

# A Framework for RDF Data Exploration and Conversion

Master Thesis

GAURAV SINGHA ROY

Riemenschneiderstr. 2, 53117 Bonn

s6gasing@uni-bonn.de

Matriculation number 2539517

Bonn, 12<sup>th</sup> January, 2015

Rheinische Friedrich-Wilhelms-Universität Bonn  
Informatik – Enterprise Information Systems  
Professor Dr. Sören Auer





# Declaration

I hereby declare and confirm that this thesis is entirely the result of my own original work. Where other sources of information have been used, they have been indicated as such and properly acknowledged. I further declare that this or similar work has not been submitted for credit elsewhere.

Bonn,

Gaurav Singha Roy





# Acknowledgement

Firstly, I would like to thank Dr. Sören Auer for his valuable time and guidance during the master thesis duration, for providing insightful and innovative solutions, and for being accessible whenever I needed. I would also like to thank Judie Attard and Dr. Fabrizio Orlandi, for their mentorship and constant guidance throughout the thesis. A special thanks especially to Judie, for her dedication and relentless help, which was a major reason for the completion of the thesis in such a small span of time.

I would also like to thank my biggest idol, my uncle, Joydip Homchowdhury, who is an innovator, an artist and a superhero. His constant guidance and vision in the past years made my life colourful with software engineering, and he had introduced me to computer science in the first place.

Last but not the least, I would like to thank my parents for their support, even when distance was such a huge barrier. And finally, a special thanks to Adela, for her patience and love which has no bounds.

# Contents

<b>Declaration</b>	<b>iii</b>
<b>Acknowledgement</b>	<b>v</b>
<b>List of Figures</b>	<b>viii</b>
<b>Abstract</b>	<b>1</b>
<b>1 Introduction</b>	<b>2</b>
1.1 Problem Context and Motivation . . . . .	2
1.2 Thesis Contributions . . . . .	3
1.3 Thesis structure . . . . .	4
<b>2 Background</b>	<b>5</b>
2.1 The semantic web . . . . .	5
2.2 Linked OPEN Data (LOD) . . . . .	5
2.3 Resource Description Framework (RDF) . . . . .	7
2.4 RDF Serialization . . . . .	9
2.4.1 RDF/XML . . . . .	9
2.4.2 Turtle . . . . .	10
2.4.3 JSON . . . . .	10
2.5 RDF Ontology . . . . .	11
2.6 SPARQL . . . . .	11
<b>3 Related Work</b>	<b>13</b>
3.1 Linked Data and REST Services . . . . .	13
3.2 RDF to Any . . . . .	14
3.3 Visualization of RDF data . . . . .	17
3.4 SPARQL Tools . . . . .	18
3.5 XSPARQL . . . . .	20
<b>4 Approach</b>	<b>23</b>
4.1 The Initial Approach . . . . .	24
4.2 RDF Exploration and Conversion API server . . . . .	25

4.2.1	RDF Exploration helper . . . . .	26
4.2.2	RDF2Any Conversion . . . . .	36
4.3	Query Builder GUI Tool . . . . .	53
4.3.1	Selecting datasets . . . . .	54
4.3.2	Exploring classes . . . . .	54
4.3.3	Properties Histogram . . . . .	57
4.3.4	Equivalent Query . . . . .	60
4.3.5	Result Set Preview . . . . .	61
4.3.6	Result Set Download . . . . .	61
4.3.7	SPARQL endpoint . . . . .	65
4.3.8	A Use Case . . . . .	65
<b>5</b>	<b>Evaluation</b>	<b>68</b>
5.1	Usability Feedback of Query Builder . . . . .	68
5.2	Evaluation of Conversion module . . . . .	70
5.2.1	JSON Conversion . . . . .	70
5.2.2	CSV Conversion . . . . .	70
5.2.3	RDB Conversion . . . . .	70
5.2.4	Generic Conversion . . . . .	70
5.3	Performance . . . . .	75
<b>6</b>	<b>Conclusion and Future Work</b>	<b>76</b>
6.1	Summary . . . . .	76
6.2	Future Work . . . . .	77
	<b>References</b>	<b>79</b>
	<b>Appendix</b>	<b>82</b>

# List of Figures

2.1	LOD Cloud in March 2009 <sup>1</sup> . . . . .	6
2.2	LOD Cloud in August 2014 <sup>2</sup> . . . . .	7
2.3	Basic RDF graph . . . . .	8
2.4	An RDF triple example . . . . .	9
2.5	RDF/XML example . . . . .	10
2.6	RDF Turtle example . . . . .	10
2.7	RDF JSON-LD example . . . . .	11
3.1	A conversion from RDBMS to RDF triples using Triplify <sup>3</sup> . .	16
3.2	R2D System Architecture <sup>4</sup> . . . . .	17
3.3	Hierarchcal facets can be used to filter information using gFacet <sup>5</sup>	18
3.4	Possible recommendations given using the SPARQL query formulation tool by the Graph Summary Project <sup>6</sup> . . . . .	19
3.5	A lifting and lowering example . . . . .	21
3.6	RDF lifting and lowering for Web service communication <sup>7</sup> . .	21
3.7	XSPARQL : schematic view and how its conceptualized from XQuery and SPARQL . . . . .	22
3.8	XSPARQL lowering example . . . . .	22
4.1	Lucene indexes stored in the project . . . . .	31
4.2	A flowchart describing the properties indexing logic in Lucene	31
4.3	Example Body text of Generic convert parsed into Body Chunks expression tree . . . . .	52
4.4	DBpedia's SPARQL Query Editor . . . . .	55
4.5	Query Builder GUI : Selecting datasets . . . . .	55
4.6	Query Builder GUI : Searching classes . . . . .	56
4.7	Query Builder GUI : After selecting a class . . . . .	56
4.8	Query Builder GUI : URI viewer . . . . .	57
4.9	Query Builder GUI : Classes histogram . . . . .	57
4.10	Query Builder GUI : Property histogram . . . . .	58
4.11	Query Builder GUI : Property filter . . . . .	59
4.12	Query Builder GUI : Select properties . . . . .	59
4.13	Query Builder GUI : Equivalent query . . . . .	60

4.14 Query Builder GUI : Equivalent query showing all the selected properties . . . . .	61
4.15 Query Builder GUI : Preview of Result Set . . . . .	62
4.16 Query Builder GUI : The result set download modal . . . . .	62
4.17 Query Builder GUI : The JSON download . . . . .	63
4.18 Query Builder GUI : The Generic download . . . . .	63
4.19 Query Builder GUI : A Sample Generic download template . . . . .	64
4.20 Query Builder GUI : A Sample Generic download template after filled by the user . . . . .	64
4.21 Query Builder GUI : SPARQL endpoint . . . . .	65
4.22 Use Case : Searching for the classes matching <i>Artist</i> . . . . .	66
4.23 Use Case : Browsing the <i>Artist</i> class histogram . . . . .	66
4.24 Use Case : Selecting the properties which the user wants for the class, <i>Artist</i> . . . . .	67
5.1 Time taken for conversions of the class <i>Person</i> <sup>8</sup> . . . . .	75

# Abstract

With the upswing increase of the use of RDF datastores, it has an increase in the number of users who want to access those data. RDF data exploration and conversion can be exhaustive and challenging. There is a lack of tools and frameworks which are needed to achieve this. Moreover, Conversion of RDF data to other formats is hard and may require the user to install a lot of softwares, if such softwares even exist. SPARQL is used mainly to access data in RDF data stores. Forming queries in SPARQL can be complex and overwhelming for even expert users. We thus propose the use of an interactive Query Builder, using which a user can formulate a SPARQL query even if they do not have any prior SPARQL experience. This thesis, therefore focuses to address the above problems as well as provide a framework with a RESTful solution which will allow developers to develop their own exploration and conversion tool by consuming those RESTful APIs. This framework can be utilized by non-expert users and also expert users, who can extend this framework easily and personalize it according to their requirements. Using the APIs from the framework, RDF data exploration and conversion is achieved. A prototype of an interactive Query builder has also been developed, which consumes these RESTful API.

**Keywords.** *RDF Exploration, SPARQL Query Builder, RESTful and Linked Data, RDF Serialization, RDF Conversion*

# Chapter 1

## Introduction

### 1.1 Problem Context and Motivation

There has been a surge in Web technologies and cloud data over the past few years. More and more users are making their data available through Linked Open Data (LOD). A common framework, Resource Description Framework (RDF) is adopted for providing meaningful semantics to the data. RDF [AG+12] was designed to provide a common way to interact with various computer applications. Since, they provide a common RDF/XML format, there can be interoperability between different systems.

The LOD data providers provide APIs for their data. These API services return data results in standard JSON and XML format. There are various tools available which allows the user to convert this data to some popular formats such as XML, JSON, Turtle, N3, etc. from RDF. Many open source tools and RESTful services are available which help the users in achieving these *RDF to any* conversions. But first, it is important that the user understands the underlying data structure. RDF has a graph structure in which the Data structure is not as rigid as its Relational Database counterparts. The data in RDF data stores can be queried using the standard RDF query language, SPARQL.

Let us consider a use case. Lets say, a user has built a Health cloud software and has an internal database which stores all the data of medicines, diseases, etc. Now this database needs to be updated on a regular basis. Lets say there is a another service, which provides this data in an RDF format by crawling through libraries like SNOMED CT<sup>1</sup>. Now, coming back to the Health cloud user. Lets say the user needs regular SQL upload scripts to update his database. Also, he is not interested in retrieving all of the data available from the RDF data source. How does he manage that ? The existing tools for conversion to RDBMS scripts from RDF usually convert the whole dataset, which in our case, the user does not want. Also, there can

---

<sup>1</sup>[http://www.nlm.nih.gov/research/umls/Snomed/snomed\\_main.html](http://www.nlm.nih.gov/research/umls/Snomed/snomed_main.html)

be a case, in which the user needs this data in a standardized interchange format, which he uses to interact with other cloud applications. How does he achieve such a generic format? These tasks can be overwhelming and challenging, and is complex for all users. Also there aren't many comparable tools and frameworks available which help in achieving this. Keeping these problems in mind, in this thesis, we strive to build a solution for it.

We can see from the problem context that exploration of RDF data is not effortless. Neither is its conversion to other data formats. And since, there is a lot of data available in RDF data sources, it becomes important that users are able to easily retrieve them. There is also a lack of tools and frameworks which help in these tasks. Therefore, the main motivation is to build a solution so that this can be achieved in a platform independent way. Because of this a RESTful approach has been taken, and a solution is made so that developers can consume those APIs to make their own conversion and exploration tools which are suitable to their requirements.

SPARQL is used for querying these data in RDF data stores. Even though SPARQL is not much tougher than SQL to master, forming a query in it can become quite overwhelming, even for expert users. This is because for forming a query, it is important to figure out exact URIs of *predicates* and *objects* which form the query. Also, it is difficult for the user to understand the data structure of the data, its relations, ontology, etc. from the first glance. Therefore, another motivation is to build an interactive Query Builder which helps in bridging these gaps.

Finally, it is important that users are able to convert the RDF data to the formats they desire. These formats include upload scripts to Relational Databases, XML, JSON, CSV, etc. If data is uploaded to the Relational databases, then the users who are comfortable with that will be able to utilize the tried and tested tools available at their disposal for them. Also, there should be a way to have a generic output, i.e., any kind of output which the users configure themselves. To attend to these issues, a state-of-the-art conversion module has been developed which is available to users and developers through RESTful services. A generic conversion has also been developed using a mini-proprietary language which allows the user to configure the download format.

## 1.2 Thesis Contributions

In this thesis, we explore a RESTful solution for achieving a better RDF exploration and conversion. The following contributions have been made

1. A RESTful solution has been made for RDF exploration and conversion. These RESTful APIs can be consumed by other softwares and tools. After a lot of research, it was concluded that there is no existing tool or framework, which solves our problem and follows the same ap-



proach. Best practices for RESTful APIs have been followed for this solution.

2. An Interactive Query Builder prototype has been developed. Using this Query Builder a user can explore the data structure of *classes* of RDF data stores, and can form his own SPARQL query. The user can extract the data in various output format he desires.
3. A powerful RDF to RDBMS conversion has been achieved which addresses the issues of the existing state-of-the-art conversion tools like RDF2RDB project [TC+07].
4. Various JSON serialization formats have been compared. Some anomalies in the existing formats have been pointed out. An experimental JSON serialization format has been suggested for RDF data extraction which removes some redundancies of JSON-LD format.
5. A proprietary mini-language has been developed using which a user is exposed to building their own download template so that they can achieve *any* kind of serialization.

### 1.3 Thesis structure

This master thesis will have the following structure :

- The first chapter has the *Introduction*. It will describe the problem statement, and the motivation for the thesis.
- The second chapter contains some literature on the *Background* information of the topics covered in the thesis. This elaborates a bit on the theoretical knowledge for the underlying topics.
- The third chapter contains the *Related Work*. This provides some existing research to the problems which we address in this thesis.
- The fourth chapter describes our *Approach* to the problem. Here the whole development process has been described including the challenges, failures, successes and the results encountered in the process.
- The fifth chapter *Evaluates* the results achieved through this thesis.
- The sixth chapter gives some more research and development suggestions as *Future work*, as part of improvement of the overall product obtained.
- The final chapter provides a *Conclusion* to the thesis.

## Chapter 2

# Background

### 2.1 The semantic web

The idea of semantic web is to have a highly interconnected data which can be both machine readable and human readable. This will pave the way for intelligent software that will utilize these semantics for more intuitive softwares. The semantic web is not much different from the World Wide Web. It is a proposed enhancement to it, which provides far more utility [FHHNS+07]. In the World Wide Web, the information is usually human readable. Since in semantic web, the idea is to make information both human readable and machine readable, it is all about adding meaning and context to the information. It will have *semantics*, *metadata* and *ontologies* in it.

Let us take an example. We all have watched the classic movie, The Godfather. How will my software find out the director of the movie? How will it find out other details like budget, running time, director of photography, etc. ? Lets go more abstract. How will it know that The Godfather is a movie ? How will it know the movie has properties like director, budget, etc. These kind of information , known as the metadata, are being added by semantic web, which we will elaborate more in the following sections.

### 2.2 Linked OPEN Data (LOD)

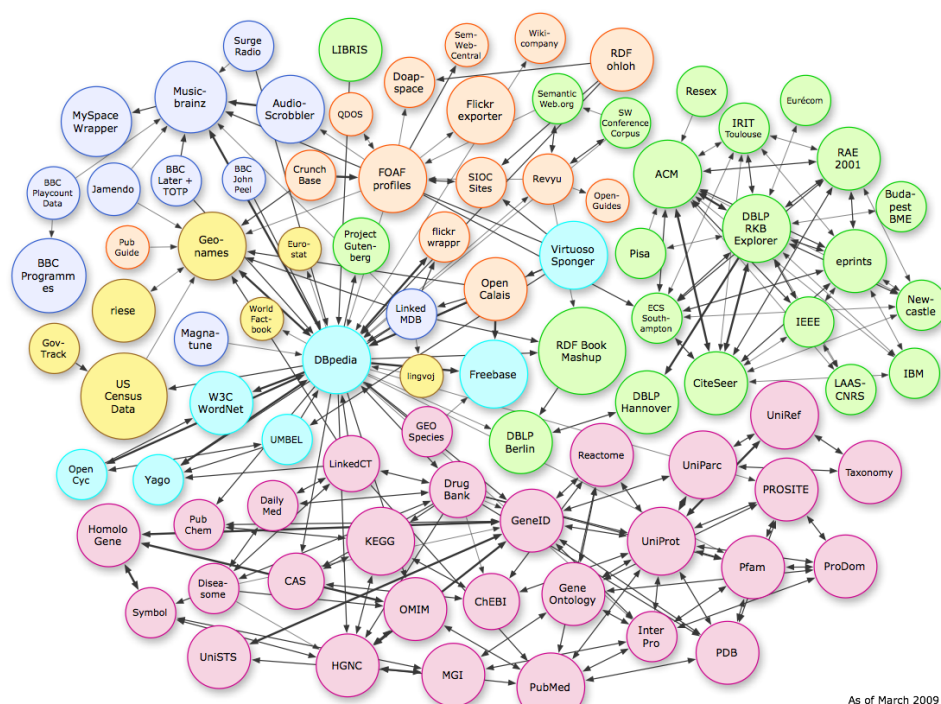
The term Linked Data refers to a set of best practices for publishing and connecting structured data on the web[BHB+08]. A lot of data providers are adopting these best practices. Due to this, over the past few years a Web of Data is being generated.

Tim Berners-Lee (2006), the inventor of the Web and Linked Data initiator, suggested a 5 star deployment scheme, which acts as a guideline for publishing open data, in a way such that this data becomes linked in the global data space[BHB+08].

The following is Tim's 5 star Open data plan :

1. The data should be available on the web (whatever format) under an open license.
2. The data should be available as structured data. eg, if there is a table, then attach an excel sheet instead of an image of the table.
3. Non-Proprietary formats should be used. e.g., CSV will be preferred more over Excel as, CSV can then be used to upload to different softwares supporting Excel.
4. URIs (Universal resource identifier) will be used as names for things. People will then be able to identify and lookup those URIs easily and point to stuff.
5. Links to other URIs should be included so that more things can be discovered

The Linked Open Data is expanding at a very fast rate. Fig. 2.1 shows the LOD cloud till March 2009. We can see in Fig. 2.2, how much the LOD cloud has expanded till August 2014.



**Figure 2.1:** LOD Cloud in March 2009<sup>1</sup>

<sup>1</sup>LOD Cloud image generated from <http://lod-cloud.net/>

<sup>2</sup>LOD Cloud image generated from <http://lod-cloud.net/>

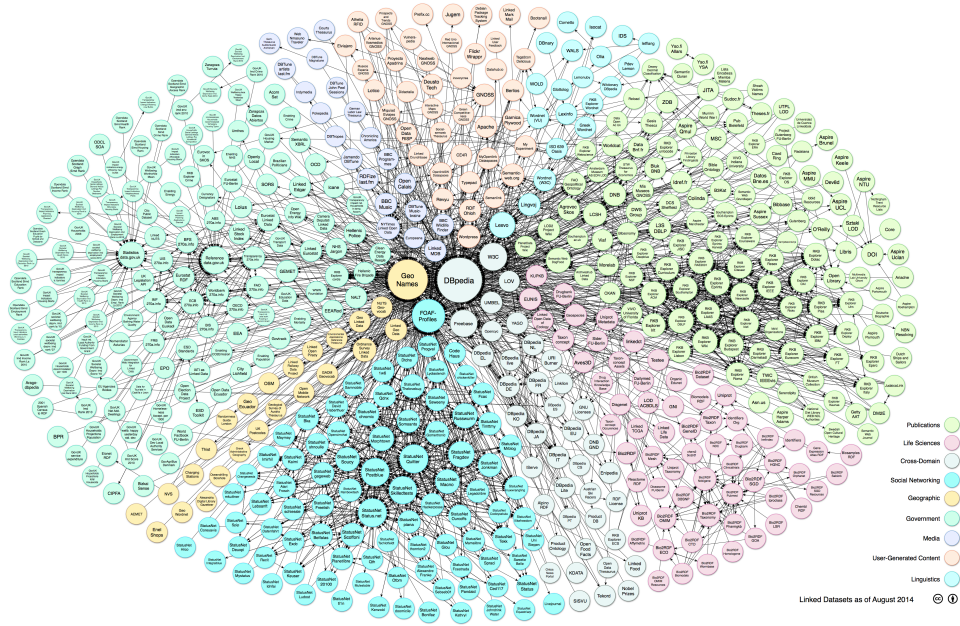


Figure 2.2: LOD Cloud in August 2014<sup>2</sup>

## 2.3 Resource Description Framework (RDF)

Initially, the world wide web was meant for humans to read. Even though everything available there was *machine readable*, it was not *machine understandable* since it did not have valuable semantics to it. It was difficult to add useful semantics to it, due to the colossal amount of data available over the World Wide Web. Therefore, it was proposed that a *metadata* should be defined for this.

**Definition 2.3.1.** *Metadata* is defined as data about data.

**Definition 2.3.2.** *Resource Description Framework* (RDF) is a foundation for processing metadata; it provides interoperability between applications that exchange machine-understandable information on the Web[LS+99].

The basic RDF model allows to represent data in the form of a directed graph. This allows in representing named properties and property values. This RDF model, represents data drawing from well established data modelling techniques such as *entity relationships* or *class diagrams*. The attributes of the resources become the RDF properties, and therefore the RDF model will resemble a lot like *entity-relationship* diagrams. In object oriented design terminology, the resources will correspond to objects and properties will correspond to instance variables[LS+99]. We will now formally define the *RDF Data Model*.

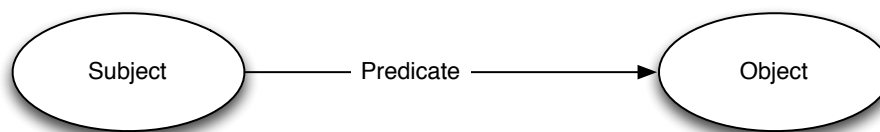
**Definition 2.3.3.** *RDF Data model* is a syntax-neutral way of representing RDF expressions [LS+99].

The RDF data model has three object types :

*Resources.* All *things* being described by RDF expressions are called *resources*[LS+99]. In the context of object modelling, these represent the objects or a collection of objects. The resources can be either URIs, *literals* or *blank nodes*. Some resources will have a unique identifier which provides a unique identity to it. In an RDF Data model it is done by a *Uniform Resource Identifier* (URI). The concept is, anything can have a URI. This opens up the possibilities for data in different sources to have links. *Literals* represent quantitative value eg, first name, weight, height, etc (values of string, boolean, integer, etc.). An RDF *blank node* is an RDF node that itself does not contain any data, but serves as a parent node to a grouping of data[MSDN].

*Properties.* A *property* is a specific aspect, characteristic, attribute, or relation used to describe a resource[LS+99]. In the context of object modelling, properties represent the relationships. This can also describe permitted values.

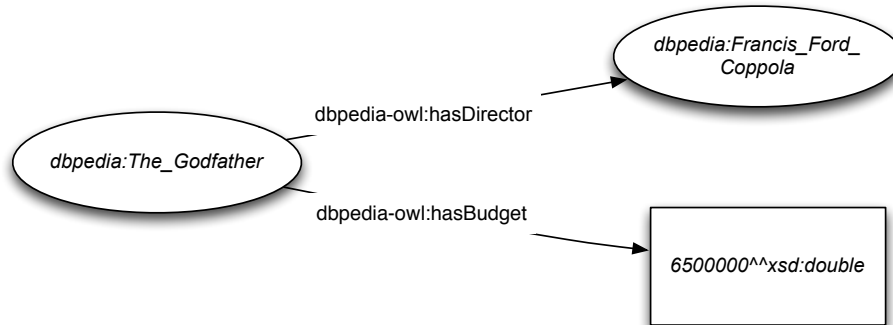
*Statements.* A specific resource together with a named property plus the value of that property for that resource is an RDF *statement*. In RDF, there are three individual parts of the statement, viz, *subject*, *predicate* and *object*[LS+99]. Predicates are the properties which we just described. The subjects are always resources. The objects can be either resources or some quantitative value called *Literals*. Fig. 2.3 shows how these statements make up a basic RDF graph. Together, the *subject*, *predicate* and *object* are called *triples*.



