



PONTIFICIA UNIVERSIDAD CATOLICA DE CHILE  
ESCUELA DE INGENIERIA

# **UPDATING SPARQL FEDERATED QUERIES TO INTEGRATE JSON API SOURCES**

**MATTHIEU MOSSER**

Proyecto para optar al grado de  
Magister en Ingeniería UC.

Profesor Supervisor:  
**JUAN REUTTER**

Santiago de Chile, (Agosto, 2017)

© 2017, Matthieu MOSSER




PONTIFICIA UNIVERSIDAD CATOLICA DE CHILE  
ESCUELA DE INGENIERIA

# UPDATING SPARQL FEDERATED QUERIES TO INTEGRATE JSON API SOURCES

MATTHIEU MOSSER

Proyecto presentado a la Comisión integrada por los profesores:

JUAN REUTTER 

DOMAGOJ VRGOC 

CARLOS BUIL ARANDA 

RODRIGO ESCOBAR MORAGA 

Para completar las exigencias del grado de  
Magister en Ingeniería UC.

Santiago de Chile, (Agosto, 2017)

## ACKNOWLEDGEMENTS

First, I would like to thank Juan L. Reutter, Assistant Professor in the Pontificia Universidad Catolica de Chile, for his proposition to collaborate on this interesting project, his availability, suggestions and understanding relative to my situation as international student. Then, I'm also truly thankful to Adrián Andrés Soto Suarez, PhD student in the Pontificia Universidad Catolica de Chile, for our weekly meetings and the continual help about technical requirements. Working with them has been a real pleasure.

Finally, I also want to thank Daniela Bahamondes Vidal from the *Pregrado* administration and Danisa Herrera Campos from the *Postgrado* administration, who helped me in the administrative procedures of my double-degree between the Pontificia Universidad Catolica de Chile and the Ecole Centrale. I wouldn't have managed to produce this work without their attention to help me in the process.

## CONTENTS

	Page.
ACKNOWLEDGEMENTS .....	3
CONTENTS .....	4
TABLES INDEX .....	1
FIGURES INDEX.....	2
RESUMEN.....	3
ABSTRACT .....	4
1. INTRODUCTION .....	5
2. NOTATIONS.....	7
2.1 RDF Graphs.....	7
2.2 SPARQL Syntax and Semantics .....	7
2.3 JSON documents and navigation instructions.....	9
2.4 Federation extension in SPARQL 1.1 .....	9
3. UPDATING THE FEDERATED QUERIES IN APACHE JENA.....	11
3.1 Antecedents .....	11
3.2 Methodology .....	13
3.3 Extending the JSON incorporation abilities.....	14
3.4 Implementation of the federated query update.....	19
4. EXPERIMENT AND OPTIMIZATION.....	23
4.1 Dynamic programming to reduce the API Calls .....	23
4.2 Streaming the results to optimize the processing times .....	29
4.3 Experimental results.....	31
5. CONCLUSION AND FUTURE WORK .....	36
BIBLIOGRAFY .....	39
APPENDICES.....	41

APPENDIX A: JAYWAY JSONPATH DOCUMENTATION .....	42
APPENDIX B-1 : SOURCE CODE – CROSS PRODUCT IMPLEMENTATION .	44
APPENDIX B-2 : SOURCE CODE – PARSER OF API SERVICE QUERY .....	46
APPENDIX B-3 : SOURCE CODE – PARSE TRIPLETS GROUPED BY BLOCK	49
APPENDIX B-4 : SOURCE CODE – API OPTIMIZATION .....	54
APPENDIX B-5 : SOURCE CODE – PIPELINE IMPLEMENTATION .....	61
APPENDIX C-1 : CONSTRUCTION METHOD – TESTING RDF DATABASE .	66
APPENDIX C-2 : SOURCE CODE - TESTING JSON API .....	67
APPENDIX D : PRACTICAL USE CASE – QUERY & RESULTS DETAILS .....	68

## TABLES INDEX

Page.

Table I-1: Parsing of the pre-incorporation SPARQL query into triples grouped by blocs. Each line corresponds to one bloc. ....	26
Table II-1: Results details for the comparison of the 4 execution methods. ....	69
Table II-2: Results mean values (obtained from details) for the comparison of the 4 execution methods. ....	69
Table II-3: Results details for the comparison of the classic and streaming methods while using a LIMIT optional bloc. ....	69
Table II-4: Results details for the comparison of the queries over a single API instead of both. ....	69

## FIGURES INDEX

	Page.
Figure 1.1: Description of the standard SPARQL query used in the project. ....	8
Figure 3.1: Cross-product between arrays extracted from a JSON document .....	17
Figure 3.2: Algorithm to incorporate the JSON objects' values into the results of the pre-incorporation SPARQL query. ....	19
Figure 4.1: Representation of a SPARQL query as a graph pattern .....	25
Figure 4.2: Example of application of the algorithms of optimization.....	28
Figure 4.3: Comparison of the classic execution chain and the pipeline one.....	29
Figure 4.4: Comparison of the processing times of the Twitter API and the Openweather one.....	32
Figure 4.5: Comparison of the processing times of the query execution methods....	33
Figure 4.6: Comparison of the processing times of the classic and streaming query execution methods with the LIMIT option.....	35

## RESUMEN

Si la publicación de datos semánticos en el formato RDF explotó en los años pasados (Bizer, Heath & Berners-Lee, 2009), la mayoría de los datos disponibles en el Web queda inaccesible a los servicios del Web Semántico (Junemann, Reutter, Soto & Vrgoc, 2016). Este proyecto propone lograr acercarse de la visión del W3C gracias a la conexión de las tecnologías y bases de datos del Web con el formato estandarizado RDF. Los datos disponibles a través de Web APIs en el formato JSON están más específicamente considerados por su alta presencia en el Web que implica numerosos casos de uso asociados.

Primero, se busca a través del proyecto extender las capacidades de incorporación de los documentos JSON en las respuestas a las consultas SPARQL, respeto a la implementación de Junemann, Reutter, Soto and Vrgoc (2016). Al fin de lograrlo se propone la integración de un módulo existente de navegación en el documento JSON en vez del uso de una implementación propia. Segundo, proponemos implementar la extensión como una actualización de las consultas federadas, accesibles por la palabra clave SERVICE según el estándar SPARQL 1.1, en vez del operador BIND propuesto por Junemann, Reutter, Soto and Vrgoc (2016). Finalmente, presentamos dos optimizaciones del módulo al fin que tenga una mejor usabilidad. Las características claves evaluadas son el número de llamadas a las APIs por las limitaciones que muchas Web APIs existentes tienen, y el tiempo de proceso de la consulta que suma el tiempo de proceso de la base de datos con el de la llamada a la(s) API(s).

Palabras Claves: Web Semántico, Consultas federadas, Web API, Incorporación de data, SPARQL 1.1



## ABSTRACT

Even with the explosion of semantic data over the past years through the publication of billions of data in the RDF format (Bizer, Heath & Berners-Lee, 2009), the W3C's vision is still far to be accomplished as most of data available on the Web remains out of reach to Semantic Web services (Junemann, Reutter, Soto & Vrgoc, 2016). That's why further advancements are proposed in this work to achieve and create Linked Data by making available existing technologies and databases for the RDF common format. The data exposed by Web APIs in a JSON format is more specifically addressed for its common use in the Web, so the numerous associated use cases.

First, the project includes new capacities to incorporate JSON into the SPARQL queries responses respecting to the implementation of Junemann, Reutter, Soto and Vrgoc (2016). One way to do it is to use an existing JSON management library instead of using a proper one. Second, we propose to integrate the module as an update for the federation extension of SPARQL 1.1, accessible by the SERVICE keyword, instead of extending the BIND operator as proposed by Junemann, Reutter, Soto and Vrgoc (2016). Finally, we present two optimizations to improve the module's usability. The key performance indicators which are evaluated are the number of API calls made by the query execution and the total processing time. First indicator is justified by the limitations imposed by numerous Web APIs. The second sums the DB processing time and the API processing time and impacts directly the usability of the implementation as huge amounts of data are susceptible to be queried by the Web Semantics users.

Keywords: Semantic Web, Federated Queries, Web API, Data Incorporation, SPARQL 1.1

## 1. INTRODUCTION

Semantic Web, also called Web 3.0, refers to W3C's vision of a machine-readable Web of data communicating cross multiple distinct sources<sup>1</sup>. Its purpose is to lead the Web to its full potential (Berners-Lee, Hendler & Lassila, 2001). It has been founded on three main standards. The Resource Description Framework (RDF)<sup>2</sup> enables publishing and connecting the data in the Web (e.g. *A is married to B* will link A to B by the “is married” relation) following the best practices referred by the term of “Linked Data”. The Web Ontology Language (OWL)<sup>3</sup> associates understandable meanings to the information (e.g. *A is married to B* means that *B is married to A*). The Service Protocol and RDF Query Language (SPARQL)<sup>4</sup> allows to retrieve those data as easily as SQL does for relational databases (e.g. *Select the name of A which is married to B* will look for the linked property “name” of the entities A linked to entities B by the relation “is married”).

In this work, we propose some advancements to achieve and create Linked Data<sup>5</sup> by making available current Web technologies to the RDF format. Indeed, even with the explosion of semantic data over the past years (Bizer, Heath & Berners-Lee, 2009), the W3C's vision is still far to be accomplished as most of data on the Web remains out of reach to Semantic Web services (Junemann, Reutter, Soto & Vrgoc, 2016). We address more specifically the data exposed by Web APIs in a JSON format. Its common use in the Web allows to develop numerous use cases. The final output is a second version of the SPARQL extension

---

<sup>1</sup> <https://www.w3.org/standards/semanticweb/>

<sup>2</sup> <https://www.w3.org/RDF/>

<sup>3</sup> <https://www.w3.org/OWL/>

<sup>4</sup> [https://www.w3.org/2009/sparql/wiki/Main\\_Page](https://www.w3.org/2009/sparql/wiki/Main_Page)

<sup>5</sup> <https://www.w3.org/standards/semanticweb/data>

developed by Junemann, Reutter, Soto and Vrgoc for the 15<sup>th</sup> International Semantic Web Conference in Kobe, Japan (2016).

In a first section, we explicit the notations which are going to be used. Then, the second section describes the core of this investigation. It consists in updating the Apache Jena implementation of the SPARQL federation extension to manage queries on JSON Web APIs. This extension has been included in SPARQL 1.1 to deal with the inherent distributivity of semantic data over the multiple SPARQL endpoints (Buil-Aranda, Arenas & Corcho, 2011). It is consequently a logical step to update it to address the distributivity and variety of the whole data present in the Web. The last section exposes the experiments and optimizations which have been produced to improve the usability of the component. Finally, this document is concluded by the identification of further works.

## 2. NOTATIONS

In this section, we explain some notations relative to the Web Semantics and the JSON language that are going to be used in the next sections.

### 2.1 RDF Graphs

An *RDF triple* is defined as the composition of a subject “s”, a predicate “p” and an object “o”. If we take back the relation “A is married to B” from the introduction, the triple would consequently be: (“s”: “A”, “p”: “is married to”, “o”: “B”). The subject and the object can be IRIs (I), literals (L) or blank nodes (N), while the predicate is necessarily an IRI. The set of *RDF terms*  $T$  is IULUB. An *RDF graph* is defined as a finite set of RDF triples. Following the approximation done by Junemann, Reutter, Soto and Vrgoc (2016), it is assumed here that an *RDF database* consists of a single RDF graph.

### 2.2 SPARQL Syntax and Semantics

It is assumed that the reader is familiar with the syntax and semantics of SPARQL 1.1 query language. The structure of SPARQL patterns, and its semantics as used in this work, are described to help the reader better understand it. The focus is on the SELECT queries, which are built over terms  $T$  and an infinite set  $V = \{?x, ?y, \dots\}$  of variables, disjoint from  $T$ . The queries are formed by blocks which can be inserted the ones into the others. In this work, it is assumed that each SPARQL query is a concatenation of a set of blocks, where each block is either a SPARQL structural block (PREFIX, SELECT, WHERE, OPTIONS) or a subset of restrictions.

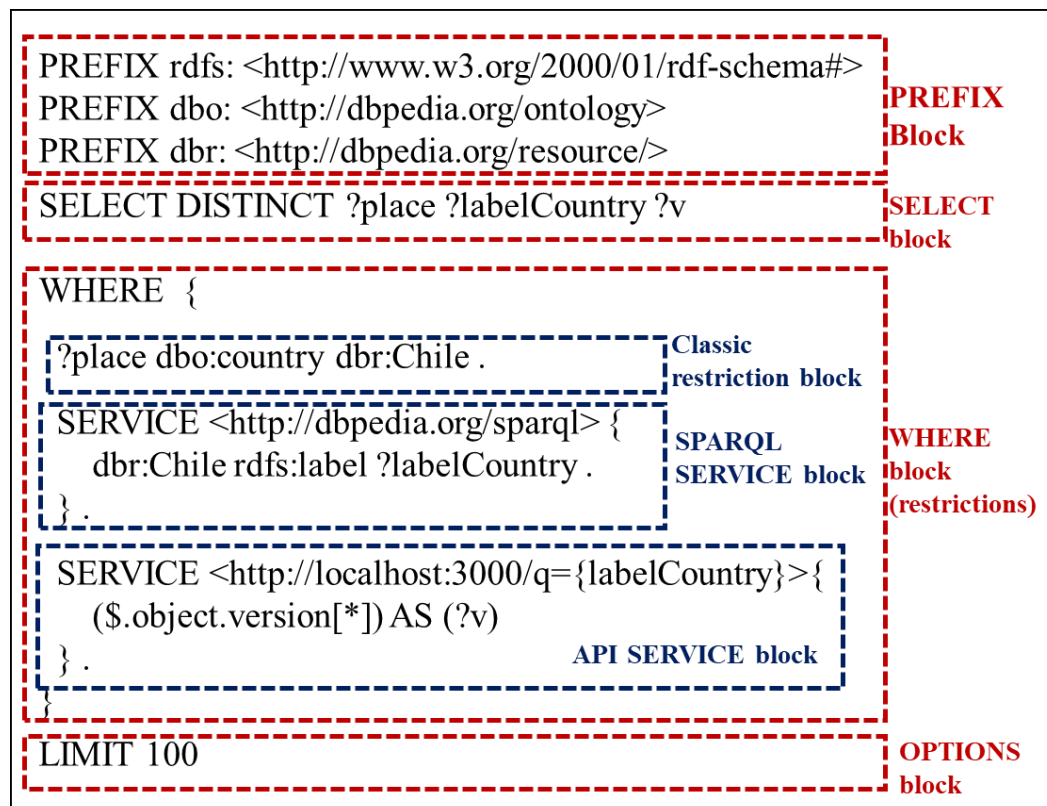


Figure 1.1: Description of the standard SPARQL query used in the project.

