

# **Semantic Querying and Search in Distributed Ontologies**

A thesis submitted to the London Metropolitan University

For the degree of **Doctor of Philosophy**

By

**Sharjeel Aslam**

School of Computing and Digital Media

London Metropolitan University

166-220 Holloway Road, London, N7 8DB

September 2021

# Acknowledgement

Firstly, I start with respectful thanks to Almighty GOD, who helped to carry out this work. There were many difficulties and stresses at times. However, he opened new gates that settled my mind and personality to look back afresh at my studies.

I must thank my supervisors, Dr Vassil Vassilev, Professor Karim Ouazzane and Professor Hassan Kazemian, for taking me on as a PhD student in computing after a data mining degree. Dr Vassi vast experience in this area helped me a lot during my four years of research. This research under his supervision was interesting, and without his support and patience, it would have been not easy to achieve. I am indebted to Professor Karim Ouazzane and Professor Hassan Kazemian for their encouragement, pushing, and support during difficulties and downtime. I also want to thank the Computing and Digital Media department at London Metropolitan University for providing me with the opportunity to complete this research.

I must express very profound gratitude to my family. Each of them has played a significant role by providing me with unfailing support and continuous encouragement throughout my years of studying and through the process of researching and writing this thesis. Not forgetting my friends and colleagues, their support has helped me immensely.

Finally, I would also like to say a massive thank you to my parents, wife and lovely daughter for their patience and incredible support during this journey, and this accomplishment would not have been possible without them. By obtaining this degree, I hope that I can rejoice with them and add some small pleasure to their life and that I can put a little smile on their faces.

# Abstract

We have observed in recent years a continuous growth in the quantity of RDF data accessible on the web. This evolution is primarily based on increasing data on the web by different sectors such as governments, life science researchers, or academic institutes. RDF data creation is mainly developed by replacing existing data resources with RDF, changing relational databases into RDF. These RDF data are usually called qualified linked data URIs and endpoints of SPARQL. Continuous development that we are experiencing in SPARQL endpoints requires accessing sets of distributed RDF data repositories is getting popularity. This research has offered an extensive analysis of accessing RDF data across distributed ontologies. The existing approaches lack a broad mix of RDF indexing and retrieving of distributed RDF data in one package. In addition, the efficiency of the current methods is not so dynamic and mainly depend on manual fixed strategies for accessing RDF data from a distributed environment. The literature review has acknowledged the need for a robust, reliable, dynamic, and comprehensive accessing mechanism for distributed RDF data using RDF indexing. This thesis presents the conceptual framework that demonstrates the SPARQL query execution process, which accesses the data within distributed RDF sets across a stored index. This thesis introduces the semantic algebra involved in the conversion of traditional SPARQL query language into different phases. The proposed framework elaborates the concepts included in selecting, projection, joins, specialisation and generalisation operators. These operators are usually in assistance during the process of processing and converting a SPARQL query. This thesis introduces the algorithms behind the proposed conceptual framework, which convert the main SPARQL query into sub-queries, sending each subquery to the required distributed repository to fetch the data and merging the sub queries results.

This research demonstrates the testing of the proposed framework using the unit and functional testing strategies. The author developed and utilised the Museum ontology to test and evaluate the developed system. It demonstrates all how the complete developed and processed system works. Different tests have been performed in this thesis, like the algebraic operator's test (e.g., select, join, outer join, generalisation, and specialisation operators test) and test the proposed algorithm. After comprehensive testing, it shows that all developed system units worked as expected, and no errors found during the testing of all phases of the tested framework. Finally, the thesis presents implemented framework's performance and accuracy by comparing it to other similar systems. Evaluation of the implemented system demonstrated that the proposed framework could handle distributed SPARQL queries very effectively. The author selected FedX, ANAPSID and ADERIS existing frameworks to compare with developed system and described the results in a graphical format to illustrate the performance and accuracy of all systems.

# List of Publication(s)

- *Parallel Querying of Distributed Ontologies with Shared Vocabulary*. **Sharjeel Aslam, Vassil Vassilev, Karim Ouazzane**. ICMSO 2019: International Conference on Metadata, Semantics and Ontologies

## Best Paper Award



# List of Abbreviations

<b>HTML</b>	Hypertext Mark-up Language
<b>OWL-S</b>	Web Ontology Language for Services
<b>RDF</b>	Resource Description Framework
<b>RDFa</b>	RDF in attributes
<b>RDFS</b>	RDF Schema language
<b>RIF</b>	Rule Interchange Format
<b>SPARQL</b>	SPARQL Protocol and RDF Query Language
<b>SWRL</b>	Semantic Web Rule Language
<b>URI</b>	Universal Resource Identifier
<b>URL</b>	Universal Resource Locator
<b>XML</b>	Extensible Mark-up Language
<b>OWL</b>	Web Ontology Language

# Table of Contents

<b>1.</b>	<b>Introduction.....</b>	<b>13</b>
1.1	Aim and Objectives of the Research .....	17
1.2	Hypothesis .....	18
1.3	Assumptions.....	19
1.4	Research Contributions .....	19
1.5	Limitations .....	20
1.6	Structure of Thesis.....	21
1.7	Chapter Summary.....	23
<b>2.</b>	<b>Background and Related work .....</b>	<b>24</b>
2.1	Introduction .....	24
2.2	Semantic Web.....	25
2.2.1	RDF.....	25
2.2.2	Ontology Web Language.....	31
2.2.3	SPARQL.....	33
2.3	Distributed Data Integration:.....	45
2.4	Federated Database Management Systems.....	49
2.5	Optimised query plans for Query Processing Systems .....	52
2.6	Query Execution Techniques .....	55
2.7	Query Federation Systems .....	59
2.8	Adaptive Query Operators .....	64
	Distributed Query Processing Systems.....	66
2.9	Research Gaps and Proposed Research.....	72
2.10	Chapter Summary.....	73
<b>3.</b>	<b>Research Methodology: Design Science Research .....</b>	<b>75</b>
3.1	Introduction .....	75
3.2	Research Paradigm.....	76
3.3	Research Methodology.....	77
3.4	A Design Science Research Process Model.....	78
3.4.1	Awareness of Problem/Objective:.....	79
3.4.2	Suggestion:.....	81

3.4.3	Development:	81
3.4.4	Evaluation:	82
3.4.5	Conclusion:	83
3.5	Chapter Summary:	84
4.	<b>Conceptual Framework of Querying Distributed RDF</b>	85
4.1	Introduction	85
4.2	Conceptual Framework	86
4.3	Semantic Algebra	88
4.3.1	Operators:	89
4.4	Algorithms:	91
4.4.1	SPARQL Query into Algebraic expression:	93
4.4.2	Converting main SPARQL query into subqueries:	94
4.4.3	Execution of SPARQL queries in distributed ontologies:	95
4.4.4	Combining results:	97
4.5	Chapter Summary:	98
5.	<b>Framework Testing</b>	100
5.1	Introduction:	100
5.2	Comparison of Unit and Functional Testing	102
5.3	Jena Framework:	104
5.4	Ontology Development Methodology:	110
5.5	Ontology Justification - Virtual Museum Exhibition	110
5.6	Framework Testing	111
5.7	Test Results Analysis:	161
5.8	Critical analysis:	166
5.9	Chapter Summary:	167
6.	<b>Framework Evaluation</b>	169
6.1	Introduction	169
6.2	Performance:	170
6.3	Results:	173
6.4	Chapter Summary:	187
7.	<b>Conclusion and Future work</b>	188
7.1	Summary of the thesis:	188



7.2 Originality and Contribution.....	192
7.3 Limitations and Future Recommendations .....	194
References.....	196
Appendix A: Museum Ontology.....	204
Appendix B: Ontology indexing code .....	226
Appendix C - Query conversion code.....	236
Appendix D – Query integration code .....	241
Appendix E – Setup and Testing Screenshots.....	244

# List of Tables

Table 2.1 - SPARQL.....	35
Table 2.2 - SPARQL 2 .....	36
Table 2.3: Relevant systems characteristics.....	72
Table 3.1 - Design Science Research Process Model .....	79
Table 4.1: Algorithm 1 - SPARQL query into Algebraic expression .....	94
Table 4.2: Algorithm 2 SPARQL query into Algebraic expression.....	95
Table 4.3: Algorithm 3 Execution of SPARQL queries in distributed Ontologies .....	96
Table 4.4: Algorithm 4. Combining results .....	98
Table 5.1 - SPARQL query.....	113
Table 5.2:Algebraic notions .....	113
Table 5.3: Cache.....	114
Table 5.4: Identifying sources .....	114
Table 5.5: subqueries .....	115
Table 5.6: case 2 SPARQL query .....	116
Table 5.7: case 2 algebraic notation.....	116
Table 5.8: case 2 cache.....	116
Table 5.9: case 2 identifying resources.....	117
Table 5.10: case 2 sub-queries and merging results.....	118
Table 5.11: case 3 SPARQL query.....	119
Table 5.12: case 3 algebraic notation.....	119
Table 5.13: case 3 cache .....	119
Table 5.14: case 3 identifying sources .....	120
Table 5.15: case 3 subqueries .....	121
Table 5.16: case 4 SPARQL query.....	121
Table 5.17: case 4 algebraic notation.....	122
Table 5.18: case 4 cache .....	122
Table 5.19: case 4 identifying sources .....	123
Table 5.20: case 4 subqueries .....	124
Table 5.21: case 5 SPARQL query.....	124
Table 5.22: case 5 algebraic notation.....	125
Table 5.23: case 5 cache .....	125
Table 5.24: case 5 identifying sources .....	126
Table 5.25: case 5 subqueries .....	127
Table 5.26: case 6 SPARQL query.....	128
Table 5.27: case 5 algebraic notation.....	128
Table 5.28: case 6 cache .....	128
Table 5.29: case 6 identifying sources .....	129
Table 5.30: case 6 subqueries .....	130
Table 5.31: case 7 SPARQL query.....	130
Table 5.32: case 7 algebraic notation.....	131

Table 5.33: case 7 cache .....	131
Table 5.34: case 7 identifying sources .....	132
Table 5.35: case 7 subqueries .....	133
Table 5.36: case 8 SPARQL query.....	134
Table 5.37: case 8 algebraic notation.....	134
Table 5.38: case 8 cache .....	135
Table 5.39: case 8 identifying sources .....	136
Table 5.40: case 8 subqueries .....	137
Table 5.41: case 9 SPARQL query.....	138
Table 5.42: case 9 algebraic notation.....	139
Table 5.43: case 9 cache .....	140
Table 5.44: case 9 identifying sources .....	141
Table 5.45: case 9 subqueries .....	142
Table 5.46 - Testing table.....	165
Table 6.1 - Details of endpoints.....	171
Table 6.2 - Features of Participated Systems .....	172
Table 6.3 - Patterns of Queries.....	172

# List of Figures

Figure 2.1 - RDF Triple .....	26
Figure 2.2 - Different RDF triples .....	27
Figure 2.3 - Blank Node 1 .....	28
Figure 2.4 - LOD cloud (Sakellariou, 2019) .....	44
Figure 4.1 - Proposed Framework.....	88
Figure 5.1 - Apache Jena framework9 (Jani and Dr. V.M. Chavda, 2011) .....	105
Figure 5.2 – case 10 - output of the query.....	145
Figure 5.3 - case 11 - Select operator query result.....	146
Figure 5.4 - caee 12 - Join query result .....	149
Figure 5.5 - case 12 - Outer join query result.....	152
Figure 5.6 - Generalisation output.....	159
Figure 6.1 - Query 1 validation results.....	174
Figure 6.2 - Query 2 validation results.....	176
Figure 6.3 - Query 3 validation results.....	177
Figure 6.4 - Query 4 validation results.....	179
Figure 6.5 - Query 5 validation results.....	180
Figure 6.6 - Query 6 validation results.....	181
Figure 6.7 - Query 7 validation results.....	183
Figure 6.8 - Query 8 validation results.....	185
Figure 6.9 -Query 9 validation results.....	187

# Chapter 1

## Introduction

At the start of this chapter, the researcher provides the aim and objectives of the research. It is critical to establish that the study aims to come up with an improved structure. The author highlights the hypothesis and assumptions made to arrive at the pre-determined goals of the research. However, the problems encountered in developing a better framework also needs to be documented. Chapter 1 hence spells out the goal and the underlying challenges to help other researchers and academicians understand the study's limitations and findings. The author highlights the contributions to help others to interpret it as intended by the new framework. Further, in Chapter 1, the author identified clarity on the research question that is being addressed. The thesis is structured across seven chapters, and Chapter 1 provides an insight into what each chapter addresses. The author also summarises Chapter 1 before proceeding to subsequent chapters to take readers along his research journey. The accessing of data from RDF indexes across various ontologies is one of the biggest concerns in this field of semantic querying (Fazzinga and Lukasiewicz, 2010). Years of research and study have brought several techniques and methods that have been implemented to resolve this problem. Chapter 1 of this research on semantic querying puts forth the motivations behind this research that it aims to gratify. It also elaborates on the research question, problems, and the contributions involved in doing this research. The last few years have shown a steady increase in quantifiable data accessible and available on the internet through different formats- spreadsheets, HTML tables, and PDF documents, among many others. While accessing data can seem as simple as the click of a button, the sub-processes underlying this

process suggest otherwise. A popular model or format of data accessibility is a framework that acts as the cornerstone for the Semantic Web, known as the Resource Description Framework or RDF. It is a set of recommendations proposed by the W3C(World Wide Web Consortium). The RDF, thus, is a primary concept that lays down the groundwork for our thesis. RDF data is obtainable through the concept of an HTTP protocol- which can be implemented through RESTful services that accept and interpret queries arranged in a query language called SPARQL. Note that the queries posed must themselves be under a prescribed SPARQL protocol that the W3C recommends.

The SPARQL code manifests the required information in the format of endpoints. Endpoints are resources that not only communicate with a network but also back up data. During interlinkage, these endpoints are contained within non-exhaustive lists. The lists are compiled to secure such endpoints, but the reader may find that it is not uncommon to find outdated and not maintained lists. These include lists like the CKAN1, The Data Hub, the W3C, and many more. As mentioned, the RDF entails many sets of data within its structure. These data sets are linked amongst themselves. It can be viewed in the Linked Open Data diagram(LOD). The LOD represents a distinctive, figurative expression of how complicated queries are formed by the navigation of individual data across distributed sets to combine with other data. It is not a far reach to define the LOD as a massive collection of interlinked data sets. Records show that the LOD diagram reported listings of over 200 data sets by September 2015. These data sets were further individually linked to some of their counterparts and shared vocabularies with others. An elaborate expression shows that data sets have as many as 25,200,042,902 triples in addition to the 437,205,908 connections they have made over time. This estimate is not inclusive of the 395,499,690 connections made to them by other data sets. The connections to and from a group are regarded separately as each association has its value. It allows for the federation of queries through the properties of varying sets of data. The specific nature of these

queries, in turn, encourage the return of complete sets of results. Unfortunately, the LOD only serves as a diagrammatic representation of the process and doesn't guarantee the practicality of its methods. Certain predominant SPARQL conditions prove that semantic querying is not as easy as the expectations crafted from a LOD. A significant limitation threatens the application of SPARQL 1.0 upon data sets. How can one define and execute a complex query on distributed data sets when the query is only stood up against a single SPARQL endpoint that restricts the information that can and should be returned to the query? Alternate solutions to this limitation have been produced wherein such queries for distributed RDFs have been federated through language extensions and other protocols. Another limitation that blocks the smooth advancement of this study is the lack of access to add extensions that serve heterogeneous data access purposes. Instead, we are forced to succumb to the use of federation extensions included in the existing working drafts of the SPARQL 1.1. The federation extension in use can be expressed through two separate operators: `SERVICE` and `BINDINGS`, written in a query language. One can specify with ease a SPARQL query endpoint within another SPARQL query through these distributed queries.

This SPARQL query endpoint can record and recall information about the timing at which a query was constructed. This characteristic of recognising and consuming knowledge about specific queries enables the `SERVICE` operator to specify the endpoint's IRI, likely facing future execution. On the other hand, a variable can also be compelled to identify the query's execution time after implementing an earlier SPARQL query fragment in the RDF, as mentioned earlier, enabled data catalogues. `BINDINGS` are operators utilised in transferring and inferring results from other sequences to restrict a query within a solution framework. `BINDINGS` are startlingly similar models of a human brain's experiential memory. They use results from earlier implementations of other semantic queries and adopt restrictions similarly placed within the user interface at the time. However, the issue is soon fixed by converting the

inflicted limitations into SPARQL queries. By adapting to such contextual processes, the query language and optimisation semantics assume significant roles in data extraction by distributing queries and processing them across different streams. Querying distributed data sets is not a technological miracle, or even close to one- its arduous nature supports the statement. The already complex process becomes more challenging as problems come and go while posing queries. However, limitations are unavoidable, and a system must be designed to act accordingly and deal respectfully with challenges. For instance, network latency problems and server availability issues seem to be reoccurring in the system. It does not help the case of remotely placed data, which can vary based on the nuances of servers and consequently affect the quantum of data received for a given query. It has been found that a routine function of SPARQL endpoints is to restrict all the data received to calculate 1000 to 5000 results carefully. This technique is a default procedure that respective endpoints are to follow for every query. Due to the minimal nature of the measure of resultant data, it is not necessary that a query plan must be optimised to access such data. The same cannot be applicable in an opposite case where hundreds of thousands of data is allowed to return in response to a query. This can put a user in a disadvantageous position where the process is costly and difficult to transfer over a network.

This thesis, thus, formalises an approach to distributed RDF data sets by dealing with them through federal extension semantics that read queries in SPARQL 1.1. Additionally, we also define the limitations of semantic querying in SPARQL. It is essential to be aware of and list these limitations to be considered, observed and solved when the study requires practical examination over several query evaluators. In such an event where the utilisation of a variable whilst specifying the endpoints of SPARQL is initiated, it can be inferred that implementation would have to pass via entire endpoints of SPARQL over the Internet to pursue a query fragment before a practically unfeasible result is delivered. The author defines service-related



limitations and service security during its execution, thereby ensuring the access of the SERVICE operator is done through a safe and sound process. Additionally, this thesis also leverages the concept of well-designed patterns and indulges in static optimisations that effectively optimise queries about the OPTIONAL operator, which is the most cost-intensive operator in the context of SPARQL. This benefits significant effects for several tuples that can be transferred into federated queries, which gives the implementation an obvious advantage. Notably, other complementary works deploy techniques for adaptive query processing that adhere to a more dynamic approach. This author, thus, introduces a new implementation mechanism for accessing RDF data. It describes and distinguishes two kinds of repositories of RDF data: RDF index and remote RDF data repositories. The present RDF index delivers RDF-based permission to read XML files, text files, relational databases, and thesauri. The author focuses on both kinds of repositories to gain access to data through a faster, more straightforward approach.

## **1.1 Aim and Objectives of the Research**

This section describes the aim and objectives of this research. This research aims to develop a framework that enables one to access a distributed RDF (Resource Description Framework) and the test environment to validate the framework. The author builds the test environment to measure the performance and accuracy of a developed framework that processes the distributed SPARQL queries.

The aims objectives can be summarised as follows:

- To investigate the current state of research in distributed RDF and identify the main problems, existing approaches, and available methods for accomplishing distributed RDF accessing mechanisms with improved performance. See [chapter 2](#) for further information.

- To develop an indexing mechanism to store the RDF repositories. See [chapter 4](#) for further information.
- To develop a mechanism to convert the main SPARQ query into subqueries that can be executed in a distributed RDF environment. See [chapter 4](#) for further information.
- To develop a test environment to check the accuracy and performance of the developed framework. See [chapter 5](#) for further information.
- To evaluate the test results and compare the proposed framework with existing approaches. See [chapter 6](#) for further information.

Thus, the research objectives can enforce an original contribution through which a user can index and compile RDF data from various sources for analysis. This RDF indexing attains placement in an advanced and reliable framework that uses its reach to retrieve and combine results from RDF resources that stretch across different data sets. In turn, these results are thoroughly evaluated and utilised to compare the proposed framework with an existing framework that determines the success of this thesis.

## 1.2 Hypothesis

By taking into consideration all the factors at stake in congruency with this research, the author strives to prove the following hypothesis through the course of this research:

We are revealing the semantic dependencies within the components of the SPARQL queries.

We can formulate a semantic algebra that can be used to translate the queries into a set of subqueries to be executed locally. After aggregating their results, we can obtain a semantically equivalent response to the original query.

### 1.3 Assumptions

Research assumptions are made to achieve the desired objectives. The assumptions involved in examining our hypothesis are listed below.

- One of the distinctive qualities of current technology is that ontologies are perceived to be monolithic by inference engines while they are distributed. It is resolved by adopting a common practice that develops a unified global ontology.
- Storing, organisation and maintenance of the ontologies do not account for the domain knowledge, which can be resolved by semantic indexing.
- The current search engines do not account for the semantics of the queries and provide answers that contain irrelevant information.

### 1.4 Research Contributions

The due process of this research aimed to provide a solution that enables the accessing of distributed RDF data. This process is followed by combining the results attained to test the validity of the research. This thesis contribution can be summarised as follows:

- Design and implementation of an efficient framework using indexing technique for querying ontologies.
- Formal Specification of a semantic algebra of the ontological queries.
- The algorithm for translating the global SPARQL queries into algebraic expressions.
- The algorithm for splitting the global SPARQL queries into a set of independent subqueries that can be executed locally by translating them into expressions of semantic algebra.
- The algorithm for aggregating the results of the execution of the subqueries.

The process was refined by addressing the need to aggregate all relevant information from various RDF sources instead of throwing up just one result. It was made possible by breaking up the main SPARQL query into sub-queries –the individual answers produced a comprehensive response. The basic RDF pattern of <Subject, Object, Predicate> triple model was employed, which illustrates that Subject S has property P, which holds O value. While Subject and Predicate are described as identical resource indicators (URIs), the object is literal. This simple semantic triple helped to optimise the RDF data and create indexing for all participant RDF data sets instead of indexing in the memory. A step-by-step process was adopted. Multiple algorithms were developed to translate the SPARQL query into an algebraic expression, convert the main SPARQL query into subqueries, and carry out SPARQL queries in distributed ontologies. Finally, the author formulated an algorithm to combine the subqueries results. Thus, triples and variables are stored in the cache and identified by the system to carry out the queries, which is more efficient than finding data each time from the source. Two new operators, Generalisation and Specialisation, were proposed to access RDF data. This suggestion contributed by diversifying the methods of access. More precisely, it helps to fetch parent and child nodes. In conclusion, the distributed ontology system allows dynamic indexing, sourcing data from distributed RDF sets, identifying resources from cache, merging, specialisation, generalisation, fetching vertical and horizontal search results. All these features are not present together in other systems.

## **1.5 Limitations**

This research has contributed by proposing and developing a framework for accessing data from different RDF resources across several indexes. However, the author would like to mention that the proposed and developed framework works very well in homogeneous environments where the same ontology's structure is used across all sites. However, the same framework cannot be applied to the heterogeneous environment where different ontology

structures are used. Therefore, there is scope for more research on how to go about indexing data sets across different domains. Applying the same proposed framework to heterogeneous environments did not produce good results as the developed system works best only when the ontology structure is the same in all sites. Perhaps the answer lies in deriving data from different structures, like XML document object structure, relational structure. We relied mainly on the Object-Oriented Model. We have taken the first step in fetching similar(homogeneous) domain data, indexing them on local or remote servers, to be fetched intelligently in response to a single query. The subsequent real challenge would be to retrieve all the participant data from cross domains(heterogeneous) and index them locally and update this stored data dynamically as and when it changes at the source. e.g. writing an algorithm to make a dynamic link between a data source and indexed data. It is a general limitation as such a heterogeneous environment is not a part of this thesis. However, there is a need for the development of different mapping algorithms that work in heterogeneous environments.

## **1.6 Structure of Thesis**

As mentioned in previous sections, the chapter introduces the research motivation and specifies both the research problem and the scope. The entire thesis has been organised in the following manner:

- [Chapter 2](#) (Background and Literature Review): This chapter gives the reader an introduction to the semantic web and an overview of its architecture. Furthermore, it discusses the processes involved in accessing data from RDF data. This chapter also discusses existing RDF data accessing frameworks. The chapter concludes with an overview of existing approaches that help in accessing the distributed RDF ontology

- [Chapter 3](#) (Research Methodology): This chapter elucidates the research methodology used in this thesis. It also discusses and justifies the different stages of the thesis that lead to its conclusion.
- [Chapter 4](#) (Conceptual Framework): Chapter 4 introduces a framework that indexes the RDF data into the central repository. This chapter discusses how any SPARQL query can be transformed into its representative algebraic expression and divided into directional sub-queries. Furthermore, it proposes the semantic algebra that forms a significant part of the research and provides details for all the framework's algorithms.
- [Chapter 5](#) (Framework testing): This chapter presents the implementation and testing of the proposed framework. It holds and supplies all information about a case study applied for comparison: Museum, which demonstrates all the stages of the proposed framework. The chapter includes the testing implementation and details about how converting SPARQL query into sub-queries can catalyse fetching and combining results. It discusses the testing strategy used in this thesis to test the given developed system. It demonstrates all how the complete developed and processed system works.
- [Chapter 6](#) (Evaluation): This chapter elaborates on the evaluation of the developed system. Furthermore, the presented developed system is also compared with other similar techniques to show the accuracy and performance of the developed system that the research suggests.
- [Chapter 7](#) (Conclusion): The final chapter is involved in reflecting on the research developed in this thesis. It discusses and recalls the aims and objectives identified in the first chapter and considers whether they have been achieved or not. It concludes the study with a discussion about the limitations incurred in the system and counters them with recommendations for future use.

## **1.7 Chapter Summary**

Chapter 1 ends on a note of anticipation directed towards the rest of the research. This chapter discussed and evaluated the motivations behind the research and the objectives to be achieved throughout the thesis. It has also provided a perspective on the limitations that have untimely effects on the study and how appropriate solutions are in order. It has created, for the reader, a sense of the study by setting specific standards and expectations that is to be met by the given criteria. It has laid down the basic outline of how its author has carried out this thesis. The second chapter follows these ideals by providing a discussion about the existing accessible RDF frameworks. It creates a background for the study by specifying existing works and contributing to this research architecture.

# Chapter 2

## Background and Related work

### 2.1 Introduction

In this chapter, the author presents the existing cutting-edge in querying distributed RDF information repositories. Besides, we offer an analysis of existing techniques, tools and systems for accessing distributed RDF and non-RDF data, highlighting their main characteristics. Lastly, we evaluate the current work for querying distributed RDF information sources and incorporating them. We evaluate the approaches and strategies employed in these approaches. This chapter provides an authentic explanation of the futuristic approach employed to query repositories of RDF data within this thesis. At the same time, it takes a step back from conventional viewpoints, tools, techniques and systems that have previously contributed to the accessing of distributed RDF data and instead tests new theories that may bring in results in their more advanced form. Apart from defining how this thesis deviates from current approaches towards data, this chapter helps the reader to understand the existing computational field better by reviewing the extensive research that has already been done in the area of RDF data source integration.

This chapter explains some details on the Semantic Web and a brief overview of the nuances of its concepts before moving onto RDF, which constitutes the wide world of the Semantic Web. This chapter provides an overview of the technical background and a detailed literature review. It specifically talks about the types and approaches of data integration. Distributed Query Processing System generates optimised query plans for Distributed Query Processing (ZHANG and XU, 2009). The chapter touches upon and explicates other Query Execution techniques before moving onto the investigation of a Query Federation system of data



processing and introspect on its various archetypes and then briefly discusses Adaptive Query Operators. Subsequently, the chapter delves deep into Ontology -Based Data Integration and Query Processing Systems, such as ANAPSID, ADAERIS, SYMMETRIC INDEX HASH JOIN, SPLENDID, SemWIQ, DARQ. After briefly detailing the challenges and limitations of this study, the chapter then summarises what has been discussed so far.

## **2.2 Semantic Web**

This thesis is a result of the study of one too many complex structures of the Semantic Web. The Semantic Web is a place on the internet that is structured and tagged in a readable way by computers (Arul and Prakash, 2020). It is essential that we understand the core concepts of the Semantic Web, as they contribute heavily to our search. The following sections examine such concepts as the Web Ontology Language, RDF Schema, RDF, RDF Query Language, and the SPARQL.

### **2.2.1 RDF**

The Resource Description Framework, better known as the RDF, is an elementary data model that constitutes the extensive and vast world of the Semantic Web. RDF is a method of decomposing knowledge into small parts, with some guidelines about the semantics of those parts. The motive is to express any fact in a structured way. Previously, RDF was used for representing metadata, i.e. data about data. Now, it has evolved and is used for representing two things. RDF represents information about things in the real world (like people, places, concepts) and relationships. Metadata represented by RDF can also act as background information through which the authenticity of the data can be verified. The RDF is usually expressed through URIs or the Uniform Resource Identifiers (these are usually portrayed through link formats like 'HTTP.' or 'mailto'). URIs help a framework by extending any Internet link into its deeper roots to identify its ends (Shadbolt, Berners-Lee and Hall, 2006).

RDF is the elementary Semantic Web data model. RDF uses URIs to extend the Internet's link structure to identify two ends, typically known as a triple shown in Figure 2.1. The model facilitates the exposing, mixing and sharing of structured as well as semi-structured data. Notably, this information is modelled within the RDF.



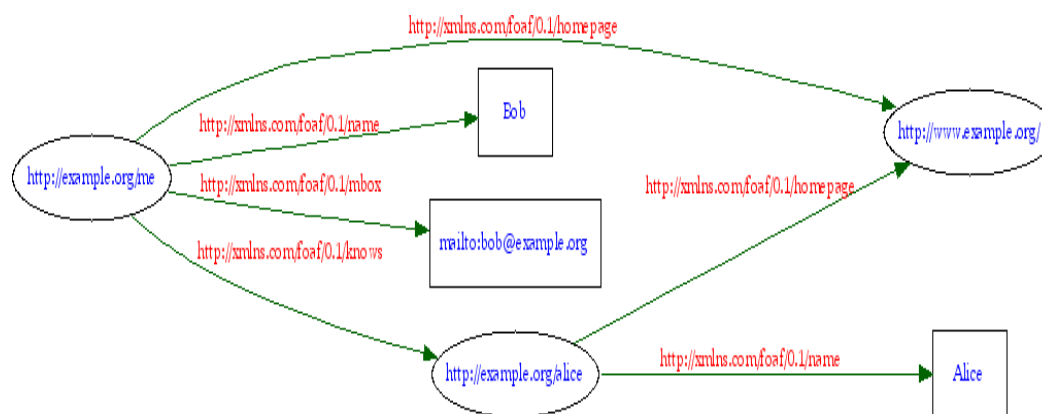
**Figure 2.1 - RDF Triple**

Due to the scale of the structure of the RDF, it is outlined by predominantly existing RDF specifications that are used to model data in an orderly fashion so that they may be better perceived (Heath, 2010). The existing RDF specification has been categorised into six recommendations from W3C:

- **The RDF Primer** elaborates on the elementary RDF concepts. It elucidates defining vocabularies via the RDF Schema or Vocabulary Description Language.
- **RDF Concepts and Abstract Syntax** specify a syntax abstract premised on RDF that links its specific syntax with previous semantics. In addition, it includes analysis of key concepts, design goals, character normalisation, data typing and handling of URI references
- **XML syntax for RDF** is identified by RDF/XML Syntax Specification based on XML Namespaces, XML Base and the XML Information Set.
- **RDF Semantics** defines semantics as well as corresponding rules systems of RDF and RDF(S).
- **(RDF Schema** explains accurate semantics for the RDF and RDF Schema (RDFS) and corresponding complete inference rules systems.
- **RDF Test Cases** elaborates on the deliverable of Test Cases for RDF concerning Core Working Group.

The following RDF graph features a triple concept (a link existing between its two endpoints in a server) in which the subject, predicate and the object of a destination are accordingly denoted in the format of < s, p, o > (Heath, 2010).

The RDF graph features a triple concept of the subject, predicate and object denoted by < s, p, o >. The example mentioned in Figure 2.2 shows that all three aspects are found in the URI of *foaf:name*, *http://example.org/me* and *Bob*. The following lines resemble a complex graph of RDF graph as per Turtle syntax:

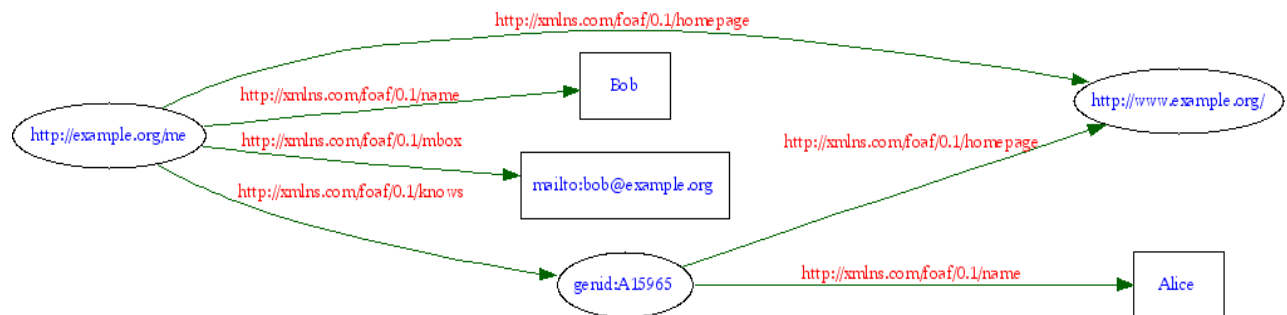


**Figure 2.2 - Different RDF triples**

The data established within the graph above uses a combination of six statements or sentences to define a triple. In statements that are executed in the RDF, only the resources are applied as 'subjects.' In the above URI, the given *http://example.org/me* caters to predicates provided by a person to retrieve specific data. The predicates included in the example are *foaf:homepage*, *foaf:name* and *foaf:mbox*. In turn, these predicates are assigned with specific values, each sent back to a related subject. The values are *http://www.example.org/*, *Peter* and *mailto:peter@example.com*. These are the details specified by the graph in Figure 2. The other elements constituting an RDF graph include nodes and data typed literals (Khozoie, 2012).

**Nodes:** Nodes refer to a point in a diagrammatic network at which two pathways meet or intersect. Within graphical representation, nodes signify a resource, its relation and contribution to the RDF. Consequently, blank nodes are not mentioned in a URI because

these nodes are blank and do not have a related URI to connect and fetch data from. A blank node represents an unknown resource that is only capable of being utilised as an RDF triple in the form of either an object or a subject (Khozoie, 2012).. Figure 2.3 portrays a way in which blank nodes can be used.



**Figure 2.3 - Blank Node 1**

**RDF Literals:** RDF literals are indicated through either typed literals or plain literals. Typed literals feature their own tag of data type, and they are denoted through a string and a data type. Let us exemplify this through an illustration: as per Turtle syntax, `"12.34"^^xsd:float` denotes the actual number, '12.34'. Thus, these literals denote the elements of a value space in a data type. Typed literal tags that define custom data types are can also abstract data types through the language of an XML Schema. On the other hand, Plain literals feature a provided language tag for the corresponding language in which it is written. For example, the following literal: `"It is written in Spanish" @en` is a direct indication that Spanish is the literal language in which the plain text is expressed. Inversely, typed literals can also be interpreted as plain literals that have a substantial XML language tag (Heath, 2010)

Note that one can also serialise the RDF into four distinct formats if required. This is made possible mainly because two of such distinct formats are simply subsets of other supported formats. For instance, the XML/RDF format provides serialisation of XML concerning RDF

data. It is signified as an initial serialisation format, which is the serialisation format that is recommended as compulsory according to the given RDF requirements. In addition, the W3C further elaborates on this distinction through the Notation 3 (N3) format, which acts as an easy-to-read format intended for humans. Notation 3, in turn, factors into N-Triples and Turtles, which, utilizing being N3 subsets, can be applied to RDF triples for their easy description.

We then establish formal notions primarily taken from that can also be found here: <http://www.w3.org/TR/rdf-schema/>. Additionally, it is assumed that the infinite set called V has disintegrated variables from their respective sets, leaving UNBOUND in the form of a reserved symbol that is not incorporated into any of the sets mentioned in the above sections or elsewhere in the document (Heath, 2010).

**RDF Schema:** It is important to remember that RDF Schema refers to the elementary vocabulary of RDF(S), which contains several predefined concepts; these include `rdfs:Property` and `rdfs:Class` that define custom classes and properties.

The list of classes relevant in the RDF(S) includes:

**rdfs:Resource:** This shows the category of different things that the RDF mentions. According to the W3C, everything that is described by RDF- called resources- are a part of this class. The `rdfs:Resource` function is the ruling class. All other listed classes are subsets of this class.

**rdfs:Class:** This class denotes a resource in the form of a class. It dictates other classes.

**rdf:Property:** This class indicates the different properties of a class, including range and a channel domain.

**rdfs:Literal:** This class takes into consideration literal values such as strings and integers. It includes RDF literals that can be typed or plain and property values such as textual strings under its definition.

**rdfs:Datatype:** It refers to the particular class of data types, and sometimes, their subclasses. All instances of rdfs:Datatype are a subsection of rdfs:Literal.

**rdf:XMLLiteral:** It represents the typed literal values' (XML) class.

The list of RDF(S) property includes:

**rdfs:range:** It mentions a data type, or the class of a particular object within a triple, followed by the subject, which portrays a predicate.

**rdfs:domain:** This property mentions the class of a respective subject. In such an instance, the predicate automatically becomes the subsequent component of a given triple.

**rdfs:subPropertyOf:** It shows an instance of rdf:Property, and mentions that all resources that find connections with a property are also connected by and to each other. Thus, it is not just classes but also resource properties that are interlinked to a network.

**rdfs:label:** It signifies an rdf:Property instance that can be utilised for producing the name version, a readable label of a written resource in a language understandable by humans.

**rdfs:subClassOf:** It allows for a clear-cut declaration of class hierarchies. This property may be used to define classes as subclasses of each other.

**rdfs:comment:** This class calls upon an instance of rdf:Property to produce an explanation of resources that humans can read.

Put succinctly, RDF(S) contains a diversified vocabulary that allows for human interaction with the different classes and properties of an RDF(S) element. People are allowed to engage with aspects like properties, inheritance, typing, or classes, thereby providing elementary components that form more complicated linkages between RDF data elements and enable us to understand them (Khozoie, 2012).

### 2.2.2 Ontology Web Language

Ontologies are used to model real-world entities and relations among them in a taxonomic structure. They are nowadays the backbone for Semantic Web applications. Several languages are developed for the formal representation of ontologies. RDF Schema (RDFS) was the first attempt towards developing an ontology language, and it became a W3C recommendation in 2004. RDFS was built upon RDF. It extends the RDF vocabulary with additional classes and properties such as `rdfs:Class` and `rdfs:subClassOf` (Simplerl, 2009). The latest W3C recommendation for ontology languages is the Web Ontology Language (OWL). OWL further extends RDFS by providing additional features such as cardinality constraints, equality, disjoint classes, efficient reasoning support and much more (Heath, 2010). The OWL language has OWL-Lite, OWL-DL and OWL-Full sub-languages. OWL-Lite and OWL-Full are not widely used because the former is too restricted, and the latter does not guarantee efficient reasoning. OWL-DL provides maximum expressibility with a complete and decidable reasoning support

The Ontology Web Language- OWL, in short- is a language that represents knowledge based on a system of formalist and descriptive logic. It utilises more remarkable and more significant expression profiles to elaborate on domain knowledge that is defined outside of RDF Schema support. This articulation also suggests the onset of more formal semantics and a broader vocabulary within the scope of knowledge. Consequently, this feature illustrates cardinalities

concerning properties, curtailments on existential and universal properties/classes, algebraic characteristics, and other valuable information. The OWL system accommodates something called an Open World Assumption, a concept that provided knowledge in any context is always deemed an incomplete measure of existing knowledge. This belief is unlike the Closed World Assumption, which has its morals rooted in the opinion that all knowledge that is not mentioned is false or non-existent, compared to information established in a knowledge base - which is considered valid (Heath, 2010). Take the case of the following function:  $\langle ex: me, rdf: type, foaf: Human \rangle$ , which shows that the concerned individual is a human but does not explicitly identify this human as an engineer or student. It does not attribute any property whatsoever to the human. For this reason, whilst querying the information for all engineers who are engineers, one of the findings should be *ex:me*. On the other hand, no results must show up while querying the same resources within a relational database (because all engineers are human, but the function of *ex:me* does not consider a human to be an engineer).

The OWL does not list distinctive name assumption as one of its features. This is because it employs a unique assumption, which states that different identifiers are required to refer to several entities within the actual, natural world (Siddiqui and Alam, 2011). In other words, *ex:me* cannot be said to be the same as  $\langle http://example.org/bob \rangle$ , as it does not touch on real-world elements. Additionally, it utilises special predicates to assess the resources' equivalence in a specific particular case about reality (Hong, 2016). OWL is also known to deliver results in various language flavours. OWL 1 covers the following variants premised on the axioms and expressiveness of the language that are used within the ontology framework. These include the following factors: OWL DL, OWL Lite and OWL full. The situational difference of OWL 1 from OWL 2 is that the latter applies profiles belonging to other language profiles and make a contribution through various stages of expressiveness, such as:



**OWL EL** encapsulates the power of expression that comes from being utilised by several ontologies; it is a specific subset of OWL 2 that identifies which elementary reasoning challenges regarding the ontology's size can be undertaken during a polynomial time.

**OWL 2 QL** implement conjunctive query responses through traditional methods in relational database systems. It is possible to perform and arouse a complete and sound conjunctive query response as long as a feasible reasoning methodology through LOGSPACE is employed. This methodology usually focuses on the data size as it works and is often referred to as an assertion.

**OWL 2 RL** is aimed at satisfying applications of OWL 2. These applications can trade the language's comprehensive clarity in exchange for efficiency in functioning and RDF(S) applications that need further expressivity. It is possible to incorporate reasoning systems of **OWL 2 RL** using rule-driven reasoning-related engines. The applications ensure that the class-expression satisfaction, ontology's consistency, answering conjunctive queries, and instance checking can be addressed during a polynomial time. OWL 2 is different from its first counterpart. It incorporates new and fresh flavours into the existing language, whose varying profiles are subjected to the setbacks arising from more restrictions than OWL DL. In addition, OWL 2 introduces new features which can be used to simplify complicated statements and make them more feasible (for example, Disjoint Classes, Disjoint Union, Negative Data Property Assertion and Negative Object Property Assertion). Other integrated features constitute constructs that heighten the expressivity factor, support for expanded data, fundamental meta-modelling capabilities, and annotations' expanded capabilities (Siddiqui and Alam, 2011).

### 2.2.3 SPARQL

Since SPARQL became an official W3C recommendation in 2008, it is currently the most widely used semantic query language. A SPARQL query consists of conjunctions and

disjunctions of triple patterns similar to RDF triples. Despite its simplicity, the usability of SPARQL is limited for the end-user (Kurgaev, 2018). First of all, formulating a query requires considerable time and effort, even for the most straightforward query. Secondly, domain knowledge is required, i.e. the exact names of classes and properties need to be known in advance.

SPARQL is a semantic query language whose function is to extract and redefine information stored in the RDF. The SPARQL has been generous with its execution. As it is one of the only languages compatible with the RDF to a large extent, the SPARQL is somewhat of a blessing. This definition implies that the responsibility of SPARQL is huge in magnitude. To ensure the effective implementation of its definition, SPARQL-WG, or the SPARQL Working Group, has been consistently supporting the language (Song, Huang and Sun, 2017). In this section, The author views a SPARQL graph pattern that resonates similarly within the RDF boundaries. Let us pause momentum and refer back to Figure 2 as mentioned above, which gives us the name of a person- Bob. It is unlikely that anyone wants to stop accessing data after gathering the name. As a user enters queries to procure more knowledge about Bob, SPARQL works on the same tangent to supply the user with more information. The manner of the SPARQL mechanism is as follows:

Based on certain SPARQL queries, all the triples with specific subjects are selected, and predicates are determined using the source graph to identify properties. As the query object is somewhat of a free spirit and is not challenged by strict boundaries, it could be attributed to any valid values that the RDF graph assigns to it. Notably, question marks are used to represent SPARQL variables before characterising them with a name (Jagvaral, Lee, Kim and Park, 2015). The syntax of triples utilised by the formats of Notation 3 and Turtle remains unwavering through the execution. There is also a linkage between the variable? Person and three additional resources. These resources continue to serve as existing objects. To give a clear

picture of the SPARQL language, we have illustrated the code in a well-structured and well-defined table as presented below. The given table 1 elaborates on specific findings of a query. As the distribution of these solutions is typical to an unordered multiset, the order of elements is irrelevant. The empty rows, then, indicate the corresponding variable (unbounded). Meanwhile, the solutions can constitute a multiset using three facilitated solution mappings that draw their basis from query variables and inhibit their respective values.

<hr/>
?human
<http://example.org/alan>
<http://example.org/alice>
<http://example.org/carl>
<hr/>

**Table 2.1 - SPARQL**

The graph is thereby restricted to a set of only three items conforming to a pattern. The lowest possible pattern about the characteristic restrictive triple is  $p ? s ? o$ . In turn, this variable determines how the entire graph is to be introduced and manipulated to maximise the information to be gathered for a specific purpose. Note that the outcomes of their respective queries that are reflective and inclusive of all triple patterns belonging to a particular graph are demonstrated in the following manner:

@prefix foaf: <http://xmlns.com/foaf/0.1/>.
---

@prefix ex: <http://example.org/>.
ex:mefoaf:name "Peter".
ex:mefoaf:knowsex:alan .
ex:mefoaf:knowsex:mark .
ex:mefoaf:knowsex:carl .
ex:alicefoaf:name "Mark" .
ex:alicefoaf:knowsex:alice .
ex:alanfoaf:name "Alice".

