge due to data heterogeneity. Indeed, KGs usually combine different data sources. Semantic Web technologies provide tools and methods to address this challenge. Technologies and standards like SPARQL and RDF enable data sharing and integration. Nevertheless, analytical workloads still need to be optimized, due to the constant growing volume of data that must be processed.

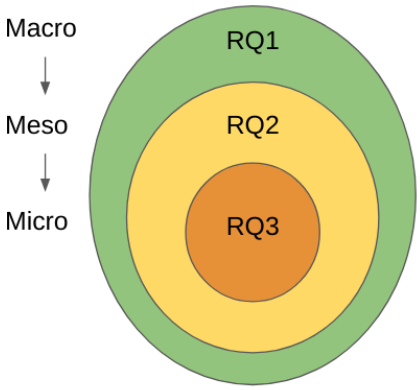
Existing approaches focus on standard query optimization techniques, e.g., indexing, caching, and operator reordering. However, in a number of application domains, the reiteration of the analytical tasks pave the way to new opportunities. In particular, when queries of interest are known upfront the data ingestion, some improvements are possible.

The area of RDF stream processing focuses on consuming unbounded streams of data by using specialized engines that enable the execution of *continuous* queries. On the other hand, to the best of our knowledge, little work has been done in the case where queries are *recurrent* but not *continuous* (i.e. they are not relevant under continuous semantics).

That is why, we want to center our study in the case of query optimization. This choice is also motivated by the research done in the area of real-time analytics. The general problem is to provide the community new ways to optimize the query-answering for RDF data and standard SPARQL queries.

3.2 RESEARCH QUESTIONS

To formalize the research questions of the thesis, we use the Macro-Meso-Micro framework [22] that allows formulating research questions on three levels of analysis, starting from a broad view of the problem and narrowing down to the level of research contribution.



Macro is the broadest unit of analysis. It captures the vision of the work but lacks specificity. We use this level to formulate our research question:

- **RQ1:** How to improve the execution time of analytical SPARQL queries?

Meso connects the Macro and Micro levels of specificity. In our case:

- **RQ2:** Can we share the same computation resources for parsing and answering recurrent SPARQL queries?.

Micro is the level where we want to position our research. It is the place where we can answer the research question:

- **RQ3:** How can we solve part of a SPARQL query on the parsing process over JSON-LD datasets?.

3.2.1 *From Macro to Meso*

The RQ1 is too broad to be answered, we need a context for optimization. Additionally, we need to define the environment of execution of analytical SPARQL queries. Therefore, we need to break down the requirement by adding an assumption:

- **A1.** Execution of recurrent queries. The solution runs over repetitive queries.

Knowledge Graphs like DBPedia [25], and Wikidata [32] are useful resources to derive knowledge. Their workload requires intensive computation of analytical operations. Hence, in order to cope with these sort of challenges, it is natural to implement strategies to optimize the execution of queries.

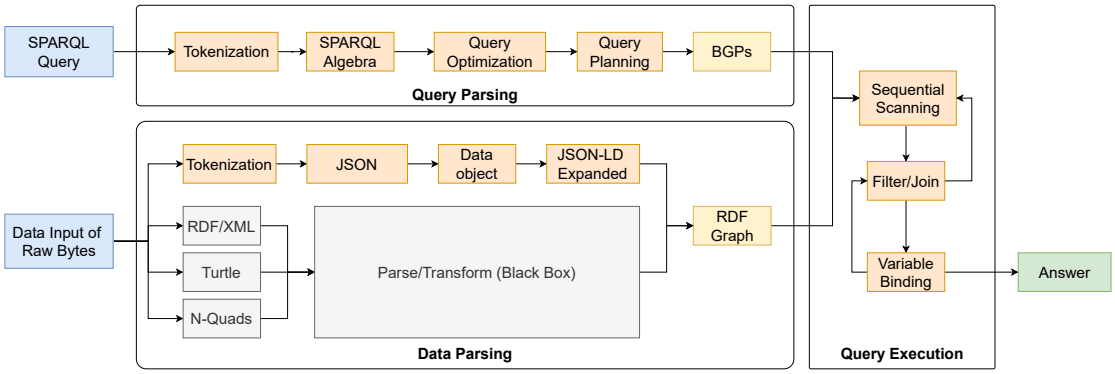


Figure 7: Flow of execution of a query using JSON-LD as data input (in memory) in Apache Jena

Multi-query optimization is a classical problem in the context of RDF and SPARQL [23]. While they focus on reorganizing the execution plan of query statements to ease the load of computation, others focus on materializing results and indexing triple stores [28]. Consequently, the subject of query optimization is well explored and the majority of techniques are based on identifying and reducing the work through prunning the search space. This insight leads to the question: Is there another way to prune the search space of SPARQL queries?.

In order to answer this question consider the flow of execution of a SPARQL query in a well-known triple store, Apache Jena in Figure 7. Given a SPARQL query, and a serialized RDF dataset, Apache Jena performs two important phases before trying to answer the query: parse the query and parse the data input. On one hand, and regardless of the four steps involved in the process (Tokenization, Construction of the SPARQL algebra, Query optimization, Query planning), the parsing of a query is an unexpensive operation because the complexity of creation is bounded by the size of the input query, which is low. On the other hand, the parsing of data is expensive. It is a complex pipeline of operations that move the bytes of data from disk to objects in memory. We observe that, no matter which data serialization format is used as input, the output is an RDF Graph. The cost of scanning and creating objects in memory is increased by the repetitive scanning that the query answering the triple stores executes. In analytical workloads has been shown that, for solving queries, the total cost of a query takes is dominated by parsing the raw JSON data by an 80% [26] (refer to Figure 8).

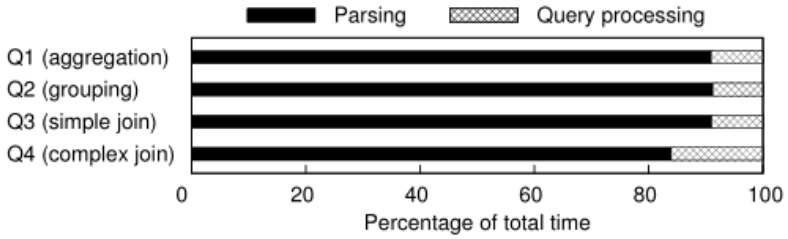


Figure 8: Percentage of time spent answering a query: Parsing and Query Processing

As shown, Figure 7 exposes the flaw of a particular native RDF triple store. The flow that data has to go through is long and reduntant. Hence, we are facing a clear flaw that can be addressed answering the RQ2: Can we share the same computation resources for parsing and answering recurrent SPARQL queries?.

Moreover, effort has been done to enable Stream processing to RDF, leading to the formation of the standard RDF Stream Processing (RSP) [2]. Existing RDF Stream engines such as C-SPARQL [18] allow processing data using a streaming model. They support operations that handle data with temporal information by enabling a continuous evaluation where the query execution happens at multiple points. Nevertheless, the access and querying to data in Machine to Machine communication environments is less unpredictable because devices run pre-defined queries for specific information, repeatedly [10]. Therefore, knowing the query of execution in advance opens up the possibility to understand the semantics of the data, and the query to provide a solution that reduces redundant scans over the data in the parsing stage and query execution, and in this way, share the same computational space on a

[2] https://www.w3.org/community/rsp/

single execution without neglecting performance. The assumption A1 fits in this scope.

3.2.2 *From Meso to Micro*

RQ3 is still broad to be answered. Analytical enviroments need to work under fast and convenient data formats. The ideal environment for this to happen is in a Machine to Machine translation. We define a last assumption for this requirement.

- **A2.** Machine to Machine communication. The solution enables automated communication between entities.

Many analytical workloads process data stored in RDF formats, e.g., RDF/XML, Turtle, or binary formats such as HDT [3]. The assumption A2 states that communication between machines (or devices) is automated, and data is generated for their consumption. Data is generated and processed in machine readable formats that provide fast explorations, e.g., XML and JSON-LD [31]. JSON-LD is a relatively new supported language for representing Linked Data. It is backed by giant companies that rely on knowledge graphs, such as Google and Microsoft [4]. JSON-LD also is a convenient format that modern approaches to JSON parsing can leverage. Therefore, now we can formulate the RQ3, which is the topic of this thesis: *How can we solve part of a SPARQL query on the parsing process over JSON-LD datasets?*. After our analysis of research questions, now we have requirements that we can achieve throughout this thesis.

- **R1.** Full support of the JSON-LD Data Format. The solution supports the JSON-LD format as RDF data serialization (Compacted, Expanded, Flattened).
- **R2.** Speed up queries that are known upfront.
- **R3.** Subsequent execution of queries. The solution should run arbitrary queries after reading data for the first time.

On the following chapter, we will address each of these requirements for the fulfillment of the research questions presented above.

[3] https://www.w3.org/Submission/HDT

[4] http://www.seoskeptic.com/open-letter-bing-regarding-json-ld/

4 DESIGN

This chapter provides an overview of Quarser. The following two sections describe details of the data structures and algorithm for the parsing/binding algorithm that evaluates SPARQL queries over JSON-LD data.

First, we describe the algorithm for scanning input raw data as a sequence of bytes. Then we define the process of triple pattern matching against triples of data. Later we provide the design of Quarser on top of Jackson, a high-performance JSON tokenizer, and Apache Jena API, a Java framework for building Semantic Web applications.

4.1 ALGORITHM DESIGN

The basic idea of Quarser is to push down both projections and filters into the parser of JSON-LD RDF datasets. The key mechanism for making this possible is to leverage the tree (directed graph) structure of the JSON format to construct triples of data while matching against triple patterns on-the-fly. As prompted, this mechanism leverages the scanning phase of raw bytes while the JSON tokenization process is performing the usual parsing routine.

4.1.1 *General Parser Design*

The normal way a parser works is using Finite State Machines (FSM) [15]. Most parsers work by passing through the input bytes once, traversing the structure top-down and doing the decoding one character a time [11]. This mechanism is broken down into two steps. The first step separates the data into defined tokens, where a token is one or more bytes of characters matching a pattern. The second step interprets the tokens and constructs in-memory JSON objects based on matched tokens. Figure 9 illustrates this idea, showing the flow of a raw input of bytes in JSON format through the parsing process where the gray color indicates the transformation steps where the data is not fully usable as a whole, and the orange shows the phase where the objects are in memory, ready to be manipulated.