As shown in the previous figure, the WHERE block is composed by sub-blocks. This work introduces the API SERVICE block. It is an update of the SERVICE feature defined by the SPARQL 1.1 standard to query data from remote SPARQL endpoints. The API SERVICE block allows to query JSON Web APIs instead of SPARQL endpoints. The WHERE restrictive block concatenates consequently:

- classic restrictive blocks composed by triples of  $TU\{?x1, \dots ?xn\}$  which are separated the ones from the others by the dot or the dot-coma ;
- SPARQL SERVICE and API SERVICE blocks ;

- VALUES block following the pattern: VALUES (?x1 ?x2 ... ?xn){(l1)(l2) ... (lm)}  
with ?x1 ?x2 ... ?xn as a list of variables and l1 l2 ... lm as lists of elements from  $T \cup \{UNDEF\}$ , each of size n.

VALUES pattern allows to restrict the variables of a query to values stored into mappings. A *mapping* would be for example: (“x”: “value1”, “y”: “value2”, “z”: “value3”). We define the concept of *Mapping Set* as a list of mappings and the list of the variables of the mappings. In the continuation, the notion of *section* of a SPARQL query will also refer to the notion of block.

### 2.3 JSON documents and navigation instructions

The JSON documents are structured with *keys* which refer to *values* such as {“key”: “value”}. To navigate in a document, we start from its *root* which is the level 0 of keys and return the whole document. Then, we navigate between the different levels of keys and arrays elements by calling them. It is also possible to filter values through numerical filters or regular expressions.

### 2.4 Federation extension in SPARQL 1.1

For the inherent distribution of data over SPARQL endpoints, the problematic of federated queries disposes already from a consistent background of investigations produced by the Semantic Web Research community (e.g. Görlitz & Staab, 2011; Rakhmawati, Umbrich, Karnstedt, Hasnain & Hausenblas, 2013). It has been recently integrated to the SPARQL 1.1 standard through the keyword SERVICE which allows to query a specified SPARQL

endpoint while imposing constraints to the linked variables. However, few works focused on the federation extension and its optimization (Buil-Aranda, Arenas & Corcho, 2011).

The Apache Jena Framework includes also the SERVICE feature through the Jena-arq library to execute query to make a SPARQL protocol to another SPARQL endpoint. The documentation specifies however that it is “not a general solution to the issues in distributed query evaluation” (The Apache Jena Manual, 2017). Indeed, the use of the feature affects considerably the speed of execution.

Considering the similarities between, in one side, the format and functions (URI call, authentication, timeouts, endpoint configuration) associated to the existing SERVICE feature, and on the other side, the needed functionalities and proposed grammar to incorporate data from Web APIs, we identified an interest to revisit the implementation of the BIND\_API extension as an extension of the SERVICE feature, allowing to execute calls to Web APIs in addition to the remote SPARQL endpoints. Indeed, the proposition also makes sense at a problematical level, as the project can be considered as a trial to federate RDF data and JSON data from Web APIs while executing the SPARQL queries.

### 3. UPDATING THE FEDERATED QUERIES IN APACHE JENA

In this section, we first describe the antecedents of this work with a focus on the paper “Incorporating API data into SPARQL query answers” presented by Junemann, Reutter, Soto and Vrgoc during the 15<sup>th</sup> International Semantic Web Conference in Kobe, Japan (2016). Then, we detail the methodology which has structured this investigation. Finally, we describe separately the improvements of JSON incorporation capabilities and the implementation of the proposed SPARQL federation query update.

#### 3.1 Antecedents

In their paper, Junemann, Reutter, Soto and Vrgoc (2016) identified an interest in incorporating Web API data into RDF data while querying it. Indeed, the principal limit to W3C’s vision remains the minority of data accessible by Web Semantic services comparing to the whole amount of data generating by the Web. Addressing in a first time JSON Web API, the authors proposed a new approach to include those inaccessible data by binding it to RDF data while processing the SPARQL queries.

There has been a lot of work to enable SPARQL as an API through usable endpoints<sup>6</sup> and some investigations to convert the data from JSON to RDF format (e.g. Kobayashi et al., 2011). However, Battle and Benson (2008) are the only ones to have proposed a similar way to access API data, which do not depend on a codification into RDF but on an on-call RDF wrapper. The strength of the Junemann, Reutter, Soto (2016) and

---

<sup>6</sup> <https://open-data.europa.eu/en/linked-data>



Vrgoc extension is to be less restrictive on the return format of API calls, and so, compatible with modern JSON APIs as the Apache Jena framework.

Although their proposition doesn't allow to achieve completely the vision of the Web Semantic as described by Berners-Lee, Hendler and Lassila (2001), as the no-RDF data will remain poorly structured and meaningful in the sense of the Web Semantic, it is an important step for the field to develop concrete and complex use cases based on the RDF and SPARQL standards. It could finally represent an issue to the complexity of producing good RDF data which limits the expansion of the Web of data.

Concretely, the researchers proposed to extend the BIND operator with the new grammar: `BIND_API <URL> { (JSON_PATHS) AS (ALIASES) }`. The implemented strategy can be resumed by a recursive decomposition of the query in three sub-queries: what comes before the BIND\_API block, the BIND\_API block, what comes after. The first sub-query corresponds to a classic SPARQL query which should be run before the API call to retrieve the JSON data. The second one produces an on-call conversion of the JSON data requested from the API to a Mapping Set. The Mapping Set is then readable in a VALUES block by the final SPARQL query.

This work attempts to develop a second version of the proposed extension to allow a better incorporation of JSON APIs data in the results of the SPARQL grammar, and so, increase the number of relative use cases. The only part of the implementation

which has been used but kept unchanged is the management of authentication strategies to access the distinct APIs.

### **3.2 Methodology**

The whole project was executed under an agile methodology of computation. It started with a step of capacitation and code appropriation and followed with three “Sprints”. The first Sprint focused on the improvement of JSON incorporation capacities (see section 3.3), the second Sprint on the test of the SERVICE feature with Apache Jena Framework, the design and the implementation of the proposed update (see section 3.4), and the last Sprint on the experiment and optimization of the implementation (see section 4).

We structured the developments with a Test & Learn approach. The cyclical execution of design, implementation and testing tasks allowed us to develop robust functions facing the multiple use cases in a short time. Under that perspective, we implemented two testing tools: an RDF local database containing triples extracted from the DBPedia endpoint, and a local JSON API programmed in NodeJS. The construction query of the RDF database tool and the source code of the JSON API tool are respectively exposed in the appendix C-1 and the appendix C-2. Those tools allowed us to deploy easily the scalable tests of our strategy.

Our battery of test on the implementation of the SPARQL federation query update was composed by queries including the next items:

- A simple API: <http://localhost:3000/api>
- A simple SPARQL remote endpoint: <http://dbpedia.org/sparql>
- Multiple APIs
- Combinations of API/remote endpoint:

API – sparql

Sparql – API

API – sparql – API

- Load test with local API: 9216 mappings

Total time processing: 591.370505972 API: 3.602020148 DB: 587.768485824

We also tested all the potential cases of the product cross while calling an API (described in the next section), and different paths including sequences of character susceptible to trouble the algorithm.

Finally, we made a consistent effort to produce code following the best practices of programming that are:

- the documentation of the functions through clear standardized commentaries ;
- the traceability of the code using git and github ;
- the modularization of the functionalities.

### **3.3 Extending the JSON incorporation abilities**

#### Integration of a JSON navigation library

The first version of the extension developed by Junemann, Reutter, Soto and Vrgoc (2016) included its proper navigation method inside the JSON document. In a first

time, this method has been extended in the current work to be able to return complete arrays from the JSON document thanks to the key `[*]`, and not only one indexed element at a time. However, if the proper method had for advantage a great flexibility to prove the concept, we needed something more powerful in this second step in terms of scalability and robustness. That's why we chose to integrate an existing module.

Until March 2017, no W3C standards had been defined to navigate or query JSON data. However, the W3C referred to three main query languages under development which are JSONpath, Jaql and JSONiq<sup>7</sup>. JSON Path is defined as a Xpath-like tool to navigate in the JSON document with no need to convert to or from XML. Jaql has been initially developed by IBM and is oriented to deal with Big Data problematic. Finally, JSONiq is based on the XQuery standard to query JSON data instead of XML data. All JSONiq functionalities were included in the XQuery 3.1 version in March 2017<sup>8</sup>, so the W3C query language standard handles now JSON data in addition to the XML ones. The standard XPath was also released in March 2017 to include JSON navigation in its version 3.1<sup>9</sup>. These recent advancements for the JSON manipulation standardization have great implications for the project. The use of XPath or XQuery would depend on the levels of liberty and complexity that the project is looking for the manipulation of data from APIs. Both would allow to easily incorporate XML data in a recent future.

---

<sup>7</sup> <https://www.w3.org/standards/webarch/metaformats>

<sup>8</sup> <https://www.w3.org/TR/xquery-31/>

<sup>9</sup> <https://www.w3.org/TR/xpath-31/>

Nevertheless, the last releases of those standards still do not offer simple and trustable implementations for Maven Java Projects which could be integrated to the Apache Jena framework. Indeed, the universal Java Xpath engine called Jaxen was last released on December 2013 and the XQJ API was last released on October 2016. On the contrary, the JSONpath disposes from the Java DSL Jayway Jsonpath<sup>10</sup> available under an Apache License 2.0 which can be easily integrate to Maven projects, offers all the navigation functionalities needed at this step of the project, is accompanied by a clear documentation and a good community responsiveness. for the chosen implementation of the project over the Apache Jena framework, we chose to use this DSL at this step of project, despite its unstandardized characteristic.

The appendix A provides an extract of the Jayway Jsonpath documentation which exposes how to integrate it in the pom.xml of a Maven Project, the operators to navigate in the JSON document, the available functions to manipulate the numerical values, and the operators to filter the values as the regex one, for example. The DSL also offers the opportunity to run a Filter API, to develop its one predicate, to choose to return paths instead of values, to tweak the configuration. It is shipped with three different JSON Provider SPI and allows API consumers to configure path caching. At this step, we only occupied the basic navigation operators to return values associated to simple key or arrays.

---

<sup>10</sup> <https://github.com/json-path/JsonPath>

### Implementation of the cross-product

The code source is available in the appendix B. The cross-product implementation refers to the incorporation of the data from two JSON-extracted arrays into the results of the SPARQL query. It takes two JSON arrays as inputs and produces a Mappings set as output. The next figure illustrates the attended result.

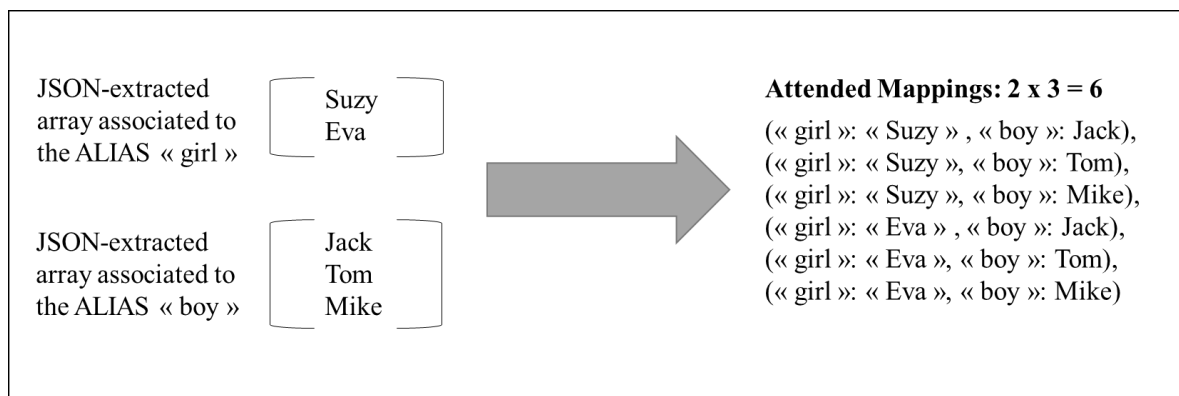


Figure 3.1: Cross-product between arrays extracted from a JSON document

The final Mappings Set should include the RDF data which had been answered by the pre-incorporation query. We defined consequently a local array of mappings which is completed and merged into the final mapping at each iteration over the results of the pre-incorporation sub-query.

Four cases need to be differentiated to complete the mapping array. First, the object returned by the path into the JSON document can be a simple JSON value, or a JSON array. The object's type has implementational consequences that need to be considered. Independently of the type, the behavior of the algorithm also changes if we are incorporating values of the JSON document for the first alias or the next ones.

Indeed, if we consider that there are  $n$  values in the returned object associated to the alias. For the first alias,  $n$  new mappings should be added in the local array of mappings, while for the next ones,  $m \times (n-1)$  mappings should be added to the array ( $m$  the number of mappings already in the array).

