Adaptive RDF Triple Partitioning For Distributed SPARQL Query Processing

by

Yash Shrivastava

(Under the Direction of Krzysztof J. Kochut)

ABSTRACT

Resource Description Framework (RDF) has been extensively used to represent the data for Semantic Web in recent times. Due to a large amount of RDF data, it is difficult to store it in a single system and query it using SPARQL. Instead, it is possible to partition the data into subsets and then query it using federated SPARQL queries. There are many challenges related to distributed querying: for instance, the processing time for a query increases in proportion to the number of distributed joins. We present a study on the impact of query-adaptive partitioning of the RDF data. We present a system called RePart that shuffles the data among the nodes of the cluster according to the incoming query workload to reduce the number of distributed joins while querying. Our evaluation based on several benchmarks demonstrates that the performance of federated queries is improved after performing the repartitioning of the triples according to the query-workload.

INDEX WORDS: RDF, RDF Partitioning, Workload Adaptive Partitioning, Ontologies, Federated Query

Adaptive RDF Triple Partitioning For Distributed SPARQL Query Processing

by

Yash Shrivastava

B.E., Institute of Engineering and Technology, India, 2015

A Thesis Submitted to the Graduate Faculty

of The University of Georgia in Partial Fulfillment

of the

Requirements for the Degree

MASTER OF SCIENCE

ATHENS, GEORGIA

2018

Adaptive RDF Triple Partitioning For Distributed SPARQL Query Processing

by

Yash Shrivastava

Major Professor:	Krzysztof J. Kochut
Committee:	Hamid R. Arabnia
	Ismailcem Budak Arpinar

Electronic Version Approved:

Suzanne Barbour
Dean of the Graduate School
The University of Georgia
August 2018

DEDICATION

Dedicated to my family and friends who helped me throughout this journey.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

With the technological innovation, World Wide Web (WWW) has become an essential part of our life. The Internet has become a universal source of information, and a medium of communication as more and more people express their thoughts by posting tweets, blogs, videos and images. In a real sense, Internet has become a "Global Village" as stated by Marshall Mchulan [2] in 1970. Around 44 billion GB of data is added on the Web daily[1]. The massive amount of data makes it difficult to manage, store, analyze and find relevant information out of it.

The users can read and comprehend the content of the Web pages available on the Internet. They are also able to navigate from one page to another using interconnected links. It has been made possible because of HTML which provides an ability of build interactive components on the Web that is meant for human consumption. The amount of data on the Web is increasing continuously and therefore; it becomes an intricate task for humans to absorb all the data. Semantic Web attempts to automate the Web by adding machine-readable semantics to the data.

1.1 Introduction to Semantic Web

Universal Resource Locator (URL) locates a Web page on the World Wide Web (WWW) using hypertext transfer protocol (HTTP). Most of the resources on the Web are written using HTML that encodes the rendering information for Web browsers and therefore is

[1]https://blog.microfocus.com/how-much-data-is-created-on-the-internet-each-day/

intended for human use. Machines are unable to read and comprehend this information for any purpose. Semantic Web aims to deliver a set of standards and best practices for sharing data and the semantics of the data over the Web for the use by applications [3]. These set of standards are defined by W3C [4] for the people to follow the universal rules to embed machine-readable information on the existing Web page. This machine-readable syntax assists the applications, machines, tools, and programs to efficiently communicate and share the meaning (semantics) of the data. The Semantic Web can be considered as a massive graph of interconnected resources through meaningful edges that represent the relationships between the resources [5]. These are the set of standards defined by W3C for Semantic Web:

- RDF Data Model
- RDF Schema and OWL standards for storing Vocabularies and Ontologies
- SPARQL Query Language

W3C defines RDF [6] as a data model for Semantic Web. RDF has been discussed extensively in section 2.1. The entities on the Web are considered as resources which can be uniquely identified using (Universal Resource Identifiers) URIs. RDF provides an ability to deploy information about the resources on the Web. Relationships between different resources can also be represented with the RDF data model.

1.2 Motivation

The popularity of representing data in RDF format has been growing in the past several years as RDF data does not require a proper schema and it is able to represent information from the diverse sources. RDF is increasingly being used to encode data for the Semantic Web and for data exchange, for example, shopping sites, search engines, social networks and scientific databases are adopting RDF for publishing Web content [7]. Many large knowledge bases

have billions of RDF triples such as Bio2RDF (bio2rdf.org), Uniprot RDF (`dev.isb-sib.ch/projects/uniprot-rdf`) and Yago [7, 8, 9].

As the amount of RDF data increases on the Web, it becomes often too large to fit in a single server. For instance, in performance-critical applications, it is common to use in-memory RDF store, however, the comparatively high cost of RAM limits the capacity of such systems [10]. When dealing with Linked-data [11], it becomes challenging task to integrate large databases that cannot be processed together even in disk-based systems.

To incorporate scalability in the applications dealing with a significant amount of RDF data, many approaches have been discussed where the RDF data is stored in shared-nothing clusters [12, 13, 14, 15, 16, 17]. Querying RDF data which is divided among the nodes of a cluster has following challenges [10]:

1. Intermediate results during distributed join evaluations may grow with the size of the data and it is possible that it might surpass the capacity of the individual node.

2. Intercommunication between nodes of a cluster increases with the evaluation of the distributed joins as triples participating in joins are stored in different nodes.

The challenges reflect that due to the random partitioning of triples among the nodes of the cluster, the number of distributed joins increases. As a result, the time required by a query to evaluate these joins also increases proportionally. Thus, overall query performance of the distributed data store is affected.

An appealing solution is to perform a sophisticated initial partitioning and even replication of the data to minimize the number of distributed joins and increase the data locality [18]. However, in a long run, the systems which performs sophisticated initial partitioning incur high overhead due to data pre-processing and the partitions are unable to adapt according to the query workload [9]. Query-Adaptive partitioning distributed systems redistribute and replicate the content of their nodes by monitoring the query workload instances of the most

frequent ones among workers. The goal is that each node has all the data it needs to evaluate the entire query and there is no need for exchanging intermediate results. In such a *parallel* query evaluation, each node contributes a subset of a complete (partial) result. As a result, the number of distributed joins are reduced and the communication cost for future queries is also drastically reduced or even eliminated [19]. The systems like WARP [14] and Partout [13] do consider the workload during the data partitioning and achieve a significant reduction in replication ratio while showing better results compare to the systems that partition the data blindly. However, both the systems assume a representative (static) workload and do not adapt to the changes. [20] showed that the system needs to continuously adapt to the workloads in order to consistently provide a good performance.

1.3 Contributions

In this thesis, we propose a distributed RDF query processing architecture that performs query workload-adaptive partitioning on RDF data in a cluster. Each query from the query workload is monitored, and data is incrementally rearranged to reduce inter-partition data exchange while executing the queries. It is possible that some amount of the data is replicated to achieve the data locality. Our primary goal is to increase the overall performance of the query workload.

For the exchange of the data to happen, it is required to have a robust, efficient system to transfer RDF triples from one processing node of a cluster to another. We introduce a distributed system RePart, which is deployed on a cluster of machines. It facilitates the exchange of RDF triples between the nodes of the cluster. RePart follow HTTP protocol to communicate between various components. Therefore, the components do not have geographical distance as a constraint which allows for a flexible setup. We also introduce RDF-Metadata Notation (RDF-MN) that provides exhaustive information about the type of triples present in every processing node of RePart. ML describes the capability of a node regarding what

kind of query patterns it can answer. Each processing node of RePart is composed of three components, (*i*) triple store to store the RDF data, (*ii*) an off-the-shelf SPARQL query engine capable of evaluating SPARQL queries on the data stored in the triple store and (*iii*) a metadata store to store the metadata described by ML. This thesis is a part of a bigger project which is focused on developing all the components of RePart. The main contribution of this thesis is to provide an underlying system that is capable of shuffling the triples among the nodes of a cluster when required. Following steps are taken manually to emulate all the other components of RePart:

1. The RDF data is manually divided into n partitions (n corresponds to the number of processing nodes in RePart). This is called the initial partition.
2. Instructions are manually encoded into RScript to repartition the data after considering the query-workload.
3. The generated RScript is provided as an input to RePart.

RePart executes that RScript and (*i*) initiates the exchange of triples from a source node to a destination node, and (*ii*) updates the metadata of each node after the process of triple repartitioning.

In summary, our contributions are:

1. We propose RePart, a distributed system that contains a triple store and an off-the-shelf SPARQL query engine. It is capable of performing efficient query workload adaptive repartitioning of RDF data. It takes RScript as an input that initiates the exchange of the triples.
2. We introduce Metadata Notation which is capable of providing insight about the type of triples that exist in each node. RePart has a metadata store that stores this information.
3. We evaluate RePart using sythetic data that is generated using LUBM and BSBM benchmarks. RePart successfully stores and shuffle RDF data. There is a significant

performance gain regarding query running time after the data is repartitioned using RePart while considering the query-workload.

The rest of the thesis is organized as follows. Chapter 2 covers preliminaries that provide a better understanding of the domain. Chapter 3 reviews existing distributed RDF systems that are related to our work. Chapter 4 presents the architecture of RePart and provides an overview of the system's components and the implementation. Chapter 6 contains the experimental results, and Chapter 7 concludes the thesis.

CHAPTER 2

BACKGROUND

In this chapter, we provide a detailed explanation of the terminology used throughout the thesis. We cover the core concepts related to the Semantic Web and some other terms that will be helpful in understanding the implementation.

2.1 Resource Description Framework (RDF)

Resource Description Framework [6] enable users to embed machine-readable information on the Web. RDF is a language for the representation of the resource. A resource is anything that can be located using URL on the Web. RDF helps to create links between different resources which are already available on the Web, thus, creating a graph. The basic building block of RDF is a statement which has three parts and therefore it is called a triple. A triple consists of the following three parts:

- Subject - a resource being described
- Predicate - a property of that resource
- Object - a value of that property, that can be another resource or a literal

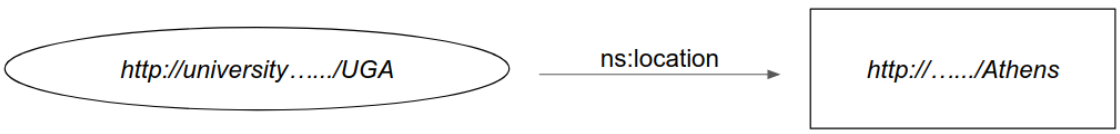

Figure. 2.1: Graphical representation of a RDF Triple

In Figure 2.1, a basic RDF triple is shown. The subject and object are represented in oval and rectangle respectively. The edge of the graph is represented by a predicate. The information described by the triple is: *UGA's location is Athens*. Here, *UGA*, *location*, and *Athens* are URIs and "ns" in the predicate represents a namespace for that URI.

Various RDF triples put together form a graph. In this graph, a subject node is connected to an object node via directed edge, called a predicate. The predicate has a dual purpose: it represents the value of an object and also expresses the relationship between the subject and the object. As stated before, each triple expresses some knowledge and an RDF graph as a whole can be read by machines to gather the information with each of its resources tagged with a Uniform Resource Identifier.

RDF graph helps to connect different data sources by explicitly defining that a resource in a dataset is similar to some different resource in another dataset. This creates a Web of data that can be read by machines. Different parts of the RDF graph can be defined as:

- URIs - they are used to refrence resources unambiguously
- Literals - they are used to describe data values with no clear identity like "abc@xyz.com".
- Blank Nodes - they represent resources for which a URI or a literal is not given

By defining their own URIs, users can add information about any resource in the RDF graph. Thus, an RDF model allow users to integrate the data sources at different location on the Web. RDF triples collectively form a directed labeled graph. Following are the popular formats in which RDF data can be serialized [21]:

- RDF/XML - the official XML serialization of RDF,
- N-Triples - a text format focusing on simple parsing,
- Turtle - a text formal focusing on human readability, and
- Notation 3 - a text format with advance features beyond RDF.

```
@prefix dc: <http://purl.org/dc/elements/1.1/> .
@prefix rdf: <http://www.w3.org/1000/02/22-rdf-syntax-ns#> .
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#>

dc:creator
         rdf:type rdf:Property ;
         # a rdf:Property ;
         rdfs:comment "An entity primarily responsible for creating the resource."@en-US ;
         rdfs:label "Creator"@en-US .
```

Figure. 2.2: Example of a Turtle file describing an RDF Schema, adopted from [1]

2.2 Resource Description Framework Schema (RDFS)

In Semantic Web development, a vocabulary [22] is a set of terms stored in a standard format that people can reuse. A vocabulary of property names has its own namespace to make it easier to use it using other sets of data. RDF Schema [23] is description language for vocabularies which adds some extra knowledge to RDF. RDF Schema is the set of triples that are used to describe other triples in the data. Figure 2.2 shows a few of the triples

from the RDF Schema vocabulary description of the Dublin Core vocabulary [24] in the form of .ttl file. *Prefix* are included at the top of the file to assign a shorter name for a namespace of a URI. This makes it easier to mention the URIs in the triples. *rdf:type* is part of a RDF vocabulary. It describe that the *creator* is an instance of the class *Property*. Similarly, RDF Schema provides an ability to describe the data in a more expresive way. For example, *rdfs:label* lets the user to add a label to describe the subject. Moreover, RDF Schema defines Classes and Properties that creates a taxonomy for arranging the RDF data [24]. Resources in the RDF data can be grouped together into Classes. A member of a class is called an instance of that class. It is possible for a resource to be an instance of more than one class. Classes are also resources, and therefore, they can be defined by properties to add more information about the class itself. For example, the domain and range are the two

properties that can be defined for a class. RDF Schema, however, provides a limited amount of reasoning. OWL overcomes this limitation.

2.3 Web Ontology Language (OWL)

W3C's Web Ontology Language [25] is used to describe complex knowledge about the entities on the Web. It provides a way to represent the relationships between a group of things. Knowledge expressed by OWL can be exploited by computers. The documents in OWL are known as Ontologies [22] which are a form of knowledge management. It is a way to represent all the entities that exists and the relationships between the different entities. Ontologies are a formal definitions of vocabularies that allow the users to define complex structures and new relationships between vocabulary terms and between members of the classes that were defined in the Ontology. Ontologies are collections of RDF triples. Information about the resources mentioned in these triples or relationships between different resources is described using the OWL vocabulary. OWL builds on RDFS, therefore; it has a vocabulary that is richer and more expressive than RDFS itself. Some of OWL's most notable features are its ability to provide a way to state transitive, inverse and symmetrical properties.

Figure 2.3 shows some triples that use OWL's vocabulary. The terms defined in the vocabulary can be accessed using the prefix *owl*. For the resource *spouse*, the property *SymmetricProperty* is defined using OWL's vocabulary. Symmetric property works both ways. Triple *ab:i0432 ab:spouse ab:i9771* means that *ab:io432* has spouse as *ab:19771*. Since *spouse* is a symmetric property, it explicitly conveys information that ab:i9771 also has ab:i0432 as a spouse. It is not required to have an extra triple describing this information. Similarly, *owl:InverseOf* property is also used to describe a relationship between two resources which adds an explicit meaning to the data.

```
@prefix ab: <http://learningsparql.com/ns/addressbook#> .
@prefix rdf: <http://www.w3.org/1992/02/22-rdf-syntax-ns#> .
@prefix rdfs: <http://www/w3/org/2000/01/rdf-schema#> .
@prefix owl: <http://www.w3.org/2002/07/owl#> .

ab:i0432
   ab:firstName  "Richard" ;
   ab:lastName   "Mutt" ;
   ab:spouse     ab:i9771 .

ab:i8301
   ab:firstName  "Craig" ;
   ab:lastName   "Ellis" ;
   ab:spouse     ab:i9771 .

ab:i9771
   ab:firstName  "Cindy" ;
   ab:lastName   "Marshall" ;

ab:spouse
   rdf:type owl:SymmetricProperty ;
   rdfs:comment "Identifies someone's spouse .

ab:patient
   rdf:type rdf:Property ;
   rdfs:comment "Identifies a doctor's patient" .

ab:doctor
   rdf:type rdf:Property ;
   rdfs:comment "Identifies a doctor treating the names resource" ;
   owl:inverseOf ab:patient .
```

Figure. 2.3: Example of a Turtle file describing OWL Schema, , adopted from [1]

There are lots of RDF dataset available on the Web. Users can extract meaningful information and make inferences from this data using the RDF query language SPARQL.