**Table 2.2 - SPARQL 2**

**Graph Patterns:** In the graphical instance mentioned above, we examined and obtained the URI of resources from only the previous example, which, unfortunately, does not cater to the assumptions and values for humans- and is not rendered as applicable. This makes the use of adding a new triple pattern by selecting the property of *foaf:name*, concerning every *?person* to help in our pursuit of identifying the names of the corresponding resources (Abdelaziz, Harbi, Khayyat and Kalnis, 2017). We can accomplish a similar feat by enlisting the Basic Patterns Of Graph or the BGP, which also deals in distinctive triple patterns. Additionally, the BGP is then depicted as a graph representing a group of RDF triples.

**Matches:** As the inconvenience of the lack of a related *foaf:name* with respect to a single individual within the source graph has been instantiated, there are only a couple of available solutions, based on the earlier instance. In this regard, it is possible to use the OPTIONAL

keyword in case the result is capable of optionally including a person's name but continues to comprise of their URI:

```
{ <http://example.org/me>foaf:knows ?person .
```

```
OPTIONAL { ?person foaf:name ?name. } }
```

Based on optional semantics, we can infer the mappings created by the initial BGP and utilise them in combination with another BGP (Kurgaev, 2018). This means that if there is an inconsistency incurring between both the binding elements, the ones on the left would be streamed as given:

<hr/> ?person	<hr/> ?name
<http://example.org/alan>	"Alan"
<http://example.org/alice>	"Alice"
<http://example.org/carl>	
<hr/>	<hr/>

The SPARQL union denotes a theoretical conjoining of two distinct sets of results. Therefore, the SPARQL is not the same as SQL union, which only adjoins two more SELECT statements. The columns of either side of the union need to be motorised into compatibility in SQL, which is not required in SPARQL. Notably, both BPGs are capable of sharing standard variables throughout fusing BGPs with the union's functioning (Dubinin et al., 2020). They are also capable of having independent variable sets. THUS, the SPARQL union signifies the first and

the second solution mappings of the BGP. The below illustration showcases SPARQL's union when compared with the earlier graph:

<hr/> ?person	<hr/> ?name
<http://example.org/alan>	
<http://example.org/alice>	
<http://example.org/carl>	
<http://example.org/Mark>	"Mark"
<http://example.org/alice>	"Alice"
<http://example.org/carl>	
<http://example.org/me>	"Robert"
<hr/>	<hr/>

The result of the above query showcases the union of two BGP findings. Notably, the first and second BPG choose individuals known by way of *http://example.org/me*, to be inclusive of their names. Finally, SELECT is the sole category of query in action and is functioning through self-reliance. However, take note that it is possible to make use of other categories just as efficiently. SPARQL contains several query forms that enable the creation of distinct types of queries based on a matching graph pattern (Rakhmawati and Fadzilah, 2019). The query elements can be selected from the entire data by using just the SELECT query. Similarly, data is provided about resources that can ensure congruity between graphic patterns utilizing the DESCRIBE query, which shows a clear-cut RDF graph (Dubinin et al., 2020). Subsequently, the CONSTRUCT query returns a graph based on the answers developed by utilising the graph

pattern within the query itself. Ultimately, ASK queries provide finality in returning results by stating either false or true based on its solvability.

The W3C SPARQL Working Group developed an upgrade called SPARQL 1.1 in response to the limitations and inadequacies that outlined the operation of the initial SPARQL language. Undeniably, SPARQL 1.1 is a visible improvement from its predecessor. It has a brand-new reach into elements that its previous version was to exercise. SPARQL 1.1 branches into different components, including subqueries, aggregation operators, other languages and protocols that are to be used in the interpretation of RDF graphs. A total of 11 documents published by the W3C- provide an insight into the additional contemporary features accompanying SPARQL 1.1. It is possible to identify relevant documents in the W3C portals, divided into titles based on their activity jurisdictions (Dubinin et al., 2020). These documents include the likes of Service Description, SPARQL 1.1 Update, Protocol, Entailment Regimes, JSON, Property Paths CSV, Federated Query and the TSV query result. The SPARQL 1.1, thus, is a query language, defined by its resident advancement over SPARQL 1.0., as a novel recovery from the complications existing in the latter. Consequently, this evolution of SPARQL 1.0 can be addressed through the previously redundant functions re-established in the primary documents of SPARQL 1.1. One such popular addition is collectively called aggregation functions. This document stipulates that aggregation functions can count over the columns of results, compute the average of the minimum and maximum values in a unit and solve other problems in a numerical context.

However, the most significant change characterising the new SPARQL 1.1 is possibly the incorporation of subqueries. Subqueries were a previously much-needed trait in qualified query processing, seeing as their presence in SPARQL 1.1 helps classify and clarify information under other queries. A subquery makes it possible to nest the findings of a specific query under the name of another. For instance, consider a blog website with many articles or pieces

scattered all over, with no clear distinction to define them. This can be resolved using a subquery, by which one can identify a recent blog post under a weblog that is based on, let us say, the name of an author. The same works for the many other blog posts included in the log. The feature of 'denial', which was earlier exhibited- through implicitly- in the SPARQL 1.0 language, reappears in both the NOT EXISTS filter and the MINUS keyword of SPARQL 1.1. The NOT EXISTS filter noticeably pertains to negation testing. It implies that things that are attributed to specific bindings have been cemented through the pattern of a particular query, regardless of what pattern is evident towards the end. The NOT EXISTS filter compares two existing patterns and removes matches based on the results. On the other hand, the MINUS keyword takes a relatively upbeat approach. The keyword takes into consideration the fact that a query has determined specific bindings. Based on the given bindings, MINUS accepts and evidences the existence of pattern matches. If nothing in common is found, then no bindings are eliminated (Dubinin et al., 2020). On the other hand, project expressions remain unrestricted. They function through SELECT queries, which help the project expressions emerge. The SELECT queries can go beyond the format of variables to project a SPARQL expression. Thus, apart from being simply expressed, a project expression can expose itself through different personas: a constant literal, a variable, URI, or even an arbitrary expression on constants and variables.

**The SPARQL 1.1 Update** extension works as an updated language for understanding RDF graphs. It derives its roots from syntax in SPARQL 1.0. In simple terms, the update function interacts with a collection of Graphs, which form a Graph Store. The update function can create, update, and remove graphs from a Graph Store in its operations. The update function also showcases features that enable a user to insert new RDF triples within a data set highlighted by a SPARQL endpoint. In turn, the SPARQL endpoint facilitates access to the operations



above. These operations include insertions and deletions of loading an RDF graph, clearing and forming new RDF graphs obtained from the Graph Store's endpoint address, etc.

**SPARQL 1.1 Protocol**, developed by the W3C Working group, defines SPARQL protocols and the corresponding RDF query language. It outlines a process to communicate SPARQL queries to a SPARQL processing service (the RDF query language, in this case), and retrieving the required information through an 'http' format, and links the results back to the client or entity that requested the (Dubinin et al., 2020). It explains how and why the SPARQL language is suitable to perform and execute these processes for accessing data. It is possible to view the SPARQL Protocol in two ways: (1) An abstract feature lacking concrete application and binding over a different system and its protocols, or (2) A HTTP binding specific to an interface.

**SPARQL 1.1 Service Description** is a design for representing information about SPARQL mechanisms. This is entailed in a document that portrays knowledge regarding a method to discover and a vocabulary to describe SPARQL services that can be enlisted through a SPARQL 1.1 RDF Protocol (ZHANG and YANG, 2011). The function of service description serves an important agenda: to make popular the awareness of SPARQL services (Dubinin et al., 2020). The well-stung-out methods and techniques of description enable clients or end-users to gather more information regarding SPARQL services. Such information may include service extension functions or details about data sets.

**SPARQL 1.1 Regimes of Entailment** outline the fundamental entailment structure for SPARQ query language. RDF triples are usually portrayed through graphs. While both the RDF and the OWL have come with strategies to help interpret these graphs to form relations between n the given assertions and additional RDF statements, such graphs can only be computed through the SPARQL mechanism, through entailment regimes. Importantly, SPARQL endpoints can lend certain types of entailment, which includes entailment towards

the RDFS. Thus, whilst putting forth a query at the remote endpoint, it is possible for users to obtain findings reflecting on all the possible RDFS ramifications (ZHANG and YANG, 2011).

**SPARQL 1.1 Graph Store HTTP Protocol** elucidates how an HTTP protocol can be used to organise and manage a set of RDF graphs. This function is more or less similar to the SPARQL 1.1 Update protocol but provides an alternative on the off chance that some clients or users may prefer its interface to that of the Update function, as it is easier to work with (ZHANG and YANG, 2011). This function also puts RDF graphs outside of a graph store in an advantageous position to be maintained under HTTP operations.

**SPARQL 1.1 Federated Query** is put into use on the basis that RDF data is distributed across the web over several SPARQL endpoints. The Federated Query function strives to translate a query among various data sources accordingly. This document elaborates on the semantics and the syntax relating to SPARQL 1.1 Federated Query extensions to circle queries over several SPARQL endpoints. Notice that the keyword - SERVICE - expands SPARQL 1.1 to support queries that merge information distributed all over the Internet.

**SPARQL 1.1 Property Paths** defines property paths that match SPARQL queries without inflicting any change upon the queries. Property paths provide the platform to draw out basic graph patterns briefly. A property path is simply a feasible path between two distinct graph nodes. A trivial case of property paths is represented through an approximate length of 1, representing a triple pattern.

**SPARQL 1.1 Query Results CSV and TSV Formats** refer to a definition of comma-separated values (CSV) and tab different values (TSV). They are simple, easy to use, and perfect for the transmission of tabulated data. This function also entails the usage of these formats to combat SELECT queries with more SPARQL findings.

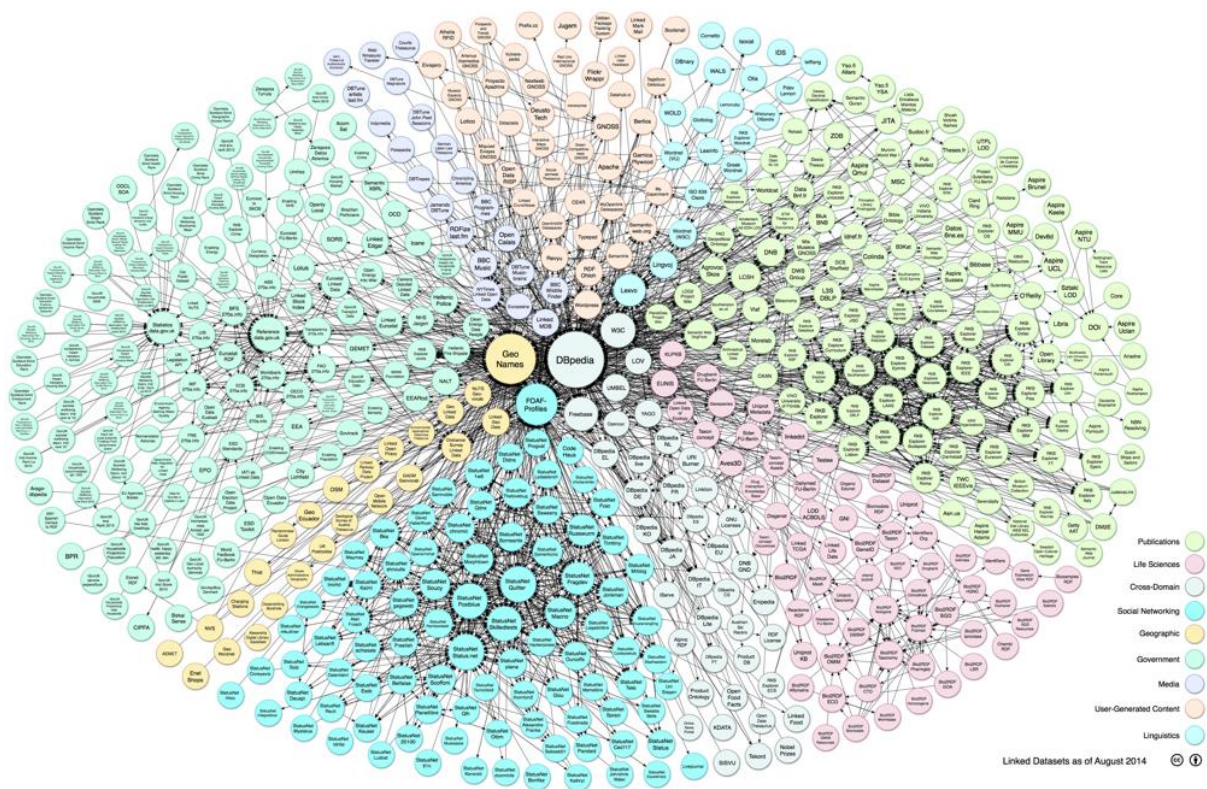
**SPARQL 1.1 Query Results JSON Format** involves a set of recommendations concerning the query, update, and access data. This document describes how ASK and the SELECT functions are to be used to gather results through JSON.

In the vast RDF structure, unknown data sets with no relational history remain unrecognised and non-existent while we are at it (ZHANG and YANG, 2011). This is because the RDF is essentially a chain of data resources that are connected through common properties that prevail amongst themselves. Therefore, the Web of data or the Semantic Web is inclusive of all such kinds of data and considers the process analogous to connecting documents through similar material. Tim Berners-Lee, the World Wide Web inventor, proposed the following principles concerning the Semantic Web.

1. HTTP URIs can be utilised to make identifying things easier.
2. It is necessary to base valuable data on specific standards (SPARQL, RDF) to cater to the possibility that the user may look at the URI.
3. Given information must be followed by extra semantic links that pinpoint other URIs to lead people along a chain of additional information.

Under the first guideline, URIs are required to identify resources and things. By leveraging a URI, one can use it as an identifier to directly access a particular object or reference them through the providence of additional resources (Hammami, Bellaaj and Kacem, 2018). This increases data source credibility. The second rule corresponds to the first one, a supplementary. It claims that HTTP URI's must be promulgated to the user through specific standards so that data sources may remain viable and identifiable (as given in the first rule) (ZHANG and YANG, 2011). Standards form an essential aspect of both the WWW and the Internet of Data. One feasible means of accomplishing the goal of accessibility can be achieved through the utilisation of standards. Meanwhile, the fourth and last rule implies that we can lead a user through portals of unseen information by attaching URI links to provided data.

Figure 2.4 displays the latest condition of the LOD cloud and depicts how possible sets of data on the internet can be retrieved through SPARQL endpoints. The figure given below entertains the idea of distinctions in the LOD cloud by expressing it through its topology. The topology of a LOD cloud is categorised into six distinct groups based on data types. These data types are namely: media data, government data, geographic data, life sciences data and cross-domain data, among others. The entire data cultivate the principles of the Web of Data- propounded by Tim Berners-Lee- into its functioning. As a result, it becomes highly possible to uncover new data content by following the connections established between different data sets. As per the LOD cloud, the DBpedia data set, manifesting at the cloud centre, is one of the most frequently referenced nodes. Interestingly, DBpedia represents the Wikipedia RDF with its numerous links about a massive bundle of data sets.



**Figure 2.4 - LOD cloud (Sakellariou, 2019)**

## 2.3 Distributed Data Integration:

To help propagate a better understanding of this thesis, the author study and elaborate on the existing relevant system. This section discusses the onsets and the pre-sets of data integration- the primary conceptual root of this thesis. Data integration is a process through which data is combined, *integrated* from the various sources from which it is derived. Data integration uses this combination of data and projects them into a unified and comprehensive view to the respective users (Shah, 2016). Data integration provides to be highly useful in the Semantic Web, as it enables the focusing of data into a standard structure that is insightful and informative to a user. Bringing together various data under one construction also makes the entire process of data accessing efficient. The primary concern of data integration is that it has to gain access to several data that is distributed across numerous data sources. In the Semantic Web, gaining access is not as simple as it sounds in a world where all data is locked into a protective and safe framework (data integration in the distributed information systems, 2012). Thus, data integration needs to draw from the federation, indexing, and materialisation approaches to gather objective data. These approaches take form through the following functions:

- Read-only views: A Read-only view, in simple terms, is an integrated and readable (it cannot be manually manipulated) view of multiple databases on a single platform. Read-only views are popularly facilitated by "mediators" in data integration, which are components deployed on a different computer. A mediator works in a unique way by which it gives users a view of data whilst keeping them locked in their respective data sources.

The mediator integrates data through the design of a single schema, also known as a global schema, which becomes a unique entry point for all queries mandated against it. Semantic mappings existing in between the mediator and the corresponding data

sources help execute this process. A mediator can also be deployed by utilising several techniques, including materialised views, virtual data integration, and hybrids born out of the two. It must be noted that these views can possess information from a plethora of sources and several other mediators (Retracted: Semantic Information Integration with Linked Data Mashups Approaches, 2015).

A unique global schema can be manufactured through both the GAV (Global as View) and the LAV (Local as View). The expression of global schema takes residence in data sources through the virtual and material viewpoints inhabited by them. These views are subsequently based on all the data and information gathered from the source before being mapped to a global schema. The following requirement is to transform global schemas into local schemas (facilitated through mapping) to supply a query with data from various sources. Thus, both the GAV and the LAV approaches operate on the same objectives (Retracted: Semantic Information Integration with Linked Data Mashups Approaches, 2015). While the global schema itself does not rely upon its data sources, the links between both components can be established by defining a global schema to its analogous data sources.

- Information sharing between multiple databases: The framework of federated architectures is the opposite of data integration because numerous databases can be introduced within a federation. In addition, every database has the feature of extending its personalised schema as a federation member. This characteristic allows the database to facilitate various data subsets across members of other databases. In most instances, this system of support is lent to virtualised data integration approaches.
- Read-write views in an integrated manner: This function expands a mediator's architecture by attributing it with the ability to update upon choice. Updating, thus, begins to play a crucial role as it allows the processing of new queries into the entire



system, including those regarding concurrency and consistency. Such feature can be achieved by outlining the concerning architecture and specifying it.

- **Arranged multiple databases:** Multiple databases are well resourced by big organisations that require the help of copious amounts of data repositories to represent their tens of thousands of data tables. Not only are numerous databases advantageous in terms of quantifying extensive data, but they also rotate a spectrum of functions that are embedded in a firm. In this regard, workflows assume importance in mathematical models of life sciences and data-intensive applications. Notably, workflow paradigms can explain the contact established between databases from a semantic standpoint.

As previously illustrated, several approaches can be used to determine and classify the flow of data integration. This section aims to inform and elaborate upon the two major approaches to integration. One of them draws a hard line between materialised and virtualised data integration, while the other approach differentiates between declarative and procedural data integration (Zangenehpour, Ali Seyyedi and Mohsenzadeh, 2012).

**Material/Virtualised Data Integration:** Material and virtual views are two approaches that effectively support data integration. Regarding material views, the system becomes an interface between the source and the user trying to access it. This structure finds appliance in distributed databases, multi-databases, as well as open systems. Inversely, extracting a query response for virtual integration enlists query rewriting techniques and provision of access directly into a source whilst query evaluation (Mishra and Mishra, 2017). In addition to being a much-complicated process compared to the material view, a virtual integration is typically bound to cost more. Meanwhile, the system also maintains a replicated version of an access point in the second instance to promote its objective of data warehousing and data system re-engineering. However, it must be remembered that maintaining materialised views can also be a cost-

intensive process concerning keeping the views updated (Mishra and Mishra, 2017). There exist several techniques for facilitating materialisation views:

- **Extract/Transform/Load** withdraws information from at least one data source, transforms it, and finally saves the finding in a separate source of data.
- **Replication** creates and maintains a distinct copy of all logfiles; it generally does that on a differential basis.
- **Caching** tracks, presents, and stores the query findings for future use.
- **Search** provides a solution and creates one specific index for data to be eventually integrated. This approach is commonly adopted in unstructured data; it is also reflective of partial materialisation, considering that the index generally defines relevant documents dynamically requested by the user.

**Procedural/Declarative Data Integration:** This is a classified data integration approach that presents a holistic perspective through the combination of declarative and procedural data. Under the procedural approach, data integration takes place haphazardly to cater to several requirements predated by predefined information (Tomaszuk and Hyland-Wood, 2020). The idea behind the procedural approach is to create feasible software modules that can access data sources in compliance with the pre-set information demands. In contrast, the declarative approach strives to model the data using a feasible language whose objective is to obtain the answers to a query. The system fulfils this requirement by utilising complementary materialised views, which establishes a feasible unified model that propagates a given query to the worldwide information system (Yang, Guo and Wei, 2017). Notably, the current declarative approach accommodates the global nature of data sources and considers it a reusable system element for consecutive data integration.



## 2.4 Federated Database Management Systems

This section has grips on explaining the query federation mechanisms underlying a declarative and virtualised approach. Federated database management systems, also known as FDMBS, refer to a set of database systems that cooperate on a heterogeneous or homogenous normative basis to diversify the data integration framework. It may be worth remembering that the system lacks any sense of centralised control within it due to its replacement through DBS components, which, in their wake, display a significant level of control over data access mechanisms (Hitzler and Janowicz, 2010). The following points entail an examination of specific FDMS attributes that overlook the management of databases:

- **Distribution:** It is possible to distribute data over several databases. These given databases, in their miscellany, are capable of being stored on one or multiple computers. They can be situated at the exact location or in geographically different directions whilst being linked through a communication system. This distribution feature allows for connectivity between databases.
- **Autonomy:** Only those who can extend significant control over a particular database can grant others access to data. Additionally, they can also regain control and retrieve the right of a user to access if they please. The authors of a database can exercise different types of control over their documents (Hitzler and Janowicz, 2010). Different levels of autonomy are attributed to different levels of control and can be specified as follows:
  - ✓ **Communication autonomy:** This refers to a component of DBMS's ability to discern whether or not to engage with other components and when to do it.
  - ✓ **Design autonomy:** The design autonomy dictates the DBS' ability in choosing an independent design, regardless of the cause. Usually, DPS components' designs are attributed to their heterogeneous structures.

- ✓ Association autonomy: This component enables a DBS to determine whether its resources and functionalities can be distributed among other members of the federation document. By the association autonomy function, one can also determine the extent of the sharing, i.e., limits perceiving a shareable about of data can be placed on specific quantities of data.
- ✓ Execution autonomy: The execution autonomy component enables a DBMS to carry out its functions through local operations without any intervening extraneous operations that may disrupt the function execution in hazardous ways (FDBMSs or DBMSs submit). By adopting execution autonomy, a DBMS component can also determine the order in which external operations are to be executed.

When focusing on schemas, five levels for managing data integration exist in the system:

- Component schema: It is extracted by translating local schemas to a standard or canonical data model (CDM).
- Local schema: It denotes the conceptual schema of DBS.
- Federated schema: This is where a multitude of export schemas get integrated. In addition, the federated schema is inclusive of information relating to the distribution of data generated whilst carrying out the integration of export schemas
- Export schema: An export schema is essentially a component schema's subset. The endeavour of describing export schemas would be to begin control as well as management of related autonomy.
- External Schema: This is specifically intended for an application, a user, or a group of applications/users.

FDMS Architectures: FDBMS architectures can be used to describe the structure of a database (Appreciation to distributed and parallel databases reviewers, 2018). FDBMS architectures are

likely to be structured according to centralized, decentralized, or hierarchical mechanisms. In addition, the architecture mechanisms can be tightly or loosely couples:

- **Tightly coupled mechanisms** are famous for their promotion of interdependent and closely-knit architecture. In these mechanisms, export schemas get established by implementing a negotiation between the federation DBA and its component DBA. Notably, the latter component, DBA, influences any elements implemented into the export schemas. The typical role played by the DBA federation in this process is granting permits to decipher the regulatory component schemas and read into the type of information constituted in these components (Babu, 2012). Once a specific type of information is confirmed, the federation DBA begins negotiations, leading to export schemas for forging connections. The external schemas validate their part in the equation by carrying out an arrangement between federation users (a group of federation users) and the federation DBA. However, the federation DBA still has authority over any information that may be delegated to external schemas and can determine the access and permissions about such information.
- In an FDBS that is represented by a **loosely coupled mechanism**, independence is prioritized. Components of such a design are usually lean, single, and micromanaged by individual entities to revoke maximum responsiveness. In loosely coupled architecture, every federation user has the power to administrate their federated schema. This can be empowering in a way that a user can determine one's benefits. Consequently, a federal user can evaluate the available export schemas and ascertain the type of data they want to initially access. After that, a federation's user extends their preferences over the federated schema by importing objects from an export schema through an application program (Babu, 2012). In addition, a schema can be employed, by use of which one can define a language query in multi-database referring

to a schema's export. The user can access change within an export federated schema and assess its semantic heterogeneity. Additional information can be referenced from either DBMS dictionaries or the "DBMS dictionary (federated)". This federated schema is initially licensed under the behest of a federal user's owner but can be removed and referenced anytime by the concerned federation user.

Critical aspects outlining the functioning and processes of FDBMS architecture include:

- Schemas being inclusive of the locally stored data descriptions.
- Mapping and defining database schemas as functions that link objects.
- Processors filter, transform, manipulate, access and build data. These processors could easily connect through their wide range of functionalities and generate new architectures to manage data.

## **2.5 Optimised query plans for Query Processing Systems**

This section undertakes a review of some current techniques to generate optimised query plans for Distributed Query Processing (DQP), such as deterministic algorithms and randomised algorithms. The dynamic programming optimisation algorithm recursively divides a problem into more straightforward subproblems. It can be implemented when the sub problems are not independent of each other (Development of a CUBRID-Based Distributed Parallel Query Processing System, 2017). After that, such sub problems are resolved on just one occasion, saving all solutions in a table before combining them to reach the overall solution. Characterise the overall structure of the best possible solution.

- Recursively define its optimal value.
- Calculate its value in a bottoms-up manner.
- Devise an optimal solution based on the computed information.

In the case of deterministic algorithms, all algorithms are known to develop a solution incrementally. It may be noted that such algorithms are either applied heuristically or through an exhaustive search. A dynamic programming algorithm is presented for distributed query processing and heuristics to adopt the most feasible query plan. Under a dynamic programming algorithm, it is possible to discard a plan if an alternative plan performs the same/additional work by incurring a lower cost (Kaneko and Chishiro, 2018). The positive attribute of dynamic programming is its ability to produce the best possible plans based on costs. In the event, these costs are sufficiently accurate. The algorithm can then identify the best possible query plan. Meanwhile, the shortcoming of dynamic programming is that it is ridden with exponential time/space complexities. Therefore, the complexity of dynamic programming could be prohibitive, particularly in a distributed environment.

The Deterministic approach also exploits other algorithms that employ heuristic science to identify some of the best possible query plans that execute queries within distributed frameworks. Heuristics usually involve operator selectivity information in constructing a fundamental process. This is because heuristics is essentially a practical and “hands-on” subject of study. Additionally, heuristics works wonders in RDF-based languages, bridging the gap between language properties, subjects, and even RDF characteristics. However, it is normative that even Heuristics is bound to leave the deterministic algorithms with subpar or sub-optimal query plans (excused because the dynamic approach procures the best possible plans and pathways for queries) (Rabhi and Fissoune, 2019). In the deterministic argument of algorithms, cost is an essential factor to consider. Costs are creditable because the performance of every executive algorithm is based on the costs incurred by them to carry out activities for the processing of queries. For instance, take data transmission process- a very much necessitated process, as it helps retain information within two points of an algorithm.

Consequently, sending data from one node to another within an algorithm is bound to incur some cost. Other functional features in an algorithm also work on this same principle. Take the event of an algorithm, wherein *node A* witnesses the performance of a scan. Consequently, the data extracted from that very scan becomes transferable because of its feasible nature, and other algorithms are likely to extract this sequence for themselves. This transfer of data, which is usually executed in between two nodes, has a set cost. Naturally, all deterministic query processing algorithms would prefer to incur the lowest cost possible in all their functional activities (Kaneko and Chishiro, 2018). There are various models used across different algorithms to estimate and reduce costs while maximising efficiency:

**Cost Estimation for Plans:** The typical manner for estimating a query plan's final cost can be initiated by ascertaining the cost of all possible operators within the set plan. This model allows us to have a precise computation of all individual costs (concerning their respective operators), challenging an algorithm. Consequently, the utility of the cost associated with a given operation is assessed through its corresponding cost metrics, including examining factors like time-cost balance, cost variance between budgeted and initiated costs, curating an analogous report of the performance statistics and the costs invested. Thus, cost metrics can be acquired from reading RDF triples involved in query assessment or resource consumption. In their turn, RDF triples are based on costs incurred during functions (as mentioned previously) like CPU consumption or data transmission. Finally, costs can be weighted to model the effect of fast/slow machines and communication links. Usually, costs incurred over transmission are expressed per byte or through any other fixed cost unit.

**Response Time Models:** The Response Time Model estimates costs based on their association with two kinds of operators: those that are concurrently executed and earmarked for possible execution if an optimum solution has not been arrived at. When queries are divided into sub-queries, they are directed into parallel systems like in which they are processed

simultaneously whenever such an instance is presented to them. A cost model working within such a parallel system gauges the operator's overall resource consumption and the effectiveness of comprehensive shared resources by a group of operators. This computational system assumes the significance of different resources and their consumption level and compares the resultant data to find the best possible solution. The possible result is realized from a resource that has the lowest response time to a query. An entire group of operators' response time denotes the total consumption of resources (both individual operators' and overall usage of shared resources). Ultimately, the most optimal and cost-effective query plan can be determined by using dynamic or randomized algorithms. Consecutively, dynamic algorithms are perpetuated through a distinct set of steps that are targeted to move at the intersecting edges of various solutions. An "edge" can be introduced in between two solutions by transforming one solution to another through precisely one move (Kaneko and Chishiro, 2018). Once an "edge" has formed, dynamic programming algorithms test the applicability of such an edge by carrying out a random walk-through along its surface, i.e. the path of this edge is traced until the edge is deemed as leading to a dead-end. The system follows through with such termination when a given time frame of work has been exceeded.

To summarize, if an edge fails to lead to a solution within a time limit, it is instantly terminated by the algorithm and replaced by a new plausible solution. This is why the most optimum solution is said to occur within dynamic algorithms. Their flexible nature allows for a "trial-and-error" methodology that cannot be employed in a deterministic algorithm- where every pathway continues to be considered a means to an end despite evidencing otherwise.

## **2.6 Query Execution Techniques**

Apart from the models of cost estimation covered previously, other query execution techniques in distributed databases are gaining popularity through a specific focus on enforcing

connectivity between nodes through distributed database tables (tyagi, 2015). That is to say that these methods aim to link information across distributed database tables and to establish how the resultant data from the connections made are to be distributed between two nodes. For this purpose, the following techniques can be utilised:

**Row blocking:** In Row Blocking, any communication between distributed database tables can be done so according to communication protocols, which state that information can go from one node to another through the form of loads. “Loads” refer to the capability of transferring multiple messages within a network. These loads are represented through “tuples”, which are single-handed records for a given table row (tyagi, 2015). Consequently, this technique of collecting tuples transported in blocks results in much lower network overhead than sending fewer, more independent messages.

**Communication Cost Optimisation:** The technique of Communication Cost Optimization has its basis for operation commonly in a data federation background. Data federation, an alternative approach to data sharing, enables most data to retain their primary source locations until they are required to satisfy downstream needs. In such cases, the network configuration accepts more responsibility than the data itself. Thus, the costs faced to undertake this technique could depend on the number of nodes and the middle ground between them that is to be covered (Sasak and Brzuszek, 2010). This process is filtered into its most optimal form using an optimiser- which finds the cheapest transmission route via the network nodes.

**Multi-threaded Execution:** Multi-threading Execution in a distributed database is an efficient route for quick functional execution. This technique uses the operating system's support and generates multiple threads of execution through its Central Processing Unit (or CPU). This technique optimizes itself by dividing queries into sub-queries and characterising them with a thread. Each expedites query execution. Such multi-threads could give query execution an edge when combined with query parallelisation (Sasak and Brzuszek, 2010).



**Horizontally Partitioned Data Joins:** Primarily, Horizontally Partitioned Data positions rows separately, rather than splitting them into columns. Several rows of horizontally partitioned data tables can form a union or joins based on specific logical protocols. These protocols include the need for the combining tables to be concurrently horizontally partitioned (Poovammal and Ponnaivaikko, 2010). For instance, if Table A is partitioned horizontally, in a way that is compliant to B, they can be calculated in the following possible manners:

- (A1 UNION A2).B
- (A1.B) UNION (A2.B)

An optimiser is required to consider the factors required for such an integration, as this technique does not find value in an uncontrolled environment like the Web of Data.

**Semi-joins:** The Semi joins technique is based on the principle of transmitting only the necessary columns to perform a joins operation, i.e., a semi-join returns columns only from Input A or does not return anything at all. Through this function essentially prevents any duplication error from occurring within the tables (Daenen et al., 2016). It is similar to a regular join in that it returns a column from one join Input (A), only if it matches at least one column from another join Input (B). The remaining but necessary tuples, discarded during the execution of a join, are dealt with later.

**Double-Pipelined Hash Joins:** The Double Pipelined Hash Join system is an augmentation to the symmetric hash algorithm. Compared to a partitioned join, this system requires less source knowledge to optimize its data. This is because its technique simplifies the query execution process. Like other join techniques, the double piped hash join system is initialized by creating a joint between Input A and Input B. Once the join is executed, a couple of empty in-memory hash tables are produced (Tang et al., 2019). These hash tables are elementarily equivalent to their Input counterparts. To begin with, tuples from Input A get processed by identifying whether they share similarities with the hash table B. If it is found that a tuple from A matches

table B in its characteristics, then this tuple is outputted and formatted into its respective hash table A. The same procedure is employed in the processing of tuples from B. Once matching tuples are established in their concurrent hash tables, they are rechecked to verify whether they identify with each other and satisfy the findings belonging to the created join in general. With the procedure being established, these hash tables can also carry potentially harmful consequences in tow. When gathering data into hash tables, there is always a detrimental chance of incrementing them with excessive data. This overloading could pose a threat to the central database memory (Tang et al., 2019). Such a scenario can be avoided if the system integrates memory backup into its functioning.

**Bindings:** Bindings limit the findings of a sub-query into a respective, specific database within data federation operating systems. This foundation of prohibition or restriction fixes a given sub-query within its four surrounding walls and necessitates its performance only to return solutions for low-level queries. This enables the regulation of a copious number of sub-queries that can otherwise return excessive results together. Thus, one sub-query is specific only to its database, while another query is restricted to a different database. Let us imagine a query, which contains a remote query execution that is unrestrained and may release unwarranted results into a database. This can be highly inconvenient in a database that is not competent enough to return these results. This negated nature of a database may render even the application of query planners ineffective (Moeller and Frings, 2014). This is because, while the query may disintegrate into subsequent queries, its solutions would be dispatched to a remote database, which could get saturated with the results of even a single query. In turn, this optimization through a remote database could ultimately lead to a network overload.

**Top/bottom queries:** Through the onset of top/bottom queries, a user can sort his final query results as he pleases. At this stage, the user controls a particular movement of the query results as per their choice (Zhu, 2015). The user can do so by selecting top or bottom queries and

executing specific values on them. As a consequence, this would generate additional movement among the solutions. For instance, *Stop* operators can be implemented to avoid unnecessary data exchange between different nodes in the database.

**Streaming results:** Once a database begins to generate results, it ships them to various federations. Here, the data gets transferred between various nodes within the federation itself. This facilitates the streamlining of results- which is essentially a positive segment of data transmission wherein a node keeps producing data simultaneously as and when another node consumes it.

Different nodes receive different functional benefits while streaming continuously. For instance, some nodes may flourish on execution efficiency, while others may be helped by reducing memory overheads. In the following sub-section, the focus is shifted to client-server architectures (Zhu, 2015). The primary traits of these architectures are described before their classification.

## 2.7 Query Federation Systems

This section investigates a Query Federation system of data processing and introspects on its various archetypes. Query federation is an information retrieval-based platform that can search for and combine data from various sources (Almourad, 2013). We strive to define and map out set routes through which several resources are utilized on this platform. The author discusses the many query optimisation techniques that are executed upon this foundation. The following architectures of Query foundation contribute to data processing in this model:

**Peer-to-peer:** The peer to peer (or P2P) is a novel distribution approach. Each site is limited to its functioning within a “server” capacity that activates the entire federation in this architecture. This server installs some aspects of the database into relevant sites. However, it is not just the server that these sites are limited as well. Sites also act under their clients' capacity and provide search result information by returning them to the federation (Almourad, 2013).

This peer to peer architecture enables communication within sites by their common attribution to a server. This server, in turn, defines policies that dictate communication protocols.

**Client-server:** In a client-server architecture, every site in the federation is fixed with a particular role, a specific part whose duties it must fulfil within the federation. The two fixed roles that are alternately employed to each site is either the client or the server. The client-server architecture also postulates that a client must operate within its capacity and not engage with other clients (Korneva and Khorev, 2018). However, where the P2P system had a standard server of engagement, that is not the case with a client-server federation model- so all engagement is strictly prohibited. This restriction extends to servers, which are not even allowed to establish a means of communication with each other.

**Middleware, multitier:** The multitier middleware architecture creates a hierarchy among sites, dictated by their different levels of processing. Every site within this federation can operate on the scale of either a client or a server. However, its identity is decided by its position within the federation and the site seeking engagement with it. Notably, a site inhibits its capacity to communicate as a client only by doing so with other client-sites on the same level. Alternatively, it can find engagement with a server through nodes- but only if the latter acts in its ability as a lower-level server. Thus, sites cannot communicate with each other at the same level or even a different level.

There has been much debate regarding a query execution regarding where a specific query must be stored- a client site or a server site? A general argument is that databases stored in and by the server sites of the federation feature better computing resources in terms of quantity compared to the client machines—more the rate of resources, less the communication costs. Thus, the differing range of federation servers and clients gives us a choice between them based on resourcefulness and communication costs (Chahal and Singh, 2021)

. The indicated question is to determine where it would be most optimal to implement a query with redundant costs yet excellent service. Would it be helpful to shift the data to the client machines from the resource-cantered servers, or would it be more prudent to shift the query to the data, considering all associated communication costs? Several alternatives appear in this context to help resolve this issue of query storage and execution. They are as follows:

**Query shipping:** Query shipping aims to establish a safe pathway between clients and the servers themselves. Queries that are to be dispatched from one to another follow the set pathway format for execution. Due to the constant nature of this exchange, a query returned from a client to a server or vice versa retains a much more intensive state than when it was initialised. Every “exchange” within the federation requires that a query experiences execution at the lowest level of a hierarchy of its objects, from where it gradually increments. The “exchange” of query that occurs is in between the client and the server. Once a client departs with a query, a server site gains remote access at a correspondingly lower level. Once processed, the query is shipped back, to the client, from where it would be returned, and so on. Suppose the given execution of a query occurs within the setting of multiple server configuration systems (Korneva and Khorev, 2018). In that case, a pre-existing middle layer (which could constitute either server or gateways) facilitates the transfer of queries between the client and server sides.

- **Data shipping:** In applications, data shipping is a general process that brings data closer to the applications to arrange interaction between the two elements. Data shipping is a similar yet raw version of this process. In data shipping models within a federation, the queries imposed exist within the client-side database. The general objective is to ship the required data from servers to the queries. Thus, this process provides a quick alternative for the middleware, multitier approach by storing the query

in clients while executing the needed data from the servers. Notably, the data gets cached at the client's machines, either on disk or main memory.

- **Hybrid shipping:** Hybrid shipping completely redefines shipping by combining two powerful query and data shipping methods. This shipping process fuses the underlying mechanisms of the prior mentioned systems and multiplies their effectiveness. In response, query operators either get executed on client or server sides based on their optimisation efficiency. Concurrently, the alternative platform allows clients to perform data caching (Korneva and Khorev, 2018). However, specific optimisation techniques can maximise the characteristic usefulness of client-server systems and their architectures. We describe the techniques of query optimisation for systems that implement these architectures.

**Site selection:** The Site selection alternative refers to selecting a site where an operator belonging to a specific query is to be possibly executed. The selection of a site for this purpose remains commonly emulating the process of data transfer. Thus, Site Selection allows the modelling of query shipping, data shipping, and hybrid shipping within its processes. The shipping is carried out through the same options used to select a site for execution. In such an event, note that a site annotation characterizes every operator in the field. The respective operator is attuned to this particular site, where it is about to be executed. The selection of sites for Query execution depends on factors such as network latency, the characteristics of servers, and the volume of data that is to be shipped or transferred (Abid, Rouached and Messai, 2019).

**Optimisation:** Optimisation is a phenomenon that can be carried out within various nodes because the client must choose the right node to execute the query. This aspect of node selection is an increasingly perplexing set of decisions to make. For instance, the node selection required to determine where a query plan is to be optimized is a heavy choice because every node is bound to multiple others (Devulapalli and Bagui, 2018). This means that one node knows a

diverse set of many more nodes, making it hard to decide among the different opportunities for execution available within the federation. A helpful option for deciding upon a site for query optimisation is to input a query into the client side. Inversely, prompting query optimisation and plan refinement within the server side is also a tempting option. Choosing by implementing a query into the server-side can provide more excellent knowledge about the system's current condition. This server contains just the correct information on the federation to curate the best possible query plan. No servers possess comprehensive knowledge of the entire system in such a configuration involving multiple servers.

**When to optimise a query:** The process of query optimising can diverge into multiple ways, which are all efficient within their spectrum. One such possibility is to facilitate query optimisation at the compile time of the system. The compile-time refers to a point wherein query operations meet source codes for execution. Currently, there is usually a considerable range of information available relating to data nodes, upon which query optimisation can be determined. However, the time may crash in unforeseen circumstances, and the query optimisation plan would fail altogether. Another possibility of query planning leaves much room for progress in optimising queries within the run time of data through a dynamically chosen or compiled plan (Devulapalli and Bagui, 2018).. As a general procedure, the execution of a query plan within this system is usually observed and regulated. If any mishaps or errors are found to occur within the query plan on the run time, the query plan is remodelled in a different direction.

Meanwhile, a different approach towards dynamic query optimisation aims to split the optimisation/execution into two distinct and operational phases. The first phase constitutes the breakdown or decomposition of a specific query into its respective sub queries. The resultant amplified quantity of subqueries is capable of being executed through a single server. Consequently, this feature allows for the parallelisation of single queries by establishing

linkages between the findings of subqueries (Devulapalli and Bagui, 2018).. The selectivity cost of joins collectively incurred to combine the subquery results is usually determined by the speed at which the servers link queries to find and yield results.

## 2.8 Adaptive Query Operators

Adaptive query operators are a quick, authentic and untraditional way of getting queries executed within a federation. While conventional operators function loose based upon a trial and error method across various models, ADPs are flexible enough to adapt and mould themselves to the need of the hour. For instance, one of the many functions of an ADP includes analysing real-time query run statistics and using this information to create customized optimisations (Chavan and Phulpagar, 2016). An adaptive query operator quickly facilitates the ease of execution as per a query executor's situation-specific demands. These demands are commonly met in the following ways.

**Symmetric Hash Join Operators:** The systematic hash join operator method is a part of the join algorithm mentioned previously in this section. It popularly operates under the double pipelined has join model. Like its ancestry, the Systematic Has Join Operators usually maintain two hash tables, each of which is attributed to a different relation. These operators usually await the arrival of data into their respective tables before processing them to yield results. Once Symmetric Hash Join stores the tuples into their related table, they are investigated against the table positioned at the opposite end, i.e., the data prevailing from different inputs are put in congregational tables compared to confirm their similarity values (Oguz et al., 2017). Thus, depending on the availability of a match between the tables, the operator processes the data from both inputs. Additionally, the operator also undertakes the performance of frequent symmetrical movements within a system (Sinuraya, Rezky, & Tarigan, 2019). Frequent symmetrical movements refer to specific points which enable altering join orders without consecutively affecting the correctness of data.



However, note that the join operator exhibits limitations within its features of adaptability. In an event where input data become unavailable due to communication loss or network traffic, the query plan execution cannot be undergone. Thus, it is essential to remember that this method is not the best configuration in a stream-based architecture, i.e., the disabilities inhibited by this model necessitate a severe evaluation and reconsideration of the operators and their functions in the system (Sinuraya, Rezky, & Tarigan, 2019). The biggest drawback of such a join operator is its high memory usage, as the hash tables also need to be constructed on more significant input relations.

**Eddy:** The founding notion of this concept is to employ a method of execution wherein the tuples of data get routed via operators. Such a model would also be operational in altering the sequence by which tuples get collectively routed into the system. This gives Eddy the leverage to track tuple execution and make router-related decisions for them.

**Symmetric Hash Joins/MJoin:** The Symmetric Hash Joins/ MJoin is a final integration of the above listed Symmetric Hash Join and Eddy operators that contributively outline the AQP system. Conclusively, MJoin signifies the generalisation of Symmetric Hash Join for more than one joint. The operator constructs one relational hash index on each joining attribute (each input side) within the concerned query. It incorporates a light tuple router—the tuple router aids in accelerating the process of touring tuples between the different hash tuples. Whenever a new tuple is introduced to the existing system, the initial response is to solidify it within hash tables, which are investigated against each other to find a match among the tuples within them. As expressed previously, a memory index must be constructed for all input joins misaligns with the hash joins' inclination to consuming high memory. This provides to be a key challenge to this operator (Chen et al., 2018).

Meanwhile, memory consumption represents a vital challenge of this operator, given the fact that a memory index must be constructed for all input joins. AQP addresses challenges that are

not confronted in conventional query processing systems in the presence of statistics, availability of servers and the constancy of costs during execution time. AQP confronts the problems generated from not lacking previous information: unexpected correlations, missing statistics, dynamic data and unpredictable costs using feedback to tune the execution. In addition, AQP can be generalised to several other contexts, especially during the intersection of query processing. Ontology-Based Data Integration is a process that uses ontology to combine data (Osman, Ben Yahia and Diallo, 2021). What sets Ontology-based data integration apart is their involvement of multiple heterogeneous sources during the execution of this procedure (Zhang, 2014). This section of the dissertation focuses on integrating RDF data and discusses how processing distributed data queries can do this. RDF is standardized as a standard data model, and rather than consuming it like a heterogeneous source, it finds no use in being mapped out to integrate various data sources. Naturally, it facilitates the onset of ontology-based data integration for itself. The final execution of the query is then performed by accessing several RDF sets through a Query Federation system (Achichi et al., 2019).