The next figure illustrates the algorithm and the mentioned considerations. The relative query of the illustration could be written as:

```
SELECT ?value ?girl ?boy WHERE {
    ?value <http://example.com.isPartOf> <http://example.com.valuesSet> .
    BIND_API <http://localhost:3000/q={value}> ((["girls"][*], ["boys"][*]) AS
    (?girl, ?boy))
}
```

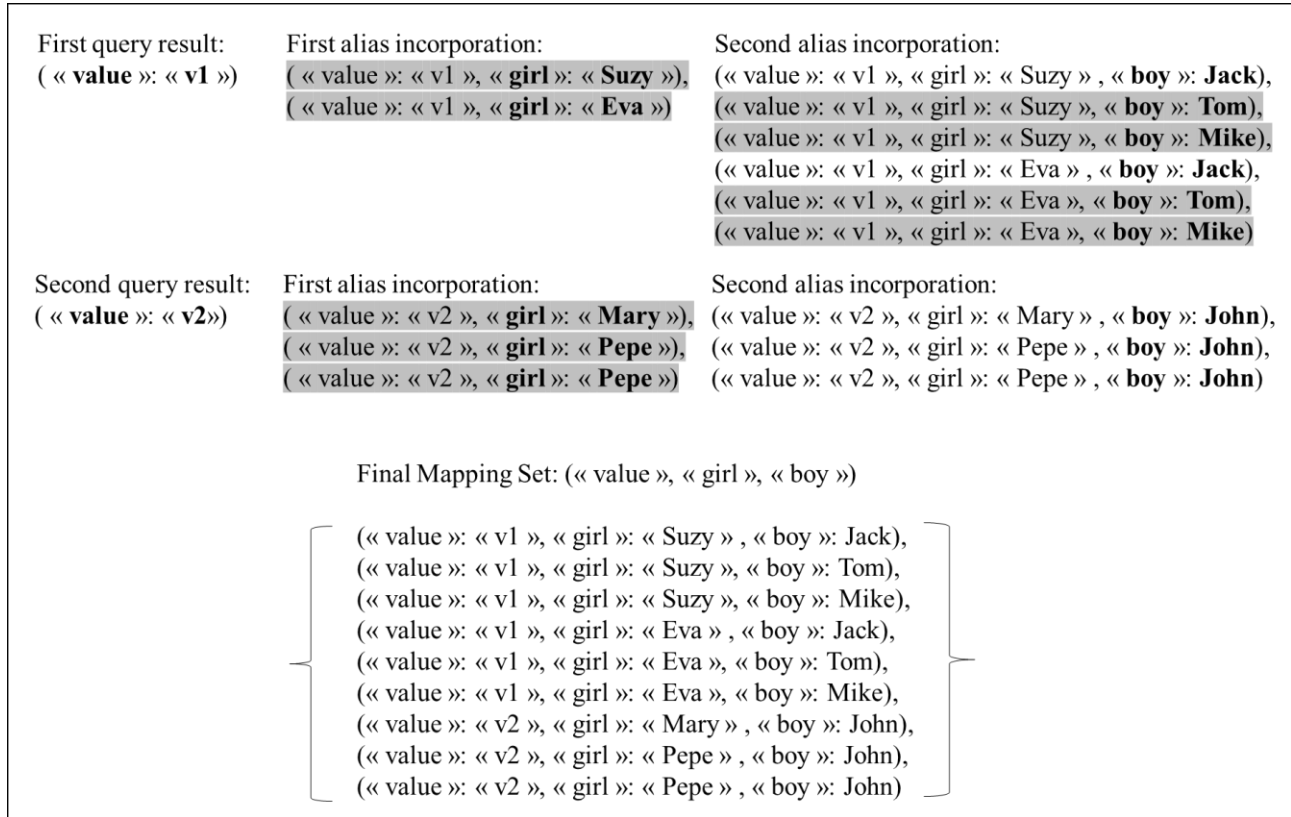


Figure 3.2: Algorithm to incorporate the JSON objects' values into the results of the pre-incorporation SPARQL query.

Note: The strong values represent the new values added to the mapping at the given step.

The mappings in grey represent the new mappings added to the local mapping array.

### 3.4 Implementation of the federated query update

The main part of the updated implementation consists in a parser capable to identify a SERVICE bloc calling a Web API instead of a SPARQL endpoint in the queries. The source code of the parser is exposed in the appendix B-2. We updated the grammar proposed by Junemann, Reutter, Soto and Vrgoc (2016) with the keyword SERVICE



and the use of the Java DSL Jayway Jsonpath. Then, a select query including one or more API SERVICE follows the format:

```
SELECT variables WHERE {
    first constraints and options
    SERVICE <url> { (paths) AS (aliases) }
    last constraints and options
}
```

Another grammatical rule is that the paths start with the root operator “\$.” as defined by the Java DSL Jayway Jsonpath and are separated by comas. The aliases start with an interrogation point, contain then only alphanumerical symbols and are separated by comas.

All this grammar allowed us to define a regex capable to match only the blocs following those rules, so the program executes the algorithm of the extension instead of the classic SPARQL protocol. The algorithm proposed by Junemann, Reutter, Soto and Vrgoc (2016) which is described in section 3.1 separates recursively the query into three sub-queries. So, the parser needs to identify distinct sections of the query to execute correctly the algorithm. The SELECT section of the query contains the selected variables to return at the end. The FIRST section contains the constraints to apply on the selected and unselected variables before calling the API. This section can be one unique block or a combination of several blocks including constraints to apply to the local data and constraints to apply to the data from a remote SPARQL endpoint accessed thanks to the feature SERVICE from SPARQL 1.1. The URL, PATH and

ALIAS sections contain together all the information to call the Web API through the URL. They allow to look for the data to return from the JSON document and associate a variable name to the extracted values while mapping them. Finally, the LAST section is submitted to the recurrence, so the algorithm looks for another API SERVICE bloc which could come later in the query.

If the parsing of the URL and ALIASES is quite easy for their constrained format. The parsing of the PATH is more complicated as it can include a regex allowing to access the data of the JSON document. So it can include all characters, sequenced in ways which could conduce to unwanted matches. That's why we chose not to run one unique regex matcher but to use several splits on the more constrained parts of the query, until we had only the paths left. On their side, the sections PATH and ALIAS are parsed into arrays containing the several paths and aliases so the precedingly mentioned method to map the JSON objects into a Mapping Set takes them directly as parameters.

The separation of the query into sub-queries is allowed using the VALUES operator of the SPARQL grammar. Indeed, the results of the intermediate queries are stocked into a Mapping Set which is serialized and transmitted to the next query thanks to this operator. It is important to notify that the Mapping Set needs to contain all the values of the variables on which the next queries depend. It means that the used but unselected variables should be selected in these queries.

The final query to be run don't include any API SERVICE bloc and takes the format:

```
SELECT variables WHERE {  
    VALUES (used variables) { (mappings) }  
    last constraints and options  
}
```

We also added in this updating the management of prefixes and improved the behaviour of the algorithm when escape characters are used to format the query, could it be in a good or a wrong way.

## 4. EXPERIMENT AND OPTIMIZATION

The query processing is evaluated through two principal key performance indicators. The first one corresponds to the total time processing of the query, which is the sum of the API and Database processing times. As users are susceptible to query huge amount of data, the indicator is essential for the usability of the extension. The second one is the number of API Calls. Web APIs limit generally the number of requests that a user can make. It is consequently essential to avoid making useless requests over those APIs. In addition, the API time processing has been identified as a limiting factor which needs to be optimized (Junemann, Reutter, Soto & Vrgoc, 2016). This section exposes some obtained results and describes two optimizations proposed to improve the performance of the updating according to those indicators.

### 4.1 Dynamic programming to reduce the API Calls

We propose to use an optimization strategy called “Dynamic programming”. It ensures in traditional relational databases to find the optimal query execution plan for any given query. In our case, the optimal query execution plan corresponds to the one which minimize the number of API Calls.

A first basic approach is to change the pre-incorporation SPARQL query to identify and select only the variables which are inserted in the URL of the API as parameters. We avoid then to have useless results over which the algorithm of incorporation would iterate and call the API. However, this strategy imposes to copy the constraints of the pre-incorporation query and to paste them in the post-incorporation one. Otherwise, information are lost and

the results of the query are changed. Because of the copy/paste, this approach has the inconvenient to increase the query processing times as the first query is duplicated. Considering that the time processing is also defined as a key performance indicator, and that the SERVICE feature of SPARQL 1.1 isn't optimized in Apache Jena so its duplication in the query can potentially alter seriously the indicator, an optimization of this method would be more than welcome.

One challenge here is so to identify which constraint of the pre-incorporation SPARQL query should be processed before the incorporation, which ones should be processed after the incorporation and which ones need to be processed in both. To do so, we designed and implemented two graph algorithms that we run on the pre-incorporation SPARQL query. The next figure illustrates how the constraints of a SPARQL query can be modeled as a graph pattern. Before explaining the two algorithms of optimization, we propose some definitions about the variables and triples of a given graph pattern  $P$ .  $V(P)$  is the set of variables of  $P$ .

**Strongly bounded variables:** We propose the notation  $(?x = ?y)_P$  for  $?x$  strongly bounded to  $?y$  in  $P$ . Considering  $?x, ?y \in V(P)$ ,  $(?x = ?y)_P$  is defined as:

$$\exists p \in I \cup V, (?x, p, ?y) \in P$$

**Bounded variables:** We propose the notation  $(?x - ?y)_P$  for  $?x$  bounded to  $?y$  in  $P$ . Considering  $?x, ?y \in V(P)$ ,  $(?x - ?y)_P$  is defined as:

$$\exists ?z_n \in V(P)^{n+1}, (?x - ?z_0)_P, \forall i \in [0, n - 1], (?z_i = ?z_{i+1})_P, (?y = ?z_n)_P$$

We assume that  $n$  can be null and we retrieve so the strong boundedness in  $P$ .

**Independent variables:** We propose the notation  $(?x|?y)_P$  for  $?x$  independent to  $?y$  in  $P$ .

Considering  $?x, ?y \in V(P)$ ,  $(?x|?y)_P$  is defined as:  $!(?x - ?y)_P$

**Unbounding triple:**  $T$  is an unbounding triple for  $?x$  in  $P \leftrightarrow V(P) \cap T = \{?x\}$

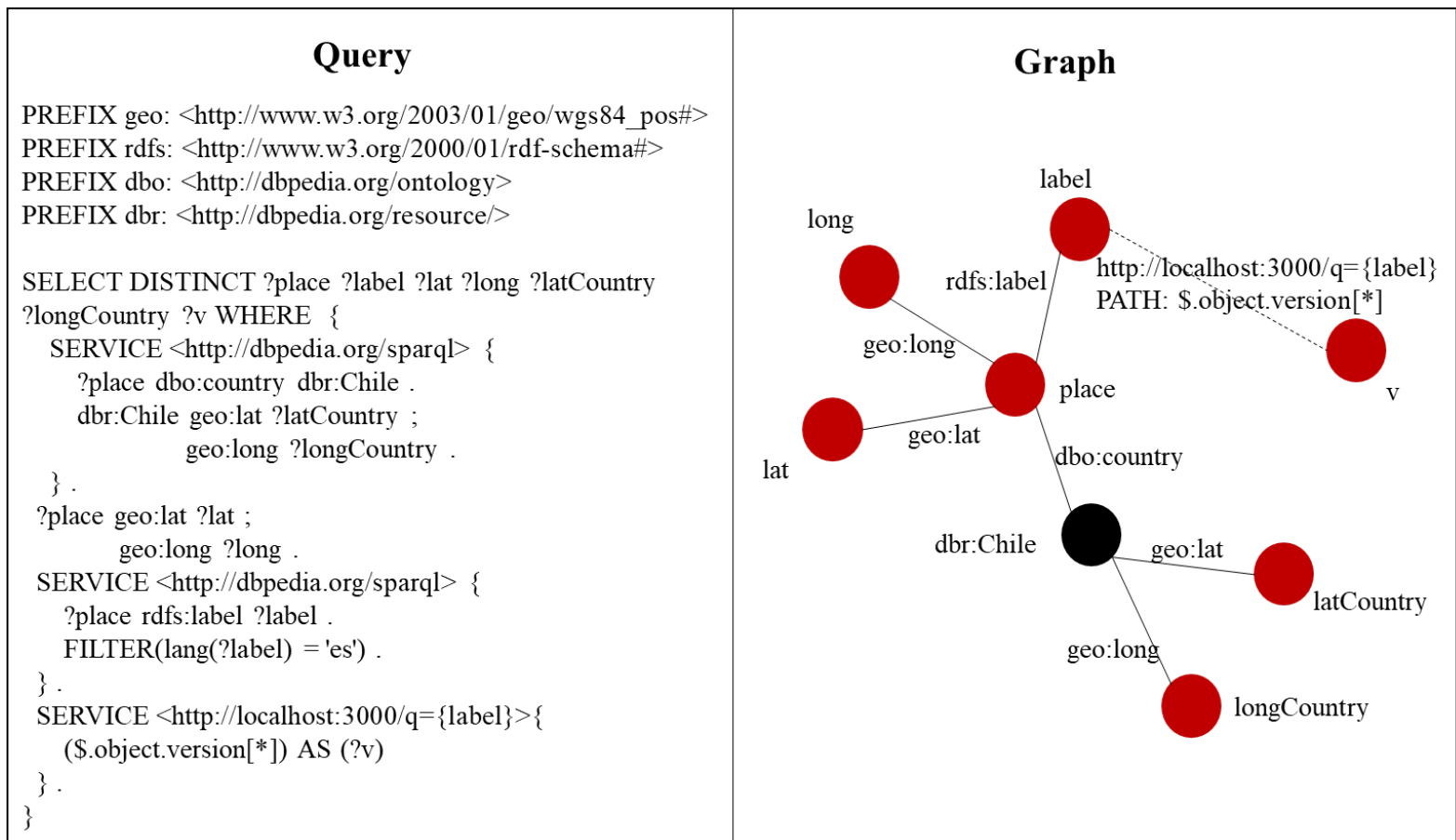


Figure 4.1: Representation of a SPARQL query as a graph pattern

Note: The circles represent nodes and the continue lines represent the links of the graph pattern  $P$ . The pointed line represents a link created using an API SERVICE bloc. A symbol (circle or line) in red represents a variable of  $V(P)$  and a symbol in black represents an element of  $T$ .

We implemented a new class to enable the transformation of the constraints from a pre-incorporation SPARQL query to a graph pattern. The source code of the class and its relative methods is exposed in the appendix B-3. The query is parsed into triples which are grouped by blocks. A block can be a basic SPARQL block or a SPARQL 1.1 SERVICE block which keeps the information of the remote endpoint's URI. Options such as a FILTER operator are also kept at the level of the block.

Table I-1: Parsing of the pre-incorporation SPARQL query into triples  
grouped by blocs. Each line corresponds to one bloc.

Type	URI	Triplets			Options
SERVICE	http://dbpedia.org/sparql	?place	dbp:country	<http://dbpedia.org/resource/Chile>	
		<http://dbpedia.org/resource/Chile>	geo:lat	?latCountry	
		<http://dbpedia.org/resource/Chile>	geo:long	?longCountry	
BASIC		?place	geo:lat	?lat	
		?place	geo:long	?long	
SERVICE	http://dbpedia.org/sparql	?place	rdfs:label	?label	FILTER(lang(?label) = 'es')

Both algorithms of optimization start from the basic case which duplicates all the triples of the pre-incorporation SPARQL query into the post-incorporation query, grouped by blocs.

Then, the first one eliminates the triples which are not needed from the pre-incorporation query and the second one does the same with the post-incorporation query.