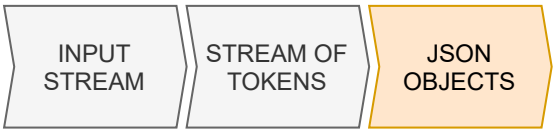


Figure 9: Flow of a JSON input from a stream of bytes up to JSON objects

4.1.2 *Quarser Design*

The two-step approach mentioned before is used in the design of Quarser. However, we leverage the tokenization step in order to perform operations related to the data model and query execution. Consider Figure 10, that depicts the flow of the parsing algorithm. In this part, we give an overview of the the algorithm, in the following subsection define the data structures and algorithms, and in the subsequent subsections we explain them in detail.

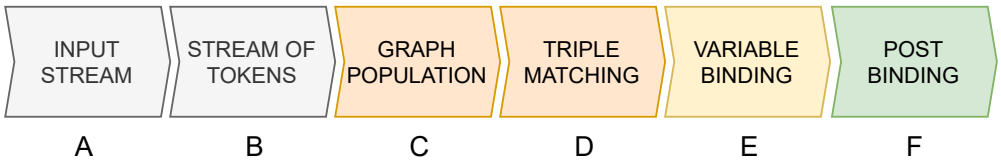


Figure 10: Flow of a JSON-LD input from a stream of bytes up to an RDF Graph and Query execution, in Quarser

Step *A* reads the JSON-LD input as a buffered stream. The parser reads the data into a buffer. This steps demands sequential access to the input data, so we need to load it into a fixed-size buffer of 2048 bytes. The buffer has to be in memory to be accessed by the tokenizer. In *B*, the tokenizer converts the input stream into tokens. Now it is convenient for the parser to interpret them and determine the type of triple currently building. The parser knows if the current token is part of the subject, predicate, or object. If the algorithm detects the current token is an object node, it creates the new triple and adds it to the model. In this step, the token type, the start and end position of the index are kept in memory for its later iterations. Next, in *C* when a new triple is captured, after traversing the JSON tree structure, a *triple constructor* routine adds a newly found triple to the graph model. The moment a new triple is found, the algorithm performs a two-step operation: it creates a new triple and adds it to the model (*C*), and matches it against the triple pattern (*D*). A *Binding* structure holds the bindings in a set of variable bindings after a successful matching (*E*). Finally in *F*, after completing the navigation of the input buffer, we use the result set of bindings to build the BGP upon the initial bindings. At this point, we can parallelize the code and make the bindings converge, although this has proven to be difficult since the canonical name of the blank nodes has to be consistent [20].

4.1.3 *Data Structures*

Quarser relies on the following list of data structures to leverage the capacity of our JSON-LD parser.

Definition 4.1 (Array)**.** *Array is a data structure consisting of a collection of elements (values or variables), each identified by at least one array index or key.*

Definition 4.2 (Stack)**.** *Stack is a LIFO (last in, first out) abstract data type that serves as a collection of elements.*

- Stack of *Predicates*, where *Predicate* is a *String* type.
- Stack of *Subjects*, where *Subject* is a *String* type.
- Stack of *Operations* where *Operation* is an enumerator with the entries: `START_OBJECT`, and `START_ARRAY`.
- *Binding*, a Class that associates variables with values.
- Array of *Binding*s.
- *Query*, a Class that stores the tree structure of a query BGP.

4.1.4 *Input Stream*

An Input Stream is a construct that process incoming data in small chunks in memory. The parser contains indices into the input buffer, which helps to keep track of the offset of the file while it is being read. In this way, it knows how to resume and apply the subsequent operations.

To enable sequential access to the data, we need to load it into memory, filling the buffer of a size defined by the parser. Loading the whole data into memory at once would require a significant amount of memory to be available, so pulling the whole file becomes unfeasible, especially on devices of constrained resources. Since every data source can be fully parsed by a single buffer efficiently and independently from other data sources, we don't need to put the whole file into memory simultaneously.

In Java, we implement this behavior with "InputByteStream," which, given a file descriptor, opens a new stream connection that consumes till the end of the file by filling up the buffer. The implementation of the Jackson parser receives this stream as input by invoking the "createParser" method. The buffer size and block size are kept the same since the compaction and filling operations requires them to be in constant exchange. Internal details of how the JSON parser deals with memory will not be discussed in this scope.

Every triple pattern that composes a BGP can be executed in this routine to process the same file in parallel to answer different parts of a query and then converge to the same result.

4.1.5 *Tokenizer*

The tokenize breaks down the data buffer into tokens. The actual parsing is performed by two operations over the iteration method, *next*. First, it returns

an identifier of the type of a token pattern. Then, it skips the current token and moves the cursor of the parser to the next token.

The identifier returned by next is an identifier of the current pattern. The list of tokens that are relevant to us are the following:

- `FIELD_NAME`
- `VALUE_STRING`
- `ARRAY_START`
- `ARRAY_END`
- `OBJECT_START`
- `OBJECT_END`

Value tokens

VALUE_STRING comes when a String token is found in the context of a value. The context can be a field value or array element. The value of a key indicates the leave of the current branch of the tree. For this reason, this is the step where the parser determines whether the current node is a literal node or a named node. First, if an active context is present, it tries to expand the given term to an IRI value. If a context is not found, a new literal node is created. Finally, if the context resolves a term or the value matches an IRI pattern, then a named node takes the value of the object of the triple. At this step, we have the three elements of a triple pattern. We construct the triple and perform the steps defined in section 4.1.6.

FIELD_NAME is returned when a String token is encountered as a field name. The full name of the field is captured by reading the next bytes up to the next token. From this point, we can determine whether the field name is a JSON-LD keyword or a predicate name. In the case of a keyword, the parsing becomes predictable, and we can take advantage of it to parse multiple tokens on the same iteration.

- The `@context` keyword, as defined in section 2, is used to describe short-hand names, called terms, that are used on the JSON-LD dataset. The keys defined in the context object help the compaction of the resource naming of nodes. The parser skips to the next token and parse the content of the raw object as a JSON object, which is used to create a JSON-LD Active Context that resolves terms while the parsing process is running.
- The `@id` keyword is used to uniquely identify node objects described in the document with IRIs or blank node identifiers. A node reference is a node object containing only the `@id` property, representing a reference

to a node object found elsewhere in the document. The cursor jumps to the next token to capture the value of the field (by invoking next once again). Then asks if the current value comes as an identifier of an inner node object or not. If so, then it means that the id has branched to a new node object so that we can perform a population, match, and binding, described in section 4.1.6.

- The `@type` keyword is used to set the type of a node or the data type of a typed value. A type can be applied to both node objects and value objects. A node can be assigned more than one type by using an array. Since the type is uniquely identified with an IRI, we push it to the stack as a predicate of the next object node.

- The `@value` keyword is used to specify the data associated with a particular property in the graph. Its presence assumes that the current scope of the object is a value object, which is used to explicitly associate a type with a value to create a typed value specified by the `@type` keyword. A value object is defined not to contain any other properties that expand to an IRI or keyword. The value of `@value` is captured by jumping to the next token and parsing the value as a string, number, true or false value. The presence of this keyword indicates the presence of a node leaf, for which it is necessary to form the triple.

- Finally, `@graph` is used to express explicitly a graph. This keyword defines statements of a graph instead of just a single node. The IRI can identify a named graph on `@id` at the same level as the tree. When a JSON-LD document's top-level structure contains the field name `@graph` and optionally `@context` the graph is considered to express the default graph. When multiple nodes, part of a graph, share the same context, they can be described in this mechanism, in which case they are represented as Flattened. The expected value of this keyword field is an array type.

If none of those mentioned above keywords is found, then we are confident that the current key is a node predicate, for which we have to expand the term if a previous context was found and push it to the stack of predicates. The next iteration captures the third piece of the triple.

Array tokens

ARRAY_START is returned when an opening bracket character is found ("[") which signals the start of a collection of values that carry the same predicate (if a previous one was matched) or if it's at the root level, then the confirmation that the format is in expanded form. Example 11 (a) shows the first case, and example (b) shows the second definition. Internally, the stack of operations are filled with this token.

```
(a):
[ { "@context": "http:...", "@graph": [ ... ] } ]
                                     ^ ARRAY_START
(b):
[ { "@context": "http:...", "@graph": [ ... ] } ]
^ ARRAY_START
```

Listing 11: Cases where the token ARRAY_START occur

ARRAY_END is found when the character "]" is found. This token indicates the end of an array; thus, the end of a triple. The appearance of this token signals the last predicate on the stack to be used, so the top of the stack of operations and predicates is removed.

Object tokens

OBJECT_START is returned when the character "{" is encountered. It indicates the start of a node or a value objects to be determined on the next iteration of the parser. The only operation performed here is the appending of the identifier to the stack of operations.

Finally, OBJECT_END indicates the closure of an object with the character "}". At this point, the parser has finished dealing with a node or value object. Consequently, the stack of operations is popped only if the last element was an opening object operation (OBJECT_START). Then the subject on the top of the stack is also popped, which indicates the scope of the subject is terminated. Finally, if the top of the stack of operations contains the aperture of an array (START_ARRAY), it also pops the stack of properties, in the case where a node was defined in the context of an array.

JSON-LD specifies three different representations for the same dataset. Chapter 2 details the differences between them. It's essential to make the distinction while reading the very first token. This determines which format the input is. As described, if the character matches an OBJECT_START, then the format may be in Flattened or Compacted form. This step is important, since it answers the requirement **R1** defined in the problem statement. On the other hand, if the first character is found to be an ARRAY_START, then the whole dataset is in Expanded form. The steps described above for each token are aware of the format because the stack of operations determines so. No further logic is needed to capture triples of data on the following iterations.

Algorithms

We present the main routine of the Quarser parser in Algorithm 2 and the routine for triple matching in Algorithm 1.