**Figure 2.3:** Basic RDF graph

Let us take an example. We would like to represent the movie *The Godfather* in an RDF graph. We will represent, the movie's Director, Francis Ford Coppola and also the budget of the movie. We have drawn the couple of *triples* required in fig. 2.4<sup>3</sup>. In the first *triple*, we will represent the information that the movie *Godfather* has director Francis Ford Coppola. We can see in the fig. 2.4, this is represented as "dbpedia:The\_Godfather

<sup>3</sup>Example triples taken from DBpedia[dbpedia-swj]



**Figure 2.4:** An RDF triple example

dbpedia-owl:hasDirector dbpedia:Francis\_Ford\_Coppola". dbpedia:The\_Godfather is the subject here, with predicate dbprop:hasDirector and object dbpedia:Francis\_Ford\_Coppola. "dbpedia" is a prefix for the namespace which has URI "http://dbpedia.org/resource/". The actual URIs for the subject dbpedia:The\_Godfather is http://dbpedia.org/resource/The\_Godfather, the predicate dbpedia-org:hasDirector is http://dbpedia.org/ontology/director and the object dbpedia:Francis\_Ford\_Coppola is http://dbpedia.org/resource/Francis\_Ford\_Coppola. We can see in this, the object is a resource with a URI. Now we will try to form a triple where the object is a quantitative value, i.e., a Literal. In fig. 2.4 the triple "dbpedia:The\_Godfather dbpedia-owl:hasBudget 6500000^^xsd:double" represents, the budget of the movie. The object here is a literal of value double. This means that the movie Godfather has a budget of \$6.5 million.

## 2.4 RDF Serialization

Several common serialization formats exist for RDF.

### 2.4.1 RDF/XML

The RDF Graph model can be serialized in an XML format along with its semantics. An RDF triple, as discussed in the previous section will have three components, viz., *subject node*, *predicate node* and *object node*. These nodes, of course will be URIs, literals or blank nodes. In order to encode this RDF graph data in XML[GS+14], the nodes and predicates have to be represented in XML terms. Fig. 2.5 shows how the RDF example given in the previous section (fig. 2.4) can be serialized.

```

<?xml version="1.0"?>
<rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
  xmlns:dbpedia="http://dbpedia.org/resource/" xmlns:dbpedia-owl="http://
  dbpedia.org/ontology/">
  <rdf:Description rdf:about="http://dbpedia.org/resource/The_Godfather">
    <dbpedia-owl:hasDirector rdf:about="http://dbpedia.org/resource/
    Francis_Ford_Coppola"/>
    <dbpedia-owl:hasBudget>6500000</dbpedia-owl:hasBudget>
  </rdf:Description>
</rdf:RDF>

```

Figure 2.5: RDF/XML example

### 2.4.2 Turtle

Turtle stands for Terse RDF Triple language [BBPC+14]. Turtle document allows representing the RDF graph data content in a compact textual content. This representation is both machine readable and human readable and provides a much simpler human readability than other serialization formats. Fig. 2.6 shows how the RDF example given in the fig. 2.4 can be serialized.

```

@prefix dbpedia-owl:<http://dbpedia.org/ontology/>.
@prefix dbpedia:<http://dbpedia.org/resource/>.
@prefix xsd:<http://www.w3.org/2000/01/rdf-schema#>.

```

```

dbpedia:The_Godfather dbpedia-owl:hasDirector dbpedia:Francis_Ford_Coppola.
dbpedia:The_Godfather dbpedia-owl:hasBudget "6500000"^^xsd:double.

```

Figure 2.6: RDF Turtle example

N3 or Notation3 is also a similar format [BC+11]. Turtle is basically a successor of N3.

### 2.4.3 JSON

JSON stands for JavaScript Object Notation. It is a very lightweight data - interchange format which is nowadays becoming the de facto format to get data from REST APIs. The popularity of JSON can be attributed, to the fact that in any programming language, the JSON data becomes one big Hash Map, which has key value pairs for data. This makes the life of programmers much simpler, as they do not have to grind over the cumbersome parsing of heavier formats like XML.

W3C already has a standard JSON format for retrieving queried SPARQL result set. A new specification, called the JSON-LD which stands for JSON Linked Data has been proposed and is undergoing review by W3C members

[SLKLL+14]. Fig. 2.7 shows how the RDF example given in the previous section (fig. 2.4) can be serialized into JSON-LD.

```
{
  "@context": {
    "director": "http://dbpedia.org/resource/hasDirector",
    "budget": "http://dbpedia.org/ontology/hasBudget"
  },
  "@id" : "http://dbpedia.org/resource/The_Godfather",
  "director" : "http://dbpedia.org/resource/Francis_Ford_Coppola",
  "budget" : 6500000
}
```

Figure 2.7: RDF JSON-LD example

## 2.5 RDF Ontology

In the context of Web semantics and knowledge sharing, ontology refers to the definition of concepts and its relationships.

**Definition 2.5.1.** *Ontology* is a specification of a conceptualization. In the context of knowledge sharing, ontology means a specification of a conceptualization[G+93].

**Definition 2.5.2.** *Vocabulary* defines the concepts and relationships, used to describe and represent an area of concern[w3c+ontology].

Defining "ontologies" or "vocabularies" organises the knowledge base. They are used to classify the terms that can be used for some particular applications, define possible relationships and constraints. These can get very complex. There is not such clear differences between "ontologies" and "vocabularies". In general, ontologies are more complex and is a formal collection of terms. Vocabularies on the other hand is not that formal.

If we try to find out the ontology of the RDF example mentioned in the previous section (fig. 2.4), we notice that dbpedia:The\_Godfather is of type say dbpedia:Movie. Now we can define ontologies in the :Movie, that it can have a director, viz., dbpedia-owl:hasDirector. This is a relationship which we are defining in the ontology. It gives more structure to the data.

## 2.6 SPARQL

SPARQL is a W3C recommended query language for querying data in RDF datastores[PS+13]. SPARQL is a rule based language in which it returns results based primarily on the triple rules which has been put in the query, ie, *subject*, *predicate* and *object*.



This is a simple SPARQL query in which am searching for subjects which have an object cat.

```
1 SELECT ?s WHERE { ?s ?p "cat" }
```

Now going back to the example (fig. 2.4), We will need a SPARQL query something like

```
1 SELECT ?o WHERE {  
2   <http://dbpedia.org/resource/The_Godfather> ?p >o.  
3 }
```

Here, the subject is the Godfather, and I will retrieve all the objects of the Godfather.

## Chapter 3

# Related Work

In this chapter, we will discuss some research which has been done in the Exploration and Conversion of RDF data. There has been a lack of comparable tools and frameworks on the lines of our approach. Since our main motivation is to provide better exploration and conversion of RDF data through RESTful API services, it is important to research on overlapping of interests of RESTful services and Linked Data. We will then move on to *RDF to Any* conversions with focus on RDBMS conversions. Since for exploration, visualization is important, we will also focus on some research done in the field of Visualization of RDF data. As one of our motivations is to provide help for non-native SPARQL users, we will explore some research done for assisted SPARQL query formulation. And finally, because we want *RDF to Any* conversion, we describe an interesting research, which suggests a new language called XSPARQL [AKKP+08]. This provides an interoperability from RDF data to XML and vice versa.

### 3.1 Linked Data and REST Services

There seems to be an obvious alignment between Linked data and REST services. An insightful paper has been published which debates whether REST and Linked Data are suitable for domain driven development [PRM+11]. In Linked data, the data is located using unique identifiers called URIs which is used for either lookup or for data retrieval. The common interests for both of them are [PRM+11]

1. The key information which is to be extracted in both REST and Linked data is a resource
2. In both of them Linking is optional. It is not always necessary to include links such that more information can be retrieved using URIs. The URIs can be used just for unique identification.
3. A big misapplication in both the cases is to assume that semantics can be encoded in a REST url or in Linked Data, its URI. They are

used just for identification and retrieval purposes and semantics are not implied implicitly.

4. Both of them are designed in a way that they are adaptable. Modification of resources are immediately reflected as resources are retrieved through either REST urls or in Linked Data through URIs.
5. Through Domain Driven Design [E+04], the focus is mainly on iteratively developing domain modeling. This is practiced and outlined both in REST and Linked Data designs.

Even though there are common interests in both of them, there are differences also. One underlying difference is when you compare API vs. Model [PRM+11]. While REST services do not clearly expose the relationships, and those are exposed only if the software allows to, but in Linked Data, this is exposed through the ontologies defined. Content negotiation is a major differentiating factor. RESTful services have a negotiation with the client so that the client knows what kind of data to send and what it will retrieve. In Linked Data such content negotiation is yet to be achieved and standard serialization formats are being defined. And finally, in Linked Data, the data is mostly read-only and delete, update actions are usually not permitted. In RESTful these are done by POST, DELETE AND PUT actions.

From this research we can conclude that REST and Linked Data can have a harmonious future. The RESTful services and Linked Data can complement each other to enable services like eResearch [PRM+11]. The key is to allow domain experts, researchers and developers to collaborate and work in harmony to share the information. Now Linked Data, using RDF, provides a unique structure to the data which eases the path for information sharing. RESTful services, can be used to access this information. Developers can even update, add, and delete these information using RESTful services. Using this harmonious approach, both of these can provide a common interface, so that Linked Data is unanimously accessible to the users.

## 3.2 RDF to Any

In this section we will explore some research which has been done in the tools which aid in conversion of RDF formats to various other popular formats. There are some tools available which enable this but our research focus mainly lies in two areas. *Firstly*, we would like to explore more into conversion to RDBMS scripts. There hasn't been much research in this field. We will discuss this in detail, later in this section. *Secondly*, we wanted an approach using which a user can configure his own output format. This approach cannot be comparable to existing tools, and the only research which came close to it is the suggestion of a language called XSPARQL [AKKP+08], which allows the user to extract the RDF data into whatever kind of XML format they desire. This has been discussed more in detail in

the next section.

There has been considerable research for serialization of RDF data to other formats. XML, JSON, Turtle, N3, etc. are some commonly available formats. The RDF Translator [SRH+13] project is one such project in which an RDF graph can be downloaded to the above mentioned popular formats using REST services. This RDF Translator takes an input of an RDF graph source and, then the user can select from various popular output formats like XML, Turtle, N3. The advantage of this tool is that apart from a Browser based UI, it also provides REST services for these conversions. But, the problem with such kind of tools is that, you will have to convert the whole graph or the datastore. Now if the user needs to convert just a subset of the dataset, then he has to rely on SPARQL to extract those subsets. Usually datastores have SPARQL endpoints using which the user can interact with them. Datastores like DBPedia<sup>1</sup> [dbpedia-swj], have APIs for converting their RDF data to formats like HTML, XML, JSON, N3, Turtle, etc. But a problem with extracting data from SPARQL Query Result sets is that, it is usually in a very one-dimensional tabular form. You only get those information which you have put in the *SELECT* statement of SPARQL.

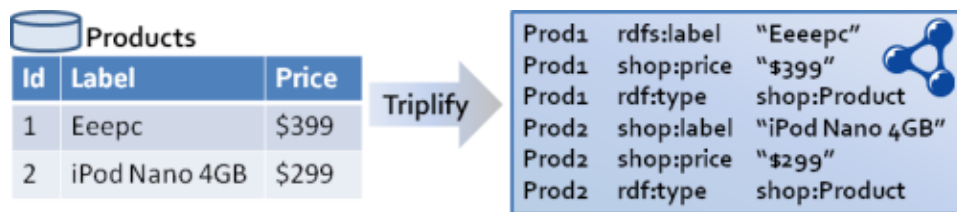
Now, we will focus more on the difficult conversion, i.e., to RDBMS scripts. With the emergence of so much data in the RDF, it becomes important to export those data into Relational Databases. There are various age old tried and tested tools for visualization, querying, logistics, etc. available for RDBMS systems, which the user would want to utilize. They can do that for RDF data only if they have the data available in Relational Database format. Hence conversion to RDBMS from RDF, becomes vital. This conversion is not an easy task. It becomes important to address various issues like, data structures, normalization, relations using foreign keys, etc. This approach is unique and has no comparable counterparts. Even though there hasn't been much research in this field, there has been some research and efforts to achieve this. This is difficult of course as, in RDF, the structure is very flexible compared to the rigid structure of RDBMS.

First let us look into some research in which Relational databases are mapped to RDF Graphs. D2RQ [BS+04] platform provides an interesting tool in which it takes a relational mapping as an input and provides a virtual RDF graph as an output. It contains a declarative mapping language D2RQ Mapping Language [BS+04] for describing the relation between an ontology and a relational data model. It also contains a useful D2RQ Engine [BS+04], which is a plug-in for Jena toolkit, using which mappings in Jena API can be rewritten for calls to SQL queries against the database. There is a similar tool, Virtuoso RDF Views [EM+07], which does the same thing. Triplify [ADLHA+09] is a very commendable effort in mapping relational databases to RDF graphs, JSON and linked Data. It is very lightweight and

---

<sup>1</sup><http://dbpedia.org/>

has been implemented using less than 500 lines of code. It uses very simple concepts, eg, it will create properties based on column names. Fig. 3.1 shows how the columns *label* and *price* is converted to properties *rdfs:label* and *shop:price*. These concepts have been used by us to do the tranformation from RDF to RDBMS. Another notable effort has been RDF123 [HFPSJ+08], which tranforms spreadsheets to RDF data.



**Figure 3.1:** A conversion from RDBMS to RDF triples using Triplify<sup>2</sup>

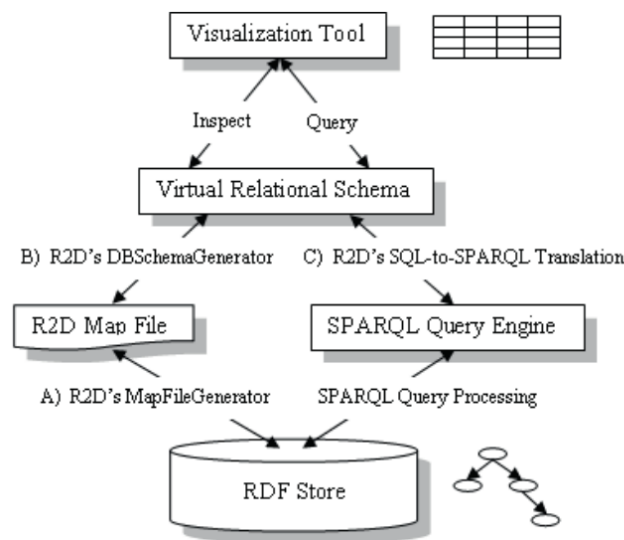
Some more research which is similar to what we require is conversion from RDF to RDBMS. RDF2RDB project [TC+07] is the most promising one in that field. It is an open source free to use tool developed in Python. It converts RDF Data from the Semantic web to relational databases. It creates SQL scripts which works in various versions of MySQL<sup>3</sup> database. The most essential part is the schema which has to be generated for the tables. So first, what it does is, it creates tables of all the *classes* for which the thing (or object) belongs. If the class is unknown then the thing is added to the table *things*. A table consists of at least two columns, viz., a primary key, and a column for *uri*. It will then create tables for both *ObjectType* and *DataType* in which these properties are mapped. These tables are named like `<class_name>_<property_name>_<datatype>`. Here we can see two issues. One is in the naming convention there is no need for *datatype*. Also, it always creates a new table for each property. For one-to-many relationships, this is fine and is necessary for normalization, but for one-to-one relationships this is inefficient, as it creates an unnecessary table. We have overcome this problem and we take into account whether the property is one-to-many or one-to-one before creating a schema for a new table. Otherwise we add the property as a column to the same class table. This software is efficient as it only stores the schema in memory and the rest data is printed on the fly. Another research R2D [RGKST+09] also converts RDF to virtual relational databases. The research's main motivation was to visualize RDF data, but they achieve so by utilizing various visualization tools available for RDBMS systems. For that reason they convert the RDF data to virtual RDBMS data. We will discuss more about this research in the following section.

<sup>2</sup>Image taken from <http://triplify.org/>

<sup>3</sup>MySQL boasts of being the most popular open source relational database <http://www.mysql.com/>

### 3.3 Visualization of RDF data

Visualization of RDF data is essential as it gives the user more information on the relations and the ontology of the data. Various tools are available which enable the user with this. One interesting and yet uncomplicated research is the R2D [RGKST+09] project. In this, they approach with a simple method to convert the underlying RDF data to RDBMS. Now once it is achieved, the user can utilize the numerous visualization tools available at their disposal for viewing Relational Data. The main logic behind is, it creates an R2D Map file which contains the underlying schema for the RDBMS conversion. Now using that map file, a virtual relational schema is formed and the data is viewed using standard visualization tools for relational databases. The advantage of this tool is that it does not duplicate any data. Fig. 3.2 shows the system architecture of the project.



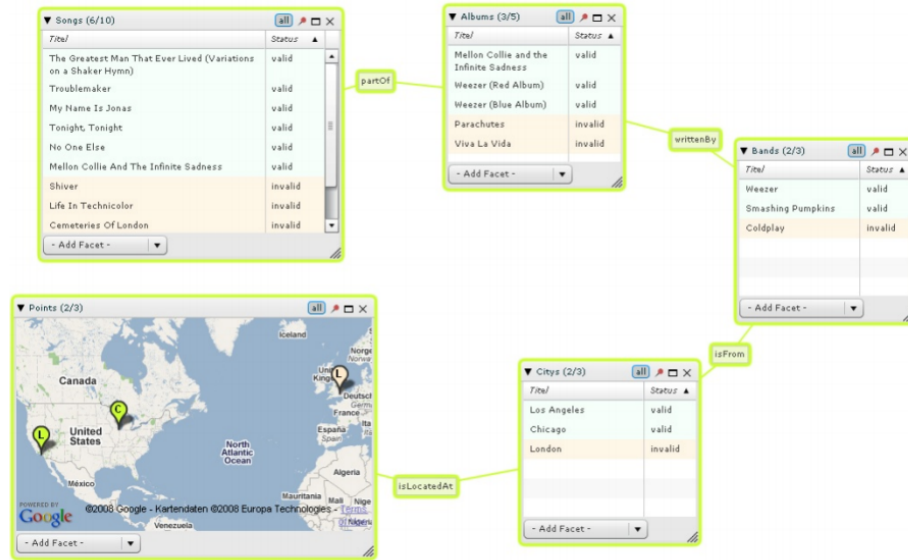
**Figure 3.2:** R2D System Architecture<sup>4</sup>

Another notable research is the Visual Data Web project<sup>5</sup>. In this project they have developed various tools which aid in Data visualization for Semantic Data. Their tools are commendable and almost require no installation as each one of them runs on web browsers. The only criticism is that, it relies heavily on flash, which is not a very popular library these days and is slowly losing support with many devices. One of the tools developed by them is gFacet [HZL+08]. Using this tool the user can create various facets to explore the data structure of the underlying RDF graph. Fig. 3.3 shows how hierar-

<sup>4</sup>Image taken from the paper [RGKST+09]

<sup>5</sup><http://www.visualdataweb.org/>

cheal facets can be used to filter information and more data can be retrieved or visualized using a simple GUI tool. These facets can be added or removed interactively using the tool. Another tool is the SemLens [HLTE+10]. Using



**Figure 3.3:** Hierarcheal facets can be used to filter information using gFacet<sup>6</sup>

this, trends and correlations in RDF data can be analysed. This provides a visual interface to combine scatter plots with semantic lenses. The scatter plots provide global visualizations which can be drilled down further using the semantic lenses. Another tool is the RelFinder [HLSZ+10]. This helps the user to garner some overview of the data. The user can visualize the data structure of the objects in the RDF graph using a visual tool.

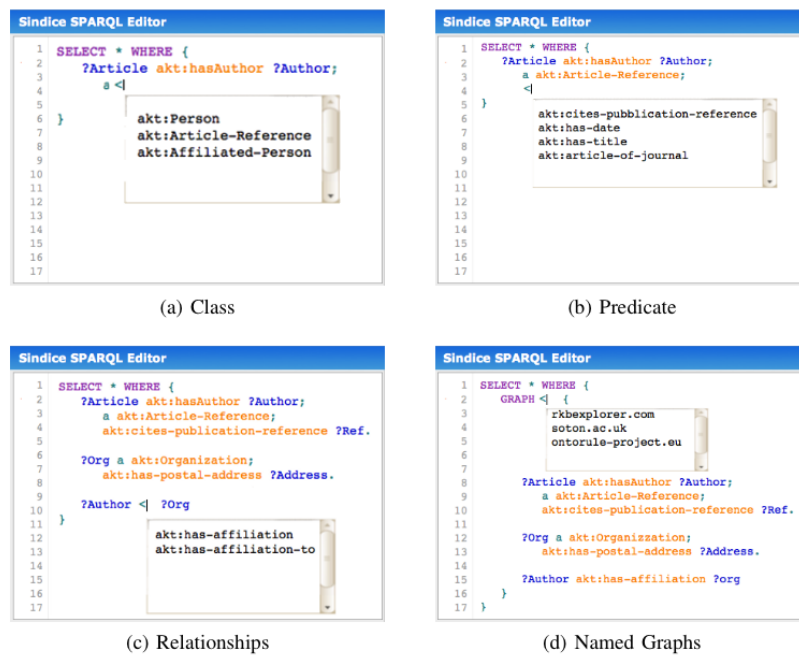
There are much more tools available for visualization. Our main focus of the thesis is not exactly visualization but to provide APIs using which such tools can be built. This is discussed more in the next chapter for Approach.

### 3.4 SPARQL Tools

SPARQL is not a simple query language to master. Because of its complexity, it has a slow adoption with the users. It is important to note that, the language itself is not more complex than, say SQL, but it still is challenging as a lot of exact URIs are needed for matching properties and filters. For these reasons, it is important to have tools at the user's disposal, which will aid him in SPARQL query formulation. This task is challenging, and complex, and some research has been made in this field.

<sup>6</sup>Image taken from the paper [HZL+08]

There has been an open source project for Graph Summary [CPCDT+12] which has an application for Assisted SPARQL Formulation. Using the tool, the user is able to formulate his queries without needing to individually explore each object's data. This will reduce the time, the user needs to study the data in order to query it. How it works is, it first builds up a graph summary data by reading through all the triples of a graph. Now when the user starts typing in the SPARQL query text editor (fig. 3.4), then recommendations show up instantly for *classes*, *predicates*, *relationships* and *named graphs* (fig. 3.4). The drawback we can see with this approach is that



**Figure 3.4:** Possible recommendations given using the SPARQL query formulation tool by the Graph Summary Project<sup>7</sup>

it is not useful for users who have absolutely no experience with SPARQL. This will still require the user to know the correct syntax of a SPARQL query and how to form a correct query.

Some research has also been done for interactive query builders for SPARQL queries in which the user does not require any knowledge of SPARQL. There is a project, QueryMed [SS+10], which provides an intuitive GUI Query Builder which can be used to query Biomedical RDF Data. There is another Query Builder tool, which formulates SPARQL queries for Drupal<sup>8</sup> [C+10]. Apart from these, there are some freely available online

<sup>7</sup>Image taken from the paper [CPCDT+12]

<sup>8</sup><https://www.drupal.org/>



tools for some RDF data sources, which allows the user to formulate their SPARQL query. They go with the simple approach, to select *classes*, and then proceed with filtering *properties*. The advantage of these tools is that, the user does not need to know SPARQL. Even non-native SPARQL users can successfully build a query. The disadvantage we came across in these tools is that, in most of them, they are not as visually powerful as a user would want. They do not reveal the data structure information to the user. So even if the user has access to UI functionalities to build a query, he does not know what is the data structure. Keeping these pros and cons in mind, we opted for this approach and have built a simple Query Builder which has all these features. We also provided a class histogram, property histogram, which provides a visual information on the underlying data structure. This has been discussed more in detail in the next chapter, in section 4.3.

### 3.5 XSPARQL

For a generic serialization, we will probably require a small programmable language using which the user can build his own serialization template. This, of course, is tricky and requires some learning curve, and it is difficult to find some comparable research. XSPARQL [AKKP+08], is one such research which provides an interoperability between the two worlds of RDF and XML.

XSPARQL [AKKP+08] stands for extensible SPARQL. XSPARQL was submitted for official W3C recommendation in 2009 and its approval is still pending. It provides a common language for querying RDF data using SPARQL like querying rules and this will retrieve output to either RDF or XML. Alternatively it can query XML data using XQuery<sup>9</sup> like rules and this will retrieve output to XML and SPARQL. It provides an interesting interoperability between these two data formats using one common language.

It is clear that the main purpose of building this language was to provide an easier lowering and lifting for RDF.