Unwanted triples in the pre-incorporation query correspond to all triples that contain one or more variables independent from the inserted variables. Indeed, those triples have no influence on the result of the query selecting the inserted variables and can consequently be included only after the incorporation. The algorithm recognizes recursively the variables which are bounded to the inserted variables. At each iteration, it recognizes the variables which are strongly bounded to the previously recognized variables and eliminates the relative bounding triples from the triples to browse. Once there is no more triple to browse or once there is no new recognized variable, the iteration stops and all triples which had not been eliminated are eliminated from the pre-incorporation query. If a block has no more triple, it is eliminated. Else, the bloc is kept with all its relative information less the eliminated triples. If we take back the example of the figure 5.1, the variables ?longCountry and ?latCountry are independent from the variable ?label. The two triples inside of the first SERVICE bloc which include those variables should consequently be eliminated from the pre-incorporation query, but not the entire bloc as it still contains one triple.

Unwanted triples for the duplication into the post-incorporation query are the unbounding triples which contain an inserted variable. Indeed, as those triples do not implicate another variable, all their restricting information is already transferred to the post-incorporation query through the operator VALUES. The second algorithm looks consequently for the triples which contain the inserted variables. If they don't contain another variable, they are



eliminated from the triples to add to the last query. As explained for the previous algorithm, a block is eliminated only if it has no more triple.

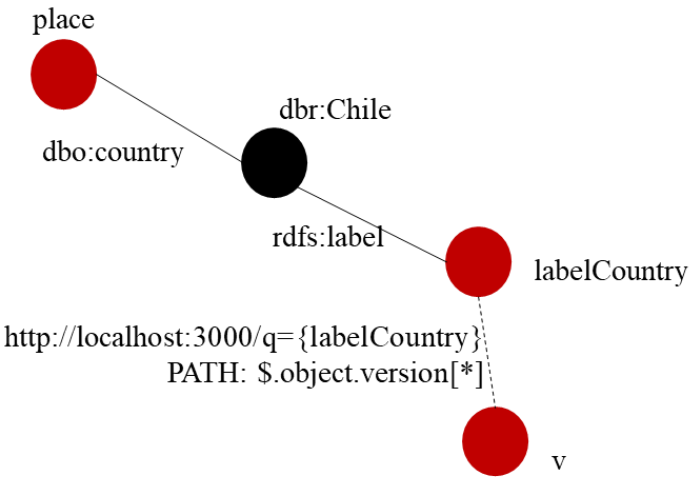
<p style="text-align: center;"><b>Query</b></p> <pre> PREFIX rdfs: &lt;http://www.w3.org/2000/01/rdf-schema#&gt; PREFIX dbo: &lt;http://dbpedia.org/ontology&gt; PREFIX dbr: &lt;http://dbpedia.org/resource/&gt;  SELECT DISTINCT ?place ?labelCountry ?v WHERE {   ?place dbo:country dbr:Chile .   SERVICE &lt;http://dbpedia.org/sparql&gt; {     dbr:Chile rdfs:label ?labelCountry .   } .   SERVICE &lt;http://localhost:3000/q={labelCountry}&gt;{     (\$.object.version[*]) AS (?v)   } . }</pre>	<p style="text-align: center;"><b>Graph</b></p> 
<p style="text-align: center;"><b>Pre-incorporation query</b></p> <pre> PREFIX rdfs: &lt;http://www.w3.org/2000/01/rdf-schema#&gt; PREFIX dbo: &lt;http://dbpedia.org/ontology&gt; PREFIX dbr: &lt;http://dbpedia.org/resource/&gt;  SELECT DISTINCT ?labelCountry WHERE {   SERVICE &lt;http://dbpedia.org/sparql&gt; {     dbr:Chile rdfs:label ?labelCountry .   } . }</pre>	<p style="text-align: center;"><b>Post-incorporation query</b></p> <pre> PREFIX rdfs: &lt;http://www.w3.org/2000/01/rdf-schema#&gt; PREFIX dbo: &lt;http://dbpedia.org/ontology&gt; PREFIX dbr: &lt;http://dbpedia.org/resource/&gt;  SELECT DISTINCT ?place ?labelCountry ?v WHERE {   VALUES (?labelCountry ?v) {("Chile","v1"), ("Chile","v2")}   ?place dbo:country dbr:Chile . }</pre>

Figure 4.2: Example of application of the algorithms of optimization

This global method also identifies the inserted variables before applying the described algorithms. It is exposed in the appendix B-4. All optimization methods relative to the API calls have been grouped in a separate ApiOptimizer java class. It also includes the cache method which had been implemented by Junemann, Reutter, Soto and Vrgoc (2016) and has been tested and maintained in this version.

## 4.2 Streaming the results to optimize the processing times

The first version implemented the execution of the SPARQL query as a chain of sub-queries and calls to the mapping method which use the results of the precedent chain link as parameters for the next one. This choice can have serious implication for the time processing performance indicator if large intermediate results are produced or some data sources (Web APIs or SPARQL endpoint) have bad response times. That's why we propose in this section a new method of execution which allows to stream the results of the complete query. Concretely, we execute the first sub-query of the chain and, instead of waiting for all its results to be generated before calling the next method, this one is immediately executed taking as parameter only one result. The process is recursive so the whole chain execution is completed over one result before starting again with the next result. The source code of the function is exposed in the appendix B-5.

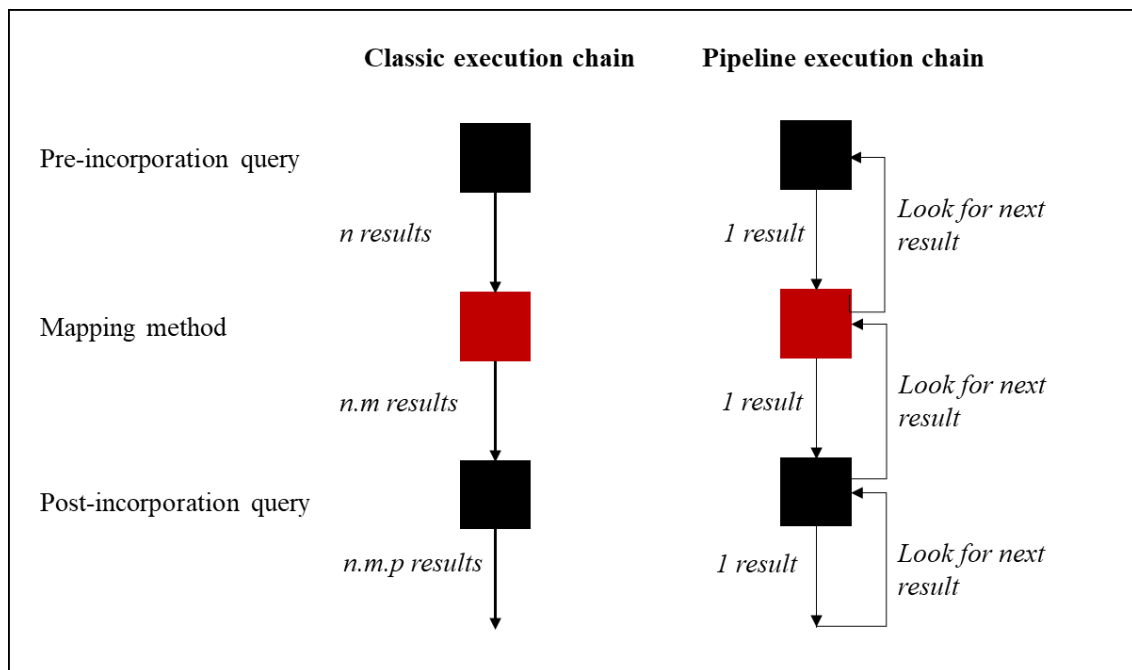


Figure 4.3: Comparison of the classic execution chain and the pipeline one.

The class `ResultSet` which is the format in which Apache Jena returns the results of a SPARQL query doesn't allow to merge two sets in once. That's why the pipeline execution method returns a `MappingSet` which has been completed at each iteration. For the consistence of the results with the ones obtained by the classic method, we convert at the end the `MappingSet` into a `ResultSet` to be print. To do so, we execute a final SPARQL query using the operator `VALUES` which contains all the mappings.

The streaming (or pipeline) method is particularly interesting for the query using the operator `LIMIT` as it allows to stop the recursive algorithm once the number of results is reached. On the contrary, the classic execution method would process all the results in a first time, and applies the `LIMIT` in a second one. That's why even if we implemented both execution methods and chose to let the control to the final user on which method he or she wants to use, we also took the decision to force the choice of the "pipeline" method when a `LIMIT` operator is used on the global query. Indeed, the time processing are significantly reduced doing so as exposed in the next section.

For the case of a `SELECT DISTINCT` query using the `LIMIT` option, we implemented a new method in the class `MappingSet` to be able to add only the distinct mappings from one `MappingSet` to another one while executing the recursive function. So, when we convert the `MappingSet` into the `ResultSet` with

the ultimate SPARQL query, all the mappings serialized as a VALUES block are already distinct and we get correctly the amount of results equals to the limit.

### 4.3 Experimental results

This section analyses the results of the distinct methods applied to a practical use case. The results details and the executed query are exposed in the appendix D. The query was constructed over the Twitter API and the Openweather API. The first one needs an authentication to be accessed while the second one only needs a valid parameter called “appId” in the URL. We propose a query which returns the number of retweets of the 15 most recent tweets, and the current weather, both relative to the 52 cities of Chile which are stored in yago with the type “Capital108518505”. We also use the remote SPARQL endpoint of DBpedia to access the labels of the cities as they are not kept in our local database (see appendix C-1).

To better understand the behaviour and results to attempt for both APIs, we ran in a first time the query keeping only one of the two API SERVICE blocs. The next figure allows to observe that a call to the Twitter API consumes more time than the one to the Openweather API, as it was predictable for their respective access strategy. Both queries count 52 API calls, one for each city. The query with the Twitter API call returns 633 results, instead of the  $52 \times 15 = 780$  attended results. This is caused by an error of connection to the API for some cities which returns one undefined number of retweets instead of the 15 mappings. The query with the Openweather API responds

52 results: one weather description is assigned to each city, although it could be more as the API returns an array of descriptions.

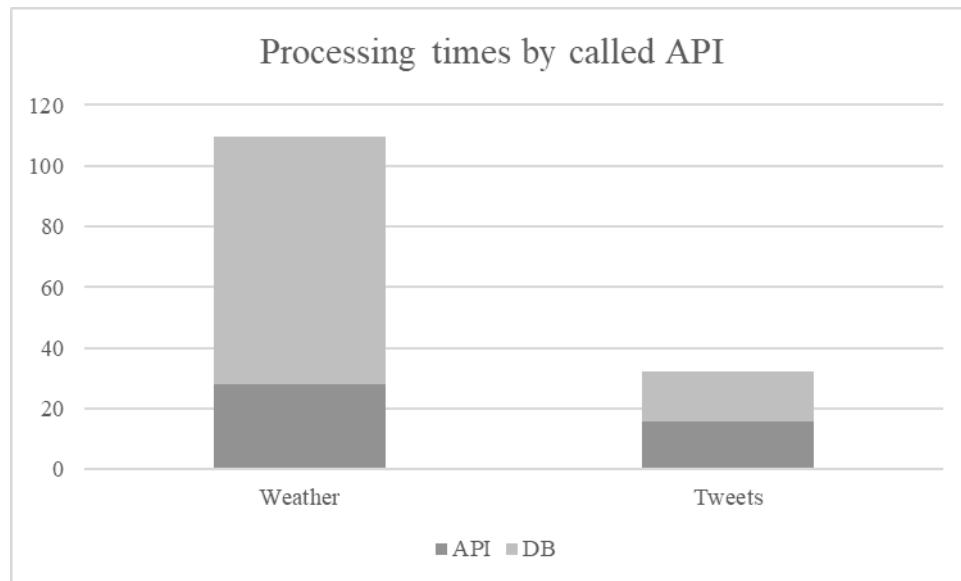


Figure 4.4: Comparison of the processing times of the Twitter API and the Openweather one.

Then, we ran three times a battery of 4 tests to compare the key performance indicators of each execution method. The classic method refers to the execution without optimization, the min API method refers to the activation of the dynamic programming over the query plan to minimize the number of API calls, the pipeline method refers to the streaming execution, and the last method combines the activation of both optimization algorithms.

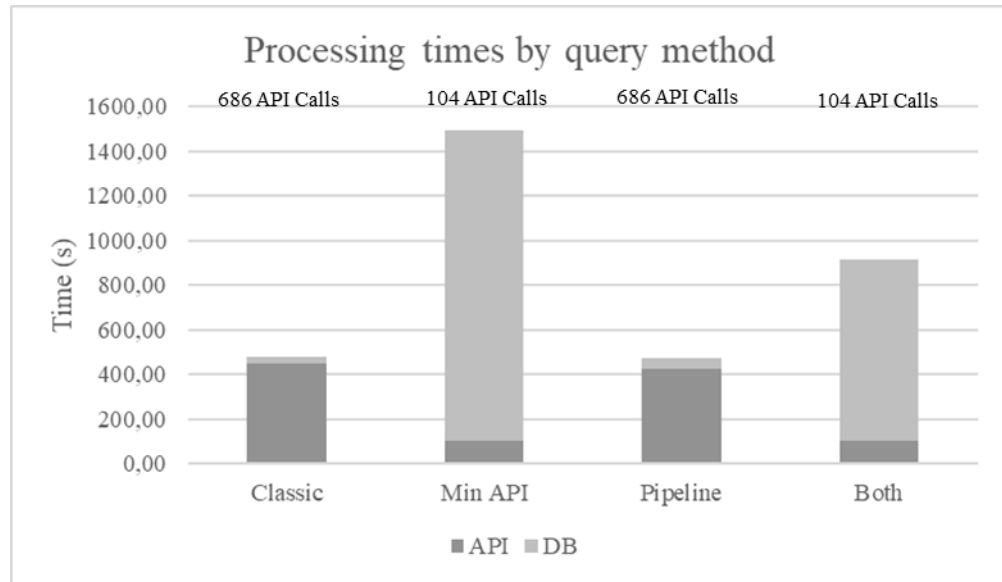


Figure 4.5: Comparison of the processing times of the query execution methods.

We can observe in the results of the previous figure that the optimization to minimize the number of API calls reduces drastically the number of calls in comparison with the classic method. We have indeed 104 calls, that means 1 for each city and for each API, while the classic method multiplies the API calls for the incorporation of twitter data which generates new mappings before the call to the Openweather API. A significant reduction of the API processing time is associated to the reduction of API calls. However, the increase in the DB processing time is much bigger and makes the total processing time more than 3 times higher than the classic one. That can be explained by the duplication of restrictions in pre-incorporation and post-incorporation SPARQL queries. The method generates huge VALUES blocks which impact the DB processing time.