2.4 SPARQL Protocol and RDF Query Language (SPARQL)

As discussed in the previous Sections, using W3Cs standards, users are able to deploy structured data on the Web that can be exploited by the machines. OWL vocabulary allows the users to create ontologies, which represents resources and relationships between various resources. SPARQL Protocol and RDF Query Language (SPARQL) [26] is the W3C

standard that is used for querying the data in RDF graphs for exploring any unknown relationships between resources. SPARQL query RDF data in the same way as SQL query data represented by the Relation Databases. A SPARQL SELECT query comprises of different components as shown in Figure 2.4 in the order they appear in a SPARQL query [27].

```
# prefix declarations
PREFIX foo: <http://example.com/resources/>
...
# dataset definition
FROM ...
# result clause
SELECT ...
# query pattern
WHERE {
    ...
}
# query modifiers
ORDER BY ...
```

Figure. 2.4: Sparql Query Structure

Functionality of each component is described below:

- Prefix deceleration - for abbreviating URIs
- Dataset definition - stating what RDF graph(s) are being queried
- Result clause - identifying what information to return from a query
- Query pattern - specifying what to query in a given dataset
- Query modifiers - allows to rearrange the query results

A SPARQL query is executed against an RDF dataset that consists of RDF graphs. A SPARQL endpoint [28] accepts SPARQL queries that returns the result via HTTP. The results of a SPARQL query can be returned or rendered in various formats: *XML*, *JSON*, *RDF*, *HTML*.

SPARQL attempts to match a triple pattern in an RDF graph. Triple patterns are similar to a triple, except that any of the parts of a triple, i.e. subject, predicate or object can be a variable. Matching of a triple pattern is called binding.

```
@prefix ab: <http://learningsparql.com/ns/addressbook#> .

ab:richard ab:homeTel "(229) 276-5135" .
ab:richard ab:email   "richard49@hotmail.com" .

ab:cindy ab:homeTel "(245) 646-5488" .
ab:cindy ab:email   "cindym@gmail.com" .

ab:craig ab:homeTel "(194) 966-1505" .
ab:craig ab:email   "craigellis@yahoo.com" .
ab:craig ab:email   "c.ellis@usairwaysgroup.com" .
```

Figure. 2.5: Simple RDF Data in TTL format, adopted from [1]

Consider that the data represented by the .ttl file in Figure 2.5 is stored in an RDF graph. The data will be available through a SPARQL endpoint. The query shown in Figure 2.6 can be used to extract the information represented by a triple pattern. The RDF graph *addressbook* will be searched for all the triples that have *ab:craig* as a subject and *ab:email* as a predicate. The answers will be bound to the variable *?craigEmail*, which is an object.

```
@prefix ab: <http://learningsparql.com/ns/addressbook#> .

SELECT ?craigEmail
FROM <http://example.../addressbook>
WHERE
   { ab:craig ab:email ?craigEmail .}
```

Figure. 2.6: Simple Sparql Query

The query is searching for all the email address available for the URI *ab:craig*. In the end, values bound to ?craigEmail are returned. The result is displaced in the Table 2.1 .

```
@prefix ab: <http://learningsparql.com/ns/addressbook#> .

ab:richard ab:homeTel "(229) 276-5135" .
ab:richard ab:email   "richard49@hotmail.com" .
```

Figure. 2.7: Data exposed through SPARQL endpoint: <http://example.../addressbook1>

```
@prefix ab: <http://learningsparql.com/ns/addressbook#> .

ab:cindy ab:homeTel "(245) 646-5488" .
ab:cindy ab:email   "cindym@gmail.com" .

ab:craig ab:homeTel "(194) 966-1505" .
ab:craig ab:email   "craigellis@yahoo.com" .
ab:craig ab:email   "c.ellis@usairwaysgroup.com" .
```

Figure. 2.8: Data exposed through SPARQL endpoint: <http://example.../addressbook2>

Table. 2.1: Query results

?craigEmail
craigellis@yahoo.com
c.ellis@usairwaysgroup.com

It is possible to query more than one RDF graphs exposed through different SPARQL endpoints in a single query. This can be achieved by using federated SPARQL queries.

2.4.1 Federated SPARQL Query

Users are increasingly publishing large amounts of RDF data on the Web. This data can be queried through open SPARQL endpoints. Federated Query [29] allow the users to combine solutions from different RDF datasets. Keyword *SERVICE* is used in the *WHERE* clause that directs a portions of a query towards a particular SPARQL endpoint. Federated query processor merges the results coming from the various SPARQL endpoints. Consider the data represented by .ttl file in Figure 2.5. Currently, the whole data is stored in a single RDF dataset and therefore a normal query is able to obtain the results. Suppose this data is divided in two different RDF datasets, as shown in Figures 2.8 and 2.7. To obtain

```
@prefix ab: <http://learningsparql.com/ns/addressbook#> .

SELECT ?email
FROM <http://example.../addressbook2>
WHERE
 { ab:craig ab:email ?email .
   UNION
 SERVICE<http://example.../addressbook1>
     { ab:richard ab:email ?email . }
 }
```

Figure. 2.9: Federated query

the email addresses of the subjects *ab:craig* and *ab:richard*, a federated query has to be executed. The query is shown in Figure 2.9 which is executed against the SPARQL endpoint <http://example.../addressbook2> . The query search for the triple pattern *ab:craig ab:email ?email* locally. Whereas, SERVICE keyword prompts the federated query processor to search for the triple pattern *ab:richard ab:email ?email* in the RDF dataset exposed through the SPARQL endpoint <http://example.../addressbook1> . Finally, results are combined and displayed as shown in Table 2.2.

Table. 2.2: Federated query results

?email
craigellis@yahoo.com
c.ellis@usairwaysgroup.com
richard49@hotmail.com

There are many available RDF triple stores that provide the functionality of storing and querying the RDF data, such as Redland [30], Sesame, Jena [31], Virtuoso, etc. For our thesis, we have selected Virtuoso [32] as the system to store and query RDF triples.

```
create table DB.DBA.RDF_QUAD (
  G IRI_ID,
  S IRI_ID,
  P IRI_ID,
  O any,
  primary key (G,S,P,O) );
create bitmap index RDF_QUAD_OGPS on DB.DBA.RDF_QUAD (O, G, P, S);
```

Figure. 2.10: RDF QUAD

2.5 Standard RDF Triple Store With Associated SPARQL Processor

According to the documentation, "OpenLink Virtuoso 7.0 [32] is a revolutionary, next generation, high-performance virtual database engine for the Distributed Computing Age". It provides support to store and query the RDF data. Virtuoso stores RDF triples in the form of Relational database tables. One of the main tables of the default RDF storage system is shown in Figure 2.10. Each triple in RDF QUAD is represented as one row in RDF triples. The table is called "QUAD" as it stores triple with one extra column representing the graph to which that triple belongs to. Therefore, the columns represent the graph, subject, predicate, and the object. To answer the SPARQL queries, Virtuoso converts them to SQL queries and executes them against the tables, where the RDF triples are stored. Virtuoso also provides the support to run federated queries over multiple SPARQL endpoints. In Virtuoso, SPARQL and SQL share the same query execution engine, query optimizer, and the cost model. For federated queries, the logic for optimizing the message flow between multiple end-points on the Web is similar to the logic for message-optimization on a cluster. Virtuoso also has an Interactive SQL (ISQL) which provides a faster way to perform some tasks, like bulk loading of RDF triples.

2.6 REpresentational State Transfer (REST)

There are numerous Web Services that are exposed to the internet for programmatic access. Published Web APIs can be accessed by application developers to perform various kind of tasks. For example, Twitter[1] has its own Twitter API[2] that allows users to engage with the Twitter platform. It provides functionalities to search tweets or filter real-time tweets. These APIs are similar to Web pages, however, their response does not contain HTML or CSS . The response is in barebone data formats like XML or JSON which helps application developers to parse the data and use it accordingly.

There are many ways to build such Web Services or APIs. One way to build them is by following REST architecture and they are called RESTful Web Services. The term REST was introduced by Roy Fielding in the year 2000. REST stands for REpresentational State Transfer. It is an architectural style which consists of certain constraints and criterions that can be used as guidelines while designing the Web Services. RESTful Web Services are lightweight, scalable and manageable services that uses concepts behind HTTP. The goal when building a RESTful Web service is to make it as RESTful as practically possible.

Following are the primary constraints required under RESTful architecture:

- Resource-based URI - Unique URIs should be created for every resource and the resources should be addressable using this identifier. URI should contain nouns and not verbs. Mostly, nouns are resource name themselves.
- Unique Interface - All interactions between client's application and resources in the system should be carried out through a constrained unique interface. The system should provide well-defined set of methods to manipulate the data related to resources.

[1]https://twitter.com/
[2]https://developer.twitter.com/en/docs

- Manipulation of resources through representation - Data related to resources can be returned in many formats such as HTML, JSON, XML, PNG, etc. These formats are the *representations* of the identified resource [33]. RESTful applications may support different *representations* of the same resource at a given URI. RESTful applications allow a client to define what *representation* of a resource is required. This can be accomplished by passing **Accept** HTTP header with each request of a resource. *Representations* can be sent from a client to a server for the purpose of updating the resource. This is achieved by a **Content-type** HTTP header that is passed by the client to the server with each resource sent to the server.

- Self-descriptive messages - Client and server communicate via messages and Restful Web Services expect these messages to be self-descriptive. Each message should contain all the information to complete the task. The messages can have a body and a meta-data.

- Stateless Interaction - All client-server communications are stateless. The server does not store information about the latest HTTP request and treats every request as a new request.

All Web Services that incorporate these constraints within their architectural design can be called RESTful Web Services. REST uses HTTP methods for exposing services provided by the system. These methods can be classified based on two aspects, (i) their functionalities and (ii) method Idempotence. Idempotence is a property of certain operations in methamatics and computer science, that can be applied multiple times without changing the result beyond the initial application. A client can perform different operations on resources by using the following four HTTP methods:

- GET - Provides a read-only access to a resource. It is used to get information about a resource which is specified by a URI. GET method does not change the state of a resource, therefore; it is idempotent.

- PUT - Provides a way to update or change a resource. The new content is sent in the body of the PUT request from the client side to the server. For a specific resource, the new data replaces the old content. This method is idempotent as well.
- DELETE - Provides a way to delete a resource and is idempotent.
- POST - Provides a way to create a resource. The body of this method contains the content of the new resource. Every time a POST request is made, a new resource is created, therefore, it is not idempotent.

In this thesis, we have used a RESTful architecture to facilitates the exchange of triples among the nodes of a cluster.

CHAPTER 3

RELATED WORK

3.1 Distributed Graph Query Processing Systems

3.1.1 LOOM

The paper [34] discuss a general graph partitioning system. According to the query workload Q, Loom allocates the new vertices and edges to various partitions. The primary focus of this paper is online graphs (graphs that continuously increase). For example, the incoming stream of data that has to be added to the already existing graph. As the graph is already divided into partitions, it is critical to decide to which partition, the incoming edges and vertices have to be allocated. Therefore, LOOM aims to efficiently partition the large, dynamic graphs to optimize for given streams of sub-graph pattern matching queries.

Loom has the following main goals:

1. First, when answering queries from a given workload, it aims to discover patterns of edge traversals that are common among various queries. For example, it tries to find the sub-graph patterns that are frequently traversed by the queries during execution.
2. Second, it detects similar instances of these sub-graph patterns in the incoming streams of the online data which is itself a combination of edges and vertices.
3. Finally, it tries to assign these matched patterns entirely within an individual partition or across as few partitions, as possible, in order to reduce the inter-partition traversal and increasing the average performance of the queries.

A query motif is a graph which occurs with a frequency of a user-defined threshold T as a sub-graph of query graphs from Q. At first, Loom employs a trie-like data structure to index all of the possible sub-graphs from all the queries in Q. Then, it identify the sub-graphs that occur most frequently (motifs). Secondly, Loom monitors all the incoming streaming data. It checks whether each new edge added to the graph creates a sub-graph which matches one of the motifs. To achieve this matching, it uses graph stream pattern matching procedure. Finally, it employs an existing partitioning heuristics to assign each sub-graph that matches a motif to an individual partition, thereby reducing the interpretation- traversal. For experiments, the graphs are streamed using three techniques: (*i*) Breadth-first, (*ii*) Random and, (*iii*) Depth-first. The tests indicate that Loom significantly reduces the number of inter-partition traversals required when executing query workload over the partitions.

3.1.2 Sedge

Sedge [15] uses similar techniques for SPARQL query execution on top to the vertex-centric processing model Pregle [35]. The authors propose a graph partition management strategy that supports overlapping partitions and replication for fast graph query processing by eliminating a constraint in Pregel that does not allow duplicate vertices in partitions. The focus is to replicate some regions of a graph and distribute them in multiple machines to serve queries in parallel. For this goal, the authors have developed three techniques in Sedge. (i) Complementary partitioning is to find multiple partition schemes such that their partition boundaries are different from one another. (ii) Partition Replication is to replicate the same partitions on multiple machines to share the workload on these partitions. (iii) Dynamic Partitioning is to construct new partitions to serve cross-partition queries locally.

3.1.3 DREAM

DREAM [16] is a distributed RDF system that avoids partitioning RDF datasets and partitions only SPARQL queries. Dataset is replicated at each node while an incoming query is decomposed into subqueries and sent to the suitable node to be evaluated. DREAM follows a master-slave architecture where each machine uses RDF-3X [36] on its assigned data for statistical estimation and query evaluation. A user submits a SPARQL query to the master that converts the query to a graph pattern and feeds it to the query planner that decomposes the graph to sub-graphs. Master then place the sub-graphs at a single-slave machine and all machines are executed in parallel. It also avoids expensive intermediate data shuffling and only exchanges small auxiliary data. Depending on the query complexity, DREAMs optimizer decides to run it either in a centralized or a distributed fashion. Although DREAM does not incur any partitioning overhead, it exhibits excessive replication and costly preprocessing because of the centralized database construction.

3.1.4 EAGRE

EAGRE [37] is a technique that provides a new representation of RDF data on Cloud platform. It also proposes an I/O efficient strategy to evaluate SPARQL queries as quickly as possible. To improve the efficiency in answering SPARQL queries on a Cloud platform, it necessary to have an RDF data remodeled and organized. Therefore, EAGRE transforms the RDF data into an entity graph by grouping triples based on the subject where each subject is called an entity. Then it groups entities with similar properties into an entity class. EAGRE then generate a compressed entity graph that contains only entity classes and partition it using METIS. At each machine, entities belonging to the same class are treated as high dimensional data indexed by a Space-Filling Curve. This maintains an order preserving the layout of the data which fits well range and order by queries. EAGRE converts SPARQL

queries into MapReduce jobs. Therefore, it suffers from the overhead of MapReduce joins for queries that cannot be evaluated locally.

3.1.5 S2RDF

S2RDF [38] is a SPARQL processor based on the in-memory cluster computing framework called Spark. It introduces relational partitioning schema for the RDF data called Extended Vertical partitioning (ExtVP) that can significantly reduce the input size of a query. The reduction of data input size tends to be more effective than decrease in join operations for Spark as it is an in-memory system. ExtVP is an extension of Vertical Partitioning (VP) introduced in [39]. In such representation of triples, a triple pattern based on a predicate can be obtained by accessing the corresponding VP table, which leads to a reduction of input size. The size of these tables is highly skewed in a typical RDF dataset with some tables containing only a few entries, while others comprising a large portion of the entire graph. Hence, there are still a lot of dangling tuples, i.e., input tuples that do not contribute to the output of a query, which are potentially shuffled during query execution. ExtVP uses semi-join reduction to minimize data skewness and eliminate dangling triples that do not contribute to any join. For every two vertical partitions, S2RDF pre-computes join reductions: (*i*) subject-subject join, (*ii*) subject-object join and (*iii*) object-subject join. As these semi-join tables are much smaller compared to the base tables, they are used for joins, while query evaluation. S2RDF converts the SPARQL queries into SQL jobs which are then executed on top of Spark SQL.

3.1.6 DARQ

DARQ [17] is an engine for federated SPARQL queries. It is a system to query an RDF data stored in a distributed environment, which is exposed through a SPARQL endpoint. It gives the user an impression to query one single RDF graph. However, the real data is distributed around the Web. A service description language enables DARQ to obtain information about

the data available on each service. This helps the DARQ's query engine to decompose the SPARQL query into sub-queries, each of which can be answered by individual services. To speed up the query execution, DARQ uses a query optimization algorithm that re-writes the query and builds a cost-based query execution. DARQ is compatible with any endpoint that supports SPARQL standards. Apart from this, nothing else is required.