Algorithm 1: MATCHTRIPLEPATTERN

Input: Multiset B of variable bindings Q in R
Input: Triple pattern T
Input: RDF Triple t
Output: Binding *bindings*

```
1
2  S ← T.getSubject() ;
3  P ← T.getPredicate() ;
4  O ← T.getObject() ;
5  s ← t.getSubject() ;
6  p ← t.getPredicate() ;
7  o ← t.getObject() ;
8  bindings ← makeSetOfBindings() ;
9  if S.isVariable()&&P.isVariable()&&O.isVariable() then
10 |   binding.set(?S ← s, ?P ← p, ?O ← o) ;
11 else if S.isVariable()&&P.isVariable()&&O.matches(o) then
12 |   binding.set(?S ← s, ?P ← p) ;
13 else if S.isVariable()&&P.matches(p)&&O.isVariable() then
14 |   binding.set(?S ← s, ?O ← o) ;
15 else if S.isVariable()&&P.matches(p)&&O.matches(o) then
16 |   binding.set(?S ← s) ;
17 else if S.matches(s)&&P.isVariable()&&O.isVariable() then
18 |   binding.set(?P ← p, ?O ← o) ;
19 else if S.matches(s)&&P.isVariable()&&O.matches(o) then
20 |   binding.set(?P ← p) ;
21 else if S.matches(s)&&P.matches(p)&&O.isVariable() then
22 |   binding.set(?O ← o) ;
```

Algorithm 2: ParseJSONLD

```
    Input: JSON-LD Input Stream File I
    Input: RDF Graph G
    Input: Triple pattern T
    Output: Multiset B of variable bindings Q in R
 1  stackOperations ← MakeStack();
 2  stackSubjectNodes ← MakeStack();
 3  stackProperties ← MakeStack();
 4  uriExpansion ← MakeContextTermExpander();
 5  tokenizer ← makeTokenizer(I);
 6  resultBindings ← makeMultiSet();
 7
 8  Iterate over every token of the input stream ;
 9  while tokenizer ≠ null do
10      switch tokenizer do
11          case FIELD_NAME do
12              key ← tokenizer.next() ;
13              switch key do
14                  case @id do
15                      key ← tokenizer.next() ;
16                      nodeId ← String(tokenizer.next()) ;
17                      if stackSubjectNodes ≠ ∅ then
18                          s ← stackSubjectNodes.peek();
19                          p ← stackProperties.peek();
20                          o ← nodeId;
21                          triple ← CreateTriple(s, p, o);
22                          MatchTriplePattern(B, T, triple) ;
23                          G.addTriple(triple) ;
24                      stackSubjectNodes.push(nodeId) ;
25                  case @graph do
26                      break ;
27                  case @value do
28                      tokenizer.next() ;
29                      literalValue ← String(tokenizer.next()) ;
30                      if stackSubjectNodes ≠ ∅ then
31                          s ← stackSubjectNodes.peek();
32                          p ← stackProperties.peek();
33                          o ← literalValue;
34                          triple ← CreateTriple(s, p, o);
35                          MatchTriplePattern(B, T, triple) ;
36                          G.addTriple(triple);
37                      if stackOperations.peek() = START_OBJECT then
38                          stackOperations.pop();
39                          if stackOperations.peek() = START_ARRAY then
40                              stackProperties.pop();
41                      tokenizer.next() ;
42                  case @context do
43                      tokenizer.next() ;
44                      contextString ← String(tokenizer.next()) ;
45                      uriExpansion ← MakeContextObject(contextString) ;
46                  otherwise do
47                      if key = @type then
48                          stackProperties.push(URI_TYPE) ;
49          case VALUE_STRING do
50              value ← String(tokenizer.next()) ;
51              s ← stackSubjectNodes.peek();
52              p ← stackProperties.peek();
53              o ← value;
54              if p = URI_TYPE then
55                  o = uriExpansion.expand(o) ;
56              triple ← CreateTriple(s, p, o);
57              MatchTriplePattern(B, T, triple) ;
58              G.addTriple(triple);
59              if stackOperations.peek() = START_ARRAY then
60                  stackProperties.pop();
61          case START_OBJECT do
62              stackOperations.push(START_OBJECT) ;
63          case END_OBJECT do
64              if stackOperations.peek() = START_OBJECT then
65                  stackOperations.pop();
66                  if stackOperations.peek() = START_ARRAY then
67                      stackProperties.pop();
68          case START_ARRAY do
69              stackOperations.push(START_ARRAY) ;
70          case END_ARRAY do
71              if stackOperations.peek() = START_ARRAY then
72                  stackProperties.pop();
73                  stackOperations.pop();
74      tokenizer.next()
```

4.1.6 *Graph population, BGP Matching and variable binding*

Section 4.1.5 details the process in which nodes are captured on every token identifier. It has been shown that there are three cases in which a complete triple is fulfilled when landing to a leaf node during the tree traversal [16]. Two of the cases happen when literal nodes are found, and the thrid case when a named node is found with the `@id` keyword. At this step, the granularity of the elements has reached the state where a new triple of data can be added to the graph. In the figure 11, we can summarize the process of the construction of the graph, as tokens are processed and new triples are found. The numbers in red square indicate the order of the triple creation during the graph population.

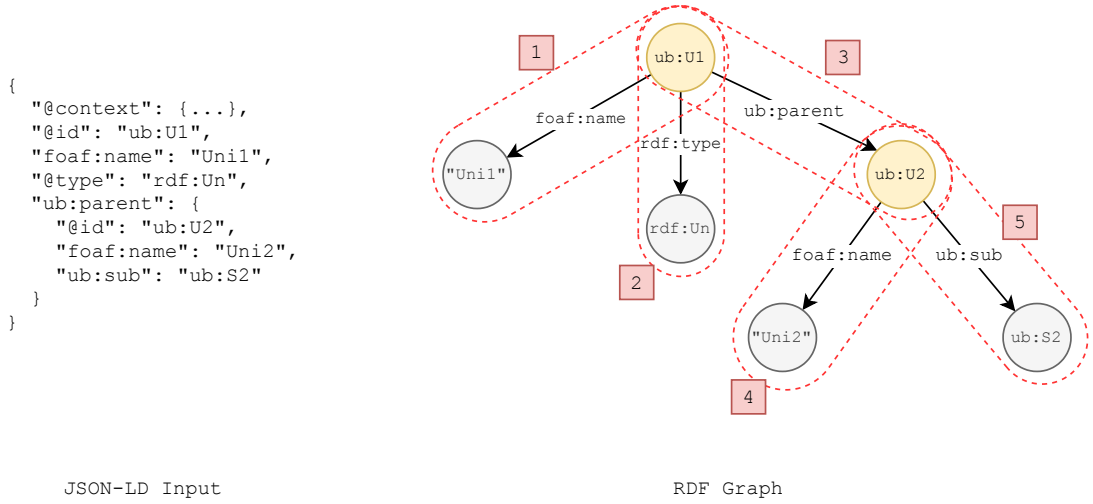


Figure 11: Example of RDF Graph in construction during parsing

During the same operation, having the triple constructed in memory, it is possible to perform the BGP matching against the new triple. Notice that knowing the BGP query upfront is a clear advantage to speed up the query execution, and also, is a requirement that we are solving (**R2**). With the variables bound in hand, we can bind their respective values if the pattern matches successfully. The table 2 shows the seven possible cases that can lie.

A BGP is composed of one or more triple patterns. It's reasonable to iterate over every triple pattern, match against the data, and accumulate the bindings. On the other hand, we can parallelize the parsing and delegate each triple pattern to a separate thread of execution, so the bindings are kept separately for later join. At this point, we could solve "join" operations of the algebra. However, the case where a blank node is used to capture an intermediate node makes the join a non-trivial operation since the canonical naming of blank nodes is non-deterministic [20]. When multiple triple patterns come in a BGP, we propose to take the one that projects fewer variables. A simple sorting subroutine shown in Algorithm 2 captures the most straightforward triple pattern to be matched.

Patterns	Operation
?s, ?p, ?o	Bind all variables s, p, o
?s, ?p, o	Bind s, p if o matches the node value
?s, p, ?o	Bind s, o if p matches the node value
?s, p, o	Bind s if p and o match the node values
s, ?p, ?o	Bind s, o if p matches the node value
s, ?p, o	Bind s if s and o match the node values
s, p, ?o	Bind s if s and p match the node values

Table 2: Possible cases of a triple variable binding

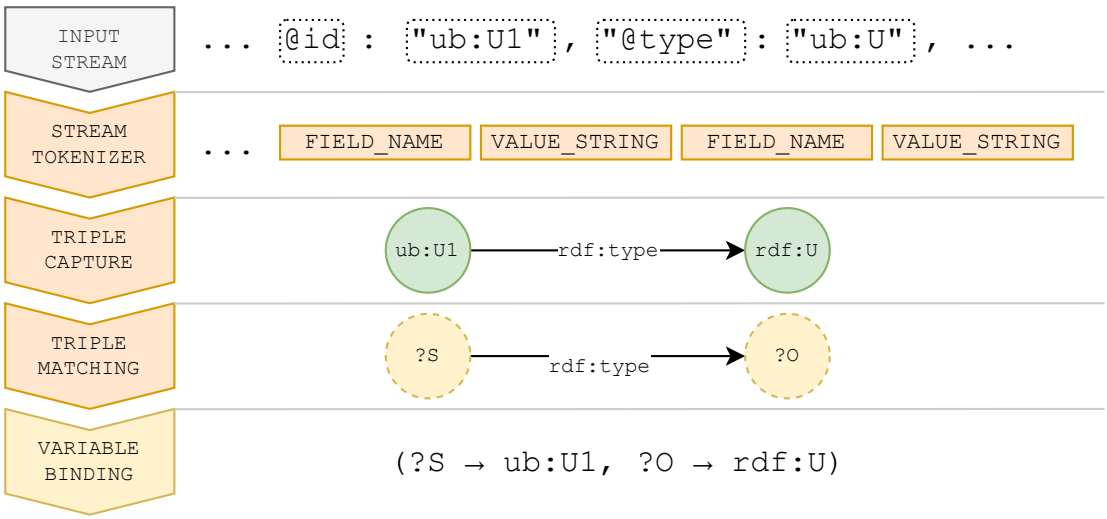


Figure 12: Running example of parsing

For the sake of explainability, consider the data input in the listing 12 and a basic BGP with one triple pattern and two variables, `(?S rdf:type ?O)`. Let L be the list of bindings to be collected along the parsing process, and let Bi be the set of bindings of the triple pattern i. The resulting list L of bindings can be empty in case no match is successfully applied. In the Figure 12 we visualize the execution of this input. When a triple pattern is bound on the iteration that captures the RDF triple we match the variables `?S` and `?P`.