## **Distributed Query Processing Systems**

The processing of SPARQL 1.1 is heavily engineered and supported by specific standard systems. Such systems commonly employed to derive value from the official SPARQL 1.1 federation extension include ARQ, RDF- Query, Rasqal RDF query Library or ANAPSID. We are more or less likely to come across these engines as often as possible within a query processing duration. Meanwhile, note that other systems can also run a query processing system for SPARQL 1.1, the most popular of Networked Graphs, DARQ, FedX, ADERIS, SPLENDID, and many other engines. However, these processing systems often fail to mandate the SPARQL 1.1 federation language, possibly due to the comprehensive protocols across which it stretches. Thus, they are unlikely to find proper compliance with the SPARQL 1.1 federation extension. However, they fail to comply with the specification of the SPARQL 1.1

Federation. The other system which lends support to distributed RDF querying is illustrated. However, this system is not considered in this study since it uses SeRQ instead of SPARQL. Below is a general discussion on some popular query processing engine systems that contribute to queries in the SPARQL 1.1 federation extension.

**ANAPSID:** ANAPSID implements *agjoin* and *adjoin* query operators. While the former incorporates a hash join apart from saving join tuples for joining operators, it can be seen that the *adjoin* operator masks the delays attributed to the data sources. After decomposing the SPARQL queries (federated) into several source queries, ADERIS integrates the results using a couple of techniques: 1) adaptive join re-order; and 2) optimisation of succeeding queries to the various data sources in order to retrieve additional data. ADERIS implements a greedy algorithm to optimise these queries, facilitating identifying the most optimal query plan (Acosta et al., 2011). After that, index nested loop joins utilise an index related to join attributes to explore tuples from appropriate inputs and outputting corresponding tuples to the subsequent operator.

**ADAERIS:** This query processing engine is known for using optimization as a means to a successful query plan. The process begins with the decomposition of SPARQL queries, which are entirely federated into various source queries to lighten the load of an extensive and complicated SPARQL query. Once the breakdown of the query has been performed, then the decomposed queries return several results. Which are combined for interpretation using a different technique, some of which include: 1) adaptive join reordering; and 2) optimisation of succeeding queries to the various data sources in order to retrieve additional data. An optimal combined query plan is formulated for execution after attaining a maximum optimisation level (Kim et al., 2017). This plan is then followed up by index nested loop joins, which utilize their respective indexes (standard join attributes bind that) to explore tuples from inputs across the

system. They also determine the output of specifically connected tuples to their relative operators.

**Symmetric Index Hash Join:** The original symmetric index hash join was implemented by Ladwig et al. to perform as an engineering algorithm in the query processing system. The Symmetric Index Hash Join system uses an integral combination of queries with remote SPARQL endpoints. Remote SPARQL endpoints are inhibited with access to RDF data stores that are only situated within a local dimension. This access to remote endpoints by the system makes it a quick and recommendable hash join approach. This is because the Data derived from the localised RDF data set is saved within an internalized hash structure. The close distance of this existing data makes it possible to obtain speedier access whilst implementing a join that utilises remote data (Liu, He and Meng, 2018). In addition, the authors share cost models concerning their preceding work, where they used non-blocking operators to join the data.

**SPLENDID:** This query engine processing system goes beyond the scale of ‘Sesame’ in query federation. SPLENDID operates its query processing by executing them through a join re-ordering system, whose functioning is based on the order of the gathered statistics. SPLENDID also premises its operations by following joint reordering rules through participation in a cost-reductive mode. The statistics used in the query processing functions are collected through VoID descriptions. This accurate measurement of data acts as a medium for the efficient implementation of join re-ordering (Saleem et al., 2016).

**SemWIQ:** The SemWIQ is a query processing system that utilizes a channel of mediator wrapper upon which it bases its operations. The mediator wrapper primarily behaves as an agency or instrument that accesses heterogeneous data sources -such as RDF data sets, or CSV files, or relational databases- an efficient and straightforward process. Mediums appear to be SemWIQ’s forte because a SPARQL processor called ARQ(of Jena) soon comes into play to

help the SemWIQ utilize them and write query plans before implementing its optimisers. These optimisers exhibit a general set of rules that must be regulated within the process of shifting all the filters or unary operators within the boundaries of a query plan itself (Langeegger and Wöß, 2008). Thus, unlike the join reordering system, which depends mainly on the given statistics, this processing method executes operations within a query space.

**DARQ:** DARQ goes on to extend the Jena SPARQL processor called ARQ. This necessitates the attachment of a configuration file into the existing query, including information about the L vocabulary of SPARQ, endpoint and statistics. DARQ implements both physical and logical optimisations that concentrate on using rules to rewrite the original query prior to planning the query (to blend elementary graph patterns at the earliest possible opportunity) and shifting constraints into subqueries to reduce the intermediate results' size. The other major drawback is that DARQ's ability is restricted to executing queries involving bound predicates. Meanwhile, Networked Graphs are known to produce graphs to depict content from different RDF graphs, facilitating graph sets that are then supposed to be queried (Quilitz and Leser, 2008). This implementation takes into consideration, optimisations including the deployment of optimisation algorithms (semi-join).

**FedX:** Extending Sesame, FedX premises its optimisations on grouping joins directed at the same endpoints (SPARQL) and a join optimiser based on specific rules. This subsequently orders groups of these patterns following a heuristics-based guideline. In addition, FedX reduces the likelihood of joins at an intermediate level by grouping mappings within a single subquery by leveraging the constructs of SPARQL UNION before dispatching it to relevant data sources (Qudus, Saleem, Ngonga Ngomo and Lee, 2021). Hartig and his peers suggested a model that attempts to take advantage of the Web of Data's navigational structure by applying executing queries. The authors have uncovered new URIs based on the first set of SPARQL query before populating a localised RDF repository. The query is then repeated in order to

seek new answers for the initial query. Meanwhile, more recently, Hartig propounded a heuristic to identify a feasible order to execute queries and incorporate a primary memory index elucidated.

**Existing frameworks summary:** This section gives a summarised information about the existing systems, which are relevant to our research are discussed in this section.

***ANAPSID:***

- ANAPSID finds use in steering the SPARQL 1.1 federation by implementing query operators called the ‘agjoin’ and the ‘adjoin’.
- ANAPSID becomes a valuable asset in the adaptability of its features, contributing to the increasing success of remote query executions (Acosta et al., 2011).

***ADAERIS:***

- This query processing engine is known for using optimization as a means to a successful query plan.
- Once the breakdown of the query has been performed, the decomposed queries return numerous results to the various data sources to retrieve additional data.
- An optimal combined query plan is formulated for execution at a maximum level of optimization of these queries (Kim et al., 2017)..

***SYMMETRIC INDEX HASH JOIN:***

- The original symmetric index hash join was implemented to perform as an engineering algorithm in the query processing system.
- Remote SPARQL endpoints are inhibited with access to RDF data stores that are only located within a local domain.

- The authors share cost models about their preceding work where they used non-blocking operators to join the data (Liu, He and Meng, 2018).

***SPLendid:***

- This query engine processing system goes beyond the scale of ‘Sesame’ in terms of query federation.
- SPLendid operates its query processing by executing them through a join re-ordering system.
- This accurate measurement of data acts as a medium for the efficient implementation of join re-ordering (Saleem et al., 2016).

***SemWIQ:***

- The SemWIQ is a query processing system that utilizes a channel of mediator wrapper upon which it bases its operations.
- The system acts according to a featured registry catalogue, which indicates the sources for querying and the vocabulary about which they must be executed (Langegger and Wöß, 2008).

***DARQ:***

- DARQ goes on to extend the Jena SPARQL processor called ARQ.
- DARQ implements both physical and logical optimisations that concentrate on using rules.
- This implementation considers optimisations, including the deployment of optimisation algorithms (semi-join) (Quilitz and Leser, 2008).

Some significant difference between Fedx, ANAPSID, ADERIS according to features are given below:

Features	FedX	ANAPSID	ADERIS
Indexing in Memory	Yes	Yes	Yes
Stored Index	No	No	No
Dynamic Indexing	No	No	No
Generating algebraic	No	No	NO
Cache	Yes	Yes	Yes
Decomposing main query	NO	NO	Yes
Static Generalization	Yes	Yes	Yes
Dynamic Generalization	NO	No	No
Static Specialization	Yes	Yes	Yes
Dynamic Specialization	No	No	No

**Table 2.3: Relevant systems characteristics**

## 2.9 Research Gaps and Proposed Research

This section elaborates upon the gaps prevailing in the existing research and consecutively identifies and introduces a thesis to cover the gaps surrounding the present research dynamics concerning RDF distribution. The following Research gaps have been identified after reviewing existing work related to the accessing distributed RDFs:

- Process and index RDF data into a centralised repository.
- Conversion of main SPARQL query into sub-queries is missing.



- While the sub queries bring back multiple data files, there is a necessity to merge all the given sub-queries results into one result to answer the main query.

This research aims to propose the framework for accessing distributed RDF (Resource Description Framework) and developing a test environment that measures the performance and accuracy of the proposed framework for processing the distributed SPARQL queries.

The proposed framework includes the following parts:

- To develop a novel technique to index the RDF data from different sources to analyse the data
- To develop a robust, reliable, and comprehensive framework that can bring RDF indexing, SPARQL query conversion, searching and combining results from different RDF resources under one umbrella.
- To evaluate the results and compare the proposed approach with existing approaches.

## **2.10 Chapter Summary**

This chapter has discussed the ins and outs of the foundation technology behind the semantic web. Furthermore, it has included a detailed review of the existing work done by other authors and a critical review of their work. It talks explicitly about data integration approaches, FDMS Architectures; Distributed Query Processing System generates optimised query plans for Distributed Query Processing, various Response Time Models. Then, the chapter touches upon and explicates other Query Execution techniques before moving on to investigating a Query Federation system of data processing. Briefly discussed Adaptive Query Operators. Subsequently, we discussed Ontology-Based Data Integration and Query Processing Systems, such as ANAPSID, ADAERIS, SYMMETRIC INDEX HASH JOIN, SPLENDID, SemWIQ,

DARQ. After briefly specifying the challenges and limitations of this study, the chapter then summarises what has been discussed so far.

# Chapter 3

## Research Methodology: Design Science Research

### 3.1 Introduction

This research is exalted through its methodology - the core processor of a developed system. The methodology that fuels the proposed framework have been highlighted in this chapter. The researcher uses analytical and constructive analogies to create a system founded on authentic and solid proof. The constructive style generally requires a form of validation that does not need to be as empirically supported as other types of research (including explanatory research). However, despite their unfounded nature, constructive methodologies must be looked at from a goal-oriented perspective. The methodology used by the author of the research to achieve them have been suggested in this chapter. This research developed the framework for accessing distributed RDF (Resource Description Framework) and developing a test environment that measures the performance and accuracy of the proposed framework for processing the distributed SPARQL queries. Thus, Design Science Research methodology, a combination of analytical and constructive methodologies, has been used, which is consists of the five stages: Awareness of problem, Suggestions, Development, Evaluation and Conclusion,

In this chapter, an analysis of the research design and methodology is presented. The following section presents the research paradigm employed in the current study and then describes the research methodology and the theoretical framework. The quantitative and qualitative approaches applied in different chosen research methodology stages. Design Science Research Methodology constitutes two substances, the "Design Science" and "Research Methodology",

to better comprehend this principle and further relate them to Information Systems and Computer Science research as a methodology. Before explaining what appropriate literature is saying about Design Science Research Methodology in the Computer Science and the Information Systems research study, we need to comprehend these substance words. Firstly, the author looks at some definitions and a short description of "Design Science" and "Research Methodology". Design Science is developed in 1957 by R. Buckminster Fuller and is viewed as an organized type of designing and is interested in an understanding acquisition that connects to styles and their activity. Design Science emphasizes systematic, testable and communicable techniques (Carstensen and Bernhard, 2018). Design Science is also seen as an outcome-based method that offers a specific guide for assessing and versioning a task. The design science as a paradigm has its root in engineering and science of the artefact, its essentially on resolving the problem through imaginative innovations which specify the concepts, practices, technical abilities, and products in which analysis, style, implementation, and details system usage which can be efficiently and effectively reached

### **3.2 Research Paradigm**

Before choosing a proper research methodology, the author picks a suitable paradigm for the current study. The research paradigm that a scientist selects to follow influences each research action, from the decision of the research study issue to be investigated to information analysis and analysis. A research paradigm can be characterised as an 'essential set of presumptions or benefits that direct a research study procedure. In social sciences, there is a series of paradigms that reveal variations in their underpinning philosophical hypotheses. For that reason, before a researcher defines a suitable research paradigm, it is essential to study its philosophical presumptions and clarify that it is suitable for his/her research. So far, there are three primary philosophical presumptions: methodology, epistemology, and ontology. Methodology refers to research methods or methods utilized in order to acquire knowledge. Epistemology explains

the type of relationship between the knower and what can be understood. Lastly, ontology suggests the type of truth and what can be known about it. The author provides the paradigm applied in the current research study and justifies why it is followed in the subsequent paragraph.

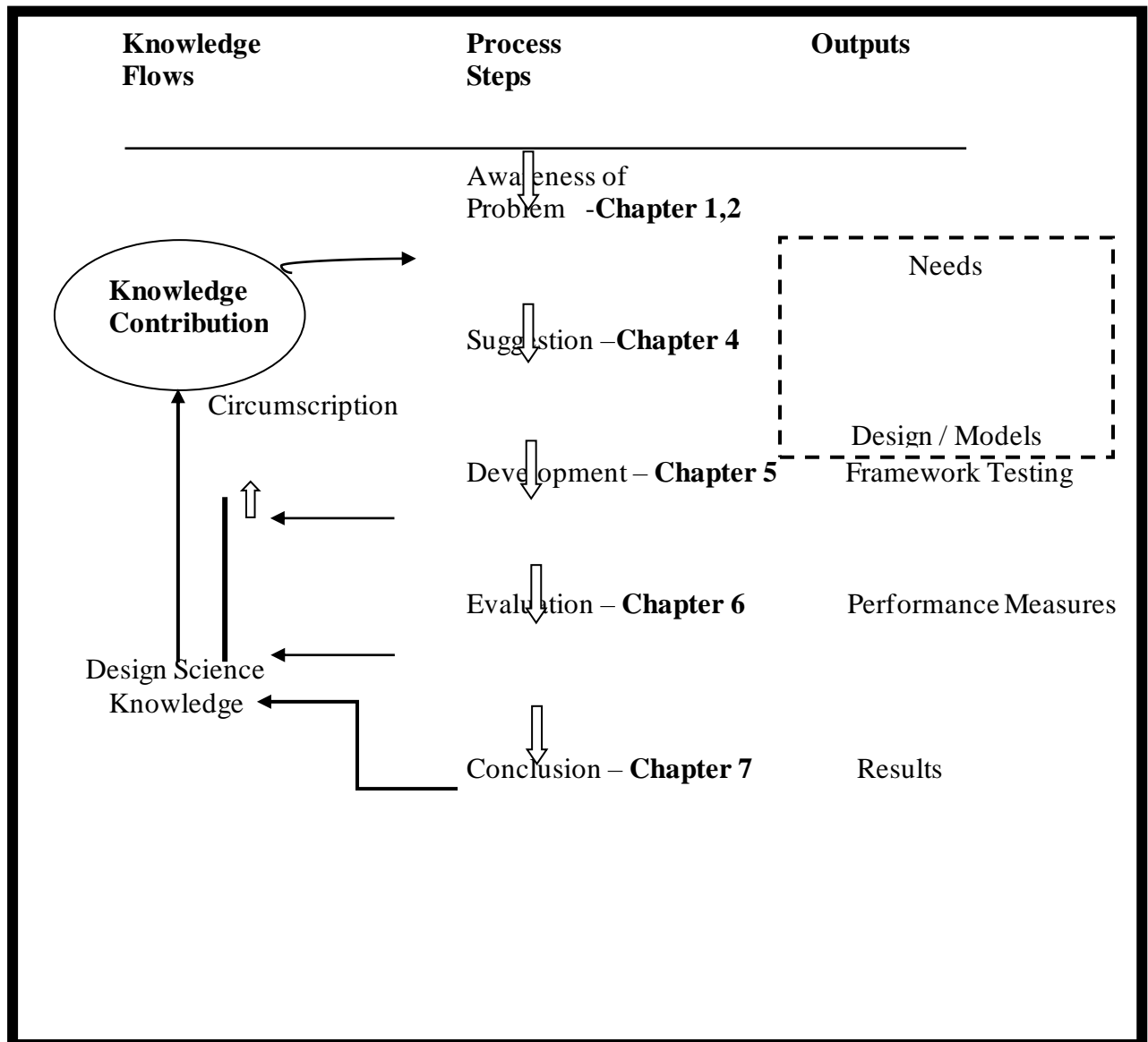
### **3.3 Research Methodology**

The author used the Design Science Research methodology, an outcome-based information technology research methodology that provides assessment and version guidelines within research projects. Design science research concentrates on the advancement and performance of (designed) artefacts with the explicit intention of enhancing the functional performance of the artefact. Design science research study is usually applied to classifications of artefacts, algorithms, computer interfaces, design methodologies, and languages. Its application is most noteworthy in the Engineering and Computer Science disciplines, though it is not limited to these and can be discovered in many disciplines and fields. In design science research study or constructive research, academic research objectives are more practical than a descriptive science research study (Carstensen and Bernhard, 2018).. A research study in these disciplines can be seen as a quest for understanding and improving performance. Such prominent research institutions as MIT's Media Lab, Stanford's Centre for Design Research, Carnegie-Mellon's Software Engineering Institute, Xerox's PARC and Brunel's Organization and System Design Centre use the Design Science Research approach. The primary objective of design science research is to establish knowledge that professionals can create for their field problems. This objective can be compared to the 'explanatory sciences', like the natural sciences and sociology, to develop understanding to describe, explain and anticipate. The primary function of design science research is to understand a problem domain by structure and application of a created artefact. Design Science Research (DSR) develops and evaluate IT artefact planned to fix the determined organizational issues. Design Science Research (DSR)

has been seen to make up the third kind of science, "Artificial", in addition to the natural sciences and the human sciences. The Design Science Research Methodology is a new Information Systems and Computer Science technique because of its fast growth in the discipline. Design Science Research Methodology basic reasoning of discovery is deductive because an unsolved issue is taken and tries to find justificatory knowledge which assists in solving the issue. Design Science Research Methodology (DSR) is seen as the opposite of IS research study cycle that develops, evaluates information Technology artefacts intended to solve problems recognized in an organization. In IS design science, research represents the essential elements of the Information System (IS) research study landscape. The Design Science Research study contribution in Information Systems is a step in how it is applied to organization requirements in a proper environment. It adds to the content of the knowledge base for more research to solve an existing problem (Carstensen and Bernhard, 2018).

### **3.4 A Design Science Research Process Model**

In this section, a design of the primary procedure followed by a design science research study in its multiplicity of as-practised variations is explained. This model is an adaptation of a computable design process design developed by Takeda et al. Even (Carstensen and Bernhard, 2018). Though the different phases in a design process and a design science research procedure are comparable, the activities carried out within these phases are substantially different. Likewise, what makes the Design Science Research study process model different from the corresponding design process design. The standard design science research phases continue as follows:



**Table 3.1 - Design Science Research Process Model**

### 3.4.1 Awareness of Problem/Objective:

This section of the Design Research Methodology describes the goal of this research and the research problems. The objectives of this research, discussed in [section 1.2](#), is to design the framework that can access a distributed RDF (Resource Description Framework) by encouraging developing a test environment. The test environment, chapter 6, is designed to supply the research with the appropriate resources to measure the performance and accuracy of the proposed framework that assists in processing the distributed SPARQL queries. Thus,

the research problem discussed in chapter 1 is used to enforce an innovative technology through which a user can index and compile RDF data from various sources for examination and analysis. In turn, these results are thoroughly evaluated in [chapter 7](#) and utilized to compare the proposed framework with an existing framework that determines the success of this thesis. By taking into consideration all the factors at stake in congruency with this research, the author attempts to prove the following hypothesis, in [chapter 4](#), through the track of this thesis: “Formalising the process of Semantic Querying through the algebraic conversion and the sub-querying of the SPARQL query language have a positive effect on the speed and accuracy of accessing data from across distributed data sets in the Resource Description Framework (RDF).” Storing and indexing Semantic data under a familiar domain is a significant assumption. This aspect of storage can intervene in the correct assessment and interference of data and must be carefully examined to determine the standard of research. Thus, the semantic web search should address the different lexical, semantically restricted and structural issues in Ontology Development to transmit efficient research that is not restricted by advancing the given drawbacks.

In [chapter 2](#) author explained the futuristic approach employed to query repositories of RDF data within this research. At the same time, it takes a step back from conventional viewpoints, tool, techniques, and systems that have previously contributed to accessing distributed RDF data. Instead, it tests new theories that may bring in results in their more advanced form. To better understand the current study by reviewing the broad research that has already been done in RDF data source integration. It precisely talks about the types and approaches of data integration. The chapter touches upon and explicates other Query Execution techniques before moving on to investigating a Query Federation system of data processing and then briefly discussing what Adaptive Query Operators require. Subsequently, chapter 2 explores deep into



Ontology-Based Data Integration and Query Processing Systems, such as ANAPSID, ADAERIS, SYMMETRIC INDEX HASH JOIN, SPLENDID, SemWIQ, DARQ.

### **3.4.2 Suggestion:**

This section of the Design Research Methodology presents the suggestions required to solve the problems identified in the earlier stage (Carstensen and Bernhard, 2018).. The suggestion is an innovative action where the proposed design is developed. [Chapter 4](#) proposes the methods, technologies, and elements to achieve the required framework. Stage 2, suggestions, introduces the framework, operators involved in developing the system and how each of these elements combines and complement each other to achieve the common goal of validating the research hypothesis. Chapter 4 proposes the conceptual framework, which presents the query execution process by gaining access to the data within distributed RDF sets across a database. This chapter also presents the elements like the semantic algebra involved in converting the traditional query language. Chapter 4 also elaborates the concepts included in the selection, projection, joins specialisation, and generalisation operators. The suggested algorithms, in chapter 4, include the RDF indexing algorithm, the converting main SPARQL query into the sub-queries algorithm and merging the results algorithm. These three algorithms work collectively to start and end to facilitate the developed query processing system.

### **3.4.3 Development:**

The developed design is implemented and tested in this stage to check the accuracy of the system (Carstensen and Bernhard, 2018). The methods for proposed algorithms implementation vary depending on the framework to be created. [Chapter 5](#) demonstrates the implementation and testing of the framework that index the RDF data into the central repository. This includes discussing how any SPARQL query can be converted into its representative algebraic expression and be separated into directional sub queries. It holds and

supplies all information about a case study applied for the comparison: **Museum ontology**, which is used to demonstrate all the stages of the proposed framework. This includes implementing and details how converting SPARQL query into sub-queries can help fetch and combine results. To implement the framework author used the apache Jena framework. It provides a programmatic environment for RDF, RDFS, OWL, and SPARQL and includes a rule-based inference engine *Apache Jena framework*.

Apache Jena is an open-source Semantic Web framework for Java. It provides an API to extract data from and write RDF graphs. The graphs are represented as an abstract "model". A model can be sourced with data from files, databases, URLs, or a combination of these. A model can also be queried through SPARQL. Jena framework is used to build semantic web and linked data applications. Chapter 5 clarifies how to create Resource Description Framework (RDF) data with Jena and query the data (running SPARQL queries). In chapter 5 author tested all the data by using unit testing. Data of museum ontology is tested and evaluated through different phases. It tests all how the complete developed and processed system works. This chapter has done different tests, like a select test, join the test, outer join test, generalization test, and algorithm testing. The tested system uses the index mechanism to store all participated RDFs data sets. Tests showed that all the developed system units worked as expected and no errors during the testing of all units as a whole.

#### **3.4.4 Evaluation:**

This section of the Design Research Methodology presents the evaluation of the implemented framework through comparison with existing similar systems (Carstensen and Bernhard, 2018). In [chapter 6](#), the author evaluated the system's performance and accuracy compared to other similar systems. All chosen systems under this evaluation have implemented the triple pattern for the SPARQL endpoints, which bears similarity with our proposed system. Functionalities of this system prevent the user from starting the URL to fetch data from

distributed resources instead of overwhelming network traffic. This chapter assessed the proposed framework's outcomes and execution with other specific frameworks that handle distributed SPARQL queries. This undertaking aims to show that the proposed framework can productively deal with distributed queries on distributed RDF stores. Evaluation of the existing framework with our developed framework exhibited that the proposed framework handles the distributed SPARQL queries adequately. The author chose FedX, ANAPSID and ADERIS systems to compare with our developed framework and presented the outcomes in a graphical format. Different distributed SPARQL queries have been tested against the developed framework to evaluate the systems

#### **3.4.5 Conclusion:**

This last phase of the Design Research Methodology concludes all research and discussion about how it fits the research objectives (Carstensen and Bernhard, 2018). In [chapter 7](#), the author presented the contribution that specifically addresses the research problems. One of the challenges faced in this research was to extract distributed ontology through SPARQL. Retrieving data effectively from distributed RDF data sources is time-intensive. An optimised and proper structure was required because SPARQL queries are sent to the distributed end to retrieve data. After examining the existing system, it is apparent that they are looking to extract data from distributed RDF data sources directly, which can be efficient for a few distributed data sets. However, these systems do not work as efficiently if we require data from many distributed sources. After evaluating the existing system, the author concluded that these systems work well with limited RDF data sources. Chapter 7 concluded that our approach is better than other frameworks, e.g., converting the main SPARQL query into algebraic expression before extracting triples and variables information to store them inside the cache. The author discussed the limitation of this research as a developed framework worked very

well on the homogeneous environment and recommended that research be done in the heterogeneous environment where data can have different formats.

### **3.5 Chapter Summary:**

Chapter 3 ends on a note of the chosen research methodology, Design Science Research Methodology. This chapter has discussed the Design Science Research process's model, where each phase of this process model has been explained. In the first phase, awareness of the problem is a starting point of this research, where research problems have been discussed, and chapter 1,2 belongs to this phase. Phase 2 of the research process model, suggestions contains information about the development of the proposed framework. Chapter 4 hold all information about this framework where all semantic algebraic and algorithms have been discussed. Chapter 5 belongs to the third phase, Development/Testing, of this process model, where the designed framework has been developed and tested to prove the accuracy of the proposed algorithms. The next phase of this research methodology is Evaluation. Chapter 6 presented the evaluation process where a comparison of the developed/tested framework with existing frameworks has been discussed, determining the success of this research. The conclusion is the last phase of the chosen research process model. The author presented the conclusion in chapter 7, concluding that the proposed framework is better than other frameworks, e.g., converting the main SPARQL query into algebraic expression before extracting triples and variables information store them inside the cache. The chapter also discussed the limitation of this research and recommended that research be done in a heterogeneous environment where data can have different formats.

# Chapter 4

## Conceptual Framework of Querying Distributed RDF

### 4.1 Introduction

This chapter proposes the authors' methods, technologies, and elements to achieve the desired outcome. Not only does Stage 2 introduce to us the framework and operators involved in developing the system, but it also shows us how each of these elements combines and complement each other to achieve the common goal of validating the research hypothesis. This chapter, thus, proposes the conceptual framework upon which the research methodology functions to help in the query execution process by gaining access to the data existing within distributed RDF sets across a database. The methodology to be used also involves elements significant to the developed system. This chapter also introduces us to such elements as the semantic algebra involved in converting a traditional query language. Chapter 4 also elaborates upon the concepts included in the selection, projection, joins, specialisation and generalisation operators. These operators are usually in assistance during the process of processing and converting a query. After applying these operators, the system converts a query into its basic algebraic expression. Accordingly, this chapter proposes the algorithms behind the conceptual framework. The algorithms substantiated in this chapter include the procedural RDF indexing algorithm, converting the main SPARQL query into the sub-queries algorithm, and joining the results algorithm. These three algorithms work collectively to start and end to facilitate the developed query processing system.

## 4.2 Conceptual Framework

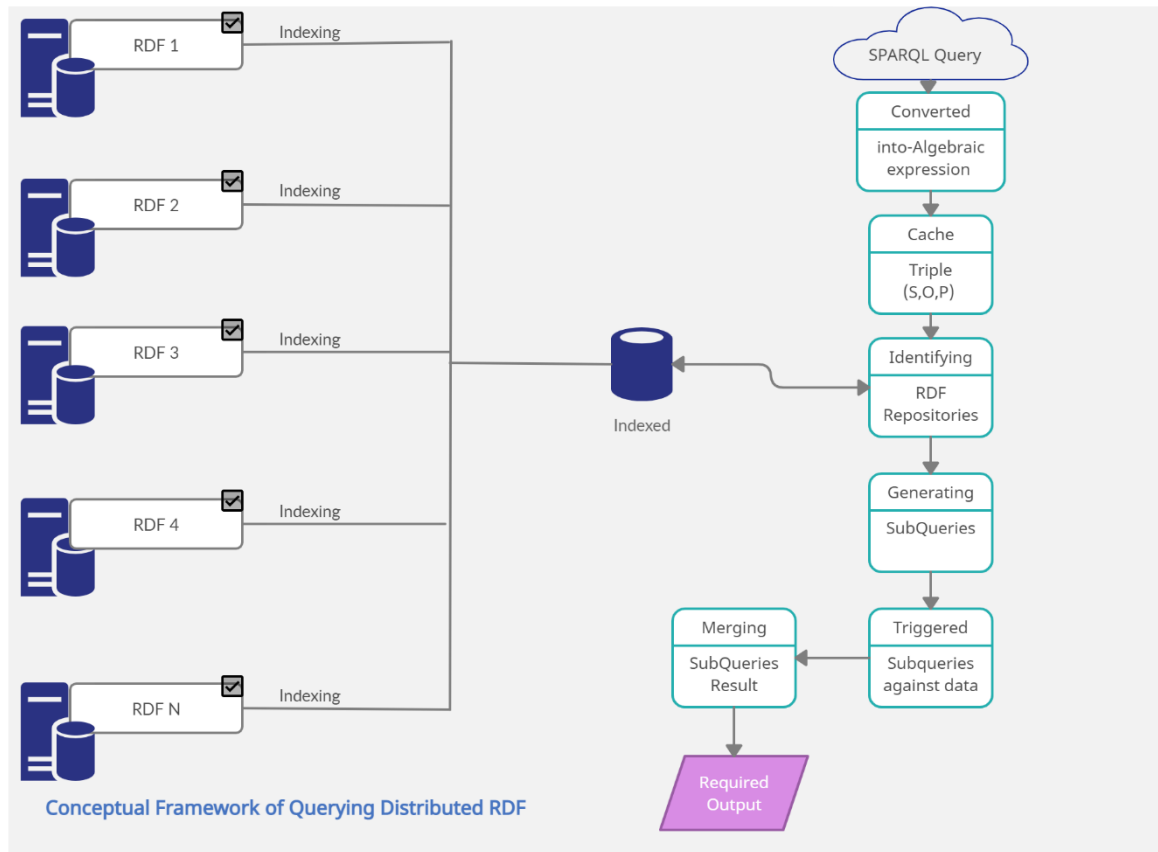
The conceptual framework used by the author of the research is generally based on a query processing language known as SPARQL. The W3C usually suggests the SPARQL to access RDF components, which is incidentally the objective that the researchers aim to obtain as these elements are essential for enabling the querying of data sets. The results about the main SPARQL query are consecutively provided in an RDF graph format, which requires another translation algorithm. To summarize, this query mechanism system proposed by the researcher for development follows the given primary stages:

1. To define the Semantic Algebra of the query.
2. To create an RDF index.
3. To access distributed RDF stores using SPARQL's subqueries generated from the single SPARQL query.
4. To create new RDF data after fetching and linking information from distributed sources.

This query system does not aim only to produce single query results. Instead, the researchers propose a model that fetches deeply rooted, meaningful and linked information from RDF sources that are distributed across the database. This can thoroughly evaluate and answer a query. In order to process the objective based on reality, the propositioned framework fulfils the agenda by converting the single main query into manageable sub queries that succeed in retrieving the information they need. The fundamental pattern of RDF information executes this strategy of query processing. A given RDF set uses a relevant <Subject, Object, Predicate> tuple model to duplicate the different parts. For instance, the subject is portrayed by S, and a given subject has a property P, which holds the value defined through O. The given Subject and Predicate of a sentence are described through URIs. An Object O can relatively go under the URI classification or exist as a literal. This strategy thereby acts as a straightforward and straightforward way of assessing information and expressing it in readable form, thereby

achieving query handling optimization of the RDF store. Figure 4.1 is given to portray an accurate representation of the researchers' technique and the design of their model. This segment outlines and depict the significance of utilizing the query analysis process to retrieve RDF data in a distributed atmosphere, which can be alternatively translated as the primary objective or goal that the researchers sought to accomplish. The principal segment makes up the heart of our SPARQL query structure. This structure is represented in the form of a centralised storage strategy. In this storage space, the indexing of RDF triples is carried out. RDF triples are indexed after fetching the required information, after which the centralised processing system links and communicates with distributed sources to extract information.

The fundamental function of this centralized processing system within the proposed framework is to investigate or analyse a given query in the form of broken and small chunks. By dividing a query into smaller chunks, the processing system directs these queries into remote areas. Information is returned to the central repository, where it is correspondingly addressed for assessment. The researchers then create a client interface that takes input as a *string*. This input then is processed on a centralised method via an HTTP URI. The author of the research then arranges indexes. These indexes hold essential data, which is essential and is used to form associations with RDF stores later. A centralized process performs the entire procedure. The framework is thus, represented through the stages. In summary, all the participated RDF data firstly is indexed in a centralized server. Following this process, the main SPARQL query is converted into its relational algebraic expression. The Subject, Object and Predicate parts are stored into a freshly created temporary cache. This cache identifies the relevant data and information by searching the stored RDF indexes. Subqueries are created and directed to the RDF store to get the required information. Finally, the results returned through the subqueries combine and form an output to display to the main SPARQL query.



**Figure 4.1 - Proposed Framework.**

### 4.3 Semantic Algebra

Semantic algebra is also known as the symbolic mathematical language that is used to represent semantic data. In simpler terms, semantic algebra functions to break down semantic information into the most basic, raw form of mathematical data that can make inference accurately by a computerized system (XU and HONG, 2012). Semantic algebra essentially helps in detailing systems down to a microscopic level. This is precisely why the technology of semantic algebra plays such a significant role in the research. The application of semantic algebra converts a SPARQL query into its algebraic notations. This process is usually done by using semantic operators, which this section discusses with an explanation and examples.



### 4.3.1 Operators

Semantic operators refer to operators that perform their tasks based on the semantic context of information. This implies that semantic operators can manipulate a given text and convert it into its semantic algebraic notation. For this research, semantic operators have been used to convert SPARQL queries into their algebraic forms. The operators used in the research are discussed below.

#### 4.3.1.1 Project:

$\pi$  Sign is used for the project operator. This operator takes the Subject (S), Object (O), and Predicate (P) as inputs and implements them into a source, usually a Schema.

**Syntax:**

$$\pi_{[S?][O?]}(source)$$

As explained previously, an RDF triple is made up of three elements: the *Subject*, *Predicate* and *Object*. In this case, the *Project* operator extracts information relating to the subject and the object from the source, a schema, and collectively bundles the three triple elements by source, replacing them with a single schema name.

#### 4.3.1.2 Select:

$\sigma$  sign is used to indicate the *Select* operator.

**Syntax:**

$$\sigma_{[logic]}(source)$$

The select operator selects and brings all required sources or nodes that meet the condition imposed by the SPARQL query. Arithmetic, Comparison or Boolean operators can be utilized

along with constants or strings inside the, after which they are source replaced with a schema name.

#### 4.3.1.3 Join

The  $\bowtie$  Sign is used to represent a join operator.

**Syntax:**

$$\pi_{[?X,?Y]} \sigma_{[condition]}(source)$$

$\bowtie$

$$\pi_{[?X,?Z]} \sigma_{[condition]}(source)$$

As identified by its name, the join operator joins or combines triples from single or multiple source points according to the conditions requested to be met by the main SPARQL query.

The researchers employ the operators, Project and Select, in their syntax for query processing into algebraic notation. The Project operator considers two parameters, “ $?X$ ” and “ $?Y$ ”. These parameters represent the different elements evaluated by the operator. As a triple consists of three aspects: “the $Subject$ ”,  $Predicate$ , and  $Object$ , the operator uses the parameters “ $?X$ ” to “represent the $subject$ ” and “ $?Y$ ” to represent the  $object$  accordingly. The  $Select$  operator is used for a condition requiring arithmetic, Comparison, or Boolean operators to be utilized along with constants or strings. These constants or strings are then source replaced with the schema name.

#### 4.3.1.4 Generalization:

Generalization involves extracting common characteristics from one or more classes and combining them into a generalized superclass. It is used to establish hierarchies of classes and subclasses in ascending order, up to the most defined level. For this research, however,

generalization is only applied to reach Level 1 in the hierarchy. The syntax for the generalization operator is as follows.

$$\mathbf{Gen}_{(rdfs:class(?class),n-level)}(Source)$$

As established, any RDF triple at a given point consists of three essential elements, *Subject*, *Predicate* and *Object*. Note that the *Subject* and *object* elements always represent classes or subclasses within a system. The Generalization operator is then used upon these elements to extract the parent class of a mentioned class (**?class**) up to the first level in the hierarchy, i.e. Level 1. The source is then replaced with a schema.

#### 4.3.1.5 Specialization:

The existing Specialization operator is the complete opposite of what constitutes Generalization. Where generalization was used in the research to retain the parent class from a subclass, the specialization operator is utilized in creating new subclasses from an existing class. In this research, the authors have gone from the bottom up to Level 1.

**Syntax:**

$$\mathbf{Spec}_{(rdfs:class(?class),n-level)}(Schema)$$

RDF triple has only three elements, namely: the *Subject*, *Predicate* and *Object*. Consecutively, the subject and object elements represent classes or subclasses within an RDF database. The Specialization operator is then used to dispense the Subject and Object to extract child classes up to 1 level of a mentioned class (**?class**). Finally, the source is replaced with the schema name, as with all the operators previously established.

## 4.4 Algorithms

This section of the research examines and elaborates on the proposed framework algorithms and how these algorithms work in the conceptualized framework.

An algorithm is a particular technique for solving a well-defined computational issue. The advancement and analysis of algorithms are fundamental to all aspects of computer science: expert system, databases, graphics, networking, operating systems and security. Algorithm creation is more than simply programming. It needs to understand the alternatives available for solving a computational issue, consisting of the hardware, networking, programming language, and efficiency restrictions that accompany any specific solution. It also requires understanding what it indicates for an algorithm to be "correct" because it fully and effectively fixes the issue at hand. Computational intricacy is a continuum in that some algorithms require linear time (that is, the time necessary boosts directly with the number of items or nodes in the list, chart, or network being processed). In contrast, others require quadratic or perhaps exponential time to complete (that is, the time required boosts with the number of items squared or with the exponential of that number). At the back of this continuum lie the muddy seas of severe problems-- those whose options can not be effectively executed. Computer system researchers seek to find heuristic algorithms that can practically solve the issue and run in a sensible quantity of time (Rahim et al., 2017). The operational processes of every utilized algorithm have been evaluated in a step-by-step measure for maximum comprehension. The algorithms involved are used for converting queries and searching them in distributed ontologies. They are as follows.

#### 4.4.1 SPARQL Query into Algebraic expression

Through this algorithm, the author initiated converting the main SPARQL query into algebraic expressions for computerized comprehension. In the following, Table 4.1, the author initialises String and a list of models. The String variable holds the main SPARQL query and a list of the model used for holding the initialised models. Next, Function, transformToAlgebricForm, has been created, which receives the two parameters, initialised String and list of models. Next, a given SPARQL query has been created using the QueryFactory's method. After that pattern has been established for the SPARQL query, and a new object, Op, has been created to compile the SPARQL query and optimise the algebra expression. Next, declared a variable varMap as HashMap and allocated memory to process the SPARQL query into an algebraic expression. In the last, created n new object NodeTransform and called the function transformToAlgebricForm, which convert SPARQL query into semantic algebraic's expression. The algorithm steps are evident in table 4.1.

Algorithm 1. Translating SPARQL query into Algebraic expression	
Step1.	Initialise String for SPARQL query
Step2.	Initialise list of models
Step3.	Create Function transformToAlgebricForm, which receive queryString and model
Step4.	Create Query of given SPARQL query string using <i>create the</i> method of QueryFactory.
Step5.	Create the pattern element of created Query
Step6.	Create an Op object to compile the query.
Step7.	Optimize the Algebra expression

Step8.	Initialize variable varMap as <i>HashMap</i> and allocate memory to Vars and put those into varMap
Step9.	Create an object of NodeTransform with varMap
Step10.	Call Function transformToAlgebraicForm to get query into semantic algebraic form

**Table 4.1: Algorithm 1 - SPARQL query into Algebraic expression**

#### 4.4.2 Converting main SPARQL query into subqueries

The author converted the main SPARQL query into sub-queries directed to various ontologies to retrieve information in this algorithm. Each subquery was fired against its subsequent data set in order to identify and capture the information. The author creates a function *generateSubQry* which receive Linked Hash Map of triplePath and set of Strings containing required models. Next, initialised variable, parentModels, as a set of Model and assigned keySet of MaodelMap. After that, a new variable modelTripleMap has been declared with key Model and allocated value as a LinkedHashSet of TriplePath. In the end, For loop has been used to process the parentModels to generate the subqueries. All the algorithm steps are evident in table 4.2.

#### **Algorithm 2. Converting main SPARQL query into subqueries**

Step1.	Create function <i>generateSubQry</i> which receive Linked Hash Map of triplePath and set of Strings containing required model names
Step2.	Declare variable parentModels as Set of Model and assign keySet of ModelMap
Step3.	Declare variable modelTripleMap with key Model and value as LinkedHashSet of TriplePath

Step4. Declare variable triplesForModel as HashSet<TriplePath>

Step5     Begin For loop

get the key of entry into tripleName

get the value of entry into a set of String

if modelSet contains modelname

Add triplename to triplesForModel

end if

Step6. Save the model and triplesForModel to map modelTripleMap

Step7.     End of for

**Table 4.2: Algorithm `2 SPARQL query into Algebraic expression**

#### 4.4.3 Execution of SPARQL queries in distributed ontologies

Through this algorithm, the researchers executed the converted sub-queries. After converting the main SPARQL query into sub-queries, each subquery was fired against required data sets to capture information. The author creates a function `runqueryonModel`, which takes `modelTripleMap` and `modelcollection` as receiving parameters. Next, declared a model as a *parentmodel* and initialised a variable, `Map<String, String>subQryDetails` which holds the sub-queries details. The author used the Loop to process the `parentModel` to identify the required distributed sources and sent each sub-query to all identified resources to get the data. All the steps that make up this algorithm are displayed in Table 4.3.

**Algorithm 3 - Execution of SPARQL queries in distributed ontologies**

Step1. Create function runqueryonModel, which takes modelTripleMap and modelcollection as input

Step2. Declare parentmodel

Step3. Declare variable Map<String, String>subQryDetails

Step4. Begin loop // for each model existingModel from parentModel

Step5. Get the model name of existingModel and prefix of ExistingModel

Step6. Execute the query using queryExecution engine to receive the resultset of an executed query

Step7. if ResultSet has the next element,  
add model name and query to subQryDetails  
split the model name with “.” and store it into array fname  
create an object of file with “subquery” appended to fname

Step8. End if

Step9 End Loop

Step10 End function

**Table 4.3: Algorithm 3 Execution of SPARQL queries in distributed Ontologies**



#### 4.4.4 Combining results

This algorithm is formulated and utilized once every individual subquery has been fired against data sets to extract information. After this step, it is a vast requirement that the results returned by the sub queries are merged and produced into a single result to display to the main SPARQL query, which is essentially what this algorithm represents. The author creates a function *runQueryonModels* which takes model collection and String, *querFinal*, as receiving parameters. Next created a function, *createReadableIndex*, and initialised Map variable, which holds the details of the sub-queries results. Next, started the For Loop, which processes each model from the *parentModel* and combines all sub-queries result into one result. All the steps of the algorithm are entailed below in Table 4.4.

##### Algorithm 4 - Combining results

Step 1 Create function *runQueryonModels(List<Model>modelCollection, String queryFinal)*

Step 2 get substring of query with index of select and last index

Step 3 Declare Function *ReadableIndex.createReadableIndex(FileFilter)*

Step 4 Declare variable *Map<String, String>subQryDetails*

*Map<String, String>subQryDetails = new HashMap<>();*

Step 5 Begin For loop –for each model *existingModel* from *parentModel*

Step 5.1. : get model name of *existingModel*

Step 5.2. :get the prefix of *ExistingModel*

Step 5.3. : Execute the query using *queryExecution* engine

Step 5.4. : get the resultset of an executed query

Step 5.5. : if *ResultSet* has next element

step 5.5.1. : add model name and query to subQryDetails and return it

step 5.5.2. : split the model name with “.” and store it into array fname

step 5.5.3. : create an object of file with “subquery” appended to fname

step 5.5.4. : create fileoutputstream of above-mentioned file

step 5.5.5. : write above result to mentioned file usingResultSetFormatter

Step 5.6: end if

Step 6. Step close fileoutputstream and queryEngine.

Step 7. End loop.

Step 8. get Map<String, String>subQryDetails. i.e-list of subqueries

Step 9. combine subqueries with string append operation

Step 10. get the list of models

Step 11 iterate over each model and execute the appended query using query engine

Step 12 create an object of file writer and write query results to CSV file.