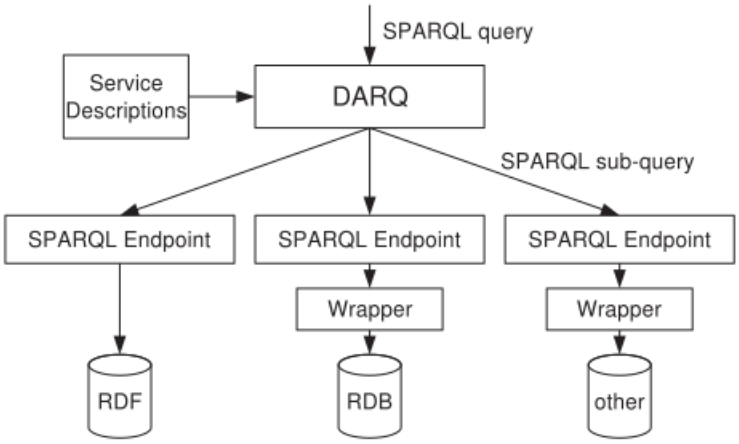

Figure. 3.1: DARQ architecture

DARQ's architectural diagram is shown in the Figure 3.1. A wrapper can be used to convert the data from other formats to RDF. Endpoints provide access to the RDF data stored in data store. An incoming query is received by the DARQ's engine, an d the query is processed in 4 steps:

1. In the first step, the query is parsed into a tree model by using the parser shipped with ARQ.

2. In the second stage, the DARQ query divides the incoming query into sub-queries by using the information from various endpoints (service descriptions). Each of the queries could be answered by one known data source (endpoint).

3. In the third stage, the sub-queries are sent to query optimizer that builds an optimized query execution plan.

4. Finally, the query plan is executed. The sub-queries are sent to the respective endpoints and at the end, the results are combined.

```
[]    a       sd:Service ;
      sd:capability [ sd:predicate foaf:name ;
                      sd:objectFilter "REGEX(?object,"^[A-R]")";
                      sd:triples 51 ] ;
      sd:capability [ sd:predicate foaf:mbox ;
                      sd:triples 51 ] ;
      sd:capability [ sd:predicate foaf:weblog ;
                      sd:triples 10 ] ;
      sd:totalTriples "112" ;
      sd:url  "EndpointURL" ;
      sd:requiredBindings [ sd:objectBinding foaf:name ] ;
      sd:requiredBindings [ sd:objectBinding foaf:mbox ] .
```

Figure. 3.2: Service Description example

Service description provides data about the RDF data available from each data store in the form of capabilities. Service descriptions are represented in the RDF format. The capability is a measure of what kind of triple patterns can be answered from a data store. For example, consider the Figure 3.2. Here, it is possible to say that a Service A can only answer queries for names starting with a letter A to R, whereas, another service can answer queries for the names with letter Q to Z. In addition to that, statistical data available from data stores in the form of service descriptions, like the number of total triples for a given predicate, help the query optimizer to generate the cost-effective execution plan. By using these service descriptions, query planner can find relevant sources and possible sub-queries. To find the appropriate data sources for an incoming query, the algorithm matches all the triple patterns in the query to the capabilities of various data sources. The Matching compares the predicate in a triple pattern to the predicates defined in the capabilities and finds constraints for subject and objects. The query plan consists of multiple sub-queries after the process of Query planning. Query optimizer builds a cost-effective execution plan.

3.1.7 PARTOUT

PARTOUT [13] is a workload-aware distributed RDF engine that generates fragments of the whole dataset based on a query log and allocates the fragments to nodes in a cluster. PARTOUT makes the following contributions:-

1. It provides a system for RDF storage that can handle updates to the RDF dataset.
2. A partitioning and allocation algorithm for RDF dataset is introduced while considering a given query workload.
3. It provides an optimizer for distributed SPARQL query processing and a cost model for proposed architecture.

There are three primary steps involved in the partitioning and allocation process: (*i*) extract representative triple patterns from a query workload by applying normalization and anonymization,(*ii*) define a load score for each fragment and sorts fragments in descending order, and finally (*iii*) for each fragment, calculate a benefit score for allocating it to each machine. The benefit score takes into account both the machine utilization well as the fragment locality.

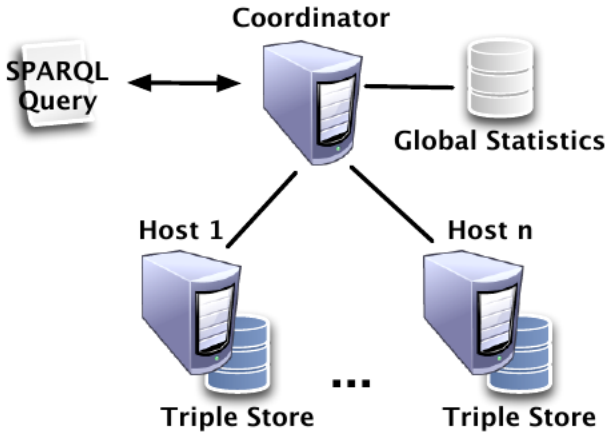

Figure. 3.3: System Architecture of Partout

3.1.8 WARP

WARP [14] is a distributed SPARQL engine that combines graph partitioning techniques with a workload-aware replication of triples across the partitioning which enables query execution for complex queries. The authors also propose cost-aware query optimization and query execution for arbitrary queries without the need for MapReduce jobs. WARP uses the underlying METIS [40] algorithm to assign each vertex of the RDF graph to a partition. Triples are then assigned to partitions according to their subjects. Each partition is stored at a dedicated host in a triple store (RDF-3X). WARP uses a representative query workload to replicate frequently accessed data by extending the $n - hop$ guarantee method [41]. WARP determines queries center node and radius. If the query is within $n - hop$ guarantee, it sends the query to all partitions to be executed in parallel. On the other hand, if it is a complex query, it is decomposed into several sub-queries for which a distributed query evaluation plan is created. Subqueries are evaluated in parallel by all machines and the results are sent to the master, which combines them using the merge join.

3.1.9 AdPart

In the paper [12], the authors propose AdPart, a distributed, in-memory RDF system, that re-partitions the RDF data incrementally, according to the query workload to increase the query performance. AdPart has two primary functions:

1. Initially, AdPart does not require expensive data preprocessing. It uses hash partitioning that avoids the cost associated with initial partitioning of data.

2. AdPart provides an ability to monitor and index workloads in the form of hierarchical heat maps. It also introduces Incremental ReDistribution (IRD) technique for data portions that are accessed by hot patterns, which are guided by query workload. Thus, AdPart adaptively partitions the data.

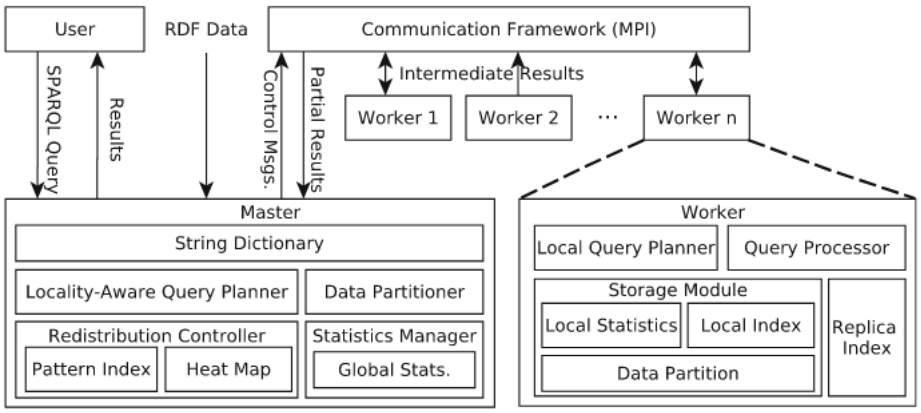

Figure. 3.4: System Architecture of AdPart

In the figure 3.4, the architecture of AdPart is shown. It follows master-slave paradigm and uses MPI (message passing interface) for communication among the nodes. Master begins by loading the partitions among the respective workers. Each query is sent to the master, which decides weather the query should be executed in distribute or parallel fashion. In parallel mode, the query is executed by each worker at the time without communication. In distributed manner, the execution of the query by all worker require communication. Various components of AdPart's system are discussed below.

Master

- **String Dictionary** encodes RDF strings into numerical IDs and build an i-directional dictionary.
- **Data Partitioner** performs node-based partitioning by hashing the subject values. Therefore, any star based query can be evaluated without any communication cost.
- **Statistics Manager** maintains statistics about the RDF graph. This is used to generate query evaluation plan and during RePartitioning.

- **Redistribution controller** monitors query workload in the form of heat maps and triggers the incremental redistribution of triples for the patterns that occur often and are called hot patterns. Data accessed by hot patterns can be answered by all workers without communication. If a hot pattern is replicated, it is indexed in a structure called pattern index (PI).
- **Locality-aware query planner** decides if an incoming query can be executed without communication. For this purpose, it uses global statics from statistics manager and PI. Query are executed by each worker independently if it can be answered without communication. On the other hand, for distributed queries, the planner uses hash-based data locality and triple patterns in the query to generate a plan which requires minimum communication cost.

Worker

- **Storage Module** Each worker stores its local set of triples in an in-memory data structure. Primarily, each triple is hashed on its predicate. Therefore, resulting *predicate* index supports search by predicate. To re-partition each bucket of triples with same predicate, two hash-maps are used that support search operations like *predicate-subject* index and *predicate-object* index. When answering a query, if a predicate value is variable, the they iterate over all the values of a predicate.
- **Replica Index** Each worker has a local replica indexes that index the data which is replicated as a result of adaptivity.
- **Query Processor** Each workers query processor can run in two modes, (i) distributed mode for the queries that require communication. (ii) parallel mode for queries that can be answered without communication.

- **Local query planner** Workers make a local plan for queries that execute in parallel. For example, star queries joining on subjects are processed in parallel.

AdPart Adaptivity

AdPart redistributes the data needed for a current workload and adapts to the workload change. The IRD component of AdPart is a combination of hash-partitioning and k-hop replication, guided by a query workload. Given a hot pattern (that often occurs in the query workload), AdPart selects a specific vertex called core vertex. It transforms the pattern into a redistribution tree rooted at core vertex to group the data accessed by pattern around the binding of this vertex. Then, starting from the vertex, first-hop triples are hash distributed based on core-bindings. AdPart uses redistributed patterns to answer the query in parallel without communication.

3.2 Comparison

This section covers detailed comparison (summarized in Table 1) of all the systems[1] discussed above with respect to various criterion mentioned below:

3.2.1 Partitioning Strategy

All the distributed systems have to partition the data among the nodes of the cluster, and they apply different techniques to do that. In this section, various partitioning techniques are compared. AdPart and Sedge use lightweight subject hashing technique where they assign a triple to a node according to the subject hash value. DREAM, instead of partitioning, replicates the data among the nodes of the cluster. EAGRE depends on METIS for the initial partitioning of the data. WARP, on the other hand, applies METIS to partition the

[1] AdPart-NA, which is a nonadaptive version of AdPart, is also included for comparison

Table. 3.1: Summary of various Distributed RDF systems

System	Partitioning Strategy	Execution Model	MapReduce Based	Replication Level	Workload Awareness	Dynamic Partitioning	Specialized Q Processor
S2RDF	Extended Vertical Partitioning	SPARQL to SQL	Yes	Partial	No	No	Yes
EAGRE	METIS	MapReduce-based Join	Yes	None	No	No	Yes
Sedge	Subject Hash	Vertex-Centric BGP matching	No	Complete	No	No	Yes
DREAM	No partitioning: full replication	RDF-3X	No	Complete	No	No	Yes
AdPart-NA	Subject Hash	Distributed Semi join	No	None	No	No	Yes
DARQ	-	ARQ + Built-in Query Engine	No	None	No	No	Yes
Partout	Workload based fragmentation	RDF-3X	No	None	Yes	No	Yes
WARP	METIS on query workload	RDF-3X	Yes	Partial	Yes	No	Yes
AdPart	Subject Hash	Distributed Semi Join	No	Partial	Yes	Yes	Yes

graphical query pattern. S2RDF uses Extended Vertical Partitioning (ExtVP) to partition the data. For every two VPs, ExtVP pre-computes its join reductions and materializes the results as new partitions (tables for Spark SQL) in HDFS. Hence, ExtVP incurs significantly higher overhead. Partout divides the data into fragments according to the query workload and stores them into different nodes. AdPart initially partitions the data using subject hashing and it incrementally updates the partitions as the query workload changes.

3.2.2 Execution Model

This section compares various execution models used by distributed RDF frameworks. DREAM, Partout, and WARP take advantage of RDF-3X to execute SPARQL queries in a distributed domain. RDF-3X engine is an implementation of SPARQL that achieves excellent performance by pursuing a RISC-style architecture developed for distributed SPARQL querying. S2RDF has been developed on Spark, which is an in-memory cluster computing system that runs on Hadoop. S2RDF converts the SPARQL queries into multiple SQL oper-

ations, which are then executed using Spark. Similarly, EAGRE also preforms SPARQL joins by leveraging MapReduce. However, using MapReduce also lead to higher pre-processing overhead in some cases. Sedge uses Vertex Centric basic graph pattern matching for query evaluations. DARQ has an inbuilt query engine that also uses ARQ for parsing the incoming queries. AdPart uses their implementation of distributed semi-join to reduce the overhead of intercommunication between the nodes while query execution.

3.2.3 MapReduce Based

TSome distributed RDF systems use the underlying framework, such as MapReduce, to accelerate the evaluation of joins that occur, while executing the SPARQL query in a distributed fashion. For example, EAGRE uses MapReduce to process the joins. S2RDF, on the other hand, converts the SPARQL queries into SQL and executes them over Spark. WARP gathers the query evaluation results from various nodes using merge join that is executed using MapReduce. S2RDF has higher preprocessing overhead but shows significant performance improvements compared to purely MapReduce-based systems.

3.2.4 Replication Level

Some systems use data replication to improve the overall query performance. For example, S2RDF, WARP, and AdPart partially replicate the data. This is shown in the Table 1 as "Partial" in the "Replication Level" column. Each system has a different technique to decide which triples to be replicated. Some systems, for example, DREAM and Sedge, reproduce the entire data in every node of a cluster. It then partitions the query and sends the sub-queries to the optimal node for execution. DREAM exhibits excessive replication and costly preprocessing because of the centralized database construction. Rest of the systems do not replicate any data.

3.2.5 Workload Awareness

Most of the distributed RDF systems blindly partition the data and rely on their execution method to reduce the intercommunication overhead while executing the query. However, according to [42], there might be no single partition that is good for all workloads. Therefore, systems like Partout, WARP and AdPart change the partitions by monitoring the query-workload. This is an efficient way to reduce the distributed-joins that occur during query execution due to random partitioning. Adapting the partitions to the workload change increases the data locality and improves the overall query performance.

3.2.6 Dynamic Partitioning

Dynamic partitioning refers to changing the data in each node incrementally by monitoring the workload. AdPart is the only system that performs incremental adaptive partitioning according to the query workload. On the other hand, Partout and WRAP also partitions based on the workload, however, if the workload is changed, they adapt only by applying expensive re-partitioning of entire data which incur high communication cost for a dynamic workload.

3.2.7 Specialized Query Processor

It can be observed from Table 1 that none of the distributed systems use standard triple store, which has an associated SPARQL query processing engine for storing and querying the RDF data. Most of the systems have either (*i*) implemented their own SPARQL query engine (AdPart), (*ii*) use the underlying framework like MapReduce to execute the query (EAGRE), or (*iii*) apply a modified SPARQL query engine like RDF-RX (WARP, Partout). Thus, they use a Specialized SPARQL query engine.

CHAPTER 4

ADAPTIVE PARTITIONING OF RDF DATA

As mentioned earlier, RDF has become a very popular framework to publish the data for Semantic Web due to its flexible and universal graph-like data model. Therefore, ever-increasing size and collections of RDF data exhibits the need for scalable RDF systems [38]. This section introduces the distributed RDF architecture and demonstrates the challenges related to static partitions that do not adapt to the workload changes. Furthermore, we discuss the need for adaptive partitioning of the RDF data and explain how it reduces the problems associated with querying RDF data which is stored in a distributed environment.

4.1 Initial Partitions

To deal with a large amount of RDF data, many clustered RDF systems distribute the RDF graph G into n number of partitions, $P_1, P_2..P_n$, which are called Initial partitions. The primary purpose is to divide the triples in a way to reduce the number of distributed joins that are evaluated during the execution of a SPARQL query. Evaluation of distributed joins increases the inter-communication cost among the workers as the intermediate data has to be exchanged. If most of the queries in the query set have many distributed joins, the query performance of the whole query set declines.