4.1.7 *Post initial-binding phase*

After the triple pattern binding of the given query, the algorithm switches to the Join phase. Quarser uses a simple heuristic to find a convenient initial triple pattern to bind. However, at this point, the query is not fully solved yet. It needs to be solved with a query executor. The rest of the algebra is solved with the use of an initial binding that is provided by the Quarser step. Remember that the parsing gives the most relevant binding variables and the model of the graph itself. In this regard, the rest of the query execution is performed

```
{
  "@context": {
    ...
    "name": "http://schema.org/name"
  },
  "@id": "ub:U1",
  "@type": "ub:U",
  ...
}
```

Listing 12: Example of a Compacted structure

in every iteration of the binding list, represented as a result set. In the case of Jena, the query executor relies on the variables bound to the current iteration and the model that has been populated.

Finally, the join phase of the initial binding and the rest of the triple patterns save time executing the queries. The result of the execution contains every variable requested to be bound in the original SPARQL query. Apache Jena does allow this partial solution to execute a query, and ARQ receives the stream of results from previous stages to the evaluation. This feature enables the execution of subsequent queries, which is the requirement **R3** being solved.

4.2 ALGORITHM IMPLEMENTATION

For the proof of concept of this thesis, we used Jackson 2.12, a largely known JSON library for Java, used in many open-source data systems, including Apache Spark. Jackson partially supports "projection push down" by offering the methods "SkipChildren()" and "NextToken()."

Apache Jena is a mature, open-source Semantic Web for Java and triple store for RDF [29]. That is why this is the choice for this thesis. We use it to implement two applications used as experiments, described in Chapter 5. Apache Jena provides an API to lower-level access to RDF-related classes, mainly Triple, Binding, Query parsing, ARQ, and Query Execution Planning. We leverage the use of these constructs.

We used a byte-streaming approach for the parser and algorithm, which is the most efficient and has the lowest processing and memory overhead, as demostrated in [26]. Real-time streaming of JSON data is common in analytic workloads, where the processing is done on the devices at the edge [26]. This feature is part of the solution to the requirement that leads to the Micro research question.

The algorithm skips the traditional way of solving the SPARQL algebra [1] after the data is converted to an RDF graph in memory. The natural sequence of bytes in the data input allows it to match for triple patterns as new keywords and entries are tokenized. First, it finds triples, adds them to the constructing

RDF graph, and given a triple pattern; it creates solution bindings. Due to this design, Quarser avoids a great deal of wasted work incurred by current state-of-the-art triple stores by reading the dataset entirely, parsing, and then rereading the dataset to solve future queries.

The idea is to implement a custom high-performance parser that is aware of the structure of the data when it is read and the graph traversed. The straightforward idea is to parse the text input to a JSON object and traverse the object in memory as a tree. However, this line of thinking is non-optimal because the computation used to deal with the data is perfectly aware of the triple patterns that are being constructed.

The binding of the variables is tied to the parsing process. We implement the parser assuming that the access pattern to the input stream of data is sequential. Either by starting at the beginning or end of the file. Reading the bytes in a sequence has the advantage of access only to the "event" or "window" that was parsed in the document sequence.

4.2.1 *Conclusion*

This chapter presented the design of Quarser. We explained why the traditional method of parsing can be leveraged to, not only parse JSON-LD structures, but also to build the graph data model, match triple patterns, and answer SPARQL Queries. We present results of a benchmark of this algorithm versus an in-memory implementation of Apache Jena executing datasets that comprise millions on triples in the Chapter 5.

5 EVALUATION

In this chapter, we present the design of the experiments, the results, and a discussion of our experimental results. Notably, in section 5.1, we present the LUBM dataset, queries, and the hardware configuration for the benchmarking. Section 5.2 presents the results and analysis of the findings.

5.1 BENCHMARK

Quarser is evaluated on the Lehigh University Benchmark (LUBM) [17]. LUBM is a synthetic benchmark that features an OWL ontology for the university domain. It enables the generation of datasets of arbitrary size and 14 queries with different complexities, ranging from small queries with no joins to large queries up to five joins. Figure 13 shows the data schema of LUBM, featuring universities, their departments, the people that work in those departments, and university activities. LUBM is suited for analytical benchmark workloads, and is one of the prefered benchmarks on the research community because of its adaptation to different research domains [6]. A proposal for a binary JSON-LD used LUBM for benchmarking devices under constrained resources [9].

Our evaluations are performed on five different scales of the dataset, considering each JSON-LD representation.

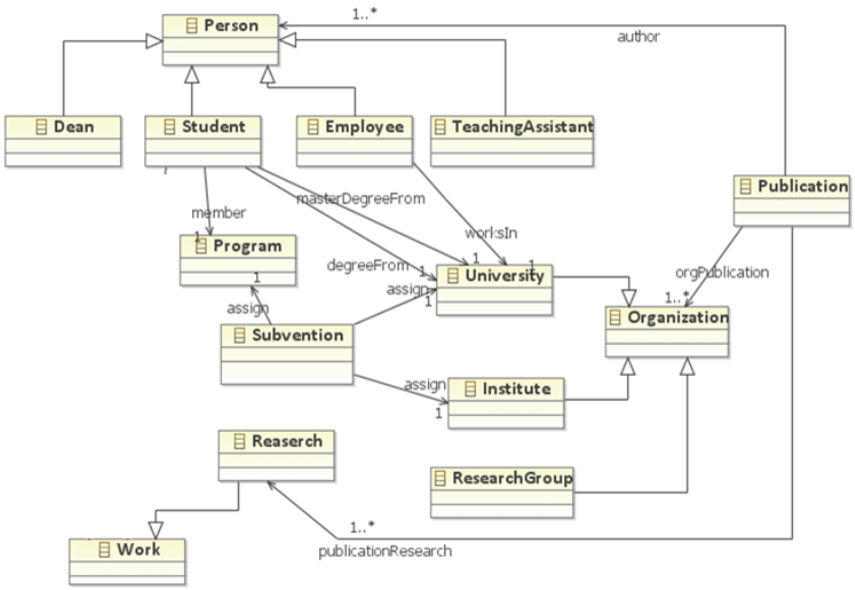


Figure 13: Data schema of the Lehigh University Benchmark (LUBM) studied in [21]

Scale	Nr. of triples	Size
10	1,272,953	Expanded: 125MB Flattened: 94MB
25	3,316,500	Expanded: 327MB Flattened: 247MB
50	6,656,560	Expanded: 657MB Flattened: 496MB
75	10,068,204	Expanded: 995MB Flattened: 751MB
100	13,409,395	Expanded: 1.3GB Flattened: 1.0GB

Table 3: Number of triples and size of file for every dataset scale

5.1.1 *Test Platform*

The experiments have been conducted on a single Linux machine with Ubuntu 18.04. The instance uses 16-AMD cores, 64GB of main memory, and a high-speed 100GB SSD drive. Java JDK 11 is installed to generate the dataset and run the experiments. The source code uses the Apache Jena API 4.0.

5.1.2 *Scales*

We evaluated LUBM over five different scales: 10, 25, 50, 75, and 100, where the scale indicates the number of universities. The amount of triples for a single university (scale 1) is 8,521 [1]. Data has been generated for every JSON-LD format: Expanded, Compacted, and Flattened using Apache Jena. However, we only use the Expanded and Flattened formats since the output generated for the Compacted is identical to the Flattened one.

The process of generation of this data consisted in loading the original OWL files represented in RDF/XML to Jena and output one single JSON-LD dataset file per format. The context used for the generation of Flattened and Compacted is shown in Listing 13. The resulting amount of triples and size on disk is shown in Table 3

5.1.3 *Queries*

Our experiments on LUBM explore all the 14 queries of the benchmark. Table 4 shows the basic graph pattern and variables asked to project for every LUBM query. Queries 1, 3, 5, 6, 10, 11, 13, and 14 involve just one-variable triple patterns. Query 6 asks for all students, which results in an operation with

[1] No parameter was changed for the code generation, we used the seed equal to 0

```
{
  "@context": {
    "name": "http://cs.ut.ee/ldparser#name",
    "subOrganizationOf": "http://cs.ut.ee/ldparser#subOrganizationOf",
    "degreeFrom": "http://cs.ut.ee/ldparser#degreeFrom",
    "doctoralDegreeFrom": "http://cs.ut.ee/ldparser#doctoralDegreeFrom",
    "mastersDegreeFrom": "http://cs.ut.ee/ldparser#mastersDegreeFrom",
    "memberOf": "http://cs.ut.ee/ldparser#memberOf",
    "teacherOf": "http://cs.ut.ee/ldparser#teacherOf",
    "undergraduateDegreeFrom": "http://cs.ut.ee/ldparser#undergraduateDegreeFrom",
    "worksFor": "http://cs.ut.ee/ldparser#worksFor",
    "publicationAuthor": "http://cs.ut.ee/ldparser#publicationAuthor",
    "advisor": "http://cs.ut.ee/ldparser#advisor",
    "takesCourse": "http://cs.ut.ee/ldparser#takesCourse",
    "teachingAssistantOf": "http://cs.ut.ee/ldparser#teachingAssistantOf",
    "headOf": "http://cs.ut.ee/ldparser#headOf",
    "Department": "http://cs.ut.ee/ldparser#Department",
    "Organization": "http://cs.ut.ee/ldparser#Organization",
    "ResearchGroup": "http://cs.ut.ee/ldparser#ResearchGroup",
    "University": "http://cs.ut.ee/ldparser#University",
    "AssistantProfessor": "http://cs.ut.ee/ldparser#AssistantProfessor",
    "Employee": "http://cs.ut.ee/ldparser#Employee",
    "Faculty": "http://cs.ut.ee/ldparser#Faculty",¿¿m
    "Person": "http://cs.ut.ee/ldparser#Person",
    "Professor": "http://cs.ut.ee/ldparser#Professor",
    "Course": "http://cs.ut.ee/ldparser#Course",
    "Work": "http://cs.ut.ee/ldparser#Work",
    "GraduateCourse": "http://cs.ut.ee/ldparser#GraduateCourse",
    "Publication": "http://cs.ut.ee/ldparser#Publication",
    "GraduateStudent": "http://cs.ut.ee/ldparser#GraduateStudent",
    "Student": "http://cs.ut.ee/ldparser#Student",
    "TeachingAssistant": "http://cs.ut.ee/ldparser#TeachingAssistant",
    "AssociateProfessor": "http://cs.ut.ee/ldparser#AssociateProfessor",
    "FullProfessor": "http://cs.ut.ee/ldparser#FullProfessor",
    "ResearchAssistant": "http://cs.ut.ee/ldparser#ResearchAssistant",
    "Lecturer": "http://cs.ut.ee/ldparser#Lecturer",
    "UndergraduateStudent": "http://cs.ut.ee/ldparser#UndergraduateStudent",
    "emailAddress": "http://cs.ut.ee/ldparser#emailAddress",
    "telephone": "http://cs.ut.ee/ldparser#telephone",
    "Chair": "http://cs.ut.ee/ldparser#Chair",
    "hasAlumnus": "http://cs.ut.ee/ldparser#hasAlumnus"
  }
}
```