**Table 4.4: Algorithm 4. Combining results**

## 4.5 Chapter Summary

Chapter 4 concludes after an intensive discussion about various aspects involved in preparing the research for implementation. This chapter is sectioned into different parts, each of which explains the various concepts combined to give the reader an insight into how different parts of a database overlay to execute an almost negligible process that plays a significant role in their everyday internet life. The chapter explains the conceptual framework that allows the methodology to access data from distributed RDF sets and consequently satisfy the main objective of the research. It also discusses semantic algebra and elaborates upon how semantic algebra is carried out through its underlying operators. The chapter also discussed the concepts and working mechanisms involved in selecting projection, joining, specialization, and

generalization operators. Following the description of the operators, the chapter proceeds to inform the reader about the algorithms that the researchers fixed to execute conversions and translations within their proposed framework. As discussed in [Section 4.4](#), include the RDF index algorithm, converting the main SPARQL query into the sub-queries algorithm and joining the results algorithm. Chapter 4, thus, lays down the primary methodology in excruciating detail and gears the reader up for the testing and evaluation of these strategies. The testing of the proposed framework, as expressed in [Chapter 5](#), is to test the viability of the methodology.

# Chapter 5

## Framework Testing

### 5.1 Introduction:

In this chapter designed framework is tested by unit testing and functional testing techniques. Museum ontology is used to test and evaluate the developed system. The testing strategy used in this chapter to test the algorithms demonstrates how the complete developed and processed system works. In this chapter, different types of tests have been performed like algebraic operator's test (e.g., select, join, outer join, generalization, and specialisation operators test) and test the proposed algorithm. Tests showed that all developed system units worked as expected, and no errors found during the testing of all phases of the tested framework. The purpose behind the test is that the developed system should function and fulfil all the objectives specified in chapter 1 and perform what it is expected to do. Generally, testing has been performed throughout the development process to determine whether the developed system fulfils the specified requirements. Testing has been performed by running the whole functionality of the system. This ensures that the developed system fulfils the requirements. It can also be determined to show that the developed software satisfies its purpose when positioned in a specific environment. This process replies to the question, "Are we developing the right product or not?". With this unit testing technique, testing has become very much easier because each part or unit of the developed system has been tested first, and then the whole program has been tested. In unit testing, the author examined each phase of the developed system individually in a sequence.

The Apache Jena framework has been used for accessing the data from distributed RDF (Resource Descriptive Framework) data sets. The RDF is an elementary data model. It implements the semantic algebraic expressions, data dictionary, cache, conversion of main SPARQL query into sub-queries, and merging the results. Algebraic semantics involves the

algebraic specification of data and language constructs. The essential idea of the algebraic approach to semantics is to call the various kinds of objects and, therefore, the objects' operations and use algebraic axioms to explain their characteristic properties. An ontology model is an extension of the Jena RDF model, providing extra capabilities for handling ontologies.

Moreover, Jena provides an open platform to use both built-in and third-party inference engines. Based on RDFS and OWL ontology languages, Inference API provides “reasoners” that could be registered to the Model and produce additional resources on top of the asserted statements. Different operations have been performed on the RDF models.

Furthermore, it also discusses the methodology used behind the developed museum’s ontology which is used as a case study. We also used the Simplified Agile Methodology (SAMOD) methodology for Museum’s ontology Development. CRM (Conceptual Reference Model) has been used to develop the museum’s ontology, an ontology model for the social heritage domain, developed by the "COM/CIDOC" Standards Group. All these different techniques and methodologies are used to output the result. The primary research is on the museum and creation of museum ontology utilizing Protégé, a functional ontology for creating and testing the distributed query methods. Protégé supports different platforms, the extension of different unique interfaces has linked the "open knowledge base connectivity" (OKBC) model. It has the power to work as RDBMS, RDF, and XML. Many research groups and individuals are part of this tool. By using these different techniques, gathered data is divided into classes and subclasses by their properties. This chapter presents the process of the implementation of the proposed framework. It holds and supplies all information about a case study that is applied for comparison: Museum, which is used to demonstrate all the stages of the proposed framework. The chapter includes the implementation and details about how to convert SPARQL queries into sub-queries and combining results. Moreover, here data

dictionary has been used to store the data. Data Dictionary reference is utilized as a focal territory as it stores all ordered data from all RDF data sources. Data dictionary reference holds data about the subject, object, predicate, property, sub-property, classes, and subclasses.

## **5.2 Comparison of Unit and Functional Testing**

The author used the unit testing and functional testing technique as the proposed framework have different individual units which work together. Other testing techniques exist as well, e.g., functional testing, integration testing, system testing, regression testing, acceptance testing, component testing, performance testing. Before selecting testing techniques, the author compared chosen testing strategies with other testing techniques as every testing method has its advantages and disadvantages. However, most software testing unit testing is used because this type of testing is beneficial in the debugging process of the software (Sam, 2019). The main difference between unit testing and functional testing is that functional testing is conducted based on the client's point of view, and unit testing is based on the programmer's point of view. Therefore, unit testing is more helpful for programmers to understand the software's logic compared to functional testing. The major difference between unit testing and integration testing is that integration testing includes testing multiple parts of the software with a direct effect on each other, and in the case of unit testing, every unit of software is tested without any interruption in the working of any other unit under testing. This shows that unit testing is far more effective in debugging and modification of software than integration testing. In comparing unit testing with system testing, unit testing conducts testing on small modules or units of the software (Divyani Shivkumar Taley, 2020). In contrast, in the case of system testing, the software is tested as a whole to examine that it is functioning correctly or not. The main difference between unit and regression testing is that unit testing performs tests on small program units.

In contrast, regression testing is the combination of both integration testing and unit testing. Regression testing is more costly than unit testing, as it is a collection of both unit and integration testing. The significant deviation between unit testing and acceptance testing is that acceptance testing determines whether the software fulfils all the requirements of the user or not. Acceptance testing is more tiring

than unit testing. We are now comparing unit testing with automated unit testing (Mohammad Shahabuddin and Prasanth, 2016). Automated is almost like unit testing but functions without the participation of human beings. Unit testing is better than automated unit testing because it involves the participation of human beings and is also cost-effective. The main difference between unit testing and component testing is that every individual modules or component's functionality is tested in component testing. All the parts are replaced by the natural objects of all the classes. Component testing is more complex than unit testing. Now we are coming to the comparison of unit testing and end to end testing. End to end testing is different from unit testing. In this testing, the software is tested in a single piece as if the user is using that software. This method is helpful when a programmer wants to observe the working of software from the user's end. However, in end-to-end testing, debugging is very difficult and complex, whereas, in unit testing, errors and bugs can be located in no time (Anwar and Kar, 2019). Now a significant deviation between unit testing and performance testing. Performance testing is used when the developers want to analyze how the software reacts under high load. This testing is usually used to check the sustainability of the software and is nonfunctional. Both unit testing and performance testing different from each other and have different features. The main difference between unit testing and smoke testing is that smoke testing includes fast and elementary tests to check the software's functionality. This type of testing is usually beneficial for newly developed software. Smoke testing sometimes becomes more expensive than unit testing.

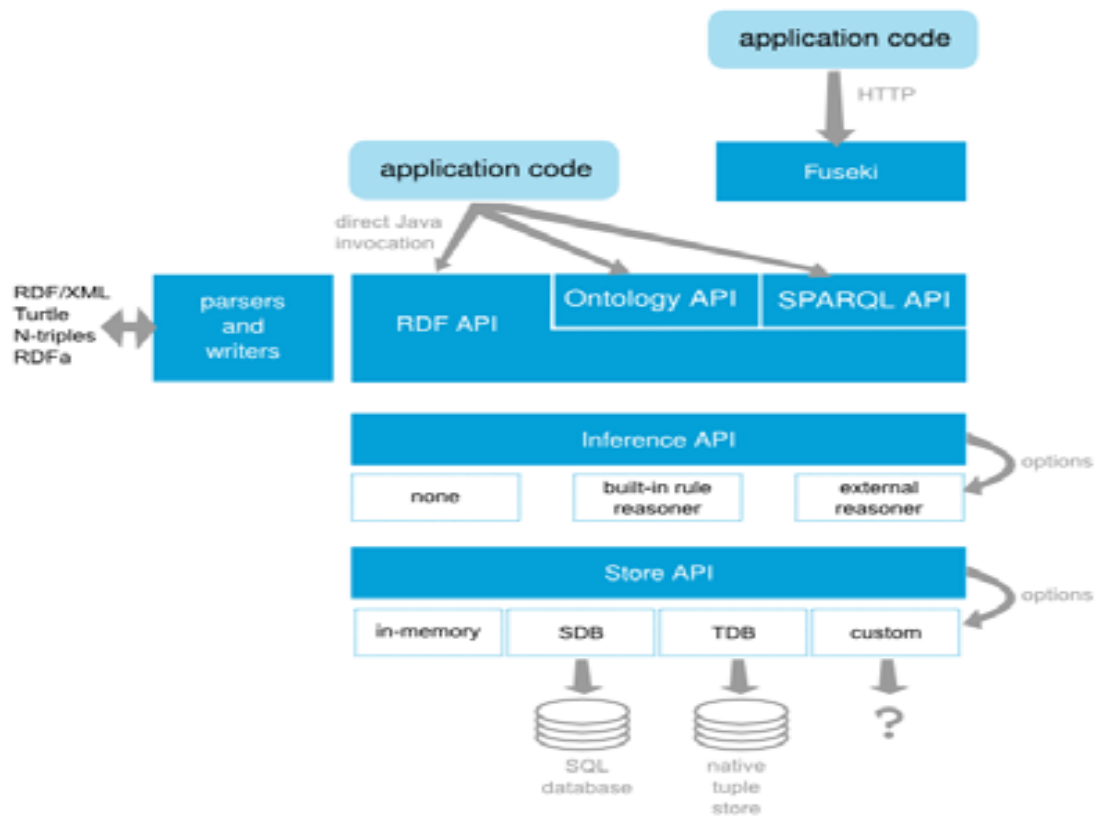
We are now analyzing the differences between unit testing and exploratory testing. In exploratory testing, the subsequent versions of the software are tested for bugs and errors. This testing assures that any previous bugs do not occur in the following versions. This testing is more valuable than unit testing, as it develops creativity and experience in the software engineers and developers. We are now comparing the unit testing with scripted testing. Scripted testing is one of the most well-known testing methods. In this type of testing, the tester writes a script or a path. This script is followed to test the software according to the specifications written in the script or path (Anwar and Kar, 2019). Besides these characteristics,

unit testing is still better than scripted testing because it has a shortcoming that we can only get the desired result due to the specifications of the script or path. Suppose we compare our chosen testing technique, unit testing, with modular testing deeply. Both unit testing and modular testing are almost the same. Besides this, these two methods of software testing have some differences. Usually, the software engineers of the software conduct unit testing. However, in the case of modular testing, the tests are conducted by another tester. Unit testing requires less pricey than modular testing, as no separate tester is required.

### **5.3 Jena Framework**

The author used the Apache Jena framework for a java programming language for developing semantic web in the form of java libraries. It helped the author to manage the various semantic components of the semantic web and linked-data application to conform to the standards of the W3C. Since 2000, Jena is an open-source project developed by researchers at HP Laboratories in Bristol city in the UK and later became popular in used widely (Jani and Dr. V.M. Chavda, 2011). It was a success to become part of the Apache Software Foundation in November of the year 2010. The Apache Jena architecture is shown in figure 5.1.





**Figure 5.1 - Apache Jena framework<sup>9</sup> (Jani and Dr. V.M. Chavda, 2011)**

Data in the Jena framework is structured in sets of RDF triples called *RDF Graph*. An RDF graph is simply a set of triples (S, P, O), where P names a binary predicate over (S, O)

Jena supports several serialization languages like RDF/XML, N3, N-triple and turtle. It also has an option for memory, file-based or database RDF persistence. Jena architecture provides different persistent, inference RDF, Ontology, Query and related API's that could be invoked using Java programming language and over the web using HTTP and SPARQL query language (Jani and Dr. V.M. Chavda, 2011).

**Ontology API:** Jena is based on RDF data structure, and the choice for ontology languages is restricted to compatibility with RDF. The most straightforward ontology language compatible with RDF is *RDF Schema*. (RDFS). Jena is also compatible with the three different OWL ontology language levels- OWL Lite, OWL DL, OWL Full. Jena Ontology API provides a language-neutral interface that can use a *profile* to set specific java classes and properties. For

instance, the URI for “ObjectProperty” in DAML (DARPA Agent Markup Language) profile is `daml: ObjectProperty`, while OWL is `owl: ObjectProperty`. The same URI in RDFS is null as there is no “ObjectProperty” implementation in the RDFS profile. Jena accepts the essential characteristic of polymorphism at the RDF level by considering that the Java abstraction (`OntClass`, `Restriction`, `DatatypeProperty`) is just a view or *facet* of the resource (Jani and Dr. V.M. Chavda, 2011).

. For example, if we declare a resource `#DigitalCamera` as an ontology class, a java instance of `OntClass` could represent.

```
<owl:Classrdf:ID="DigitalCamera">
</owl:Class>
```

This same resource can be an OWL *Restriction* that proves no unique mapping between RDF resources and Java abstraction.

```
<owl:Classrdf:ID="DigitalCamera">
    <rdf:typeowl:Restriction/>
</owl:Class>
```

Jena provides `as()` method to create a new facet on run-time depending on the resource property. The following example creates a resource (`res`) and instantiates two facets of the same resource that shows the flexibility of Jena in managing `Ontology`.

```
Resource res = myModel.getResource( myNS + "DigitalCamera" );
OntClass cls = res.as(OntClass.class);
Restriction rest = cls.as(Restriction.class);
```

An ontology model is an extension of the Jena RDF model, providing extra capabilities for handling ontologies. Ontology models are created through the `Jena ModelFactory`

```
// create ontology model
```

```
Model model = ModelFactory.createOntologyModel();
```

This creates an ontology with the following default settings:

- OWL-Full language
- In-memory storage
- RDF inference (i.e. entailments from sub-class and sub-property)

If the ontology model is for a simple model display, then inferencing is unnecessary, and a model should be created with no reasoned (OWL\_MEM) (Siddiqui and Alam, 2011).

```
OntModel model = ModelFactory.createOntologyModel( OntModelSpec.OWL_MEM  
);
```

To create an ontology model with a built in or custom specification ModelFactory should be invoked as follows.

```
OntModel m = ModelFactory.createOntologyModel( <model spec> );
```

In OWL a meta-data about the ontology can be set in the ontology using owl:Ontology.

```
<owl:Ontology rdf:about="SpaceKnowledgeManagement">  
    <rdfs:comment>SpaceManagement Ontology</rdfs:comment>  
    <rdfs:label>MappingRelational Databases</rdfs:label>  
    <owl:versionInfo>1.0</owl:versionInfo>  
</owl:Ontology>
```

The resources represented in the Ontology create a taxonomical hierarchy. The relationship between the different classes, properties, relations, restrictions, axioms create a direct (*asserted*) and indirect (*inferred*) link amongst the Ontology components.

The distinction between the asserted and inferred relationships helps to organise the Ontology into “facts” and “deductions”. Jena’s listRDFTypes() is one of the methods to list different types of resources in the Ontology.

```
// Shows direct relationships only if direct=true,
```

```
// else shows indirect relationships
```

```
listRDFTypes( boolean direct )
```

**Reasoners:** Jena provides an open platform to use both built-in and third-party inference engines. Based on RDFS and OWL ontology languages, Inference API provides “*reasoners*” that could be registered to the Model and produce additional resources on top of the asserted statements. ModelFactory is used to associate reasoners with a Model. The inference is implemented at the Graph SPI level so the different model interfaces could share the result (Jani and Dr. V.M. Chavda, 2011). Ontology API provides OntModelS to link reasoners to models. Jena also provides InfModel, an extension of the Model that provides additional control over the underlying Graph. Methods like createRDFSModel provide built-in RDFS inference rules with basic implementation. For different built-in and generic reasoning systems, Reasoners are required. ReasonerRegistry static class is used to register reasoners dynamically, ranging from built-in *transitive*, *RDFS*, *OWL* to *generic user-defined* rule reasoners.

- getOWLReasoner(): prebuilt standard OWL inference reasoner
- getRDFSReasoner(): prebuilt standard RDFS inference reasoner
- getTransitiveReasoner(): prebuilt subclass and subproperty transitive closure reasoner
- Generic User-defined: different forward/backward chaining and hybrid executions

The example below shows an excerpt of a Jena inference implementation using *OWL Ontology Schema*, *RDF Data* and a built-in *OWL*.

```
//Read Ontology
```

```
Model schema = FileManager.get().loadModel("file:source/mappedSchema.owl");
```

```
//Read Data
```

```
Model data = FileManager.get().loadModel("file:source/mappedData.rdf");
```

```

//Get built-in OWL Reasoner

Reasoner reasoner = ReasonerRegistry.getOWLReasoner();

//Bind Reasoner to Ontology

reasoner = reasoner.bindSchema(schema);

//Create Model using Reasoner

InfModel infmodel = ModelFactory.createInfModel(reasoner, data);

```

A similar Jena implementation below shows an inference program excerpt using *user-defined* rule instead of a built in reasoner.

```

//Read Ontology

Model schema = FileManager.get().loadModel("file:source/mappedSchema.owl");

//Read Data

Model data = FileManager.get().loadModel("file:source/mappedData.rdf");

/* Set User-defined rule*/

String ruleString = [ transitiveChainSubClassOf: (?xrdfs:subClassOf ?y),(?y
rdfs:subClassOf ?z) -> (?x rdfs:subClassOf ?z) ];

//Parse Rule

List rules = Rule.parseRules(ruleString);

//Create User-defined Reasoner

Reasoner reasoner = new GenericRuleReasoner(rules);

//Bind Reasoner to Ontology

reasoner = reasoner.bindSchema(schema);

//Create Model using Reasoner

InfModel inf = ModelFactory.createInfModel(reasoner, data);

```

## 5.4 Ontology Development Methodology

Museum ontology has been developed, and Simplified Agile Methodology (SAMOD) methodology has been adopted for Museum's ontology Development ([Appendix A](#)) to test the developed framework. SAMOD focuses on iterative tests to ensure that the developed ontology is consistent and matches the requirements. These tests have been performed on this ontology, and these tests are model tests, data tests and query tests. This methodology is very lightweight, and it has the following three stages (Abdelghany, Darwish and Hefni, 2019)

- Understanding the requirements
- Merging the Ontology
- Refactoring the main ontology branch

## 5.5 Ontology Justification - Virtual Museum Exhibition

CRM (Conceptual Reference Model) has been used to industrial museum's ontology, an ontology model for the societal legacy sphere, and it is developed by "COM/CIDOC" Standards Group. The author employed the CIDOC Conceptual Reference Model (CRM), a theoretical and practical technique for information integration within cultural heritage. It can help scientists, controllers, and the public check out complex queries regarding our history across numerous and distributed datasets (Gaitanou and Gergatsoulis, 2012). The CIDOC CRM achieves that by simply providing meanings and a proper design for explaining the implicit and specific concepts and relationships employed in cultural heritage documentation and vital interest for querying and exploring such details. Such designs are also described as formal ontologies. These formalistic explanations enable the combination of information from several sources in software and schema-agnostic fashion. It emphasizes concepts and connections in an object-oriented way of cultural domain. It covers a profound

measure of events, antiques and people recognized in the museum domain (Gaitanou and Gergatsoulis, 2012).

[Appendix A](#) represents the distributed historical museum ontology utilizing protégé to test the distributed environment. This museum case study has been selected and developed for the demonstration as it provides the flexibility to demonstrate all developed proposed framework's operations. Protégé support different platforms; an extension of different unique interfaces has linked the "open knowledge base connectivity" (OKBC) model. It has the power to work as RDBMS, RDF and XML. Many research groups and individuals are part of this tool.

## 5.6 Framework Testing

In this section, the author tested proposed algorithms against different predefined selected cases. [Appendix E](#) presents the screenshots of different phases of testing. We have different phases in our proposed architecture. In the first phase, we indexed all participated data sets in a local and centralised server. In phase 2, we converted our main SPARQL query into an algebraic expression. In phase 3, the local cache holds the information about the subject, predicate, object, and filters. In phase 4, we identified the required data sets repositories required to fetch the data with the help of a temporary cache. In Phase 5, we converted our main SPARQL query into subqueries. In phase 6, we sent each subquery to different data sets repository, which we identified in the local cache. In phase 7, we combined the sub queries results and produced the required output. All phases are clear with each case.

Following the list of cases, the author tested against the proposed algorithms.

1. Case 1: Parent class is a child class in another repository
2. Case 2: Child class is a parent class in another repository
3. Case 3: Parent property is a sub Property in another repository
4. Case 4: sub Property is a parent property in another repository
5. Case 5: Subject is an abject in another repository

6. Case 6: Object is a subject in another repository
7. Case 7: Repository 1's property, P1, between subject and an object is a sub Property of Property, P2, in another repository between Subject 1 of the first repository and Object 2 of another repository
8. Case 8: Property, P2, in the second repository is a sub Property of repository 1's property, P1, between repository 2's subject and repository 1's object
9. Case 9: Property, P2, in the second repository is a sub Property of repository 1's property, P1, between repository 2's subject and repository 1's object, and repository 2-s subject is a subclass of repository 1's subject

One data dictionary is created after indexing all participated rdf files, which holds the index information about all rdf files and help locate the appropriate rdf resources and produce inner queries built on recognised rdf sources.

The data dictionary is used as a central point as it stores all indexed information from all RDF repositories. Data dictionaries hold information about subject, object, predicate, property, subProperty, classes and subclasses.

**Case 1:** Parent class is a child class in another repository.

**Query 1:** Show all paintings where Artefact's craft is wood

Firstly, Algebraic notions are produced of the main SPARQL query, and a further cache is created to store Algebraic notions information, which helps name the required RDF files. For instance, the following SPARQL query, table 7, is for case 1 and is converted to Algebraic notions

```
PREFIX m:<http://allahm.museum.org/museum#>

SELECT ?painting ,?Artefacts, ?craft

WHERE {

    ?artefacts rdf:represented-by m:Craft
```





?Artefact	rdf:represented-by	?craft			
?wood	rdf:is-a	?craft			(?craft "wood")

**Table 5.3: Cache**

**Identifying sources:** Cache's predicates are searched inside the data dictionary to identify required data repositories. As we can see in the following cache, table 5.4, a new column, data source, which is added after identification of required sources

Subject	Predicate	Object	Conditions	Special ization	Generali zation	Data Source
?Artefact	rdf: represented-by	?craft				Ds 1, Ds 2
?wood	rdf:is-a	?craft	(?craft "wood")			Ds 1, Ds 2

**Table 5.4: Identifying sources**

**Subqueries and merging results:** According to identified data sources, subqueries are generated. From the above table 5.4, we can see that the required subject and object exist inside the data sources mentioned in the data source column. Each subquery is sent to the identified data source to get the required data, and then results are combined through the union. As stated in the data source column, the following subqueries, table 5.5, are generated.

<u>Sub Query for Data Source 1</u>	<u>Sub Query for Data Source 2</u>
SELECT ?artefact ?painting ?craft	SELECT ?artefact ?painting ?craft

WHERE {  ?artefactrdf:represented-by m:craft  FILTER {?craft,wood"}  }	WHERE {  ?artefactrdf:represented-by m:craft  FILTER {?craft,wood"}  }
--	--

**Table 5.5: subqueries**

**Case 2:** Child class is a parent class in another repository

**Query 2:** Show parent details of all artists where the artist wrote handwritten documents.

Firstly, Algebraic notions are produced from the main SPARQL query, and a cache is created to store Algebraic notions information, which helps name the required RDF files. For instance, the following SPARQL query, table 5.6, is for case 2 and is converted to Algebraic notions

PREFIX m:<http://allahm.museum.org/museum#>  SELECT ?Parent, ?father, ?mother ?artist, ?writer, ?HandwrittenDocuments  WHERE { <b>?father : rdfs:subClassOf :parent.</b>  <b>?mother : rdfs:subClassOf :parent.</b>  <b>?writer : rdfs:subClassOf :artist.</b>  ?artist rdf:hasParents m: ?Parents  ?writer rdf:writes m: ?HandwrittenDocuments  }
---

**Table 5.6: case 2 SPARQL query**

**Algebraic notions:** Above case 2 SPARQL query is converted into the following algebraic expression, table 5.7.

(	Π	(	?Parent, ?artist, ?HandwrittenDocuments)	)
(	⋈			)
(	σ			)
			$Spec_{(rdfs:class(?parent)1)}$	
			$Spec_{(rdfs:class(?artist)1)}$	
			?artist rdf:hasParents m: ?Parents	
			?writer rdf:writes m: ?HandwrittenDocuments	
		)		
)				

**Table 5.7: case 2 algebraic notation**

**Cache:** the cache is utilized To store information about the subject, object and predicate. For instance, the following table 5.8 is the example of cache for case 2.

Subject	Predicate	Object	Specialization	Generalization	Condition
? Artist	rdf:hasParents	? Parents	Artist,1: Parents,1		
?Writer	rdf: writes	? HandwrittenDocuments			

**Table 5.8: case 2 cache**

**Identifying sources:** Cache's predicates are searched inside the data dictionary to identify required data repositories. As we can see in the following cache, table 5.9, a new column, data source, is added after identifying required sources.

Subject	Predicate	Object	Specialization	Generalization	Conditions	DS
? Artist	rdf: hasParents	? Parents	Artist, 1: Parents, 1			Ds 1, Ds 2
? Writer	rdf: writes	? HandwrittenDocuments				Ds 1, Ds 2

**Table 5.9: case 2 identifying resources**

**Subqueries and merging results:** According to identified data sources, subqueries are generated from the above table 5.9. We can see that compulsory subject and object exist inside data sources mentioned in the data source column. Each subquery is sent to the identified data source to get the required data, and then results are combined through the union. As stated in the data source column, the following subqueries, table 5.10, are generated.

<u>Sub Query for Data Source 1</u>	<u>Sub Query for Data Source 2</u>
SELECT ?Parent, ?father, ?mother ?artist, ?writer, ?HandwrittenDocuments	SELECT ?Parent, ?father, ?mother ?artist, ?writer, ?HandwrittenDocuments

WHERE {  ?father : rdfs:subClassOf :parent.  ?mother : rdfs:subClassOf :parent.  ?writer : rdfs:subClassOf :artist.  ?artistrdf:hasParents m: ?Parents  ?writerrdf:writes m: ?HandwrittenDocuments  }	WHERE {  ?father : rdfs:subClassOf :parent.  ?mother : rdfs:subClassOf :parent.  ?writer : rdfs:subClassOf :artist.  ?artistrdf:hasParents m: ?Parents  ?writerrdf:writes m: ?HandwrittenDocuments  }
---	---

**Table 5.10: case 2 sub-queries and merging results**

**Case 3:** Parent property is a subProperty in another repository

**Query 3: Show all museums addresses where the city is London**

Firstly, Algebraic notions are produced of the main SPARQL query, and a further cache is created to store Algebraic notions information, which helps name the required RDF files. For instance, the following SPARQL query, table 5.11, is for case 3 and is converted to Algebraic notions

PREFIX m:<http://allahm.museum.org/museum#>  SELECT ?museum , ?address ,?city  WHERE {  ?Museum rdf:hasAddress m:Address  ?Place rdf: hasCity ?city  FILTER {city ,"London"}
--

}
---

**Table 5.11: case 3 SPARQL query**

**Algebraic notions:** Above case 3 SPARQL query is converted into the following algebraic expression, table 5.12.

( [ []?museum , ?address ,?place,?city)
( ⋈
(σ
(?Museum rdf:hasAddress m:Address)
(?Place rdf: hasCity ?city)
(?City ,"London")
)
))

**Table 5.12: case 3 algebraic notation**

**Cache:** the cache is used to store information about the subject, object and predicate. For instance, the following table 5.13 is the example of cache for case 3.

Subject	Predicate	Object	Specializati on	Generalizati on	Conditions
?Museu m	rdf: hasAddress	?Addres s			
?Place	rdf: hasCity	?City			(?City "London")

**Table 5.13: case 3 cache**

**Identifying sources:** Cache's predicates are searched inside the data dictionary to identify required data repositories, as we can see in table 5.14, cache new column, data source, which is added after identifying required sources.

Subject	Predicate	Object	Specializati on	Generalizati on	Condition s	Data Source
?Museum	rdf: hasAddress	?Address				Ds 1 , Ds 2
?Place	rdf: hasCity	?City			(?City "London")	Ds 1 , Ds 2

**Table 5.14: case 3 identifying sources**

**Subqueries and merging results:** According to identified data sources, subqueries are generated. From the above table 20, we can see that the required subject and object exist inside the data source column's data sources. Each subquery is sent to the identified data source to get the required data, and then results are combined through the union. As stated in the data source column, table 5.14, 2 following subqueries are generated as shown in table 5.15.

<u>Sub Query for Data Source 1</u>	<u>Sub Query for Data Source 2</u>
SELECT ?Museum ?Address ?City WHERE { ?Museumrdf:hasAddress m:Address FILTER {?City ,"London"}	SELECT ?Museum ?Address ?City WHERE { ?Museumrdf:hasAddress m:Address FILTER {?City ,"London"}



}	}
---	---

**Table 5.15: case 3 subqueries**

**Case 4:** subProperty is a parent property in another repository.

**Query 4:** Show parent details of all artists where parent's beliefs are Christianity

Firstly, Algebraic notions are produced of the main SPARQL query, and a further cache is created to store Algebraic notions information, which helps to name the required RDF files. For instance, following table 4.16, the SPARQL query is for case 4 and is converted to Algebraic notions

```

PREFIX m:<http://allahm.museum.org/museum#>

SELECT ?Parents, ?father, ?mother ?artist, ?beliefs

WHERE {

  ?father : rdfs:subClassOf :parents.

      ?mother : rdfs:subClassOf :parents.

  ?artist rdf:hasParents m: ?Parents

      ?parents rdf:hasBeliefs m: ?Beliefs

      FILTER { ? Beliefs ,"Christianity" }
```

**Table 5.16: case 4 SPARQL query**

**Algebraic notions:** Above case 4 SPARQL query, table 5.16, are converted into the following algebraic expression, table 5.17.

```

( [ [ (?Parents, ?father, ?mother ?artist, ?beliefs)

  ( ⋈
```

( $\sigma$
$Spec_{(rdfs:class(?parent)1,)$
$Spec_{(rdfs:class(?artist)1,)$
?artist rdf:hasParents m: ?Parents
?parents rdf:hasBeliefs m: ?Beliefs
? Beliefs ,"Christianity"
)
))

**Table 5.17: case 4 algebraic notation**

**Cache:** The cache is used to store information about the subject, object and predicate. For instance, the following table 5.18 is an example of cache for case 4.

Subject	Predicate	Object	Specialization	Generalization	Conditions
? Artist	rdf:hasParents	? Parents	Parents,1		
?Parent s	rdf:hasBeliefs	?Beliefs			Christianity

**Table 5.18: case 4 cache**

**Identifying sources:** Cache's predicates are searched inside the data dictionary to identify required data repositories. As we can see in the following cache, table 5.19, a new column, data source, is added after identifying required sources.

Subject	Predicate	Object	Specialization	Generalization	Conditions	DS
? Artist	rdf:hasParents	? Parents	Parents,1			Ds 1, Ds 2
?Parents	rdf:hasBeliefs	? Beliefs				Ds 1, Ds 2

**Table 5.19: case 4 identifying sources**

**subqueries and merging results:** According to identified data sources, subqueries are generated from the above table 5.19, and we can see that the required subject and object exist inside the data source column's data sources. Each subquery is sent to the identified data source to get the required data, and then results are combined through the union. As stated in the data source column following subqueries are generated; table 5.20.

**Sub Queries:**

<u>Sub Query for Data Source 1</u>	<u>Sub Query for Data Source 2</u>
SELECT ?Parents, ?father, ?mother ?artist, ?beliefs WHERE { <b>?father : rdfs:subClassOf :parents.</b> <b>?mother : rdfs:subClassOf :parents.</b> ?artistrdf:hasParents m: ?Parents ?parentsrdf:hasBeliefs m: ?Beliefs	SELECT ?Parents, ?father, ?mother ?artist, ?beliefs WHERE { <b>?father : rdfs:subClassOf :parents.</b> <b>?mother : rdfs:subClassOf :parents.</b> ?artistrdf:hasParents m: ?Parents ?parentsrdf:hasBeliefs m: ?Beliefs

FILTER { ? Beliefs , "Christianity" }          }	FILTER { ? Beliefs , "Christianity" }          }
---	--

**Table 5.20: case 4 subqueries**

**Case 5:** Subject is an abject in another repository

**Query 5:** Show all exhibition’s artefacts where used craft is an oil painting, and material is gold

Firstly, Algebraic notions are produced of the main SPARQL query, and a further cache is created to store Algebraic notions information, which helps name the required RDF files. For instance, the following SPARQL query, table 5.21, is for case 5 and converted to algebraic notions.

PREFIX m:<http://allahm.museum.org/museum#>  SELECT ?exhibition, ?artefacts, ?craft, ?material  WHERE { ?exhibition rdf:contains m: ?artefacts ?artefacts rdf:hasMaterial m: ?material ?artefacts rdf:representedBy m: ?craft FILTER { ? craft, "OilPainting" } { ? material, "gold" } }
--

**Table 5.21: case 5 SPARQL query**

**Algebraic notions:** Above case 5 SPARQL query, table 5.21, is converted into the following algebraic expression, table 5.22.

( $\Pi$ (?exhibition, ?artefacts, ?craft, ?material)
( $\bowtie$
( $\sigma$
?exhibition rdf:contains m: ?artefacts
?artefacts rdf:hasMaterial m: ?material
?artefacts rdf:representedBy m: ?craft
? craft, "OilPainting"
? material, "gold"
)

**Table 5.22: case 5 algebraic notation**

**Cache:** The cache is used to store information about the subject, object and predicate. For instance, the following is the example of cache, table 5.23, for case 5.

Subject	Predicate	Object	Specializati on	Generalizati on	Conditio ns
? exhibition	rdf: contains	? artefacts			
? artefacts	rdf: hasMaterial	? material			gold
? artefacts	Rdf: representedBy	?craft			oilPaintin g

**Table 5.23: case 5 cache**

**Identifying sources:** Cache's predicates are searched inside the data dictionary to identify required data repositories, as we can see in the following cache, table 5.24, a new column, data source, which is added after identifying required sources.

Subject	Predicate	Object	Specializat ion	Generalizat ion	Condi tions	DS
? exhibitio n	rdf: contains	? artefacts				Ds 1, Ds 2
? artefacts	rdf: hasMaterial	? material			gold	Ds 1, Ds 2
? artefacts	Rdf: representedBy	?craft			oilPainti ng	Ds 1, Ds 2

**Table 5.24: case 5 identifying sources**

**Subqueries and merging results:** According to identified data sources, subqueries are generated. From the above table 5.24, we can see that the required subject and object exist inside the data source column's data sources. Each subquery is sent to the identified data source to get the required data, and then results are combined through the union. As stated in the data source column, table 5.24, 2 following subqueries are generated as shown in table 5.25.

## Sub Queries

<u>Sub Query for Data Source 1</u>	<u>Sub Query for Data Source 2</u>
<pre> SELECT    ?exhibition, ?artefacts, ?craft, ?material WHERE { ?exhibitionrdf:contains m: ?artefacts ?artefactsrdf:hasMaterial m: ?material ?artefactsrdf:representedBy m: ?craft          FILTER {? craft, "OilPainting"}                 {? material, "gold"}  }</pre>	<pre> SELECT    ?exhibition, ?artefacts, ?craft, ?material WHERE { ?exhibitionrdf:contains m: ?artefacts ?artefactsrdf:hasMaterial m: ?material ?artefactsrdf:representedBy m: ?craft          FILTER {? craft, "OilPainting"}                 {? material, "gold"}  }</pre>

**Table 5.25: case 5 subqueries**

**Case 6:** Object is a subject in another repository.

**Query 6:** Show all details of museums management who manage the exhibition

Firstly, Algebraic notions are produced of the main SPARQL query, and a further cache is created to store Algebraic notions information, which helps name the required RDF files. For instance, the following SPARQL query, table 5.26, is for case 6 and is converted to Algebraic notions.

```

PREFIX m:<http://allahm.museum.org/museum#>

SELECT ?museum, ?exhibition, ?management
WHERE {
```

```

?museumrdf:hasManagement m: ?Management
?managementrdf:manages m: ?Exhibition
}

```

**Table 5.26: case 6 SPARQL query**

**Algebraic notions:** Above case 6 SPARQL query, table 5.26, is converted into the following algebraic expression, as we can see in table 5.27.

```

( [ ( ?Parent, ?artist, ?HandwrittenDocuments)
  ( ⋈
    (σ
      ?museum rdf:hasManagement m: ?Management
      ?management rdf:manages m: ?Exhibition
    )
  )
)

```

**Table 5.27: case 5 algebraic notation**

**Cache:** The cache is used to store information about the subject, object and predicate, for instance, the following. Table 5.28 is the example of cache for case 6.

Subject	Predicate	Object	Specializati on	Generalizati on	Conditio ns
?museum	rdf: hasManagement	? Managemen t			
?Managemen t	rdf: manages	? exhibition			

**Table 5.28: case 6 cache**



**Identifying sources:** Cache's predicate are searched inside the data dictionary To identify required data repositories. As we can see in the following cache, table 5.29, a new column, data source, is added after identifying required sources.

Subject	Predicate	Object	Specialization	Generalization	Conditions	DS
?museum	rdf:hasManagement	?Management				Ds 1, Ds 2
?Management	rdf:manages	?exhibition				Ds 1, Ds 2

**Table 5.29: case 6 identifying sources**

**Subqueries and merging results:** According to identified data sources, subqueries are generated from above table 5.29. We can see that the required subject and object exist inside the data source column's data sources. Each subquery is sent to the identified data source to get the required data, and then results are combined through the union. As stated in the data source column, table 5.29, 2 following subqueries, table 5.30, is generated.

<u>Sub Query for Data Source 1</u>	<u>Sub Query for Data Source 2</u>
SELECT       ?museum,   ?exhibition, ?management WHERE { ?museumrdf:hasManagement       m: ?Management	SELECT ?museum, ?exhibition, ?management WHERE { ?museumrdf:hasManagement m: ?Management ?managementrdf:manages m: ?Exhibition



**Algebraic notions:** The above case 7 SPARQL query is converted into table 5.32, algebraic expression.

$\Pi (?Museum, ?Place, ?City ?Address)$ $(\bowtie$ $(\sigma$ $Spec_{(rdfs:class(?city)1,)}$ $? place \text{ rdf:hasCity } m: ?city$ $? museum \text{ rdf:hasAddress } m: ?address$ $? museum, "science"$ $)$ $))$
---

**Table 5.32: case 7 algebraic notation**

The cache is to store information about the subject, object and predicate. For instance, the following table 5.33 is the example of cache for case 7.

Subject	Predicate	Object	Specializati on	Generalizati on	Conditio ns
? Place	rdf: hasCity	? City	City,1		
?Museu m	rdf: hasAddress	? Address			science

**Table 5.33: case 7 cache**

**Identifying sources:** Cache's predicates are searched inside the data dictionary to identify required data repositories. As we can see in the following table 5.34, cache new column, the data source is added after identifying required sources.

Subject	Predicate	Object	Specializat ion	Generalizat ion	Condi tions	DS
? Place	rdf: hasCity	? City	City,1			Ds 1, Ds 2
?Museum	rdf: hasAddress	? Address			science	Ds 1, Ds 2

**Table 5.34: case 7 identifying sources**

**Subqueries and merging results:** According to identified data sources, subqueries are generated. From the above table 5.34, we can see that the required subject and object exist inside the data source column's data sources. Each subquery is sent to the identified data source to get the required data, and then results are combined through the union. As stated in the data source column, table 5.34, 2 following subqueries, table 5.35, are generated.

<u>Sub Query for Data Source 1</u>	<u>Sub Query for Data Source 2</u>
SELECT ?Museum, ?Place, ?City ?Address WHERE { ? address: rdfs: subClassOf: city. ? place rdf:hasCity m: ?city ? museum rdf:hasAddress m: ?address	SELECT ?Museum, ?Place, ?City ?Address WHERE { ? address: rdfs: subClassOf: city. ? place rdf:hasCity m: ?city ? museum rdf:hasAddress m: ?address

FILTER {? museum, "science"}  }	FILTER {? museum, "science"}  }
---------------------------------------	---------------------------------------

**Table 5.35: case 7 subqueries**

**Case 8:** Property, P2, in the second repository is a sub-property of repository 1's property, P1, between repository 2's subject and repository 1's object

**Query 8:** Show museum addresses of Europe region which holds the artefacts of Asian's artist. Firstly, Algebraic notions are produced of the main SPARQL query, and a further cache is created to store Algebraic notions information, which helps name the required RDF files. For instance, the following SPARQL query, table 5.36, is for case 8 and converted to algebraic notions.

```

PREFIX m:<http://allahm.museum.org/museum#>

SELECT ?Museum, ?Address, ?Region, ?Place, ?City, ?Country, ?Atrist,
?Atrefacts

WHERE {

  ? address: rdfs: subClassOf: city.

      ? place: rdfs: subClassOf: region.

  ? place rdf:hasCity m: ?city

      ? country rdf:hasCity m: ?city

      ? museum rdf:hasAddress m: ?address

      ? museum rdf:hasArtefacts m: ?artefacts

      ? artist rdf:hasCountry m: ?Country

```

<pre> FILTER {? region, "Europe"}        {? artist, "Asian"}      } </pre>
--

**Table 5.36: case 8 SPARQL query**

**Algebraic notions:** Above case 8 SPARQL query, table 5.36, is converted into the following algebraic expression.

<pre> (Π (?Museum, ?Address, ?Region, ?Place, ?City, ?Country, ?Atrist, ?Atrefacts)   ( ⋈     (σ       Gen(rdfs:class(?address)1,)       Gen(rdfs:class(?place)1,)       ? place rdf:hasCity m: ?city       ? country rdf:hasCity m: ?city       ? museum rdf:hasAddress m: ?address       ? museum rdf:hasArtefacts m: ?artefacts       ? artist rdf:hasCountry m: ?Country       { ? region, "Europe" }       {? artist, "Asian"}     )   )) </pre>
---

**Table 5.37: case 8 algebraic notation**

### Cache:

The cache is used to store information about the subject, object and predicate. For instance, the following table 5.38 is the example of cache for case 8.

Subject	Predicate	Object	Specialization	Generalization	Conditions
? Place	rdf: hasCity	? City		Address,1: Place,1	
?Country	rdf: hasCity	? City			
?Museum	rdf: hasAddress	?Address			
?Museum	rdf: hasArtefacts	?Artefacts			{? region, "Europe"}
?Artist	rdf: hasCountry	?Country			{? artist, "Asian"}

**Table 5.38: case 8 cache**

**Identifying sources:** Cache's predicates are searched inside the data dictionary To identify required data repositories. As we can see in the following cache, table 5.39, a new column, the data source, is added after identifying the required sources.

Subject	Predicate	Object	Specialization	Generalization	Conditions	DS
? Place	rdf: hasCity	? City		Address,1: Place,1		Ds 1, Ds 2
?Country	rdf: hasCity	? City				Ds 1, Ds 2

?Museum	rdf:hasAddress	?Address				Ds 1, Ds 2
?Museum	rdf:hasArtefacts	?Artefacts			{?region, "Europe"}	Ds 1, Ds 2
?Artist	rdf:hasCountry	?Country			{?artist, "Asian"}	Ds 1, Ds 2

**Table 5.39: case 8 identifying sources**

According to identified data sources, subqueries are generated. From the above table 5.39, we can see that the required subject and object exist inside the data sources mentioned in the data source column. Each subquery is sent to the identified data source to get the required data, and then results are combined through the union. As stated in data source column table 5.39, 2 following subqueries, table 5.40, are generated.

**Sub Queries:**

<u><i>Sub Query for Data Source 1</i></u>	<u><i>Sub Query for Data Source 2</i></u>
SELECT ?Museum, ?Address, ?Region, ?Place, ?City, ?Country, ?Artist, ?Artefacts WHERE { ?address rdfs:subClassOf city. ?place rdfs:subClassOf region. ?place rdf:hasCity m: ?city ?country rdf:hasCity m: ?city	SELECT ?Museum, ?Address, ?Region, ?Place, ?City, ?Country, ?Artist, ?Artefacts WHERE { ?address rdfs:subClassOf city. ?place rdfs:subClassOf region. ?place rdf:hasCity m: ?city ?country rdf:hasCity m: ?city



<pre> ? museum rdf:hasAddress m: ?address  ? museum rdf:hasArtefacts m: ?artefacts  ? artist rdf:hasCountry m: ?Country FILTER { ? region, "Europe" }         { ? artist, "Asian" } } </pre>	<pre> ? museum rdf:hasAddress m: ?address  ? museum rdf:hasArtefacts m: ?artefacts  ? artist rdf:hasCountry m: ?Country FILTER { ? region, "Europe" }         { ? artist, "Asian" } } </pre>
--	--

**Table 5.40: case 8 subqueries**

**Case 9:** Property, P2, in the second repository is a sub-property of repository 1's property, P1, between repository 2's subject and repository 1's object, and repository 2's subject is a subclass of repository 1's subject.

**Query 9:** Show all museum addresses of the European region, which holds the artefacts of Asian's artists who used oil painting craft for paintings.

Firstly, Algebraic notions are produced of the main SPARQL query, and a further cache is created to store Algebraic notions information, which helps name the required RDF files. For instance, the following SPARQL query, table 5.41, is for case 9 and converted to algebraic notions.

```

PREFIX m:<http://allahm.museum.org/museum#>

SELECT ?Painter,? Painting, ?Craft, ?Museum, ?Address, ?Region, ?Place, ?City, ?Country,
?Atrist, ?Atrefacts

WHERE {

? address: rdfs: subClassOf: city.