There are numerous ways to generate the initial partitions. For example, RDF triples can be randomly partitioned among the machines, based on the hash values of their subject, predicate or object. Initial partitions can also be generated according to the given query-workload. We assume that the given query-workload QW is a set of SPARQL queries $Q_1, Q_2...Q_n$.

Systems that partition the triples randomly incur a low pre-processing cost. However, this method introduces many distributed joins. On the other hand, systems that use sophisticated partitioning, suffer from high pre-processing cost and sometimes high replication of triples.

4.2 Metadata

RDF Metadata can be termed as a data that describes the RDF triples. In a distributed domain, it is essential to keep track of what kind of triples are stored in each partition. Primarily, RDF metadata helps in discovering if the existing triples in any partition can answer a given query pattern. This helps to identify all the nodes that contains the required data to answer the query. Based on the metadata, the original SPARQL query is decomposed into several sub-queries, where each sub-query is sent to its relevant SPARQL endpoints. The results of the sub-queries are then joined together to answer the original SPARQL query.

Various distributed RDF stores use metadata differently. For example, DARQ [17] has a service description which provides a declarative description of the data available from an endpoint. AdPart [12] uses global statistics to decide if the query can be processed without communication. Some systems, like FedX [43] send SPARQL ASK to collect the metadata on the fly. Based on the results of SPARQL ASK queries. FedX decomposes the query into subqueries and assign subqueries with relevant SPARQL endpoints.

4.3 Query Rewriting

The incoming query can be rewritten for query optimization [44]. Another reason for query rewriting, especially in a distributed domain is to reorder the basic query pattern according to the availability of these pattern in the nodes of the cluster [17]. For instance, when the RDF data is distributed on multiple nodes, the result of the incoming query might include the data from any given node. The metadata is used to decide which query pattern a node

Table. 4.1: Initial Partition

$Node1$	$Node2$
rdf:type	rdf:type
ub:worksFor	ub:undergraduateDegreeFrom
ub:name	ub:mastersDegreeFrom
ub:telephone	ub:emailAddress
ub:advisor	

can answer. With this information, a query can be rewritten to ask a given node only for the pattern it can answer. Later, all the data is aggregated and presented as a complete result. In our proposed architecture, the incoming query is rewritten into a federated query (Section 2.4.1). The federated query is then sent to the suitable node where a standard SPARQL query processor executes it and gathers the result (Section 5.2).

```
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX ub: <http://swat.cse.lehigh.edu/onto/univ-bench.owl#>
SELECT ?prof ?name ?email ?tel
WHERE
{
?prof rdf:type ub:Professor.
?prof ub:worksFor <http://www.Depart0.Univ0.edu>.
?prof ub:name ?name.
?prof ub:emailAddress ?email.
?prof ub:telephone ?tel
}
```

Figure. 4.1: First query added to the QW

To understand this process, consider the two partitions shown in the table 4.1. Let's call this the initial partition. $Node1$ and $Node2$ contains all the triples that have the predicates which are listed in the table under the column name $Node1$ and $Node2$ respectively. The data is generated using the benchmark LUBM [45]. Table 4.2 lists some triples with predicates included in the initial partition. Initially, the query workload is empty. Then, the first query is added to the query workload which is shown in Figure 4.1. The query asks for the name, email address and the telephone number of all the professors that work for the specified department. It is evident that the data from both nodes have to be accessed to obtain the

Table. 4.2: Sample RDF triples

PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#
PREFIX ub: <http://swat.cse.lehigh.edu/onto/univ-bench.owl#>

Subject	**Predicate**	**Object**
http://www.Depart0.Univ0.edu/Professor0	rdf: type	ub:Professor
http://www.Depart7.Univ6.edu/Student3	rdf: type	ub:Student
http://www.Depart0.Univ0.edu/Professor0	ub: worksFor	http://www.Depart0.Univ0.edu
http://www.Depart0.Univ0.edu/Professor0	ub: name	"Professor0"
http://www.Depart7.Univ6.edu/Professor0	ub: telephone	"xxx-xxx-xxxx"
http://www.Depart0.Univ0.edu/Student3	ub: advisor	http://www.Depart0.Univ0.edu/Prof0
http://www.Depart7.Univ6.edu/Professor0	ub: emailAddress	"Prof0@Depart0.Univ0.edu"
http://www.Depart11.Univ9.edu/Student3	ub: undergraduate DegreeFrom	http://www.University0.edu
http://www.Depart5.Univ2.edu/Student3	ub: masters DegreeFrom	http://www.University0.edu

answer to the query because the triples that have *ub:emailAddress* as their predicates are stored in the different node than the rest of the triples. This shows that the query involves one distributed join. There are two available options, either we can execute the query on $Node1$ and send the partial results to $Node2$, or the inverse could be done. Executing the query on $Node2$ will lead to huge intercommunication cost because of the excessive partial results will be transferred between $Node1$ and $Node2$. Executing the query on $Node1$ node would be more cost-effective as most of the triples that are accessed by the query are stored in that processing node. Using this knowledge, the query is rewritten into a federated query, as shown in Figure 4.2. It is then executed on the $Node1$.

```
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX ub: <http://swat.cse.lehigh.edu/onto/univ-bench.owl#>
SELECT ?prof ?name ?email ?tel
WHERE
{
?prof rdf:type ub:Professor.
?prof ub:worksFor <http://www.Depart0.Univ0.edu>.
?prof ub:name ?name.
?prof ub:telephone ?tel.
SERVICE <http://255.255.255.2:8080/sparql> {?prof ub:emailAddress ?email}
}
```

Figure. 4.2: Federated query to be executed on $Node2$

4.4 Adaptive Partitioning

As discussed in the previous section, the query workload currently consists of only one query, which can be denoted as $QW = \{Q_1\}$. Q_1 is rewritten into a federated query because it involves a distributed join which requires the intermediate results to be transferred between the processing nodes. This incurs a communication cost and degrades the query performance. As the query workload increases, the queries that includes distributed joins may also increase.

To improve the average performance of the query set, it is imperative to reduce the number of these joins. By regularly monitoring the query workload, it is possible to detect the distributed joins that frequently occur in the queries and transfer the triples that are part of these joins from one node to another to make that data local to a single node. This increases the data locality, which means that all the queries that earlier required the data from different nodes can now execute from within a single node to produce the result. This significantly reduces the inter-node communication cost and improves the query performance. This process is called adaptive partitioning of the dataset based on the observed queries.

To understand the process of adaptive partitioning, consider a list DS that shows the number of distributed joins introduced by each query from QW. Currently, there is only one query in QW. Therefore, the lists can be represented as

Table. 4.3: 1st Adaptive Partition

$Node1$	$Node2$
rdf:type	rdf:type
ub:worksFor	ub:undergraduateDegreeFrom
ub:name	ub:mastersDegreeFrom
ub:telephone	
ub:advisor	
ub:emailAddress	

$$QW = [Q_1]$$

$$DJ = [1]$$

because Q_1 introduce one distributed join. The aim is to minimize the every element in the list DJ. Again, consider Q_1 (figure 4.2) and the initial partition (table 4.3). If all the triples with predicate $ub : emailAddress$ are moved to $Node1$, the distributed join is eliminated, which increases the query performance. Let's call this new partition the 1st adaptive partition, shown in the table 4.3. The list DJ can now be represented as,

$$DJ = [0]$$

```
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX ub: <http://swat.cse.lehigh.edu/onto/univ-bench.owl#>
SELECT ?student ?advisor
WHERE
{
?student ub:undergraduateDegreeFrom ?univ.
?student ub:mastersDegreeFrom ?univ.
?student ub:advisor ?advisor
}
```

Figure. 4.3: Second query added to the QW

Suppose, a new query, shown in Figure 4.3 is added to the query workload. The query requests for all the students and their advisors who have an undergraduate degree and master's degree from the same college. The query adds a distributed join as the data from both the nodes have to be accessed. Therefore, both the lists are updated.

Table. 4.4: 2nd Adaptive Partition

$Node1$	$Node2$
rdf:type	rdf:type
ub:worksFor	ub:undergraduateDegreeFrom
ub:name	ub:mastersDegreeFrom
ub:telephone	ub:advisor
ub:emailAddress	

$$QW = [Q_1, Q_2]$$
$$DJ = [0, 1]$$

This distributed join is removed by transferring all the triples with the predicate $ub : advisors$ from $Node1$ to $Node1$. The query can now be executed entirely on $Node2$, thus eliminating the expensive inter-communication cost between the nodes and increases the overall performance. The 2nd adaptive partition is shown in the table 4.4.

The updated lists looks like,

$$QW = [Q_1, Q_2]$$
$$DJ = [0, 0]$$

Notice that the partitioning can be changed from initial (table 4.1) to 2nd adaptive partition (table 4.4) based on the queries that were introduced to maximize the performance. Similarly, as the query workload changes, our proposed system generates a plan to perform adaptive partitioning of the triples which increases the performance of the whole query set.

CHAPTER 5

SYSTEM ARCHITECTURE AND IMPLEMENTATION

We propose a system which stores an RDF graph by partitioning it into sub-graphs and distributing it. It changes the partition by transferring a chunk of RDF triples from one partition to another based on the query workload. In this chapter, we describe the system architecture of our system RePart. We discuss in details about its various component and their functions.

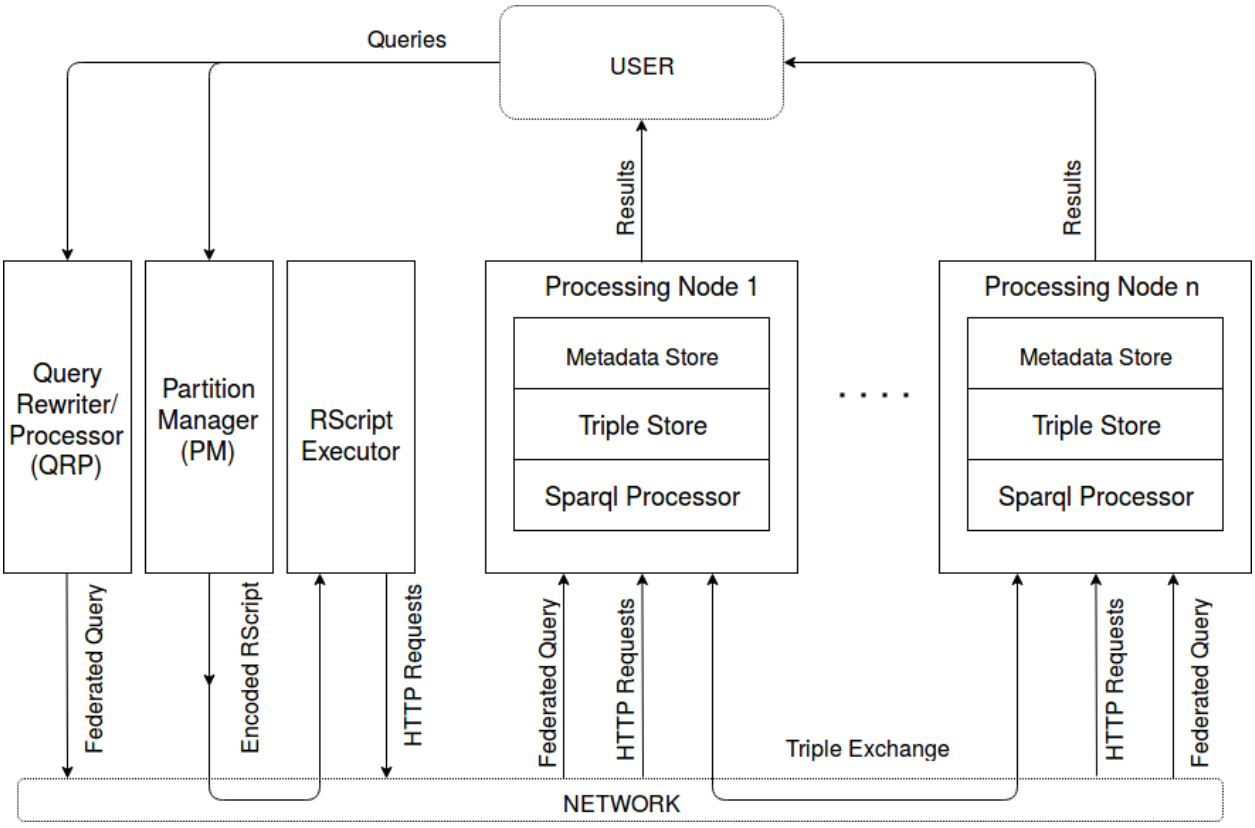

Figure. 5.1: System architecture of RePart

RePart is deployed on the shared-nothing cluster of machines. Each computing node in the system is called a Processing Node as shown in Figure 5.1. In the context of distributed RDF stores, the triples of graph G have to be assigned to different processing nodes. Partition Manager completes this task. It divides G into n partitions and sends the individual partitions to the respective processing nodes. QRP (Query Rewriter/Processor) rewrites the incoming query into a federated query according to the current partition of the triples (Section 5.2). The Federated query is then sent to the appropriate node where it is executed. The node where the query is executed is called Primary Processing Node (PPN) for that particular query. Partition manager monitors the query workload and if necessary, initiates the repartitioning of the triples. PM generates an RScript (Section 5.3), which is a series of operations that need to be executed to (i) initiate the process of exchanging triples from one processing node to another and (ii) also to update the metadata of each processing node. This RScript is sent to Repartitioning Script executor that executes the operations (Sections 5.4) mentioned in the RScript.

For the rest of the thesis, the finite set of processing nodes will be denoted as PN, and the ith node will be indicated as PN_i.

All the nodes in the cluster are exposed to the network by a RESTful interface. Each node acts as a RESTful Web application and the communication among the nodes is achieved using the HTTP Protocol by accessing the resources which are described below. Each resource serves a different purpose in RePart.

- **Triple resource** represents RDF data (triples) in any node. This resource can be accessed using the URI[1] $http://PN_i/rdfpartition/webapi/triples$.
- **Metadata resource** represents the data that provides information about the type of triples stored in any node. It is accessed with the URI $http://PN_i/rdfpartition/webapi/metadata$.

[1] PN_i is the IP address of ith processing node

- **RScript resource** represents the RScript which is used to trigger the repartition of data among the nodes of the cluster. Further details are mentioned in the coming sections. It is accessed with the URI $http://PN_i/rdfpartition/webapi/rscript$.

Various operations are defined which are executed by RePart to provide specific functionalities. These operations, which are listed below, provide an overview of ReParts capability.

- **TRANSFER TRIPLES** - transfer triples from one processing node to another based on given constraints. This operation is a combination of various other operations listed below.
  - **GET TRIPLES** - obtain triples from a processing node.
  - **UPLOAD TRIPLES** - upload triples to a processing node.
  - **DELETE TRIPLES** - delete triples from a processing node.
- **GET METADATA** - request metadata from a processing node.
- **UPDATE METADATA** - update the metadata of a processing node.
- **SEND RScript** - send a RScript to RScript executor.
- **EXECUTE RScript** - RScript executor executes the RScript.

As mentioned earlier, this thesis is a part of a bigger project. The primary contribution of this thesis is to provide an underling distributed system which is able to transfer the triples within the nodes of the cluster (adaptive partitioning) and update metadata based on the incoming instructions. PM, with the help of the QRP are responsible for generating the instructions (RScript) that initiate the re-partitioning. We have explained the functions of these components, however for the evaluations, their working is simulated manually. For instance, the RScript is generated manually by observing the query workload and given as an input to RScript Executor. Therefore, this thesis provides a full implementation of the components RScript executor, triple store, and Metadata store. All the functions that QRP

and PM performs are realized manually to simulate the entire system RePart and observe the performance gain. In the following sections, we describe the functionalities of each component of RePart's in detail.

5.1 Processing Node

Processing node comprises of three major components, triple store, metadata store and **Sparql** query processor. Triple store acts as a storage for RDF triples. SPARQL query processor execute the queries and metadata store keeps track of information about the type of triples present in any processing node at a given time.

5.1.1 Triple Store and Sparql Query Processor

Triple store acts as a database for RDF triples. Each processing node has a triple store where corresponding partition P_i of RDF graph G is stored. On the other hand, SPARQL query processor can execute SPARQL queries over the triples which are stored in the triple store. It is also possible to run federated SPARQL queries which provide an ability to collect the results from multiple processing nodes. For RePart's evaluation, we have used OpenLink Virtuoso (section 2.5) that serves a dual purpose. It acts as a triple store, as well as the SPARQL query processor. We make sure that any RDF store that is used as a triple store in RePart is consistent with the ACID properties.