Listing 13: Context of LUBM used to transform the dataset to JSON-LD

high selectivity. Queries 7 and 12 involve two variables. Queries 1, 3, and 7 are complex queries with large intermediate results but very few, or none, final results. Finally, queries 2 and 9 are the most complicated LUBM queries since they involve large intermediate result because it evaluates expensive join operations refered as "triangular".

5.2 EXPERIMENTS AND RESULTS

5.2.1 *Implementation*

We implemented two applications used for comparing the performance of the query answering of SPARQL queries given multi-format JSON-LD datasets.

Query	BGP	Variables
Q1	**?X rdf:type ub:GraduateStudent .** **?X ub:takesCourse <http://www.Departmento.Universityo.edu/GraduateCourseo>**	?X
Q2	**?X rdf:type ub:GraduateStudent .** **?Y rdf:type ub:University .** **?Z rdf:type ub:Department .** **?X ub:memberOf ?Z .** **?Z ub:subOrganizationOf ?Y .** **?X ub:undergraduateDegreeFrom ?Y .**	?X ?Y ?Z
Q3	**?X rdf:type ub:Publication .** **?X ub:publicationAuthor <http://www.Departmento.Universityo.edu/AssistantProfessoro>.**	?X
Q4	**?X rdf:type ub:Professor .** **?X ub:worksFor <http://www.Departmento.Universityo.edu>.** **?X ub:name ?Y1 .** **?X ub:emailAddress ?Y2 .** **?X ub:telephone ?Y3 .**	?X ?Y1 ?Y2 ?Y3
Q5	**?X rdf:type ub:Person .** **?X ub:memberOf <http://www.Departmento.Universityo.edu>.**	?X
Q6	**?X rdf:type ub:Student**	?X
Q7	**?X rdf:type ub:Student .** **?Y rdf:type ub:Course .** **?X ub:takesCourse ?Y .** **<http://www.Departmento.Universityo.edu/AssociateProfessoro>ub:teacherOf ?Y .**	?X ?Y
Q8	**?X rdf:type ub:Student .** **?Y rdf:type ub:Department .** **?X ub:memberOf ?Y .** **?Y ub:subOrganizationOf <http://www.Universityo.edu>.** **?X ub:emailAddress ?Z .**	?X ?Y ?Z
Q9	**?X rdf:type ub:Student .** **?Y rdf:type ub:Faculty .** **?Z rdf:type ub:Course .** **?X ub:advisor ?Y .** **?Y ub:teacherOf ?Z .** **?X ub:takesCourse ?Z .**	?X ?Y ?Z
Q10	**?X rdf:type ub:Student .** **?X ub:takesCourse <http://www.Departmento.Universityo.edu/GraduateCourseo>.**	?X
Q11	**?X rdf:type ub:ResearchGroup .** **?X ub:subOrganizationOf <http://www.Universityo.edu>.**	?X
Q12	**?X rdf:type ub:Chair .** **?Y rdf:type ub:Department .** **?X ub:worksFor ?Y .** **?Y ub:subOrganizationOf <http://www.Universityo.edu>.**	?X ?Y
Q13	**?X rdf:type ub:Person .** **<http://www.Universityo.edu>ub:hasAlumnus ?X .**	?X
Q14	**?X rdf:type ub:UndergraduateStudent .**	?X

Table 4: Basic graph pattern and variable(s) to bind per each query

First, a pure-Jena in-memory application, called *Jena in-memory* that uses the API of Jena to parse the query, parse the dataset and populate the model, create the execution plan and finally bind the variables on a list of binding results. Second, an application for *Quarser* that also uses the Jena API for the query parsing, model interface, and post-initial binding through a query execution plan that outputs the final bindings on a set of results. Both applications expect the same query, same data input, and same variables to be bound as output. Both applications report the execution of time spend, in milliseconds, for the following operations:

- *Jena in-memory*: Time of Parsing
- *Jena in-memory*: Time of Binding
- *Jena in-memory*: Time of Parsing+Binding
- *Jena in-memory*: Total time
- *Quarser*: Time of Parsing+Binding
- *Quarser*: Time of Post-binding (Executing the selection once again)
- *Quarser*: Total time

In all the experiments discussed in this thesis, we provided the dataset as a single file. We ran all experiments using a single thread and gave the full capacity of available memory to the JVM heap.

5.2.2 *Results*

The experiments on LUBM ran over five scales, measuring the time of parsing, binding, and post-execution execution binding. The measure of Jena-in memory splits the time of parsing and binding because both are different executions. For Quarser, this result is joint.

Figure 14 displays the averaged results of the execution times for every tested scale [2]. The Y axis reflects the results of the total time spent parsing and querying every format and query.

5.2.3 *Analysis and Discussion*

As mentioned earlier, every query on LUBM has different complexity. Some of the queries define a BGP with a substantial number of triple patterns and join between them. Others are more simple and contain a single triple pattern or two. Hence, it is not surprising that for certain queries, the ARQ optimizer

[2] Execution times of every operation are shown in the Appendix A.2.

Figure 14: Total time of parsing and query of Jena in-memory vs. Quarser in Flattened and Extended JSON-LD datasets for the scales 10, 25, 50, 75, and 100

yields faster results. To this extent, we analyze how and why the execution time differs from one format to another. Finally, we explore how the initial binding optimizes the execution of the ARQ execution in Jena.

Queries 1, 3, 5, 6, 10 involve just one variable triple pattern. Each of these queries has two triple patterns; one specifies the type of the variable "?X", and the other specifies a constraint on "?X". In the scale of 10, shown in Figure 14 (a) shows a performance gain with Quarser because the selected triple that represents the constraint, binds the variables on the parsing process. Thus, it speeds up the rest of the query execution.

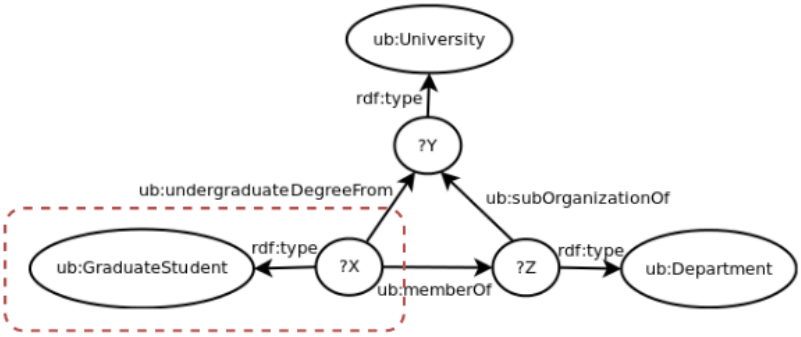


Figure 15: Shape of the LUBM Query 2

Consider the shape of the query 2 in 15 and the results in Figure 14 (e) [3]. This query increases in complexity: three classes and three properties are involved. Additionally, there is a triangular pattern of relationships between the objects involved. Solving the first pattern (framed in red) specifying the type of "?X" invariably leads to many matches, almost all of which are discarded. The results of scale 100 Expanded confirm this hyphotesis. While Jena in-memory takes approximately 160 seconds to bind all the variables, Quarser does it in 145 seconds. The experiment on Quarser helps Jena select the constraint of the given triple, as the degree on the specified object is relatively small in all cases. We observe a similar behavior on the counterpart Flattened version. While Jena in-memory solves the query in about 5 minutes, Quarser improves it to about 10 seconds less. The flat and direct node object representation of this format is traduced in faster bindings and faster query response.

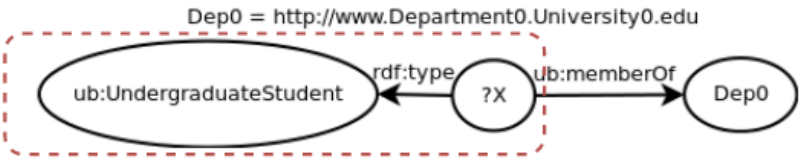


Figure 16: Shape of the LUBM Query 6

Query 6, shown in 16 is about one class, but it assumes that the relationship between *UndergraduateStudent* and *Student* is explicit, and the one between *Student* and *GraduateStudent* is implicit. As a result, the execution of this query is extensive in input and low in selectivity. The Quarser solves the only triple pattern, so the ARQ optimizer does a little further to complete the query answering. On the scale of 100, we see the difference between 157 (Jena in-memory) versus 136 (Quarser) seconds for the expanded dataset and 152 (Jena in-memory) versus 122 (Quarser) for the flattened. This gain in time represents 20% of the total time of Jena without optimization. A similar behavior we

[3] for more detailed results look at A.2

observe in query 5, represented in the figure 14 (e): by solving the first type constraint, the performance gain is alike.