**Definition 3.5.1.** *Lifting* refers to extracting RDF from XML.

**Definition 3.5.2.** *Lowering* refers to extracting XML from RDF.

Fig. 3.5 gives an example of lowering and lifting. The web service architecture to achieve this has been summarized in fig. 3.6.

Fig. 3.7 shows a schematic view of XSPARQL. We notice that all three languages here have 3 parts, viz., The *prolog*, the *head* and the *body*. Fig. 3.8 shows how *lowering* can be achieved using XSPARQL. Here the *prolog* stands for any global declaration we need for the rules we are using. The

---

<sup>9</sup>XQuery is a language used to query XML data

<sup>10</sup>Image taken from the paper [AKKP+08]

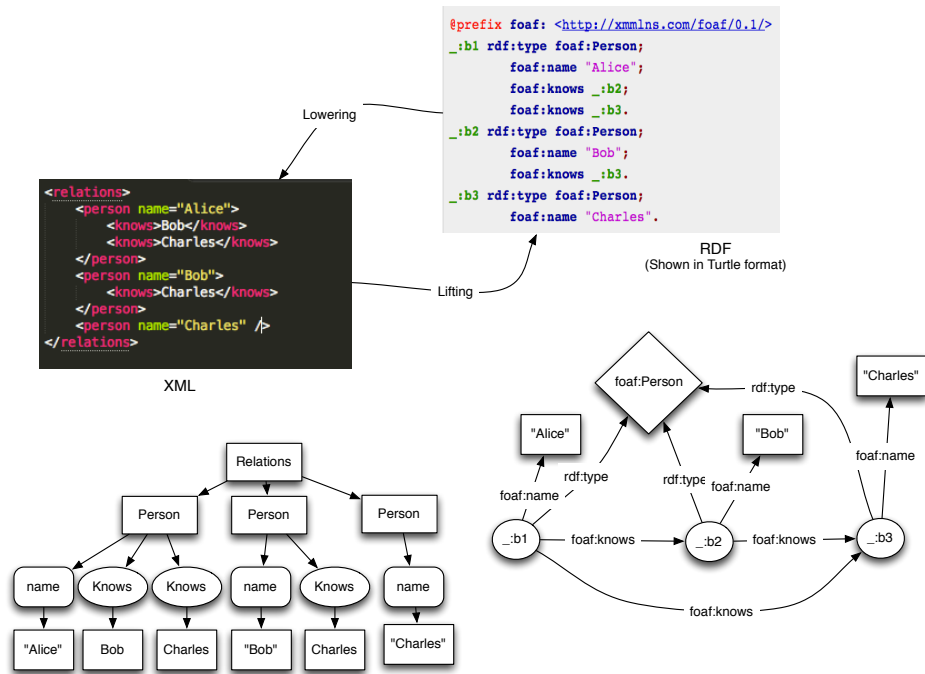
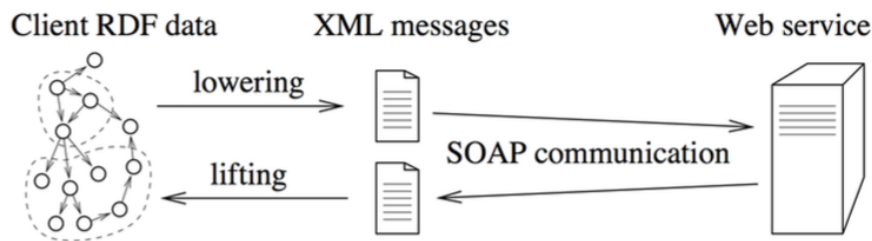


Figure 3.5: A lifting and lowering example

Figure 3.6: RDF lifting and lowering for Web service communication<sup>10</sup>

*head* is the return values of the query. It will return data in the format we want. The *body* is the main part of the query in which we form rules on what to return. In the language XSPARQL, what the authors have proposed is that they can use a combination of the 3 parts of both SPARQL and XQuery to have an output format which they desire.

We notice that from the example in fig. 3.7 we can achieve any kind of XML serialization of RDF data using XSPARQL. But there is a problem with that. For such kind of conversion the user still needs to know SPARQL.

Prolog	P	<b>declare namespace</b> prefix = "namespace-URI"
Body	F L W O	<b>for var in</b> XPath-expression <b>let</b> var := XPath-expression <b>where</b> XPath-expression <b>order by</b> XPath-expression
Head	R	<b>return XML+nested XQuery</b>

Schematic view on XQuery

Prolog	P	<b>prefix</b> prefix: <namespace-URI>
Head	C	<b>construct</b> [template]
Body	D W M	<b>from / from named</b> <dataset-uri> <b>where</b> (pattern) <b>order by</b> expression <b>limit</b> integer > 0 <b>offset</b> integer > 0

Schematic view on SPARQL

Prolog	P	<b>declare namespace</b> prefix = "namespace-URI" or <b>prefix</b> prefix: <namespace-URI>
Body	F L W O	<b>for var in</b> XPath-expression <b>let</b> var := XPath-expression <b>where</b> XPath-expression <b>order by</b> XPath-expression
	F' D W M	<b>for varlist</b> <b>from / from named</b> <dataset-uri> <b>where</b> (pattern) <b>order by</b> expression <b>limit</b> integer > 0 <b>offset</b> integer > 0
Head	C	<b>construct</b> [template (with nested XSPARQL)]
	R	<b>return XML+nested XQuery</b>

Schematic view on XSPARQL

**Figure 3.7:** XSPARQL : schematic view and how its conceptualized from XQuery and SPARQL

Prolog	P	<b>declare namespace</b> prefix = "namespace-URI" or <b>prefix</b> prefix: <namespace-URI>
Body	F L W O	<b>for var in</b> XPath-expression <b>let</b> var := XPath-expression <b>where</b> XPath-expression <b>order by</b> XPath-expression
	F' D W M	<b>for varlist</b> <b>from / from named</b> <dataset-uri> <b>where</b> (pattern) <b>order by</b> expression <b>limit</b> integer > 0 <b>offset</b> integer > 0
Head	C	<b>construct</b> [template (with nested XSPARQL)]
	R	<b>return XML+nested XQuery</b>

```

declare namespace foaf = "http://xmlns.com/foaf/0.1/";
<relations>
{ for $Person $Name from <http://xsparql.deri.org/data/relations.rdf>
  where { $Person foaf:name $Name }
  order by $Name
  return <person name="{ $Name }">
    { for $FName from <http://xsparql.deri.org/data/relations.rdf>
      where { $Person foaf:knows $Friend.
              $Person foaf:name $Name.
              $Friend foaf:name $FName. }
      return <knows> { $FName }</knows>
    }
  }
</person>
}
</relations>

```

```

<relations>
  <person name="Alice">
    <knows>Charles</knows>
    <knows>Bob</knows>
  </person>
  <person name="Bob">
    <knows>Charles</knows>
  </person>
  <person name="Charles"/>
</relations>

```

**Figure 3.8:** XSPARQL lowering example

Not only that, he needs to know XQuery. Now understanding both these languages results in a steep learning curve. Moreover, probably for most of the conversions we do not need so many complicated features presented here. What we need probably some simple programmable features which will help us in achieving *any* conversion from RDF. In the next chapter for *Approach*, we will take up the good things of XSPARQL and come up with a simpler language, which will aid the user to build his own serialization output.

## Chapter 4

# Approach

We discussed in the previous chapter various research which has been done related to what we want to achieve in our thesis. But most of the applications do not specifically solve our problem. We need a lightweight solution to solve many of these problems.

1. We need a visual tool to explore the data structure of various classes in a particular dataset. For this we need various information for the data structure like subclasses, properties, count of objects etc. This can be easily achieved through SPARQL. But then again, it has a learning curve as understanding rule based languages is not as simple even for expert users. Moreover it requires a lot of exact URI lookup, which can become cumbersome, even for an expert SPARQL user.
2. We need the visual tool to enable a user to form his own SPARQL query without having any prior knowledge of SPARQL. This can be achieved by providing a responsive GUI, using which the user can select properties, add filters for those properties, etc., using some drag-and-drop options.
3. We need a smarter conversion of RDF data to RDBMS data. So far we saw in the previous chapter, there are tools which do this (section 3.2), require the whole RDF dataset. We need a solution in which the conversion is smarter for a subset of the RDF dataset, which the user most probably requires.
4. We need to address a smarter JSON-LD conversion reducing a lot of data redundancy and inefficiency which can arise due to the way how keys are defined in the JSON output structure.
5. An RDF to any conversion has to be attempted. A solution has to be provided, which is simple and easy to use and will allow the user to convert to any kind of serialization he wants.
6. Finally, We have to make our software interoperable. All these has to be achieved in such a way that any software developer can utilize this

exploration tool and build his own using GUI tool very easily. In short, we had to achieve this in such a way that standard APIs are provided which can be consumed by any application.

## 4.1 The Initial Approach

At first we opted for an approach in which we will provide a universal API for all the developers using standard REST API principles. We needed to build an API service. This API service will be consumed by tools so that people without any knowledge of SPARQL will be able to use those tools.

We proposed a simple REST action which would require the following parameters

- *dataset* - This will be the URL of the SPARQL endpoint of the dataset
- *classes* - This will be a list of classes (comma separated) which we want to download.
- *property\_filters* - property filters would have been passed (comma separated) through this.
- *output\_format* - This will specify the output format, viz., *CSV*<sup>1</sup>, *RDB*<sup>2</sup>, *JSON*, etc.

Now a full fledged REST action for this can look like<sup>3</sup>

```
1 GET /v1.0/convert?dataset=http://example.dataset &
  ↪ classes=http://example/resource/Person &
  ↪ property_filters=height>180,weight<80 & output_format=RDB
```

We notice from the above list that, there is no parameter for a SPARQL query. So that means a user not familiar with SPARQL can easily use it. The input parameters *dataset*, *classes*, *output\_format* all seem to be alright and in place. Defining the *property\_filters* seems to be a bit tricky. A sample value of *property\_filters* can be

```
1 property_filters=height>180,weight<80
```

There are a couple of disadvantages for this parameter. *Firstly*, the user needs to again learn proper formats on how to send these parameters. Even though the learning curve for this task is not that steep as learning a new language, but its still some learning curve. *Secondly*, it is assumed that the various properties here is comma separated, so how will we specify if the filter is actually an OR filter and not just AND filters. We can have very powerful AND filters but it does not allow the user to have OR features which will not give the user the flexibility he desires. Can we also have nested filters using this ? Now nested OR and AND can also be achieved by having a complex structure to the parameter's value like

<sup>1</sup>Comma separated values

<sup>2</sup>RDBMS SQL upload scripts

<sup>3</sup>The URL is not shown with proper URL encoding for better readability

```
1  property_filters=(height>180)||((height<170)&&(weight<60))
```

But again, this will mean that the user will have to understand this format properly.

One important demographic of users which won't benefit from this, will be the users who are familiar with SPARQL and do not want to learn these new formats. How do we deal with such requests ? Because of such issues this approach was abandoned and a two-way approach was suggested. The issues are summarized below

1. **New format for property filters.** The property filters need to be sent in a new format. This will require the user to again master something non-standard and new.
2. **Nested property filters difficult.** Using this approach, nested property filters will be difficult to handle. By default, this approach would have just handled AND operators among multiple filters.
3. **No support for SPARQL.** Since this approach does not need SPARQL, users already familiar with SPARQL would not have benefited much with this approach.

Now, let's come to the proposed final two-way approach. *Firstly*, A full fledged API server was made with complete REST API which had various actions like searching of classes, helping in filters, conversion, etc. Now the input parameter *property\_filters* has been discarded and instead a parameter *query* is passed. This will be a SPARQL query. Now how will a user who does not know SPARQL form the query?

For this we came up with another part. We built a GUI tool which will consume the actions by our exploration API server. In this the user will explore the RDF data, perform various actions like selecting filters using GUI tools and our library will build an equivalent sparql query. So both native and non-native SPARQL users can use the same REST API. Also, this means that, anyone can build their GUI tool for this using the above APIs.

## 4.2 RDF Exploration and Conversion API server

This will provide 2 main tasks viz., RDF Exploration helper and RDF2Any Conversion. To provide both these tasks all actions are provided as REST API services so that it takes care of software interoperability.

**Technologies used** - The following technologies/libraries have been used to implement this portion

- **Java** - This software is built purely on Java. The reason Java was chosen as it provides some strong Object Oriented data structures, and since Java is very common, this can be extended in future by other developers. Also, there is a powerful open source library, Jena

which is available for Java.

- **Jena**<sup>4</sup> - This helps you in executing SPARQL queries from a Java software and also has some powerful in-built methods for exploring the SPARQL Result Set.
- **Jersey** - This is a Glassfish<sup>5</sup> based lightweight server. It provides the main REST API structure. The software has been developed in a very modular manner so that this can be replaced later with a more powerful server like the apache tomcat<sup>6</sup>.
- **Jackson**<sup>7</sup> - This is a high performance Java based JSON processor which provides a powerful JSON output of Java objects.
- **Lucene**<sup>8</sup> - This provides an in-house indexing solution which is used to retrieve properties and its statistics of classes very fast.

#### 4.2.1 RDF Exploration helper

In this section we will explain various actions we have developed which essentially helps in exploration of RDF data, its structure etc. This is useful and can be consumed by a GUI tool (We have developed a prototype of such a GUI tool which we will discuss more in section 4.3), to provide proper visual information to the user.

##### 4.2.1.1 Classes search

This action returns a list of classes which match a particular search string. The base action looks like

```
1 GET /v1.0/builder/classes
```

This action requires the following input parameters.

- *dataset* - SPARQL endpoint of the dataset
- *search* - search string which will match the class
- *force\_uri\_search* - This is a boolean value and is optional. If set true, then it forces to search for string in the uri patterns of classes which do not have any labels. Set this to true only if you want to search for classes which do not probably have any labels. By default it is false since REGEX searches are pretty expensive in SPARQL especially if we have to search in the middle of a token. In such cases it opts for brute force techniques which is not very optimal.

This action like any other standard REST API call, will return a JSON output.

---

<sup>4</sup><https://jena.apache.org/>

<sup>5</sup><https://glassfish.java.net/>

<sup>6</sup><http://tomcat.apache.org/>

<sup>7</sup><http://jackson.codehaus.org/>

<sup>8</sup><http://lucene.apache.org/>

We had to achieve an important search challenge with this action, i.e., to make sure we return our search result matching in various languages. We should somehow be able to return results in such a way that the same class's multiple language labels are returned (which have matched). The following shows a sample JSON response for a search of string "anim" in DBPedia<sup>9</sup>

```

1  {
2    "dataset": "http://dbpedia.org/sparql",
3    "search_string": "anim",
4    "searched_items": [
5      {
6        "labels": {
7          "it": "animale",
8          "pt": "animal",
9          "fr": "animal",
10         "en": "animal"
11       },
12       "sequence": 1,
13       "uri": "http://dbpedia.org/ontology/Animal"
14     },
15     {
16       "labels": {
17         "it": "personaggio animanga",
18         "en": "animanga character"
19       },
20       "sequence": 2,
21       "uri": "http://dbpedia.org/ontology/AnimangaCharacter"
22     },
23     ....
24   ]
25 }

```

Notice line 6. Here we have provided multiple labels as a hashmap. Hence, the user can only get results of the languages he wants. This gives a lot of freedom to the developer of the visual tool. Notice line 12. A sequence is provided so that the search results can be ordered (1 having the highest priority). This has been included as many times the order of the returned objects in JSON varies from library to library. Adding this makes our software more scalable. In our current version, it always returns in correct order.

We are doing a SPARQL search query which will return us a result of searched classes in the dataset. The following query is being used

```

1  SELECT distinct ?class ?label
2  WHERE {
3    {?class rdf:type owl:Class} UNION {?class rdf:type rdfs:Class}.
4    ?class rdfs:label ?label.
5    FILTER(bound(?label)
6           && REGEX(?label, "\\b<search_string>", "i"))
7  }
8  ORDER BY ?class

```

---

<sup>9</sup><http://dbpedia.org/>



In line 3 above, we match objects of type `owl:Class` or `rdfs:Class`, the two available standard ontology for classes. Now a disadvantage is that this won't return those classes which do not have these two types defined. A way to solve this is to replace it with `"?object rdf:type ?class"`, but we had to abandon that approach as the query was too expensive and always returned `Http Exception 500`<sup>10</sup>. In line 6, we pass a regex like that because we want to search only the beginning of tokens of a label. So if we search `"ani"` it will match classes *Animal*, *Farm animal* etc. but it won't match the class *Banished princes*.

We notice that this will search through only labels of classes. Now if a class does not have a label then it becomes a problem. For such scenarios, we have provided another input parameter *force\_uri\_search* which if set true will result in searching through URI patterns of classes too. For such scenarios we will execute another SPARQL query apart from the previous to match URI patterns and the results of both the queries are appended.

```

1  SELECT distinct ?class ?label
2  WHERE {
3    {?class rdf:type owl:Class} UNION {?class rdf:type rdfs:Class}.
4    OPTIONAL {?class rdfs:label ?label}.
5    FILTER(!bound(?label))
6    && REGEX(?class, "<search_string>", "i")
7  }
8  ORDER BY ?class
```

Notice line 5. We are matching for URIs which do not have any labels. We are doing this because we want to reduce the number of URIs to be searched as a free search REGEX is an extremely expensive search and no indexing solutions are available which speeds up this search.

This search can be speeded up by providing an in-house index lookup which we will discuss more in Chapter 6 for Future work.

#### 4.2.1.2 Class properties

This action will provide the end application with information about the data structure of the class. It will return a list of properties. The base action looks like

```
1  GET /v1.0/builder/properties
```

This action requires the following input parameters.

- *dataset* - SPARQL endpoint of the dataset
- *class* - URI of the class for which the properties are to be retrieved.

This action like any other standard REST API call will return a JSON output.

<sup>10</sup>This means an internal server error has happened. In our case it is because the RDF resultset the SPARQL query is dealing with is huge and it cannot handle it

It is always important that the user knows what is the data structure of a class. In any graph database this can be tricky as the structure is always dynamic and not as rigid as RDBMS systems. Hence, we decided to build this action which will provide with the properties of the class.

There are two kinds of properties. One is an *Object type* property and the other, *Data type* property.

**Definition 4.2.1.** *Object type property* - In a ?subject ?predicate ?object triple, the ?predicate is an *Object type property* if its range is a URI Object i.e., ?object is a URI.

**Definition 4.2.2.** *Data type property* - In a ?subject ?predicate ?object triple, the ?predicate is a *Data type property* if its range is a Literal Object i.e., ?object is a literal.

Apart from the properties it is essential that the user gets some information on how common is the property for that class. So we return a count of the range objects<sup>11</sup> of that property. We also return whether for some objects of the class have multiple range objects for that property. The following is the properties JSON output when the API is called for a class "Actor" with URI "http://dbpedia.org/ontology/Actor" from DBPedia.

```

1 {
2   "rdfClass": {
3     "dataset": "http://dbpedia.org/sparql",
4     "indexCreated": true,
5     "label": "actor",
6     "properties": [
7       {
8         "count": 18249,
9         "label": "has abstract",
10        "multiplePropertiesForSameNode": true,
11        "range": {
12          "label": "langString",
13          "uri":
14            ↪ "http://www.w3.org/1999/02/22-rdf-syntax-ns#langString"
15        },
16        "type": "data",
17        "uri": "http://dbpedia.org/ontology/abstract"
18      },
19      {
20        "count": 907,
21        "label": "birth name",
22        "multiplePropertiesForSameNode": true,
23        "range": {
24          "label": "langString",
25          "uri":
26            ↪ "http://www.w3.org/1999/02/22-rdf-syntax-ns#langString"
27        }
28      }
29    ]
30  }
31 }

```

<sup>11</sup>In ?s ?p ?o, if ?s is an object of the searched class and ?p is the property then the range object is ?o.

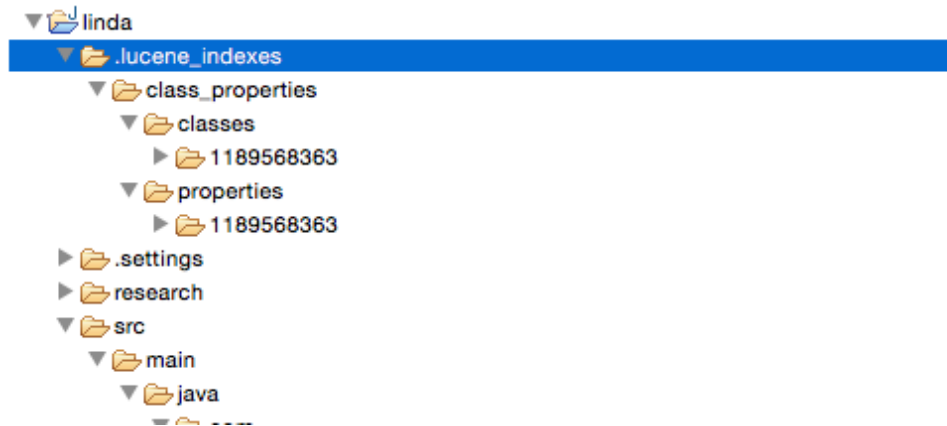
```

25         },
26         "type": "data",
27         "uri": "http://dbpedia.org/ontology/birthName"
28     },
29     .....
30     {
31         "count": 1807,
32         "label": "birth place",
33         "multiplePropertiesForSameNode": true,
34         "range": {
35             "label": "place",
36             "uri": "http://dbpedia.org/ontology/Place"
37         },
38         "type": "object",
39         "uri": "http://dbpedia.org/ontology/birthPlace"
40     },
41     .....
42 ],
43 "uri": "http://dbpedia.org/ontology/Actor"
44 }
45 }

```

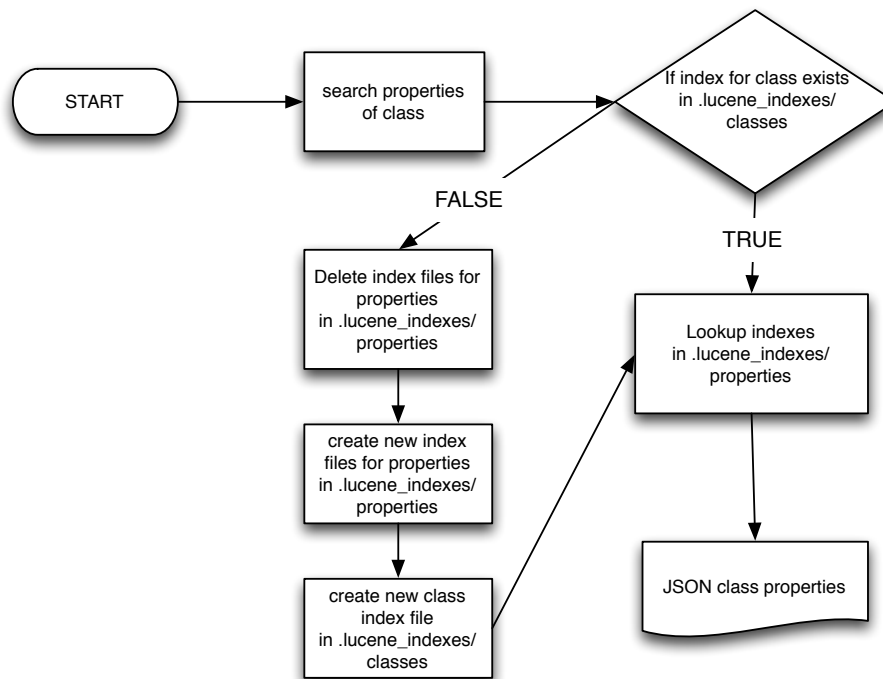
We can see from the output above that we return a list of properties. Each property has a key "type", whose value can be either *object* or *data* which specifies if it is an *object type* property or a *data type* property. In line 10 we notice a key "multiplePropertiesForSameNode" which specifies if some objects of the class have multiple range objects for that property. Its a boolean value. In Line 11, we notice that there is a key "range" which specifies the range of the property.