```

```

? place: rdfs: subClassOf: region.

? painter: rdfs: subClassOf: artist.

? oilpainting: rdfs: subClassOf: painting.

? oilpaitnng: rdfs: subClassOf: craft.

? painter rdf:draws m: ?painting

? place rdf:hasCity m: ?city

? place rdf:hasCity m: ?city

? country rdf:hasCity m: ?city

? museum rdf:hasAddress m: ?address

? museum rdf:hasArtefacts m: ?artefacts

? artist rdf:hasCountry m: ?Country

FILTER { ? region, "Europe" }

      { ? artist, "Asian" }

      { ? painting, "oilpainting" }      }

```

**Table 5.41: case 9 SPARQL query**

**Algebraic notions:** Above case 9 SPARQL query, table 5.41, is converted into table 5.42, algebraic expression.

```

(Π (?Painter, ? Painting, ?Craft, ?Museum, ?Address, ?Region, ?Place, ?City, ?Country, ?Atrist,
?Atrefacts)

( ⋈

(σ

Gen(rdfs:class(?address)1,)

Gen(rdfs:class(?place)1,)

Gen(rdfs:class(?painter)1,)

Gen(rdfs:class(?oilpainting)1,)

```

$Spec_{(rdfs:class(?craft)1,)}($ $ \begin{aligned} &? painter \text{ rdf:draws } m: ?painting \\ &? place \text{ rdf:hasCity } m: ?city \\ &? country \text{ rdf:hasCity } m: ?city \\ &? museum \text{ rdf:hasAddress } m: ?address \\ &? museum \text{ rdf:hasArtefacts } m: ?artefacts \\ &? artist \text{ rdf:hasCountry } m: ?Country \\ &\quad \{ ? region, "Europe" \} \\ &\quad \{ ? artist, "Asian" \} \\ &\quad \{ ? painting, "oilpainting" \} \end{aligned} $ $)$ $))$
--

**Table 5.42: case 9 algebraic notation**

**Cache:** the cache is used to store information about the subject, object and predicate. For instance, the following table 5.43 is the example of cache for case 9.

Subject	Predicate	Object	Specialization	Generalization	Conditions
? Place	rdf: hasCity	? City	Craft,1	Address,1: Place,1: Painter,1: Oilpainting,1	
?painter	rdf: draws	?Painting			

?Country	rdf: hasCity	? City			
?Museum	rdf: hasAddress	?Address			
?Museum	rdf: hasArtefacts	?Artefacts			{? region, "Europe"}
?Artist	rdf: hasCountry	?Country			{? artist, "Asian"}

**Table 5.43: case 9 cache**

**Identifying sources:** Cache's predicates are searched inside the data dictionary to identify required data repositories. As we can see in the following cache, table 5.44, a new column, the data source, is added after identifying the required sources.

Subject	Predicate	Object	Specialization	Generalization	Conditions	DS
? Place	rdf: hasCity	? City	Craft,1	Address,1: Place,1: Painter,1: Oilpainting,1		Ds 1, Ds 2
?painter	rdf: draws	?Painting				Ds 1, Ds 2
?Country	rdf: hasCity	? City				Ds 1, Ds 2

?Museum	rdf: has Address	?Address				Ds 1, Ds 2
?Museum	rdf: hasArtefacts	?Artefacts			{? region, "Europe" }	Ds 1, Ds 2
?Artist	rdf: hasCountry	?Country			{? artist, "Asian"}	Ds 1, Ds 2

**Table 5.44: case 9 identifying sources**

**Subqueries and merging results:** According to identified data sources, in table 5.44, subqueries are generated. We can see that the required subject and object exist inside the data source column's data sources. Each subquery is sent to the identified data source to get the required data, and then results are combined through the union. As stated in the data source column, table 5.44, 2 following subqueries, table 5.45, are generated.

<u><i>Sub Query for Data Source 1</i></u>	<u><i>Sub Query for Data Source 2</i></u>
SELECT ?Painter, ? Painting, ?Craft, ?Museum, ?Address, ?Region, ?Place, ?City, ?Country, ?Artist, ?Artefacts WHERE { ? address: rdfs: subClassOf: city. ? place: rdfs: subClassOf: region. ? painter: rdfs: subClassOf: artist.	SELECT ?Painter, ? Painting, ?Craft, ?Museum, ?Address, ?Region, ?Place, ?City, ?Country, ?Artist, ?Artefacts WHERE { ? address: rdfs: subClassOf: city. ? place: rdfs: subClassOf: region. ? painter: rdfs: subClassOf: artist.

? oilpainting: rdfs: subClassOf: painting.  ? oilpainting: rdfs: subClassOf: craft.  ? painter rdf:draws m: ?painting  ? place rdf:hasCity m: ?city  ? place rdf:hasCity m: ?city  ? country rdf:hasCity m: ?city  ? museum rdf:hasAddress m: ?address  ? museum rdf:hasArtefacts m: ?artefacts  ? artist rdf:hasCountry m: ?Country  FILTER { ? region, "Europe" }  { ? artist, "Asian" }  { ? painting, "oilpainting" }  }	? oilpainting: rdfs: subClassOf: painting.  ? oilpainting: rdfs: subClassOf: craft.  ? painter rdf:draws m: ?painting  ? place rdf:hasCity m: ?city  ? place rdf:hasCity m: ?city  ? country rdf:hasCity m: ?city  ? museum rdf:hasAddress m: ?address  ? museum rdf:hasArtefacts m: ?artefacts  ? artist rdf:hasCountry m: ?Country  FILTER { ? region, "Europe" }  { ? artist, "Asian" }  { ? painting, "oilpainting" }  }
--	--

**Table 5.45: case 9 subqueries**

**Semantic operators Testing:** Semantic operators refer to operators that perform their tasks based on the semantic context of information. This implies that semantic operators can

manipulate a given text and convert it into its semantic algebraic notation. In this section, semantic operators have been tested to convert SPARQL queries into their algebraic forms.

**Semantic Algebra:** Semantic algebra is also known as the symbolic mathematical language that is used to represent semantic data. In simpler terms, semantic algebra functions to break down semantic information into the most basic, raw form of mathematical data that can identify inference accurately by a computerized system. Semantic algebra essentially helps in detailing systems down to a microscopic level. This is precisely why the technology of semantic algebra plays such a significant role in the research (XU and HONG, 2012).. The application of semantic algebra converts a SPARQL query into its algebraic notations. This process is usually done by using semantic operators, which this section discusses with an explanation and examples. Semantic operators refer to operators that perform their tasks based on the semantic context of information. This implies that semantic operators have the capacity to manipulate a given text and convert it into its semantic algebraic notation. In the following section, semantic operators have been tested to convert SPARQL queries into their algebraic forms. The operators tested in the section are discussed below.

**Project Test:**  $\pi$  sign is used for a project which takes S O and P as input, and the source is schema

**Syntax:**

$$\pi_{\begin{smallmatrix} S? \\ O? \end{smallmatrix}}(source$$

Triplet has three elements, Subject, Predicate and Object. The project operator,  $\pi$  , extracts information about the subject and object from schema and source replace with the schema name.

**Case 10:** Exhibit a list of resources from Museum about the writer and handwritten documents

$$\pi \left[ \begin{array}{l} ?writer \xleftarrow{?S} \\ ?HD-Doc \xleftarrow{?O} \end{array} \right] (Schema$$

As we can see, that we are using? Writer as a subject and? HD-Doc as an object inside Project operator,  $\pi$ . We aim to get information about the writer and handwritten documents from the required schema. We can see that schema have many different classes, subclasses, properties, sub Properties, Domain and range. Our query asks about just writer and hand-written documents, so it brings results to the required information. As we know, the predicate is used to make sense between subject and object, so the predicate is displayed, which links them.

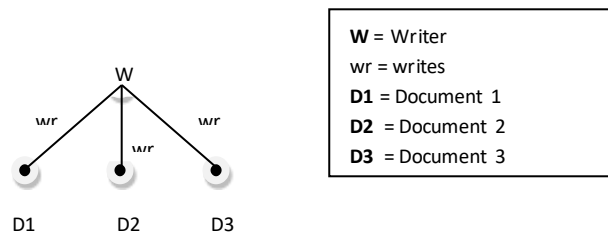
Results in Triplets from Museum schema

The writer writes Handwritten-document 1

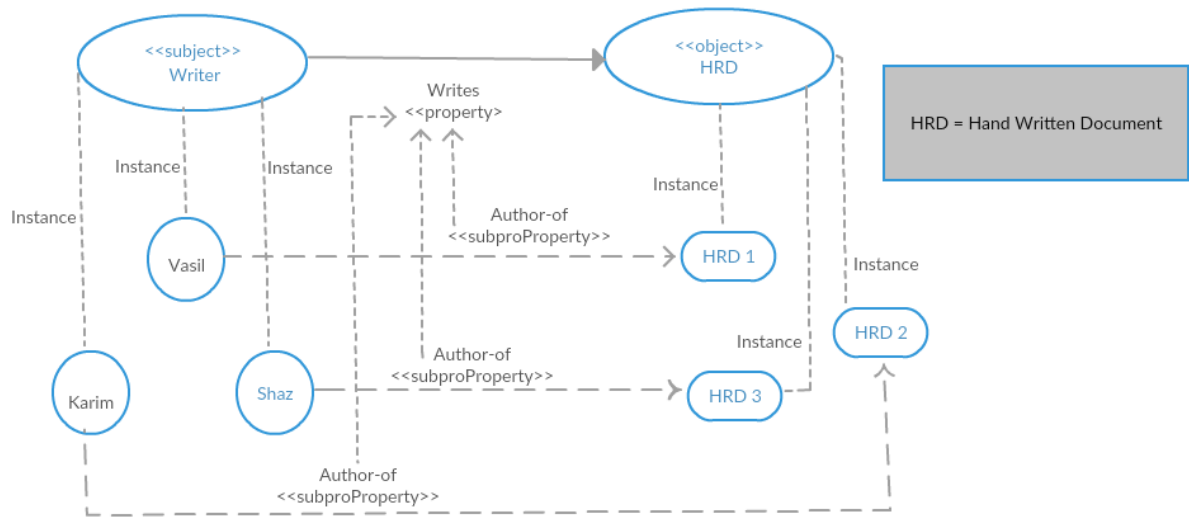
The writer writes Handwritten-document 2

The writer writes Handwritten-document 3

## Model







**Figure 5.2 – case 10 - output of the query**

Figure 5.2 represent the output of our query. In the diagram, we can see the link between **writer** and **HRD**.

**Select Test:**  $\sigma$  Sign is used to select and bring all required sources or nodes which meets the condition. Arithmetic, Comparison, or Boolean operators can be utilized along with constants or strings inside the "**logic**" and source replace with the schema name.

**Syntax:**

$$\sigma_{[logic]}(source)$$

**Case 11:** Exhibit all paintings of OilPainting from Schema

$$\sigma_{painting=oilpainting} (Schema)$$

We aim to get information about all paintings which come under the OilPainting category from the required schema. We can see that schema has many different classes, subclasses, properties, sub Properties, Domain, and range. Our query asks about just OilPainting paintings to bring results related to the required information. In the following diagram, we have created the

instances of OilPainting to represent the output. As we know, that predicate is used to make sense between subject and object so that predicate is displayed, which links them.

### Output from Museum schema

Painting 1 Represented-By OilPainting

Painting 2 Represented-By OilPainting

Painting 3 Represented-By OilPainting

### Model

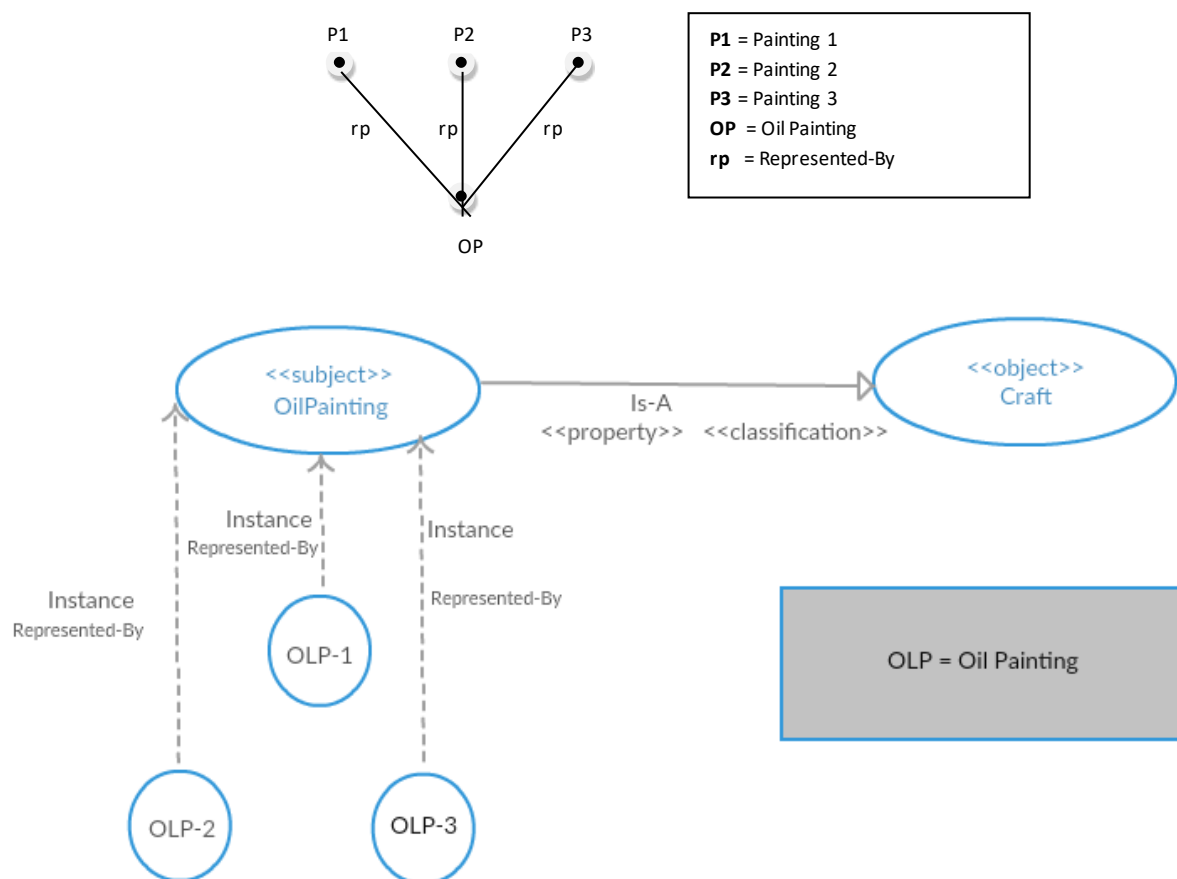


Figure 5.3 - case 11 - Select operator query result

Figure 5.3 represent the output of our query. In the diagram, we can see different instances of OilPainting, OLP1, OLP2 and OLP3.

**Join Test:**  $\bowtie$  The sign is used for a join. It combines triplets from single or multiple sources according to the requested query. We are using both operators, Project and Select, in our syntax. The project operator takes two parameters, “?X” and “?Y”. As the triplet has three elements, Subject, Predicate and Object, so ?X represent the subject and ?Y represent an object. The select operator,  $\sigma$ , is used for a condition and Arithmetic, Comparison, or Boolean operators can be utilized along with constants or strings inside the "*condition*" and source replace with the schema name.

**Syntax:**

$$\pi_{[?X,?Y]} \sigma_{[condition]}(source)$$

$\bowtie$

$$\pi_{[?X,?Z]} \sigma_{[condition]}(source)$$

**Case 12:** Exhibit all paintings of all painters from schemas where used crafts is watercolour

$$\pi_{[?painter,?painting,?craft]} \sigma_{[craft='watercolour']}(Schema A)$$

$\bowtie$

$$\pi_{[?painter,?painting,?craft]} \sigma_{[B.craft=A.craft]}(Schema B)$$

We are using a join operator ( $\bowtie$ ) to join the sources. We aim to get information about all painter's paintings where used craft is watercolour. We can see that schema have many different classes, subclasses, properties, sub Properties, Domain and range. Our query is asking about painting where used craft is watercolour as we have two homogeneous schemas. The first query fetches information about craft from Schema A, where used craft is watercolour. The second query fetches information from schema B where the used craft is the same as in schema A. Join operator joins both results display as an output, as shown in the following example.

#### Schema A

?Painter	?Painting	?Craft
Shaz	Painting 1	Water Colour
Dr Vasil	Painting 2	Water Colour
John	Painting 3	Water Colour
David	Painting 4	Water Colour

#### Schema B

?Painter	?Painting	?Craft
Peter	Painting 5	Water Colour
Tony	Painting 6	Water Colour
Alexander	Painting 7	Water Colour

A  $\bowtie$  B

?Painter	?Painting	?Craft
----------	-----------	--------

Shaz	Painting 1	Water Colour
Dr Vasil	Painting 2	Water Colour
John	Painting 3	Water Colour
David	Painting 4	Water Colour
Peter	Painting 5	Water Colour
Tony	Painting 6	Water Colour
Alexander	Painting 7	Water Colour

Model

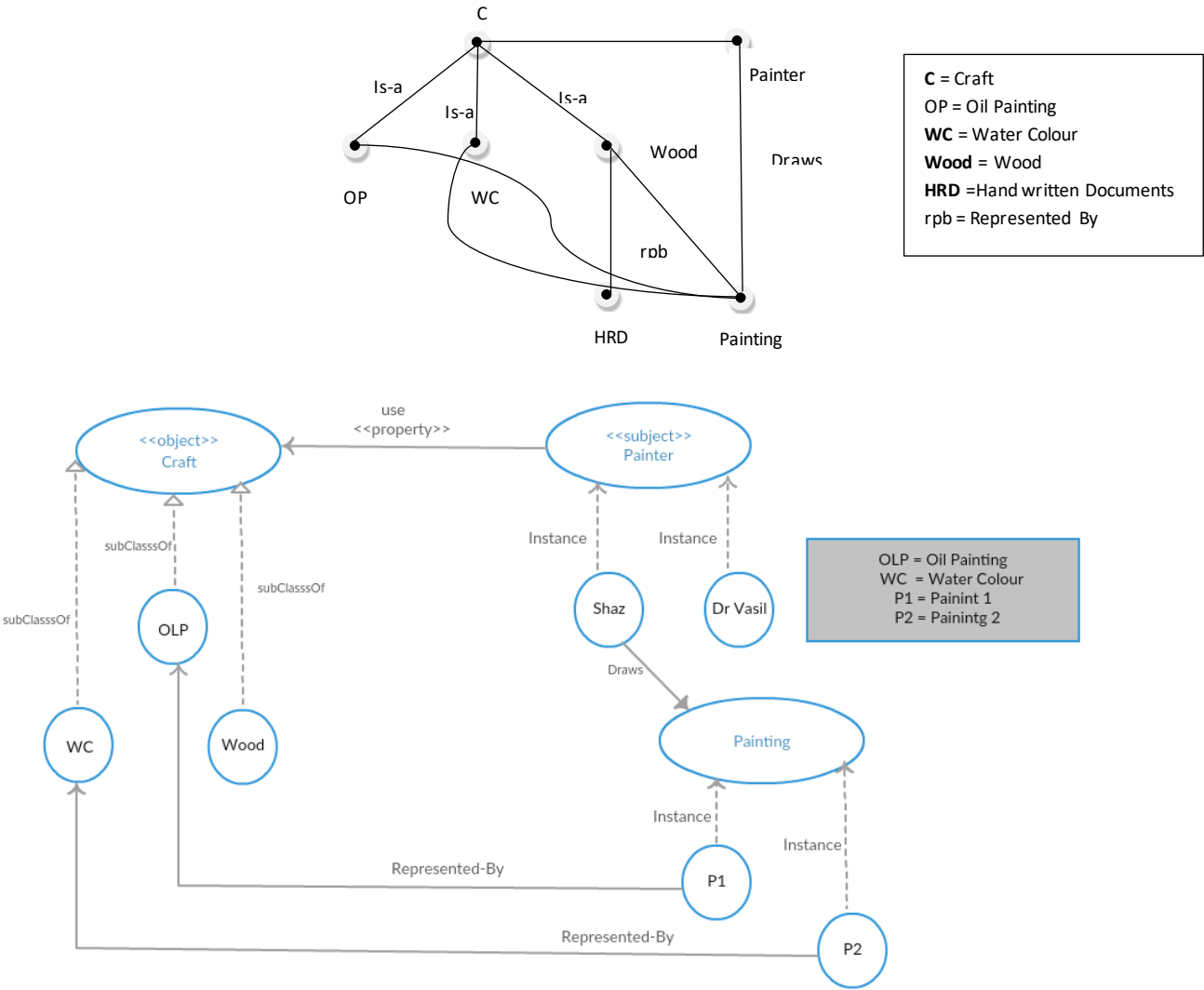


Figure 5.4 - caee 12 - Join query result

**Outer Join Test:** We are using an outer join operator ( $\bowtie$ ) to join the not-matched triplets. We aim to get information about all painter's paintings where used craft is watercolour. We can see that schema have many different classes, subclasses, properties, sub Properties, Domain and range. Our query is asking about painting where used craft is watercolour as we have two homogeneous schemas. The first query fetches information about the craft from schema A where the used craft is watercolour. The Second query fetches information from schema B where the used craft is the same as in schema A. outer join operator joins both results plus unmatched triplets from SchemaB and display as an output as shown in the following example.

$$\pi_{[?painter, ?painting, ?craft]} \sigma_{[craft='watercolour']} (Schema A)$$

—  $\bowtie$  —

$$\pi_{[?painter, ?painting, ?craft]} \sigma_{[B.craft=A.craft]} (Schema B)$$

#### Schema A

?Painter	?Painting	?Craft
Shaz	Painting 1	Water Colour
Dr Vasil	Painting 2	Water Colour
John	Painting 3	Water Colour
David	Painting 4	Water Colour

#### Schema B

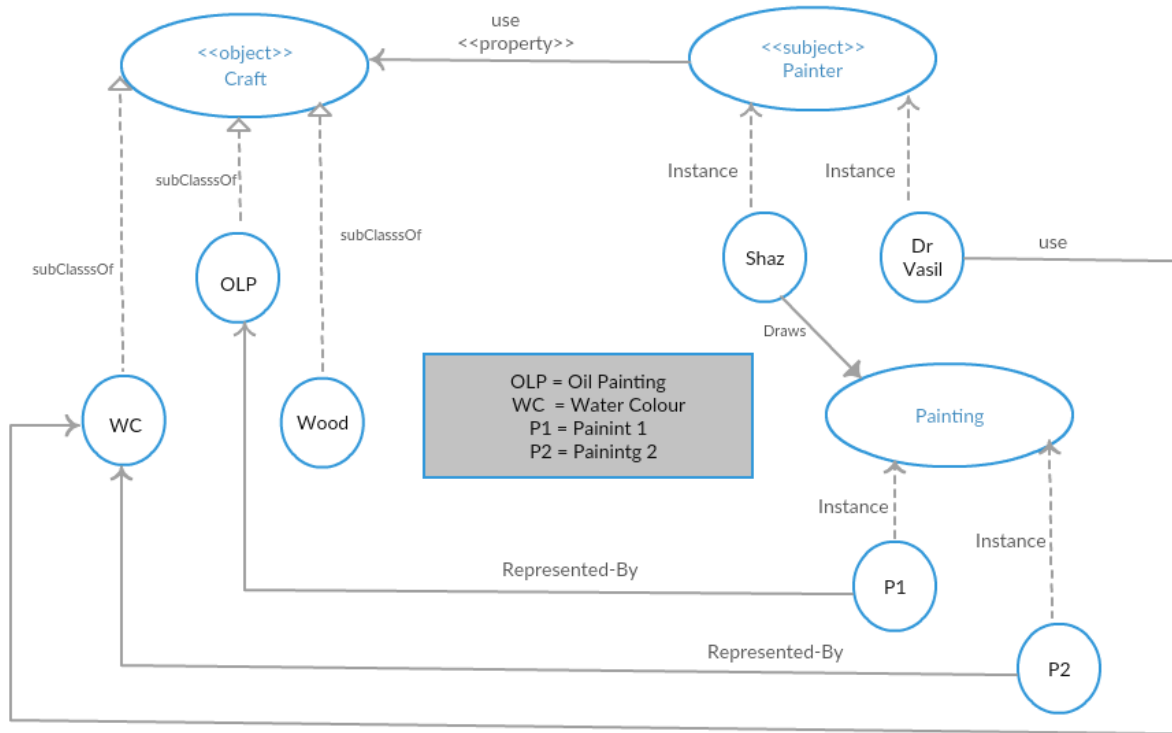
?Painter	?Painting	?Craft
Peter	Painting 5	Water Colour

Tony	Painting 6	Water Colour
Alexander	Painting 7	Water Colour
Rosy	Painting 8	Wood
Raja	Painting 9	Stone

A - ∞ - B

?Painter	?Painting	?Craft
Shaz	Painting 1	Water Colour
Dr Vasil	Painting 2	Water Colour
John	Painting 3	Water Colour
David	Painting 4	Water Colour
Peter	Painting 5	Water Colour
Tony	Painting 6	Water Colour
Alexander	Painting 7	Water Colour
Rosy	Painting 8	Wood
Raja	Painting 9	Stone

## Model



**Figure 5.5 - case 12 - Outer join query result**

**Case 13:** Exhibit all-female writer's origin and handwritten documents where used craft is wood

$$\pi_{?Writer,?Country,HRD,Gender} [\sigma_{Gender='Women' \wedge craft='wood'}] (\text{Schema A})$$

⋈

$$\pi_{?Writer,?Country,HRD,Gender} [\sigma_{Gender=A.Gende \wedge craft=A.craft}] (\text{Schema B})$$

We are using join operator ( $\bowtie$ ) to join the sources. In this example, we require females writer's origin and handwritten documents where the used craft is wood. We can see that schema have many different classes, subclasses, properties, sub Properties, Domain and range. Our query asks about the females' writer's origin and handwritten documents where the used craft is wood.



The first query fetches information about writers, writers' origins, handwritten documents, gender and restricted women and woodcraft from Schema in this scenario. The second query fetches information from schema B where writers, writer's origin, handwritten documents, gender are similar to the first query's output. The following example is found in the following example: join operator joins both results and display as output and eliminate duplicate entry .

**Schema A results:**

<b>?Writer</b>	<b>Hand Written Document</b>	<b>Craft</b>	<b>?Country</b>	<b>?Gender</b>
Zain	HRD1	Wood	UK	Women
Taby	HRD2	Wood	UK	Women
Tara	HRD3	Wood	America	Women
Valin	HRD4	Wood	America	Women

**Schema B Results:**

<b>?Writer</b>	<b>?Hand Written Document</b>	<b>?Craft</b>	<b>?Country</b>	<b>?Gender</b>
Zain	HRD1	Wood	UK	Women
Mauna	HRD5	Wood	UK	Women

**A ⋈ B**

<b>?Writer</b>	<b>Hand Written Document</b>	<b>Craft</b>	<b>?Country</b>	<b>?Gender</b>
Zain	HRD1	Wood	UK	Women
Taby	HRD2	Wood	UK	Women
Tara	HRD3	Wood	America	Women
Valin	HRD4	wood	America	Women

Mauna	HRD5	wood	UK	Women
-------	------	------	----	-------

**Case 14:** Exhibit all artefacts where used material is cooper and origin is Pakistan and India

$\pi_{?Artifacts,?Origin,?Material} [\sigma_{Origin \text{ IN } (Pak,IND) \wedge Material \text{ IN } (cooper,bronze)}'] (\text{SCHEMA A})$



$\pi_{?Artifacts,?Origin,?Material} [\sigma_{Origin='A.origin \wedge Material=a.Material'}'] (\text{SCHEMA B})$

In this example, we require all artefacts where used material is cooper and origin is Pakistan and India. The query is asking about all artefacts where used material is cooper and origin is Pakistan and India. In this scenario, the first query fetches information about Artefacts, Origin, used material, and restrict Pak and India's origin from Schema A. The second query fetches information from schema B Artefacts, Origin, and used material is similar to the first query's output. Join operator joins both results and display as output and eliminate duplicate entry, as shown in the following example.

**Schema A output**

Artifacts	Material	Origin
Painting 1	Cooper	Pak
Painting 2	Cooper	Pak
HRD 1	Cooper	Pak

**Schema B output**

Artifacts	Material	Origin

Painting 1	Cooper	Pak
Painting 3	Cooper	IND
HRD 1	Cooper	IND
HRD7	Bronze	Pak
HRD8	Bronze	Pak

**A ⋈ B**

<b>Artifacts</b>	<b>Material</b>	<b>Origin</b>
Painting 1	Cooper	Pak
Painting 2	Cooper	Pak
Painting 3	Cooper	IND
HRD 1	Cooper	IND
HRD7	Bronze	Pak
HRD8	Bronze	Pak

**Case 15:** Exhibit all artefacts where their artist's beliefs are Buddhism and region is America

$\pi_{?Artifacts,?Artists,?Region,?Beliefs} [\sigma_{Belief='BuddHism,^Region \text{ IN } (America,Asia)}'](\text{SCHEMA A})$

**⋈**

$\pi_{?Artifacts,?Artists,?Region,?Beliefs} [\sigma_{Belief='A.Belief,^Region=A.Region'}](\text{SCHEMA B})$

Query aim is to get information about all Artefacts where Artist's beliefs are Buddhism and Regions are America and Asia. Schema has many different classes, subclasses, properties, sub Properties, Domain and range. Our query asks about all Artefacts where Artist's beliefs are Buddhism and Region are America and Asia. Before joining, the first query fetches information

about Artefacts, Artist's beliefs and Origin from Schema A where beliefs are Buddhism and Region are America, Asia and Second query fetches the same information from schema B before joining them. The join operator joins both results and displays as an output, as shown in the following example.

#### Schema A Output

Artefacts	Artists	Region	Beliefs
HRD 2	Smith	America	Buddhism

#### Schema B Output

Artefacts	Artists	Region	Beliefs
Painting 9	Shaby	Asia	Buddhism
Painting 10	Tabby	America	Buddhism
HRD 2	Smith	America	Buddhism

#### A ⋈ B

Artefacts	Artists	Region	Beliefs
HRD 2	Smith	America	Buddhism
Painting 9	Shaby	Asia	Buddhism
Painting 10	Tabby	America	Buddhism
HRD 2	Smith	America	Buddhism

**Generalization Test:** Generalization is the process of extracting common characteristics from one or more classes and combining them into a generalized superclass. It is used to get hierarchies of classes and subclasses in up level, up to defined level. In our case, we are going up to 1 level. Triplet has three elements, Subject, Predicate and Object. Subject and object

always represent classes or subclasses. The generalization operator extracts parent class up to 1 level of mentioned class (**?class**). The source is replaced with the schema.

$$\mathbf{Gen}_{(rdfs:class(?class),n-level)}(Source)$$

**Case 16:** Exhibit all the hierarchies of painting at level 1

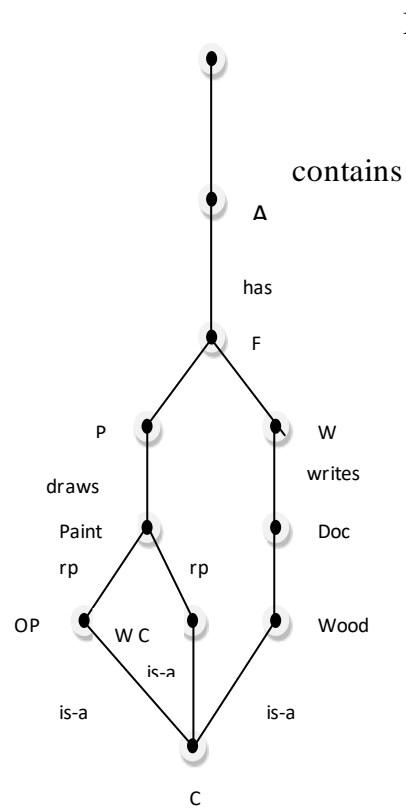
$$\mathbf{Gen}_{(rdfs:class(?painting)1)}(Schema)$$

In the query, we are asking about show the parent class of painting at level 1. The required schema has many classes. The query starting point is **?painting at level 1**, which we have mentioned in the query,  $\mathbf{Gen}_{(rdfs:class(?painting)1)}(Schema)$ , the operator (**Gen**) fetches and displays the painting's superclass as shown in the following output and graphical representation of the schema. According to our Museum schema, Painting has just 1 level up, so we get the following triplet:

**Output triplet from Museum schema**

Painting is-a an Artefact.

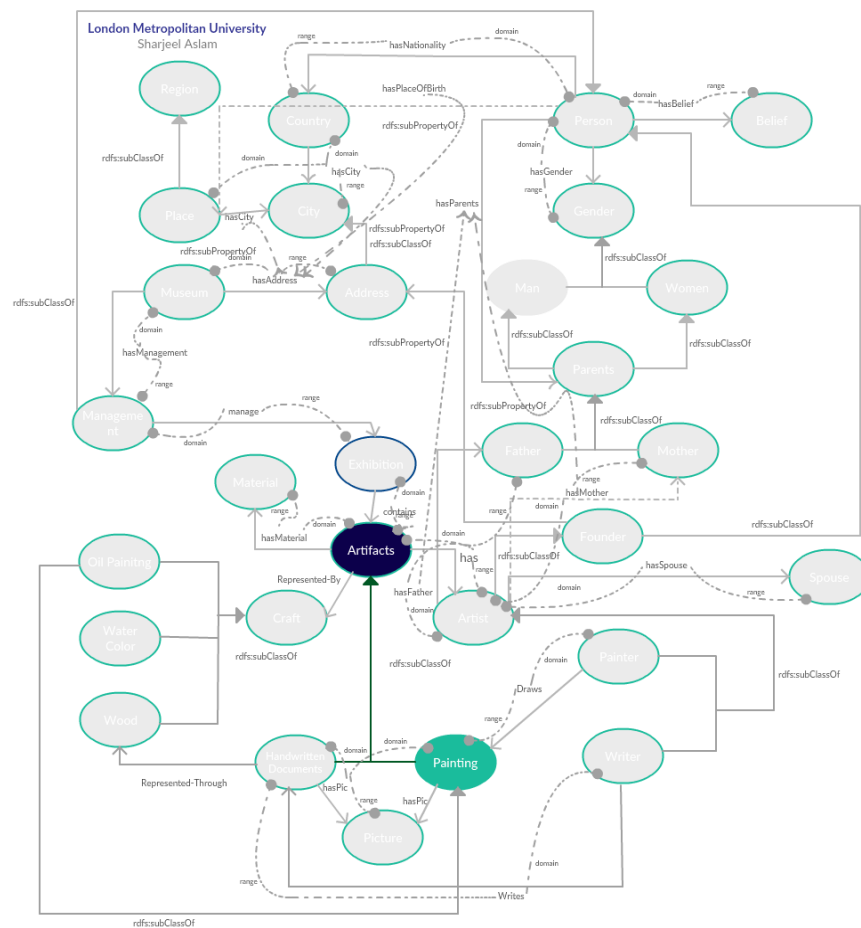
## Model:



M

**M** = Museum  
**A** = Artefact  
**F** = Founder  
**P** = Painter  
**W** = Writer  
**Paint** = Painting  
**Doc** = Documents  
**OP** = Oil Painting  
**WC** = Water Colour  
**Wood** = Wood  
**C** = Craft

### 6.3.1.1 Graphical representation of Generalization



**Figure 5.6 - Generalisation output**

**Specialization Test:** Specialization is the reverse process of Generalization, which means creating new subclasses from an existing class. In our case, we are going bottom-up to 1 level.

Abstract from Museum schema:

Triplets:	
<Place <b>is-a</b> Region>	<Artist <b>is-a</b> Founder>
<Place <b>has</b> City>	<Founder <b>is-a</b> Person>
<Country <b>has</b> City>	<Painter <b>is-a</b> Artist>
<Museum <b>hasAddress</b> Address>	<Writer <b>is-a</b> Artist>

<Address <b>is-a</b> City>	<Artist <b>hasSpouse</b> Spouse>
<Museum <b>hasManagement</b> Management>	<Artist <b>hasFather</b> Father>
<Management <b>manage</b> Exhibition>	<Artist <b>hasMother</b> Mother>
<Exhibition <b>contains</b> Artifacts>	<Father <b>is-a</b> Parents>
<Artifacts <b>hasMaterial</b> Material>	<Mother <b>is-a</b> Parents>
<Artifacts <b>represented-By</b> Craft>	<Parents <b>is-a</b> Man>
<OilPainting <b>is-a</b> Craft>	<Parents <b>is-a</b> Women>
<Watercolour <b>is-a</b> Craft>	<Man <b>is-a</b> Gender>
<Wood <b>is-a</b> Craft>	<Women <b>is-a</b> Gender>
<Painting <b>is-a</b> Artifacts>	<Person <b>hasGender</b> Gender>
<HandWrittenDocuments <b>is-a</b> Artifacts>	<Person <b>hasBelief</b> Belief>
<HandWrittenDocuments <b>represented-through</b> Wood>	<Person <b>hasNationality</b> Nationality>
<Painting <b>hasPic</b> Picture>	<Person <b>hasPlaceOfBirth</b> Place>
<HandWrittenDocuments <b>hasPic</b> Picture>	
<Artifacts <b>has</b> Artist>	

### Syntax:

$$Spec(rdfs:class(?class),n-level)(Schema)$$

Triplet holds three elements, Subject, Predicate and Object. Subject and object always represent classes or subclasses. The operator (***Spec***) extracts child classes up to 1 level of mentioned class (**?class**). The source is replaced with the schema



**Case 17:** Exhibit all the hierarchies of the craft down to 1 level (bottom)

***Spec***<sub>(*rdfs:class*(?craft)1,)(*Schema*)</sub>

In the query, we are asking about show the child classes of craft class at level 1. The required schema has many classes. The query starting point is? **craft at level 1**, which we have mentioned in the query, ***Spec***<sub>(*rdfs:class*(?craft)1,)(*Schema*)</sub> . ***Spec*** operator fetches and displays the craft's subclasses at level 1 as shown in the following output and graphical representation of the schema. According to our requirement, query fetches information at 1 level down from Museum schema so that we get the following triplets:

**Abstract from Museum schema**

<OilPainting is-a Craft>

<Watercolour is-a Craft>

<Wood is-a Craft>#

## 5.7 Test Results Analysis:

This section includes the analysis of all tests that we performed to check the accuracy of the developed system according to this thesis's aims and objectives. The first part of the test was that the main SPARQL query should be converted into the algebraic expression, and we noticed during the test that this part worked as expected without any errors. The next part system had to store the subject, object, and predicate into the cache memory. According to tests, it stored all subjects, objects, and predicates of the query into the cache memory. It has been mentioned that unit testing and functional testing strategies were used in this test. All units of the developed system need to check individually and as a whole system. The developed system uses the index mechanism to store all participated RDFs data sets. In the next part, after storing subjects, objects and predicates into cache memory, the system had to check and identify required RDF data repositories from the indexed data. Tests results showed that this section

worked well without any errors. The next part was crucial and vital as the main SPARQL query converted into subqueries according to the identified RDF data sources. Test results showed the conversion of the SPARQL query into multiple subqueries. The next part was to send each generated subquery to the required source to get the data, join all subqueries results into a single result, and display it to the end-user. Tests showed that all developed system units worked as expected, and no errors during the testing of all units. Following table 52 illustrates the testing outcomes.

**Testing table:**

<b>Test ID</b>	Algo-786	<b>Description</b>	The author test the proposed/developed framework. The developed framework can retrieve the RDF data from distributed homogeneous ontologies. Homogeneous ontologies mean that all distributed ontologies structures should be the same. The developed framework has multiple stages, and for each stage, a unique algorithm has been created. The author tested each algorithm to check efficiency and accuracy. Different algorithms are as follows: converting SPARQL queries into an algebraic expression, storing subject, object, and predicate into a cache, searching the index to identify the required distributed RDF repositories, converting subqueries, and merging the subqueries results.	
<b>Developed By</b>	Sharjeel Aslam	<b>Version</b>	Final	

<b>Tester's Name</b>	Sharjeel Aslam	<b>Date</b>	23-10-2019	<b>Test Result</b>	Pass
----------------------	----------------	-------------	------------	--------------------	------

<b>S #</b>	<b>Prerequisites:</b>
1	Distributed museum ontologies
2	Windows 10 Laptop with Intel® Core™ i7, RAM: 16 GB, Quad-core, 1.8 GHz / 4.9 GHz
3	Apache Jena framework for a java programming language
4	Protégé – Ontology editor
5	Tester

<b>S #</b>	<b>Test framework stages - Algorithms</b>
1	Converting SPARQL query into an algebraic expression
2	Storing subject, object, and predicate into a cache
3	Identifying distributed RDF repositories from the index
4	Converting main SPARQL query into subqueries
5	Sending each subquery to the required distributed RDF repository
6	Merging the multiple subqueries results into one result

**Test Scenario**

The tester checked the validity of the developed framework, which can retrieve data from the homogeneous RDF repositories.

Step #	Details	Expected Results	Actual Results	Pass / Fail / Not executed / Suspended
1	Converting SPARQL query into an algebraic expression	After triggering the main SPARQL query, the query must be converted into an algebraic expression	Main SPARQL query converted into algebraic expression successfully	Pass
2	Storing subject, object, and predicate into a cache	Subject, object and predicate must be stored inside the cache	Subject, object and predicate stored inside the cache successfully	Pass
3	Identifying distributed RDF repositories from the index	Must match the stored cache data against the stored index to identify the distributed RDF repositories	Identified the distributed RDF repositories after matching the stored cache data against the index data	Pass
4	Converting main SPARQL query into subqueries	The main SPARQL query must be converted into multiple subqueries according to the identified distributed RDF repositories	Generated multiple subqueries successfully after identifying the distributed RDF sources	Pass
5	Sending each subquery to the required distributed RDF repository	Each subquery must be triggered against the distributed RDF repository	Each subquery triggered against the required distributed RDF source successfully to fetch the data	Pass
6	Merging the multiple subqueries results into one result	Subqueries results must be merged semantically into one result	Each subquery results were merged into one result successfully.	Pass

**Table 5.46 - Testing table**

## 5.8 Critical analysis:

In this chapter, proposed framework testing has been done with distributed museum RDF ontologies. It was a complex procedure to obtain a result, as multiple algorithms were in a sequence to perform the complete task. Starting algorithm's task was to convert the main SPARQL query into an algebraic expression, challenging and essential. The subsequent algorithm had to take the algorithm's output as an input to perform the further task. Test results showed that the relevant algebraic expression of the main SPARQL query was converted successfully. The indexing mechanism had a leading role. It had to index all predefined selected RDF distributed repositories, and irrelevant entries into the index could lead to a wrong match between the cache algorithm, where we stored all subject, object, and predicate of SPARQL query. The caching algorithm helped us to identify the distributed RDF repositories which hold the required information. The subqueries' algorithm had to generate multiple subqueries based on this data, which had to retrieve data from the required RDF repositories. Combining the returned results of subqueries into a single semantic output was tricky as one wrong result of the subquery could lead to irrelevant data into the joining result. The author had to face challenges during the data retrieval from a heterogeneous environment. Data belongs to different formats in the heterogeneous environment, such as relational data, XML data, NoSQL data, and RDF data. The author discussed the limitation of this research in chapter 1. The proposed framework only works in a homogeneous environment where all participated ontologies have to be in the same format. For this purpose, we used CRM (Conceptual Reference Model) to develop our museum ontology structure and format.

The author used RDF, apache Jenna framework, SPARQL query and many more techniques related to them. Jena architecture provides different persistent, inference RDF, Ontology, Query, and related API's that could be invoked using Java programming language and over the web using HTTP and SPARQL query language. In RDF, a reified triple is a description of a triple-token using other RDF triples. RDF reification was intended to make provenance statements and other statements about RDF triples with a unique vocabulary that includes `ref:Statement`. An ontology model is an extension of the Jena RDF model, providing extra capabilities for handling ontologies. Ontology models are created through the Jena Model Factory. It specifically talks about the types and approaches of data integration. Simplified Agile Methodology (**SAMOD**) methodology has been adopted for the Museum's ontology Development. In this thesis, CRM (Conceptual Reference Model) has been used to develop the museum's ontology. Protégé has been used to implement the museum's ontology. All the gathered data is divided into class and subclass. On these data, after applying different query and sub-queries. The data is divided into different forms by applying different properties. These separate all the data according to nature and properties. It was furthermore classified into different groups. Data dictionary holds information about the subject, object, predicate, property, sub Property, classes, and subclasses. The cache's predicate was used to search inside the data dictionary. It implements the semantic algebraic expressions, data dictionary, cache, conversion of main SPARQL query into sub-queries, and merging.

## 5.9 Chapter Summary:

Several significant challenges and approaches have been identified throughout this chapter concerning the semantic web in distributed ontologies. This chapter discussed the testing of the conceptual framework using the Jena framework, unit testing and functional testing to access and test the data from distributed museum RDF data sets. Furthermore, it also discussed the methodology used behind the developed museum's ontology used as a case study. The first is

the trade-off in ontology language. A step-by-step process was adopted. Multiple algorithms were tested, translating the SPARQL query into an algebraic expression, converting the main SPARQL query into sub queries, and carrying out SPARQL queries in distributed ontology's. Finally, another algorithm was tested to combine the results. Thus, triples and variables are stored in the cache and identified by the system to carry out the queries, which is more efficient than sourcing data each time from the source. Simplified Agile Methodology (**SAMOD**) methodology has been adopted for Museum's ontology Development. Jena is also compatible with the three different OWL ontology language levels- OWL Lite, OWL DL, OWL Full. Jena Ontology API provides a language-neutral interface that can use a profile to set specific java classes and properties. Jena uses OWL for providing extra capabilities for handling ontology. Ontology models are created through the Jena Model Factory. In the Apache Jena framework, Jena provides an open platform to use built-in and third-party inference engines. In this chapter, the author executed proposed algorithms against various test cases. The framework had multiple stages where the main SPARQL query converted into algebraic expression and cache had to hold the information about the subject, object and predicate. Later, cache data matched with index data to identify the required RDF repositories as, based on this information, multiple subqueries had to generate to retrieve the data from distributed sources. Finally, subqueries returned data had to combine into one semantic result.