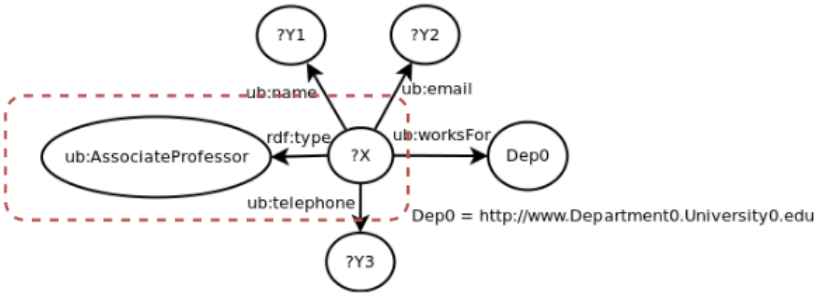


Figure 17: Shape of the LUBM Query 4

Query 4 (Figure 17) involves four variables, but they are still simple. The query has small input and high selectivity. It assumes *AssociateProfessor* and its subclasses are related to a common object. This class has a wide hierarchy, thus it features multiple properties of a single class. The results in scale 100 show that by binding the variables in-advance, Jena improves the query answering by 20%. This applies to both the flattened and expanded datasets.

5.3 CONCLUSION

We have shown that for different shapes of query and JSON-LD data formats, Quarser outperforms Jena by 20% of reduction of query time on every scale. Speaking about formats, we conclude that Flattened is faster than Expanded because the tokenization process is less expensive due to the decreased amount of characters to traverse, and also due to the direct triple representation that is leveraged by the Quarser algorithm. We provide additional scalability plots in Appendix A.1 for the readers interested in the aggregation times per scale per query execution.

6 CONCLUSION

With the increasing demand for analytics over Knowledge Graphs, and JSON-LD adopted as a data serialization format, it is critical the need for a efficient RDF Triple Store that is responsive on this constraints. For this reason, the presented thesis develops Quarser as an alternative parser for JSON-LD data that speeds up the execution time of recurrent SPARQL queries.

In Chapter 2, we set the background to understand the underlying principles of Semantic Web, and the technologies needed to construct a JSON-LD parser.

In Chapter 3, we presented the problematic behind the analytical workloads of recurrent SPARQL queries using the Macro, Meso-Micro-framework. We formulated the Macro-level research question: *How to improve the execution time of analytical SPARQL queries?*, the Meso-level question: *Can we share the same computation resources for parsing and answering recurrent SPARQL queries?*, and the Micro-level question: *How can we solve part of a SPARQL query on the parsing process over JSON-LD datasets?*. These questions helped us to design the solution of the problem.

In Chapter 4 we developed the design and implementation of the parsing algorithm for SPARQL queries. We discussed the role of the Buffered Input Stream and the Tokenizer, which are the building blocks for our solution. We implemented the JSON-LD parser by understanding each keyword and notation of its formats. Our solution pushed down the answering of queries on the parsing step, and allowed to construct the RDF graph for its subsequent use.

In Chapter 5 we performed a systematic experimental evaluation of our implementation. We used the Lehigh University Benchmark (LUBM) and executed the 14 queries to measure their execution time. Our evaluations ran over five different scales of the dataset, considering each JSON-LD representation (Flattened, Expanded, Compacted). We compared our execution times with a similar experiment executed on Apache Jena. The comparison of results shown a time-saving of 20% of every query executed when using Quarser. We provided an analysis of these results, and theorized why the Flattened format is the best performant.

Regardless of our positive results, we can not generalize them for any SPARQL analytical workload. Our results are bound by the limitation of one benchmark, and one particular Triple store, Apache Jena. Furthermore, we consider data that is static, which is the case of recurrent query workloads.

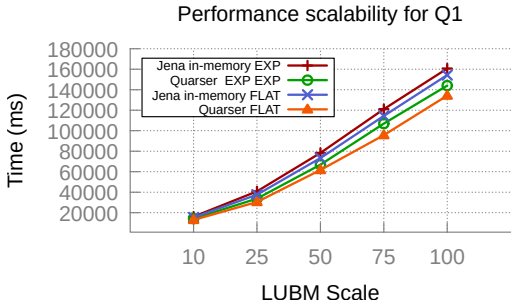
As a future extension of this work, additional tests against other benchmark datasets (i.e. WatDiv, SP2Bench) and other Triple Stores (i.e. Virtuoso, Blaze-

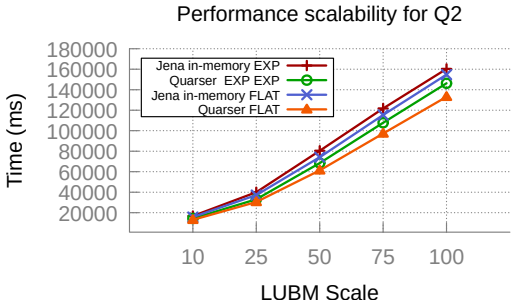
graph) can be studied. In addition, a native implementation of Quarser for RDF Stream Processing for Java (RSP4J) would enable rapid JSON-LD parsing and query answering in streaming-enabled environments.

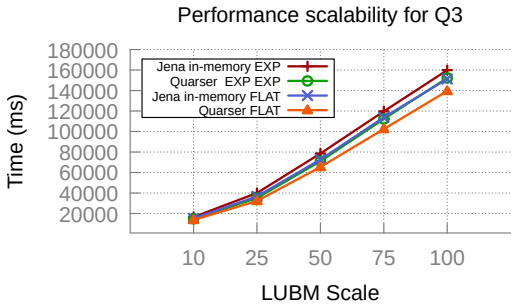
A

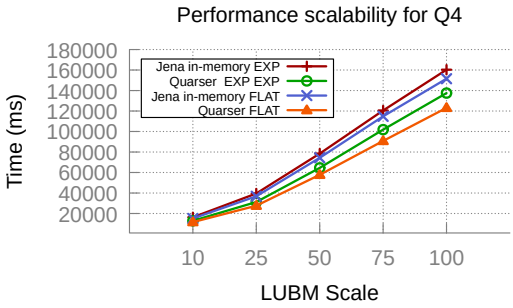
APPENDIX

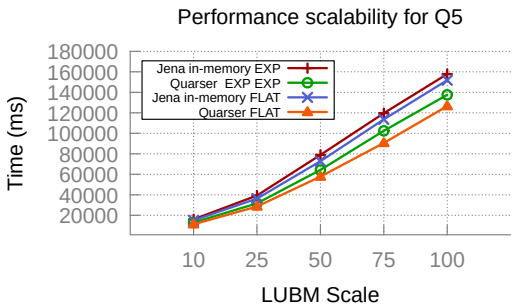
A.1 SCALABILITY OF THE QUERY EXECUTION TIME FOR ALL 14 QUERIES OF LUBM