By taking a look at the output above, it seems obvious that to have a fast response we need to crunch the statistics before hand and store it somewhere. For this purpose we use a lucene indexing solution. We opted for lucene because we can include it in our Java project pretty easily as it also runs on Java. We are storing the indexes in the in the folder `.lucene_indexes` in our Java project (fig. 4.1). In the fig. 4.1, we can see there is a folder "class\_properties". This folder holds the indexes for the properties of classes. Now inside this folder, there are two other folders, viz., "classes" and "properties". In both of these folders we can see there is a folder "1189568363". This is nothing but a hashed value of the sparql string endpoint of the dataset "Dbpedia". These means, in that folder it will have index files for DBPedia. The main indexes are stored in the "properties" folder. Indexes are stored as standard documents format of lucene. The lookup is through a direct key search for the property's URI. The "classes" folder is used just as an internal check. In that an entry for a class is created if all the indexes for its properties are created in the properties folder. Using this technique we solve two things. *Firstly*, in case there is an error in creating indexes then there is no entry in the "classes" folder. Which means the indexes will be created again from scratch. *Secondly*, this helps in a fast lookup for whether



**Figure 4.1:** Lucene indexes stored in the project

the properties indexes have been created for the class in our index creation algorithm which we will discuss below.



**Figure 4.2:** A flowchart describing the properties indexing logic in Lucene

In fig. 4.2 we have described the flowchart which explains the logic for creating indexes. The first step is, to lookup in the `.lucene_indexes/classes`

folder for index entry of that particular class. If it exists we just lookup for the properties in the `.lucene_indexes/properties` folder. If it does not, we create index files and repeat the above process. Usually, this process takes from a few seconds to a few minutes, depending upon the number of objects of the class. It is recommended that the indexes should be created before hand so that there is no waiting time for the response in the first time hit of the class. For that we have created an administrative action, which will create indexes for all the classes in the dataset. This will be discussed more in section 4.2.1.6.

Now we will move on to index creation. There are three queries which we execute. First we find out the common *Object type* properties of the class. This is done by the query below.

```

1      SELECT DISTINCT ?property ?label
2      WHERE
3      {
4          ?concept rdf:type <class_uri>.
5          ?concept ?property ?o.
6          ?property rdfs:label ?label.
7          ?property rdf:type owl:ObjectProperty.
8          ?property rdfs:range ?range.
9          FILTER(langMatches(lang(?label), 'EN'))
10     } LIMIT 25

```

In line 6 above, we are specifying to search for properties which are of ontology `ObjectProperty`. In line 7 we are making sure the property has some range objects. We are not sorting this search result with an aggregate function of *count* so as to get the most common top 25 properties. That query we abandoned as it was very expensive and in most of the cases never finished executing. Therefore we opted for a simpler solution and we are letting SPARQL deal with it. Using the concept of randomized algorithms, this query will return in most of the cases, the most common properties and the probability of getting hits for those properties will be higher. Also, notice in line 8, that we are filtering the results for just english labels. This can be removed and multiple language labels can be dealt with as discussed in the previous sub-section.

Below is the query for getting *Data type* properties.

```

1      SELECT DISTINCT ?property ?label
2      WHERE
3      {
4          ?concept rdf:type <class_uri>.
5          ?concept ?property ?o.
6          ?property rdfs:label ?label.
7          ?property rdf:type owl:DatatypeProperty.
8          FILTER(langMatches(lang(?label), 'EN'))
9     } LIMIT 25

```

This also follows a similar concept like the *Object property* search. In line 6 above, we are specifying to search for properties which are of ontology `DataProperty`.

Now finally, we will search for properties which are defined in the rdf schema, i.e., the properties for which the searched class is the domain. We achieve that using the query below.

```

1  SELECT DISTINCT ?property ?label
2  WHERE
3    {   ?property rdfs:domain <class_uri>
4        ?property rdfs:range ?range.
5        ?property rdfs:label ?label.
6        FILTER(langMatches(lang(?label), 'EN'))
7    }
```

Once we have the list of properties from all three queries, we make a unique list of them, then find out the count of the objects of those properties. After gathering all these information we create index documents for these properties so that next time this is looked up really fast.

#### 4.2.1.3 Class Subclasses

This action will provide the end application with the subclasses of the inputted class. The base action looks like

```
1  GET /v1.0/builder/classes/subclasses
```

This action requires the following input parameters.

- *dataset* - SPARQL endpoint of the dataset
- *class* - URI of the class for which the subclasses are to be retrieved.

This action like any other standard REST API call will return a JSON output.

The development of this action is fairly straightforward. We use the RDF schema's ontology for subclasses to find the subclasses of the class inputted. This can be achieved using the following SPARQL query.

```

1  SELECT DISTINCT ?subclass_uri ?subclass_label
2  WHERE {
3    ?subclass_uri rdfs:subClassOf <class_uri>.
4    ?subclass_uri rdfs:label ?subclass_label.
5    FILTER(langMatches(lang(?subclass_label), 'EN'))
6  }
```

If you see line 3, we find classes which are subClassOf the searched class. The subclasses are defined using this standard RDFS ontology. In line 4 we assert for just English labels. This can be removed and other languages can also be handled easily.

Below, we can see a sample output of this action, when we look for subclasses of the class "Artist" in DBpedia.

```

1  {
2    "dataset": "http://dbpedia.org/sparql",
3    "label": "artist",
4    "subclasses": [
```

```

5      {
6          "label": "actor",
7          "uri": "http://dbpedia.org/ontology/Actor"
8      },
9      {
10         "label": "comedian",
11         "uri": "http://dbpedia.org/ontology/Comedian"
12     },
13     {
14         "label": "comics creator",
15         "uri": "http://dbpedia.org/ontology/ComicsCreator"
16     },
17     {
18         "label": "fashion designer",
19         "uri": "http://dbpedia.org/ontology/FashionDesigner"
20     },
21     .....
22 ],
23 "uri": "http://dbpedia.org/ontology/Artist"
24 }

```

#### 4.2.1.4 Class Examples

This action will provide the end application with some example objects of the inputted class. The base action looks like

```
1 GET /v1.0/builder/classes/examples
```

This action requires the following input parameters.

- *dataset* - SPARQL endpoint of the dataset
- *class* - URI of the class for which the example objects are to be retrieved.
- *limit* - This is the number of example objects which has to be retrieved. By default we have set this to 5.

This action like any other standard REST API call will return a JSON output.

The reason this action was developed was because we felt, it adds a little bit of usefulness to the user as he can get a simple overview of what kind of objects are there in the class(es) they are exploring. Also, the user can know the kind of objects that are present in the subclasses of the class, as essentially, the subclass is also a class. This method also returns the total number of objects for the class.

The following is the output obtained when this API is called for the class "Artist" in DBPedia

```

1 {
2     "dataset": "http://dbpedia.org/sparql",
3     "label": "artist",
4     "sample_objects": [

```

```

5      {
6          "label": "Aaron Lines",
7          "uri": "http://dbpedia.org/resource/Aaron_Lines"
8      },
9      {
10         "label": "Alex Reid (actress)",
11         "uri": "http://dbpedia.org/resource/Alex_Reid_(actress)"
12     },
13     {
14         "label": "Alma Cogan",
15         "uri": "http://dbpedia.org/resource/Alma_Cogan"
16     },
17     {
18         "label": "Andrew Foley (writer)",
19         "uri": "http://dbpedia.org/resource/Andrew_Foley_(writer)"
20     },
21     {
22         "label": "Andy Park (comics)",
23         "uri": "http://dbpedia.org/resource/Andy_Park_(comics)"
24     }
25 ],
26 "total_objects": 96300,
27 "uri": "http://dbpedia.org/ontology/Artist"
28 }

```

We can see in the line 4, that "sample\_objects" contains an array of sample objects for the class. The count of the objects is returned by "total\_objects" as can be seen in the line 26.

The logic for retrieving the objects is pretty straightforward. The following SPARQL query is used to retrieve the sample objects.

```

1  SELECT distinct ?object ?label
2  WHERE {
3      ?object rdf:type <class_uri>.
4      ?object rdfs:label ?label.
5      FILTER(bound(?label) && langMatches(lang(?label), "EN"))
6  }
7  LIMIT <limit>

```

Similarly, the "total\_objects" is generated by executing a SPARQL query with the COUNT aggregate function.

#### 4.2.1.5 Objects Search

This action will return a list of objects matching the search string. The base action looks like

```
1  GET /v1.0/builder/objects/
```

This action requires the following input parameters.

- *dataset* - SPARQL endpoint of the dataset
- *search* - Search string which will match the object



- *classes* - This will contain comma separated classes. Basically, the objects can be an object of any of these classes
- *for\_class* - This is the URI of the class for which the the searched object is an object in the RDF triple. In the triple ?s ?p ?o, ?s is the class here and ?o the returned object.
- *for\_property* - This is the URI of the property. In a triple ?s ?p ?o, ?o is the object which will be returned and ?p the property's URI.

This action like any other standard REST API call will return a JSON output.

This action is very similar to the Classes search in section 4.2.1.1. The output returned is similar to that of classes search. This action is useful for adding filters in properties from the Front end application, as the user can do free text search of objects. Using this action you can do a search like

```
1  ?class_object rdf:type ?for_class.
2  ?class_object ?for_property ?object.
```

In this you will search for ?object such that they are the objects of predicate *for\_property* and subject a class object of *for\_class*.

#### 4.2.1.6 Properties Index creation

This action will initiate creation of indexes of properties of all classes of a dataset. Its base action is as follows

```
1  GET /v1.0/builder/properties/indexes/create
```

This action requires the following input parameters.

- *dataset* - SPARQL endpoint of the dataset

This is the same index creation process which was explained in detail in section 4.2.1.2. We have decided to have an API for this action, as then it requires pretty much no knowledge and software installation for the administrator to execute this functionality. It usually takes about a few hours to create indexes for all the classes of a dataset.

#### 4.2.2 RDF2Any Conversion

Our main motivation was always to provide an easier way to extract data from RDF datastores. Right now users do not have that much flexibility and in most of the cases, he has to download the whole dataset to the format he wants. What we wanted was an added flexibility, so that can user can download smaller sets of data based on what he wants to filter. We would want the user to abstract data in the format he wants. We would want a state of the art RDBMS conversion. We now will explain in detail our approach and implementation in the following subsections.

But before we go to that, we will explain a bit on our approach on how to get subsets of a dataset. The simplest way of course is through SPARQL.

There are standard SPARQL result set serializations recommendations by W3C. But these serialization techniques are very one dimensional, and they just represent the data as it is returned in the tabular format. We therefore figured out a simpler technique which does not require the user to convert the whole dataset. We will approach with conversion of objects of just one class for the time being. And to find the objects we will use a SPARQL query which will return those objects. So we need the following three parameters to achieve this

- *dataset* - SPARQL endpoint of the dataset
- *for\_class* - URI of the objects of the class which will be abstracted.
- *query* - This is a SPARQL query which returns the URIs of the object of the class which has to be abstracted. It is of the format

```
1   SELECT ?concept WHERE{
2       ....
3       ....
4   }
```

Here, the variable ?concept will contain the URIs of the objects of the class.

This kind of exploration is a prototype. There is a flaw in it. From the definitions of the parameters above we notice that we can explore just one class. This can be easily rectified by passing comma separated values of classes and an array of their respective queries. For lossless sending of these parameters, it is recommended to send them as Base64 encoded strings.

Another notable thing which we implemented for all the Conversion file downloads is we are not having a Java return type File. Instead, we are creating an Output buffer stream, which creates a file download and we write binary data on the file on the fly. This means, we do not need to store the whole file temporarily in the main memory of the server reducing the unnecessary slug size of the system. So our conversions are swift and extremely memory efficient.

#### 4.2.2.1 CSV Conversion

CSV as we know stands for Comma separated values. CSV files are very useful in uploading data to other formats like excel sheets, databases, etc. In this we built an API action using which the user can download The RDF data in CSV. The base API looks like

```
1   GET /v1.0/convert/csv-converter.csv
```

This API action requires the following parameters

- *dataset* - SPARQL endpoint of the dataset
- *for\_class* - URI of the objects of the class which will be abstracted.

- *query* - This is a SPARQL query which returns the URIs of the object of the class which has to be abstracted.
- *properties* - Comma separated URIs of properties which need to be extracted. If all the properties are required then simply "all" can be passed.

The API returns a **.csv** file as an output. There are two kinds of outputs available from this API action. One is the traditional one for SPARQL query Result Set. It returns the data in a simple tabular format in which the headers are the variable names. For this action it is not required to pass the variables "for\_class" and "properties".

e.g., Let us take a simple a simple SPARQL query

```
1 SELECT DISTINCT ?s WHERE {
2     ?s ?p ?o.
3 } LIMIT 100
```

This will return a CSV output

```
1 rowID,s
2 1,http://www.w3.org/1999/02/22-rdf-syntax-ns#Property
3 2,http://www.openlinksw.com/schemas/virttrdf#QuadMapFormat
4 3,http://www.openlinksw.com/schemas/virttrdf#QuadStorage
5 ....
```

We notice in line 1 above that there are two headers. First is *rowID* which is nothing but a unique integer which is the row number. The other ones will be the variable names of the SPARQL query. In this case it is *s*.

Now let us move on to the exploration of Class objects. For such cases, the header items become various parameters, like object URI, object name, then individual headers for object properties which need to be downloaded. Check out the output below which is a CSV convert of the objects of Actors from DBpedia. In this case we are just getting a few properties, viz., *birth\_date*, *birth\_place*

```
1 rowID,rdfsMember,uri,name,birth_date,birth_place
2 1,http://dbpedia.org/resource,
   ↪ http://dbpedia.org/resource/Alex_Reid_(actress), Alex Reid
   ↪ (actress), 1980-12-23+02:00,
   ↪ http://dbpedia.org/resource/United_Kingdom;
   ↪ http://dbpedia.org/resource/Cornwall;
   ↪ http://dbpedia.org/resource/Penzance
3 2,http://dbpedia.org/resource,
   ↪ http://dbpedia.org/resource/Henri_Cogan,Henri Cogan,
   ↪ 1924-09-13+02:00, http://dbpedia.org/resource/Paris
4 ...
```

We notice that there are 4 headers apart from the properties *birth\_date* and *birth\_place*. *rowID* is similar to before. *rdfsMember* returns the base URI of the returned object's URI. *uri* returns the URI of the object. *name* returns the name of the object. This for now is in English. This can be configured later to return in the language the user wants. The rest are simply properties

which return the property of that object. If for some case (like in line 1 *birth\_place*) there are multiple properties for that property of that object, then those properties are separated by a special character ‘;’.

Implementing multiple classes in this case is tricky as CSV is a rigid tabular format and it does not give the flexibility of having different data-structures for different rows. The most optimal solution will be a download of a .zip file which contains multiple csv files, each for the objects of different class.

#### 4.2.2.2 RDB Conversion

In RDB conversion, the resultset is converted into SQL scripts such that they can be uploaded to a RDBMS database. The Base action of this API looks like

```
1 GET /v1.0/convert/rdb-converter.sql
```

This API action requires the following parameters

- *dataset* - SPARQL endpoint of the dataset
- *for\_class* - URI of the objects of the class which will be abstracted.
- *query* - This is a SPARQL query which returns the URIs of the object of the class which has to be abstracted.
- *properties* - Comma separated URIs of properties which need to be extracted. If all the properties are required then simply "all" can be passed.

The API returns a .sql file as an output. In this output file, standard SQL upload scripts are written which has been tested out with PostgreSQL database. We opted for PostgreSQL because it is usually the number one choice for open source databases by Enterprise applications.

Like the CSV convert, this also deals with two kinds of converts. The first is a simple convert of the SPARQL query result set. The other is that of the class objects. Check the following query

```
1 SELECT DISTINCT ?subject WHERE {
2     ?subject ?predicate ?object.
3 } LIMIT 100
```

This gives the following output.

```
1 DROP TABLE IF EXISTS things;
2 CREATE TABLE things
3 (
4 ID int,
5 subject varchar(1000),
6 PRIMARY KEY ID
7 );
8 INSERT INTO things VALUES(1,
    ↪ 'http://www.openlinksw.com/formats#default-iid');
```

```

9 INSERT INTO things VALUES(2,
    ↪ 'http://www.openlinksw.com/formats#default-iid-nullable');
10 INSERT INTO things VALUES(3,
    ↪ 'http://www.openlinksw.com/formats#default-iid-nonblank');
11 ...

```

This output is pretty much tabular. A table *things* is created with columns for the variables in the SPARQL query. In this case, its *subject* which we can see is defined in line 5. All the rows are converted to a simple insert script to the table *things* which we can see is there in lines 8-10.

The class convert was challenging. We built a state of the art conversion to RDBMS systems which is powerful than the existing ones. It deals with normalizations very smartly. The conversion process starts with first evaluating the properties which have to be extracted for the class. These properties are passed through the API. If "all", is passed then the properties are retrieved from the *class properties* API from section 4.2.1.2. Once the properties are there, there are some useful information available with the properties which helps in normalization. The *class properties* API returns also whether the property has multiple objects for some node or not. That means if it has then, it is a one-to-many relationship in terms of RDBMS. Otherwise it is a one-to-one relationship. Based on this the table creation scripts are written. First, there will be a table for the class. All class tables will have columns id, uri, and name. The main class table will have other columns which is basically the one-to-one properties. If its a *Data type* property, then it will be a text/integer field based on the data type. If its an *Object type* property, then it will be an integer foreign key linking to another class table. So based on this various class tables which are going to be required in this context will be created. This will be clear with a small example. In the following script, we are retrieving the class objects of *Comedian* from DBPedia with only three properties viz., *alias*, *abstract* and *birthplace*.

```

1 -- START table creation scripts properties pointing to other classes
2
3 DROP TABLE IF EXISTS places CASCADE;
4 CREATE TABLE places
5 (
6 id int PRIMARY KEY,
7 uri varchar(300),
8 name text
9 );
10
11 -- END table creation scripts properties pointing to other classes
12
13 -- START Table creation section for main class table
14
15 DROP TABLE IF EXISTS comedians CASCADE;
16 CREATE TABLE comedians
17 (

```

```

18 id int PRIMARY KEY,
19 uri varchar(300),
20 name text,
21 alias text
22 );
23
24 -- END Table creation section for main class table
25
26 -- START table creation scripts for normalized property tables
27
28 DROP TABLE IF EXISTS comedianhasabstracts CASCADE;
29 CREATE TABLE comedianhasabstracts
30 (id int PRIMARY KEY,
31 comedian_id int,
32 hasabstract text,
33 hasabstractLang varchar(6)
34 );
35
36 DROP TABLE IF EXISTS comedianbirthplaces CASCADE;
37 CREATE TABLE comedianbirthplaces
38 (id int PRIMARY KEY,
39 comedian_id int,
40 birthplace_id int REFERENCES places(id)
41 );
42
43 -- END table creation scripts for normalized property tables

```

There are three sets of tables which will be created. First, it will be the related class tables (In this case *places* in line 4). Then the main class table (In this case *comedians* in line 16). After that the normalized class table (*comedianabstracts* and *comedianbirthplaces* in lines 29 and 37). The order of these table creation scripts are important, because the normalized class tables will reference (will have foreign keys referencing there) the main class tables and also some related class tables. The main class table might reference some related class tables similarly. So it is important that those tables exist before hand as the referencing is done through foreign key constraints and if the table does not exist it will throw a constraint error.

Now lets check the properties. Since there are no instances for which the class *Comedian* can have two *aliases*, it is a one-to-one relationship. Therefore it is defined in the table *comedians* in line 21. This definition is text because *alias* is defined as a *Data type* property. Now *abstract* is also defined as a *Data type* property but there are some instances of objects for which there are multiple *abstracts*. Therefore, to normalize this, another table *comedianabstracts* has been created which references the table *comedians* (Line 29). The property *abstract* has a range Language Literal. The RDBMS tools which we came across do not have specific methods to store language separately. We came up with a solution of simply having a small string column for language for such properties. Hence, you can see in line 33, along with the column *hasabstract* there is another column *hasabstractLang*

which will store the language as a string. Now coming to the final property, i.e., *birthplace*, which is an *Object type* property. This property has range *Place*, so a table *places* has been defined (Line 4). Since its an *Object type* property, it will have a foreign key referencing the table *places*. Once the tables has been created the actual data insertion scripts are written.

```

1 --1. #####
2 INSERT INTO comedians (ID, uri, name) VALUES (1,
    ↪ 'http://dbpedia.org/resource/Tom_Wrigglesworth','Tom
    ↪ Wrigglesworth');
3 INSERT INTO comedianhasabstracts(id,comedian_id,hasabstract,
    ↪ hasabstractLang) VALUES(1, 1, 'Tom Wrigglesworth is an English
    ↪ stand-up comedian He was born and raised in Sheffield, South
    ↪ Yorkshire. In 2009 he was nominated for the main Edinburgh
    ↪ Comedy Award (formerly the Perrier awards) at the Edinburgh
    ↪ festival.', 'EN');
4 UPDATE comedians SET alias = 'Thomas' WHERE id=1;
5 INSERT INTO places(id, uri, name) VALUES (1,
    ↪ 'http://dbpedia.org/resource/Sheffield', 'Sheffield');
6 INSERT INTO comedianbirthplaces(id,comedian_id,birthplace_id) VALUES(1,
    ↪ 1, 1);
7 INSERT INTO places(id, uri, name) VALUES (2,
    ↪ 'http://dbpedia.org/resource/South_Yorkshire', 'South
    ↪ Yorkshire');
8 INSERT INTO comedianbirthplaces(id,comedian_id,birthplace_id) VALUES(2,
    ↪ 1, 2);
9
10 --2. #####
11 INSERT INTO comedians (ID, uri, name) VALUES (2,
    ↪ 'http://dbpedia.org/resource/Charles_Firth_(comedian)','Charles
    ↪ Firth (comedian)');
12 INSERT INTO comedianhasabstracts(id,comedian_id,hasabstract,
    ↪ hasabstractLang) VALUES(2, 2, 'Charles Henry Burgmann Firth is
    ↪ an Australian comedian, best known as a member of The Chaser
    ↪ productions CNNNN and The Chaser''s War on Everything. He is the
    ↪ brother of Verity Firth who was a Minister for the Labor
    ↪ Government of New South Wales.', 'EN');
13
14 --3. #####
15 INSERT INTO comedians (ID, uri, name) VALUES (3,
    ↪ 'http://dbpedia.org/resource/Charlie_Callas','Charlie Callas');
16 INSERT INTO comedianhasabstracts(id,comedian_id,hasabstract,
    ↪ hasabstractLang) VALUES(3, 3, 'Charlie Callas ..... ', 'EN');
17 INSERT INTO comedianhasabstracts(id,comedian_id,hasabstract,
    ↪ hasabstractLang) VALUES(4, 3, 'Charles ..... ', 'DE');
18 INSERT INTO comedianhasabstracts(id,comedian_id,hasabstract,
    ↪ hasabstractLang) VALUES(5, 3, 'Charlie Callas (20 de diciembre
    ↪ de 1927 - 27 de enero de 2011) .... ', 'ES');
19 INSERT INTO comedianhasabstracts(id,comedian_id,hasabstract,
    ↪ hasabstractLang) VALUES(6, 3, 'Charlie Callas ..... ', 'FR');
20 INSERT INTO comedianhasabstracts(id,comedian_id,hasabstract,
    ↪ hasabstractLang) VALUES(7, 3, 'Charlie Callas (Brooklyn, 20 de
    ↪ dezembro de 1924 - 27 de janeiro de 2011) foi um ..... ', 'PT');

```

```

21 INSERT INTO comedianhasabstracts(id,comedian_id,hasabstract,
    ↪ hasabstractLang) VALUES(8, 3, 'Charlie Callas .....', 'PL');
22 UPDATE comedians SET alias = 'Callias, Charles' WHERE id=3;
23 INSERT INTO places(id, uri, name) VALUES (3,
    ↪ 'http://dbpedia.org/resource/Brooklyn', 'Brooklyn');
24 INSERT INTO comedianbirthplaces(id,comedian_id,birthplace_id) VALUES(3,
    ↪ 3, 3);
25 .....