Triple resource represents RDF triples in each processing node. RDF triples can be manipulated by executing various HTTP methods on the triple resource with the URI: $http : //PN_i/rdfpartition/webapi/triples$. To obtain the triples from the triple store, an HTTP GET request is made on the mentioned URI which corresponds to the operation **GET TRIPLE**. To specify the type of triples that have to be obtained, following query parameters can be used while using HTTP GET :

- **metadata** - triples can be requested based on the metadata. Metadata provides information about the type of triples that have to be obtained. It is itself is a resource and is explained in Section 5.1.2. Example of such URI is:

- To get all the triples stored in a processing node, no query parameter has to be used. Example of URI for such request is $http://PN_i/rdfpartition/webapi/triples$.

The operation **DELETE TRIPLES** is performed by executing HTTP delete request on the triple resource. Similar to the GET request, metadata is used as a query parameters to denote which triples have to be removed from the triple store.

To load the triples into the triple store, Virtuoso's ISQL bulk load[2] is used. In RePart, ISQL bulk load performs the operation **UPLOAD TRIPLES**.

5.1.2 Metadata Store

Metadata store maintains the information about the type of triples present in the triple store. Metadata is the data that describes the RDF data in a Processing Node. For exact, informative and concise representation of the type of triples in the triple store, we have designed a *RDF-Metadata Notation*. The PM and QRP use metadata for different purposes. Repartitioning of RDF data is triggered by the partition manager after searching for patterns in the metadata that reflects the need for shuffling of triple for better query performance. Metadata, of each processing node that was involved in data repartitioning is updated to show the current state of triples in the triple store (Section 5.5). Furthermore, the information presented by metadata helps the query analyzer to compute a cost-effective query execution plan. By analyzing the metadata, QRP rewrites the incoming query and sends it to the most suitable processing node where the query is executed, which increases the query performance by reducing the distributed joins while query execution. *Metadata resource* represents the

[2]https://virtuoso.openlinksw.com/dataspace/doc/dav/wiki/Main/VirtBulkRDFLoader

metadata of any processing node. PM and QRP obtains the metadata by executing an HTTP GET request on the URI: $http://PN_i/rdfpartition/webapi/metadata$. This corresponds to the task **GET METADATA**.

RDF-Metadata Notation (RDF-MN)

We have developed a *Metadata Notation* that uses predicate values to describe various query patterns that could be answered by a given processing node. Following are the various techniques in which the patterns are described using RDF-MN:

- **Subject-Subject Join (SSJ)**: It represents all the triples with the predicates that share a same subject or join on same subject. For example, as shown in Figure 5.2, $PI, P2, P3, P4$ share A as their subject. Similarly, $P5, P6, P7$ also share the same subject F. With such representation of data, it is shown that all the triples that have $P1, P2, P3$ or $P4$ as predicates and A as a subject exists in the triple store. This information gives indicates that this triples store can answer a star query involving $P1, P2, P3, P4$ predicates. Similarly, it can also answer a star query that includes $P5, P6$, and $P7$ predicates. Figure 5.2 provides a graphical representation of the star queries.

- **Object-Subject Join (OSJ)**: It represents all the triples with the predicates that are involved in the object-subject join. As shown in Figure 5.3, J is an object for $P8$, whereas for $P9$ it is a subject. All the triples where predicates $P8$ and $P9$ are connected through J exists in the triples store. Similarly, predicates $P10$ and $P11$ are also represented. Such data representation is beneficial when a query includes an object-subject join. This kind of a join is also called an elbow join. It helps to identify if a query can be answered by a processing node after analyzing the metadata.

- **Object-Object Join (OOJ)**: It represents all the triples with given predicates that points to the same object or join on the same object. For example, Figure 5.4 provides a

graphical representation of all the triples that have a predicates values of $P12$,$P13$,$P14$, or $P15$ share the same object O. Similarly, triples with the predicate values of $P16$, $P17$, or 18 points to the same object T. This kind of a representation helps to detect if a node can answer a query pattern that involves joins on the object.

- **All the triples with a given Predicate (P)**: This representation is used when all the triples with a given predicate are present in a single node. For example, if all the triples that has $P19$ as their predicate value are stored in a single processing node, then $P19$ will be represented under this section (Figure 5.5).

- **All the triples with a given Predicate-Object (PO)**: It is sometimes required to divide the triples with not only a given predicate value but also on a combination of a predicate and an object value. This is done to reduce the replication of triples in multiple processing nodes. Therefore, if the triples are further divided based on the object value, then they are represented using this section. For example, consider Figure 5.6 that graphically show all the triples that have a predicate value of $P20$ and an object value as *obj*.

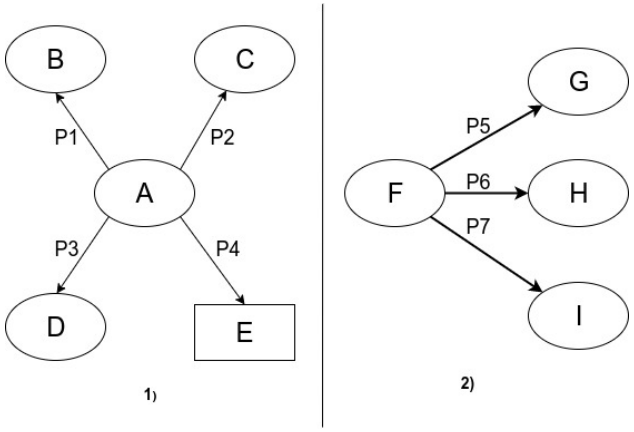

Figure. 5.2: Graphical representation of a Subject-Subject Join (Star Pattern)

As mentioned before, one of these metadata representations is used to determine what kind of triples have to be transferred between two processing node. Suppose, some triples that are represented using SSJ are moved from PN_1 to PN_2, triples with predicates that do not

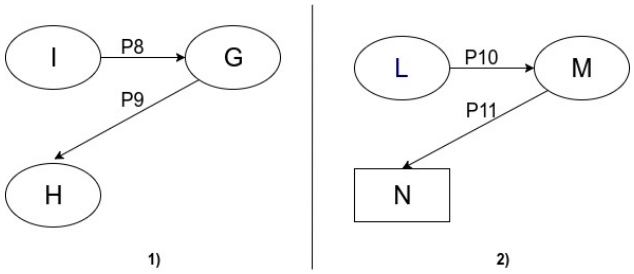

Figure. 5.3: Graphical representation of a Object-Subject join (Elbow join)

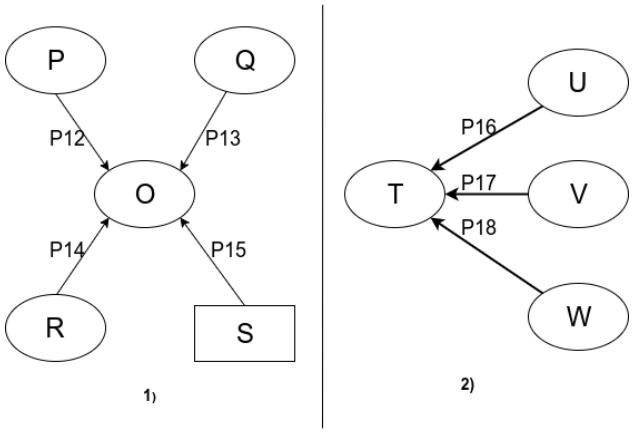

Figure. 5.4: Graphical representation of a Object-Object join

share the same subject are left behind in PN_1. It is important to represent these triples too, or otherwise, there is no way to know the location of the triples that are left behind. Therefore, there are three more representations that are included in RDF-MN that act as a complement of SSJ, OSJ or OOJ.

- **Subject-Subject Join Complement (SSJ-C)**: It represents all the triples with the given predicates that do not share the subjects. Therefore, it acts as a complement of SSJ. For example, consider Figure 5.2. If the predicates $P1$, $P2$, $P3$, or $P4$ are mentioned in the SSJ, it means that all the triples with these predicates that have a common subject, exists in the same node. However, when these predicates are men-

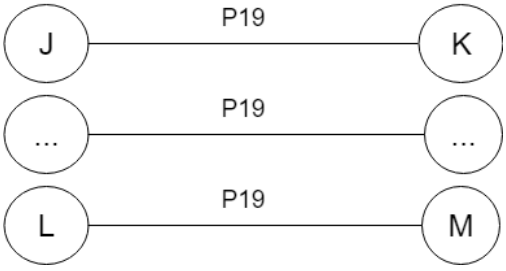

Figure. 5.5: Graphical representation of all the triples with the predicate P19

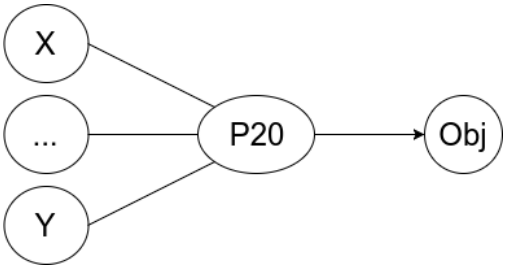

Figure. 5.6: Graphical representation of all the triples with given Predicate-Object

tioned in SSJ-C, it means that the particular processing node contains all the triples that have $P1$, $P2$, $P3$, or $P4$ as their predicate values and do not share the same subject.

- **Object-Subject Join Complement (OSJ-C)**: It represents all the triples with the given predicates that are not involved in the object-subject join. Therefore, it acts as a complement of OSJ. For example, consider Figure 5.3. If the predicates $P8$ and $P9$ are mentioned in the OSJ, it means that all the triples with these predicates that are involved in the object-subject join, exists in the same node. However, when these predicates are mentioned in OSJ-C, it means that the particular processing node contains all the triples that have $P8$ and $P9$ as their predicate values and do not have an object-subject join.

- **Object-Object Join Complement (OOJ-C)**: It represents all the triples with the given predicates that do not share the objects. Therefore, it acts as a complement of OOJ. For example, consider Figure 5.4. If the predicates $P12$, $P13$, $P14$, or $P15$ are mentioned under OOJ, it means that all the triples with these predicates that have a common subject, exists in the same node. However, when these predicates are mentioned in OOJ-C, it means that the particular processing node contains all the triples that have $P12$, $P13$, $P14$, or $P15$ as their predicate values and do not share the same subject.

Triple Metadata Notation Encoding In JSON

The information represented by RDF-MN is encoded in JSON to make it easier to parse and transfer between various components of RePart. Figure 5.7 shows the encoded version in JSON. Below are the details about every key-value pair in the metadata JSON object.

```
{
  "nodeIndex": 1,
  "SSJ": [ [ "P1","P2","P3","P4" ], [ "P5","P6","P7" ] ],
  "OSJ": [ [ "P8","P9" ], [ "P10","P11" ] ],
  "OOJ": [ [ "P12","P13","P14","P15" ], [ "P16","P17","P18" ] ],
  "P": [ "P19" ],
  "PO": [ "P20,Obj" ],
  "SSJ-C": [],
  "OSJ-C": [],
  "OOJ-C": []
}
```

Figure. 5.7: RDF-MN encoding in JSON

- **nodeIndex**: The value of nodeIndex is an integer, and it represents the index of the processing node for which the triples are described.

- **SSJ**: The value of SSJ is a 2-D array. Predicates that share the same subject are grouped together and it represents that all the triples with grouped predicates that share the same subject are present in this node.
- **OSJ**:The value of OSJ is also a 2-D array. It represents that all the triples with grouped predicates that are involved in an object-subject join are present in this node. Note that OSJ only groups two predicates at a time as shown in the encoded version of metadata.
- **OOJ**: The value of SSJ is also a 2-D array. Predicates that share the same object are grouped together, and it represents that all the triples with grouped predicates that share the same object are present in this node.
- **P**: The value of P is an array which contains the predicate values. The values are included in this array only if every triple of a given predicate is present in this processing node.
- **PO**: The value of PO is also an array. The elements of the array are a combination of predicate and object value separated using ",". For each value of the array, it is true that all the triples with the given predicate and object values exist in this node.
- **SSJ-C, OOJ-C** and **OSJ-C**: The representation of SSJ-C, OSJ-C, and OOJ-C is the same as their counterpart SSJ. OSJ and OOJ respectively, which is a 2-d array. However, currently it is empty as there are no triples that can be represented by them.

The Process Of Updating The Metadata

To understand the process of how the metadata is updated when the triples are transferred between the nodes, consider the initial partitioning shown in Table 4.1. The metadata for this initial partitioning is shown in Figure 5.8. It can be seen that every representation is

empty except P, because the triples are initially divided based on the predicate values. As the triples are shuffled between the nodes, the metadata will be updated.

For this example, assume that all the triples that have the predicate value *rdf:type* can also be divided based on only two object values, *ub:Student* and *ub:Professor*. The triples will be shuffled in three steps and each time the metadata has to be updated.

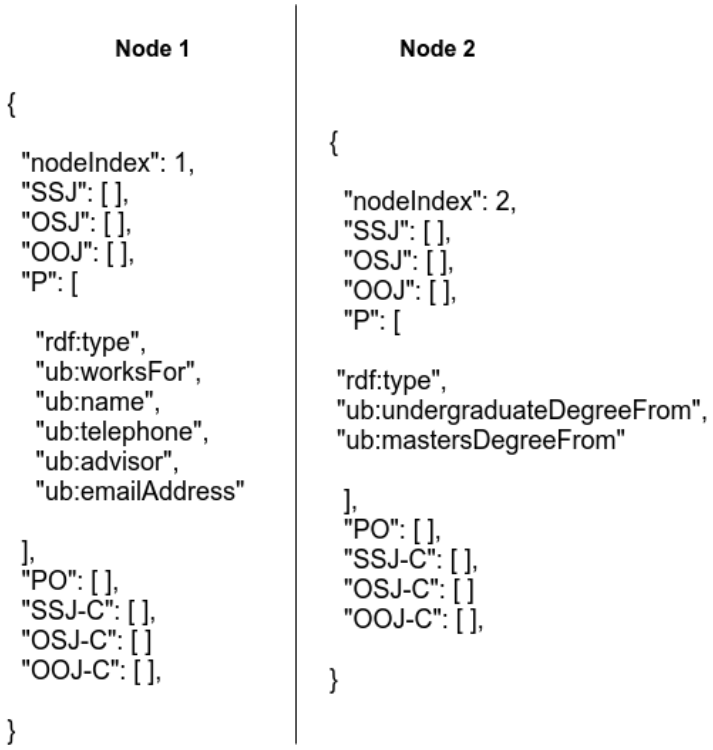

Figure. 5.8: Metadata for the initial Partitioning

- STEP 1: Move all the triples with the predicate values *ub:name* and *ub:worksFor* that share the same subject from PN_1 to PN_2. The updated metadata for both the nodes is shown in Figure 5.9. It can be seen from the metadata of PN_1 that the triples with predicate values *ub:name* and *ub:workFor* that do not share the same subject are represented under SSJ-C. Similarly, the triples that were moved to the PN_2 are represented under SSJ. Now that PN_1 no longer has all the triples with the predicate values *ub:name* and *ub:worksFor*, the entries are removed from P.

- STEP 2: Move all the triples with the predicate values *ub:undergraduateDegreeFrom* and *ub:emailAddress* that share the same object from PN_2 to PN_1. The updated metadata for both the nodes is shown in Figure 5.10. It can be seen from the metadata of PN_2 that the triples with the predicate values *ub:undergraduateDegreeFrom* and *ub:emailAddress* that do not share the same object are represented under OOJ-C. Similarly, the triples that were moved to the PN_1 are represented under OOJ. Now that PN_2 no longer has all the triples with the predicate values *ub:undergraduateDegreeFrom* and *ub:emailAddress*, the entries are removed from P.

- STEP 3: Notice that all the triples with the predicate value *rdf:type* are replicated in both the processing nodes. However, it might not be necessary that all the triples of *rdf:type* are used during the processing of the queries by both the nodes. Therefore, triples can be further divided using both, the predicate and the object values. Let us delete all the *rdf:type* triples from the PN_2 and move all the triples with the predicate rdf:type and object ub:Professor from PN_1 to PN_2. The updated metadata after moving triples is shown in Figure 5.11. As the triples are now distributed using the combination of predicate-object value, there are new entries in the PO section. Now that both PN_1 and PN_2 no longer has all the triples with the predicate values *rdf:type* , the entries are removed from P.