The pipeline execution method gets values approximatively equal to the ones of the classic execution method for the key performance indicators. However, we observe that combining both the minimization of API calls and the streaming generates better performance than using just the minimization. Indeed, the number of API calls remains the same and there is a significative reduction of the DB processing time. This can be explained by the fact that the streaming method allows to manage smaller VALUES blocks as one value after the other is processed. That would mean that SPARQL queries are faster processed with numerous simple queries than with a complex one.

Finally, we made two tests with the same query to compare the classic method and the pipeline method while using the LIMIT option (putting to 100 results). The result justifies the choice to force the use of the pipeline method in this case. The classic method has the same processing times as without the LIMIT option because it executes the complete query and applies the limit only once all the results have been produced. In comparison, the streaming method allows to inform the system when the limit is reached and to short it so. The time processing is reduced and the number of API calls as well. We chose to apply a limit equals to 100 results so the difference between the two methods is obvious. It could however be even bigger while querying huge amounts of data.

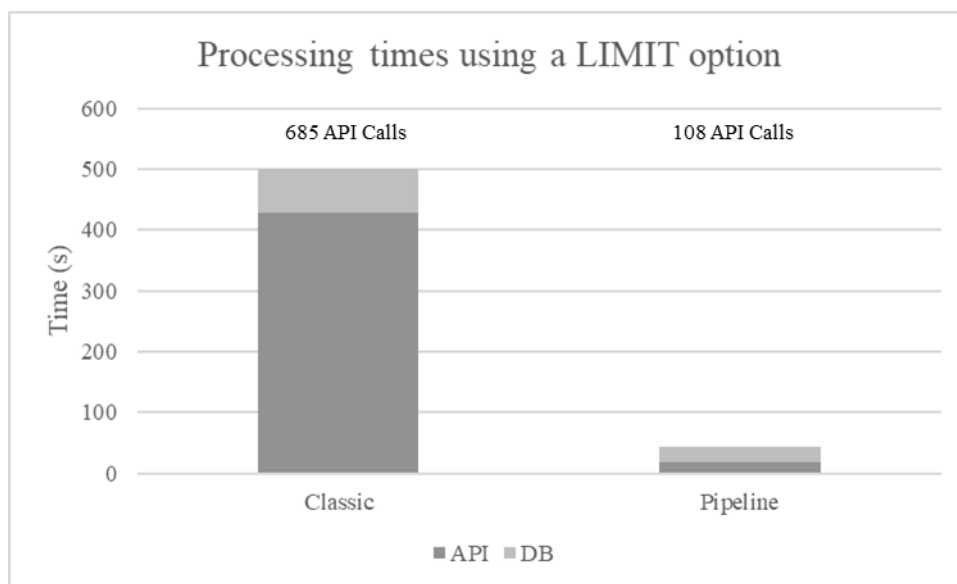


Figure 4.6: Comparison of the processing times of the classic and streaming query execution methods with the LIMIT option.



## 5. CONCLUSION AND FUTURE WORK

This work proposes a second version of the SPARQL extension for on-call incorporations of JSON API data into the results of SPARQL query, developed by Junemann, Reutter, Soto and Vrgoc and published for the 15<sup>th</sup> International Semantic Web Conference in Kobe, Japan (2016). We followed three axes of improvement which correspond to the addition of new incorporation capacities, the re-orientation of the strategy for the module integration into the SPARQL grammar, the implementation and test of optimizations to make the module better fit with its potential contexts of use.

The integration of an existing JSON navigation library for Java to the module allows to extend the capacities of incorporation, and to gain in robustness and scalability. If the choice of the Jayway Jsonpath appeared to be the best opportunity at this step of the project, it could be interesting in the future to consider preferring to this DSL the integration of Xpath or Xquery for being both standardized, dealing with the JSON documents thanks to their recent release, and generating an immediate possibility to incorporate XML API data. The JAQL project initiated by IBM researchers is also an alternative to further investigate for its capacity to query XML, CSV, flat files, structured SQL and Hadoop unstructured data in addition to JSON data. Indeed, one next step of the project is to start considering incorporating more types of data. The meta-format JSON, Yaml and XML are hugely generated by Web APIs and their incorporation should only change at the level of the navigation in the document. Incorporating data from relational database has also a big potential in terms of use cases generation.

We chose to implement the module as an update of the SERVICE feature for federated queries in SPARQL 1.1. It would be interesting to further investigate the possibilities to merge the API SERVICE implementation with the classic SERVICE feature. As explained, this project didn't focus on the authentication which is dealt by our module through the out-of-query addition of parameters. However, the Apache Jena implementation of SERVICE allows to manage a basic authentication to the SPARQL endpoint using `srv:queryAuthUser` and `srv:queryAuthPwd`. So, it could be considered to develop an update of those functionalities to manage extending authentication capacities enabling the API calls. Those considerations lead to consider also the optimization of the classic SERVICE feature to keep progressing in the usability of the federation queries and the Web Semantic. The state of the art survey about querying over federated SPARQL endpoints from Rakhmawati, Umbrich, Hasnain and Hausenblas (2013) concludes indeed with the need to further improvements to make current frameworks more effective in a broader range of applications.

Although the proposed implementations have passed successfully a large range of tests, we recommend more tests against practical use cases to identify problems and keep improving the module. Some limitations to resolve in the future have already been notified relative to the use of the method to minimize the API calls. Despite the proposed over-optimizations to reduce the time processing, the improvement is not sufficient as much of the constraining triples are kept duplicated in the pre-incorporation queries and the post-incorporation queries. Huge VALUES blocks are consequently manipulated and have a significant impact on the time processing of the query. A new optimization could be thought around the dependence of the variables used by distinct API calls. The purpose would be to identify

when a VALUES block generated by an incorporation should be pasted in the pre-incorporation query for another API call. As an API call has no impact on the values of the variables used as parameters (it adds data but do not alter the previously selected data), the VALUES block should be necessary in the first query only if the new incorporated variables are used as parameters in the next API SERVICE block. Finally, under the light of the exposed results, the caching strategy is for now the best option to use to reduce the number of API calls.

We identified a problem to take in consideration with the storage of float result values, such as the latitude and the longitude, in a SPARQL VALUES bloc. It implies an approximation of the values which makes incompatible the use of new (or copied/pasted) restrictions implicating them in the post-incorporation query. This problem appears especially with the minimization of API calls method as the constraining triples which allows to construct the VALUES bloc are copied and pasted in the last query. It could also appear in wrong formulations of queries. The user should however be aware that asking the query in the right order matters also a lot with the standardized SERVICE feature.

Finally, the experimental results reveal that the streaming method generates always similar or better performance indicators than the classic method. Its implementation as a definitive method and not only an optional one should consequently be considered and investigated with further comparative tests. The ultimate step which transforms the final Mapping Set into a Results Set needs to be reviewed to generate a correct streaming method.

## BIBLIOGRAFY

W3C (2015). Linked data. <https://www.w3.org/standards/semanticweb/>

Junemann, M., Reutter, J. L., Soto, A., & Vrgoc, D. (2016). Incorporating API Data into SPARQL Query Answers. In *International Semantic Web Conference (Posters & Demos)*.

Bizer, C., Heath, T., & Berners-Lee, T. (2009). Linked data-the story so far. *Semantic services, interoperability and web applications: emerging concepts*, 205-227.

Görlitz, O., & Staab, S. (2011). Federated data management and query optimization for linked open data. In *New Directions in Web Data Management 1* (pp. 109-137). Springer Berlin Heidelberg.

The Apache Jena Manual (2017). <http://jena.apache.org>, 2015.

Rakhmawati, N. A., Umbrich, J., Karnstedt, M., Hasnain, A., & Hausenblas, M. (2013). Querying over Federated SPARQL Endpoints---A State of the Art Survey. *arXiv preprint arXiv:1306.1723*.

Buil-Aranda, C., Arenas, M., & Corcho, O. (2011). Semantics and optimization of the SPARQL 1.1 federation extension. *The Semantic Web: Research and Applications*, 1-15.

Berners-Lee, T., Hendler, J., & Lassila, O. (2001). The semantic web. *Scientific american*, 284(5), 28-37.

Kobayashi, N., Ishii, M., Takahashi, S., Mochizuki, Y., Matsushima, A., & Toyoda, T. (2011). Semantic-JSON: a lightweight web service interface for Semantic Web contents integrating multiple life science databases. *Nucleic acids research*, 39(suppl\_2), W533-W540.

Battle, R., & Benson, E. (2008). Bridging the semantic Web and Web 2.0 with representational state transfer (REST). *Web Semantics: Science, Services and Agents on the World Wide Web*, 6(1), 61-69.

W3C (2017). Xquery 3.1. release specifications. <https://www.w3.org/TR/xquery-31/>

W3C (2017). Xpath 3.1. release specifications. <https://www.w3.org/TR/xpath-31/>

W3C (2015). Metaformats. <https://www.w3.org/standards/webarch/metaformats>

Open Data Portal (2017). <https://open-data.europa.eu/en/linked-data>

Discovery Team (2017). Wikidata query service / User Manual.  
[https://www.mediawiki.org/wiki/Wikidata\\_query\\_service/User\\_Manual#Federation](https://www.mediawiki.org/wiki/Wikidata_query_service/User_Manual#Federation)

Coombs, K. (2016). Federated Queries with SPARQL.

<https://www.oclc.org/developer/news/2016/federated-queries-with-sparql.en.html>

Cagle, K. (2016). Why the Semantic Web has failed. <https://www.linkedin.com/pulse/why-semantic-web-has-failed-kurt-cagle>

Goessner, S. (2007). JSONPath - XPath for JSON. <http://goessner.net/articles/JsonPath/>

Robie, J., Fourny, G., Brantner, D. F., Westmann, T. and Zaharioudakis, M. (2017). JSONiq, the JSON query language. <http://www.jsoniq.org/>

## **APPENDICES**

## APPENDIX A: JAYWAY JSONPATH DOCUMENTATION

### Source

<https://github.com/json-path/JsonPath>

### Getting Started

JsonPath is available at the Central Maven Repository. Maven users add this to your POM.

```
<dependency>
  <groupId>com.jayway.jsonpath</groupId>
  <artifactId>json-path</artifactId>
  <version>2.3.0</version>
</dependency>
```

If you need help ask questions at [Stack Overflow](#). Tag the question 'jsonpath' and 'java'.

JsonPath expressions always refer to a JSON structure in the same way as XPath expression are used in combination with an XML document. The "root member object" in JsonPath is always referred to as \$ regardless if it is an object or array.

JsonPath expressions can use the dot-notation

```
$.store.book[0].title
```

or the bracket-notation

```
$['store']['book'][0]['title']
```

### Operators

#### Operator

#### Description

\$

The root element to query. This starts all path expressions.

@

The current node being processed by a filter predicate.

\*

Wildcard. Available anywhere a name or numeric are required.

..

Deep scan. Available anywhere a name is required.

.<name>

Dot-notated child

['<name>' (,  
'<name>') ]

Bracket-notated child or children

[<number> (,  
<number>) ]

Array index or indexes

[start:end]

Array slice operator

[? (<expression>)]

Filter expression. Expression must evaluate to a boolean value.

## Functions

Functions can be invoked at the tail end of a path - the input to a function is the output of the path expression. The function output is dictated by the function itself.

Function Description		Output
min()	Provides the min value of an array of numbers	Double
max()	Provides the max value of an array of numbers	Double
avg()	Provides the average value of an array of numbers	Double
stddev()	Provides the standard deviation value of an array of numbers	Double
length()	Provides the length of an array	Integer

## Filter Operators

Filters are logical expressions used to filter arrays. A typical filter would be `[?(@.age > 18)]` where `@` represents the current item being processed. More complex filters can be created with logical operators `&&` and `||`. String literals must be enclosed by single or double quotes (`[?(@.color == 'blue')]` or `[?(@.color == "blue")]`).

### Operator Description

<code>==</code>	left is equal to right (note that 1 is not equal to '1')
<code>!=</code>	left is not equal to right
<code>&lt;</code>	left is less than right
<code>&lt;=</code>	left is less or equal to right
<code>&gt;</code>	left is greater than right
<code>&gt;=</code>	left is greater than or equal to right
<code>=~</code>	left matches regular expression <code>[?(@.name =~ /foo.*?/i)]</code>
<code>in</code>	left exists in right <code>[?(@.size in ['S', 'M'])]</code>
<code>nin</code>	left does not exists in right
<code>subsetof</code>	left is a subset of right <code>[?(@.sizes subsetof ['S', 'M', 'L'])]</code>
<code>size</code>	size of left (array or string) should match right
<code>empty</code>	left (array or string) should be empty



## APPENDIX B-1 : SOURCE CODE – CROSS PRODUCT IMPLEMENTATION

Function extracted from the file DatabaseWrapper.java

```

/*
 * FUNCTION: Update a mapping_array by mapping a new jsonValue
 * @param {ArrayList<HashMap<String, String>>} mapping_array
 * @param {Object} jsonValue
 * @param {String} bindName
 * @param {int} bindName_index
 * @param {HashMap<String, String>} initial_mapping
 * @return {ArrayList<HashMap<String, String>>}
 */
public ArrayList<HashMap<String, String>>
updateMappingArray(ArrayList<HashMap<String, String>> mapping_array, Object
jsonValue, String bindName, int bindName_index, HashMap<String, String>
initial_mapping) {
    int mapping_array_size = mapping_array.size();
    // CASE 1: value.class = Array of Elements
    if (jsonValue.getClass().equals(net.minidev.json.JSONArray.class)){
        for (int j=0; j<((net.minidev.json.JSONArray)jsonValue).size();
j++){
            // CASE 1.A: json_nav = first argument
            if(bindName_index==0){
                Object mapping_clone = initial_mapping.clone();
                mapping_array.add((HashMap<String,
String>)mapping_clone); // I initiate by cloning the mapping I had built into
all the mapping_array mappings
                mapping_array.get(mapping_array.size()-
1).put(bindName,
serializeValue(((net.minidev.json.JSONArray)jsonValue).get(j))); // I add to
the mapping_array mappings the relative JSON of the JSONArray
            }
            // CASE 1.B: json_nav = next arguments
            else {
                // I assign to each element of mapping_array the first
value of the new argument
                if(j==0){
                    for (int k=0; k<mapping_array.size(); k++){
                        mapping_array.get(k).put(bindName,
serializeValue(((net.minidev.json.JSONArray)jsonValue).get(j))); // I add to
the mapping_array mappings the relative JSON of the JSONArray
                    }
                }
                // For each next values, I first "duplicate" the
original mapping_array and then assign the value to the duplicate
                else {
                    for (int k=0; k<mapping_array_size; k++){
                        Object mapping_clone =
mapping_array.get(k).clone();
                        mapping_array.add((HashMap<String,
String>)mapping_clone);