```

In the above example, we are continuing with the *Comedian* case and have shown 3 sample objects. First it always creates an INSERT script to insert data into the main class table. In this case its *comedians* (Line 2). An important aspect in these scripts, are the retrieving of correct foreign keys. This is done by a complex HashMap which stores the primary keys for objects of each table. First it looks up whether the object uri exists in the hashmap. If it exists then the primary key is retrieved otherwise a new one is created. Now coming back to the conversion, the algorithm for that is, it goes to each property to be dealt and just writes scripts for it. If it is for a new row entry then an INSERT script is created (Line 3). Now if it comes to a property which is a column of the main table, then an UPDATE script is written to modify the entry of the table (Line 4). This approach of writing UPDATE scripts is actually inefficient and increases the number of lines of scripts for the downloaded file. But it reduces the space complexity of the algorithm as no lookup is required to find out which are the properties for that class. It is just dealt as it comes. Also, it is notable how different languages are dealt with for Language literals (Lines 16 and 17). This dealing of languages are not present in the existing RDBMS conversions.

In this section we saw, how we achieved a powerful state-of-the-art RDF to RDB conversion in which its not required to input the whole dataset. One minor flaw we can notice is that we are dealing with just one main class (In the above example it was the class *Comedian*). Now this can be easily improved by extending the same principle for multiple classes. There will be no changes required in the existing algorithm to achieve that.

#### 4.2.2.3 JSON Conversion

In JSON conversion the result set is converted into JSON output. The Base action of this API looks like

```
1 GET /v1.0/convert/json/
```

This API action requires the following parameters

- *dataset* - SPARQL endpoint of the dataset
- *for\_class* - URI of the objects of the class which will be abstracted.
- *query* - This is a SPARQL query which returns the URIs of the object of the class which has to be abstracted.



- *properties* - Comma separated URIs of properties which need to be extracted. If all the properties are required then simply "all" can be passed.
- *json\_output\_format* - can have values {"virtuoso", "sesame"}. This determines the JSON serialization structure based on this value. By default its "virtuoso" which is the W3C recommended one. When doing a convert for class objects, it does not matter which value is passed in this parameter.

Like all standard REST API, this will return a JSON output.

JSON conversion becomes very important in this age of cloud computing. JSON is the de facto output format for REST API calls. Moreover it is directly converted to a Hash Map in any language it is imported to, removing the complicated programming which the programmers need to do for parsing. They can just do *key* lookups instead of parsing to get values in JSON. Because of such reasons we focused on providing many variants of JSON convert so that the user is comfortable in accessing the data. They are the official W3C format for SPARQL query result set, Sesame<sup>12</sup> format for SPARQL query result sets and an improved conversion to JSON format for class convert which is an improvement to the JSON-LD recommendation. One thing which has to be noted for JSON output is that, it is not practically recommended to have larger JSON files. The reason behind that is, JSON files are loaded completely into the main memory and if its large then the software can run out of memory and becomes sluggish. For larger outputs, it is recommended to have XML output so that one block at a time can be read in that.

The W3C recommended conversion is a lightweight and neat format for getting SPARQL query result sets. Let us take the following query

```
1  SELECT DISTINCT ?concept ?label WHERE {
2      ?concept rdf:type dbpedia-owl:Comedian.
3      ?concept rdfs:label ?label.
4      FILTER(langMatches(lang(?label), "EN"))
5  } LIMIT 20
```

This will give us the following output format.

```
1  {
2      "head": {
3          "vars": [
4              "concept",
5              "label"
6          ],
7          "link": []
8      },
9      "results": {
10         "bindings": [
11             {
```

---

<sup>12</sup><http://rdf4j.org/>

```

12         "concept": {
13             "value":
14             ↪ "http://dbpedia.org/resource/Tom_Wrigglesworth",
15             "type": "uri"
16         },
17         "label": {
18             "value": "Tom Wrigglesworth",
19             "xml:lang": "en",
20             "type": "literal"
21         }
22     },
23     ....
24 ]
25 }
26 "distinct": true,
27 "ordered": true,
28 "time_taken": 0.088
29 }

```

We notice in line 2 that there is a key *vars* which contains an array of the SPARQL variables. The results are inside the key *bindings* (line 10) as an array of hashmaps. The result rows are a hashmap with keys for variable names (lines 12 and 16). This means the lookup for these values is very easy. Each result binding value will have a key *type* which can have values {"uri", "literal", "typed-literal"} (lines 14 and 19). The values are there in the key *value* (lines 13 and 17). If it is a language literal then a new key is added *xml:lang* (line 18) which contains the language.

Now the same query will give the following output if we want the one provided by Sesame servers.

```

1 {
2     "sparql": {
3         "head": {
4             "variable": [
5                 {
6                     "@name": "concept"
7                 },
8                 {
9                     "@name": "label"
10                }
11            ]
12        },
13        "@xmlns": "http://www.w3.org/2005/sparql-results#"
14    },
15    "results": {
16        "result": [
17            {
18                "binding": [
19                    {
20                        "@name": "concept",
21                        "uri":
22                        ↪ "http://dbpedia.org/resource/Tom_Wrigglesworth"
23                    },

```

```

23         {
24             "@name": "label",
25             "literal": {
26                 "#text": "Tom Wrigglesworth",
27                 "@xml:lang": "en"
28             }
29         }
30     ]
31 },
32 ....
33 ]
34 }
35 }

```

We notice that this version is not that efficient and seems to be a bit heavier than the official recommended version. The variables are set as an array of hashmaps instead of a simple array of strings (line 4). Also in the results, it is an array of hashmap for each row instead of being a hashmap for each row (line 18). This means, if I want to access the value of a variable, say in this case "concept" in a row, I need to iterate through it and match whichever has the value of "@name" , "concept" then I will get the value (Line 20). But we won't have this situation with the official W3C version as in that each result row is a hashmap with the variable names as keys. All I need to do is lookup the key with "concept". Another issue with this format is the the nomenclature of keys. We can see a lot of keys starting with special characters, i.e., "@" and "#". The JSON format is made primarily for JavaScript language and in it we have symbols to access the keys. eg, if the key is "name" I can access it by row.name. But if the key is "@name" then I cannot access by row.@name because @name is not a valid symbol like name. In such case I need to define a string to access it, ie I need to do something like row["@name"]. Now the difference between a symbol and a string is that a symbol is immutable. This means if I create two symbols with the value name then both of them will have the same object ids and two objects won't be instantiated. Now this is not the case for strings. Now imagine the key "@name" being accessed like row["@name"] in a loop of 1000 records. 1000 String objects will be instantiated which is extremely inefficient.

Let us now check out the new suggested output format which we came up with for exploring object instances of classes. We will again go with the same *Comedian* class of DBPedia. We will just abstract the properties *alias*, *abstract* and *birthplace*. This will give the following output

```

1 {
2     "classes": [
3         {
4             "label": "comedian",
5             "properties": [
6                 "alias",
7                 "birth_place",
8                 "abstract"

```

```

9         ],
10        "uri": "http://dbpedia.org/ontology/Comedian"
11    }
12 ],
13 "properties": {
14     "abstract": {
15         "range": {
16             "label": "langString",
17             "uri":
18 ↪ "http://www.w3.org/1999/02/22-rdf-syntax-ns#langString"
19         },
20         "label": "has abstract",
21         "type": "data",
22         "uri": "http://dbpedia.org/ontology/abstract"
23     },
24     "alias": {
25         "range": {
26             "label": "string",
27             "uri": "http://www.w3.org/2001/XMLSchema#string"
28         },
29         "label": "alias",
30         "type": "data",
31         "uri": "http://dbpedia.org/ontology/alias"
32     },
33     "birth_place": {
34         "range": {
35             "label": "place",
36             "uri": "http://dbpedia.org/ontology/Place"
37         },
38         "label": "birth place",
39         "type": "object",
40         "uri": "http://dbpedia.org/ontology/birthPlace"
41     }
42 },
43 "dataset": "http://dbpedia.org/sparql",
44 "objects": [
45     {
46         "class": "http://dbpedia.org/ontology/Comedian",
47         "label": "Tom Wigglesworth",
48         "properties": {
49             "abstract": [
50                 {
51                     "xml_lang": "en",
52                     "value": "Tom Wigglesworth is an English
53 ↪ stand-up comedian He was born and raised in Sheffield, South
54 ↪ Yorkshire. In 2009 he was nominated for the main Edinburgh
55 ↪ Comedy Award (formerly the Perrier awards) at the Edinburgh
56 ↪ festival.",
57                     "type": "literal"
58                 }
59             ],
60             "alias": [
61                 {
62                     "value": "Thomas",

```

```

58         "type": "literal"
59     }
60 ],
61     "birth_place": [
62         {
63             "value":
64             ↪ "http://dbpedia.org/resource/Sheffield",
65             "type": "uri"
66         },
67         {
68             "value":
69             ↪ "http://dbpedia.org/resource/South_Yorkshire",
70             "type": "uri"
71         }
72     ],
73     "uri": "http://dbpedia.org/resource/Tom_Wrigglesworth"
74 },
75 ...
76 ]
77 }

```

For our format, we are going with the standard naming conventions for keys of JSON output. We are providing spider case<sup>13</sup> key values. First, we specify the classes with the key *classes* (line 2). This will contain an array of Hashmaps with more information on the classes which are retrieved. Right now only one has been implemented, and in future this can easily be extended to include multiple classes in future. Now coming back to the classes, here we can see some information of the *Comedian* class, like the *uri* (line 10), the name/label (line 4) and the *properties* which will be extracted (line 5). The *properties* are just an array of strings. Now there is a key *properties* which have more information of all the properties which are being extracted (Line 13). We notice there are information on each property's range, uri, etc. We have a universal array for properties, to avoid data redundancy. Multiple classes can have the same property so to avoid defining them more than once, we have a separate array for all the properties. The results are in the key *objects* as an array of objects (Line 43). Each object will have a key, *class* (Line 45) which will contain the URI of the class. It will also have keys for *label* (line 46) and *uri* (line 72). Each object will have a key *properties* which will contain the hashmap of the properties (Line 47) with the property identifier as the key for simpler lookup. Each property will have multiple values as the same property can have multiple values, so an array as value (line 49). Each value is similar to that of the W3C recommendation with a key *type*, *value*. The only modification added is the key *xml:lang* has been replaced to *xml\_lang* so that a symbol can be defined for that as *xml:lang* is not a valid symbol in most of the languages.

<sup>13</sup>spider case variables are those in which all the characters are lowercase and tokens are separated by underscores.

Because of the same reason we are not using the character '@' to initialize the variables (here the property names), as a symbol cannot be formed in most languages starting with '@'.

Thus, with this new format we were able to achieve the following

- Removed data redundancy by adding a unique array which contains all the properties.
- Added more information on the properties like its URI, range, etc.
- Made the keys standard for JSON so that symbols can be created rather than strings to access the corresponding values
- Objects have a dynamic structure which one can expect from the XML serialization.

#### 4.2.2.4 Generic Conversion

Now for the final Conversion API. This is called *Generic Conversion* because in this the user can specify with some parameters, the output format he wants. He can abstract the data to his own kind of XML, CSV, TSV and pretty much any kind of serialization. The Base action of this API looks like

```
1 GET /v1.0/convert/configured-convert
```

- *dataset* - SPARQL endpoint of the dataset
- *for\_class* - URI of the objects of the class which will be abstracted.
- *query* - This is a SPARQL query which returns the URIs of the object of the class which has to be extracted.
- *properties* - Comma separated URIs of properties which need to be abstracted. If all the properties are required then simply "all" can be passed.
- *variable\_dictionary* - this will define the variables for the property uris. The definitions are comma separated.
- *header* - This will constitute the head of the serialization output. It will be printed once and is static
- *body* - This will constitute the body of the serialization output. This part will be looped with every object of the resultset
- *footer* - This will constitute the footer of the serialization output. It will be printed once in the end and is static.

We got the main idea to develop an API like this from the language XSPARQL[AKKP+08]. But the major problem with XSPARQL is that the user still needs to know two languages, viz., SPARQL and XQuery. We figured, if just use a few programmable items like *for each loop*, *if condition* and *print* we can print pretty much all kinds of serialization.

So to describe your own generic output format you need to pass four parameters.

1. *variable\_dictionary* - this will define the variables for the property uris. The variable definitions are comma separated. The following is how the variable dictionary should be passed.

```
1 variable_name1::property_uri1,variable_name2::property_uri2
```

2. *header* - This will constitute the head of the serialization output. It will be printed once in the top and is static. This is a simple string. Right now we do not support any variables in the *header*.
3. *body* - This will constitute the body of the serialization output. This part will be looped with every object of the resultset. In the body we have our own scripting language with three programmable items. For any programmable part, it has to be escaped using

```
1 $[programmable_part_here]
```

- *print* - this is a simple print of a variable. variable names, *URI* and *NAME* are reserved for uri and name of the object. *OBJECT\_COUNTER* returns a unique integer counter for the object. The print syntax is as follows. It should begin with "=" followed by the variable name.

```
1 $[=variable_name]
```

For language literals the language and text can be separately printed. `textitvariable_name.text` gives the text and `variable_name.lang` returns the language.

- *if condition* - This is basically an assertion check. It checks whether the variable value exists or not. It is used to mainly check if the property exists.

```
1 $[if property_variable_name] some body here $[end]
```

- *for each loop* - This loops over the values of a particular property.

```
1 $[for property : property_variable_name]
2     some body here
3 $[end]
```

4. *footer* - This will constitute the footer of the serialization output. It will be printed once in the end and is static. Right now, no programmable items are supported in the footer.

So the main programmable elements will be in the body part of the input parameter. Even though we have just 3 programmable items, it is not that easy to implement. We went back to the basics of language parsers. We built our own proprietary language parser without using any libraries. The basic logic is that it parses the body text to an expression tree[C+86]. Now specific to our case, it parses to an expression tree of *Body chunks*. We have defined a *BodyChunk* class which looks like

```

1 Class BodyChunk{
2     String type;
3     String value;
4     String additionalValue;
5     List<BodyChunk> bodyChunks;
6 }

```

*type* has values "text", "condition", "variable", "loop". When the *type* is "text", then it means that it is a simple BodyChunk and the *value* will be printed. When it has *type* "condition", then it will first evaluate the condition which will be stored in *value* field. If condition satisfies, then the list of *bodyChunks* will be evaluated. If the *type* is "variable", then it is the print part. It will print the variable. And finally if the *type* is "loop", then it will loop through the *bodyChunks* inside it till the condition satisfies.

Let us now work with an example to display how a generic output can be abstracted. We will try to abstract some object instances of the Class *Comedian* of DBpedia. For this example We will consider just two properties, say *alias* and *abstract*. So we will pass the following *variable\_dictionary*

```

1 abstracts :: http://dbpedia.org/ontology/abstract, alias ::
  ↪ http://dbpedia.org/ontology/alias

```

We now pass the *header*

```

1 <?xml>
2 <comedians>

```

And the *body*

```

1 <comedian uri="$[=URI]" rowID="$[=OBJECT_COUNTER]">
2   <name>$[=NAME]</name>
3   $[if alias]<alias>$[=alias]</alias>${end}
4   $[for abstract:abstracts]
5   <description language="$[=abstract.lang]">
6     $[=abstract.text]
7   </description>
8   ${end}
9 </comedian>

```

And finally the footer

```

1 </comedians>

```

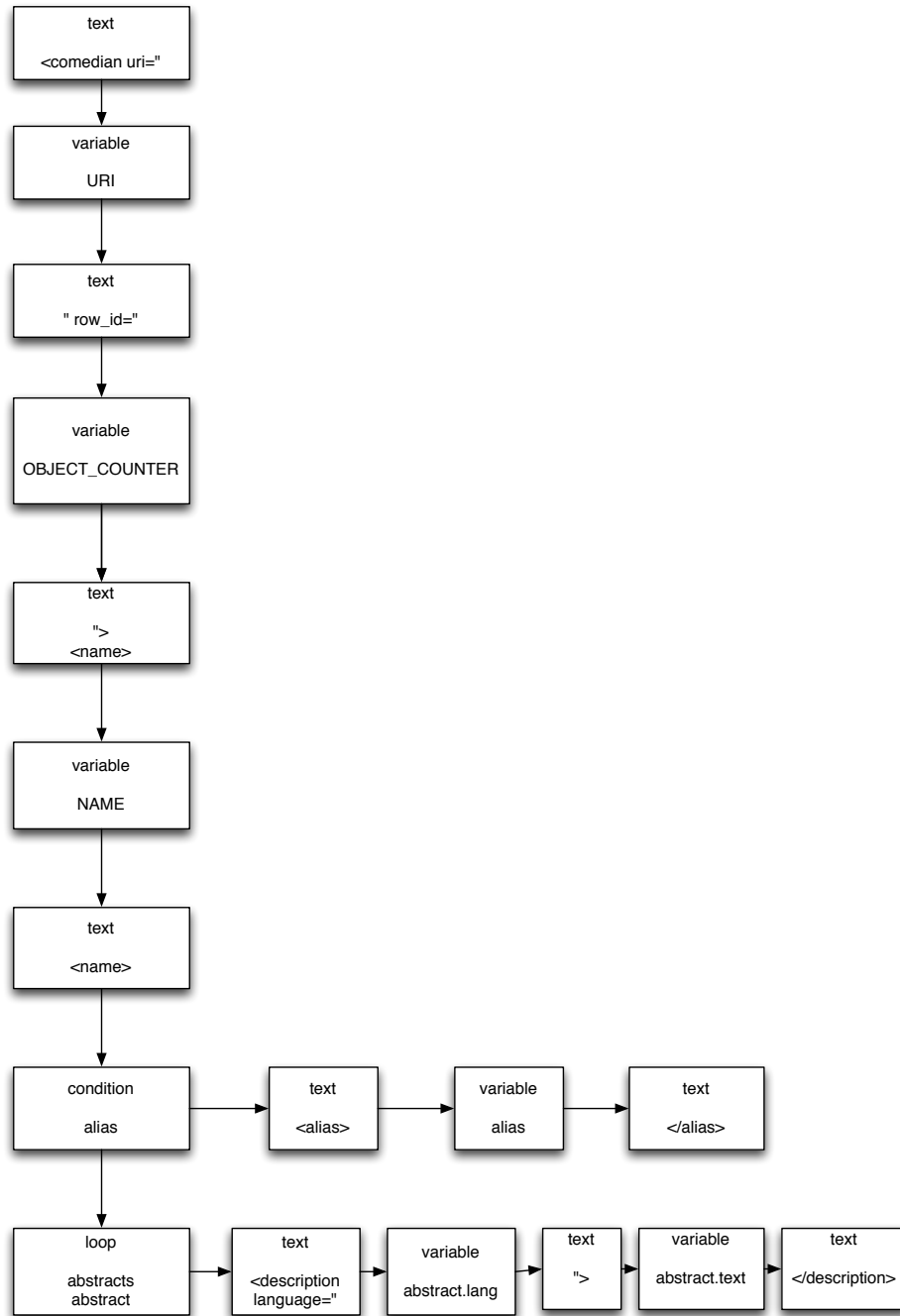
Now our proprietary language parser will parse the *body* text and will convert it into an expression tree with *body chunks* (fig. 4.3). After this, these body chunks will be recursively evaluated to give the following output.

```

1 <?xml>
2 <comedians>
3   <comedian
4     ↪ uri="http://dbpedia.org/resource/Charles_Firth_(comedian)"
5     ↪ row_id="1">
6     <name>Charles Firth (comedian)</name>
7     <description language="en">
8       .....

```





**Figure 4.3:** Example Body text of Generic convert parsed into Body Chunks expression tree

```

7      </description>
8    </comedian>
9    <comedian uri="http://dbpedia.org/resource/Charlie_Callas"
    ↪ row_id="2">
10      <name>Charlie Callas</name>
11      <description language="en">
12        ...
13      </description>
14      <description language="de">
15        ...
16      </description>
17      <description language="es">
18        ...
19      </description>
20      ...
21    </comedian>
22    ....
23 </comedians>

```

Notice for the first comedian (line 2), there is now *alias* printed. It is because we added the *if condition* in the body (line 3 of body). Also notice how the *for each loop* iterates through each abstract (lines 11-19).

In this Conversion we achieved

- Built a proprietary scripting language and parser which has a few programmable items.
- Any kind of serialization which does not require some memory storage (Like storing of foreign key values for RDBMS serialization), can be implemented using these features.

### 4.3 Query Builder GUI Tool

The main motivation for Building this GUI tool was to allow users to explore the RDF data. Using this, the user can abstract the data in the format he wants by using UI tools. All the information for this exploration is provided by the Exploration and Conversion API which we described in the previous section. **Technologies used** - the following technologies/libraries have been used to develop the GUI tool

- **Ruby on Rails** - The whole GUI tool was built as a web application using the framework Rails<sup>14</sup>. The server side scripting language used is Ruby. This was preferred as we wanted to show some interoperability as we are consuming actions from a Java server. Also, development of Ruby on Rails is very fast compared to other web frameworks.
- **Javascript** - Javascript is the standard client side scripting language for browsers.

---

<sup>14</sup><http://rubyonrails.org/>

- **Coffescript** - Coffescript<sup>15</sup> is a language in which your Ruby code is converted to Javascript from the server. This is useful as your code size becomes smaller and it becomes more modular.
- **JQuery**<sup>16</sup> - This is a javascript library which simplifies various functions which you would like to achieve using javascript.
- **Bootstrap**<sup>17</sup> - This is a CSS library published by Twitter. Using this you can easily build responsive and beautiful UI by already including the CSS items available.
- **Httparty**<sup>18</sup> - This is a gem available for Rails applications, using which you can call HTTP requests from within the application. We use this to call the APIs from our Exploration and Conversion API server.

In the following sub-sections we will describe the salient features of the Query Builder GUI tool and why we developed those features.

#### 4.3.1 Selecting datasets

As a user, the first step to explore data is to find a datastore. The datastores usually build a small GUI where a user can execute SPARQL queries. e.g., DBpedia has a SPARQL query editor at <http://dbpedia.org/sparql>. It is a simple UI where a user can just input the SPARQL query and the output format he requires (fig. 4.4).

We have given the user the option to select any Datastore which he wants to explore. We have provided with a drop down list from which the user can select the Datastore (fig. 4.5). Please note that right now it's showing only "DBpedia", but more datasets can be added here. There is no hardcoding done in any of our implementations. This addition is very simple. We have a method which returns pairs of *Display value of dataset*, and *SPARQL endpoint of the dataset*. Any dataset with a SPARQL endpoint can be added by the developer of the tool like this so that it is available to the user. Now we will proceed to the next step.