5.2 Query Rewriter/Processor (QRP)

QRP is a combination of Query rewriter and Query Processor. It receives a query q sent by the user. It then sends an HTTP Get request on the metadata resource of each processing node to gather the information about the triples that each processing node contains. Based on this knowledge and the query pattern of q, QRP decides the most suitable node that can execute the query with minimum communication cost. This node is called as Primary Processing Node (PPN). Query q is rewritten into a federated query fq by query rewriter if

Node 1


```
{
 "nodeIndex": 1,
 "SSJ": [ ],
 "OSJ": [ ],
 "OOJ": [ ],
 "P": [
  "rdf:type",
  "ub:telephone",
  "ub:advisor",
  "ub:emailAddress"
 ],
 "PO": [ ],
 "SSJ-C": [
  [
   "ub:worksFor",
   "ub:name"
  ]
 ],
 "OSJ-C": [ ],
 "OOJ-C": [ ]
}
```


Node 2


```
{
 "nodeIndex": 2,
 "SSJ": [
  [
   "ub:worksFor",
   "ub:name"
  ]
 ],
 "OSJ": [ ],
 "OOJ": [ ],
 "P": [
  "rdf:type",
  "ub:undergraduateDegreeFrom",
  "ub:mastersDegreeFrom"
 ],
 "PO": [ ],
 "SSJ-C": [ ],
 "OOJ-C": [ ],
 "OSJ-C": [ ]
}
```


Figure. 5.9: STEP 1 Metadata

more then one processing node has to be reached to answer the query. Query processor then sends fq to PPN for the execution. Finally, results are combined at PPN and sent to the user. This process is shown in Section 4.3.

5.3 Partition Manager (PM)

As stated before, data placement strategies in the RDF stores plays a very crucial role with respect to query performance. Partition Manager has two primary responsibilities of (i) uploading the initial partitions (triples) to the respective processing nodes and (ii) to perform Repartitioning the RDF data that is distributed among the nodes of a cluster.

Node 1

```
{
 "nodeIndex": 1,  "SSJ": [ ],  "OSJ": [ ],
 "OOJ": [
  [
   "ub:undergraduateDegreeFrom",
   "ub:mastersDegreeFrom"
  ]
 ],
 "P": [
  "rdf:type",
  "ub:telephone",
  "ub:advisor",
  "ub:emailAddress"
 ],
 "PO": [ ],
 "SSJ-C": [
  [
   "ub:name",
   "ub:workdFor"
  ]
 ],
 "OSJ-C": [ ],  "OOJ-C": [ ]
}
```

Node 2

```
{
 "nodeIndex": 2,
 "SSJ": [
  [
   "ub:name"
   "ub:worksFor",
  ]
 ],
"OSJ": [ ], "OOJ": [ ],
"P": [
 "rdf:type"
],
"PO": [ ],  "SSJ-C": [ ],
"OOJ-C": [
  [
   "ub:undergraduateDegreeFrom",
   "ub:mastersDegreeFrom"
  ]
 ],
"OSJ-C": [ ]
}
```

Figure. 5.10: STEP 2 Metadata

Partition manager maintains the Query Workload (QW) which is the set of queries that have to be executed by RePart. Any new incoming query is added to the QW. The initial partition IP_i is the division of the large RDF graph G that has to be inserted in the triple store of PN_i. There are two ways to generate the initial partitions. If a user already has a QW along with the G, both can be given as an input to RePart and PM then analyzes the QW to create the initial partitions. In this case, the data is divided in a way to reduce inter-communication between nodes when the queries are executed and also to increase the data locality. As the set of queries are already known, it is possible to divide the data in a way that involves minimum number of nodes while executing a query. On the other hand, if the user does not have queries to assist the generation of initial partitions, they can select

Node 1


```
{
 "nodeIndex": 1, "SSJ": [ ], "OSJ": [ ],
 "OOJ": [
  [
   "ub:undergraduateDegreeFrom",
   "ub:mastersDegreeFrom"
  ]
 ],
 "P": [
  "ub:telephone",
  "ub:advisor",
  "ub:emailAddress"
 ],
 "PO": [
  "rdf:type,ub:Student"
 ],
 "SSJ-C": [
  [
   "ub:name",
   "ub:workdFor"
  ]
 ],
 "OSJ-C": [ ], "OOJ-C": [ ]
}
```


Node 2


```
{
 "nodeIndex": 2,
 "SSJ": [
  [
   "ub:name",
   "ub:worksFor"
  ]
 ],
 "OSJ": [ ], "OOJ": [ ],
 "P": [
  "ub:undergraduateDegreeFrom",
  "ub:mastersDegreeFrom"
 ],
 "PO": [
  "rdf:type,ub:Professor"
 ],
 "SSJ-C": [ ],
 "OOJ-C": [
  [
   "ub:undergraduateDegreeFrom",
   "ub:mastersDegreeFrom"
  ]
 ],
 "OSJ-C": [ ]
}
```


Figure. 5.11: STEP 3 Metadata

numerous ways to partition G, for example, predicate-based partitioning or subject-hash partitioning.

Partition Manager is also responsible for repartitioning the RDF graph that is distributed among the nodes of a cluster by analyzing the QW. Repartitioning refers to the migration of RDF triples from one processing node to another which corresponds to the operation **TRANSFER TRIPLES**. PM generates a list of operations, which is collectively called as a *Repartitioning Script (RScript)*. In RePart, RScript encodes the information to perform two operations: TRANSFER TRIPLES and **U**PDATE METADATA. An example of such RScript is shown in Figure 5.12. The operation 1 denotes that triples that have to be transferred from the source node PN_1 to the destination node PN_2. Metadata represented

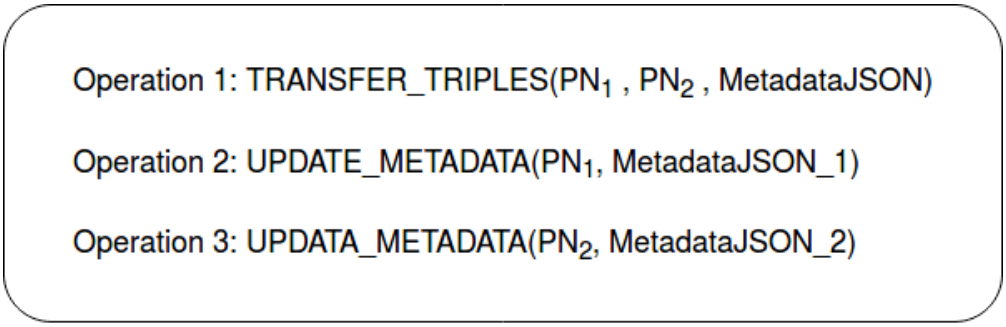

Figure. 5.12: List of operations in an RScript

by $MetadataJSON$ describes what type of triples has to be transferred. It is important to notice that the operation UPDATE METADATA is usually followed by TRANSFER TRIPLES as the content of the triples store is changed after the shuffling of the triples. Therefore, the operation 2 and 3 depicts the need to update the metadata of PN_1 and PN_2 respectively with the newly computed $MetadataJSON\_1$ and $MetadataJSON\_2$. All the metadata mentioned in the RScript is encoded in JSON (Section 5.1.2).

As discussed before, the RScript is a resource and it can be accessed by the URI $http : //PN_i/rdfpartition/webapi/rscript$. This resource is part of the component Repartitioning Script Executor. The generated RScript is encoded into JSON and is sent to Repartitioning script executor as a payload of the HTTP PUT request on the mentioned URI. This process corresponds to the operation **SEND RScript**. Repartitioning script executor receives the RScript and executes all the operations in a sequential order.

5.4 Repartitioning Script Executor (RSE)

The primary purpose of Repartitioning Script Executor (RSE) is to execute the RScript which is sent by the partition manager. The process where RSE execute the Rscript corresponds to the operation **EXECUTE RSCRIPT**. Figure 5.13 shows a JSON encoded

```
[
  {
    "operation": "TRANSFER TRIPLES",
    "destinationNodeIndex": 2,
    "sourceNodeIndex": 1,
    "metadata": "metadataJSON"
  },
  {
    "operation": "UPDATE METADATA",
    "nodeIndex": 1,
    "metadata": "MetadaJSON_1"
  },
  {
    "operation": "UPDATE METADATA",
    "nodeIndex": 2,
    "metadata": "MetadaJSON_2"
  }
]
```


Figure. 5.13: RScript encoded in JSON

version of the RScript shown in Figure 5.12. In Figure 5.13, it can be seen that the each JSON object denotes an operation, and a collection of these objects is a RScript.

The value associated with the key *operation* helps RSE to determine the type of operation it has to execute. The values of the key *sourceNodeIndex* and *destinationNodeIndex*, which are integers, identifies the source node and destination node respectively. Then, RSE generates a URI to access the triple resource of a source node and add metadataJSON as its query parameter. For example, the generated URI with PN_1 as a source node looks like $http://PN_1/rdfpartition/webapi/triples?metadata = metadataJSON$. RSE then sends an HTTP PUT request on the triple resource of a destination node. Here, HTTP PUT is used because the content of the triple store of desination node has to be updated. In the payload of this request, the generated URI is sent. In this way, the RSE directs the destination node PN_2 to send an HTTP GET request on the generated URI that was sent as a payload. This process triggers the transfer of triples from PN_1 to PN_2 (Section 5.5.1).

To execute the operation UPDATE METADATA, RSE first identifies the processing node whose metadata has to be updated. Then, it sends an HTTP PUT request on metadata resource of the mentioned node. In the payload, the metadata is sent which overwrites the already existing metadata on the specified processing node. For example, in operation 2, RSE will send an HTTP PUT request on metadata resource of PN_1. The request will be sent on the URI $http://PN_1/rdfpartition/webapi/metadata$ with $metadataJSON\_1$ as a payload. In the next section, we discuss the operations TRANSFER TRIPLE and UPDATE METADATA in detail.

5.5 Executing RScript

In this section, we discuss the repartitioning of the RDF data using RePart. As mentioned before, repartitioning refers to the transfering of the triples among the various partitions. This is done to reduce the exchange of intermediate results between the nodes while executing the queries that involve distributed joins. Repartitioning of the data includes two operations, **TRANSFER TRIPLES** and **UPDATE METADATA**. TRANSFER TRIPLES triggers the exchange of data from one node to another. Due to this transfer, the content of the participating nodes is changed, and their metadata also has to be updated. Therefore, the operation METADATA UPDATE always follows the operation TRANSFER TRIPLES. As discussed in the previous sections, RSE receives an RScript that contains the information about all the operations that have to be executed. Then, RSE sequentially executes the operations.

Consider the RScript which is shown in Figure 5.13. It is given as an input to the RSE. There are three operations mentioned in the RScript. We will first focus on the first operation which is TRANSFER TRIPLES and discuss all the steps related to it.

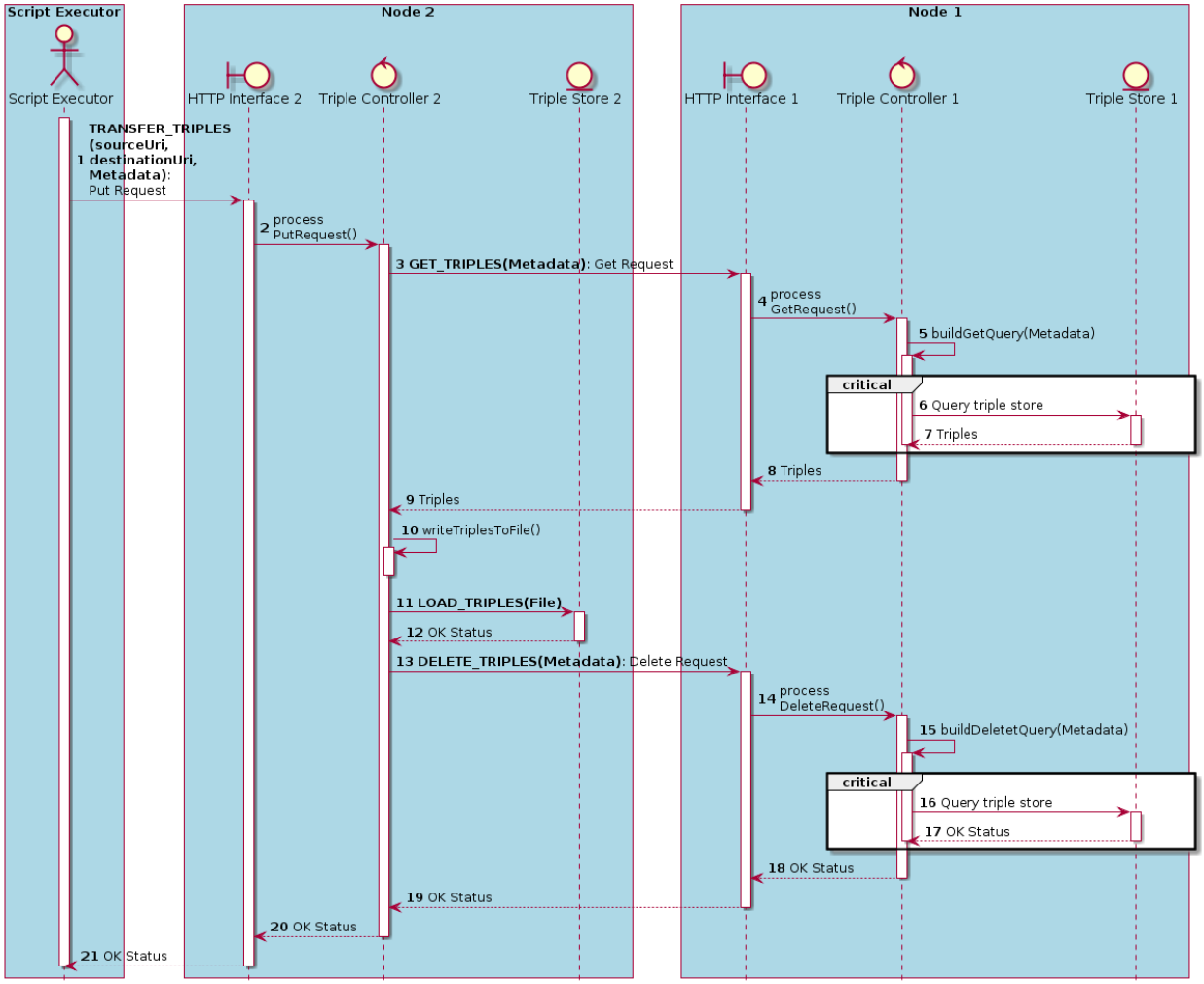

Figure. 5.14: Sequence Diagram for the operation TRANSFER TRIPLES

5.5.1 TRANSFER TRIPLES

To demonstrate the steps involved in the execution of the operation TRANSFER TRIPLES, let us consider transferring all the triples that have $ub : emailAddress$ as a predicate from PN_1 to PN_2 (Table 4.1). In the RScript JSON encoding, the destination and the source nodes are mentioned as PN_2 and PN_1 respectively. The metadata JSON object, shown in Figure 5.15 depicts the information about the type of triples that has to be transferred. The

predicate value $ub : emailAddress$ is mentioned under key P. This means that all the triples that have predicate value as ub:emailAddress has to be transferred from the source node to the destination node.

```
{
  "operation": "TRANSFER TRIPLES",
  "destinationNodeIndex": 2,
  "sourceNodeIndex": 1,
  "metadata": {
                "P": [ "ub:emailAddress" ]
              }
}
```

Figure. 5.15: TRANSFER TRIPLES JSON object

The sequence diagram (Figure 5.14) depicts the transfer of triples from PN_1 to PN_2, which is initiated by the script executor. As discussed earlier, TRANSFER TRIPLES is a combination of three other operations: GET TRIPLES, UPLOAD TRIPLES and DELETE TRIPLES. The messages numbered from 2 to 10 shows the GET TRIPLES operation, 11 and 12 are related to the LOAD TRIPLES, while the rest of the message describes the DELETE TRIPLES operation.

The sequence diagram describes an Entity-Control-Boundary system which consists of Script executor, PN_1 and PN_2. Each processing node includes three components:

- **HTTP Interface** is a boundary object that represents the HTTP endpoints through which various components of a processing node communicate with each other.
- **Triple Store** is an entity object that represents the system data (Triples).
- **Triple Controller** is a controller object that represents functionalities to manipulate the triples in the triple store.

GET TRIPLES

MESSAGE 1. RSE makes an HTTP PUT request on the triple resource of PN_2, and in the payload, it sends a URI which encodes the information about what kind of triples has to be requested from PN_1 .

MESSAGE 2. PN_2 processes the HTTP PUT request.

MESSAGE 3. PN_2 makes an HTTP GET request on the URI that it received in the payload of the PUT request. The metadata JSON object is attached as a query parameter in the URI. In this way, PN_2 askes PN_1 for the triples. Metadata denotes what kind of triples PN_2 is requesting.