```

```

mapping_array.get(mapping_array.size()-
1).put(bindName,
serializeValue(((net.minidev.json.JSONArray)jsonValue).get(j)));
    }
    }
    }
}
// CASE 2: value.class = Single Element
else {
    // CASE 2.A: json_nav = first argument
    if(bindName_index==0){
        initial_mapping.put(bindName, serializeValue(jsonValue));
        mapping_array.add(initial_mapping); // the ArrayList has a
size=1
    }
    // CASE 2.B: json_nav = next arguments
    else {
        for (int k=0; k<mapping_array.size(); k++){
            mapping_array.get(k).put(bindName,
serializeValue(jsonValue));
        }
    }
}
return mapping_array;
}

```

```

public class SPARQLSonParser {
    /*
     * FUNCTION: Parse a SPARQL query into the sections PREFIX, SELECT,
     FIRST, URL, PATH, ALIAS, LAST
     * @param {String} queryString
     * @param {Boolean} replace
     * @return {HashMap<String, Object>}
     */
    public static HashMap<String, Object> parseSPARQLSonQuery(String
queryString, boolean replace) {
        String[] firstParse = getSelectSection(queryString, replace);
        HashMap<String, Object> querySections =
getAPIServiceSection(firstParse[2]);
        querySections.put("PREFIX", firstParse[0]);
        querySections.put("SELECT", firstParse[1]);
        return querySections;
    }

    /*
     * FUNCTION: Get the Prefix, Select, and PostSelect parts of the query
     * @param {String} queryString
     * @param {Boolean} replace
     * @return {String[]}
     */
    public static String[] getSelectSection(String queryString, boolean
replace) {
        // Eliminate the multi-spaces before and after the query
        String newQueryString = queryString.trim();
        if(replace) {
            //Eliminate the line breaks which are not quoted
            newQueryString =
newQueryString.replaceAll("\\\\n(?:=([\\\\\\\\\\\\\\\\]|^\\\\\\\\\\\\\\\\))*\"([\\\\\\\\\\\\\\\\]|^\\\\\\\\\\\\\\\\])*\"*(\\\\\\\\\\\\\\\\\\\\\\\\|[^\\\\\\\\\\\\\\\\\\\\\\\\])*$\"", " ");
            //Eliminate the multi-spaces which are not quoted into the
request
            newQueryString =
newQueryString.replaceAll("\\\\s+(?:=([\\\\\\\\\\\\\\\\\\\\\\\\]|^\\\\\\\\\\\\\\\\\\\\\\\\))*\"([\\\\\\\\\\\\\\\\\\\\\\\\]|^\\\\\\\\\\\\\\\\\\\\\\\\))*\"*(\\\\\\\\\\\\\\\\\\\\\\\\|[^\\\\\\\\\\\\\\\\\\\\\\\\])*$\"", " ");
        }
        // Separate what is before the WHERE from what comes after
        int cutIndex = newQueryString.indexOf('{');
        String preSelectSection = newQueryString.substring(0, cutIndex +
1);

        String postSelectSection = newQueryString.substring(cutIndex + 1,
newQueryString.length());

        // Separate the PREFIX section from the SELECT section
        String prefixSection = "";
        String selectSection = "";
        String navigation_string = "(.*) (SELECT.*) $";
        Pattern pattern_variables = Pattern.compile(navigation_string);
        Matcher m = pattern_variables.matcher(preSelectSection);

```

```

        if (m.find()) {
            prefixSection = m.group(1);
            selectSection = m.group(2);
        }
        else {
            System.out.println("ERROR : No select query");
        }
        String[] retArray = {prefixSection, selectSection,
postSelectSection};
        return retArray;
    }

    /*
     * FUNCTION: Get the sections of the PostSelect part of the query which
are: FIRST, URL, PATH, ALIAS, LAST
     * @param {String} postSelectSection
     * @return {HashMap<String, Object>}
     */
    public static HashMap<String, Object> getAPIServiceSection(String
postSelectSection) {
        /*
         * Regex to match the format of a SERVICE call to an API:
         * This format is: FIRST SERVICE <URL> {($.PATH1, $.PATH2) AS
(ALIAS1, ALIAS 2)} LAST
         * Matched groups are:      Group 1: URL          Group 2: $.PATH1,
$.PATH2) AS (ALIAS1, ALIAS 2)} LAST
         * regexr.com:  +SERVICE +<([\w\-\%?\&\=\.\{\}\:\\/\,]+)> *\{ *\{
*(\$.*)$
         */
        // LAST
        String api_url_string = " +SERVICE +<([\w\-\%?\&\=\.\{\}\:\\/\,]+)> *\{ *\{ *(\$.*)$";
        Pattern pattern_variables = Pattern.compile(api_url_string);
        Matcher m1 = pattern_variables.matcher(postSelectSection);
        HashMap<String, Object> bindSections = new HashMap<String,
Object>();
        // Divide the query to keep the FIRST section, which comes before
the word 'SERVICE'
        String[] dividedQuery = postSelectSection.split(api_url_string, 2);
        bindSections.put("FIRST", dividedQuery[0]);

        if (m1.find()) {
            bindSections.put("URL", m1.group(1));
            /*
             * Regex to match the sections which come after the word
'AS' which are ALIAS and LAST
             * Matched groups are: Group 1: ALIAS1, ALIAS 2
             Group 2: LAST
             * regexr.com:  \) *AS *(((?: *\[?[\d\w]+ *,* *)*)\)) *\{*(.*)$
             */
            String post_api_url_string = "\\) *AS *(((?: *\[?[\d\w]+
*,* *)*)\)) *\{ *\\.(.*)$";

```

```

// Divide the Group 2 of m1 to keep only the PATH section of
the query
dividedQuery = m1.group(2).split(post_api_url_string, 2);
String json_nav_string = dividedQuery[0];
// Split the different paths which are separated by the
unquoted chain of character: , $
String[] json_navs = json_nav_string.split(",
*\\$(?=((\\\\\\\\[\\\\\\\\"]|[^\\\\\\\\"])*\\(\\\\\\\\[\\\\\\\\"]|[^\\\\\\\\"])*\\(\\\\\\\\[\\\\\\\\"]|
[^\\\\\\\\"])*$)");
for (int i=1 ; i<json_navs.length ; i++) {
    json_navs[i] = "$"+json_navs[i];
}
bindSections.put("PATH", json_navs);

pattern_variables = Pattern.compile(post_api_url_string);
Matcher m2 = pattern_variables.matcher(postSelectSection);
if (m2.find()) {
    String aliases_string = m2.group(1).trim();
    // Split the different aliases which are separated by
a ,
String[] aliases = aliases_string.split(", *");
for (int i = 0; i < aliases.length; i++) {
    // Eliminate the ? which is in front of the
alias
    aliases[i] = aliases[i].substring(1);
}
bindSections.put("ALIAS", aliases);
String post_aliases_string = m2.group(2).trim();
String options_regex = "(.*\\})([^\\}]*$)";
String options_section = "";
pattern_variables = Pattern.compile(options_regex);
Matcher m =
pattern_variables.matcher(post_aliases_string);
if (m.find()) {
    bindSections.put("LAST", m.group(1));
    bindSections.put("OPTIONS", m.group(2));
}
/*
 * Show the distinct sections of the query
 */
System.out.println("Request to the URL: " +
bindSections.get("URL"));
System.out.println("FIRST: " +
bindSections.get("FIRST"));
System.out.println("PATH: " + json_nav_string);
System.out.println("ALIAS: " + aliases_string);
System.out.println("LAST: " +
bindSections.get("LAST"));
}
}
return bindSections;
}
}

```

## APPENDIX B-3 : SOURCE CODE – PARSE TRIPLETS GROUPED BY BLOCK

```

public class TripletParser {

    /*
     * PROPERTIES
     */
    public String section_type;
    public ArrayList<String[]> triplets;
    public String service_uri;
    public String options;

    /*
     * CONSTRUCTORS
     */

    public TripletParser(String service_uri, String[] triplet, String
options) {
        if(service_uri!=null) {
            this.section_type = "sparql_service";
            this.service_uri = service_uri;
        }
        else {
            this.section_type = "basic";
            this.service_uri = null;
        }
        this.triplets = new ArrayList<String[]>();
        this.triplets.add(triplet);
        this.options = options;
    }

    public TripletParser(String service_uri, String queryPart) {
        if(service_uri!=null) {
            this.section_type = "sparql_service";
            this.service_uri = service_uri;
        }
        else {
            this.section_type = "basic";
            this.service_uri = null;
        }
        this.triplets = new ArrayList<String[]>();
        // Match ?a ?b ?c ; ?d ?e . and transform into ?a ?b ?c . ?a ?d ?e
        .
        String triplet_regex = "((<[\\w\\-
\\%\\?\\&|=\\.\\{\\}\\:\\:\\/\\,]+|\\?\\w+|\\w+:\\w+) +(<[\\w\\-
\\%\\?\\&|=\\.\\{\\}\\:\\:\\/\\,]+|\\?\\w+|\\w+:\\w+) +(<[\\w\\-
\\%\\?\\&|=\\.\\{\\}\\:\\:\\/\\,]+|\\?\\w+|\\w+:\\w+)) *;(.*$)";
        Pattern pattern_triplet = Pattern.compile(triplet_regex);
        Matcher matcherTriplet = pattern_triplet.matcher(queryPart);
        while(matcherTriplet.find()) {
            queryPart = matcherTriplet.group(1) + " . " +
matcherTriplet.group(2) + " " + matcherTriplet.group(5).trim();
            matcherTriplet = pattern_triplet.matcher(queryPart);
        }
    }
}

```

```

        // Match the triplets add them to the TripletParser
        triplet_regex = "((<[\\w\\-
\\%\\?\\&\\|=\\.\\.\\{\\}\\:\\:\\/\\\\,]+>|\\?\\w+|\\w+:\\w+) +(<[\\w\\-
\\%\\?\\&\\|=\\.\\.\\{\\}\\:\\:\\/\\\\,]+>|\\?\\w+|\\w+:\\w+) +(<[\\w\\-
\\%\\?\\&\\|=\\.\\.\\{\\}\\:\\:\\/\\\\,]+>|\\?\\w+|\\w+:\\w+) *\\.\\.(\\.\\$)";
        pattern_triplet = Pattern.compile(triplet_regex);
        matcherTriplet = pattern_triplet.matcher(queryPart);
        while(matcherTriplet.find()) {
            String[] triplets = new String[]{matcherTriplet.group(1),
matcherTriplet.group(2), matcherTriplet.group(3)};
            this.triplets.add(triplets);
            queryPart = matcherTriplet.group(4);
            matcherTriplet = pattern_triplet.matcher(queryPart);
        }
        this.options = queryPart.trim();
    }

    public TripletParser(String queryPart) {
        this.section_type = "basic";
        this.service_uri = null;
        this.triplets = new ArrayList<String[]>();
        // Match ?a ?b ?c ; ?d ?e . and transform into ?a ?b ?c . ?a ?d ?e
        .

        String triplet_regex = "((<[\\w\\-
\\%\\?\\&\\|=\\.\\.\\{\\}\\:\\:\\/\\\\,]+>|\\?\\w+|\\w+:\\w+) +(<[\\w\\-
\\%\\?\\&\\|=\\.\\.\\{\\}\\:\\:\\/\\\\,]+>|\\?\\w+|\\w+:\\w+) +(<[\\w\\-
\\%\\?\\&\\|=\\.\\.\\{\\}\\:\\:\\/\\\\,]+>|\\?\\w+|\\w+:\\w+) *\\.\\.(\\.\\$)";
        Pattern pattern_triplet = Pattern.compile(triplet_regex);
        Matcher matcherTriplet = pattern_triplet.matcher(queryPart);
        while(matcherTriplet.find()) {
            queryPart = matcherTriplet.group(1) + " . " +
matcherTriplet.group(2) + " " + matcherTriplet.group(5).trim();
            matcherTriplet = pattern_triplet.matcher(queryPart);
        }
        // Match the triplets add them to the TripletParser
        triplet_regex = "((<[\\w\\-
\\%\\?\\&\\|=\\.\\.\\{\\}\\:\\:\\/\\\\,]+>|\\?\\w+|\\w+:\\w+) +(<[\\w\\-
\\%\\?\\&\\|=\\.\\.\\{\\}\\:\\:\\/\\\\,]+>|\\?\\w+|\\w+:\\w+) +(<[\\w\\-
\\%\\?\\&\\|=\\.\\.\\{\\}\\:\\:\\/\\\\,]+>|\\?\\w+|\\w+:\\w+) *\\.\\.(\\.\\$)";
        pattern_triplet = Pattern.compile(triplet_regex);
        matcherTriplet = pattern_triplet.matcher(queryPart);
        while(matcherTriplet.find()) {
            String[] triplets = new String[]{matcherTriplet.group(1),
matcherTriplet.group(2), matcherTriplet.group(3)};
            this.triplets.add(triplets);
            queryPart = matcherTriplet.group(4);
            matcherTriplet = pattern_triplet.matcher(queryPart);
        }
        this.options = queryPart.trim();
    }

}

/*
 * METHODS
 */