#### 4.3.2 Exploring classes

Finding out classes which someone wants to explore can become messy. In SPARQL a user has to know the exact URI to get more information of the class. We have implemented a simple searchbox where the user can search the Class which he wants to explore. Fig. 4.6 shows how the search results look like. It is a responsive design with auto-complete and the results get updated with the change of search value in the textbox. We use the *Classes*

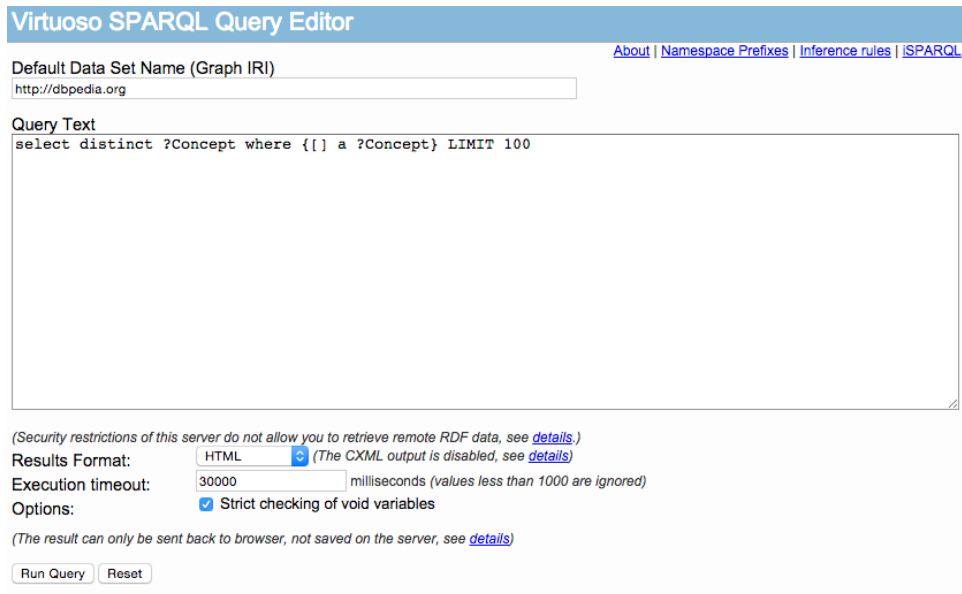
---

<sup>15</sup><http://coffeescript.org/>

<sup>16</sup><http://jquery.com/>

<sup>17</sup><http://getbootstrap.com/>

<sup>18</sup><https://github.com/jnunemaker/httparty>



**Virtuoso SPARQL Query Editor**

[About](#) | [Namespace Prefixes](#) | [Inference rules](#) | [SPARQL](#)

Default Data Set Name (Graph IRI)

Query Text  

```
select distinct ?Concept where {[ ] a ?Concept} LIMIT 100
```

(Security restrictions of this server do not allow you to retrieve remote RDF data, see [details](#).)

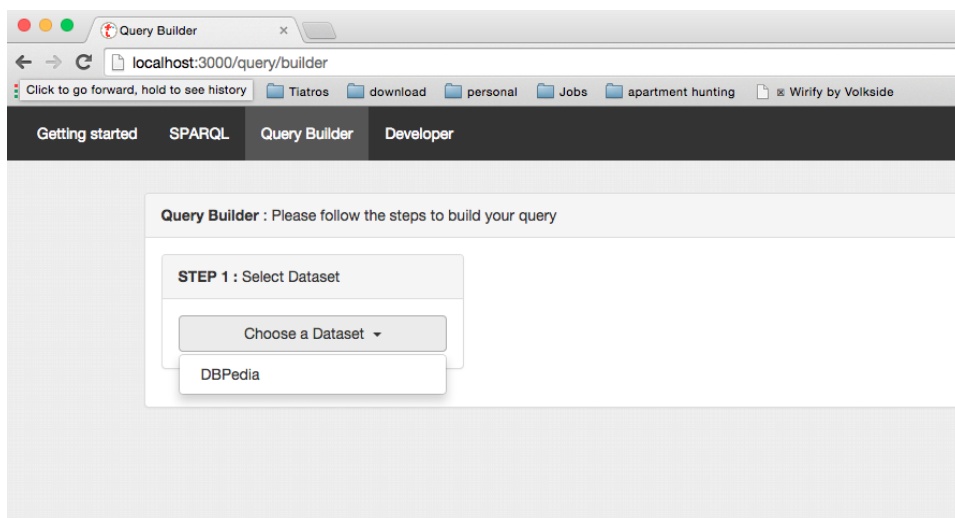
Results Format:  (The CXML output is disabled, see [details](#))

Execution timeout:  milliseconds (values less than 1000 are ignored)

Options: ☒ Strict checking of void variables

(The result can only be sent back to browser, not saved on the server, see [details](#))

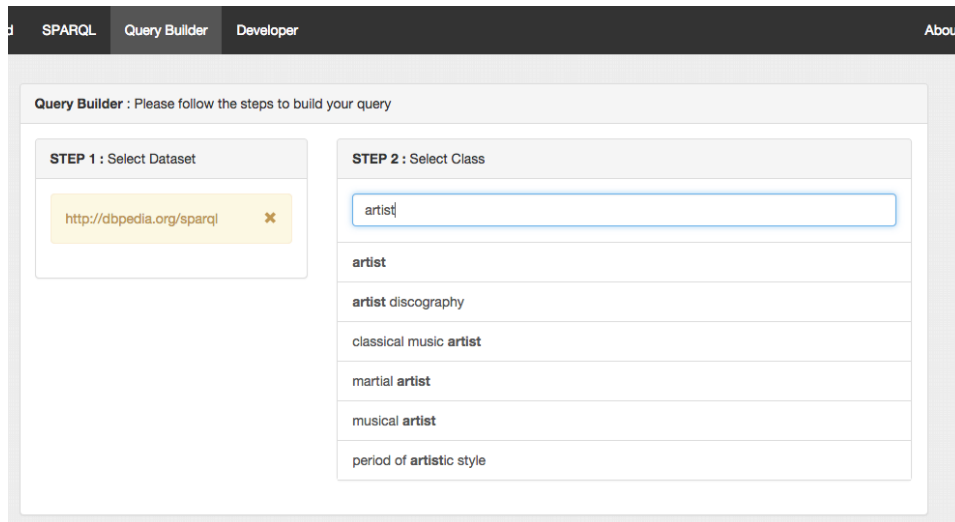
**Figure 4.4:** DBpedia's SPARQL Query Editor



**Figure 4.5:** Query Builder GUI : Selecting datasets

*search* API (section 4.2.1.1) to retrieve the classes matching the searched string.

After a class is clicked some information about the class is displayed (fig. 4.7). Five instances of the objects of the class are displayed. Also, the total number of objects of the class is displayed so that the user knows how many objects the class has. In this example it is around 96.3K. There is also a URI



**Figure 4.6:** Query Builder GUI : Searching classes

viewer which is opened whenever I click on any of the objects or I click the button for "More details on <class\_name>". The URI viewer opens the uri in the same application without redirecting to the uri so that the user does not lose focus (fig. 4.8).



**Figure 4.7:** Query Builder GUI : After selecting a class

Now a user can explore more on the data structure of the class by clicking on the '+' button next to the selected class in fig. 4.7. Clicking that will show up the classes histogram. Fig. 4.9 shows how the classes histogram looks like. It will have subclasses to the class. It follows a similar pattern of showing the number of objects and five object instances. This feature gives the user a fair amount of idea about the subclasses of the class. There is also a discover icon (represented by a globe icon) next to all the subclasses, clicking on which will select that class. We use the *Class subclasses* API action (section 4.2.1.3) to get the subclasses into the UI. To get the example instances and count of objects of the class we use the API *Class examples* (section 4.2.1.4).

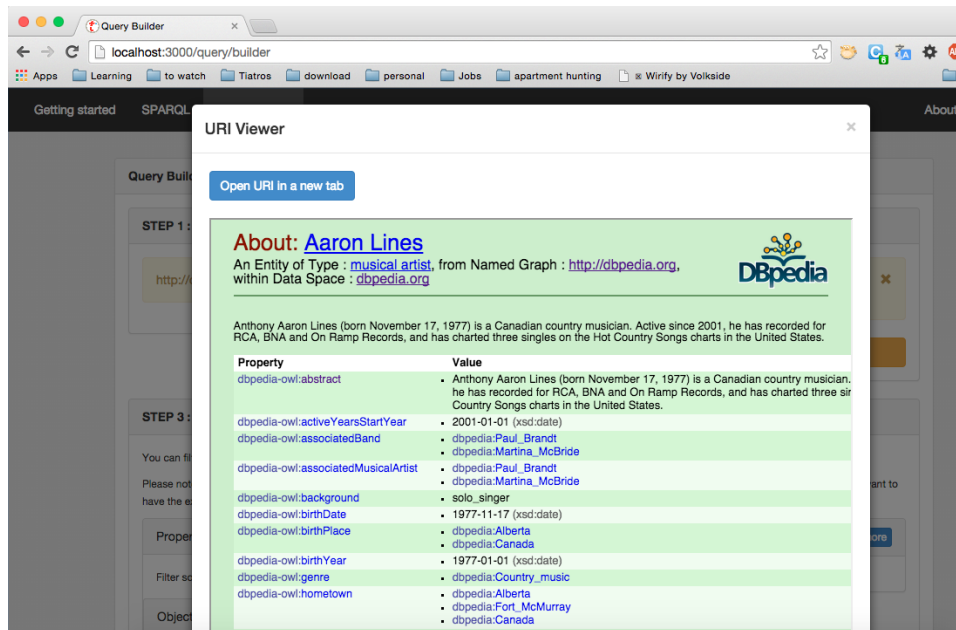


Figure 4.8: Query Builder GUI : URI viewer

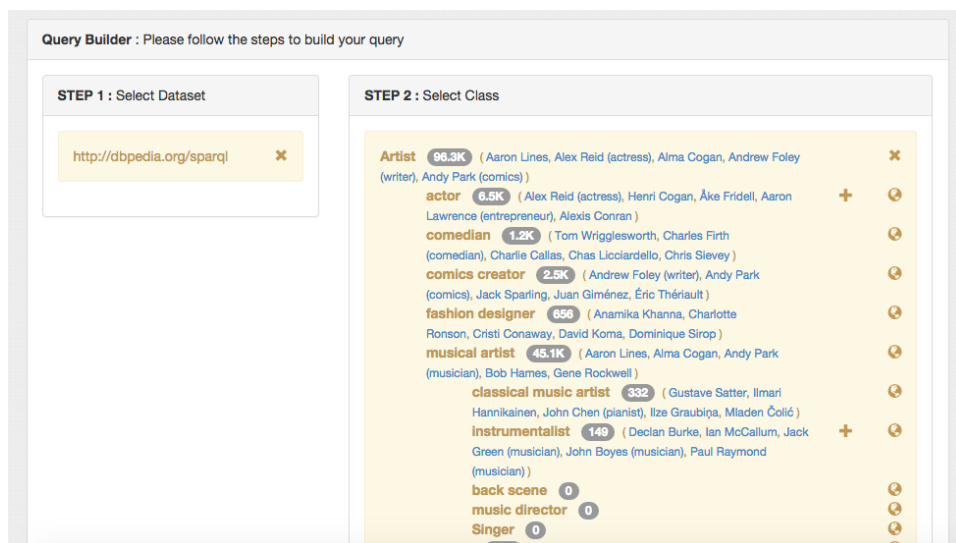
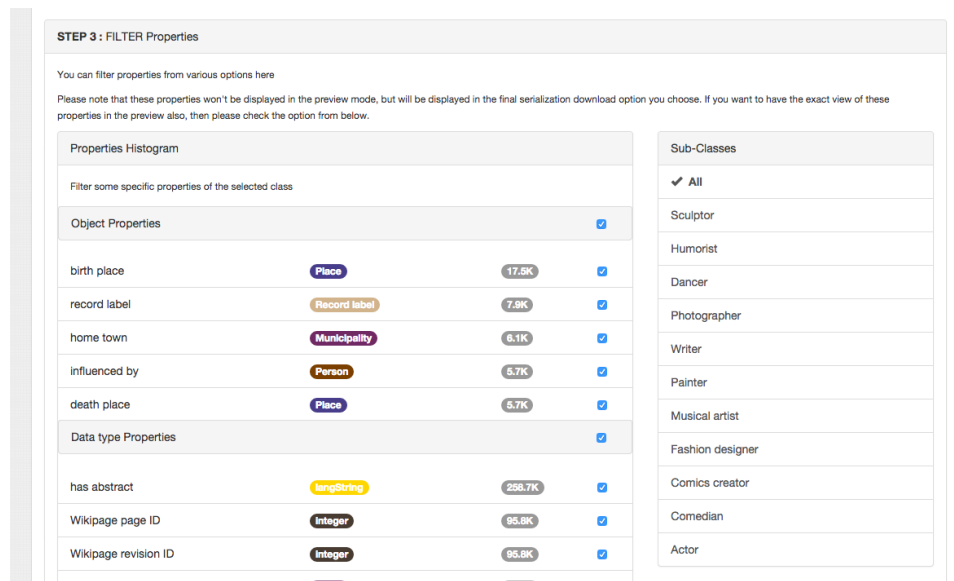


Figure 4.9: Query Builder GUI : Classes histogram

### 4.3.3 Properties Histogram

The next step for the user is to explore the properties of the selected class and if he wants then add some filters to those properties. We provide a properties histogram which separates out the *Object type* properties and

the *Data type* properties (fig. 4.10). These properties are retrieved from the *Class properties* API (section 4.2.1.2). Three things are displayed for each property. Apart from the name, the range class and also the count of the objects which have that property are also displayed. Notice that same range classes have the same color in the property histogram. This helps the user to quickly identify properties which have similar ranges. Subclasses are also displayed if there are any for the class. The user can click on those subclasses to filter the search results.



**Figure 4.10:** Query Builder GUI : Property histogram

The user can click on any of these properties to add a filter. Adding the filters is similar to the *class search* and objects can be searched using free text search (fig. 4.11). There is also an option to either match these objects or to NOT match them. This can be done by clicking on the buttons "equals to" or "not equals to". These objects are retrieved from the *Objects search* API (section 4.2.1.5). Similarly, *Data type* properties can also be filtered. We do not have a responsive filter for that, but a simple textbox where the user types the filter values. This improvement is part of our future work.

The next step is to select the properties which the user wants to abstract. This can be done by clicking on the checkboxes on the right end of each property (fig. 4.12). Notice in the same figure, the filter i.e., the actors who have their birthplace in *Berlin*, *Amsterdam* and *Athens*, which we had added in fig. 4.11. After this the user can check the equivalent SPARQL query.

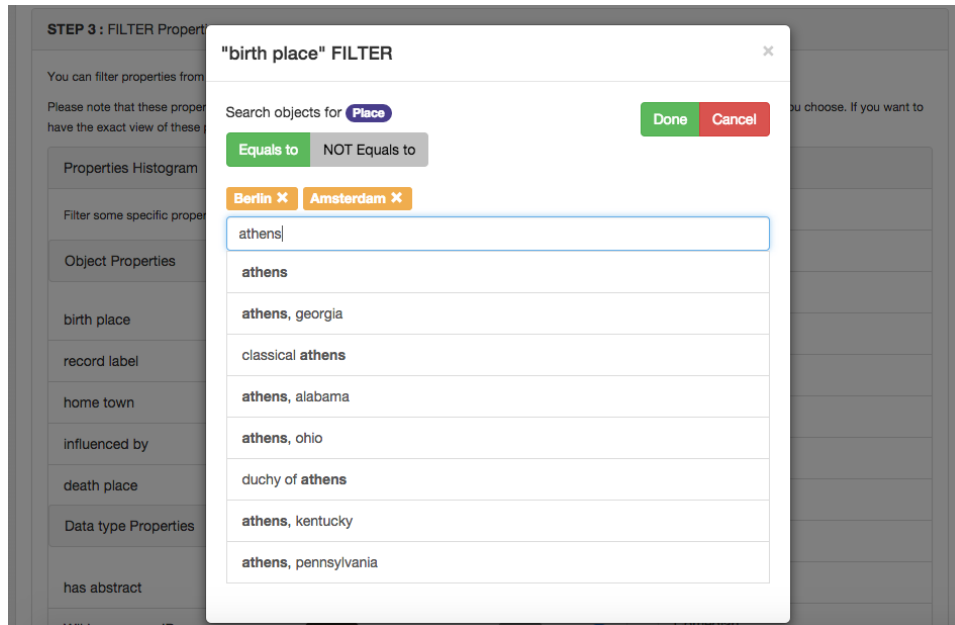


Figure 4.11: Query Builder GUI : Property filter

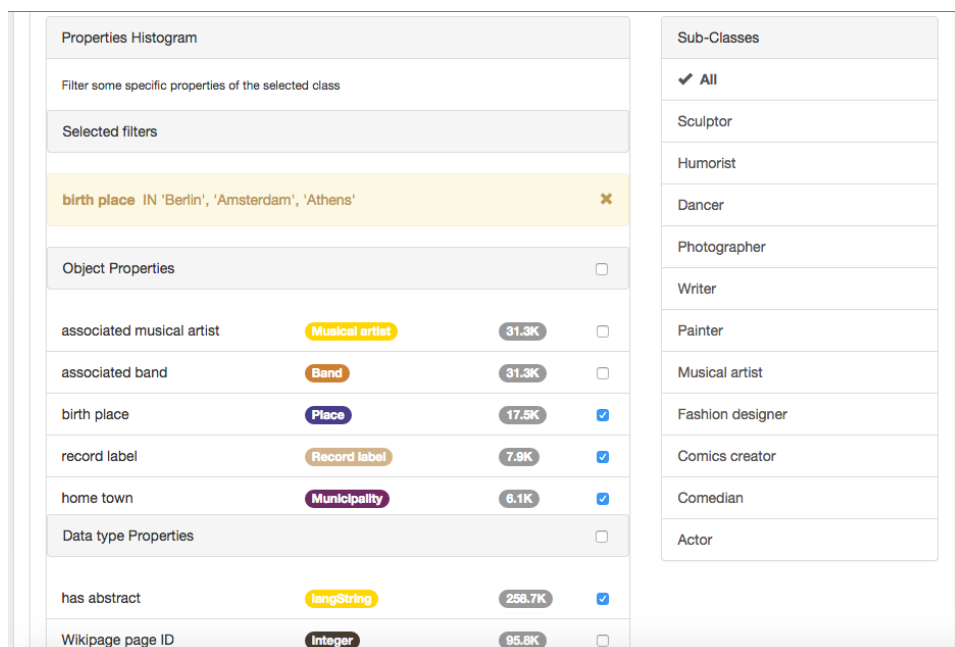


Figure 4.12: Query Builder GUI : Select properties



#### 4.3.4 Equivalent Query

After following the steps which in the previous sub-sections, an equivalent SPARQL query is formed (fig. 4.13). By default the selected properties are not displayed in the return variables as this will look fairly cluttered to the user. But there is a button (fig. 4.13) clicking on which will display the properties as return elements too. We can see in the fig. 4.14 how cluttered it looks once its set to "YES". There is a small textbox in which the limit can be specified.

Generating the equivalent query is tricky. We have implemented this using Javascript. The main logic involved in it has 4 parts.

1. First the return variables are formed. This is based on the selection of whether to return selected properties or not. This is the SELECT part
2. Then the WHERE clause is populated by going through the selected property filters.
3. The FILTER is populated next. This is usually based on whether *Data type* property filters have been added or not.
4. Finally the LIMIT is generated from the LIMIT textbox.

**Equivalent SPARQL Query** Yes No

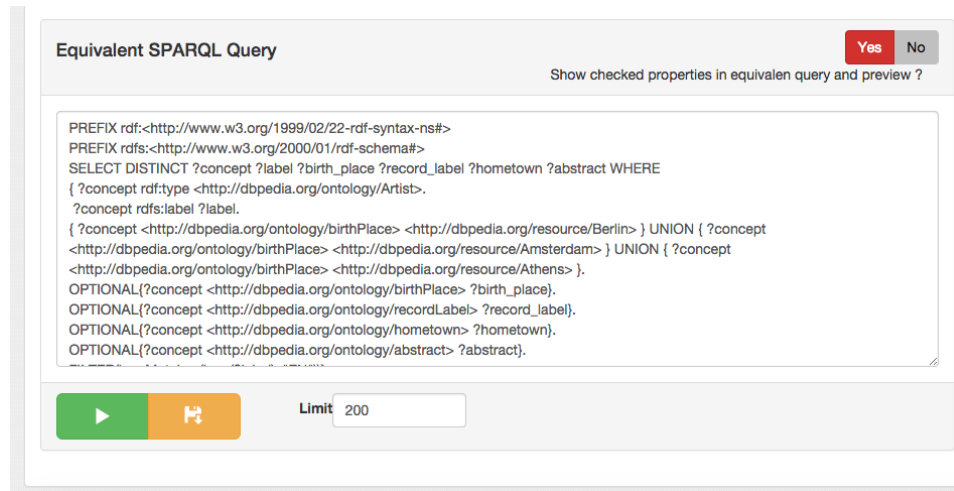
Show checked properties in equivalent query and preview ?

```
PREFIX rdf:<http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX rdfs:<http://www.w3.org/2000/01/rdf-schema#>
SELECT DISTINCT ?concept ?label WHERE
{ ?concept rdf:type <http://dbpedia.org/ontology/Artist>.
  ?concept rdfs:label ?label.
  { ?concept <http://dbpedia.org/ontology/birthPlace> <http://dbpedia.org/resource/Berlin> } UNION { ?concept
    <http://dbpedia.org/ontology/birthPlace> <http://dbpedia.org/resource/Amsterdam> } UNION { ?concept
    <http://dbpedia.org/ontology/birthPlace> <http://dbpedia.org/resource/Athens> }.
  FILTER(langMatches(lang(?label), "EN")) }
LIMIT 200
```

Limit 200

**Figure 4.13:** Query Builder GUI : Equivalent query

For non-native SPARQL users this is not much of a help but this feature is helpful for native SPARQL users. Nevertheless, the next step will be to either download the result set or to get a preview of the result set.



**Figure 4.14:** Query Builder GUI : Equivalent query showing all the selected properties

#### 4.3.5 Result Set Preview

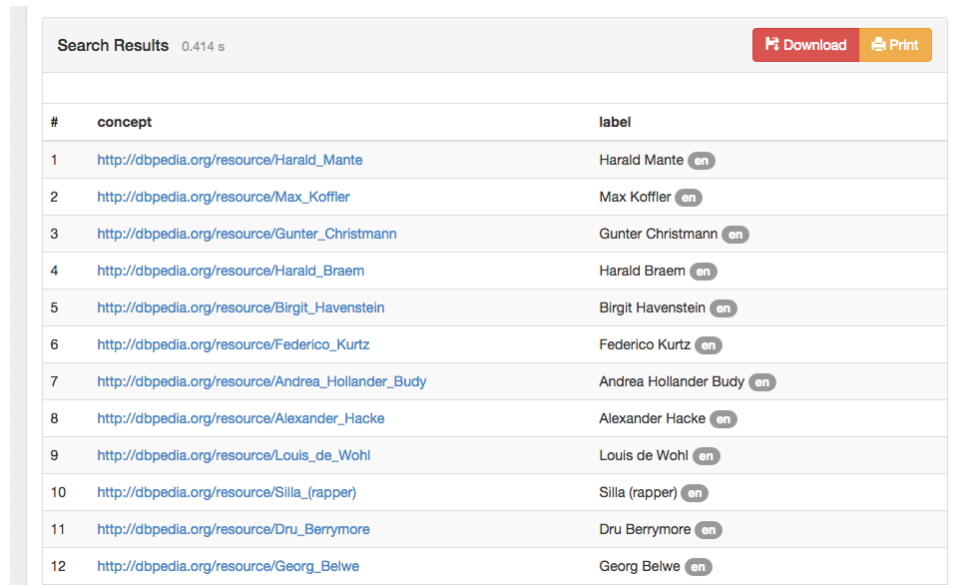
The user has an option to preview the Result Set by clicking the Play icon (fig. 4.13). This will display the results in the same GUI tool (fig. 4.15). This is useful as the user gets a picture of what objects he will be abstracting. These results are populated from our JSON convert API (section 4.2.2.3). The user can click on the object's uri and it will open in our URI Viewer. The time taken for displaying this result set is also displayed. Notice in the fig. 4.15 how beautifully the language of the labels are displayed as a badge.

#### 4.3.6 Result Set Download

Using this GUI tool, the user can abstract the data in various formats. For that a couple of download buttons are provided. One is at the bottom of the equivalent query (fig. 4.13), and the other one at the top right of the results preview (fig. 4.15). On clicking of any of these download buttons a Download modal pops up (fig. 4.16) which has various output formats.