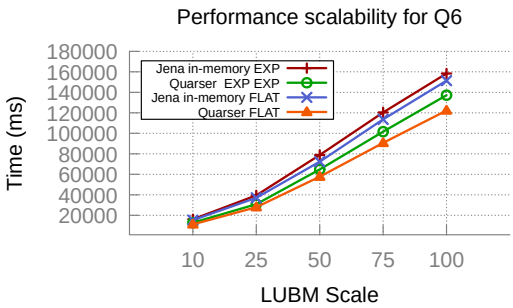


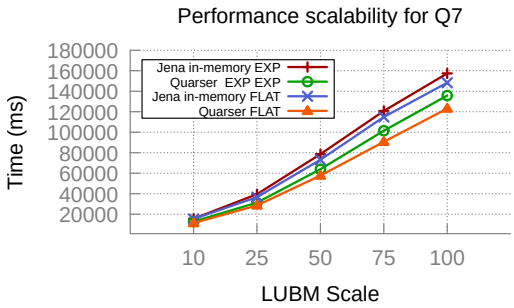


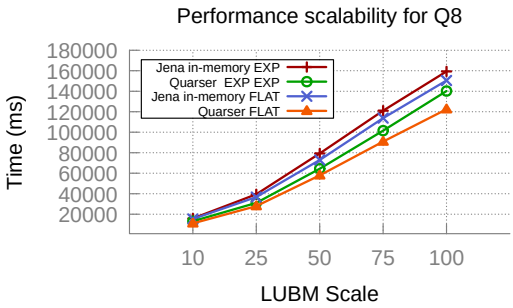


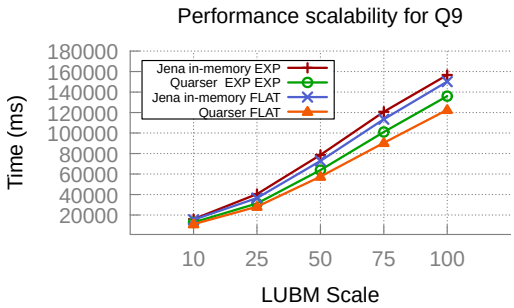


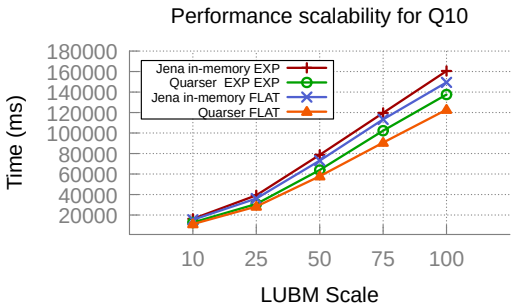


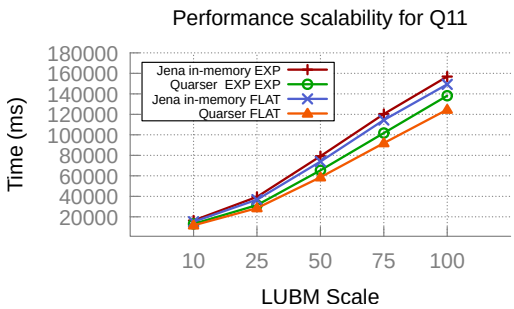


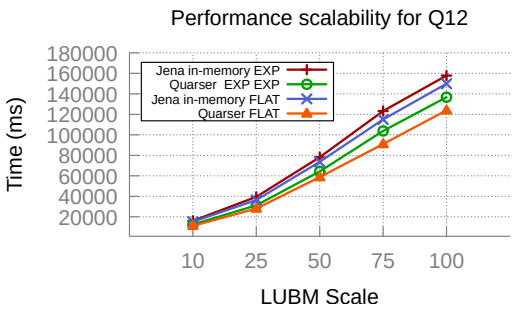


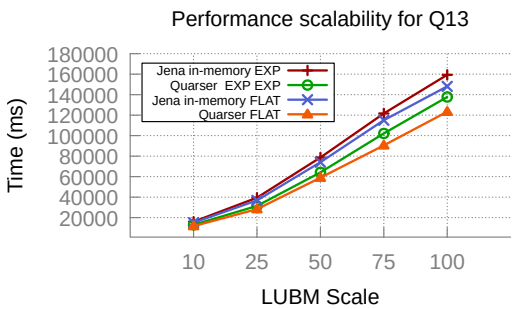


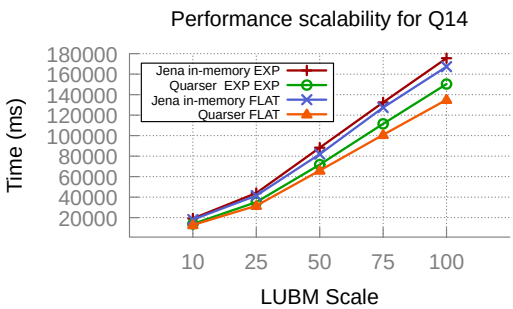












A.2 EXECUTION TIMES FOR EXPERIMENTS IN JENA IN-MEMORY AND QUARSER

SCALE 10		Jena in-memory					Quarser		
Query	**Format**	**parsing**	**binding**	**parsing + binding**	**post**	**total (ms)**	**parsing + binding**	**post**	**total (ms)**
Q1	**EXP**	15924	112	16036	26	16062	13017	1232	14249
	FLAT	15151	136	15287	26	15312	11772	1170	12943
Q2	**EXP**	16257	110	16367	366	16734	13118	1253	14371
	FLAT	15211	114	15326	366	15691	11725	1324	13050
Q3	**EXP**	16397	106	16503	27	16530	12749	2064	14812
	FLAT	15068	104	15172	26	15198	11574	2117	13691
Q4	**EXP**	16186	107	16292	8	16300	12835	0	12835
	FLAT	15034	108	15142	9	15151	11564	0	11564
Q5	**EXP**	15908	106	16014	20	16034	12903	0	12903
	FLAT	14898	105	15003	23	15027	11232	0	11232
Q6	**EXP**	15991	105	16096	2	16098	12760	2	12762
	FLAT	15144	99	15243	2	15245	11131	2	11133
Q7	**EXP**	15431	112	15543	8	15551	12775	0	12775
	FLAT	15197	108	15304	7	15311	11324	0	11324
Q8	**EXP**	15739	112	15850	53	15904	13147	0	13147
	FLAT	15156	107	15263	55	15318	11061	0	11061
Q9	**EXP**	15866	110	15976	2	15978	12768	0	12768
	FLAT	15325	108	15433	2	15435	11039	0	11039
Q10	**EXP**	16138	114	16251	5	16257	12709	0	12709
	FLAT	15204	102	15305	5	15310	11119	0	11119
Q11	**EXP**	16261	106	16366	5	16371	12653	402	13055
	FLAT	15088	136	15223	5	15228	11222	409	11631
Q12	**EXP**	15943	106	16049	23	16072	12493	0	12493
	FLAT	15027	108	15134	22	15156	11226	0	11226
Q13	**EXP**	15943	106	16049	3	16052	12938	0	12938
	FLAT	14918	105	15023	3	15026	11765	0	11765
Q14	**EXP**	16169	103	16272	2854	19126	12302	1530	13832
	FLAT	15491	109	15601	2464	18065	11303	1508	12811

Table 5: Results for the time of execution for the scale 10

SCALE 25		Jena in-memory					Quarser		
Query	**Format**	**parsing**	**binding**	**parsing + binding**	**post**	**total (ms)**	**parsing + binding**	**post**	**total (ms)**
Q1	**EXP**	40686	113	40799	28	40827	31716	2010	33725
	FLAT	37462	102	37564	26	37590	28517	1902	30419
Q2	**EXP**	39210	104	39314	653	39966	30656	2434	33089
	FLAT	36430	109	36540	664	37203	28089	2321	30410
Q3	**EXP**	39823	108	39930	29	39959	30944	4150	35094
	FLAT	36503	107	36610	26	36635	28305	3878	32183
Q4	**EXP**	39385	108	39493	8	39501	31201	0	31201
	FLAT	36699	110	36808	9	36817	27590	0	27590
Q5	**EXP**	39177	116	39292	22	39314	31609	0	31609
	FLAT	36200	107	36307	20	36327	28539	0	28539
Q6	**EXP**	39368	104	39472	2	39474	30803	2	30804
	FLAT	36931	108	37038	3	37041	27858	2	27860
Q7	**EXP**	39385	109	39494	8	39501	31248	0	31248
	FLAT	36591	105	36695	7	36702	28544	0	28544
Q8	**EXP**	39287	105	39392	53	39445	31056	0	31056
	FLAT	36566	108	36673	54	36728	27917	0	27917
Q9	**EXP**	40304	110	40414	2	40416	31223	0	31223
	FLAT	36419	100	36519	2	36521	28085	0	28085
Q10	**EXP**	39111	109	39219	5	39225	30764	0	30765
	FLAT	35936	106	36042	5	36047	28269	0	28269
Q11	**EXP**	39330	109	39439	6	39445	30825	604	31428
	FLAT	36624	107	36731	5	36737	27925	573	28498
Q12	**EXP**	39331	106	39437	23	39460	31204	0	31204
	FLAT	36238	114	36351	25	36376	28068	0	28068
Q13	**EXP**	39350	108	39458	3	39461	31419	0	31419
	FLAT	36720	106	36826	4	36830	28227	0	28227
Q14	**EXP**	39374	104	39478	4465	43943	31807	3298	35105
	FLAT	36829	111	36939	4547	41486	28036	3491	31526

Table 6: Results for the time of execution for the scale 25

SCALE 50		Jena in-memory					Quarser		
Query	**Format**	**parsing**	**binding**	**parsing + binding**	**post**	**total (ms)**	**parsing + binding**	**post**	**total (ms)**
Q1	**EXP**	78207	109	78315	27	78343	63736	3114	66850
	FLAT	73159	117	73276	30	73306	58445	3225	61670
Q2	**EXP**	79128	108	79236	1208	80444	64244	4200	68444
	FLAT	72900	111	73011	1157	74168	57184	4074	61258
Q3	**EXP**	78413	110	78523	27	78550	63919	7155	71074
	FLAT	72548	106	72654	27	72680	58116	7220	65336
Q4	**EXP**	78403	110	78513	8	78521	64548	0	64548
	FLAT	74229	107	74336	8	74344	57790	0	57790
Q5	**EXP**	78832	110	78941	22	78963	64317	0	64317
	FLAT	72648	109	72757	20	72777	57734	0	57734
Q6	**EXP**	78742	103	78845	3	78848	64765	1	64766
	FLAT	72246	106	72353	2	72355	57684	2	57685
Q7	**EXP**	78550	107	78657	8	78665	63927	0	63927
	FLAT	73000	113	73114	7	73121	57660	0	57661
Q8	**EXP**	79107	113	79220	57	79276	64377	0	64377
	FLAT	72758	112	72870	54	72924	57940	0	57940
Q9	**EXP**	78497	116	78613	2	78615	63845	0	63845
	FLAT	72664	120	72784	2	72786	57464	0	57464
Q10	**EXP**	78582	117	78700	5	78705	63896	0	63896
	FLAT	72964	106	73070	5	73075	57853	0	57853
Q11	**EXP**	79068	122	79190	7	79196	64474	851	65325
	FLAT	73774	107	73881	6	73887	57817	808	58625
Q12	**EXP**	78020	116	78135	22	78157	64285	0	64285
	FLAT	73573	108	73681	24	73705	58712	0	58712
Q13	**EXP**	78610	110	78720	4	78724	64126	0	64126
	FLAT	73967	113	74079	3	74083	58928	0	58928
Q14	**EXP**	80278	110	80388	7878	88265	65068	6647	71715
	FLAT	74065	111	74176	7785	81961	59337	6644	65981

Table 7: Results for the time of execution for the scale 50

SCALE 75		Jena in-memory					Quarser		
Query	**Format**	**parsing**	**binding**	**parsing + binding**	**post**	**total (ms)**	**parsing + binding**	**post**	**total (ms)**
Q1	**EXP**	120894	108	121002	26	121028	102197	4727	106924
	FLAT	114307	106	114413	26	114440	90990	4695	95685
Q2	**EXP**	120010	108	120118	1635	121753	101791	5956	107746
	FLAT	113383	109	113492	1835	115327	91155	6097	97252
Q3	**EXP**	119588	108	119696	27	119723	101739	10561	112300
	FLAT	113710	112	113821	30	113851	91486	10857	102343
Q4	**EXP**	120419	119	120537	8	120545	101817	0	101817
	FLAT	114701	114	114815	10	114825	90820	0	90820
Q5	**EXP**	119526	113	119639	23	119663	102413	0	102413
	FLAT	113484	112	113595	23	113618	90591	0	90591
Q6	**EXP**	120264	99	120363	2	120365	101612	2	101614
	FLAT	113527	112	113638	2	113641	90659	1	90661
Q7	**EXP**	120682	108	120789	7	120796	101470	0	101470
	FLAT	114761	109	114871	8	114879	90634	0	90634
Q8	**EXP**	120928	110	121037	58	121095	101574	0	101574
	FLAT	113578	107	113685	56	113741	90879	0	90879
Q9	**EXP**	120617	109	120726	2	120728	101023	0	101023
	FLAT	113384	104	113488	2	113490	90458	0	90458
Q10	**EXP**	119550	111	119661	5	119666	102317	0	102317
	FLAT	113172	118	113290	6	113296	90627	0	90627
Q11	**EXP**	120197	106	120303	6	120309	100622	1163	101784
	FLAT	114368	112	114479	6	114485	90875	1086	91961
Q12	**EXP**	123121	117	123238	22	123261	103771	0	103771
	FLAT	114892	112	115004	24	115027	91036	0	91036
Q13	**EXP**	121559	109	121668	3	121672	102145	0	102145
	FLAT	114967	110	115078	3	115081	90498	0	90498
Q14	**EXP**	120256	107	120363	12141	132504	101183	10360	111544
	FLAT	115345	105	115450	11989	127438	90547	10235	100783