# Chapter 6

## Framework Evaluation

### 6.1 Introduction

This chapter evaluates the performance of the implemented framework and its accuracy compared to other similar systems. Against this backdrop, the evolution aim is to demonstrate that the proposed system can efficiently handle distributed SPARQL queries. In particular, the chapter shall compare the proposed system with other similar systems. These include **FedX**, **ANAPSID** and **ADERIS**, which we reviewed in [section 2.9](#). The author selected these systems because of their similar functionalities proposed in our system. All chosen systems under this evaluation have implemented the triple pattern for the SPARQL endpoints, which bears similarity with our proposed system. These systems' functions prevent the client from stating the URL to fetch data from distributed resources instead of overwhelming network traffic.

Given that our research topic is very trendy, many other systems propose and implement a distributed extension to SPARQL. The selection choice is based on the fact that these systems focus on Sesame, and their models implemented the join. Generally, the system's efficiency goes down when adding or merging more RDF data sources. The selected system's query plan includes statistics from the triple pattern, and query performance goes up when all RDF sources are mentioned in the SPARQL query. However, as we discussed the limitations and gaps of these systems during the literature review in [section 2.10](#) when these systems try to add more RDF sources after query results, the results are not as accurate as they are perceived to be. These systems first get results from RDF sources which frequently get a no-connect error if the required data source is unavailable.

## 6.2 Performance

In this section, the author evaluates the results and performance of our proposed system with other particular systems that provide distributed SPARQL query processing mechanism. This endeavour aims to demonstrate that the proposed system can efficiently handle distributed queries on distributed RDF data stores. For the demonstration, the author used the **Virtual Exhibition Museum** domain specifically for this purpose. All validations were completed in the windows system with an i7 processor and 8 GB of memory.

**Virtual Exhibition Museum Data Description:** The virtual exhibition museum holds 3600 triples in 12 RDF museum data sets. We used RDF museum data sets: London Museum, Scotland Museum, Birmingham Museum, Manchester Museum, Wales Museum, Chester Museum, Taxila Museum, Peshawar Museum, Multan Museum, Chitral Museum, Lahore Museum and Sawat Museum. The following table 53 provides the details of endpoints.’The following table 6.1 shows the namespace column, which organises all participated distributed museum ontologies. We can also see that all museums have the same triples. We are using a homogeneous environment where all participated ontologies have to be in the same format. For this purpose, we used CRM (Conceptual Reference Model) to develop our museum ontology format and created the same ontology multiple times to execute identical SPARQL queries.

<b>Museums</b>	<b>Triples</b>	<b>Namespace</b>
London Museum	300	http://allahm.museum.org/museum#
Scotland Museum	300	http://allahm.museum.org/museum#
Birmingham Museum	300	http://allahm.museum.org/museum#
Manchester Museum	300	http://allahm.museum.org/museum#
Wales Museum	300	http://allahm.museum.org/museum#
Chester Museum	300	http://allahm.museum.org/museum#
Taxila Museum	300	http://allahm.museum.org/museum#
Peshawar Museum	300	http://allahm.museum.org/museum#
Multan Museum	300	http://allahm.museum.org/museum#
Multan Museum	300	http://allahm.museum.org/museum#
Lahore Museum	300	http://allahm.museum.org/museum#
Sawat Museum	300	http://allahm.museum.org/museum#

**Table 6.1 - Details of endpoints**

Following Tables 6.2 and 6.3 outlines the features of participated systems as the author discussed the features of these systems in [section 2.9](#).

<b>Features</b>	<b>FedX</b>	<b>ANAPSID</b>	<b>ADERIS</b>	<b>Our System</b>
Indexing in Memory	Yes	Yes	Yes	NO
Stored Index	No	No	No	Yes
Dynamic Indexing	No	No	No	Yes
Generating algebraic	No	No	NO	Yes
Cache	Yes	Yes	Yes	Yes

Decomposing main query	NO	NO	Yes	Yes
Static Generalization	Yes	Yes	Yes	Yes
Dynamic Generalization	NO	No	No	Yes
Static Specialization	Yes	Yes	Yes	Yes
Dynamic Specialization	No	No	No	Yes

**Table 6.2 - Features of Participated Systems**

The following table 6.3 provides details on queries patterns, Generalization, Specialization, Joins, and Filters.

Query	Specialization	Generalization	Joins	Filters	Variables
1	No	No	Yes	Yes	Yes
2	Yes	No	Yes	Yes	Yes
3	No	No	Yes	Yes	Yes
4	Yes	No	Yes	Yes	Yes
5	No	No	Yes	Yes	Yes
6	No	No	Yes	No	Yes
7	Yes	No	Yes	Yes	Yes
8	No	Yes	Yes	Yes	Yes
9	Yes	Yes	Yes	Yes	Yes

**Table 6.3 - Patterns of Queries**

## 6.3 Results

This validation aims to demonstrate how the proposed system can handle and retrieve information from distributed resources. In this regard, the author used nine SPARQL queries to exemplify this objective, starting from section 6.3 Results. Author used Protégé software to build Virtual Museum Exhibition's ontology and used Intel i7 with two core and 16GB RAM. Additionally, the author configured Apache Jena Fuseki 3, a SPARQL server, to handle our queries. To derive correct performance results, the author executed each query 10 times for the accurate result. Author used CRM (Conceptual Reference Model) to develop museum ontology format and created the same ontology multiple times to execute identical SPARQL queries. Author stored all distributed ontologies locally in different endpoints but under the same namespace to create the distributed environment. The author used the Apache Jena framework for a java programming language to develop the test environments in java libraries.

The first query demonstrates fetching data if the Parent class is a subclass in other repositories. Query 1 shall include all artefacts of woodcraft. All other systems, FexX (Qudus, Saleem, Ngonga Ngomo and Lee, 2021), ANAPSID (Acosta et al., 2011) and ADERIS (Kim et al., 2017), used the memory index technique and our system indexed all repositories in a local server. All systems, including us, used cache storage, where systems stored the subject, object and predicate. Cache storage helped to identify the resources as cache data was matched with the indexed data. Other systems did not generate subqueries after identifying the resources. They sent the main SPARQL query to distributed repositories to fetch the data. Our system converted the main SPARQL query into multiple subqueries, and then each subquery triggered against the distributed repository to fetch the data. Figure 6.1 shows the results of this query from all systems, given that FedX took less time to execute. The memory index of FedX is much faster than other systems. The main SPARQL query has only four variables and one filter.

```
PREFIX m:<http://allahm.museum.org/museum#>
```

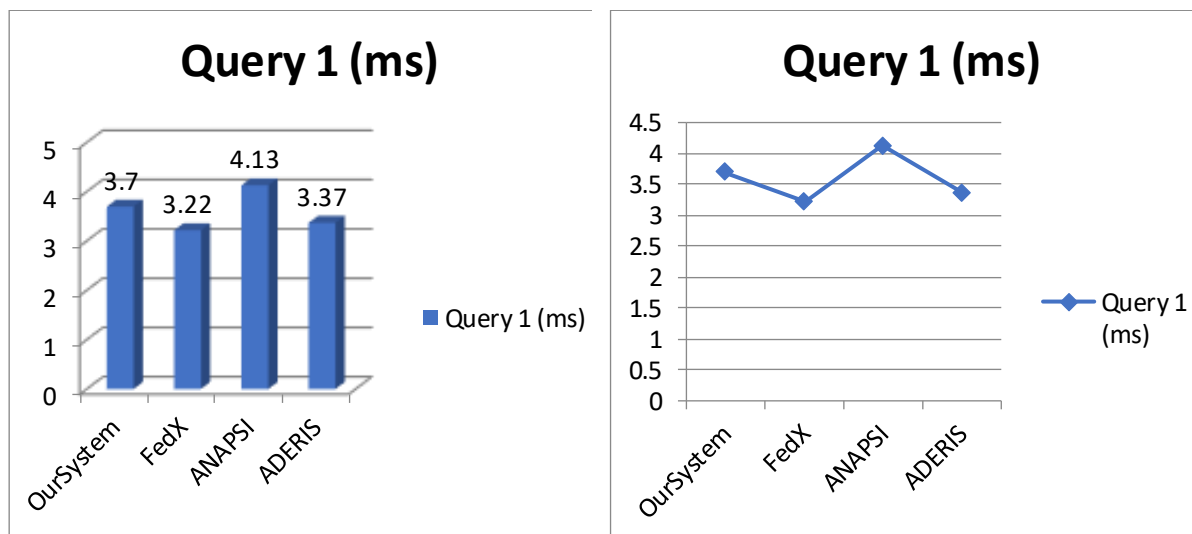
```
SELECT ?painting ,?Artefacts, ?craft
```

```
WHERE {
```

```
?artefacts rdf:represented-by m:Craft
```

```
    FILTER { ?craft ,"wood" }
```

```
}
```



**Figure 6.1 - Query 1 validation results**

The second query shows how to fetch and handle data if the child class appears as a parent class in other RDF repositories. The query is asking about parents' details of all artists who wrote handwritten documents. All other systems did not have dynamic indexing, where they can not add more RDF repositories if required. Our system had the dynamic index mechanism where we first indexed all repositories locally then added more repositories into the index when required. Other system had static specialisation where they could only search in the local repository one by one for the parent class of the child class.

In contrast, our system had dynamic specialisation, and it searched not only locally but also in other distributed repositories. FexX, ANAPSID, ADERIS, used the memory index technique and our system indexed all repositories in a local server. All systems, including us, used cache storage, where systems stored the subject, object and predicate. Cache storage helped to identify the resources as cache data was matched with the indexed data. Other systems did not generate subqueries after identifying the resources. They sent the main SPARQL query to distributed repositories to fetch the data. Our system converted the main SPARQL query into multiple subqueries, and then each subquery triggered against the distributed repository to fetch the data. For this purpose, we have introduced a new operator *Spec* (specialisation) that extracts specific subclasses of parent class Figure 6.2 illustrates the results of this query from all systems since OurSystem took less time to execute. This is because our system uses dynamic specialisation features, something that does not exist in other systems. Query only has six variables without any filters and specialisation functions.

```
PREFIX m:<http://allahm.museum.org/museum#>

SELECT ?Parent, ?father, ?mother ?artist, ?writer, ?HandwrittenDocuments

WHERE {

  ?father : rdfs:subClassOf :parent.

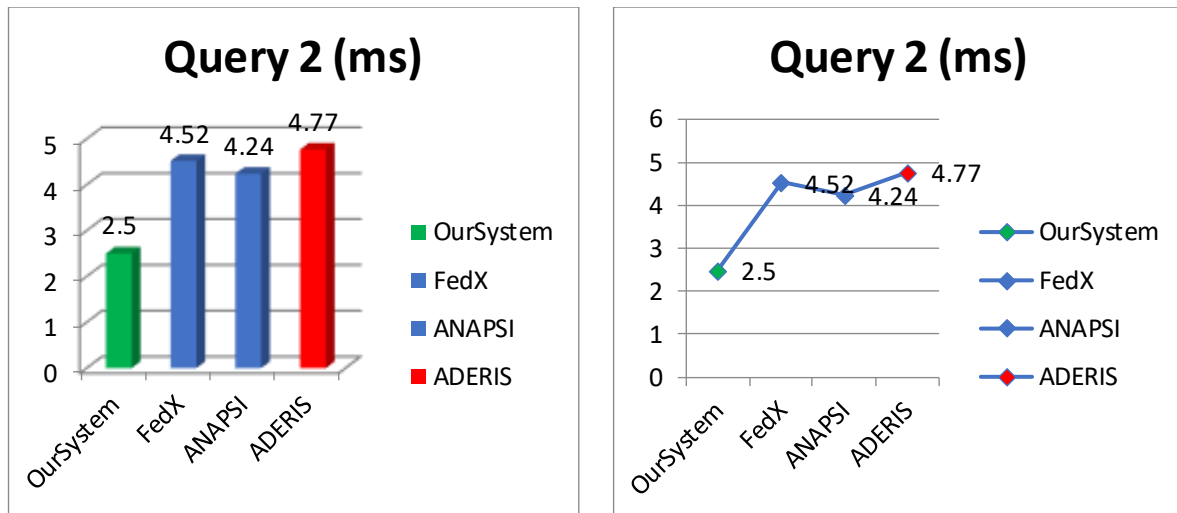
  ?mother : rdfs:subClassOf :parent.

  ?writer : rdfs:subClassOf :artist.

  ?artist rdf:hasParents m: ?Parents

  ?writer rdf:writes m: ?HandwrittenDocuments

}
```



**Figure 6.2 - Query 2 validation results**

Under the third query, we demonstrate how to fetch and handle data if the parent property is a sub-property in other RDF repositories. This query covers all museum addresses of London city. All other systems. FedX, ANAPSI, ADERIS, used the memory index technique and our system indexed all repositories in a local server. All systems, including us, used cache storage, where systems stored the subject, object and predicate. Cache storage assisted in identifying the information as cache information was matched with the indexed data. Other systems did not generate subqueries after detecting the resources. They sent the primary SPARQL query to distributed repositories to bring the information. Our system transformed the main SPARQL query into multiple subqueries, and then each subquery was activated versus the distributed repository to bring the information. Figure 6.3 shows the results of this query from all systems. The OurSystem and FedX took less time to execute as the OurSystem fetches information from the stored index first instead of indexing in memory. FedEx did indexing in memory but performed well as required data was limited to few repositories. The query has only three variables and one filter.



```
PREFIX m:<http://allahm.museum.org/museum#>
```

```
SELECT ?museum , ?address ,?city
```

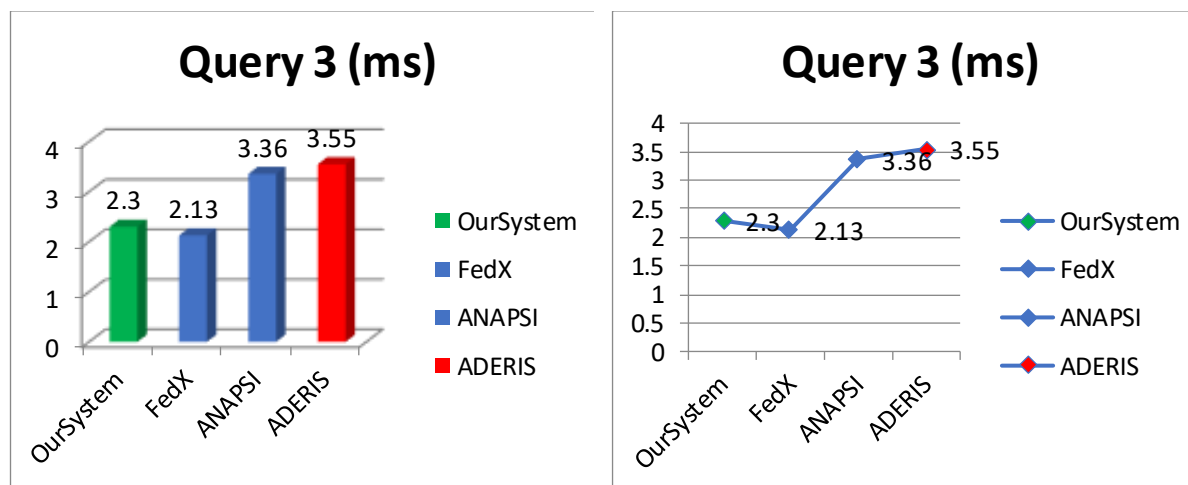
```
WHERE {
```

```
?Museum rdf:hasAddress m:Address
```

```
?Place rdf:hasCity ?city
```

```
    FILTER {city ,"London"}
```

```
}
```



**Figure 6.3 - Query 3 validation results**

Under the fourth query, we demonstrate how to fetch and handle data if a sub-property is a parent property in other RDF repositories. Here, we are asking parents' details of all artists where parent's beliefs are Christianity. For this purpose, we have introduced a new operator *Spec* (specialisation) to extract the sub-property of the parent property. Other systems had static specialisation where they could only search in the local repository for sub-property relationships. In contrast, our system had dynamic specialisation, and it searched not only

locally but also in other distributed repositories. FexX, ANAPSID, ADERIS, used the memory index technique and our system indexed all repositories in a local server. All systems, including us, used cache storage, where systems stored the subject, object and predicate. Cache storage helped to identify the resources as cache data was matched with the indexed data. Other systems did not generate subqueries after identifying the resources. They sent the main SPARQL query to distributed repositories to fetch the data. Our system converted the main SPARQL query into multiple subqueries, and then each subquery triggered against the distributed repository to fetch the data. This can be seen in Figure 6.4, which shows the results of this query from all systems. We can see that our system took less time to execute because OurSystem used dynamic specialisation features that do not exist in other systems. The query has only five variables, one filter and one dynamic specialisation function.

```
PREFIX m:<http://allahm.museum.org/museum#>

SELECT ?Parents, ?father, ?mother ?artist, ?beliefs

WHERE

?father : rdfs:subClassOf :parents.

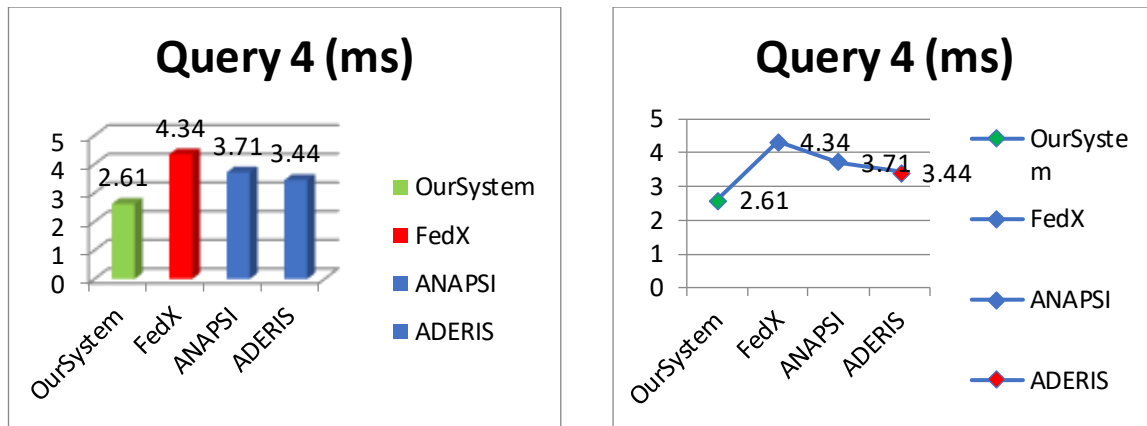
?mother : rdfs:subClassOf :parents.

?artistrdf:hasParents m: ?Parents

?parentsrdf:hasBeliefs m: ?Beliefs

        FILTER {? Beliefs ,"Christianity"}

}
```



**Figure 6.4 - Query 4 validation results**

In the fifth, we show how to fetch and handle data if the Subject is an Object in other RDF repositories. This query asks about artefacts from all repositories where oil painting is the craft, and the used material is gold. Other systems used the memory index technique, and our system indexed all repositories in a local server. All systems used cache storage, where systems stored the subject, object and predicate. Cache storage helped to identify the resources as cache data was matched with the indexed data. Other systems did not generate subqueries after identifying the resources. They sent the main SPARQL query to distributed repositories to fetch the data. Our system converted the main SPARQL query into multiple subqueries, and then each subquery triggered against the distributed repository to fetch the data. Figure 6.5 illustrates the results of this query from all systems. It can be seen that our system took less time to execute as OurSystem fetched information from the stored index first instead of memory indexing. The query has only four variables and one filter.

```
PREFIX m:<http://allahm.museum.org/museum#>
```

```
SELECT ?exhibition, ?artefacts, ?craft, ?material
```

```
WHERE {
```

```
?exhibitionrdf:contains m: ?artefacts
```

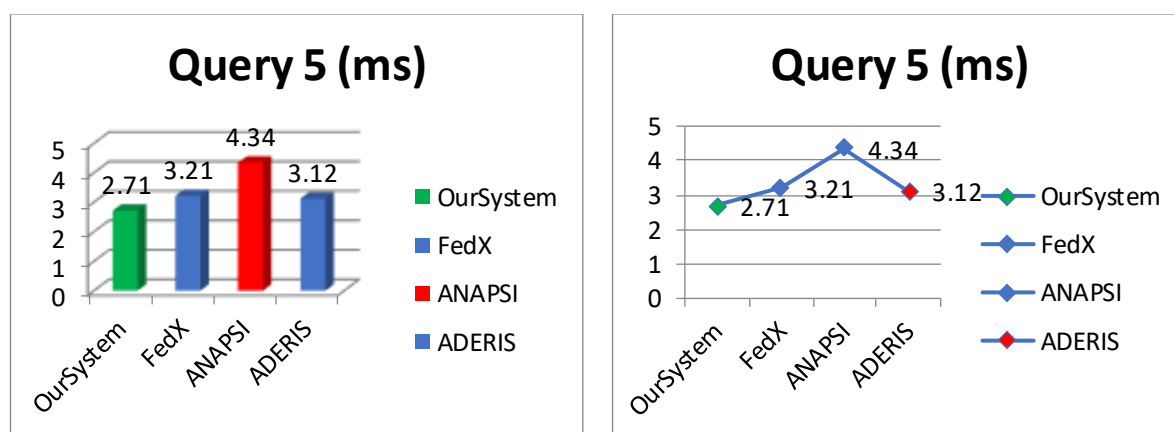
```
?artefactsrdf:hasMaterial m: ?material
```

```
?artefactsrdf:representedBy m: ?craft
```

```
    FILTER { ? craft, "OilPainting" }
```

```
        { ? material, "gold" }
```

```
}
```



**Figure 6.5 - Query 5 validation results**

In the sixth query, we show how to fetch and handle data if the Object is a Subject in other RDF repositories. This query asks about the museum's management details, which manages the exhibition from all repositories. Other systems used the memory index technique, and our system indexed all repositories in a local server. All systems had the cache storage functionality to store the subject, object and predicate. Other systems did not generate subqueries after identifying the resources. They sent the main SPARQL query to distributed repositories to fetch the data. Our system converted the main SPARQL query into multiple subqueries, and then

each subquery triggered against the distributed repository to fetch the data. Figure 6.6 shows the results of this query from all systems as we can see that our system took less time to execute as OurSystem fetched information from stored index first instead of doing live indexing. The query has only three variables and no filters.

```
PREFIX m:<http://allahm.museum.org/museum#>
```

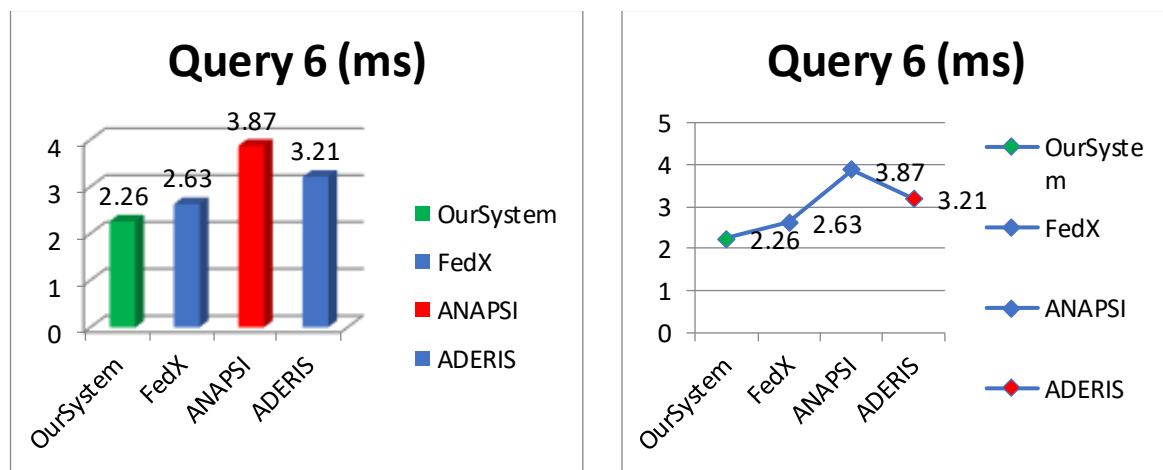
```
SELECT ?museum, ?exhibition, ?management
```

```
WHERE {
```

```
?museumrdf:hasManagement m: ?Management
```

```
?managementrdf:manages m: ?Exhibition
```

```
}
```



**Figure 6.6 - Query 6 validation results**

Under the seventh query, we check the relationship of properties between one repository's subject and object to another repository's subject and object. This would cover Asia's science

museums city's addresses. For this purpose, we have introduced a new operator *Spec* (specialisation), which extract the sub-property of parent property and child class of the parent class. Other systems did not generate subqueries after identifying the resources. They sent the main SPARQLquery to distributed repositories to fetch the data. Our system converted the main SPARQL query into multiple subqueries, and then each subquery triggered against the distributed repository to fetch the data. Other systems used the memory index technique, and our system indexed all repositories in a local server. Figure 6.7 illustrates the results of this query from all systems. We can see that our system took less time to execute as OurSystem used dynamic specialisation features that do not exist in other systems. The query has only four variables, one filter and a dynamic specialisation function.

```
PREFIX m:<http://allahm.museum.org/museum#>
```

```
SELECT ?Museum, ?Place, ?City ?Address
```

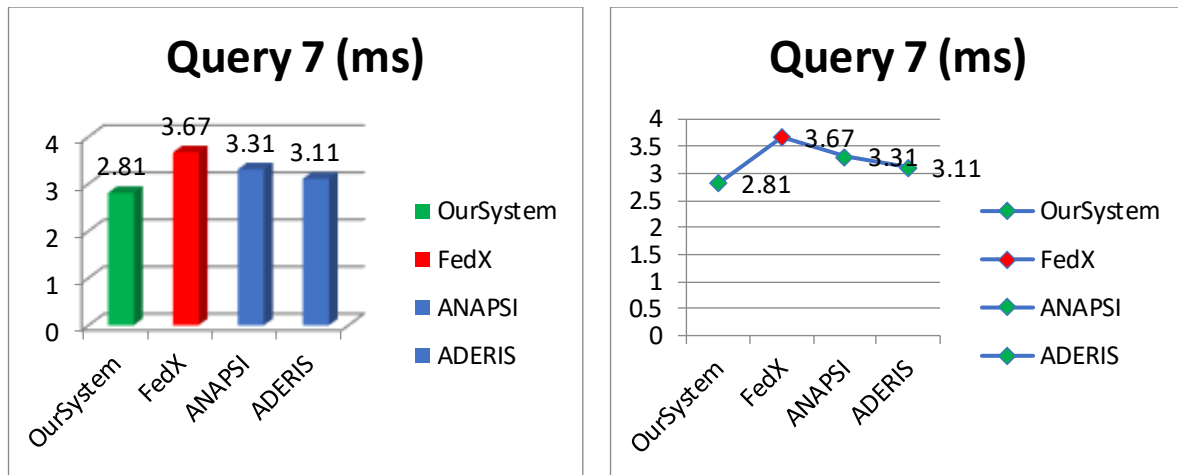
```
WHERE {
```

```
  ? address: rdfs: subClassOf: city.
```

```
  ? place rdf:hasCity m: ?city
```

```
    ? museum rdf:hasAddress m: ?address
```

```
  FILTER { ? museum, "science" } }
```



**Figure 6.7 - Query 7 validation results**

In the eighth query, we observe that Property in the second repository is a sub-property of repository 1's property between repository 2's subject and repository 1's object. Here, we are asking about all European museum's addressees which hold Asian's artist artefacts. For this purpose, we introduced and used a new operator **Gen** (generalization) which extract common characteristics between classes, subclasses, properties and sub-properties. All other systems did not have dynamic indexing, where they could not add more RDF repositories if required. Our system had the dynamic index mechanism where we first indexed all repositories locally then added more repositories into the index when required. Other systems had static generalization where they could only search in the local repository one by one for the child class of the parent class.

In contrast, our system had dynamic generalization, and it searched not only locally but also in other distributed repositories. Other systems used the memory index technique, and our system indexed all repositories in a local server. All systems used cache storage, where systems stored the subject, object and predicate. Cache storage helped to identify the resources as cache data was matched with the indexed data. Other systems did not generate subqueries after identifying the resources. They sent the main SPARQL query to distributed repositories to fetch the data. Our system converted the main SPARQL query into multiple subqueries, and then each

subquery triggered against the distributed repository to fetch the data. For this purpose, we have introduced a new operator **Gen** (generalization), which extracts the subclasses' specific parent class. Figure 6.8 shows the results of this query from all systems. It can be seen that our system took less time to execute as OurSystem used dynamic generalization features that do not exist in other systems. The query has only eight variables, two filters, and a dynamic specialisation function

```
PREFIX m:<http://allahm.museum.org/museum#>

SELECT ?Museum, ?Address, ?Region, ?Place, ?City, ?Country, ?Artist, ?Artefacts
WHERE {
  ? address: rdfs:subClassOf: city.
      ? place: rdfs:subClassOf: region.
  ? place rdf:hasCity m: ?city
      ? country rdf:hasCity m: ?city
      ? museum rdf:hasAddress m: ?address
      ? museum rdf:hasArtefacts m: ?artefacts
      ? artist rdf:hasCountry m: ?Country
  FILTER { ? region, "Europe" }
      { ? artist, "Asian" }
}
```



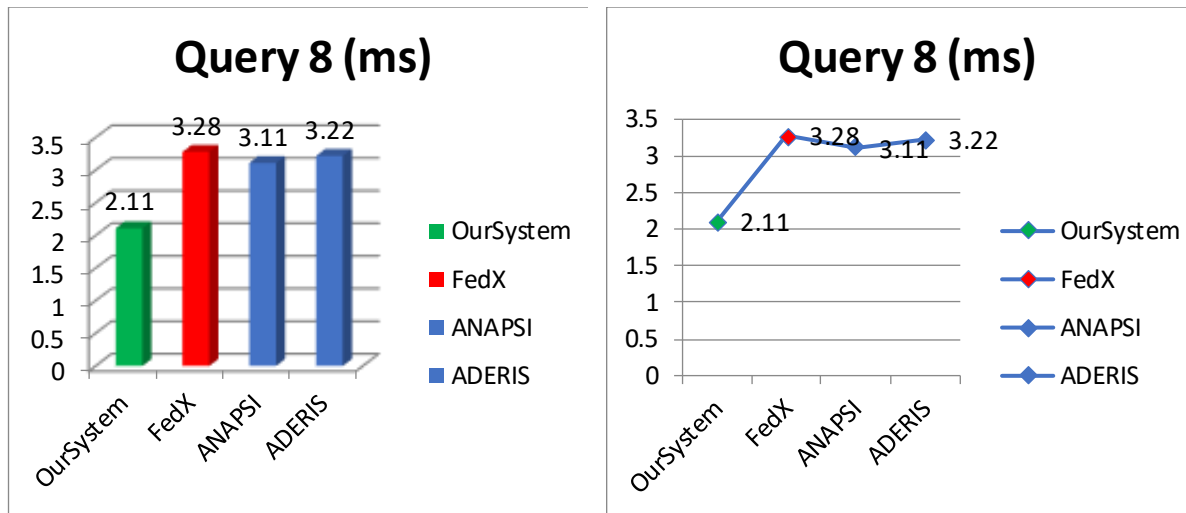


Figure 6.8 - Query 8 validation results

In the ninth query, we extend the eight queries to add more filters to demonstrate both Gen (generalization) and Spec (specialisation) operators. Here, we ask about all European museum's addresses holding Asia's artist artefacts who used oil painting craft. For this purpose, we used both operators, *Gen* (generalization) and *Spec* (specialisation), to extract common characteristics between classes, subclasses, properties and sub-properties. Other systems had static specialisation and generalization functionalities to search for sub-property and parent property relationships in the local repository. In contrast, our system had dynamic specialisation/ generalization, and it searched locally and in other distributed repositories. Figure 6.9 illustrates the results of this query from all systems. It can be seen that our system took less time to execute since OurSystem used dynamic specialisation and generalization features that do not exist in other systems. The query has only 11 variables, three filters and dynamic specialisation and generalization functions.

```
PREFIX m:<http://allahm.museum.org/museum#>
```

```
SELECT ?Painter, ?Painting, ?Craft, ?Museum, ?Address, ?Region, ?Place, ?City,
?Country, ?Artist, ?Artefacts
```

```

WHERE {
  ? address: rdfs:subClassOf: city.

    ? place: rdfs:subClassOf: region.

    ? painter: rdfs:subClassOf: artist.

    ? oilpainting: rdfs:subClassOf: painting.

    ? oilpaintng: rdfs:subClassOf: craft.

  ? painter rdf:draws m: ?painting
  ? place rdf:hasCity m: ?city
  ? place rdf:hasCity m: ?city

    ? country rdf:hasCity m: ?city

  ? museum rdf:hasAddress m: ?address
  ? museum rdf:hasArtefacts m: ?artefacts
  ? artist rdf:hasCountry m: ?Country

  FILTER { ? region, "Europe" }

    {? artist, "Asian"}

    {? painting, "oilpainting"}

```

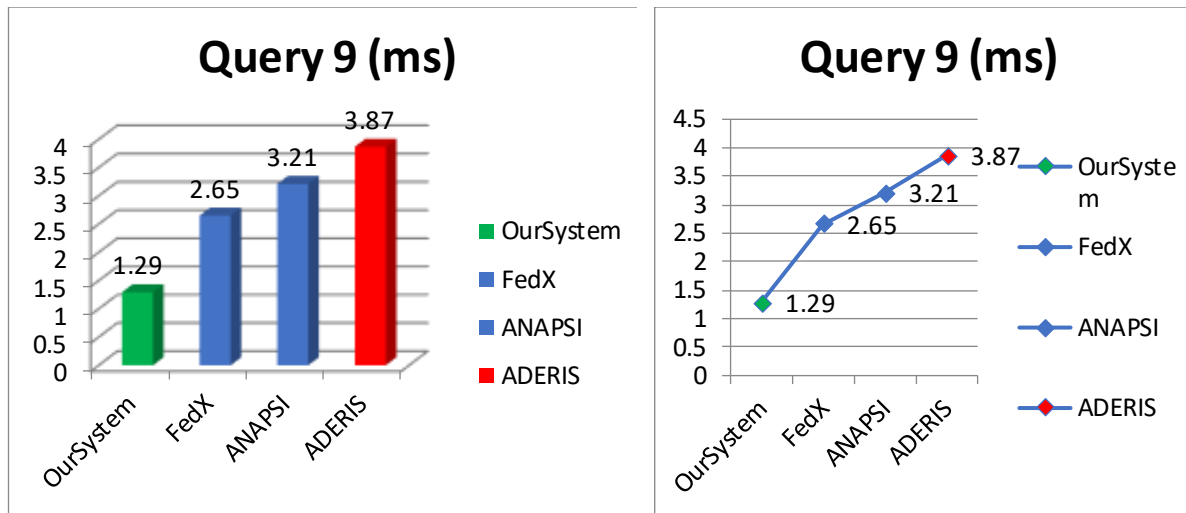


Figure 6.9 -Query 9 validation results

## 6.4 Chapter Summary

This chapter undertook a discussion on the implemented framework's performance and accuracy compared to other similar systems. Evaluation of the implemented system demonstrated that the proposed system could handle distributed SPARQL queries very efficaciously. As we discussed in [section 2.9](#), we selected **FedX**, **ANAPSID** and **ADERIS** to compare with our developed system and depicted the results in a graphical format to exemplify the performance and accuracy of all systems. We used the virtual exhibition museum's ontology that held 3600 triples for evaluation purposes and existed in 12 different RDF museum data sets. We used nine SPARQL queries against systems to demonstrate how the system responds to such queries. We required a robust machine for this evaluation, which is why we leveraged Intel i5 with two core and 8GB RAM. In addition, we utilised Protégé software to build the ontology of the Virtual Museum Exhibition. Finally, we configured Apache Jena Fuseki 3, a SPARQL server, to handle the queries

# Chapter 7

## Conclusion and Future work

### 7.1 Summary of the thesis

In this thesis, we have presented a contribution that addresses the research problems explicitly. One of the challenges this research faced was extracting required semantic data from distributed RDF repositories through SPARQL. Retrieving data from distributed RDF data sources is time-intensive. An optimised and proper structure is required because SPARQL queries are triggered to the distributed end to retrieve data. As the author discussed in [chapter 2](#), the existing system clarifies that they retrieve data directly from distributed RDF repositories without using a centralised index mechanism. Direct access to RDF repositories without using a centralised index mechanism can not be adequate for an unlimited number of distributed RDF repositories. The author compared the proposed system, in [chapter 6](#), with other similar systems. Other similar systems were **FedX**, **ANAPSID** and **ADERIS**. After validating the chosen systems, we can conclude that these systems work well with limited RDF data sources. The author discussed in chapter 2 the ins and outs of the foundation technology behind the semantic web. The author included a detailed review of the existing work done by other authors and a critical review of their work. It talks explicitly about data the types and approaches of data integration, and Distributed Query Processing System generates optimised query plans for Distributed Query Processing, various Response Time Models. Then, the chapter touches upon and explicates other Query Execution techniques before moving onto the investigation of a Query Federation system of data processing and introspect on its various standards and then briefly discuss what Adaptive Query Operators require. Consequently, [section 2.9](#) explores

deep into Query Processing Systems, such as ANAPSID, ADAERIS, SYMMETRIC INDEX HASH JOIN, SPLENDID, SemWIQ and DARQ.

Given that one of the thesis aims was to put forward the complete architecture to fetch RDF data from distributed sources efficiently, we discussed all our phases of the proposed architecture. As a case in point, we discussed indexing, algebraic notations, introducing Specialisation and Generalisation operators, caching mechanism, identifying resources from the cache, generating subqueries, and joining the subquery results. This thesis has provided all algorithms of the planned architecture. For this purpose, we have utilized Apache Jena Fuseki framework for handling SPARQL queries. For this purpose, Protégé software was used to develop a virtual museum ontology. We undertook the development of a system using Java under the planned algorithms to test our system. We chose nine different case scenarios in this process, as presented in chapter 6, that fulfilled all distributed fetching from all angles, such as dynamic indexing, fetching data from distributed RDF data sets, joins, merging, specialisation, and generalisation.

We chose three different similar systems to make the comparisons. These systems are some of the more popular ones to retrieve data from distributed RDF data sets. The architecture of these systems is different from our proposed system. Query results very clearly point out that our proposed system is better than other systems. The difference between our system and other systems was that in our system, we first created the index for all participants given data sets, and other systems were indexing directly in the memory, which is hugely time intensive. We converted the main SPARQL query into algebraic expression in our proposed system before extracting triples and variables information to store them inside the cache. This enabled the system to identify the data source, where data exist, and subqueries generated according to identified data sources. Other systems fetched the data directly from the data sources before joining them, whereas our system used cache to identify data sources and generating

subqueries. We introduced two new dynamic operators, Specialisation and Generalisation, to fetch the semantic data from parent and child nodes. The testing mechanism encompassed all complex scenarios and helped us to evaluate our proposed system that effectively fetched distributed RDF data sets.

Museum ontology was developed, and Simplified Agile Methodology (SAMOD) methodology was adopted for Museum's ontology Development ([Appendix A](#)). SAMOD focuses on iterative tests to ensure that the developed ontology is consistent and matches the requirements. Different tests were performed on this ontology, and these tests were model tests, data tests and query tests. The purpose of choosing this methodology was that it was very lightweight, and it had three simple stages: understanding the requirements, Merging the Ontology, and Refactoring the main ontology branch. The author utilized the CIDOC Conceptual Reference Model (CRM), a theoretical and practical technique for information integration within cultural heritage. It can help scientists, controllers, and the public check out complex queries regarding our history across numerous and distributed datasets. The CIDOC CRM achieves that by simply providing meanings and a proper design for explaining the implicit and specific concepts and relationships employed in cultural heritage documentation and primary interest for querying and exploring such details.

The author used the Apache Jena framework for a java programming language to develop a proposed framework in java libraries. It helped the author to manage the various semantic components of the semantic web and linked-data application to conform to the standards of the W3C. Since 2000, Jena is an open-source project developed by researchers at HP Laboratories in Bristol city in the UK and later became widely used. It was a success to become part of the Apache Software Foundation in November of the year 2010. The author used the unit and functional testing techniques as the proposed framework have different individual units which works together. Other testing techniques were compared to chosen testing strategies, e.g.,

integration testing, system testing, regression testing, acceptance testing, component testing and performance testing. The unit testing and functional technique have tested the developed framework. The author used the Museum ontology to test and evaluate the developed system. It demonstrated all how the complete developed and processed system works. Different types of tests have been performed in this thesis, like the algebraic operator's test (e.g., select, join, outer join, generalization, and specialisation operators test) and test the proposed algorithm. Test results showed that all developed system units worked as expected, and no errors were found during testing all phases of the tested framework.

The purpose behind the test was that the developed system should function and fulfil all the objectives specified in chapter 1 and perform what it is expected to do. Generally, **testing** has been performed throughout the development process to determine whether the developed system fulfils the specified requirements. **Testing** has been performed by running the whole phases of the framework. This ensured that the developed system fulfils the requirements. It also determined to show that the developed software satisfies its purpose when arranged in a specific environment. This process replied to the question, "Are we developing the right product or not?". Testing techniques had become very much more manageable because in the unit and functional testing, each part or unit of the developed system was tested first, and after this, the whole program was tested. In unit testing, the author examined each phase of the developed system individually in a sequence. Finally, the author evaluated the performance of the implemented framework and its accuracy with other related systems. Against this framework, the evaluation demonstrated that the proposed system could efficiently handle distributed SPARQL queries. In particular, the author compared the proposed system with other similar subdivision of systems. These systems were FedX, ANAPSID and ADERIS. These systems were selected because of their similar functionalities proposed in our system. All chosen systems under this evaluation have implemented the triple pattern for the SPARQL

endpoints, which holds similarity with our proposed system. These systems' functions prevent the user from starting the URL to fetch data from distributed resources instead of overwhelming the amount of complex data. Our selection choice based on the fact that these systems are based on Sesame and their models implemented the join. Generally, the other system's proficiency goes down when adding or merging more RDF data sources as they used memory index. The selected system's query plan does include data from the triple pattern, and query performance goes up when all RDF sources are mentioned in the SPARQL query. However, when these systems tried to add more RDF sources after query results, they were not as accurate as they were supposed to be. These systems first obtained results from RDF sources which frequently got a no-connection error if the required data source was unavailable.

## **7.2 Originality and Contribution**

This research aimed to offer an approach that enables the accessing of distributed RDF information. This process is followed by combining the results obtained to evaluate the validity of the research study. This research has made the following original contributions.

- Design and implementation of an efficient framework using indexing technique for querying ontologies.
- Developed formal Specification of a semantic algebra of the ontological queries.
- Developed algorithms for translating the global queries into algebraic expressions.
- Developed algorithms for splitting the global queries into a set of independent subqueries that can be executed locally by translating them into expressions of semantic algebra.



- Developed algorithms for aggregating the results of the execution of the subqueries.

The author presented the methods, technologies, and elements to achieve the desired outcome. The author introduced the framework and operators involved in developing the system. It also showed how each element was combined to achieve the common goal of validating the research hypothesis. The author proposed the conceptual framework upon which the research methodology functions to help in the query execution process by gaining access to the data present within distributed RDF sets across a database. The methodology to be used also involved elements significant to the developed system. [Chapter 4](#) introduced such elements as the semantic algebra involved in converting a traditional SPARQL query. The author elaborated the concepts included in the selection, projection, joins, specialisation and generalisation operators. These operators were usually in assistance during the process of processing and converting a query. After applying these operators, the system converted a query into its primary algebraic expression. Accordingly, chapter 4 proposed the algorithms behind the conceptual framework. The algorithms as substantiated in this chapter included the procedural RDF indexing algorithm, converting the main SPARQL query into the sub-queries algorithm, and joining the results algorithm. These algorithms worked collectively to start and end to facilitate the developed query processing system. Semantic algebra is the symbolic mathematical language that was used to represent semantic data. In simpler terms, the function of semantic algebra was to break down semantic information into the most basic, raw form of mathematical data that could make inference accurately by a computerized system. Semantic algebra essentially helped in detailing systems down to a mini level. This was precisely why the technology of semantic algebra played such a significant role in the research. SPARQL query was converted into its algebraic notations. This process was usually done by using semantic operators. Semantic operators refer to operators that perform their tasks based on the

semantic context of information. This implies that semantic operators can manipulate a given text and convert it into its semantic algebraic notation. For this research, semantic operators have been used to convert SPARQL queries into their algebraic forms.

The process was refined by addressing the need to aggregate all relevant information from various RDF sources instead of throwing up just one result. It was made possible by breaking up the main SPARQL query into sub-queries –the individual answers produced a comprehensive response. The basic RDF pattern of the <Subject, Object, Predicate> triple model was employed. This simple semantic triple pattern helped to optimise RDF data in creating indexing for all participant data sets instead of indexing in the memory. A step-by-step process was adopted. Multiple algorithms were developed to translate the SPARQL query into an algebraic expression, converted the main SPARQL query into subqueries, and carried out the SPARQL queries search in distributed ontologies. Finally, the author formulated an algorithm to combine the subqueries results. Two new operators, Generalisation and Specialisation, were proposed to access RDF parent and child nodes. In conclusion, the proposed/developed system allowed dynamic indexing, sourcing data from distributed RDF sets, identifying resources from cache, merging results, specialisation, generalisation, fetching parent and child nodes.