MESSAGE 4. PN_1 receives and process the HTTP GET request sent by PN_2.

MESSAGE 5. The triple controller builds a SPARQL query using the information in the metadata JSON object to get the required triples from the triple store.

MESSAGE 6 and 7. The triple controller then executes the SPARQL query to get the triples from the PN_1 triple store. This process is enclosed under a "critical box" because no other query can be executed on this triple store at the same time, otherwise the data will be manipulated. The requested triples are sent back to the controller.

MESSAGE 8 and 9. In response to the HTTP GET request, PN_1 sends the requested triples to PN_2. The triples are sent as a stream of data.

LOAD TRIPLES

Ones the triples are received by PN_2, it has to be uploaded to the triple store.

MESSAGE 10. PN_2 receives the streaming data and writes it to a file in chunks. The entire data is never kept in the main memory at the same time.

MESSAGE 11 and 12. The triple controller then executes a shell script to upload the triples stored in the file to the triple store of PN_2.

DELETE TRIPLES

Currently, both, PN_1 and PN_2 had the same set of triples in their triple store. Therefore, the triples has to be deleted from the triple store of PN_1.

MESSAGE 13. PN_2 sends an HTTP DELETE request on the URI that is received in the payload of PUT method. The metadata JSON object encodes the information about the triples that have to be deleted. This directs PN_1 to delete the required triples.

MESSAGE 14. PN_1 receives the HTTP DELETE request and process it.

MESSAGE 15. The triple controller builds a SPARQL query using the information in the metadata JSON object to delete the required triples from the triple store.

MESSAGE 16 and 17. The triple controller then executes the SPARQL query to delete the triples from the PN_1 triple store. This process is enclosed under a "critical box" because no other query can be executed on this triple store at the same time, otherwise the data will be manipulated. The status is sent back.

MESSAGE 18 and 19. If the triples are deleted successfully, the "OK" status is sent back to PN_2.

MESSAGE 20 and 21. PN_2 sends an "OK" status to the script executor if all the operations are executed successfully. This also confirms that the required triples are successfully transferred from PN_1 to PN_2.

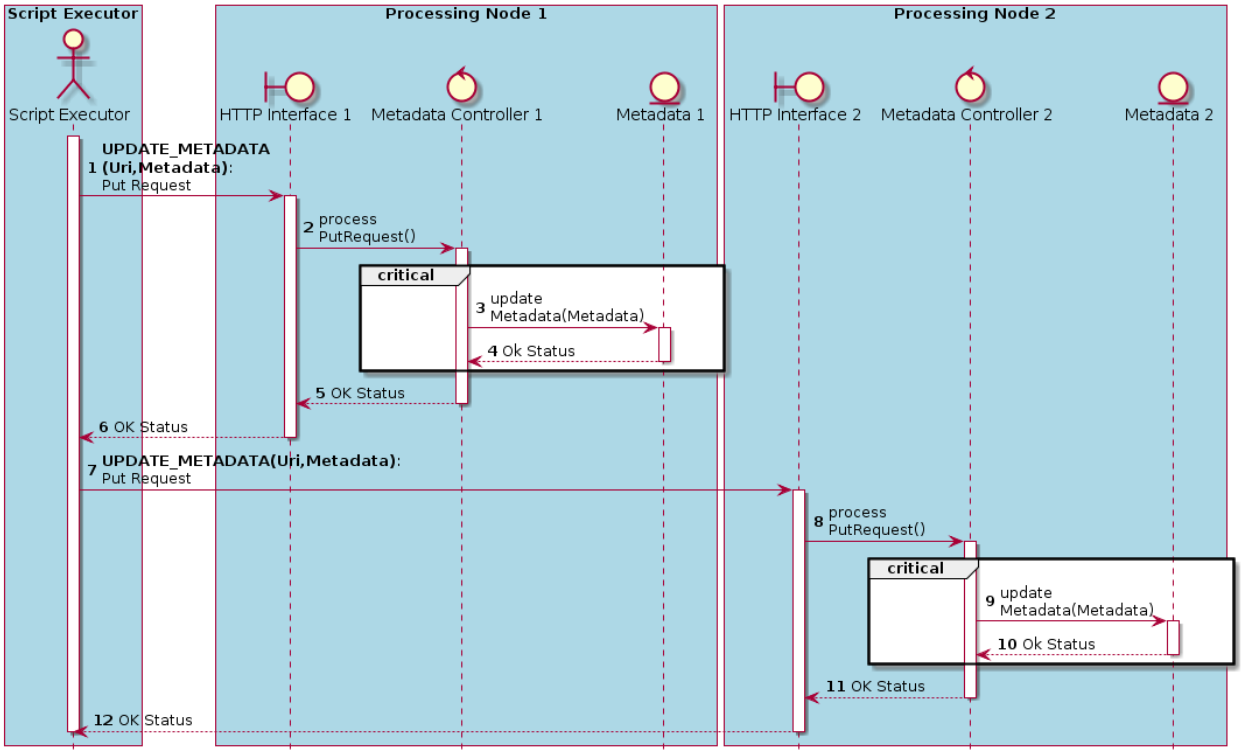

Figure. 5.16: Sequence Diagram for the operation UPDATE METADATA

5.5.2 UPDATE METADATA

When the triples are successfully transferred, the RSE receives an OK status from PN_2. After this, RSE moves on to execute the next operations in the queue. As discussed earlier, operation UPDATE METADATA always follows the operation TRANSFER TRIPLES because after the shuffling of triples is completed, the content of the triple stores of the participating node is changed. Therefore, the metadata has to be updated to reflect the correct content of the triple stores.

In the RScript shown in Figure 5.13, next two operations are UPDATE METADATA. The sequence diagram (Figure 5.16) depicts these operations. The sequence diagram describes an Entity-Control-Boundary system which consists of Script executor, PN_1 and PN_2. Each pro-

cessing node includes three components: HTTP Interface, Metadata Controller, and Metadata Entity.

MESSAGE 1. RSE makes an HTTP PUT request on the metadata resource of PN_1, and in the payload, it sends the new metadata JSON object which encodes the information about what kind of triples exists in the triple store.

MESSAGE 2. PN_1 processes the HTTP PUT request.

MESSAGE 3 and 4. The triple controller updates the metadata JSON object with the new JSON object that was sent in the payload of the PUT request. This process is enclosed under a "critical box".

MESSAGE 4 and 5. If the metadata is successfully updated, RSE receives an OK status and it executes the next operation.

CHAPTER 6

EXPERIMENTS AND EVALUATIONS

RePart is a distributed SPARQL query processing system that performs an adaptive partitioning of triples among the nodes of the cluster according to the query workload. As discussed before, the thesis's major contribution is to implement the system within RePart which can shuffle the triples among the various partitions based on RScript.

For evaluation, we present a study on the impact of a query adaptive partitioning of RDF triples. LUBM [45] and BSBM 1.0 [46] are used for Benchmarking Semantic Web knowledge-based systems. We use them to generate synthetic data (triples). They also provide a set of queries that can be executed on the generated data. To perform the experiments, we used four processing nodes, which means that the triples had to be divided into four partitions. We used OpenLink Virtuoso 7.0 [32] as a triple store and SPARQL query processor. Table 6.1 shows the SPARQL endpoints to access the triple store of the processing nodes. The following steps were undertaken for both the benchmarks to show the effect of repartitioning the data when the query workload changes to increase the performance of the queries. We used our implemented system to rearrange the data among the various partitions.

Table. 6.1: SPARQL Endpoints to acces the triple store of the processing nodes

Processing Node	**SPARQL Endpoint**
PN_1	http://172.17.151.171:8890/sparql/
PN_2	http://172.17.151.170:8890/sparql/
PN_3	http://172.17.151.172:8890/sparql/
PN_4	http://172.17.151.173:8890/sparql/

- **STEP 1:** Generated the synthetic data (triples) using LUBM and BSBM.
- **STEP 2:** The entire data (triples) was divided into four partitions. For the experiments, we divided the triples based on their predicate value. For example, if a predicate is assigned to PN_1, all the triples with that predicate value will be stored in the triple store of PN_1. This is called a **Predicate Based Partitioning**. We obtained all the unique predicates in the dataset and randomly assigned them to a processing node while balancing the load among the partitions. We call this the initial partition. We did not use query-workload to generate this partition initially. However, we repartitioned the data after analyzing the workload and also adapted to the changes that occur in the workload.
- **STEM 3:** The queries were added to the query workload.
- **STEP 4:** The queries in the query workload were re-written into federated queries according to the initial partition. They were then executed using Virtuoso. The query performance was measured using two criterions, (i) no. of distributed join the query introduce and (ii) the time taken by the query to execute. This is done for each query in the query workload that has at least one distributed join.
- **STEP 5:** All the queries were analyzed, and an RScript was generated manually which encodes the information for optimally repartitioning the triples to reduce the average distributed joins count in the query workload. This RScript was giving as an input to the RSE and our system repartitioned the triples and updated the metadata automatically. This is called the 1^{st} Adaptive Partition. Again, the query performance was measured using two criterions, (i) no. of distributed join the query introduce and (ii) the time taken by the query to execute. We ploted a graph to compare the query results of 1^{st} Adaptive Partition to Initial Partition.
- **STEP 6:** To show that the system is capable of adapting to the query workload change, a query was removed, and a new query was added to the query workload. The

performance of all the queries in the current query workload was measured using the two criterion mention in the previous steps.

- **STEP 7:** All the queries were analyzed, and an RScript was generated manually which encodes the information for optimally repartitioning the triples to reduce the average distributed joins count in the query workload. This RScript was giving as an input to the RSE and our system repartitioned the triples and updated the metadata automatically. Let's call this the 2^{nd} Adaptive Partition. Again, the query performance was measured using two criterions, (i) no. of distributed join the query introduce and (ii) the time taken by the query to execute. We plot a graph to compare the query results of 2^{nd} Adaptive Partition to 1^{st} Adaptive Partition.
- **STEP 8:** Steps 6 and 7 were repeated one more time. The graph was plotted to compare the query performance between 2^{nd} Adaptive Partition and 3^{rd} Adaptive Partition.

6.1 BSBM Results

The Berlin SPARQL Benchmark 1.0 (BSBM) [46] is used for comparing the performance of the SPARQL systems across architectures. BSBM provides a dataset of an e-commerce use case where the vendors can offer their products, and the consumers can list their reviews about these products.

Using BSBM, we generated a total of $100,031,2$ triples and divide them across the four processing nodes. BSBM provides a query-set to test the performance of a system. We used seven queries from the query-set to evaluate our system. The queries are mentioned in the Appendix A and they are numbered similarly to the BSBM's query-set [1].

[1] `http://wifo5-03.informatik.uni-mannheim.de/bizer/berlinsparqlbenchmark/V1/spec/index.html#queriesTriple`

Table. 6.2: BSBM Initial Partition

PN_1	PN_2	PN_3	PN_4
foaf:name bsbm:rating1 bsbm:rating2 bsbm:raiting3 bsbm:raiting4 dc:title bsbm:reviewDate dc:date rdfs:subClassOf bsbm:country bsbm:productPropertyNumeric3 bsbm:productPropertyTextual2 bsbm:productPropertyTextual3 bsbm:productPropertyTextual4 bsbm:productPropertyNumeric6 bsbm:producer foaf:homepage bsbm:productPropertyTextual6 foaf:mbax_sha1sum	bsbm:productPropertyNumeric1 bsbm:productPropertyTextual1 rev:text bsbm:validTo Bsbm:vendor dc:publisher Rdfs:comment bsbm:productPropertyTextual5 bsbm:productPropertyNumeric4 bsbm:productPropertyNumeric5	rdfs:label bsbm:product bsbm:productFeature rdf:type rev:reviewer	bsbm:productPropertyNumeric2 bsbm:deliveryDays bsbm:offerWebpage bsbm:price bsbm:validFrom bsbm:reviewFor

Initial Partition

As discussed before, we use predicate based partitioning to generate the initial partition. We obtain the distinct predicate values in the BSBM dataset and assign each of them to a processing node. This division is shown in Table 6.2. All the triples with a given predicate value are stored in the triple store of the assigned processing node.

Rewriting Queries According To The Initial Partition

All the queries in the query workload are rewritten into federated queries because the data is divided across multiple SPARQL endpoints. The rewritten queries are shown in the Appendix A.1. For example, let us consider the query shown in Figure 6.1 to understand the process of re-writing the query.

PN_1 is selected the primary processing node to execute the query because it has the highest amount of data that the query uses compared to the other nodes. It can be observed that

Table. 6.3: BSBM 1^{st} Adaptive Partition

PN 1	PN 2	PN 3	PN 4
bsbm:country bsbm:price bsbm:vendor dc:publisher foaf:name bsbm:rating1 bsbm:product	bsbm:validTo rdfs:comment bsbm:productPropertyNumeric5 bsbm:rating2 bsbm:raiting3 bsbm:raiting4 rdfs:subClassOf foaf:homepage bsbm:productPropertyTextual6 foaf:mbax_sha1sum bsbm:reviewDate dc:title bsbm:reviewFor bsbm:productPropertyNumeric3 bsbm:productPropertyNumeric6 rev:reviewer	rdfs:label bsbm:productFeature rdf:type bsbm:productPropertyNumeric1 bsbm:productPropertyNumeric2 bsbm:productPropertyNumeric4 bsbm:productPropertyTextual1 bsbm:productPropertyTextual2 bsbm:productPropertyTextual3 bsbm:productPropertyTextual4 bsbm:productPropertyTextual5 bsbm:producer bsbmOfferWebPage	bsbm:deliveryDays bsbm:validFrom dc:date rev:text

running the query on PN_1 introduces nine distributed joins which are shown in bold. The goal is to reduce the number of distributed joins to increase the performance of the query.

1^{st} Adaptive Partition

After analyzing all the queries in the query workload, the triples were re-shuffled among the nodes while maintaining the load balance. The data which was accessed together was kept in a single triple store to reduce the total number of distributed joins in the queries. The new partition was called the 1^{st} adaptive partition as shown in Table 6.3. After the first adaptive partitioning, the number of distributed joins in Q7 was reduced from 9 to 7. The PPN for Q7 was changed to PN_3 from PN_1. The query was re-written according to the current partitioning as shown in Figure 6.2. For example, all the triples with the predicate value $rev : reviewer$ were moved from PN_1 to PN_3. Now that the query is executed on PN_3, the data is local to its triple store.

```
SELECT ?productLabel ?offer ?price ?vendor ?vendorTitle ?review ?revTitle?reviewer ?revName ?
rating1 ?rating2
WHERE {
SERVICE<http://172.17.151.172:8890/sparql/bsbm> {
<http://www4.wiwiss.fuberlin.de/bizer/bsbm/v01/instances/dataFromProducer1/Product15> rdfs:label ?
productLabel .}
OPTIONAL {
SERVICE<http://172.17.151.172:8890/spqrql/bsbm> {
?offer bsbm:product
<http://www4.wiwiss.fuberlin.de/bizer/bsbm/v01/instances/dataFromProducer1/Product15> . }
SERVICE<http://172.17.151.173:8890/spqrql/bsbm { ?offer bsbm:price ?price . }
SERVICE<http://172.17.151.170:8890/spqrql/bsbm { ?offer bsbm:vendor ?vendor . }
SERVICE<http://172.17.151.172:8890/spqrql/bsbm> {  ?vendor rdfs:label ?vendorTitle . }
?vendor bsbm:country <http://downlode.org/rdf/iso-3166/countries#DE> .
SERVICE<http://172.17.151.170:8890/spqrql/bsbm> { ?offer dc:publisher ?vendor .
?offer bsbm:validTo ?date . }
FILTER (?date > 2007-01-04 )}
OPTIONAL {
SERVICE<http://172.17.151.173:8890/spqrql/bsbm> {  ?review bsbm:reviewFor <http://www4.wiwiss.fu-
berlin.de/bizer/bsbm/v01/instances/dataFromProducer1/Product15> .}
SERVICE<http://172.17.151.172:8890/spqrql/bsbm> {   ?review rev:reviewer ?reviewer . }
?reviewer foaf:name ?revName .
?review dc:title ?revTitle .
OPTIONAL { ?review bsbm:rating1 ?rating1 . }
OPTIONAL { ?review bsbm:rating2 ?rating2 . }}}
```

Figure. 6.1: Re-written Federated Query7 According To The Initial Partition

Figure 6.3 shows the comparison of the running time for every query in the query workload between the Initial and 1st adaptive partition. Similarly, Figure 6.4 depicts the change in the number of distributed joins for the queries when the triples are shuffled. It can be observed that the running time of the queries is improved remarkably. This happens because the 1st adaptive partition tries to make the data local to a single node for every query. The runtime for Q5 is decreased more compared to the other queries because, after the 1^{st} adaptive partitioning, no distributed join has to be computed. The graphs attest that adapting to the query workload decreased the computational overhead and increased the average performance of the system.