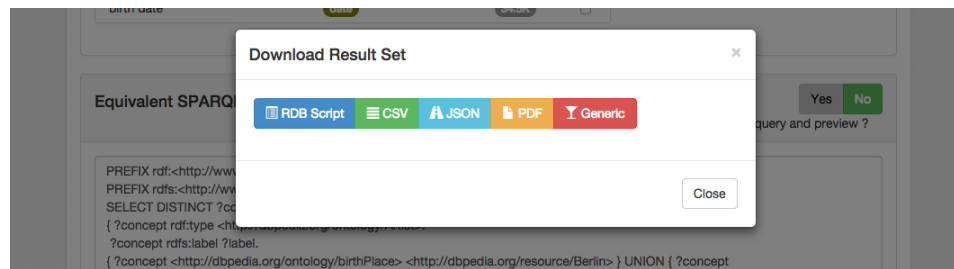
By clicking on the "RDB Script" the user can download the RDBMS upload scripts for the output result set. The *RDB Conversion* API (section 4.2.2.2) is consumed to provide this output. Similarly, the result set can be abstracted to a CSV format by clicking on the "CSV" button. This consumes the *CSV conversion* API (section 4.2.2.1).

If the user clicks the "JSON" button then three download formats for JSON shows up (fig. 4.17). All the 3 JSON convert downloads are done by the *JSON Conversion* API (section 4.2.2.3). The first one is the standard W3C recommended version for converting SPARQL query results. We are



#	concept	label
1	<a href="http://dbpedia.org/resource/Harald_Mante">http://dbpedia.org/resource/Harald_Mante</a>	Harald Mante <span>en</span>
2	<a href="http://dbpedia.org/resource/Max_Koffler">http://dbpedia.org/resource/Max_Koffler</a>	Max Koffler <span>en</span>
3	<a href="http://dbpedia.org/resource/Gunter_Christmann">http://dbpedia.org/resource/Gunter_Christmann</a>	Gunter Christmann <span>en</span>
4	<a href="http://dbpedia.org/resource/Harald_Braem">http://dbpedia.org/resource/Harald_Braem</a>	Harald Braem <span>en</span>
5	<a href="http://dbpedia.org/resource/Birgit_Havenstein">http://dbpedia.org/resource/Birgit_Havenstein</a>	Birgit Havenstein <span>en</span>
6	<a href="http://dbpedia.org/resource/Federico_Kurtz">http://dbpedia.org/resource/Federico_Kurtz</a>	Federico Kurtz <span>en</span>
7	<a href="http://dbpedia.org/resource/Andrea_Hollander_Budy">http://dbpedia.org/resource/Andrea_Hollander_Budy</a>	Andrea Hollander Budy <span>en</span>
8	<a href="http://dbpedia.org/resource/Alexander_Hacke">http://dbpedia.org/resource/Alexander_Hacke</a>	Alexander Hacke <span>en</span>
9	<a href="http://dbpedia.org/resource/Louis_de_Wohl">http://dbpedia.org/resource/Louis_de_Wohl</a>	Louis de Wohl <span>en</span>
10	<a href="http://dbpedia.org/resource/Silla_(rapper)">http://dbpedia.org/resource/Silla_(rapper)</a>	Silla (rapper) <span>en</span>
11	<a href="http://dbpedia.org/resource/Dru_Berrymore">http://dbpedia.org/resource/Dru_Berrymore</a>	Dru Berrymore <span>en</span>
12	<a href="http://dbpedia.org/resource/Georg_Belwe">http://dbpedia.org/resource/Georg_Belwe</a>	Georg Belwe <span>en</span>

**Figure 4.15:** Query Builder GUI : Preview of Result Set

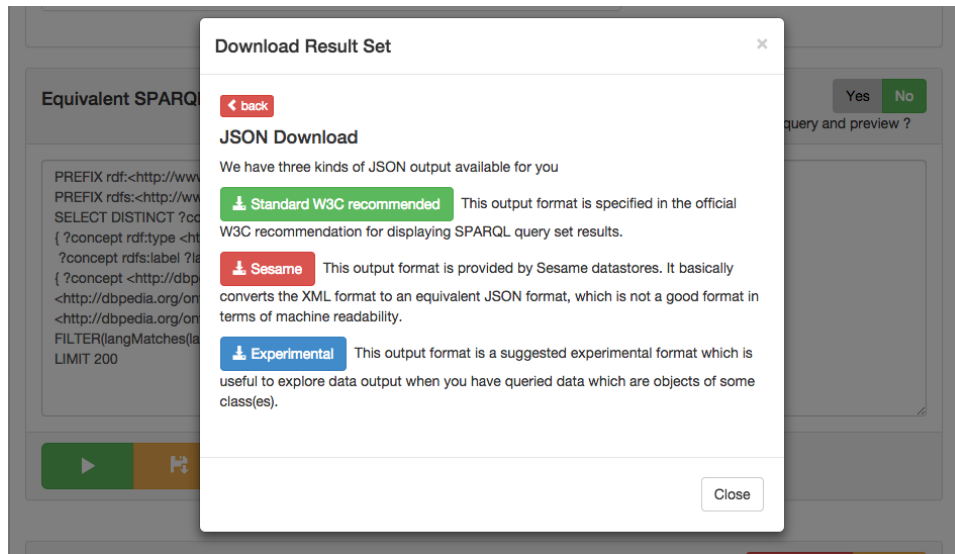


**Figure 4.16:** Query Builder GUI : The result set download modal

actually using this Conversion API to display the Preview results (section 4.3.5). The second download option is the inefficient format for SPARQL query results provided by Sesame<sup>19</sup> servers (This has been discussed more in detail in page 46 of section 4.2.2.3). And finally, the last one is the experimental format to get more information about the properties and classes along with the abstracted result set.

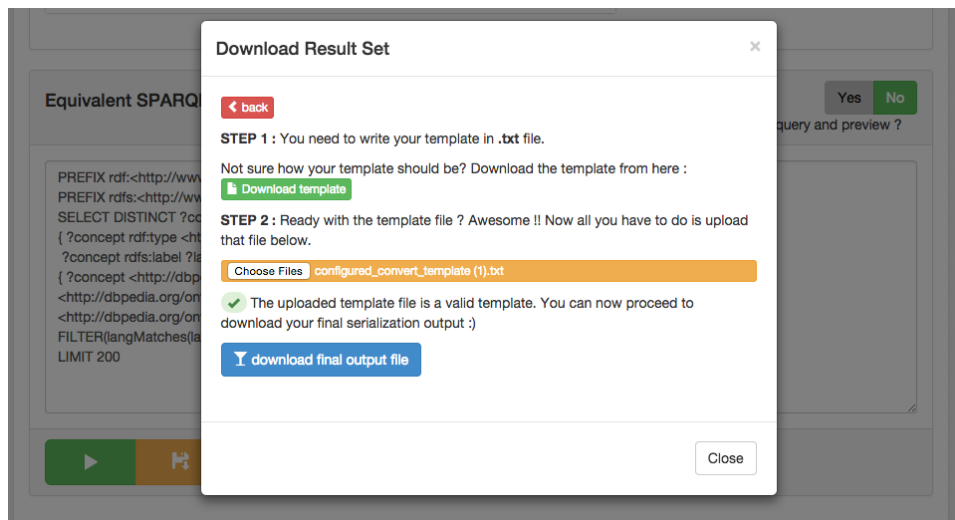
The final download option is the Generic one which shows up on clicking the "Generic" button. Using this generic download option the user can create an output template which will reflect how the output should look like. This gives the user a lot of flexibility on the format in which he wants to abstract the data to. This is done by consuming the *Generic download API* (section 4.2.2.4). On clicking that an interactive modal shows up showing the various

<sup>19</sup><http://rdf4j.org/>



**Figure 4.17:** Query Builder GUI : The JSON download

steps the user has to follow to provide an output template for the generic download (fig 4.18).

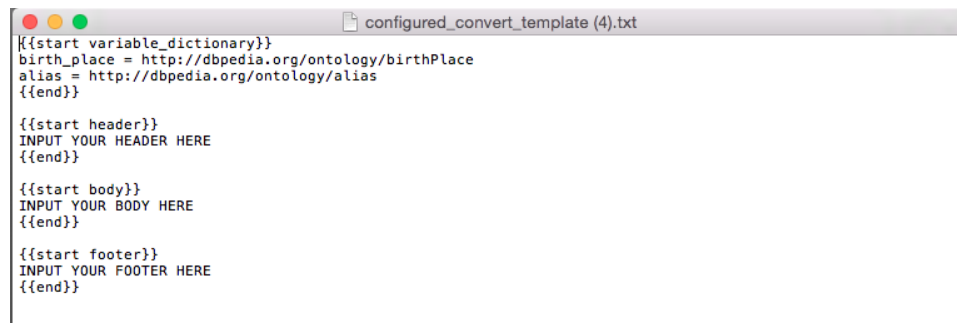


**Figure 4.18:** Query Builder GUI : The Generic download

The user is then shown a download button using which he can download the template. The downloaded template is a simple .txt file which has four sections (fig. 4.19). In the first section the variables for the properties are defined. By default this is already filled out by the Query Builder tool so

that the user does not have to copy paste the URIs of the properties. After this there are sections for *header*, *body* and *footer*. The user has to fill out in these sections. Fig. 4.20 shows a finished template file. Once it has been filled, the user has to upload it (fig. 4.19). If the format is proper then the final download option is displayed. This will abstract the data according to the user's template.

We decided to go with this approach rather than providing a UI where the user can fill out the template details because this gives the user more freedom to add new lines, tabs, etc. Also, since the file is in his local machine, he can save it and there is less chances of data loss due to internet connection loss.



```

configured_convert_template (4).txt
{{start variable_dictionary}}
birth_place = http://dbpedia.org/ontology/birthPlace
alias = http://dbpedia.org/ontology/alias
{{end}}

{{start header}}
INPUT YOUR HEADER HERE
{{end}}

{{start body}}
INPUT YOUR BODY HERE
{{end}}

{{start footer}}
INPUT YOUR FOOTER HERE
{{end}}

```

**Figure 4.19:** Query Builder GUI : A Sample Generic download template



```

configured_convert_template (4).txt
{{start variable_dictionary}}
birth_place = http://dbpedia.org/ontology/birthPlace
alias = http://dbpedia.org/ontology/alias
{{end}}

{{start header}}
<?xml>
<comedians>
{{end}}

{{start body}}
<comedian uri="[$URI]" rowID="[$OBJECT_COUNTER]">
  <name>[$NAME]</name>
  <birthPlace>[$birth_place]</birthPlace>
  $[if alias]<alias>[$alias]</alias>[$end]
</comedian>
{{end}}

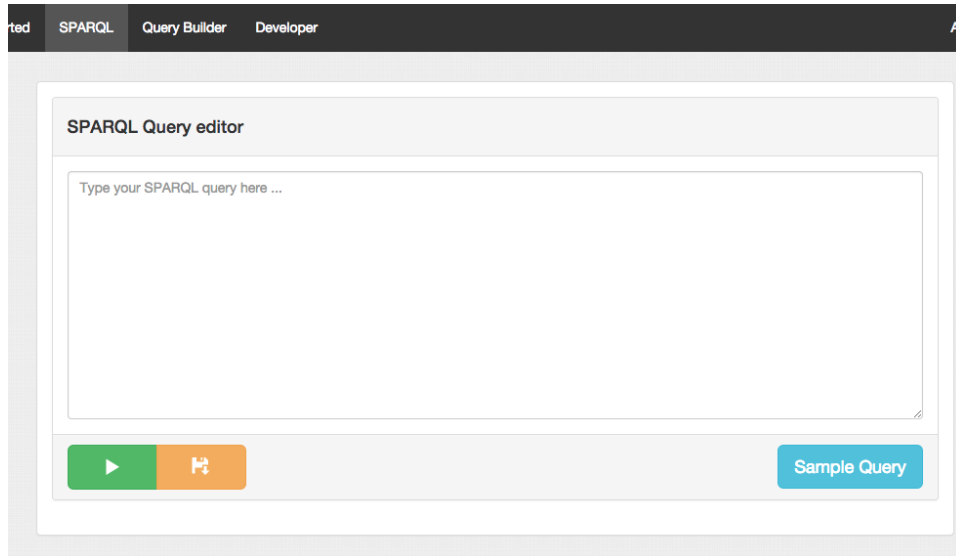
{{start footer}}
</comedians>
{{end}}

```

**Figure 4.20:** Query Builder GUI : A Sample Generic download template after filled by the user

### 4.3.7 SPARQL endpoint

The GUI tool also has a SPARQL end point, in which a query editor is provided where native SPARQL users can directly type SPARQL queries (fig. 4.21). Alternatively, the native users can also use the Query Builder's equivalent query editor to modify the query further (fig. 4.13).



**Figure 4.21:** Query Builder GUI : SPARQL endpoint

### 4.3.8 A Use Case

Now that we have explained the functionalities of the Query Builder tool, we will now look into a use case. Let us say there is a user, who wants to have a list of *artists* who are born in the cities *Berlin* and *Athens*. The user is just interested in getting some description of the *Artist* in different languages, its place of birth and date of birth. The user wants to export this data to his PostgreSQL database. So now the user will follow the following steps

1. First the user will select "DBpedia" from the list of datasets (fig. 4.5).
2. Then in the class search textbox, the user will start typing "Artist". You can see in the fig. 4.22 that recommendations based on the search already show up. The user will now click on "Artist".
3. The user now gets some information about *Artist*. If he clicks the '+' button then the subclasses of *Artist* show up. This is the class histogram (fig. 4.23). On checking this histogram, the user realises that he is just interested in *actors* and *musical artists* among *artists*.
4. Since the user just wants to extract the properties *abstract*, *birth place*

**Query Builder** : Please follow the steps to build your query

**STEP 1 : Select Dataset**

<http://dbpedia.org/sparql> ✕

**STEP 2 : Select Class**

artist|

- artist
- artist discography
- classical music artist
- marital artist
- musical artist
- period of artistic style

**Figure 4.22:** Use Case : Searching for the classes matching *Artist*

**STEP 1 : Select Dataset**

<http://dbpedia.org/sparql> ✕

**STEP 2 : Select Class**

**Artist** 96.3K ( Aaron Lines, Alex Reid (actress), Alma Cogan, Andrew Foley (writer), Andy Park (comics) ) ✕

**actor** 6.5K ( Alex Reid (actress), Henri Cogan, Åke Fridell, Aaron Lawrence (entrepreneur), Alexis Conran ) + 🔍

**comedian** 1.2K ( Tom Wrigglesworth, Charles Firth (comedian), Charlie Callas, Chas Licciardello, Chris Stevey ) 🔍

**comics creator** 2.5K ( Andrew Foley (writer), Andy Park (comics), Jack Sparling, Juan Giménez, Éric Thériault ) 🔍

**fashion designer** 656 ( Anamika Khanna, Charlotte Ronson, Cristi Conaway, David Koma, Dominique Sirop ) 🔍

**musical artist** 45.1K ( Aaron Lines, Alma Cogan, Andy Park (musician), Bob Hames, Gene Rockwell ) + 🔍

**painter** 2.9K ( Ferdinand Fellner (painter), Albert Harjo, Ambrogio Besozzi, Ambrogio Casati, Angelica Veronica Airola ) 🔍

**writer** 26K ( David Wellington (author), Jill Soloway, Junnosuke Yoshiyuki, Juvenal, Kate Constable ) + 🔍

**photographer** 453 ( Anita Pollitzer, David Montgomery (photographer), Ian Shive, Joel Sternfeld, John Florea ) 🔍

**dancer** 0 🔍

**humorist** 0 🔍

**sculptor** 0 🔍

More details on Artist

**Figure 4.23:** Use Case : Browsing the *Artist* class histogram

and *birth date*, he checks only the checkboxes in the property histogram which are adjacent to them (fig. 4.24).

- As the user has realised that he is just interested in *musical artists* and *artists*, the user checks them from the subclass filter (fig. 4.24).
- Now the user needs to add a filter that the user is born in *Athens* and *Berlin*. He clicks on the property histogram tile for *birth place* and a filter opens up to add the property object filter (check out fig. 4.11 on

The screenshot shows a web interface for selecting properties for the class 'Artist'. It is divided into two main sections: 'Properties Histogram' and 'Sub-Classes'.

**Properties Histogram:**

- Header: Properties Histogram
- Filter: Filter some specific properties of the selected class
- Object Properties (checkbox):
  - associated musical artist: Musical artist (31.3K) [checkbox]
  - associated band: Band (31.3K) [checkbox]
  - birth place: Place (17.5K) [checked]
  - record label: Record label (7.9K) [checkbox]
  - home town: Municipality (8.1K) [checkbox]
- Data type Properties (checkbox):
  - has abstract: langString (258.7K) [checked]
  - Wikipedia page ID: Integer (95.6K) [checkbox]
  - Wikipedia revision ID: Integer (95.6K) [checkbox]
  - VIAF Id: string (41.6K) [checkbox]
  - birth date: date (34.5K) [checked]

**Sub-Classes:**

- All
- Sculptor
- Humorist
- Dancer
- Photographer
- Writer
- Painter
- ✓ Musical artist
- Fashion designer
- Comics creator
- Comedian
- ✓ Actor

**Figure 4.24:** Use Case : Selecting the properties which the user wants for the class, *Artist*

how this filter UI looks like).

- The next step will be to preview the data. The user clicks on the "play" button to see the results.
- If he is happy that this is the set of objects he wants to extract, then all he has to do is click on the "download" button and then opt for "RDB download" (fig. 4.16).
- Once the download is finished, the user just needs to run the downloaded **.sql** file in his PostgreSQL database and all the downloaded data with the relationships will be present in the database.



## Chapter 5

# Evaluation

We have evaluated our work in three main categories. First we evaluated the **usability** of the Query Builder tool. Then we evaluated the conversion module and discovered some gaps in it. Our main focus in this one was to find out the **technical** gaps in the generic conversion solution(section 4.2.2.4). Finally, we made a comparison of the **performance** of the different conversion modules.

### 5.1 Usability Feedback of Query Builder

It was important to evaluate the usability of the Query builder tool developed. The evaluation was done by regular feedbacks provided by the Linda<sup>1</sup> working group. The framework developed is part of the Linda project, which is also consuming the Exploration and Conversion REST API and have built their own Query Builder which consumes this. The following findings on usability were made :

1. Initially in the properties histogram, there were three categories, viz., *ObjectType*, *DataType* and a third one which contained the properties defined with the *rdfs:domain* ontology. Since this was found to be confusing to the users, it was later changed to just two categories, the *ObjectType* and *DataType* properties (fig. 4.10).
2. The property histogram initially was very long and took up most of the screen. It did not have any scrollable window. Because of this, it was suggested to have a fixed size for that with a scrollable window. This change was incorporated accordingly (fig. 4.10).
3. It was pointed out that the user should be given the control on whichever properties they want to extract instead of getting all the properties. So a checkbox in the property histogram (check on the right side of fig. 4.10) was added. Only the checked properties are ex-

---

<sup>1</sup>Linda is a European project <http://linda-project.eu/>

tracted. There is also a "check all" option, clicking on which will check all the properties, which is useful if the user wants to extract all the properties.

4. In the property histogram, the properties are arranged in the descending order of the count of the range objects. It would have been useful if a user is given more options to arrange them, say by alphabetically, range classes, etc. This has not been implemented and is part of future work.
5. While filtering for object properties, it was suggested that a *NOT EQUALS* option should also be added to match for inequalities. Based on that two options for *EQUALS* and *NOT EQUALS* have been added (fig. 4.11).
6. The only way to browse for *classes* is by searching through the textbox (fig. 4.6). This might be a bit problematic, if the user is not familiar with the schema underlying the datastore and does not know what *classes* are present in the datastore. It has been hence suggested to have an option for the user to browse for classes without searching. It should be shown in a paginated manner. This has not been implemented and is part of future work.
7. It was pointed out that the user might get confused when the preview results are shown, as it shows just the object's name and its uri (fig. 4.13) and does not show the selected properties. So to avoid this, there has to be an option to even show the properties in the results preview. Because of that a button has been added so that the user can choose if they want to view all the selected properties in the results preview (see the button on the right side and not the difference in equivalent queries of fig. 4.13 and fig. 4.14). This option is made as if the properties are also displayed then the screen becomes cluttered as a class can have even 50 properties. The user can see the properties of that object by just clicking on it, which opens the URI of the object in the URI viewer. This feature was developed to cater for the users who are familiar with SPARQL and would like to edit the SPARQL query before executing.
8. It was suggested to have a class histogram also, so that the user can get a sense of the class's sub-classes. Because of this a class sub-classes has been added (fig. 4.9).
9. It was suggested that some example instances of the selected class with the total number of objects it contains, should be displayed. This was added in later iterations of the product development (fig. 4.9).
10. The addition of filters for *Data Type* properties can be improved. Right now, it just has a textbox where the user can input the filter. Now it will be useful if some drag-and-drop UI features can be added to it so that it becomes simpler for the user to add a filter there. This has not been implemented and is part of the future work.

## 5.2 Evaluation of Conversion module

### 5.2.1 JSON Conversion

Flaws in the existing JSON format has been identified and a new experimental format which is suitable to our approach has been provided. This has been elaborated in detail in the section 4.2.2.3. One big problem with JSON conversion is that, the programming languages which read JSON input, read the whole file or value at once. Its is then treated as a dictionary or a hashmap. So bigger JSON files become useless for consumption by most of the languages. Even while conversion, since the whole output is stored in the memory before it is sent to the client, for larger objects, this becomes sluggish. And for huge number of objects, the conversion fails sometimes by running out of memory. Usually for larger sets XML serialization is recommended.

### 5.2.2 CSV Conversion

The CSV conversion works well for all of the classes tested. To test it, it was uploaded in a an excel sheet and it was checked if the entries show up correctly. Now there is no official W3C format to display RDF data with the approach we are taking, into CSV format. We are having the properties of the class, which is being downloaded, as the column names of the CSV output file. For SPARQL query result conversions to CSV, the variable names become the column names of the CSV file. This had been discussed more in detail in the previous chapter, in section 4.2.2.1.

### 5.2.3 RDB Conversion

The RDB conversion works well for all of the classes tested. To test this, the upload scripts generated was run in a PostgreSQL terminal. The tables, its tuples and its relations are then checked manually. These upload scripts do not work for any other database. It will be a good idea to also provide upload scripts for MySQL databases. This won't be a difficult task, as only the table creation scripts will be different. The INSERT and UPDATE scripts for both these databases are the same.

### 5.2.4 Generic Conversion

For evaluation of the Generic conversion, we tried to reproduce some standard serializations, using the generic conversion described in the section 4.2.2.4. We then try to find out the gaps of this solution.

#### 5.2.4.1 CSV serialization using Generic Conversion

Conversion of CSV can be done very easily using this solution. Let us again go back to DBpedia. Here we will convert the objects of the class *Comedian*. For simplicity we will just extract two properties, viz., *birth place*, and *birth date*. First we pass the comma separated column headers as the *header*

```
1 rowId, uri, label, birthPlace, birthDate
```

Then we pass the following *body*.

```
1 $[=OBJECT_COUNTER], $[=URI], $[=NAME], "$[=birth_place]", $[=birth_date]
```

This gives a correct CSV output.

```
1 rowID, uri, label, birthPlace, birthName
2 1, http://dbpedia.org/resource/Tom_Wrigglesworth, Tom Wrigglesworth,
   ↪ "http://dbpedia.org/resource/Sheffield;
   ↪ http://dbpedia.org/resource/South_Yorkshire", 1976-05-05+02:00
3 2, http://dbpedia.org/resource/Charles_Firth_(comedian), Charles Firth
   ↪ (comedian), "",
4 3, http://dbpedia.org/resource/Charlie_CallasC, harlie Callas,
   ↪ "http://dbpedia.org/resource/Brooklyn", 1927-12-20+02:00
5 4, http://dbpedia.org/resource/Chas_Licciardello, Chas Licciardello,
   ↪ "", 1977-05-10+02:00
6 ...
```

The user can also select any other separator apart from ",". TSV<sup>2</sup>, can be achieved by replacing "," with "\t". Another useful format are the RRF files which are upload scripts expected by some relational databases. In that values are separated by "|".