### **7.3 Limitations and Future Recommendations**

This section discusses the research problems that continue to exist and does not form part of this study. In this thesis, our main achievement is to index all the participated data sets and propose a comprehensive mechanism of accessing distributed RDF data sets via the generation of algebraic expression from the main query. All data was stored in a temporary cache. Converting the main SPARQL query into sub-queries and then sending each subquery to separate data sets before combining the returning results.

Future work may need to research how to index all different datasets as we successfully indexed them in the homogeneous environment in this research. However, when we apply the same proposed architecture to a heterogeneous environment, the results are inaccurate as they have been with the homogeneous environment. There is also a need to research identifying ways of retrieving data from different models; in this thesis, we used an objected-oriented model. Proper research must fetch data from other formats, such as the relational model, XML format. Eventually, the objective of semantic data is to generate interlink gigantic amounts of data. In the semantic web world, millions of triplets are already connected and available on demand. This research showed how to get all similar domain data sets in the first instance before indexing them all on a local or remote server. However, more research needs to be conducted on directly fetching all participated cross domains and different model from their location and indexing them locally. Furthermore, there needs to be a mechanism that data must be updated on the stored index if it is changed or updated.

# References

- Shadbolt, N., Berners-Lee, T. and Hall, W. (2006). The Semantic Web Revisited. IEEE Intelligent Systems, 21(3), pp.96–101.
- Heath, T. (2010). A taskonomy for the Semantic Web. Semantic Web, 1(1,2), pp.75–81.
- Khozoie, N. (2012). Health Information Management On Semantic Web :(Semantic HIM). International journal of Web & Semantic Technology, 3(1), pp.61–68.
- Fazzinga, B. and Lukasiewicz, T. (2010). Semantic search on the Web. Semantic Web, 1(1,2), pp.89–96.
- Dubinina, A.V., Yang, C.-W. and Vyatkin, V.V. (2020). THE USE OF SPARQL LANGUAGE IN ONTOLOGICAL MODELING OF MULTI-AGENT SYSTEMS IN SEMANTIC WEB. University proceedings. Volga region. Technical sciences, (1).
- Siddiqui, F. and Alam, M.A. (2011). Web Ontology Language Design and Related Tools: A Survey. Journal of Emerging Technologies in Web Intelligence, 3(1).
- Yang, S., Guo, J. and Wei, R. (2017). Semantic interoperability with heterogeneous information systems on the internet through automatic tabular document exchange. Information Systems, 69, pp.195–217.
- Simplerl, E. (2009). Reusing ontologies on the Semantic Web: A feasibility study. Data & Knowledge Engineering, 68(10), pp.905–925.
- Hitzler, P. and Janowicz, K. (2010). Semantic Web – Interoperability, Usability, Applicability. Semantic Web, 1(1,2), pp.1–2.
- ZHANG, Z. and YANG, T. (2011). SPARQL ontology query based on natural language understanding. Journal of Computer Applications, 30(12), pp.3397–3400.

Arul, U. and Prakash, S. (2020). Toward automatic web service composition based on multilevel workflow orchestration and semantic web service discovery. *International Journal of Business Information Systems*, 34(1), p.128.

Sakellariou, A.E. (2019). Job Position: Data steward. [online] Duchenne Data Foundation. Available at: <https://www.duchennedatafoundation.org/job-position-data-steward/> [Accessed 16 Jun. 2020].

Shah, V. (2016). Comparative Study Of Semantic Search Engines. *International Journal Of Engineering And Computer Science*.

Retracted: Semantic Information Integration with Linked Data Mashups Approaches. (2015). *International Journal of Distributed Sensor Networks*, 11(12), p.431342.

Development of a CUBRID-Based Distributed Parallel Query Processing System. (2017). *Journal of Information Processing Systems*.

Chahal, P. and Singh, M. (2021). An Efficient Approach for Ranking of Semantic Web Documents by Computing Semantic Similarity and Using HCS Clustering. *International Journal of Semiotics and Visual Rhetoric*, 5(1), pp.45–56.

Hammami, R., Bellaaj, H. and Kacem, A.H. (2018). Semantic Web Services Discovery. *International Journal on Semantic Web and Information Systems*, 14(4), pp.57–72.

Abid, A., Rouached, M. and Messai, N. (2019). Semantic web service composition using semantic similarity measures and formal concept analysis. *Multimedia Tools and Applications*, 79(9-10), pp.6569–6597.

Appreciation to distributed and parallel databases reviewers. (2018). *Distributed and Parallel Databases*, 36(1), pp.1–3.

Babu, S. (2012). Massively Parallel Databases and MapReduce Systems. *Foundations and Trends® in Databases*, 5(1), pp.1–104.

Osman, I., Ben Yahia, S. and Diallo, G. (2021). Ontology Integration: Approaches and Challenging Issues. *Information Fusion*, 71, pp.38–63.

Moeller, B. and Frings, C. (2014). Long-term response-stimulus associations can influence distractor-response bindings. *Advances in Cognitive Psychology*, 10(2), pp.68–80.

Zhu, L. (2015). Processing Recommender Top-N Queries in Relational Databases. *Journal of Software*, 10(2), pp.162–171.

Almourad, M.B. (2013). Measuring Database Objects Relatedness in Peer-to-Peer Databases. *International Journal of Computer and Communication Engineering*, pp.289–293.

Korneva, L.A. and Khorev, P.B. (2018). Development of framework secure application with client-server architecture. *Informacionno-technologicheskij vestnik*, 15(1), pp.112–119.

Devulapalli, K. and Bagui, S. (2018). Comparison of Hive's query optimisation techniques. *International Journal of Big Data Intelligence*, 5(4), p.243.

Sinuraya, J., Rezky, S. F., & Tarigan, M. (2019). Data Search Using Hash Join Query and Nested Join Query. *Journal of Physics: Conference Series*, 1361, 012079. <https://doi.org/10.1088/1742-6596/1361/1/012079>.

Chavan, Mr.Prashant.S. and Phulpagar, Prof.Dr.B.D. (2016). Adaptive Query Interface for Database Search. *International Journal Of Engineering And Computer Science*.

Chen, H., Zeng, Q., Zhang, Y. and Tang, D. (2018). Performance evaluation of main-memory hash joins on KNL. *International Journal of Computational Science and Engineering*, 1(1), p.1.

Achichi, M., Bellahsene, Z., Ellefi, M.B. and Todorov, K. (2019). Linking and Disambiguating Entities Across Heterogeneous RDF Graphs. SSRN Electronic Journal.

Acosta, M., Vidal, M.-E., Lampo, T. and Castillo, J. eds., (2011). ANAPSID: An Adaptive Query Processing Engine for SPARQL Endpoints. The Semantic Web - ISWC 2011 - 10th International Semantic Web Conference At: Bonn, Germany, 7031.

Kim, B., Kim, D., Ko, G., Noh, Y., Lim, J., Bok, Lee, B. and Yoo, J. (2017). A Distributed SPARQL Query Processing Scheme Considering Data Locality and Query Execution Path. KIISE Transactions on Computing Practices, 23(5), pp.275–283.

Liu, Y., He, Z. and Meng, X.W. (2018). Accelerating Hash Join Performance by Exploiting Data Distribution. International Journal of Computer Trends and Technology, 56(1), pp.6–20.

Saleem, M., Khan, Y., Hasnain, A., Ermilov, I. and Ngonga Ngomo, A.-C. (2016). A fine-grained evaluation of SPARQL endpoint federation systems. Semantic Web, 7(5), pp.493–518.

Langegger, A. and Wöß, W. (2008). SemWIQ - Semantic Web Integrator and Query Engine. 3rd International Applications of Semantic Web Workshop (AST'08)At: Munic, Germany, Informatik 2008.

Quilitz, B. and Leser, U. (2008). Querying Distributed RDF Data Sources with SPARQL. ESWC , volume 5021 of Lecture Notes in Computer Science, page 524-538. Springer, (2008), p.

Mishra, J.P. and Mishra, S.K. (2017). Evaluating performance with implementation of virtualised data in the cloud using metaheuristic approach. International Journal of Knowledge Engineering and Data Mining, 4(3/4), p.187.

tyagi, G. (2015). Ontology Based Fuzzy Query Execution. American Journal of Networks and Communications, 4(3), p.16.

Sasak, A. and Brzuszek, M. (2010). Speculative execution plan for multiple query execution systems. Annales UMCS, Informatica, 10(2).

Poovammal, E. and Ponnaivaikko, M. (2010). Utility Independent Privacy Preserving Data Mining - Horizontally Partitioned Data. Data Science Journal, 9, pp.62–72.

Daenen, J., Neven, F., Tan, T. and Vansummeren, S. (2016). Parallel evaluation of multi-semi-joins. Proceedings of the VLDB Endowment, 9(10), pp.732–743.

Tang, D., Zhang, Y., Zeng, Q. and Chen, H. (2019). Performance evaluation of main-memory hash joins on KNL. International Journal of Computational Science and Engineering, 20(4), p.425.

XU, S. and HONG, M. (2012). Translating SQL Into Relational Algebra Tree-Using Object-Oriented Thinking to Obtain Expression Of Relational Algebra. International Journal of Engineering and Manufacturing, 2(3), pp.53–62.

Rahim, R., Nurarif, S., Ramadhan, M., Aisyah, S. and Purba, W. (2017). Comparison Searching Process of Linear, Binary and Interpolation Algorithm. Journal of Physics: Conference Series, 930, p.012007.

Qudus, U., Saleem, M., Ngonga Ngomo, A.-C. and Lee, Y.-K. (2021). An empirical evaluation of cost-based federated SPARQL query processing engines. Semantic Web, pp.1–26.



Rakhmawati, N. and Fadzilah, L., 2019. Dataset Characteristics Identification for Federated SPARQL Query. *Scientific Journal of Informatics*, 6(1), pp.23-33.

Kurgaev, A.F. (2018). New definition of the SPARQL — query language for the Semantic Web. Reports of the National Academy of Sciences of Ukraine, (11), pp.19–31.

Kaneko, S. and Chishiro, E. (2018). Query Conversion for Aggregate Operations in Distributed SPARQL Query Processing Using Summary Information. *Journal of Information Processing*, 26(0), pp.747–754.

Carstensen, A.-K. and Bernhard, J. (2018). Design science research – a powerful tool for improving methods in engineering education research. *European Journal of Engineering Education*, 44(1-2), pp.85–102.

Divyani Shivkumar Taley (2020). Comprehensive Study of Software Testing Techniques and Strategies: A Review. *International Journal of Engineering Research and*, V9(08).

Anwar, N. and Kar, S. (2019). Review Paper on Various Software Testing Techniques & Strategies. *Global Journal of Computer Science and Technology*, pp.43–49.

Jani, K. and Dr. V.M. Chavda, Dr.V.M.C. (2011). A Study on Semantic Web Framework: JENA and Protégé. *Indian Journal of Applied Research*, [online] 4(1), pp.143–145. Available at: [https://www.worldwidejournals.com/indian-journal-of-applied-research-\(IJAR\)/recent\\_issues\\_pdf/2014/January/January\\_2014\\_1388583791\\_ee78c\\_43.pdf](https://www.worldwidejournals.com/indian-journal-of-applied-research-(IJAR)/recent_issues_pdf/2014/January/January_2014_1388583791_ee78c_43.pdf) [Accessed 1 Dec. 2019].

Abdelghany, A., Darwish, N. and Hefni, H. (2019). An Agile Methodology for Ontology Development. *International Journal of Intelligent Engineering and Systems*, 12(2), pp.170–181.

Gaitanou, P. and Gergatsoulis, M. (2012). Defining a semantic mapping of VRA Core 4.0 to the CIDOC conceptual reference model. *International Journal of Metadata, Semantics and Ontologies*, 7(2), p.140.

ZHANG, G. and XU, Q. (2009). Design of distributed search engine system for apparel □ based on semantic Web services. *Journal of Computer Applications*, 29(6), pp.1601–1604.

Zangenehpour, S., Ali Seyyedi, M. and Mohsenzadeh, M. (2012). A New Framework for Mapping Business Domain Ontologies. *International Journal of Computer Applications*, 55(12), pp.16–20.

Hong, J.L. (2016). Automated Data Extraction with Multiple Ontologies. *International Journal of Grid and Distributed Computing*, 9(6), pp.381–392.

Jagvaral, B., Lee, W., Kim, K.-P. and Park, Y.-T. (2015). SPARQL Query Processing in Distributed In-Memory System. *Journal of KIISE*, 42(9), pp.1109–1116.

Tomaszук, D. and Hyland-Wood, D. (2020). RDF 1.1: Knowledge Representation and Data Integration Language for the Web. *Symmetry*, 12(1), p.84.

Oguz, D., Yin, S., Ergenç, B., Hameurlain, A. and Dikenelli, O., 2017. Extended Adaptive Join Operator with Bind-Bloom Join for Federated SPARQL Queries. *International Journal of Data Warehousing and Mining*, 13(3), pp.47-72.

Song, S., Huang, W. and Sun, Y. (2017). Semantic query graph based SPARQL generation from natural language questions. *Cluster Computing*, 22(S1), pp.847–858.

Rabhi, A. and Fissoune, R. (2019). WODII: a solution to process SPARQL queries over distributed data sources. *Cluster Computing*.

Abdelaziz, I., Harbi, R., Khayyat, Z. and Kalnis, P. (2017). A survey and experimental comparison of distributed SPARQL engines for very large RDF data. Proceedings of the VLDB Endowment, 10(13), pp.2049–2060.

Mohammad Shahabuddin, S. and Prasanth, Y. (2016). Integration Testing Prior to Unit Testing: A Paradigm Shift in Object-Oriented Software Testing of Agile Software Engineering. Indian Journal of Science and Technology, 9(20).

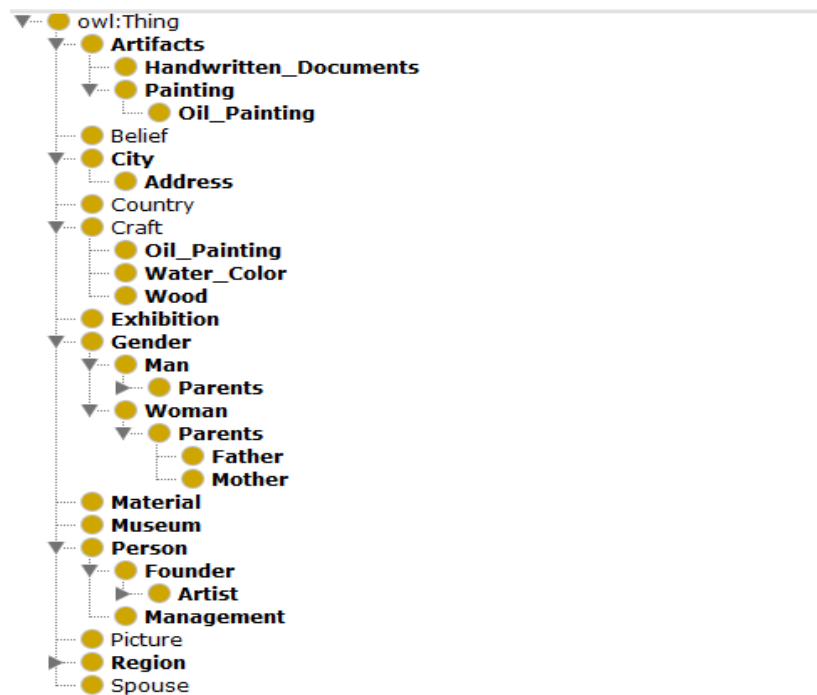
Sam, K.M. (2019). Unit Testing to Support Reusable for Component-Based Software Engineering. International Journal of Trend in Scientific Research and Development, Volume-3(Issue-2), pp.638–640

# Appendix A: Museum Ontology

## Classes and Properties

This segment portrays the components of the MUSEUM ontology, which we are going to use in our Virtual exhibitions.

## Classes



## Museum Ontology's Classes

## Entities

<b>Subclass</b>	None
<b>Description</b>	The primitive category having no differentiae

## Entity Class

## Artefacts

<b>Subclass</b>	None
<b>Description</b>	It is a subclass of none and stores information about artefacts of museums

### **Artefacts Class**

#### **Painting**

<b>Subclass</b>	Artefacts
<b>Description</b>	It is a subclass of Artefacts and stores information about paintings

### **Painting Class**

#### **Handwritten Documents**

<b>Subclass</b>	Artefacts
<b>Description</b>	It is a subclass of Artefacts and stores information about hand written documents

### **Handwritten Documents**

#### **Artist**

<b>Subclass</b>	Founder
<b>Description</b>	It is a subclass of Founder class and stores information about Artist

### **Artist Class**

#### **Painter**

<b>Subclass</b>	Artist
<b>Description</b>	It is a subclass of Artist class and stores information about painters

#### **Painter Class**

#### **Writer**

<b>Subclass</b>	Artist
<b>Description</b>	It is a subclass of Artist and stores information about writers

#### **Writer Class**

#### **Oil painting**

<b>Subclass</b>	Craft
<b>Description</b>	It is a subclass of craft class and stores information about oil paintings

#### **Oil Painting Class**

#### **Water Colour**

<b>Subclass</b>	Craft
<b>Description</b>	It is a subclass of craft class and stores information about water colour

#### **Water Colour Class**

#### **Wood**

<b>Subclass</b>	Craft
-----------------	-------

<b>Description</b>	It is a subclass of craft class and stores information about wood
--------------------	---

### **Wood Class**

### **Place**

<b>Subclass</b>	Region
<b>Description</b>	It is a subclass of region class and stores information about region

### **Place Class**

### **Man**

<b>Subclass</b>	Gender
<b>Description</b>	It is a subclass of gender class and stores information about male category

### **Man Class**

### **Women**

<b>Subclass</b>	Gender
<b>Description</b>	It is a subclass of gender class and stores information about women

### **Women Class**

### **Father**

<b>Subclass</b>	Parents
<b>Description</b>	It is a subclass parents of and stores information about fathers

### **Father Class**

### **Mother**

<b>Subclass</b>	Parents
<b>Description</b>	It is a subclass of parent and stores information about mothers

### Mother Class

#### Belief

<b>Subclass</b>	None
<b>Description</b>	It is a subclass none of and stores information about person's beliefs

### Belief Class

#### Beginning-of-Existence

<b>Subclass</b>	Entity
<b>Description</b>	A primitive ontology category for sub-classing categories of entities that provide time existential contexts

### End-of-Existence

<b>Subclass</b>	Entity
<b>Description</b>	A primitive ontology category for sub-classing categories of entities that provide end of existence of object

#### Temporalities

<b>Subclass</b>	Entity
<b>Description</b>	A primitive ontology category for sub-classing categories of entities that provide end of existence of object

#### Actualities

<b>Subclass</b>	<i>Entity</i>
<b>Description</b>	A primitive ontology category for sub-classing categories of entities that have a tangible existence in some world view



## Abstractions

<b>Subclass</b>	<i>Entity</i>
<b>Description</b>	A primitive ontology category for sub-classing categories of entities that are pure information or concepts.

## Events

<b>Subclass</b>	<i>Temporality</i>
<b>Description</b>	An <i>Event</i> marks a transition between <i>Situations</i> , one that is associated with the event through a <i>precedes</i> property and another through a <i>follows</i> property

## Situations

<b>Subclass</b>	<i>Temporality</i>
<b>Description</b>	A <i>Situation</i> is a context for making time dependent or <i>existential</i> assertions about <i>Actualities</i> .

## Actions

<b>Subclass</b>	<i>Temporality</i>
<b>Description</b>	An activity or verb performed by some <i>Agent</i> or <i>Agents</i> in the context of an <i>Event</i> .

## Agents

<b>Subclass</b>	<i>Actuality</i>
<b>Description</b>	An <i>Actuality</i> that is present during an <i>Event</i> or is the party of some <i>Action</i> . <i>Agents</i> may be persons, instruments, organizations, etc.

## Works

<b>Subclass</b>	<i>Abstraction</i>
<b>Description</b>	A <i>Work</i> is an abstract concept which cannot exist in a model in isolation, but is only revealed when it has been actualized in some <i>Manifestation</i> .

## Manifestations

<b>Subclass</b>	<i>Artifact</i>
<b>Description</b>	A form of an <i>Artifact</i> that stands as the sensible realization of a <i>Work</i> . <i>Works</i> and <i>Manifestations</i> stand in a one too many relationship

## Items

<b>Subclass</b>	<i>Artifact</i>
<b>Description</b>	A form of an <i>Artifact</i> used to establish a set of identical copies.

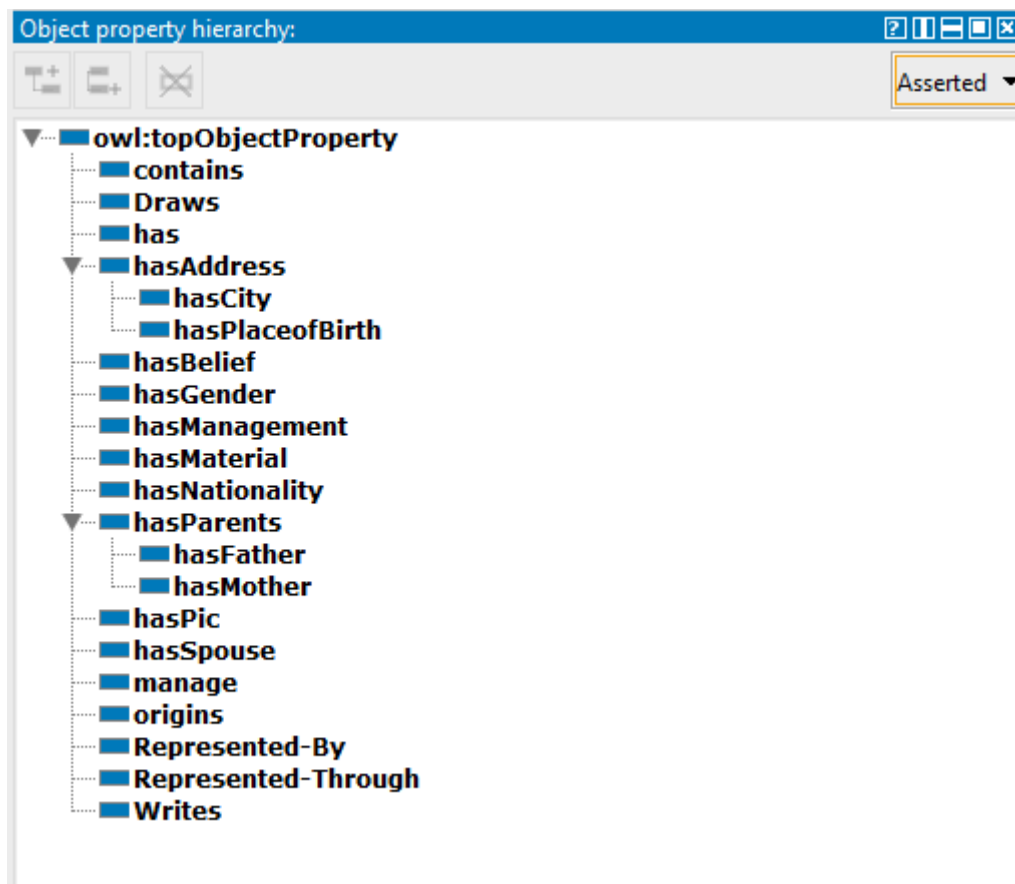
## Time

<b>Subclass</b>	<i>Entity</i>
<b>Description</b>	An entity which represents either a time span or point in time and which can be used to confine the temporal extent of <i>Temporalities</i> ( <i>Events</i> or <i>Situations</i> ).

## Places

<b>Subclass</b>	<i>Entity</i>
<b>Description</b>	An entity which represents spatial location. It can be used to specify the location of either <i>Temporalities</i> ( <i>Events</i> and <i>Situations</i> ) or <i>Actualities</i> .

## Properties

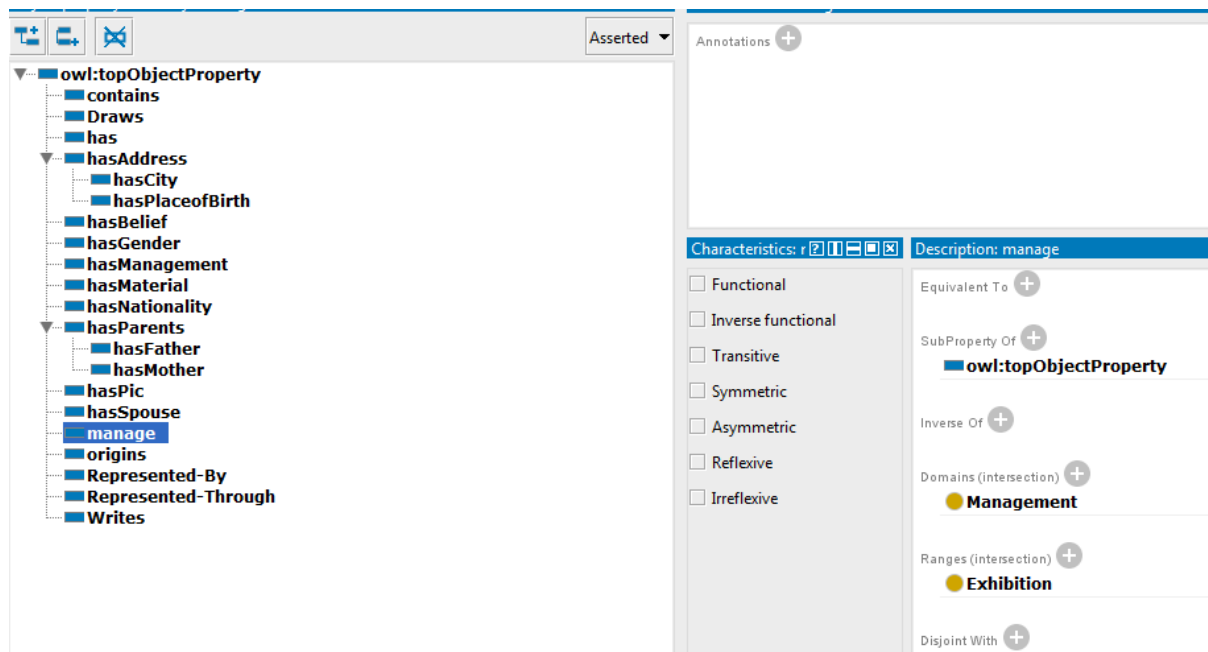


## Museum Ontology's Properties

### Manage

Sub Property	TopObjectProperty
Domain	Management
Range	Exhibition
Description	It connects Management and Exhibition and make sense to triple

### manage property



## Contains

Sub Property	TopObjectProperty
Domain	Exhibition
Range	Artefacts
Description	It connects Exhibition and Artefacts and make sense to triple

**contains property**

The screenshot displays a software interface for managing ontological properties. On the left, a hierarchical tree shows the structure of properties, with 'owl:topObjectProperty' at the root. It includes sub-properties such as 'contains', 'Draws', 'has', 'hasAddress', 'hasCity', 'hasPlaceofBirth', 'hasBelief', 'hasGender', 'hasManagement', 'hasMaterial', 'hasNationality', 'hasParents', 'hasFather', 'hasMother', 'hasPic', 'hasSpouse', 'manage', 'origins', 'Represented-By', 'Represented-Through', and 'Writes'. The right panel provides a detailed view of the selected 'contains' property. It shows characteristics like Functional, Inverse functional, Transitive, Symmetric, Asymmetric, Reflexive, and Irreflexive. The description section indicates that 'contains' is equivalent to 'owl:topObjectProperty', is a sub-property of 'owl:topObjectProperty', and has a domain of 'Exhibition' and a range of 'Artifacts'.

## Draws

Sub Property	TopObjectProperty
Domain	Painter
Range	Painting
Description	It connects Painter and painting and make sense to triple

## draws property

This screenshot shows the same software interface as the first one, but with the 'Draws' property selected. The left-hand tree remains the same. The right-hand panel now displays the details for 'Draws'. It shows the same set of characteristics (Functional, Inverse functional, Transitive, Symmetric, Asymmetric, Reflexive, Irreflexive). The description section indicates that 'Draws' is equivalent to 'owl:topObjectProperty', is a sub-property of 'owl:topObjectProperty', and has a domain of 'Painter' and a range of 'Painting'.

## hasAddress

Sub Property	TopObjectProperty
Domain	Museum
Range	Address
Description	It connects Museum and Address and make sense to triple

## hasAddress property

The screenshot shows a web ontology editor interface. On the left, a tree view displays the ontology structure, with 'owl:topObjectProperty' expanded to show 'hasAddress'. The main panel on the right is titled 'Description: hasAddress' and contains a 'Characteristics' section with several unchecked checkboxes: Functional, Inverse functional, Transitive, Symmetric, Asymmetric, Reflexive, and Irreflexive. Below this, the 'Equivalent To' section is empty. The 'SubProperty Of' section shows 'owl:topObjectProperty'. The 'Inverse Of' section is empty. The 'Domains (intersection)' section shows 'Museum'. The 'Ranges (intersection)' section shows 'Address'.

## hasBelief

Sub Property	TopObjectProperty
Domain	Person
Range	Belief
Description	It connects Person and Belief and make sense to triple

## hasBelief property

The screenshot displays a web interface for the 'hasBelief' property. On the left, a tree view shows the 'Object property hierarchy: hasBelief' with a list of properties including 'owl:topObjectProperty', 'contains', 'Draws', 'has', 'hasAddress', 'hasCity', 'hasPlaceofBirth', 'hasBelief', 'hasGender', 'hasManagement', 'hasMaterial', 'hasNationality', 'hasParents', 'hasFather', 'hasMother', 'hasPic', 'hasSpouse', 'manage', 'origins', 'Represented-By', 'Represented-Through', and 'Writes'. The 'hasBelief' property is highlighted. On the right, the 'Annotations: hasBelief' section is empty. Below it, the 'Characteristics: hasBelief' section lists various properties with checkboxes: Functional, Inverse functional, Transitive, Symmetric, Asymmetric, Reflexive, and Irreflexive. The 'Description: hasBelief' section shows 'Equivalent To' as 'owl:topObjectProperty', 'SubProperty Of' as 'owl:topObjectProperty', 'Inverse Of' as 'Person', 'Domains (intersection)' as 'Person', and 'Ranges (intersection)' as 'Belief'.

## hasGender

Sub Property	TopObjectProperty
Domain	Person
Range	Gender
Description	It connects Person and Gender and make sense to triple

## hasGender property

Object property hierarchy: hasGender

owl:topObjectProperty

contains

Draws

has

hasAddress

hasCity

hasPlaceofBirth

hasBelief

hasGender

hasManagement

hasMaterial

hasNationality

hasParents

hasFather

hasMother

hasPic

hasSpouse

manage

origins

Represented-By

Represented-Through

Writes

Asserted

Annotations: hasGender

Annotations

Characteristics:

☐ Functional
☐ Inverse functional
☐ Transitive
☐ Symmetric
☐ Asymmetric
☐ Reflexive
☐ Irreflexive

Description: hasGender

Equivalent To

SubProperty Of

owl:topObjectProperty

Inverse Of

Domains (intersection)

Person

Ranges (intersection)

Gender

## hasManagement

Sub Property	TopObjectProperty
Domain	Museum
Range	Management
Description	It connects Museum and Management and make sense to triple

## hasManagement property





Object property hierarchy: hasMaterial

owl:topObjectProperty

contains

Draws

has

hasAddress

hasCity

hasPlaceofBirth

hasBelief

hasGender

hasManagement

hasMaterial

hasNationality

hasParents

hasFather

hasMother

hasPic

hasSpouse

manage

origins

Represented-By

Represented-Through

Writes

Annotations

+

Annotations

+

Annotations

+

Annotations

+

Annotations

+

Annotations

+

Annotations

+

Annotations

+

Annotations

+

Sub Property	TopObjectProperty
Domain	Person
Range	Country
Description	It connects Person and Country and make sense to triple

**hasNationality**

Object property hierarchy: hasNationality

owl:topObjectProperty

contains

Draws

has

hasAddress

hasCity

hasPlaceofBirth

hasBelief

hasGender

hasManagement

hasMaterial

hasNationality

hasParents

hasFather

hasMother

hasPic

hasSpouse

manage

origins

Represented-By

Represented-Through

Writes

Asserted

Annotations: hasNationality

Annotations +

Characteristics:

Functional

Inverse functional

Transitive

Symmetric

Asymmetric

Reflexive

Irreflexive

Description: hasNationality

Equivalent To +

SubProperty Of +

Inverse Of +

Domains (intersection) +

Ranges (intersection) +

owl:topObjectProperty

Person

Country

Sub Property	TopObjectProperty
Domain	Artist
Range	Parents
Description	It connects Artist and Parents and make sense to triple

### hasParents property



Object property hierarchy: hasFather

owl:topObjectProperty

contains

Draws

has

hasAddress

hasCity

hasPlaceofBirth

hasBelief

hasGender

hasManagement

hasMaterial

hasNationality

hasParents

hasFather

hasMother

hasPic

hasSpouse

manage

origins

Represented-By

Represented-Through

Writes

Annotations

+

Annotations

+

Annotations

+

Annotations

+

Annotations

+

Annotations

+

Annotations

+

Annotations

+

Annotations

+

## hasMother

Sub Property	hasParents
Domain	Artist
Range	Mother
Description	It connects Artist and Mother and make sense to triple

## hasMother property

Object property hierarchy: hasMother

owl:topObjectProperty

contains

Draws

has

hasAddress

hasCity

hasPlaceofBirth

hasBelief

hasGender

hasManagement

hasMaterial

hasNationality

hasParents

hasFather

hasMother

hasPic

hasSpouse

manage

origins

Represented-By

Represented-Through

Writes

Annotations

Annotations

Annotations

Annotations

Annotations

Annotations

Annotations

Annotations

Annotations

Annotations

Annotations

Annotations

Annotations

Annotations

Annotations

Annotations

Annotations

Annotations

hasPic

Sub Property	TopObjectProperty
Domain	Handwritten Documents , Painting
Range	Picture
Description	It connects and make sense to triple

hasPic property

Object property hierarchy: hasPic

owl:topObjectProperty

contains

Draws

has

hasAddress

hasCity

hasPlaceofBirth

hasBelief

hasGender

hasManagement

hasMaterial

hasNationality

hasParents

hasFather

hasMother

hasPic

hasSpouse

manage

origins

Represented-By

Represented-Through

Writes

Asserted

Annotations: hasPic

Annotations

Characteristics:

☐ Functional
☐ Inverse functional
☐ Transitive
☐ Symmetric
☐ Asymmetric
☐ Reflexive
☐ Irreflexive

Description: hasPic

Equivalent To

SubProperty Of

owl:topObjectProperty

Inverse Of

Domains (intersection)

Handwritten\_Documents

Painting

Ranges (intersection)

Picture

hasSpouse

Sub Property	TopObjectProperty
Domain	Artist
Range	Spouse
Description	It connects Artist and Spouse and make sense to triple

hasSpouse property







## Appendix B: Ontology indexing code

```
package apache.apacheJena;

import java.io.File;

import java.io.FileWriter;

import java.io.IOException;

import java.io.InputStream;

import java.util.ArrayList;

import java.util.List;

import java.util.Scanner;


import org.apache.jena.rdf.model.Model;

import org.apache.jena.rdf.model.ModelFactory;

import org.apache.jena.sparql.core.TriplePath;

import org.apache.jena.util.FileManager;

import org.apache.lucene.queryParser.ParseException;


public class Main {

    // File which you want to show in index file any file from Data directory

    static final String INPUTFILENAME = "/home/cis/Desktop/Data/Museum.rdf";

    // path of directory where you want data files to be kept

    static String outputDir = "E:\\ALLAHM\\sharjeel\\updated-17-2-2018\\Data";

    // path of directory where you want index files to be kept1

    static String inputDir = "E:\\ALLAHM\\sharjeel\\updated-17-2-2018\\index";

    static Tester tester = new Tester(inputDir, outputDir);


    public static String prefix = "PREFIX:<http://www.semanticweb.org/Sharjeel/ontologies/#>"

        + "PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>"
```

```

        + "PREFIX owl:<http://www.w3.org/2002/07/owl#>" + "PREFIX
rdfs:<http://www.w3.org/2000/01/rdf-schema#>";

```

```

static boolean optional = false;

```

```

public static void main(String args[]) throws IOException {

```

```

    List<TriplePath> subqueries = new ArrayList<>();

```

```

        String queryString = "PREFIX :
<http://www.semanticweb.org/Sharjeel/ontologies/2017/10/untitled-ontology-6#>"

```

```

        + "PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>"

```

```

        + "PREFIX owl:<http://www.w3.org/2002/07/owl#>" + "PREFIX
rdfs:<http://www.w3.org/2000/01/rdf-schema#>"

```

```

        + "SELECT distinct ?Subject ?Predicate ?Object WHERE { ?Predicate
rdf:type owl:ObjectProperty. ?Predicate rdfs:domain ?Subject. ?Predicate rdfs:range ?Object . }";

```

```

        String mergedQuery = "PREFIX :
<http://www.semanticweb.org/Sharjeel/ontologies/2017/10/untitled-ontology-6#>"

```

```

        + "PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>"

```

```

        + "PREFIX owl:<http://www.w3.org/2002/07/owl#>" + "PREFIX
rdfs:<http://www.w3.org/2000/01/rdf-schema#>"

```

```

        + "SELECT distinct ?Subject ?Subject_subclasses ?Predicate
?Predicate_Domain ?Prdicate_Range ?Predicate_subProperty ?Object ?Object_Subclasses WHERE {
?Predicate rdf:type owl:ObjectProperty. ?Predicate rdfs:domain ?Subject. ?Predicate rdfs:range
?Object ."

```

```

        + "optional {?Subject_subclasses rdfs:subClassOf ?Subject.
?Subject_subclasses a owl:Class.}"

```

```

        + "optional {?Predicate rdfs:domain ?Predicate_Domain.}"

```

```

        + "optional {?Predicate rdfs:range ?Prdicate_Range.}"

```

```

        + "optional {?Predicate rdfs:subPropertyOf
?Predicate_subProperty.}"

```

```

+ "optional {?Object_Subclasses rdfs:subClassOf ?Object.
?Object_Subclasses a owl:Class.}"}";

```

```

// Show all sub-classes and superclasses of any class/entity

```

```

String query1 = "select distinct ?class ?superclass ?subclass WHERE {{?class a
owl:Class }.

```

```

+ "optional {?class rdfs:subClassOf ?superclass.}"+
"optional{?subclass rdfs:subClassOf ?class}}";

```

```

// Show all sub-classes and superclasses of any Woman(specific class)

```

```

String query2 = "select distinct ?superclass ?subclass WHERE { ?superclass a
owl:Class. :Woman rdfs:subClassOf ?superclass."

```

```

+ "?subclass rdfs:subClassOf :Woman}";

```

```

// Show all painters painting where used material is bronz

```

```

String query3 = "select Distinct ?PaintingName ?MaterialName ?firstName
?lastName ?y where {?s a owl:NamedIndividual. {?s :Draws ?x.} ?x :Painting_Name ?PaintingName."

```

```

+ "?x :hasMaterial ?y. ?y :Material_Name 'Bronz'." + "?y
:Material_Name ?MaterialName."

```

```

+ "?s :First_Name ?firstName." + "?s :Last_Name ?lastName." + "}";

```

```

//Show all the artefacts of exhibitions where used craft is oil painting and material is
gold

```

```

String query4 =

```

```

"select Distinct ?painting ?paintingName ?MaterialName
?craftName ?painterFname ?painterLname ?x where{"

```

```

+ "?s a owl:NamedIndividual. ?s :Craft_Name 'Oil Painting'."

```

```

+ "?painting :usedCraft ?s. ?painting :Painting_Name
?paintingName."

```

```

+ "?s :Craft_Name ?craftName." + "?painting :hasMaterial
?material."

```

```

+ "?material:Material_Name 'Gold'." + "?material:Material_Name
?MaterialName."

```

```

+ "?x:Draws ?painting." + "?x:Last_Name ?painterFname." + "?x
:First_Name ?painterLname.}";

```

//Show all addresses including cities of Museums where region is Europe and Museum category is archaeology

```

String query5 = "select Distinct ?RegionName ?addressOfMuseum
?countryOfMuseum ?categoryOfMuseum ?cityOfMuseum where{ "
+ "?s a owl:NamedIndividual. ?s :Region_Name 'Europe'." + "?s
:Region_Name ?RegionName."
+ "?category:Category_Name 'Archaeology'." + "?z :hasAddress
?address." + "?z :hasCity ?city."
+ "?z :hasCountry ?country." + "?z :hasCategory ?category."
+ "?address :Museum_Address ?addressOfMuseum." + "?city
:City_Name ?cityOfMuseum."
+ "?country:Country_Name ?countryOfMuseum." + "?category
:Category_Name ?categoryOfMuseum."
+ "}";

```

//Show addresses of all museums in Europe region which holds the artefacts of Asian's artist whoused oil painting craft for paintings

```

String query6="select ?CraftName ?PaintingName ?Artist_firstName
?Artist_LastName ?ArtistRegion ?Museum ?MuseumRegion ?cityOfMuseum where"
+ "{?s :Craft_Name 'Oil Painting'. ?x:usedCraft ?s."
+ "?s :Craft_Name ?CraftName."
+ "?Y:Draws ?x."
+ "?Y:First_Name ?Artist_firstName."
+ "?Y:Last_Name ?Artist_LastName."
+ "?Y:hasRegion ?ArtistReg."
+ "?ArtistReg:Region_Name ?ArtistRegion."

```

```

+"?x:Painting_Name ?PaintingName."
+"?Museum:hasArtifact ?x. ?Museum:hasRegion ?r."
+"?Museum:hasAddress ?adress."
+"?Museum:hasCity ?city."
+"?city:City_Name ?cityOfMuseum."
+"?r:Region_Name ?MuseumRegion."
+"?r:Region_Name 'Europe'.  }";

```

```

ReadableIndex readableIndex = new ReadableIndex(outputDir, inputDir);
Model model = readableIndex.createReadableIndex(new TextFileFilter());

```

```

// Get List of Models in output directory.
List<Model> modelCollection = new ArrayList<>();
modelCollection = readableIndex.createModelList(new TextFileFilter());
System.out.println("Total Models in Directory: " + modelCollection.size());

```

```

try{
    tester.search("Painting");
} catch (ParseException e){
    // TODO Auto-generated catch block
    e.printStackTrace();
}

```

```

File file = new File(outputDir + "/" + "queries.txt");
if (!file.exists()){
    file.createNewFile();
}

```

```

    }

    FileWriter writer = new FileWriter(file);

    AlgebraExec transformer = new AlgebraExec();

    String algebraForm = null;

    // options for selecting query

    String queryFinal = null;

    int queryIndex;

    //do {

        readableIndex.queryExecutor(model, mergedQuery,
"IndexFile_Merged.csv");

        System.out.println("-----");

        System.out.println("1. Show all sub-classes and superclasses of any
entity/class");

        /*System.out.println("2. Show all sub-classes and superclasses of specific
entity/class - woman");*/

        System.out.println(

            "3. Show all painters painting where used material is bronz
including painting's image(3)");

        System.out.println(

            "4. Show all the artefacts of exhibitions where used craft is
oil painting and material is gold(8)");

        System.out.println(

            "5. Show all addresses including cities of Museums where
region is Europe and Museum category is archaeology(10)");

        System.out.println(

            "6. Show addresses of all museums in Europe region which
holds the artefacts of Asian's artist whoused oil painting craft for paintings(11)");

        System.out.println("7. Press 7 to exit");
    }

```

```

System.out.println();

System.out.println("Enter query(number) to be executed - ");

Scanner input = new Scanner(System.in);

queryIndex = input.nextInt();

switch (queryIndex) {

case 1:

        readableIndex.queryExecutor(query1, "IndexFile_query1.csv", new
TextFileFilter());

        algebraForm = transformer.transformToAlgebraicForm(query1,
model);

        queryFinal = query1;

        break;

case 2:

        readableIndex.queryExecutor(query2, "IndexFile_query2.csv", new
TextFileFilter());

        algebraForm = transformer.transformToAlgebraicForm(query2,
model);

        queryFinal = query2;

        break;

case 3:

        readableIndex.queryExecutor(query3, "IndexFile_query3.csv", new
TextFileFilter());

        algebraForm = transformer.transformToAlgebraicForm(query3,
model);

```



```

        queryFinal = query3;

        break;

    case 4:

        readableIndex.queryExecutor(query4, "IndexFile_query4.csv", new
TextFileFilter());

        algebraForm = transformer.transformToAlgebraicForm(query4,
model);

        queryFinal = query4;

        break;

    case 5:

        readableIndex.queryExecutor(query5, "IndexFile_query5.csv", new
TextFileFilter());

        algebraForm = transformer.transformToAlgebraicForm(query5,
model);

        queryFinal = query5;

        break;

    case 6:

        readableIndex.queryExecutor(query6, "IndexFile_query6.csv", new
TextFileFilter());

        algebraForm = transformer.transformToAlgebraicForm(query6,
model);

        queryFinal = query6;

        break;

    case 7:

        System.exit(0);

```

```

        default:
            /*readableIndex.queryExecutor(modelCollection, mergedQuery,
"IndexFile_Merged.csv");

            algebraForm =
transformer.transformToAlgebraicForm(mergedQuery, model);

            queryFinal = mergedQuery;*/

            break;
        }

    if (algebraForm != null && algebraForm.contains("project")) {
        algebraForm.replaceAll("project", "π");
    }

    // Writes the sparql query and its algebraic form to the file.
    writer.write("\n" + queryFinal + "\n\n SPARQL Algebra :\n " + algebraForm);

    subqueries = transformer.getSubqueries(queryFinal);

    // get models for triples
    SubQueryGenerator generator = new SubQueryGenerator(subqueries);
    generator.getModelsForQuery(modelCollection);
    generator.runQueryonModels(modelCollection, queryFinal);

    writer.flush();
    writer.close();

```

```
        System.out.println("\nConsolidated and subquery results are present under
directory : " + outputDir);

        System.out.println();

//        } while (queryIndex != 7);

        System.exit(0);

    }

}
```

## Appendix C - Query conversion code

```
package apache.apacheJena;

import java.io.IOException;

import java.util.ArrayList;

import java.util.HashMap;

import java.util.Iterator;

import java.util.List;

import java.util.Map;


import org.apache.jena.graph.Triple;

import org.apache.jena.query.Query;

import org.apache.jena.query.QueryFactory;

import org.apache.jena.rdf.model.Model;

import org.apache.jena.sparql.algebra.Algebra;

import org.apache.jena.sparql.algebra.Op;

import org.apache.jena.sparql.algebra.Transform;

import org.apache.jena.sparql.algebra.op.Op2;

import org.apache.jena.sparql.algebra.op.OpConditional;

import org.apache.jena.sparql.algebra.op.OpJoin;

import org.apache.jena.sparql.algebra.op.OpLeftJoin;

import org.apache.jena.sparql.algebra.op.OpQuadPattern;

import org.apache.jena.sparql.algebra.optimize.Optimize;

import org.apache.jena.sparql.algebra.optimize.TransformFilterPlacement;

import org.apache.jena.sparql.core.TriplePath;

import org.apache.jena.sparql.core.Var;
```

```

import org.apache.jena.sparql.graph.NodeTransform;

import org.apache.jena.sparql.graph.NodeTransformLib;

import org.apache.jena.sparql.sse.SSE;

import org.apache.jena.sparql.syntax.Element;

import org.apache.jena.sparql.syntax.ElementPathBlock;

import org.apache.jena.sparql.syntax.ElementVisitorBase;

import org.apache.jena.sparql.syntax.ElementWalker;

import org.apache.jena.sparql.syntax.syntaxtransform.NodeTransformSubst;


public class AlgebraExec {

    static String outputDir = "E:\\ALLAHM\\sharjeel\\updated-17-2-2018\\Data";

    // path of directory where you want index files to be kept

    static String inputDir = "E:\\ALLAHM\\sharjeel\\updated-17-2-2018\\index";

    static Tester tester = new Tester(inputDir, outputDir);

    static List<TriplePath> subqueries= new ArrayList<>();


    public String transformToAlgebraicForm(String queryString, Model model) throws
IOException {

        Query query = QueryFactory.create(Main.prefix+queryString);

        Element e = query.getQueryPattern();

        Op op = Algebra.compile(query);

        Transform transform = new TransformFilterPlacement();

        op = Optimize.apply(transform, op);

        // op = Optimize.optimize(op, new Context());

        Map<Var, Var> varMap= new HashMap<Var, Var>();

        varMap.put(Var.alloc("s"), Var.alloc("x"));

        NodeTransform nodeTrans = new NodeTransformSubst(varMap);

```

```

        op = NodeTransformLib.transform(nodeTrans, op);

        System.out.println("-----");

        System.out.println(op);

        return op.toString();

    }

    public List<TriplePath> getSubqueries (String queryString){

        Query query = QueryFactory.create(Main.prefix+queryString);

        Element e = query.getQueryPattern();

        System.out.println("-----");

        System.out.println("triple(s) from compiled query as below - ");

        ElementVisitorBase elementVisitor = new ElementVisitorBase() {

            int i = 1;

            @Override

            public void visit(ElementPathBlock el) {

                Iterator<TriplePath> iterator = el.getPattern().iterator();

                while (iterator.hasNext()) {

                    TriplePath triplePath = iterator.next();

                    System.out.println((i) + ": " + triplePath);

                    //TriplePaths to list

```

```

        subqueries.add(triplePath);

        i++;
    }

    super.visit(el);
}

};

ElementWalker.walk(e, elementVisitor);

System.out.println("-----");

return subqueries;

}

// indexing and searching datasources
// https://jena.apache.org/documentation/larq/

public static void main(String[] args) throws IOException {
    /*
    * String queryString =
    * "PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>"
    * +"PREFIX owl: <http://www.w3.org/2002/07/owl#>"
    * +"PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>"
    * +"PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>"
    * +"PREFIX clb: <https://www.caliberresearch.org/PhenotypeOntology#>"
    * +"    SELECT ?subject WHERE { ?subject rdf:type
    clb:subject_with_diabdiag_gprd_3_code ."