```

```

/*
 * FUNCTION: Add a triplet
 * @param {String[]} triplet
 * @return {}
 */
public void addTriplet(String[] triplet) {
    this.triplets.add(triplet);
}

/*
 * FUNCTION: Parse SPARQL Query constraints into a list of TripletParsers
 * corresponding to the relative SPARQL blocks
 * @param {String} sparqlQuerySection
 * @return {ArrayList<TripletParser>}
 */
public static ArrayList<TripletParser> getParsedSPARQLBlocks(String
sparqlQuerySection) {
    // Match ?a ?b ?c ; ?d ?e . and transform into ?a ?b ?c . ?a ?d ?e
    .

    String triplet_regex = "(.*)((<[\\w\\-
\\%\\?\\&|=\\.\\{\\}\\:\\:\\/\\,\\,]+|\\|?\\w+|\\w+:\\w+) +(<[\\w\\-
\\%\\?\\&|=\\.\\{\\}\\:\\:\\/\\,\\,]+|\\|?\\w+|\\w+:\\w+) +(<[\\w\\-
\\%\\?\\&|=\\.\\{\\}\\:\\:\\/\\,\\,]+|\\|?\\w+|\\w+:\\w+)) *;(.*$)";
    Pattern pattern_triplet = Pattern.compile(triplet_regex);
    Matcher matcherTriplet =
pattern_triplet.matcher(sparqlQuerySection);
    while(matcherTriplet.find()) {
        sparqlQuerySection = matcherTriplet.group(1) +
matcherTriplet.group(2) + " . " + matcherTriplet.group(3) + " " +
matcherTriplet.group(6).trim();
        matcherTriplet =
pattern_triplet.matcher(sparqlQuerySection);
    }

    ArrayList<TripletParser> parsedFirstQuery = new
ArrayList<TripletParser>();
    String api_url_string = "(.*) *SERVICE +(<[\\w\\-
\\%\\?\\&|=\\.\\{\\}\\:\\:\\/\\,\\,]+> *\\{\\{([^\\}]*\\}\\}\\} *(.*$)";
    Pattern pattern_variables = Pattern.compile(api_url_string);
    triplet_regex = "<[\\w\\-
\\%\\?\\&|=\\.\\{\\}\\:\\:\\/\\,\\,]+|\\|?\\w+|\\w+:\\w+) +(<[\\w\\-
\\%\\?\\&|=\\.\\{\\}\\:\\:\\/\\,\\,]+|\\|?\\w+|\\w+:\\w+) +(<[\\w\\-
\\%\\?\\&|=\\.\\{\\}\\:\\:\\/\\,\\,]+|\\|?\\w+|\\w+:\\w+) *\\.\\.(.*$)";
    pattern_triplet = Pattern.compile(triplet_regex);
    String query_string = sparqlQuerySection;
    Matcher m = pattern_variables.matcher(query_string);
    while(m.find()) {
        matcherTriplet = pattern_triplet.matcher(m.group(4));
        if(matcherTriplet.find()) {
            TripletParser basic_section = new
TripletParser(m.group(4));
            parsedFirstQuery.add(0, basic_section);
        }
    }

```



```

        TripletParser sparql_service_section = new
TripletParser(m.group(2), m.group(3));
        parsedFirstQuery.add(0, sparql_service_section);

        query_string = m.group(1);
        m = pattern_variables.matcher(query_string);
    }
    matcherTriplet = pattern_triplet.matcher(query_string);
    if(matcherTriplet.find()) {
        TripletParser basic_section = new
TripletParser(query_string);
        parsedFirstQuery.add(0, basic_section);
    }
    return parsedFirstQuery;
}

/*
 * FUNCTION: Add a triplet to a list of TripletParsers representing a
query
 * @param {String} sparqlQuerySection
 * @return {ArrayList<TripletParser>}
 */
    public static void addTripletToParsedQuery(ArrayList<TripletParser>
list_parsed_triplets, TripletParser parsed_triplets, int index_triplet) {
        // The triplet is added to a new section
        if (list_parsed_triplets.size()==0) {
            TripletParser new_section = new
TripletParser(parsed_triplets.service_uri,
parsed_triplets.triplets.get(index_triplet), parsed_triplets.options);
            list_parsed_triplets.add(new_section);
        }
        else if (parsed_triplets.service_uri !=
list_parsed_triplets.get(list_parsed_triplets.size()-1).service_uri) {
            TripletParser new_section = new
TripletParser(parsed_triplets.service_uri,
parsed_triplets.triplets.get(index_triplet), parsed_triplets.options);
            list_parsed_triplets.add(new_section);
        }
        // The triplet is added to the last created section
        else {
            list_parsed_triplets.get(list_parsed_triplets.size()-
1).addTriplet(parsed_triplets.triplets.get(index_triplet));
        }
    }

/*
 * FUNCTION: Transform a list of TripletParsers into a SPARQL Query
section
 * @param {ArrayList<TripletParser>} parsedTriplets
 * @return {String}
 */
    public static String reverseParsedSPARQLBlocks(ArrayList<TripletParser>
parsedTriplets) {
        String query= "";

```

```

        for (int i=0; i<parsedTriplets.size();i++) {
            String triplets_string = "";
            for (int j=0; j< parsedTriplets.get(i).triplets.size(); j++)
            {
                for (int k=0; k<3; k++) {
                    triplets_string +=
parsedTriplets.get(i).triplets.get(j)[k] + " ";
                }
                triplets_string += ". ";
            }
            if(parsedTriplets.get(i).section_type == "sparql_service" &&
triplets_string!="") {
                query += "SERVICE <" +
parsedTriplets.get(i).service_uri + "> {" + triplets_string +
parsedTriplets.get(i).options + "} " ;
            }
            else if (triplets_string!=""){
                query += triplets_string +
parsedTriplets.get(i).options;
            }
        }
        return query;
    }
}

```

## APPENDIX B-4 : SOURCE CODE – API OPTIMIZATION

```

public class ApiOptimizer {

    /*
     * PROPERTIES
     */

    int apiCalls;
    long timeApi;
    ArrayList<String> cacheKeys;
    HashMap<String, Object> cache;
    static final int CACHE_SIZE = 400;

    /*
     * CONSTRUCTORS
     */

    public ApiOptimizer() {
        this.cacheKeys = new ArrayList<String>();
        this.cache = new HashMap<>();
        this.timeApi = 0;
        this.apiCalls = 0;
    }

    /*
     * METHODS
     */

    /*
     * FUNCTION: Apply the cache option to retrieve the JSON response
     * @param {String} url_req
     * @param {HashMap<String, String>} params
     * @param {GetJSONStrategy} strategy
     * @return {Object}
     */
    public Object retrieve_json(String url_req, HashMap<String,String>
params, GetJSONStrategy strategy) throws JSONException, Exception {
        if (params.containsKey("cache") &&
params.get("cache").equals("true")) {
            if (cache.containsKey(url_req)) {
                return cache.get(url_req);
            }
            else {
                this.apiCalls += 1;
                Object json = ApiWrapper.getJSON(url_req, params,
strategy);

                if (cacheKeys.size() < CACHE_SIZE) {
                    cacheKeys.add(url_req);
                    cache.put(url_req, json);
                }
                else {
                    String removed_key = cacheKeys.remove(0);
                    cache.remove(removed_key);

```

```

        cache.put(url_req, json);
    }
    return json;
}
}
else {
    this.apiCalls += 1;
    return ApiWrapper.getJSON(url_req, params, strategy);
}
}

/*
 * FUNCTION: Transform the parsed query to minimize the number of calls
to API by Service
 * @param {HashMap<String, Object>} parsedQuery
 * @return {HashMap<String, Object>}
 */
public HashMap<String, Object> minimizeAPICall(HashMap<String, Object>
parsedQuery) {
    // Look for inserted variables in the URL of the API-Service call
and keep them
    String variable_in_URL = "(.*)=\\{([\\^\\}]+)\\}.*$";
    ArrayList<String> inserted_variables = new ArrayList<String>();
    Pattern pattern_variables = Pattern.compile(variable_in_URL);
    Matcher m =
pattern_variables.matcher((String)parsedQuery.get("URL"));
    while(m.find()){
        inserted_variables.add(m.group(2));
        m = pattern_variables.matcher(m.group(1));
    }
    for (int i=0; i<inserted_variables.size(); i++) {
        inserted_variables.set(i, "?" + inserted_variables.get(i));
    }

    // Separate an eventual values_section from the rest of the FIRST
section
    String values_regex = "(.*) ( *VALUES *\\{([\\^\\}]*\\) *\\{([\\^\\}]* *\\}
*)(.*)";
    String values_section = "";
    pattern_variables = Pattern.compile(values_regex);
    m = pattern_variables.matcher((String)parsedQuery.get("FIRST"));
    while(m.find()){
        values_section += m.group(2);
        parsedQuery.put("FIRST", m.group(1) + " " + m.group(3));
        m =
pattern_variables.matcher((String)parsedQuery.get("FIRST"));
    }

    // Parse the first section into a list of TripletParsers
corresponding to the relative SPARQL blocks
    ArrayList<TripletParser> triplets_to_put_first =
TripletParser.getParsedSPARQLBlocks((String)parsedQuery.get("FIRST"));
    ArrayList<TripletParser> triplets_to_put_last = new
ArrayList<TripletParser>();

```

```

        if (inserted_variables.size() > 0) {
            // Eject the triplets constraining the inserted variables
            // (no other variable in the triplet)
            triplets_to_put_last =
            ejectConstrainingTriplets(inserted_variables, triplets_to_put_first);

            // Eject the triplets which are independent from the
            // selected_variables from the first query part
            ejectIndependantTriplets(inserted_variables,
            triplets_to_put_first);

            // Remove from the first part of the query the useless
            // triplets which are called later
            parsedQuery.put("FIRST", values_section + " " +
            TripletParser.reverseParsedSPARQLBlocks(triplets_to_put_first));
            // Add to the last part of the query the triplets which are
            // needed to Select the other variables later
            parsedQuery.put("LAST", values_section + " " +
            TripletParser.reverseParsedSPARQLBlocks(triplets_to_put_last) +
            (String)parsedQuery.get("LAST"));
        }
        else {
            // Put all the FIRST section after the API call
            triplets_to_put_last = triplets_to_put_first;
            parsedQuery.put("FIRST", "");
            parsedQuery.put("LAST", values_section + " " +
            TripletParser.reverseParsedSPARQLBlocks(triplets_to_put_last) +
            (String)parsedQuery.get("LAST"));
        }
        // Stock in a String the variables to Select before calling the API
        String needed_variables_string = "";
        for (String var: inserted_variables) {
            needed_variables_string += var + " ";
        }
        // Add to the parsed Query the variables to Select before calling
        // the API through Service
        parsedQuery.put("VARS", needed_variables_string);
        return parsedQuery;
    }

    /**
     * FUNCTION: Eject the triplets constraining the specified variables (no
     * other variable in the triplet)
     * @param {ArrayList<String>} vars
     * @param {ArrayList<TripletParser>} list_parsed_triplets
     * @return {ArrayList<TripletParser>}
     */
    public ArrayList<TripletParser>
    ejectConstrainingTriplets(ArrayList<String> vars, ArrayList<TripletParser>
    list_parsed_triplets) {

```

```

        ArrayList<TripletParser> selected_triplets = new
ArrayList<TripletParser>();
        // Iterate over the linked variables
        for (String var: vars) {
            // Iterate over the (basic and SPARQL-Service) sections of
the first part of the query
            for (int section=0; section< list_parsed_triplets.size();
section++) {
                // Iterate over the list of triplets in the section
                for (int triplet=0;
triplet<list_parsed_triplets.get(section).triplets.size(); triplet++) {
                    int element = 0;
                    boolean constraint_triplet = false;
                    // Iterate over the elements of the triplet
                    while (element<3) {

                        if(var.equals(list_parsed_triplets.get(section).triplets.get(triplet)[ele
ment])) {
                            constraint_triplet = true;
                            for (int other_element=0;
other_element<3; other_element++) {
                                if(other_element!=element &&
list_parsed_triplets.get(section).triplets.get(triplet)[other_element].startsWith("?")) {
                                    // The linked
                                    constraint_triplet =
false;
                                }
                            }
                            element = 3;
                        }
                        else {
                            element +=1;
                        }
                    }
                    // The triplet does not contain only the linked
variable as a variable
                    if (!constraint_triplet) {

                        TripletParser.addTripletToParsedQuery(selected_triplets,
list_parsed_triplets.get(section), triplet);
                    }
                }
            }
        }
        return selected_triplets;
    }

    /*
    * FUNCTION: Eject the triplets which are not linked in the data graph
with the specified variables
    * @param {ArrayList<String>} vars

```

```

    * @param {ArrayList<TripletParser>} list_parsed_triplets
    * @return {ArrayList<TripletParser>}
    */
    public void ejectIndependantTriplets(ArrayList<String> vars,
ArrayList<TripletParser> list_parsed_triplets) {
        // Create a local list of variables
        ArrayList<String> local_vars = new ArrayList<String>();
        local_vars.addAll(vars);
        // Initialize the loop
        HashMap<String, Object> opt = ejectIncludingTriplets(local_vars,
list_parsed_triplets);
        ArrayList<TripletParser> triplets_to_eject = new
ArrayList<TripletParser>();
        boolean loop = true;
        // Loop which excludes the triplets linking variables to anterior
linked variables from the triplets to eject
        while (loop) {
            triplets_to_eject =
(ArrayArrayList<TripletParser>)opt.get("selected triplets");
            if
(!local_vars.containsAll((ArrayList<String>)opt.get("linked variables")))) {

                local_vars.addAll((ArrayList<String>)opt.get("linked variables"));
                opt =
ejectIncludingTriplets((ArrayList<String>)opt.get("linked variables"),
triplets_to_eject);
            }
            else {
                loop = false;
            }
        }
        for (int section=0; section< triplets_to_eject.size(); section++) {
            for (int triplet=0;
triplet< triplets_to_eject.get(section).triplets.size(); triplet++) {
                for (int s=0; s<list_parsed_triplets.size(); s++) {
                    for (int t=0;
t<list_parsed_triplets.get(s).triplets.size(); t++) {
                        if
(list_parsed_triplets.get(s).triplets.get(t)==triplets_to_eject.get(section).tr
iplets.get(triplet)
&&
list_parsed_triplets.get(s).service_uri==triplets_to_eject.get(section).service
_uri) {

                            list_parsed_triplets.get(s).triplets.remove(t);
                        }
                    }
                }
            }
        }
    }
}
/*

```

```

    * FUNCTION: Eject the triplets including the specified variables and
    stock the other variables linked by those triplets
    * @param {ArrayList<String>} vars
    * @param {ArrayList<TripletParser>} list_parsed_triplets
    * @return {HashMap<String, Object>}
    */
    public HashMap<String, Object> ejectIncludingTriplets(ArrayList<String>
vars, ArrayList<TripletParser> list_parsed_triplets) {
        ArrayList<String> linked_variables = new ArrayList<String>();
        ArrayList<TripletParser> selected_triplets = new
ArrayList<TripletParser>();
        // Iterate over the inserted variables
        for (String var: vars) {
            // Iterate over the (basic and SPARQL-Service) sections of
the first part of the query
            for (int section=0; section< list_parsed_triplets.size();
section++) {
                // Iterate over the list of triplets in the section
                for (int triplet=0;
triplet<list_parsed_triplets.get(section).triplets.size(); triplet++) {
                    int element = 0;
                    boolean include = false;
                    // Iterate over the elements of the triplet
                    while (element<3) {

                        if(var.equals(list_parsed_triplets.get(section).triplets.get(triplet)[ele
ment])) {
                            for (int other_element=0;
other_element<3; other_element++) {
                                if(other_element!=element &&
list_parsed_triplets.get(section).triplets.get(triplet)[other_element].startsWi
th("?")) {
                                    // Stock the
variables linked to inserted variables by a triplet

                                    linked_variables.add(list_parsed_triplets.get(section).triplets.get(tripl
et)[other_element]);
                                }
                            }
                            element=3;
                            include = true;
                        } else { element +=1; }
                    }
                    // Add the triplet which doesn't contain an
inserted variable to the selected triplets
                    if (!include) {

                        TripletParser.addTripletToParsedQuery(selected_triplets,
list_parsed_triplets.get(section), triplet);

                    }
                }
            }
        }
    }
}