#### 5.2.4.2 XML serialization using Generic Conversion

We will now attempt a standard RDF/XML output using the generic conversion. We again will convert the objects of the class *Comedian*. We will just extract three properties, viz., *birth place*, *birth date* and *abstract*. We will use the following *header*.

```
1 <?xml version="1.0"?>
2
3 <rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
   ↪ xmlns:dbpedia-owl="http://dbpedia.org/ontology/"
   ↪ xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#">
```

The following *body* will be used.

```
1 <rdf:Description rdf:about="$[=URI]">
2 <rdfs:class>
3 <rdf:Description rdf:about="http://dbpedia.org/ontology/Comedian">
4 </rdfs:class>
5 <rdfs:label>${[=NAME]}</rdfs:label>
6 ${[for b:birth_place]}
```

---

<sup>2</sup>Tab separated values

```

7   <dbpedia-owl:birthPlace>
8     <rdf:Description rdf:about="[$=b]" />
9   </dbpedia-owl:birthPlace>
10  $[end]
11  $[if birth_date] <dbpedia-owl:birthDate> $[=birth_date]
12    ↪ </dbpedia-owl:birthDate> $[end]
13  $[for a:abstract]
14    <dbpedia-owl:abstract
15      ↪ xml:lang="[$a.lang]">[$a.text]</dbpedia-owl:abstract>
16    $[end]
17  </rdf:Description>

```

We will need a closing tag. So we will use the following *footer*

```
1 </rdf:RDF>
```

The following output is obtained.

```

1 <?xml version="1.0"?>
2
3 <rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
4   ↪ xmlns:dbpedia-owl="http://dbpedia.org/ontology/"
5   ↪ xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#">
6   <rdf:Description
7     ↪ rdf:about="http://dbpedia.org/resource/Tom_Wrigglesworth">
8     <rdfs:label>Tom Wrigglesworth</rdfs:label>
9     <dbpedia-owl:birthPlace>
10      <rdf:Description
11        ↪ rdf:about="http://dbpedia.org/resource/Sheffield" />
12      </dbpedia-owl:birthPlace>
13      <dbpedia-owl:birthPlace>
14        <rdf:Description
15          ↪ rdf:about="http://dbpedia.org/resource/South_Yorkshire" />
16        </dbpedia-owl:birthPlace>
17        <dbpedia-owl:birthDate>1976-05-05+02:00</dbpedia-owl:birthDate>
18        <dbpedia-owl:abstract xml:lang="en">...</dbpedia-owl:abstract>
19        <dbpedia-owl:abstract xml:lang="de">...</dbpedia-owl:abstract>
20        <dbpedia-owl:abstract xml:lang="es">...</dbpedia-owl:abstract>
21      </rdf:Description>
22    </dbpedia-owl:birthPlace>
23  </rdf:Description>
24  ...
25 </rdf:RDF>

```

We notice that this output is accurate and our generic conversion features are sufficient to provide it. There is a bit of an issue though. We can see in the template which we provided that we had to hardcode the property names of the class *Comedian*. So the same template cannot be used to download objects of another class. Therefore, it might be useful if the language had a variable say *PROPERTIES*, in which we can iterate through all of the properties. It might be also better if this supports some other *conditions* such as checking for property type. A suggested rewrite can be with such changes :

```

1  <rdf:Description rdf:about="[$=URI]">
2    <rdfs:class>
3      <rdf:Description rdf:about="[$=OBJECT.class_uri]">
4    </rdfs:class>
5    <rdfs:label>[$=NAME]</rdfs:label>
6    $[for property:PROPERTIES]
7    $[for p:property]
8    <dbpedia-owl:[$=property.name] $[if
9      ↪ property.language_literal]xml:lang="[$=p.lang]"$[end]>
10   $[if property.type = "OBJECT"]
11     <rdf:Description rdf:about="[$=p]" />
12   $[else]
13     $[=p.text]
14   $[end]
15 </dbpedia-owl:[$=property.name]>
16 $[end]
17 </rdf:Description>

```

We can see now there is no hardcoding. We are iterating through the *PROPERTIES* and printing our result according to the property.

#### 5.2.4.3 Turtle serialization using Generic Conversion

Conversion to Turtle format can be done very easily using this solution. Again we will convert the objects of the class *Comedian* from DBPedia. For simplicity we will just extract two properties, viz., *birth place*, and *birth date*. First we pass the prefixes in the *header*.

```

1 @prefix rdf:<http://www.w3.org/1999/02/22-rdf-syntax-ns#>.
2 @prefix rdfs:<http://www.w3.org/2000/01/rdf-schema#>.
3 @prefix dbpedia-owl:<http://dbpedia.org/ontology/>.

```

Now we pass the following *body*.

```

1 <[$=URI]> rdfs:class dbpedia-owl:Comedian.
2 <[$=URI]> rdfs:label "[$=NAME]"@EN.
3 $[for b:birth_place]
4 <[$=URI]> dbpedia-owl:birthPlace <[$=b]>.
5 $[end]
6 $[if birth_date]<[$=URI]> dbpedia-owl:birthDate
7   ↪ "[$=birth_date]"^^<http://www.w3.org/2001/XMLSchema#date>. $[end]

```

This will give the following output.

```

1 @prefix rdf:<http://www.w3.org/1999/02/22-rdf-syntax-ns#>.
2 @prefix rdfs:<http://www.w3.org/2000/01/rdf-schema#>.
3 @prefix dbpedia-owl:<http://dbpedia.org/ontology/>.
4 <http://dbpedia.org/resource/Tom_Wrigglesworth> rdfs:class
5   ↪ dbpedia-owl:Comedian.
6 <http://dbpedia.org/resource/Tom_Wrigglesworth> rdfs:label "Tom
7   ↪ Wrigglesworth"@EN.
8 <http://dbpedia.org/resource/Tom_Wrigglesworth> dbpedia-owl:birthPlace
9   ↪ <http://dbpedia.org/resource/Sheffield>.

```

```

7 <http://dbpedia.org/resource/Tom_Wrigglesworth> dbpedia-owl:birthPlace
  ↪ <http://dbpedia.org/resource/South_Yorkshire>.
8 <http://dbpedia.org/resource/Tom_Wrigglesworth> dbpedia-owl:birthDate
  ↪ "1976-05-05+02:00"^^<http://www.w3.org/2001/XMLSchema#date>.
9
10 <http://dbpedia.org/resource/Charles_Firth_(comedian)> rdfs:class
  ↪ dbpedia-owl:Comedian.
11 <http://dbpedia.org/resource/Charles_Firth_(comedian)> rdfs:label
  ↪ "Charles Firth (comedian)"@EN.
12 ....

```

This output is a valid Turtle format. But we notice that there are some redundancies. eg, it is redundant to print the whole URI of the object every time. It can be prefixed with *dbpedia-resource*. The following shows how the output format should ideally look.

```

1 @prefix rdf:<http://www.w3.org/1999/02/22-rdf-syntax-ns#>.
2 @prefix rdfs:<http://www.w3.org/2000/01/rdf-schema#>.
3 @prefix dbpedia-owl:<http://dbpedia.org/ontology/>.
4 @prefix dbpedia-resource:<http://dbpedia.org/resource/>.
5 dbpedia-resource:Tom_Wrigglesworth rdfs:class dbpedia-owl:Comedian.
6 dbpedia-resource:Tom_Wrigglesworth rdfs:label "Tom Wrigglesworth"@EN.
7 dbpedia-resource:Tom_Wrigglesworth dbpedia-owl:birthPlace
  ↪ <http://dbpedia.org/resource/Sheffield> ;
  ↪ <http://dbpedia.org/resource/South_Yorkshire>.
8 dbpedia-resource:Tom_Wrigglesworth dbpedia-owl:birthDate
  ↪ "1976-05-05+02:00"^^<http://www.w3.org/2001/XMLSchema#date>.
9
10 dbpedia-resource:Charles_Firth_(comedian) rdfs:class
  ↪ dbpedia-owl:Comedian.
11 dbpedia-resource:Charles_Firth_(comedian) rdfs:label "Charles Firth
  ↪ (comedian)"@EN.
12 ....

```

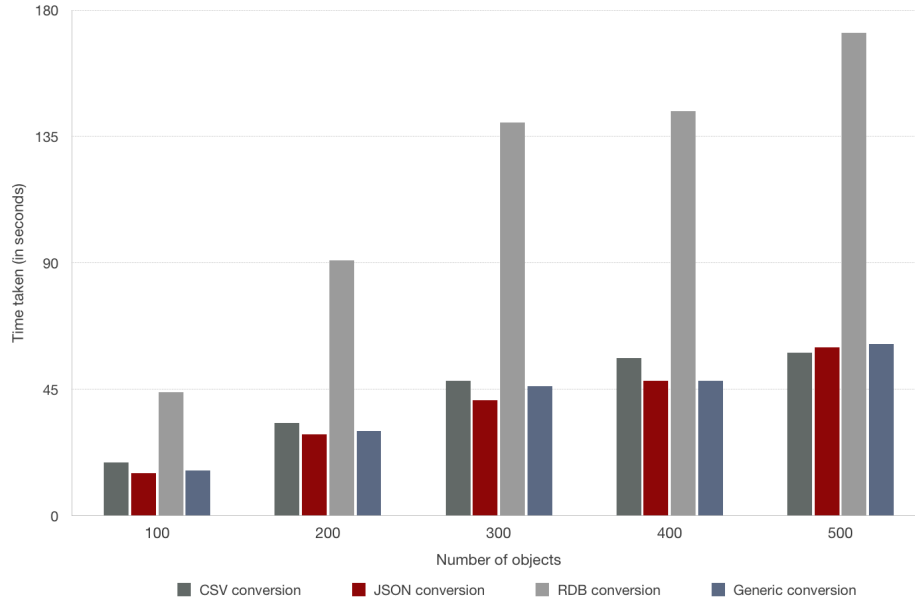
This can easily be achieved by adding a feature to the generic conversion. There can be a feature just to print the URI's last value instead of the whole value. e.g., from the URI `http://dbpedia.org/resource/Tom_Wrigglesworth`, just printing `Tom_Wrigglesworth`.

#### 5.2.4.4 RDBMS serialization using Generic Conversion

The difficult aspects of RDBMS serializations are building table structures, normalization, generating unique primary keys and linking foreign keys to them. We can see that, with generic conversion, we can probably hard code the table structures in the *header* but the normalization could be tricky and will require the user to know the property details. This is still possible as these information are provided through our Exploration REST API (Class properties API in section 4.2.1.2). But generating primary keys and foreign keys becomes too complex for this solution. This solution therefore has some gaps when it comes to RDBMS serialization, and such kind of output becomes too overwhelming for this solution.

### 5.3 Performance

To evaluate the performance of the software, its speed and efficiency needs to be compared to its counterpart. This is difficult as extracting RDF data as *objects* of *classes* is a unique approach in itself. Fig. 5.1 shows the time



**Figure 5.1:** Time taken for conversions of the class *Person*<sup>3</sup>

performance when the different conversion modules are called for the conversion of the class *Person* in DBpedia. We found out that the RDB conversion takes much more time than the other conversion techniques. This is because it has an internal lookup feature which generates the foreign keys and its linking to primary keys which are all stored in the memory as a hashmap. Also, the RDB conversion .sql files are larger as they have more number of lines.

The performance here is measured with the *number of objects*, ranging from 100-500, to the *time taken*, in seconds. In practical use, the number of objects will be much more, but the time taken will be proportionate to what is found in our results. This is because, nothing is stored in the memory during the conversion process. The output is written on the fly in an output stream. This performance can be drastically improved by utilizing the multi-threading capabilities of Java.

<sup>3</sup>All the properties of the class, *Person* have been extracted for the conversion without any filters.



## Chapter 6

# Conclusion and Future Work

In the previous chapter we evaluated our thesis work. We found out the gaps in our solution and also on ways by which the performance can be improved. In this chapter, we will summarize the thesis briefly and will provide some recommendations for future work.

### 6.1 Summary

The work described in the thesis has been concerned with the development of a solution for better exploration and conversion of RDF Data. Due to the lack of existing tools to achieve this, a state-of-the-art solution has been proposed. Taking into consideration a lot of good things from the existing tools, a RESTful solution to the problem has been implemented so that it can be enhanced further by other developers with their own exploration and conversion tools (section 4.2). A working prototype of the framework for the conversion and exploration solution has been successfully implemented. This framework has been successfully extended by a prototype of a Query Builder GUI tool which we implemented, and this consumes the APIs provided by the framework (section 4.3). Many features have been developed in the framework to improve its usability. Various modern Information Retrieval techniques have been utilized in the exploration module to improve its performance. A powerful state-of-the-art RDB conversion module was developed which has improved the flaws in the existing tools (section 4.2.2.2). An important contribution was building a solution for generic conversion (section 4.2.2.4). Using that most of the serialization output formats could be achieved (refer to the evaluation for this in section 5.2.4). Some faults have been identified in the existing JSON serialization formats and an improved experimental JSON format, which is suitable to our approach has been provided (4.2.2.3). A Query Builder tool has also been built, which consumes this conversion and exploration API (section 4.3). It is a GUI tool which allows the users, both native and non-native SPARQL users, to effi-

ciently explore the RDF data. The user is able to convert RDF data to the format they desire. This GUI tool, using some complex JavaScript methods, builds an equivalent query from the selections made. The RESTful exploration API is part of a European project, Linda<sup>1</sup>, and is being used by the project to build their own Query Builder GUI tool.

## 6.2 Future Work

Although we have provided a powerful RDF exploration and conversion solution, it can be further improved in a number of ways :

1. **Features to handle multiple classes.** One major thing which needs to be improved is the ability to handle multiple *classes*. Right now our APIs handle exploration and conversion of just one class. This can be improved further by adding changes to handle multiple classes. In the API, the classes should be sent as a comma separated string containing the URIs of the classes.
2. **Usability improvements.** There are some Usability improvements which needs to be incorporated in the Query Builder tool. These had been discussed in detail in chapter 5 for Evaluation in section 5.1. There are a few enhancements, which will improve the user experience.
3. **Multi-threaded conversion approach.** We figured out from the Performance evaluation of the conversion module, that it needs to be improved, for faster conversion for larger number of objects. In future a multi-threaded approach can be used to improve the performance.
4. **Indexed lookup for class search.** The class search in the exploration framework (section 4.2.1.1) can be speeded up by utilizing indexing solutions of Lucene. The Lucene framework has already been imported for denenerating properties of the class, so this can be easily re-used.
5. **Features to explore through multiple datasets.** Right now, our framework supports exploration and conversion in only one dataset or datastore. It will be really useful to have features to do this in multiple datasets. This is challenging, as a SPARQL query can be executed only at one dataset at a time. So, to implement this, we would need to have some kind of federated queries where the same query is executed in the different datasets, and their results are merged according to the query. This feature has a fair bit of complexity attached to it and will require a thorough research before implementation.
6. **RDB Conversion scripts for databases apart from PostgreSQL.** Right now, the RDB conversion API (section 4.2.2.2) returns upload scripts for PostgreSQL databases. It will be a good fea-

---

<sup>1</sup><http://linda-project.eu/>

ture, to also have scripts for other popular databases like MySQL. This is not a difficult task as only the table creation scripts are different. The *INSERT* and *UPDATE* statements are usually same for all the databases.

# References

- [ADLHA+09] J.Lehmann S.Hellman S.Auer S.Dietzold and D.Aumueller. “Triplify - Lightweight Linked Data Publication from Relational Databases”. In: *18th International World Wide Web Conference* (2009).
- [AG+12] Renzo Angles and Claudio Gutierrez. “Querying RDF Data from a Graph Database Perspective”. In: (2012). URL: <http://users.dcc.uchile.cl/~cgutierrez/papers/eswc05.pdf> (visited on 10/17/2014).
- [AKKP+08] Thomas Krennwallner Waseem Akhtar Jacek Kopecky and Axel Polleres. “XSPARQL: Traveling between the XML and RDF worlds – and avoiding the XSLT pilgrimage”. In: (2008).
- [BBPC+14] Eric Prud’hommeaux David Becket Tim Berners-Lee and Gavin Carothers. *RDF 1.1 Turtle - Terse RDF Triple language*. W3C. 2014. URL: <http://www.w3.org/TR/turtle/> (visited on 01/08/2015).
- [BC+11] Tim Berners-Lee and Dan Connolly. *Notation3 (N3): A readable RDF syntax*. W3C. 2011. URL: <http://www.w3.org/TeamSubmission/n3/> (visited on 01/08/2015).
- [BHB+08] Tom Heath Christian Bizer and Tim Berners-Lee. “Linked Data - The Story So Far”. In: (2008).
- [BS+04] C.Bizier and A.Seaborne. “D2RQ - Treating non-RDF Databases as Virtual RDF graphs”. In: *3rd International Semantic Web conference* (2004).
- [C+10] Lin Clark. “SPARQL Views: A Visual SPARQL Query Builder for Druptal”. In: (2010).
- [C+86] Keith Clarke. “The top-down parsing of expressions”. In: *Research Report 383, Dept of Computer Science, Queen Mary College* (1986).

- [CPCDT+12] Diego Ceccarelli Renaud Delbru Stephane Campinas Thomas E.Perry and Giovanni Tummarello. “Introducing RDF Graph Summary with Application to Assisted SPARQL Formulation”. In: (2012).
- [dbpedia-swj] Jens Lehmann, Robert Isele, Max Jakob, Anja Jentzsch, Dimitris Kontokostas, Pablo N. Mendes, Sebastian Hellmann, Mohamed Morsey, Patrick van Kleef, Sören Auer, and Christian Bizer. “DBpedia - A Large-scale, Multilingual Knowledge Base Extracted from Wikipedia”. In: *Semantic Web Journal* (2014).
- [E+04] E.Evans. *Domain-driven design : Tackling Complexity in the heart of Software*. Longman, 2004.
- [EM+07] O.Erling and I.Mukhailov. “RDF Support in Virtuoso DBMS”. In: *1st Conference on Social Semantic Web* (2007).
- [FHHNS+07] Tonya Hongsermeier Eric Neumann Lee Feigenbaum Ivan Herman and Susie Stephens. *The Semantic Web in Action*. 2007.
- [G+93] Tom Gruber. “A Translation Approach to Portable Ontology Specifications”. In: (1993).
- [GS+14] Fabien Gandon and Guus Schreiber. *RDF 1.1 XML Syntax*. W3C. 2014. URL: <http://www.w3.org/TR/rdf-syntax-grammar/> (visited on 12/04/2014).
- [HFPSJ+08] C.Parr J.Sachs L.Han T.Finin and A.Joshi. “RDF123: From spreadsheets to RDF”. In: *International Semantic Web Conference* (2008).
- [HLSZ+10] S.Lohmann P.Heim and T.Stegemann J.Ziegler. “The RelFinder User Interface: Interactive Exploration of Relationships between Objects of Interest”. In: *Proceedings of the 14th International Conference on Intelligent User Interfaces* (2010).
- [HLTE+10] D.Tsendragchaa P.Heim S.Lohmann and T.Ertl. “SemLens: Visual Analysis of Semantic Data with Scatter Plots and Semantic Lenses”. In: *Proceedings of the 7th International Conference on Semantic Systems* (2010).
- [HZL+08] J.Ziegler P.Heim and S.Lohmann. “gFacet: A Browser for the Web of Data”. In: *Proceedings of the International Workshop on Interacting with Multimedia Content in the Social Semantic Web* (2008).

- [LS+99] Ralph R. Swick Ora Lassila and W3C. “Resource Description Framework (RDF) Model and Syntax Specification”. In: (1999). URL: <http://www.w3.org/TR/1999/REC-rdf-syntax-19990222/> (visited on 10/18/2014).
- [MSDN] *Microsoft Developer Network*. Microsoft. URL: <http://msdn.microsoft.com/en-us/>.
- [PRM+11] David C.De Roure Kevin R.Page and Kirk Martinez. “REST and Linked Data : a match made for domain driven development?” In: (2011).
- [PS+13] Andy Seaborne Eric Prud’hommeaux. *SPARQL Query Language for RDF*. W3C. 2013. URL: <http://www.w3.org/TR/rdf-sparql-query/> (visited on 12/04/2014).
- [RGKST+09] Latifur Khan Steven Seida Sunitha Ramanujam Anubha Gupta and Bhavani Thuraisingham. “R2D : A brifge between the Semantic Web and Relational Visualization Tools”. In: *IEEE International Conference on Semantic Computing* (2007).
- [SLKLL+14] Gregg Kellogg Markus Lanthaler Manu Sporny Dave Longley and Niklas Lindström. *JSON-LD 1.0*. W3C. 2014. URL: <http://www.w3.org/TR/json-ld/> (visited on 12/04/2014).
- [SRH+13] Bene Rodriguez-Castro Alex Stolz and Martin Hepp. “RDF Translator: A RESTful Multi-format Data Converter for the Semantic Web”. In: (2013).
- [SS+10] Oshani Seneviratne and Rachel Sealfon. “QueryMed: An Intuitive SPARQL Query Builder for Biomedical RDF Data”. In: (2010).
- [TC+07] W.Teswanich and S.Chittayasothorn. “A Transformation of RDF Documents and Schemas to Relational Databases”. In: *IEEE PacificRim Conferences on Communications, Computers and Signal Processing* (2007).
- [w3c+ontology] *W3C Standards : ontology*. W3C. URL: <http://www.w3.org/standards/semanticweb/ontology> (visited on 01/08/2015).

# APPENDIX

## APPENDIX A : Technical Specifications for Exploration and Conversion API Server

The exploration and Conversion API server has been implemented in Java.

### Project source

It is an open source project and it can be downloaded from the github repository <https://github.com/LinDA-tools/RDF2Any>.

### Requirements

1. **Java** - Version 1.7.
2. **Apache Maven**<sup>2</sup> - Apache Maven is a software project management and comprehension tool. Because of this we do not need to include other libraries which are being used in the project. It will be downloaded automatically when the project is built for the first time.

### Installation

The following is the Installation guide for the operating systems **MacOSX** and **Linux**.

1. The project dependencies has to download by the following command

```
1 mvn clean install
```

### Server Start

To run the API server, execute the following command inside the "linda" directory of the project

```
1 mvn exec:java -X
```

---

<sup>2</sup><http://maven.apache.org/>

## API Documentation

A detailed and comprehensive API documentation can be downloaded from <https://github.com/LinDA-tools/RDF2Any/blob/master/RESTfulAPIsforRDF2Any.pdf>.

## APPENDIX B : Technical Specifications for Query Builder GUI Tool

The Query Builder GUI Tool has been developed in Ruby on Rails

### Project source

It is an open source project and it can be downloaded from the github repository <https://github.com/LinDA-tools/QueryBuilder>.

### Requirements

1. **Ruby** - Version 2.0.
2. **Ruby Version Manager**<sup>3</sup> - RVM is a command-line tool which allows you to easily install, manage, and work with multiple ruby environments from interpreters to sets of gems.

Apart from these, the Exploration and Conversion server must be running on your local machine.

### Installation

The following is the Installation guide for the operating systems **MacOSX** and **Linux**.

1. Ruby version 2.0.0 must be installed using rvm<sup>4</sup>.

```
1 rvm install 2.0.0
```

2. Create a separate environment for your Query Builder project for the dependencies. We will call it "qbuilder". This is called a gemset.

```
1 rvm use 2.0.0@qbuilder
```

3. Load your gemset.

```
1 rvm gemset create qbuilder
```

4. Install rails framework

```
1 gem install rails
```

---

<sup>3</sup><https://rvm.io/>

<sup>4</sup>Ruby version manager



5. Install the gem dependencies of the project.

```
1 bundle install
```

## Server Start

To run the server, execute the following commands directory of the project

```
1 rvm use 2.0.0@qbuilder
2 rails s
```

To run the server as a daemon

```
1 rvm use 2.0.0@qbuilder
2 rails s -d
```

You can now go to the url **<http://localhost:3000/>** to access the Query Builder.