```

```

* +" FILTER NOT EXISTS {?subject rdf:type
clb:subject_with_type_unknown_diabetes.}}"
* ;
*
* (project (?subject) (filter (notexists (bgp (triple ?subject
* <http://www.w3.org/1999/02/22-rdf-syntax-ns#type>
* <https://www.caliberresearch.org/PhenotypeOntology#
* subject_with_type_unknown_diabetes>))) (bgp (triple ?subject
* <http://www.w3.org/1999/02/22-rdf-syntax-ns#type>
* <https://www.caliberresearch.org/PhenotypeOntology#
* subject_with_diabdiag_gprd_3_code>))))))
*
*
*
* SELECT ?patient ?phoneType ?phoneNumber WHERE { ?phoneType
* rdfs:subPropertyOf example:phone . ?patient ?phoneType ?phoneNumber .
* }
*
*
* AlgebraExec queryTransformer = new AlgebraExec(); String
* transformedQuery =
* queryTransformer.transformToAlgebraicForm(queryString);
*
* System.out.println("----- : " + transformedQuery);
* /}
}

```



## Appendix D – Query integration code

```
package apache.apacheJena;

import java.io.IOException;

import org.apache.lucene.document.Document;

import org.apache.lucene.queryParser.ParseException;

import org.apache.lucene.search.ScoreDoc;

import org.apache.lucene.search.TopDocs;


public class Tester {

    //path of directory where you want index files to be kept
    String indexDir = "E:\\ALLAHM\\sharjeel\\updated-17-2-2018\\index";

    //path of directory where rdf files are kept
    String dataDir = "E:\\ALLAHM\\sharjeel\\updated-17-2-2018\\Data";

    Indexer indexer;

    Searcher searcher;


    public Tester(String indexDir,String dataDir) {

        this.indexDir = indexDir;

        this.dataDir = dataDir;

        createIndex();

    }


    //    public static void main(String[] args) {
    //
    //        Tester tester;
    //
    //        try {
```

```

//            tester = new Tester();
//            tester.createIndex();
//            // change keyword here which you want to search.
//            tester.search("Painting");
//        } catch (IOException e) {
//            e.printStackTrace();
//        } catch (ParseException e) {
//            e.printStackTrace();
//        }
//    }

```

```

private void createIndex() {
    try {
        indexer = new Indexer(indexDir);

        int numIndexed;

        long startTime = System.currentTimeMillis();

        numIndexed = indexer.createIndex(dataDir, new TextFileFilter());

        long endTime = System.currentTimeMillis();

        indexer.close();

        System.out.println(numIndexed + " File indexed, time taken: " + (endTime -
startTime) + " ms");
    } catch (IOException e) {
        e.printStackTrace();
    }
}

```

```

public void search(String searchQuery) throws IOException, ParseException {
    searcher = new Searcher(indexDir);
}

```

```
        long startTime = System.currentTimeMillis();

        TopDocs hits = searcher.search(searchQuery);

        long endTime = System.currentTimeMillis();

        System.out.println(hits.totalHits + " documents found. Time : " + (endTime -
startTime));

        for (ScoreDoc scoreDoc : hits.scoreDocs) {

            Document doc = searcher.getDocument(scoreDoc);

            System.out.println("File: " + doc.get(Constants.FILE_PATH));

        }

        searcher.close();

    }

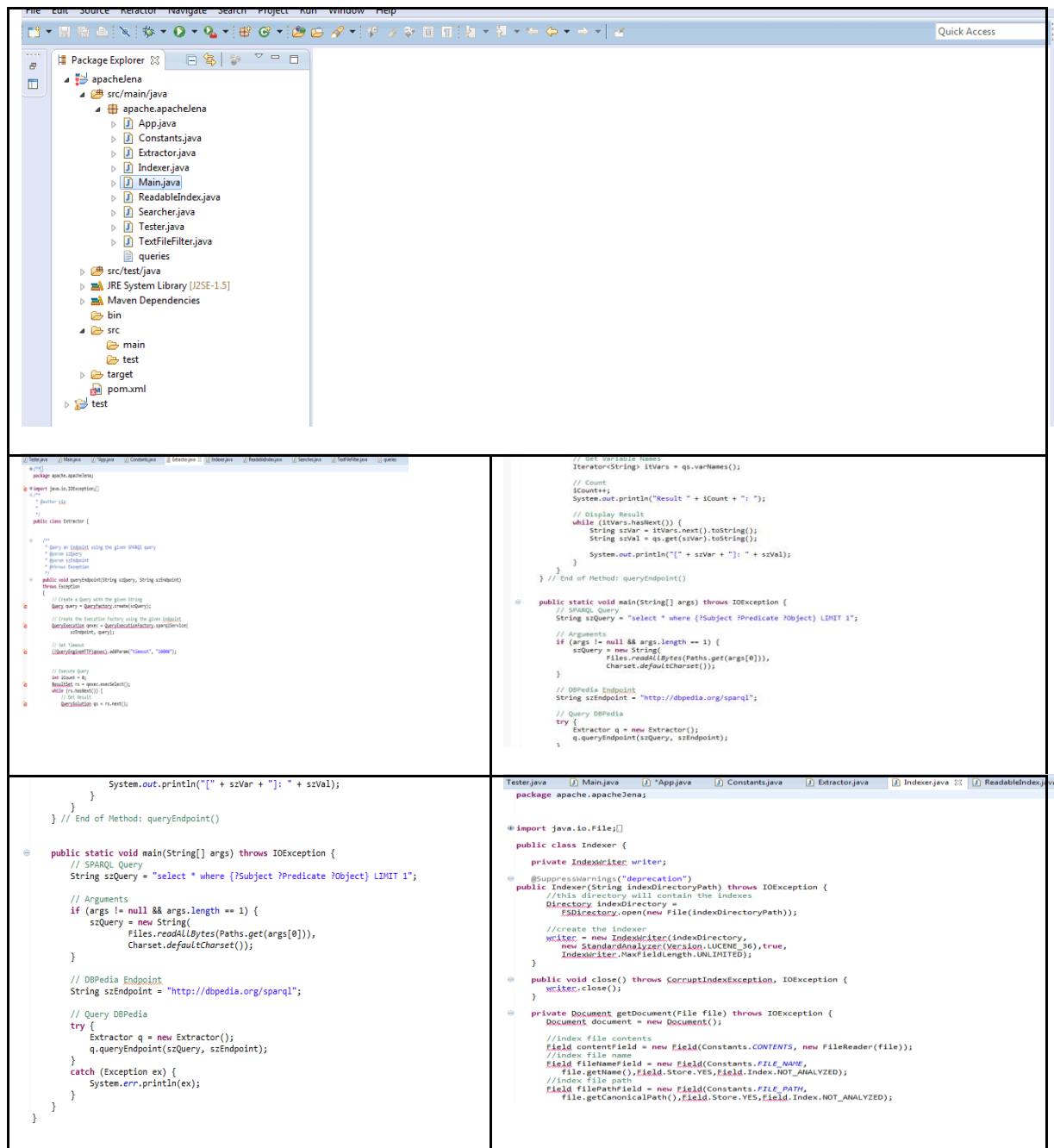
}
```

## Appendix E – Setup and Testing Screenshots

### Setup:

- 1) Jdk 1.8 must be installed. Maven must be installed. Eclipse with Maven plugin must be installed.
- 2) Open eclipse and go to **File** option from there choose **import**.
- 3) In import choose **Maven** in that select **Existing maven projects into Workspace**.
- 4) After that from browse select **RDF-Jena** folder. Where you have extracted it and click on **Finish**.
- 5) Once it is completed right click on project go to **Maven** option and click on **mvn install**.  
After you see build successful in console proceed further.
- 6) Open project and go to **apache.apacheJena** package.
- 7) Open **Main.java** file.
- 8) Make changes on Line number 26, 27 and 28 for location where you want to index and location where rdf files are present respectively. (**Files must have .rdf extention**)
- 9) Then go to line number and put keyword which you want to search.
- 10) After changes save file then right click in code screen go to **Run as..** and select **Java application**.
- 11) In console you will see list of files which contain that keyword.
- 12) Also you will see new file with name **IndexFile.txt** which you want to show in Directory where other RDF files are present.

### Code screenshots :



<pre> Field fileNameField = new Field(Constants.FILE_NAME,     file.getName(),Field.Store.YES,Field.Index.NOT_ANALYZED); //index file path Field filePathField = new Field(Constants.FILE_PATH,     file.getCanonicalPath(),Field.Store.YES,Field.Index.NOT_ANALYZED);  document.add(contentField); document.add(fileNameField); document.add(filePathField);  return document; }  private void indexFile(File file) throws IOException {     System.out.println("Indexing " +file.getCanonicalPath());     Document document = getDocument(file);     writer.addDocument(document); }  public int createIndex(String dataDirPath, FileFilter filter)     throws IOException {     //get all files in the data directory     File[] files = new File(dataDirPath).listFiles();      for(File file : files) {         if(!file.isDirectory())             &amp;&amp; !file.isHidden()             &amp;&amp; file.exists()             &amp;&amp; file.canRead()             &amp;&amp; filter.accept(file)         ){             indexFile(file);         }     }     return writer.numDocs(); } </pre>	<pre> Tester.java  Main.java  App.java  Constants.java  Extractor.java  Indexer.java  ReadableIndex.java  package apache.apacheJena;  import java.io.InputStream;  import org.apache.jena.query.Query; import org.apache.jena.query.QueryExecution; import org.apache.jena.query.QueryExecutionFactory; import org.apache.jena.query.QueryFactory; import org.apache.jena.query.ResultSet; import org.apache.jena.query.ResultSetFormatter;  /*  * To change this license header, choose License Headers in Project Properties.  * To change this template file, choose Tools   Templates  * and open the template in the editor.  */  import org.apache.jena.rdf.model.Model; import org.apache.jena.rdf.model.ModelFactory; import org.apache.jena.rdf.model.Property; import org.apache.jena.rdf.model.RDFNode; import org.apache.jena.rdf.model.Resource; import org.apache.jena.rdf.model.Statement; import org.apache.jena.rdf.model.StmtIterator; import org.apache.jena.util.FileManager;  public class Main {     static final String INPUTFILENAME = "/home/cis/Desktop/Data/Museum.rdf";      public static void main(String args[]) {         // create an empty model         Model model = ModelFactory.createDefaultModel();          InputStream in = FileManager.get().open(INPUTFILENAME);         if (in == null) {             throw new IllegalArgumentException("File: " + INPUTFILENAME + " not found");         }          // read the RDF/XML file         model.read(in, "");     } } </pre>
<pre> // Read the RDF/XML file model.read(in, "");  // write it to standard out //model.write(System.out);  Model read = ModelFactory.createDefaultModel().read(INPUTFILENAME);  String queryString = "PREFIX : &lt;http://www.semanticweb.org/Sharjeel/ontologies/2017/10/untitled-ontology-6/&gt; +PREFIX owl:&lt;http://www.w3.org/2002/07/owl#&gt; +PREFIX rdfs:&lt;http://www.w3.org/2000/01/rdf-schema#&gt; +\"SELECT ?x WHERE { rdfs:type &lt;http://www.semanticweb.org/Sharjeel/ontologies/2017/10/untitled-ontology-6#Parents&gt; ?x}";  String queryString = "SELECT DISTINCT ?s ?p ?o" + " WHERE { ?s ?p ?o" + " }";  Query query = QueryFactory.create(queryString);  // Execute the query and obtain results QueryExecution qe = QueryExecutionFactory.create(query, model); ResultSet results = qe.execSelect();  // Output query results ResultSetFormatter.out(System.out, results, query);  // Important - free up resources used running the query qe.close(); StmtIterator si; si = read.listStatements(); while(si.hasNext()) {     Statement s = si.nextStatement();     Resource r = s.getResource();     Property p = s.getProperty();     RDFNode o = s.getObject(); } } }  package apache.apacheJena; import java.io.File;  public class Main {     //File which you want to show in index file any file from Data directory     static final String INPUTFILENAME = "/home/cis/Desktop/Data/Museum.rdf";      //path of directory where you want data files to be kept     static String outputDir = "/home/cis/Desktop/Data";     //path of directory where you want index files to be kept     static String inputDir = "/home/cis/Desktop/index";     static Tester tester = new Tester(inputDir, outputDir); } </pre>	<pre> // Important - free up resources used running the query qe.close(); StmtIterator si; si = read.listStatements(); while(si.hasNext()) {     Statement s = si.nextStatement();     Resource r = s.getResource();     Property p = s.getProperty();     RDFNode o = s.getObject();     System.out.println("subject - " + r.getURI());     System.out.println(p.getURI());     System.out.println(o.asResource().getURI()); } } }  package apache.apacheJena; import java.io.File;  public class Main {     //File which you want to show in index file any file from Data directory     static final String INPUTFILENAME = "/home/cis/Desktop/Data/Museum.rdf";      //path of directory where you want data files to be kept     static String outputDir = "/home/cis/Desktop/Data";     //path of directory where you want index files to be kept     static String inputDir = "/home/cis/Desktop/index";     static Tester tester = new Tester(inputDir, outputDir); } </pre>
<pre> String queryString = "PREFIX : &lt;http://www.semanticweb.org/Sharjeel/ontologies/2017/10/untitled-ontology-6/&gt; +PREFIX rdfs: &lt;http://www.w3.org/2000/01/rdf-schema#&gt; +PREFIX owl: &lt;http://www.w3.org/2002/07/owl#&gt; +PREFIX rdf: &lt;http://www.w3.org/2000/01/rdf-schema#&gt; +\"SELECT distinct ?subject ?predicate ?object WHERE { ?predicate rdfs:type owl:ObjectProperty, ?predicate rdfs:domain ?subject, ?predicate rdfs:range ?object}";  //Classes and subclasses query working /*PREFIX : &lt;http://www.semanticweb.org/Sharjeel/ontologies/2017/10/untitled-ontology-6/&gt; PREFIX rdfs: &lt;http://www.w3.org/2000/01/rdf-schema#&gt; PREFIX owl: &lt;http://www.w3.org/2002/07/owl#&gt; PREFIX rdf: &lt;http://www.w3.org/2000/01/rdf-schema#&gt; SELECT ?class ?subClass WHERE { ?class rdfs:subClassOf ?subClass. ?subClass a owl:Class. } */  // Clipped subject and object and find their subclasses if any String querySubClass = "PREFIX : &lt;http://www.semanticweb.org/Sharjeel/ontologies/2017/10/untitled-ontology-6/&gt; +PREFIX rdfs: &lt;http://www.w3.org/2000/01/rdf-schema#&gt; +PREFIX owl: &lt;http://www.w3.org/2002/07/owl#&gt; +PREFIX rdf: &lt;http://www.w3.org/2000/01/rdf-schema#&gt; +\"SELECT distinct ?Subject ?superClass WHERE { ?predicate rdfs:type owl:ObjectProperty, [?predicate rdfs:domain ?Subject] UNION [?predicate rdfs:range ?Object] }";  ReadableIndex readableIndex = new ReadableIndex(outputDir, inputDir); Model model = readableIndex.createReadableIndex(new TestFileFilter()); tester.search("Painting");  //Executing query for retrieving subjects, objects and predicates readableIndex.queryExecutor(model, queryString, "IndexFile_sub_obj_predicates.csv");  //Executing query for retrieving subjects/objects and their subclasses readableIndex.queryExecutor(model, querySubClass, "IndexFile_sub_subcls.csv"); } } </pre>	<pre> Tester.java  Main.java  App.java  Constants.java  Extractor.java  Indexer.java  ReadableIndex.java  Searcher.java  package apache.apacheJena;  import java.io.File;  public class ReadableIndex {     // path of directory where you want data files to be kept     static String outputDir = "";     // path of directory where you want index files to be kept     static String inputDir = "";      public ReadableIndex(String outputDir, String inputDir) {         this.outputDir = outputDir;         this.inputDir = inputDir;     }      //     * @param FileFilter     * @return : Consolidated model     //      public Model createReadableIndex(FileFilter filter) throws IOException {         Model superModel = ModelFactory.createDefaultModel();         String fileExtension = null;         // get all files in the data directory         File[] files = new File(outputDir).listFiles();         for (File file : files) {             Model model = ModelFactory.createDefaultModel();             if (!file.isDirectory() &amp;&amp; !file.isHidden() &amp;&amp; file.canRead() &amp;&amp; filter.accept(file)) {                 InputStream in = FileManager.get().open(file.getAbsolutePath());                 System.out.println(file.getAbsolutePath());                  // read the RDF/XML file                 model.read(in, "");                 superModel = ModelFactory.createUnion(model, superModel);             }         }     } } </pre>
<pre> // Read the RDF/XML file model.read(in, ""); superModel = ModelFactory.createUnion(model, superModel); } } return superModel; }  /*  * @param Model : consolidated model on which query to be executed.  * @param queryString : Query to be executed.  * @param outputFile : Name of file which should be generated as output containing query result.  */ public void queryExecutor(Model superModel, String queryString, String outputFile) throws IOException {     Query query = QueryFactory.create(queryString);      // Execute the query and obtain results     QueryExecution qe = QueryExecutionFactory.create(query, superModel);     ResultSet results = qe.execSelect();      FileOutputStream fop = null;     File file = new File(outputDir + "/" + outputFile);     fop = new FileOutputStream(file);      // Output query results     ResultSetFormatter.out(fop, results);      // Important - free up resources used running the query     qe.close(); } } </pre>	<pre> Tester.java  Main.java  App.java  Constants.java  Extractor.java  Indexer.java  ReadableIndex.java  package apache.apacheJena;  import java.io.IOException;  public class Tester {     //path of directory where you want index files to be kept     String indexDir = "/home/cis/Desktop/index";     //path of directory where rdf files will be kept     String dataDir = "/home/cis/Desktop/Data";     Indexer indexer;     Searcher searcher;      public Tester(String indexDir, String dataDir) {         this.indexDir = indexDir;         this.dataDir = dataDir;         createIndex();     }      // public static void main(String[] args) {     //     Tester tester;     //     try {     //         tester = new Tester();     //         tester.createIndex();     //         // change keyword here which you want to search.     //         tester.search("Painting");     //     } catch (IOException e) {     //         e.printStackTrace();     //     } catch (ParseException e) {     //         e.printStackTrace();     //     }     // }     // } } </pre>

```
//
// change keyword here which you want to search.
//
// tester.search("Painting");
//
// } catch (IOException e) {
//     e.printStackTrace();
// }
// } catch (ParseException e) {
//     e.printStackTrace();
// }
// }

private void createIndex() {
    try {
        Indexer = new Indexer(indexDir);
        int numIndexed;
        long startTime = System.currentTimeMillis();
        numIndexed = Indexer.createIndex(dataDir, new TextFileFilter());
        long endTime = System.currentTimeMillis();
        Indexer.close();
        System.out.println(numIndexed + " File indexed, time taken: " + (endTime - startTime) + " ms");
    } catch (IOException e) {
        e.printStackTrace();
    }
}

public void search(String searchQuery) throws IOException, ParseException {
    searcher = new Searcher(indexDir);
    long startTime = System.currentTimeMillis();
    TopDocs hits = searcher.search(searchQuery);
    long endTime = System.currentTimeMillis();

    System.out.println(hits.totalHits + " documents found. Time: " + (endTime - startTime));
    for (ScoreDoc scoreDoc : hits.scoreDocs) {
        Document doc = searcher.getDocument(scoreDoc);
        System.out.println("File: " + doc.get(Constants.FILE_PATH));
    }
    searcher.close();
}
}
```

## Museum RDF format:

<pre>1  &lt;?xml version="1.0"?&gt; 2  &lt;? rdf:RDF xmlns="http://www.semanticweb.org/Sharjeel/ontologies/2017/10/untitled-ontology-64" 3  &gt; 4  &lt;owl:Class rdf:type="http://www.w3.org/1999/02/22-rdf-syntax-ns#" 5  &gt; 6  &lt;owl:Class rdf:type="http://www.w3.org/2002/07/owl#" 7  &gt; 8  &lt;owl:Class rdf:type="http://www.w3.org/2000/01/rdf-schema#" 9  &gt; 10 &lt;owl:ontology rdf:about="http://www.semanticweb.org/Sharjeel/ontologies/2017/10/untitled-ontology-64"&gt; 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 51 52 53 54 55 56 57 58 59 60 61 62 63 64 65 66 67 68 69 70 71 72 73 74 75 76 77 78 79 80 81 82 83 84 85 86 87 88 89 90 91 92 93 94 95 96 97 98 99 100 101 102 103 104 105 106 107 108 109 110 111 112 113 114 115 116 117 118 119 120 121 122 123 124 125 126 127 128 129 130 131 132 133 134 135 136 137 138 139 140 141 142 143 144 145 146 147 148 149 150 151 152 153 154 155 156 157 158 159 160 161 162 163 164 165 166 167 168 169 170 171 172 173 174 175 176 177 178 179 180 181 182 183 184 185 186 187 188 189 190 191 192 193 194 195 196 197 198 199 200 201 202 203 204 205 206 207 208 209 210 211 212 213 214 215 216 217 218 219 220 221 222 223 224 225 226 227 228 229 230 231 232 233 234 235 236 237 238 239 240 241 242 243 244 245 246 247 248 249 250 251 252 253 254 255 256 257 258 259 260 261 262 263 264 265 266 267 268 269 270 271 272 273 274 275 276 277 278 279 280 281 282 283 284 285 286 287 288 289 290 291 292 293 294 295 296 297 298 299 300 301 302 303 304 305 306 307 308 309 310 311 312 313 314 315 316 317 318 319 320 321 322 323 324 325 326 327 328 329 330 331 332 333 334 335 336 337 338 339 340 341 342 343 344 345 346 347 348 349 350 351 352 353 354 355 356 357 358 359 360 361 362 363 364 365 366 367 368 369 370 371 372 373 374 375 376 377 378 379 380 381 382 383 384 385 386 387 388 389 390 391 392 393 394 395 396 397 398 399 400 401 402 403 404 405 406 407 408 409 410 411 412 413 414 415 416 417 418 419 420 421 422 423 424 425 426 427 428 429 430 431 432 433 434 435 436 437 438 439 440 441 442 443 444 445 446 447 448 449 450 451 452 453 454 455 456 457 458 459 460 461 462 463 464 465 466 467 468 469 470 471 472 473 474 475 476 477 478 479 480 481 482 483 484 485 486 487 488 489 490 491 492 493 494 495 496 497 498 499 500 501 502 503 504 505 506 507 508 509 510 511 512 513 514 515 516 517 518 519 520 521 522 523 524 525 526 527 528 529 530 531 532 533 534 535 536 537 538 539 540 541 542 543 544 545 546 547 548 549 550 551 552 553 554 555 556 557 558 559 560 561 562 563 564 565 566 567 568 569 570 571 572 573 574 575 576 577 578 579 580 581 582 583 584 585 586 587 588 589 590 591 592 593 594 595 596 597 598 599 600 601 602 603 604 605 606 607 608 609 610 611 612 613 614 615 616 617 618 619 620 621 622 623 624 625 626 627 628 629 630 631 632 633 634 635 636 637 638 639 640 641 642 643 644 645 646 647 648 649 650 651 652 653 654 655 656 657 658 659 660 661 662 663 664 665 666 667 668 669 670 671 672 673 674 675 676 677 678 679 680 681 682 683 684 685 686 687 688 689 690 691 692 693 694 695 696 697 698 699 700 701 702 703 704 705 706 707 708 709 710 711 712 713 714 715 716 717 718 719 720 721 722 723 724 725 726 727 728 729 730 731 732 733 734 735 736 737 738 739 740 741 742 743 744 745 746 747 748 749 750 751 752 753 754 755 756 757 758 759 760 761 762 763 764 765 766 767 768 769 770 771 772 773 774 775 776 777 778 779 780 781 782 783 784 785 786 787 788 789 790 791 792 793 794 795 796 797 798 799 800 801 802 803 804 805 806 807 808 809 810 811 812 813 814 815 816 817 818 819 820 821 822 823 824 825 826 827 828 829 830 831 832 833 834 835 836 837 838 839 840 841 842 843 844 845 846 847 848 849 850 851 852 853 854 855 856 857 858 859 860 861 862 863 864 865 866 867 868 869 870 871 872 873 874 875 876 877 878 879 880 881 882 883 884 885 886 887 888 889 890 891 892 893 894 895 896 897 898 899 900 901 902 903 904 905 906 907 908 909 910 911 912 913 914 915 916 917 918 919 920 921 922 923 924 925 926 927 928 929 930 931 932 933 934 935 936 937 938 939 940 941 942 943 944 945 946 947 948 949 950 951 952 953 954 955 956 957 958 959 960 961 962 963 964 965 966 967 968 969 970 971 972 973 974 975 976 977 978 979 980 981 982 983 984 985 986 987 988 989 990 991 992 993 994 995 996 997 998 999 1000</pre>	<pre>&lt;owl:ObjectProperty&gt; 51 52 53 54 55 56 57 58 59 60 61 62 63 64 65 66 67 68 69 70 71 72 73 74 75 76 77 78 79 80 81 82 83 84 85 86 87 88 89 90 91 92 93 94 95 96 97 98 99 100 101 102 103 104 105 106 107 108 109 110 111 112 113 114 115 116 117 118 119 120 121 122 123 124 125 126 127 128 129 130 131 132 133 134 135 136 137 138 139 140 141 142 143 144 145 146 147 148 149 150 151 152 153 154 155 156 157 158 159 160 161 162 163 164 165 166 167 168 169 170 171 172 173 174 175 176 177 178 179 180 181 182 183 184 185 186 187 188 189 190 191 192 193 194 195 196 197 198 199 200 201 202 203 204 205 206 207 208 209 210 211 212 213 214 215 216 217 218 219 220 221 222 223 224 225 226 227 228 229 230 231 232 233 234 235 236 237 238 239 240 241 242 243 244 245 246 247 248 249 250 251 252 253 254 255 256 257 258 259 260 261 262 263 264 265 266 267 268 269 270 271 272 273 274 275 276 277 278 279 280 281 282 283 284 285 286 287 288 289 290 291 292 293 294 295 296 297 298 299 300 301 302 303 304 305 306 307 308 309 310 311 312 313 314 315 316 317 318 319 320 321 322 323 324 325 326 327 328 329 330 331 332 333 334 335 336 337 338 339 340 341 342 343 344 345 346 347 348 349 350 351 352 353 354 355 356 357 358 359 360 361 362 363 364 365 366 367 368 369 370 371 372 373 374 375 376 377 378 379 380 381 382 383 384 385 386 387 388 389 390 391 392 393 394 395 396 397 398 399 400 401 402 403 404 405 406 407 408 409 410 411 412 413 414 415 416 417 418 419 420 421 422 423 424 425 426 427 428 429 430 431 432 433 434 435 436 437 438 439 440 441 442 443 444 445 446 447 448 449 450 451 452 453 454 455 456 457 458 459 460 461 462 463 464 465 466 467 468 469 470 471 472 473 474 475 476 477 478 479 480 481 482 483 484 485 486 487 488 489 490 491 492 493 494 495 496 497 498 499 500 501 502 503 504 505 506 507 508 509 510 511 512 513 514 515 516 517 518 519 520 521 522 523 524 525 526 527 528 529 530 531 532 533 534 535 536 537 538 539 540 541 542 543 544 545 546 547 548 549 550 551 552 553 554 555 556 557 558 559 560 561 562 563 564 565 566 567 568 569 570 571 572 573 574 575 576 577 578 579 580 581 582 583 584 585 586 587 588 589 590 591 592 593 594 595 596 597 598 599 600 601 602 603 604 605 606 607 608 609 610 611 612 613 614 615 616 617 618 619 620 621 622 623 624 625 626 627 628 629 630 631 632 633 634 635 636 637 638 639 640 641 642 643 644 645 646 647 648 649 650 651 652 653 654 655 656 657 658 659 660 661 662 663 664 665 666 667 668 669 670 671 672 673 674 675 676 677 678 679 680 681 682 683 684 685 686 687 688 689 690 691 692 693 694 695 696 697 698 699 700 701 702 703 704 705 706 707 708 709 710 711 712 713 714 715 716 717 718 719 720 721 722 723 724 725 726 727 728 729 730 731 732 733 734 735 736 737 738 739 740 741 742 743 744 745 746 747 748 749 750 751 752 753 754 755 756 757 758 759 760 761 762 763 764 765 766 767 768 769 770 771 772 773 774 775 776 777 778 779 780 781 782 783 784 785 786 787 788 789 790 791 792 793 794 795 796 797 798 799 800 801 802 803 804 805 806 807 808 809 810 811 812 813 814 815 816 817 818 819 820 821 822 823 824 825 826 827 828 829 830 831 832 833 834 835 836 837 838 839 840 841 842 843 844 845 846 847 848 849 850 851 852 853 854 855 856 857 858 859 860 861 862 863 864 865 866 867 868 869 870 871 872 873 874 875 876 877 878 879 880 881 882 883 884 885 886 887 888 889 890 891 892 893 894 895 896 897 898 899 900 901 902 903 904 905 906 907 908 909 910 911 912 913 914 915 916 917 918 919 920 921 922 923 924 925 926 927 928 929 930 931 932 933 934 935 936 937 938 939 940 941 942 943 944 945 946 947 948 949 950 951 952 953 954 955 956 957 958 959 960 961 962 963 964 965 966 967 968 969 970 971 972 973 974 975 976 977 978 979 980 981 982 983 984 985 986 987 988 989 990 991 992 993 994 995 996 997 998 999 1000</pre>
---	---
















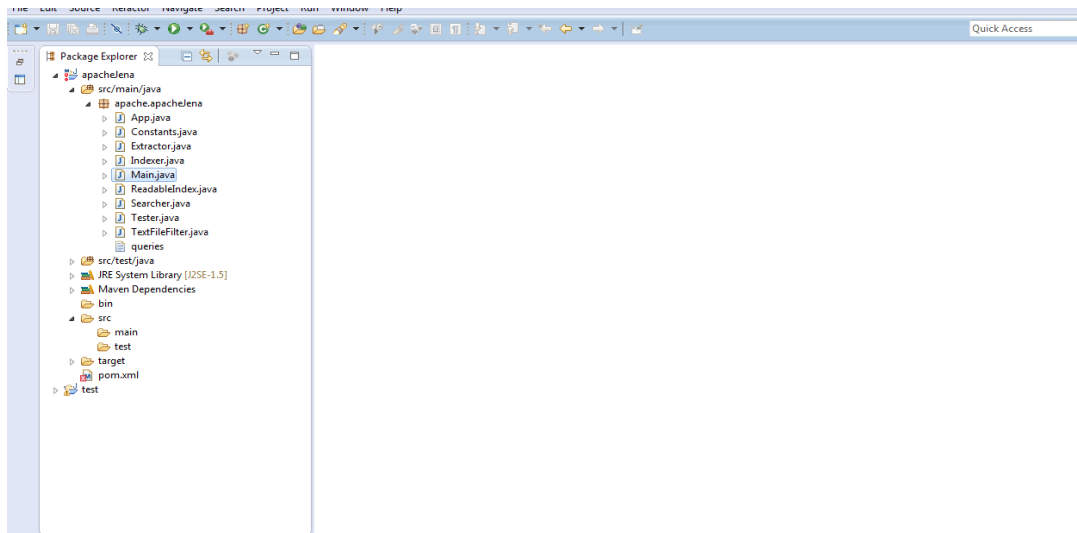


<pre> 480 &lt;!-- http://www.semanticweb.org/Sharjeel/ontologies/2017/10/untitled-ontology-60kmm --&gt; 481 482 &lt;owl:Class rdf:about="http://www.semanticweb.org/Sharjeel/ontologies/2017/10/untitled-ontology-60kmm"&gt; 483   &lt;rdf:subClassOf rdf:resource="http://www.semanticweb.org/Sharjeel/ontologies/2017/10/untitled-ontology-60kmm"&gt; 484   &lt;/owl:Class&gt; 485 486 487 &lt;!-- http://www.semanticweb.org/Sharjeel/ontologies/2017/10/untitled-ontology-60kmm --&gt; 488 489 &lt;owl:Class rdf:about="http://www.semanticweb.org/Sharjeel/ontologies/2017/10/untitled-ontology-60kmm"&gt; 490   &lt;rdf:subClassOf rdf:resource="http://www.semanticweb.org/Sharjeel/ontologies/2017/10/untitled-ontology-60kmm"&gt; 491   &lt;/owl:Class&gt; 492 493 494 495 496 &lt;!-- http://www.semanticweb.org/Sharjeel/ontologies/2017/10/untitled-ontology-60kmm --&gt; 497 498 &lt;owl:Class rdf:about="http://www.semanticweb.org/Sharjeel/ontologies/2017/10/untitled-ontology-60kmm"&gt; 499   &lt;rdf:subClassOf rdf:resource="http://www.semanticweb.org/Sharjeel/ontologies/2017/10/untitled-ontology-60kmm"&gt; 500   &lt;/owl:Class&gt; 501 502 503 504 &lt;!-- 505   /// 506   // 507   // Individuals 508   // 509   // 510   --&gt; 511 512 513 514 &lt;!-- http://www.semanticweb.org/Sharjeel/ontologies/2017/10/untitled-ontology-60kmm --&gt; 515 </pre>	<pre> 515 &lt;!-- http://www.semanticweb.org/Sharjeel/ontologies/2017/10/untitled-ontology-60kmm --&gt; 516 517 &lt;owl:NamedIndividual rdf:about="http://www.semanticweb.org/Sharjeel/ontologies/2017/10/untitled-ontology-60kmm"&gt; 518   &lt;rdf:type rdf:resource="http://www.semanticweb.org/Sharjeel/ontologies/2017/10/untitled-ontology-60kmm"&gt; 519   &lt;first_name rdf:datatype="http://www.w3.org/2001/XMLSchema#string"&gt;Leander&lt;/first_name&gt; 520   &lt;last_name rdf:datatype="http://www.w3.org/2001/XMLSchema#string"&gt;Da Vinci&lt;/last_name&gt; 521   &lt;/owl:NamedIndividual&gt; 522 523 524 525 &lt;!-- http://www.semanticweb.org/Sharjeel/ontologies/2017/10/untitled-ontology-60kmm --&gt; 526 527 &lt;owl:NamedIndividual rdf:about="http://www.semanticweb.org/Sharjeel/ontologies/2017/10/untitled-ontology-60kmm"&gt; 528   &lt;rdf:type rdf:resource="http://www.semanticweb.org/Sharjeel/ontologies/2017/10/untitled-ontology-60kmm"&gt; 529   &lt;first_name rdf:datatype="http://www.w3.org/2001/XMLSchema#string"&gt;Vincent&lt;/first_name&gt; 530   &lt;last_name rdf:datatype="http://www.w3.org/2001/XMLSchema#string"&gt;Van Gogh&lt;/last_name&gt; 531   &lt;/owl:NamedIndividual&gt; 532 533 534 535 &lt;!-- http://www.semanticweb.org/Sharjeel/ontologies/2017/10/untitled-ontology-60kmm --&gt; 536 537 &lt;owl:NamedIndividual rdf:about="http://www.semanticweb.org/Sharjeel/ontologies/2017/10/untitled-ontology-60kmm"&gt; 538   &lt;rdf:type rdf:resource="http://www.semanticweb.org/Sharjeel/ontologies/2017/10/untitled-ontology-60kmm"&gt; 539   &lt;first_name rdf:datatype="http://www.w3.org/2001/XMLSchema#string"&gt;Claude&lt;/first_name&gt; 540   &lt;last_name rdf:datatype="http://www.w3.org/2001/XMLSchema#string"&gt;Monet&lt;/last_name&gt; 541   &lt;/owl:NamedIndividual&gt; 542 543 544 545 &lt;!-- http://www.semanticweb.org/Sharjeel/ontologies/2017/10/untitled-ontology-60kmm --&gt; 546 547 &lt;owl:NamedIndividual rdf:about="http://www.semanticweb.org/Sharjeel/ontologies/2017/10/untitled-ontology-60kmm"&gt; 548   &lt;rdf:type rdf:resource="http://www.semanticweb.org/Sharjeel/ontologies/2017/10/untitled-ontology-60kmm"&gt; 549   &lt;first_name rdf:datatype="http://www.w3.org/2001/XMLSchema#string"&gt;Pablo&lt;/first_name&gt; 550   &lt;last_name rdf:datatype="http://www.w3.org/2001/XMLSchema#string"&gt;Picasso&lt;/last_name&gt; 551   &lt;/owl:NamedIndividual&gt; 552 </pre>
<pre> 554 &lt;!-- http://www.semanticweb.org/Sharjeel/ontologies/2017/10/untitled-ontology-60kmm --&gt; 555 556 557 &lt;owl:NamedIndividual rdf:about="http://www.semanticweb.org/Sharjeel/ontologies/2017/10/untitled-ontology-60kmm"&gt; 558   &lt;rdf:type rdf:resource="http://www.semanticweb.org/Sharjeel/ontologies/2017/10/untitled-ontology-60kmm"&gt; 559   &lt;first_name rdf:datatype="http://www.w3.org/2001/XMLSchema#string"&gt;August&lt;/first_name&gt; 560   &lt;last_name rdf:datatype="http://www.w3.org/2001/XMLSchema#string"&gt;Renoir&lt;/last_name&gt; 561   &lt;/owl:NamedIndividual&gt; 562 563 564 565 &lt;!-- http://www.semanticweb.org/Sharjeel/ontologies/2017/10/untitled-ontology-60kmm --&gt; 566 567 &lt;owl:NamedIndividual rdf:about="http://www.semanticweb.org/Sharjeel/ontologies/2017/10/untitled-ontology-60kmm"&gt; 568   &lt;rdf:type rdf:resource="http://www.semanticweb.org/Sharjeel/ontologies/2017/10/untitled-ontology-60kmm"&gt; 569   &lt;first_name rdf:datatype="http://www.w3.org/2001/XMLSchema#string"&gt;Jan&lt;/first_name&gt; 570   &lt;last_name rdf:datatype="http://www.w3.org/2001/XMLSchema#string"&gt;Vermeer&lt;/last_name&gt; 571   &lt;/owl:NamedIndividual&gt; 572 573 574 575 &lt;!-- http://www.semanticweb.org/Sharjeel/ontologies/2017/10/untitled-ontology-60kmm --&gt; 576 577 &lt;owl:NamedIndividual rdf:about="http://www.semanticweb.org/Sharjeel/ontologies/2017/10/untitled-ontology-60kmm"&gt; 578   &lt;rdf:type rdf:resource="http://www.semanticweb.org/Sharjeel/ontologies/2017/10/untitled-ontology-60kmm"&gt; 579   &lt;first_name rdf:datatype="http://www.w3.org/2001/XMLSchema#string"&gt;Paul&lt;/first_name&gt; 580   &lt;last_name rdf:datatype="http://www.w3.org/2001/XMLSchema#string"&gt;Cezanne&lt;/last_name&gt; 581   &lt;/owl:NamedIndividual&gt; 582 583 584 </pre>	<pre> 585 &lt;!-- 586   /// 587   // 588   // General axioms 589   // 590   // 591   --&gt; 592 593 &lt;rdf:description 594   &lt;rdf:type rdf:resource="http://www.w3.org/2002/07/owl#ObjectClasses"&gt; 595   &lt;owl:members rdf:resource="Collection"&gt; 596   &lt;rdf:description rdf:about="http://www.semanticweb.org/Sharjeel/ontologies/2017/10/untitled-ontology-60kmm"&gt; 597   &lt;rdf:description rdf:about="http://www.semanticweb.org/Sharjeel/ontologies/2017/10/untitled-ontology-60kmm"&gt; 598   &lt;rdf:description rdf:about="http://www.semanticweb.org/Sharjeel/ontologies/2017/10/untitled-ontology-60kmm"&gt; 599   &lt;rdf:description rdf:about="http://www.semanticweb.org/Sharjeel/ontologies/2017/10/untitled-ontology-60kmm"&gt; 600   &lt;rdf:description rdf:about="http://www.semanticweb.org/Sharjeel/ontologies/2017/10/untitled-ontology-60kmm"&gt; 601   &lt;rdf:description rdf:about="http://www.semanticweb.org/Sharjeel/ontologies/2017/10/untitled-ontology-60kmm"&gt; 602   &lt;rdf:description rdf:about="http://www.semanticweb.org/Sharjeel/ontologies/2017/10/untitled-ontology-60kmm"&gt; 603   &lt;rdf:description rdf:about="http://www.semanticweb.org/Sharjeel/ontologies/2017/10/untitled-ontology-60kmm"&gt; 604   &lt;/owl:members&gt; 605   &lt;/rdf:description&gt; 606   &lt;/rdf:description&gt; 607   &lt;/rdf:description&gt; 608   &lt;/rdf:description&gt; 609   &lt;/rdf:description&gt; 610   &lt;/rdf:description&gt; 611   &lt;/rdf:description&gt; 612   &lt;/owl:members&gt; 613   &lt;/rdf:description&gt; 614   &lt;/rdf:RDFS&gt; 615 616 617 &lt;!-- Generated by the OWL API (version 4.2.3, 201704-2018) https://github.com/owlcs/owlapi --&gt; 618 </pre>

### Musuems RDF data on distributed locations:

 Manchester-Museum	Resource Description Framework Document	 Taxila Museum	Resource Description Framework Document
 Scotland Museum	Resource Description Framework Document	 Peshawar Museum	Resource Description Framework Document
 London Museum	Resource Description Framework Document	 Lahore Museum	Resource Description Framework Document
 Birmingham Museum	Resource Description Framework Document	 Multan Museum	Resource Description Framework Document
 Chester Museum	Resource Description Framework Document	 Sawat Museum	Resource Description Framework Document
 Wales Museum	Resource Description Framework Document		

## Testing screenshots:



## Main screen :