```



```
HashMap<String, Object> result = new HashMap<String, Object>();  
result.put("linked_variables", linked_variables);  
result.put("selected_triplets", selected_triplets);  
return result;  
}  
}
```

## APPENDIX B-5 : SOURCE CODE – PIPELINE IMPLEMENTATION

Functions extracted from the file DatabaseWrapper.java

```

/*
 * FUNCTION: One mapping after the other, map the JSON data from the API with
 the RDF data from the complete sparql query
 * @param {HashMap<String, Object>} parsedQuery
 * @param {ArrayList<GetJSONStrategy>} strategy
 * @param {ArrayList<HashMap<String, String>>} params
 * @param {int} limit
 * @return {MappingSet}
 */
public MappingSet execQueryPipeURL(HashMap<String, Object> parsedQuery,
    ArrayList<GetJSONStrategy> strategy, ArrayList<HashMap<String,
String>> params, int limit)
    throws JSONException, Exception {
    String firstQuery = retrieve_firstQuery (parsedQuery, params.get(0));
    System.out.println(firstQuery);
    Boolean distinctQuery =
retrieve_distinct((String)parsedQuery.get("SELECT"));
    String[] bindName = (String[])parsedQuery.get("ALIAS");
    String[] jpath = (String[])parsedQuery.get("PATH");
    Query query = QueryFactory.create(firstQuery);
    QueryExecution qexec;
    Dataset dataset;
    if(!FUSEKI_ENABLED) {
        dataset = TDBFactory.createDataset(this.TDBdirectory);
        qexec = QueryExecutionFactory.create(query, dataset);
    }
    else {
        qexec =
QueryExecutionFactory.sparqlService("http://localhost:3030/ds", query);
    }
    MappingSet ms = new MappingSet();
    ArrayList<String> ms_varnames = new ArrayList<String>();
    try {
        // Assumption: it's a SELECT query.
        ResultSet rs = qexec.execSelect() ;
        // Materialize the results to be able to free the system resources
for the next query executions
        rs = ResultSetFactory.copyResults(rs) ;
        // QueryExecution objects should be closed to free any system
resources
        qexec.close();

        List<String> vars_name = rs.getResultVars();
        for (String vn: vars_name) {
            ms_varnames.add(vn);
        }
        for (String bn: bindName) {
            ms_varnames.add(bn);
        }
    }

```

```

// The order of results is undefined.
while (rs.hasNext() && this.mappingCount < limit) {
    MappingSet ms_temp = new MappingSet();
    ms_temp.set_var_names(ms_varnames);

    QuerySolution rb = rs.nextSolution() ;
    HashMap<String, String> mapping = mappQuerySolution(rb,
vars_name);

    String url_req =
ApiWrapper.insertValuesURL((String)parsedQuery.get("URL"), rb,
params.get(0).get("replace_string"));
    Object json = null;
    try {
        long start = System.nanoTime();
        json = this.apiOptimizer.retrieve_json(url_req,
params.get(0), strategy.get(0));
        long stop = System.nanoTime();
        this.apiOptimizer.timeApi += (stop - start);
    }
    catch (Exception name) {
        System.out.println("ERROR: " + name);
        for (int i = 0; i < bindName.length; i++) {
            mapping.put(bindName[i], "UNDEF");
        }
    }
    if (json != null) {
        ArrayList<HashMap<String, String>> mapping_array = new
ArrayList<HashMap<String, String>>();
        for (int i = 0; i < bindName.length; i++) {
            try {
                Object value =
JsonPath.parse(json).read(jpath[i]);
                mapping_array =
updateMappingArray(mapping_array, value, bindName[i], i, mapping);
            }
            catch (Exception name) {
                System.out.println("ERROR: " + name);
                // CASE 0.A: json_nav = first argument
                if(i==0){
                    mapping.put(bindName[i], "UNDEF");
                    mapping_array.add(mapping); // the
ArrayList has a size=1
                }
                // CASE 0.B: json_nav = next arguments
                else {
                    for (int k=0;
k<mapping_array.size(); k++){
                        mapping_array.get(k).put(bindName[i], "UNDEF");
                    }
                }
            }
        }
    }
}
}

```

```

// Add all the mappings relative to the result rb to
the MappingSet to return
    for (int k=0; k<mapping_array.size(); k++){
        ms_temp.addMapping(mapping_array.get(k));
    }
}
// Recursive condition is that the LAST section of
parsedQuery includes another API service section
String api_url_string = " +SERVICE +<([\\w\\-
\\%\\?\\&\\=\\.\\{\\}\\:\\/\\,\\+)> *\\{ *\\( *(\\$.*$)";
Pattern pattern_variables = Pattern.compile(api_url_string);
Matcher m = pattern_variables.matcher(" " + (String)
parsedQuery.get("LAST"));
String recursive_query_string = ((String)
parsedQuery.get("PREFIX")) + ((String) parsedQuery.get("SELECT")) +
ms_temp.serializeAsValues() + (String) parsedQuery.get("LAST") + (String)
parsedQuery.get("OPTIONS");
    if (m.find()) {
        strategy.get(1).set_params(params.get(1));
        ms_temp =
execQueryPipeURL(SPARQLSonParser.parseSPARQLSonQuery(recursive_query_string,
false),
new
ArrayList<GetJSONStrategy>(strategy.subList(1, strategy.size())),
new
ArrayList<HashMap<String,String>>(params.subList(1, params.size())),
limit);
    }
    else {
        ms_temp = mappPostBindQuery(recursive_query_string,
limit);
    }
ms.set_var_names(ms_temp.var_names);
if(distinctQuery) {
    int ms_size = ms.mappings.size();
    ms.addDistinctMappingsFromMappingSet(ms_temp);
    // We reduce the mappingCount as the no-distinct
mappings are not added
    this.mappingCount += ms.mappings.size() - ms_size -
ms_temp.mappings.size();
}
    else {
        for (int k=0; k<ms_temp.mappings.size(); k++){
            ms.addMapping(ms_temp.mappings.get(k));
        }
    }
}
}
finally
{
    if(!FUSEKI_ENABLED) {
        dataset.close();
    }
}

```

```

        return ms;
    }

    /*
     * FUNCTION: Retrieve a SELECT DISTINCT query
     * @param {HashMap<String, Object>} parsedQuery
     * @param {HashMap<String, String>} params
     * @param {String} selectQuery
     * @return {Boolean}
     */
    public Boolean retrieve_distinct(String selectQuery) {
        String regex = "SELECT DISTINCT";
        Pattern pattern_variables = Pattern.compile(regex);
        Matcher m = pattern_variables.matcher(selectQuery);
        if (m.find()) {
            return true;
        }
        else {
            return false;
        }
    }

    /*
     * FUNCTION: Transform a sparql queryString into a MappingSet
     * @param {String} queryString
     * @param {int} limit
     * @return {MappingSet}
     */
    public MappingSet mappPostBindQuery(String queryString, int limit) {
        System.out.println(queryString);
        Query query = QueryFactory.create(queryString);
        QueryExecution qexec;
        Dataset dataset;
        if (!FUSEKI_ENABLED) {
            dataset = TDBFactory.createDataset(this.TDBdirectory);
            qexec = QueryExecutionFactory.create(query, dataset);
        }
        else {
            qexec =
QueryExecutionFactory.sparqlService("http://localhost:3030/ds", query);
        }
        MappingSet ms = new MappingSet();
        ArrayList<String> ms_varnames = new ArrayList<String>();
        try {
            // Assumption: it's a SELECT query.
            ResultSet rs = qexec.execSelect();
            rs = ResultSetFactory.copyResults(rs);
            // QueryExecution objects should be closed to free any system
resources
            qexec.close();

            List<String> vars_name = rs.getResultVars();
            for (String vn: vars_name) {
                ms_varnames.add(vn);
            }
        }
        catch (Exception e) {
            e.printStackTrace();
        }
    }
}

```

```

    }
    ms.set_var_names(ms_varnames);
    // The order of results is undefined.
    while (rs.hasNext()) {
        QuerySolution rb = rs.nextSolution() ;
        System.out.println("--DEBUG-- "+rb);
        HashMap<String, String> mapping = mappQuerySolution(rb,
vars_name);
        ms.addMapping(mapping);
        this.mappingCount += 1;
    }
}
finally
{
    if(!FUSEKI_ENABLED) {
        dataset.close();
    }
}
return ms;
}

```

## APPENDIX C-1 : CONSTRUCTION METHOD – TESTING RDF DATABASE

Query that we ran on the DBPedia endpoint with the online tool virtuoso:

```
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX xmlns: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX geo: <http://www.w3.org/2003/01/geo/wgs84_pos#>

construct { ?place xmlns:type <http://dbpedia.org/class/yago/Capital108518505> .
?place <http://dbpedia.org/ontology/country> ?country .
?place geo:long ?long .
?place geo:lat ?lat .
}
where { ?place xmlns:type <http://dbpedia.org/class/yago/Capital108518505> .
?place <http://dbpedia.org/ontology/country> ?country .
?place geo:long ?long .
?place geo:lat ?lat .
}
```

The results were added to a file input.ttl which has been used as parameter of the LoadTDB.

The next error appeared while loading the database:

```
1619 [main] ERROR org.apache.jena.riot - [line: 5527, col: 14] Failed to find
a prefix name or keyword: -(8211;0x2013)
```

The cause was the use of the “-“ character in the next triple that we chose so to delete:

```
dbr:Phan_Rang-Tháp_Chàm rdf:type yago:Capital108518505 ;
    geo:lat "11.566666603088378906"^^xsd:float ;
    geo:long "108.98332977294921875"^^xsd:float ;
    dbo:country dbr:Vietnam .
```

➔ DELETE 1 entity

**APPENDIX C-2 : SOURCE CODE - TESTING JSON API**

Language : NodeJS

```
var express = require('express');
var app = express();

var object1 =
    {
        "value_1": ["el_a", "el_b", "el_c"],
        "value_2": ["el_1", "el_2", "el_3"],
        "version": ["v1", "v2"]
    };

app.get('/api', function (req, res) {
    res.send(object1);
});

app.get('/:variable', function (req, res) {
    var result = { };
    result.variable = req.params.variable;
    result.object = object1;
    res.send(result);
});

app.listen(3000, function () {
    console.log('Example app listening on port 3000!')
});
```



## APPENDIX D : PRACTICAL USE CASE – QUERY & RESULTS DETAILS

The executed query is:

```
PREFIX geo: <http://www.w3.org/2003/01/geo/wgs84_pos#>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX xmlns: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX dbo: <http://dbpedia.org/ontology/>
PREFIX dbr: <http://dbpedia.org/resource/>
PREFIX yago: <http://dbpedia.org/class/yago/>

SELECT ?label ?retweet ?weather WHERE {
  ?place dbo:country dbr:Chile ;
          xmlns:type yago:Capital108518505 .
  SERVICE <http://dbpedia.org/sparql> {
    ?place rdfs:label ?label .
    FILTER(lang(?label) = 'es') .
  } .
  SERVICE <https://api.twitter.com/1.1/search/tweets.json?
    q={label}&result_type=recent>{
    ($.statuses[*].retweet_count) AS (?retweet)}
  SERVICE <http://api.openweathermap.org/data/2.5/weather?
    q={label},Chile&appid=99ac6530dcdb78fa4c02d08ec5297a52>{
    ($.weather[*].description) AS (?weather) }
}
```

In the “query” column of the next tables, the value “T&W” refers to the execution of the whole query while the values “Weather” and “Tweets” refer respectively to the execution of the query without the SERVICE bloc calling the Twitter API, and without the Openweather API.

Table II-1: Results details for the comparison of the 4 execution methods.

N° Tests Set	Query	Pipeline	Min_API	LIMIT	Mappings	API Calls	Total (s)	API (s)	DB (s)
1	T & W	false	false	-	635	687	583,11	523,97	59,14
2	T & W	false	false	-	631	683	414,56	397,95	16,61
3	T & W	false	false	-	636	688	438,78	417,91	20,87
1	T & W	false	true	-	637	104	1685,05	128,24	1556,82
2	T & W	false	true	-	632	104	1370,72	82,95	1287,77
3	T & W	false	true	-	625	104	1423,11	92,1	1331,01
1	T & W	true	false	-	636	688	529,05	478,1	50,95
2	T & W	true	false	-	634	687	458,99	403,52	55,47
3	T & W	true	false	-	631	683	429,82	397,34	32,48
1	T & W	true	true	-	636	104	884,47	93,45	791,02
2	T & W	true	true	-	633	104	993,5	116,94	876,55
3	T & W	true	true	-	648	104	867,25	102,93	764,32

Table II-2: Results mean values (obtained from details) for the comparison of the 4 execution methods.

	Query	Pipeline	Min_API	LIMIT	Mappings	API Calls	Total (s)	API (s)	DB (s)
Classic	T&W	false	false	-	634,00	686,00	478,82	446,61	32,21
Min API	T&W	false	true	-	631,33	104,00	1492,96	101,10	1391,87
Pipeline	T&W	true	false	-	633,67	686,00	472,62	426,32	46,30
Both	T&W	true	true	-	639,00	104,00	915,07	104,44	810,63

Table II-3: Results details for the comparison of the classic and streaming methods while using a LIMIT optional bloc. (\*We eliminated the forcing use of streaming method in case of LIMIT bloc to make the tests).

	Query	Pipeline	Min_API	LIMIT	Mappings	API Calls	Total (s)	API (s)	DB (s)
Classic	T&W	false*	false	100	100	685	445,71	428,24	17,47
Pipeline	T&W	true*	false	100	100	108	96,33	70,8	25,53

Table II-4: Results details for the comparison of the queries over a single API instead of both.

Query	Pipeline	Min_API	LIMIT	Mappings	API Calls	Total (s)	API (s)	DB (s)
Weather	false	false	-	52	52	43,96	28,09	15,87
Tweets	false	false	-	633	52	97,97	81,53	16,44