Table 8: Results for the time of execution for the scale 75

SCALE 100		Jena in-memory					Quarser		
Query	**Format**	**parsing**	**binding**	**parsing + binding**	**post**	**total (ms)**	**parsing + binding**	**post**	**total (ms)**
Q1	**EXP**	160567	120	160687	27	160714	138208	5793	144001
	FLAT	154215	113	154328	25	154354	128224	6049	134273
Q2	**EXP**	157936	129	158065	2276	160341	138268	8042	146310
	FLAT	152394	121	152515	2279	154794	125058	8033	133092
Q3	**EXP**	159699	115	159814	29	159843	137616	14653	152269
	FLAT	151124	114	151237	30	151267	124776	14775	139551
Q4	**EXP**	160184	121	160305	8	160313	137533	0	137533
	FLAT	151400	118	151518	8	151526	123103	0	123103
Q5	**EXP**	157746	111	157857	18	157875	137612	0	137612
	FLAT	151655	131	151787	22	151809	126539	0	126539
Q6	**EXP**	158276	107	158383	2	158385	137225	2	137227
	FLAT	151249	109	151357	2	151360	122281	1	122282
Q7	**EXP**	157232	124	157356	7	157363	135771	0	135771
	FLAT	148220	118	148338	7	148345	123096	0	123096
Q8	**EXP**	159152	134	159286	62	159348	140171	0	140171
	FLAT	150296	113	150408	57	150466	122385	0	122385
Q9	**EXP**	156508	119	156628	2	156630	136009	0	136010
	FLAT	150045	110	150154	2	150156	122599	0	122599
Q10	**EXP**	160426	112	160539	5	160544	137544	0	137544
	FLAT	149312	110	149422	5	149427	122773	0	122773
Q11	**EXP**	156714	117	156832	7	156839	137007	1170	138177
	FLAT	149141	124	149266	7	149273	123315	1256	124570
Q12	**EXP**	157742	117	157859	25	157885	136764	0	136764
	FLAT	149722	115	149837	24	149861	123950	0	123950
Q13	**EXP**	159252	109	159360	3	159363	137692	0	137692
	FLAT	147960	106	148067	4	148070	123306	0	123307
Q14	**EXP**	158873	112	158985	16549	175534	136615	13764	150378
	FLAT	150737	105	150841	16462	167303	121611	13441	135052

Table 9: Results for the time of execution for the scale 100

BIBLIOGRAPHY

[1] I. Abdelaziz, R. Harbi, S. Salihoglu, and P. Kalnis. Combining vertex-centric graph processing with sparql for large-scale rdf data analytics. *IEEE Transactions on Parallel and Distributed Systems*, 28(12):3374–3388, 2017.

[2] D. Beckett, T. Berners-Lee, E. Prud'hommeaux, and G. Carothers. Rdf 1.1 turtle–terse rdf triple language. w3c recommendation. *World Wide Web Consortium (Feb 2014), available at http://www. w3. org/TR/turtle*, 2014.

[3] T. Berners-Lee. Artificial intelligence and the semantic web: Aaai2006 keynote. *W3C Web site*, 2006.

[4] C. Bizer, T. Heath, and T. Berners-Lee. Linked data - the story so far. *International Journal on Semantic Web and Information Systems*, 5(3):1–22, 2009.

[5] C. Böhm, J. Lorey, and F. Naumann. Creating void descriptions for web-scale data. *Journal of Web Semantics*, 9(3):339–345, 2011.

[6] A. Bonifati, G. Fletcher, J. Hidders, and A. Iosup. A survey of benchmarks for graph-processing systems. In *Graph Data Management*, pages 163–186. Springer, 2018.

[7] T. Bray et al. The javascript object notation (json) data interchange format. 2014.

[8] D. Brickley, M. Burgess, and N. Noy. Google dataset search: Building a search engine for datasets in an open web ecosystem. In *The World Wide Web Conference*, pages 1365–1375, 2019.

[9] V. Charpenay, S. Käbisch, and H. Kosch. Towards a binary object notation for rdf. In *European Semantic Web Conference*, pages 97–111. Springer, 2018.

[10] S. Chun, J. Jung, and K.-H. Lee. Proactive policy for efficiently updating join views on continuous queries over data streams and linked data. *IEEE Access*, 7:86226–86241, 2019.

[11] J. Cohen and M. S. Roth. Analyses of deterministic parsing algorithms. *Communications of the ACM*, 21(6):448–458, 1978.

[12] W. W. W. Consortium et al. Sparql 1.1 overview. 2013.

[13] W. W. W. Consortium et al. Rdf 1.1 concepts and abstract syntax. 2014.

[14] W. W. W. Consortium et al. Rdf 1.1 primer. 2014.

[15] A. M. D'Agostino, A. Ometov, A. Pyattaev, S. Andreev, and G. Araniti. Performance limitations of parsing libraries: State-of-the-art and future perspectives. In O. Galinina, S. Andreev, S. Balandin, and Y. Koucheryavy, editors, *Internet of Things, Smart Spaces, and Next Generation Networks and Systems*, pages 405–418, Cham, 2018. Springer International Publishing.

[16] E. Goodman, E. S. Jimenez, D. Haglin, C. Joslyn, R. Adolf, and S. al Saffar. Sprinkle sparql. Technical report, Sandia National Lab.(SNL-NM), Albuquerque, NM (United States), 2011.

[17] Y. Guo, Z. Pan, and J. Heflin. Lubm: A benchmark for owl knowledge base systems. *Journal of Web Semantics*, 3(2-3):158–182, 2005.

[18] A. Gupta and I. S. Mumick. *Materialized views: techniques, implementations, and applications*. MIT press, 1999.

[19] P. Hitzler. A review of the semantic web field. *Communications of the ACM*, 64(2):76–83, 2021.

[20] A. Hogan. Canonical forms for isomorphic and equivalent rdf graphs: algorithms for leaning and labelling blank nodes. *ACM Transactions on the Web (TWEB)*, 11(4):1–62, 2017.

[21] S. Khouri and L. Bellatreche. Lod for data warehouses: managing the ecosystem co-evolution. *Information*, 9(7):174, 2018.

[22] S. Langhi. Towards extream processing with keplr. 2020.

[23] W. Le, A. Kementsietsidis, S. Duan, and F. Li. Scalable multi-query optimization for sparql. In *2012 IEEE 28th International Conference on Data Engineering*, pages 666–677. IEEE, 2012.

[24] J. Lee, M.-D. Pham, J. Lee, W.-S. Han, H. Cho, H. Yu, and J.-H. Lee. Processing sparql queries with regular expressions in rdf databases. In *Proceedings of the ACM fourth international workshop on Data and text mining in biomedical informatics*, pages 23–30, 2010.

[25] J. Lehmann, R. Isele, M. Jakob, A. Jentzsch, D. Kontokostas, P. N. Mendes, S. Hellmann, M. Morsey, P. van Kleef, S. Auer, and C. Bizer. Dbpedia - a large-scale, multilingual knowledge base extracted from wikipedia. *Semantic Web*, 6(2):167–195, 2015.

[26] Y. Li, N. R. Katsipoulakis, B. Chandramouli, J. Goldstein, and D. Kossmann. Mison: a fast json parser for data analytics. *Proceedings of the VLDB Endowment*, 10(10):1118–1129, 2017.

[27] J. Pérez, M. Arenas, and C. Gutierrez. Semantics and complexity of sparql. *ACM Transactions on Database Systems (TODS)*, 34(3):1–45, 2009.

[28] P. Roy, S. Seshadri, S. Sudarshan, and S. Bhobe. Efficient and extensible algorithms for multi query optimization. In *Proceedings of the 2000 ACM SIGMOD international conference on Management of data*, pages 249–260, 2000.

[29] S. Siemer. Exploring the apache jena framework. 2019.

[30] M. Sporny, D. Longley, G. Kellogg, M. Lanthaler, and N. Lindström. Json-ld 1.0. *W3C recommendation*, 16:41, 2014.

[31] D. Ventura, R. Verborgh, V. Catania, and E. Mannens. Autonomous composition and execution of rest apis for smart sensors. In *SSN-TC/OrdRing@ ISWC*, pages 13–24, 2015.

[32] D. Vrandečić. Wikidata: A new platform for collaborative data collection. In *Proceedings of the 21st international conference on world wide web*, pages 1063–1064, 2012.

[33] F. Wang, P. Cui, J. Pei, Y. Song, and C. Zang. Recent advances on graph analytics and its applications in healthcare. In *Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, pages 3545–3546, 2020.

[34] Q. Wang, Z. Mao, B. Wang, and L. Guo. Knowledge graph embedding: A survey of approaches and applications. *IEEE Transactions on Knowledge and Data Engineering*, 29(12):2724–2743, 2017.

[35] Q. Yasin, Z. Xiaowang, R. Haq, Z. Feng, and S. Yitagesu. *A Comprehensive Study for Essentiality of Graph Based Distributed SPARQL Query Processing*, pages 156–170. 01 2018.

Non-exclusive licence to reproduce thesis and make thesis public

I, Juan Carlos Javier Ramos Martínez

1. herewith grant the University of Tartu a free permit (non-exclusive licence) to

reproduce, for the purpose of preservation, including for adding to the DSpace digital archives until the expiry of the term of copyright,

Quarser: a graph-aware JSON-LD Parser

supervised by Riccardo Tommasini.

2. I grant the University of Tartu a permit to make the work specified in p. 1 available to the public via the web environment of the University of Tartu, including via the DSpace digital archives, under the Creative Commons licence CC BY NC ND 3.0, which allows, by giving appropriate credit to the author, to reproduce, distribute the work and communicate it to the public, and prohibits the creation of derivative works and any commercial use of the work until the expiry of the term of copyright.

3. I am aware of the fact that the author retains the rights specified in p. 1 and 2.

4. I certify that granting the non-exclusive licence does not infringe other persons' intellectual property rights or rights arising from the personal data protection legislation.

Juan Carlos Javier Ramos Martínez
14/05/2021