2^{nd} Adaptive Partition

To show how the content of each node was adapted according to the change in queries, we changed the query workload. The Q8 is removed, and a new query shown in Figure 6.7 was added to the query workload. All the queries were analyzed again, and if possible, the triples

```
SELECT ?productLabel ?offer ?price ?vendor ?vendorTitle ?review ?review ?reviewer ?revTitle ?
rating1
from<http://localhost:8890/sparql/bsbm>
WHERE {
SERVICE<http://172.17.151.172:8890/sparql/bsbm> {  <http://www4.wiwiss.fu-
berlin.de/bizer/bsbm/v01/instances/dataFromProducer1/Product15> rdfs:label ?productLabel .}
OPTIONAL {SERVICE<http://172.17.151.172:8890/sparql/bsbm> {
?offer bsbm:product <http://www4.wiwiss.fu-
berlin.de/bizer/bsbm/v01/instances/dataFromProducer1/Product15> .}
?offer bsbm:price ?price .
?offer bsbm:vendor ?vendor .
SERVICE<http://172.17.151.172:8890/sparql/bsbm>{?vendor rdfs:label ?vendorTitle .}
?vendor bsbm:country <http://downlode.org/rdf/iso-3166/countries#DE> .
?offer dc:publisher ?vendor .
SERVICE<http://172.17.151.170:8890/sparql/bsbm> { ?offer bsbm:validTo ?date .
FILTER (?date > 2007-01-04 )}}
OPTIONAL {
SERVICE<http://172.17.151.170:8890/sparql/bsbm>{?review bsbm:reviewFor <http://www4.wiwiss.fu-
berlin.de/bizer/bsbm/v01/instances/dataFromProducer1/Product15> .}
?review rev:reviewer ?reviewer .
?reviewer foaf:name ?revName .
SERVICE<http://172.17.151.170:8890/sparql/bsbm>{   ?review dc:title ?revTitle . }
OPTIONAL { ?review bsbm:rating1 ?rating1 .}
OPTIONAL { SERVICE<http://172.17.151.170:8890/sparql/bsbm> {?review bsbm:rating2 ?rating2 .} }}}
```

Figure. 6.2: Re-written Federated Query 7 According To The 1^{st} Adaptive Partitioning

are shuffled to reduce the number of distributed joins and increase the average performance of the query workload. This was called the 2^{nd} adaptive partitioning. The newly introduced query was re-written according to the new partitioning as shown in Figure 6.7. The graph 6.5 compares the run-time of the queries, before and after the 2^{nd} adaptive partitioning. Only the queries that are affected by the shuffling of the triples are included. Similarly, Figure 6.6 shows that the number of distributed joins have decreased. From these graphs, it can be concluded that as the queries are changed, the content of the nodes also has to be changed to increase the data locality. For example, the distributed join in the new query was decreased from one to zero, and therefore it runs much faster.

3^{rd} Adaptive Partition

We continue the process of changing the query workload and see how the partitions are adapted to the change. We again added a new query shown in Figure 6.8, and Q1 was removed from the query workload. The query currently runs on PN_3 and has one distributed join. To reduce the running time of this query and remove the distributed join, all the triples

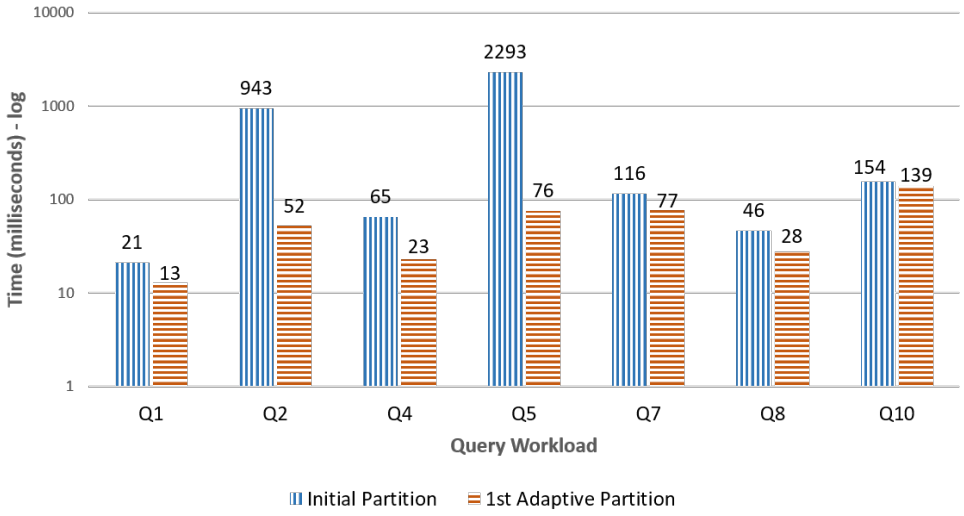

Figure. 6.3: Query run-time comparison between Initial and 1^{st} Adaptive Partition

with the predicate value $rdfs:comment$ were moved to PN_3 from PN_2. We called this the 3rd Adaptive Partition. This makes all the data that was accessed by the new query local to PN_3 and it reduced the running time significantly as shown in Figure 6.9. There were no other queries in the query workload which were affected by 3rd Adaptive Partitioning.

Final Analysis

To show the importance of re-partitioning the data according to the query workload, we changed the query workload twice. After each query-workload change, the queries are analyzed, and if possible, the partitions are changed incrementally. Figure 6.11 depicts that shuffling the triples to increase the data locality for each query workload significantly enhance the performance of the queries. It can be seen that running the queries on the initial partition (without any adaptive partitioning of triples) leads to the computation of many distributed joins and therefore the running time for the entire query workload is huge. After 1st adaptive partition, it can be observed that the number of distributed joins are reduced to almost half.

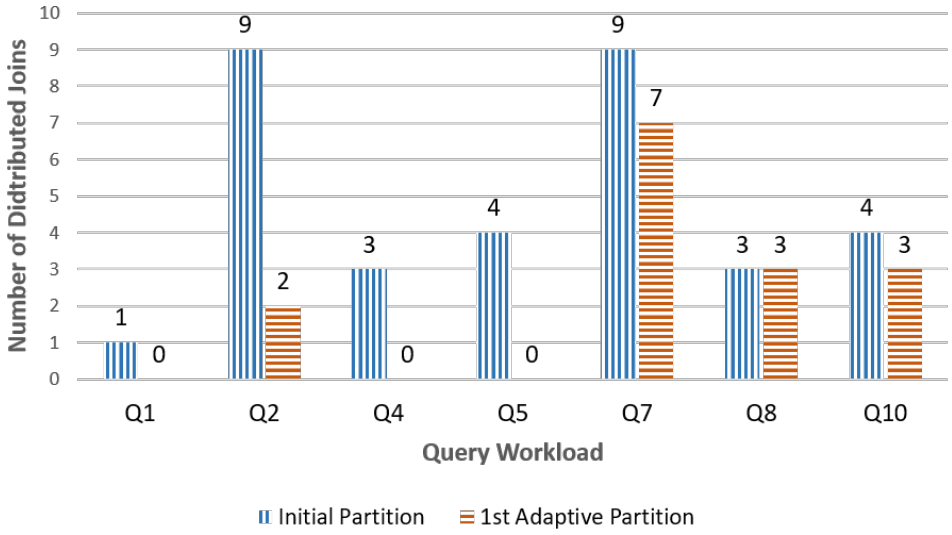

Figure. 6.4: Graphical representation of the change in the number of distributed joins from Initial to 1^{st} Adaptive Partition

Due to this, less data has to be exchanged among the nodes to evaluate the query results and consequently, the running time of the query workload improves. Every time after the query workload changes, there is a possibility of re-partition the data that suits the current queries. Therefore, an RScript is generated manually and given as an input to RePart, which initiates the process of shuffling of the triples.

Notice when the query workload is changed for the first time, there is a decrease in the number of distributed joins. This happens because when Q8 is removed from the workload, it reduces three distributed joins and the new query add only two distributed joins. When the 2^{nd} Adaptive Partitioning happens, the triples are re-shuffled in a way to reduce the maximum number of the distributed joins from the entire workload, thus also reducing the average run time of the query workload. It can be concluded that incrementally shuffling the triples among the nodes of the cluster can lead to a better SPARQL distributed system with efficient query execution.

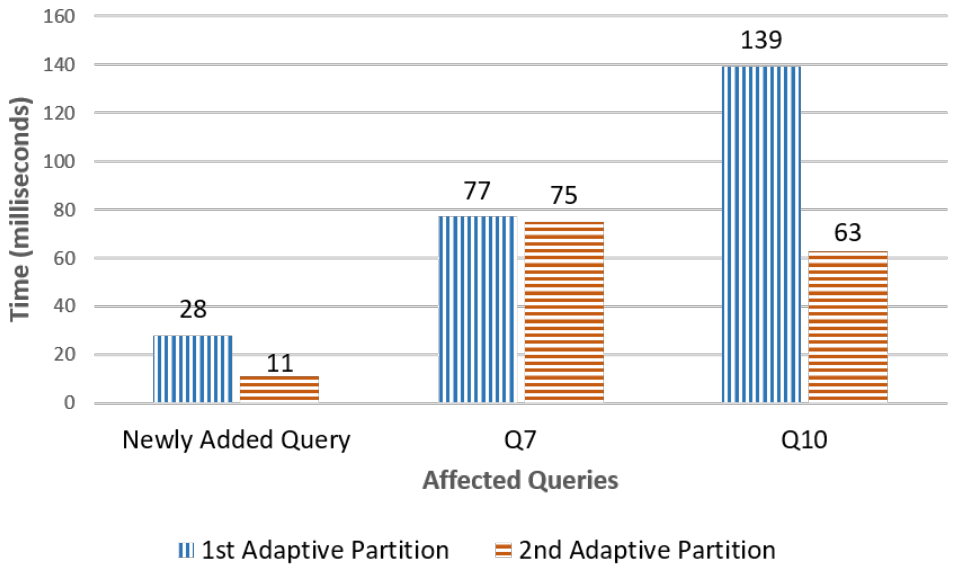

Figure. 6.5: Query runtime comparison between 1^{st} and 2^{nd} Adaptive Partition

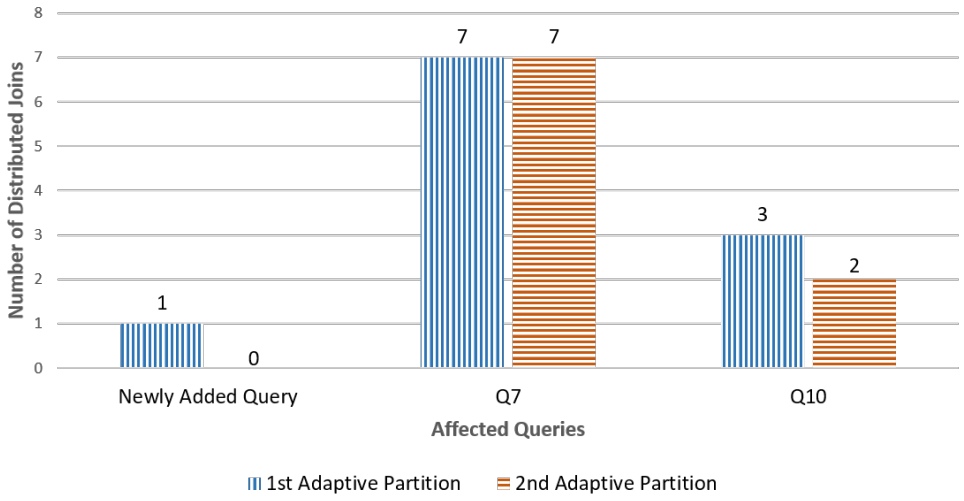

Figure. 6.6: Graphical representation of the change in the number of distributed joins from 1^{st} to 2^{nd} Adaptive Partition

Added Query

```
select distinct ?pub
from <http://localhost:8890/sparql/bsbm>
where{
SERVICE<http://172.17.151.172:8890/sparql/bsm> {?s bsbm:product
<http://www4.wiwiss.fu-
berlin.de/bizer/bsbm/v01/instances/dataFromProducer32/Product1465>.}
?s dc:publisher ?pub}
```

Re-written Query

```
select distinct ?pub
from <http://localhost:8890/sparql/bsbm>
where{
?s bsbm:product http://www4.wiwiss.fu-
berlin.de/bizer/bsbm/v01/instances/dataFromProducer32/Product1465>.
?s dc:publisher ?pup}
```

Figure. 6.7: Newly Added Query

Added Query

```
select ?product ?productFeature ?comment from <http://localhost:8890/sparql/bsbm>
where {
?s rdf:type <http://www4.wiwiss.fuberlin.de/bizer/bsbm/v01/instances/ProductType110>.
?product bsbm:productFeature ?productFeature.
SERVICE<http://172.17.151.170:8890/sparql/bsbm> {?productFeature rdfs:comment ?
comment}} limit 100
```

Re-written Query

```
select ?product ?productFeature ?comment from <http://localhost:8890/sparql/bsbm>
where {
?s rdf:type <http://www4.wiwiss.fuberlin.de/bizer/bsbm/v01/instances/ProductType110>.
?product bsbm:productFeature ?productFeature.
?productFeature rdfs:comment ?comment
} limit 100
```

Figure. 6.8: Newly Added Query

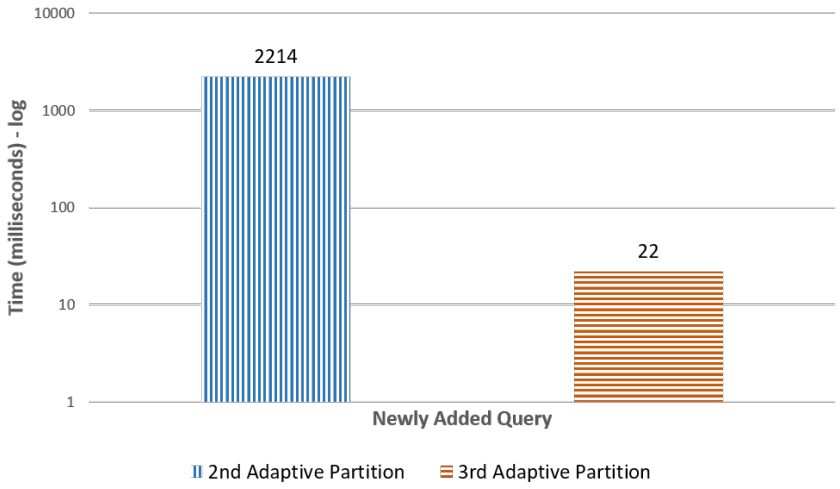

Figure. 6.9: Query runtime comparison between 2^{nd} and 3^{rd} Adaptive Partition

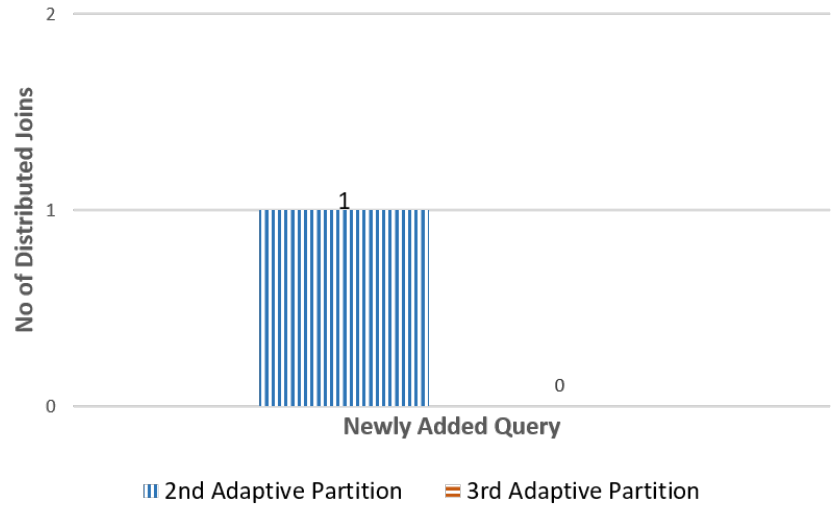

Figure. 6.10: Graphical representation of the change in the number of distributed joins from 2^{nd} to 3^{rd} Adaptive Partition

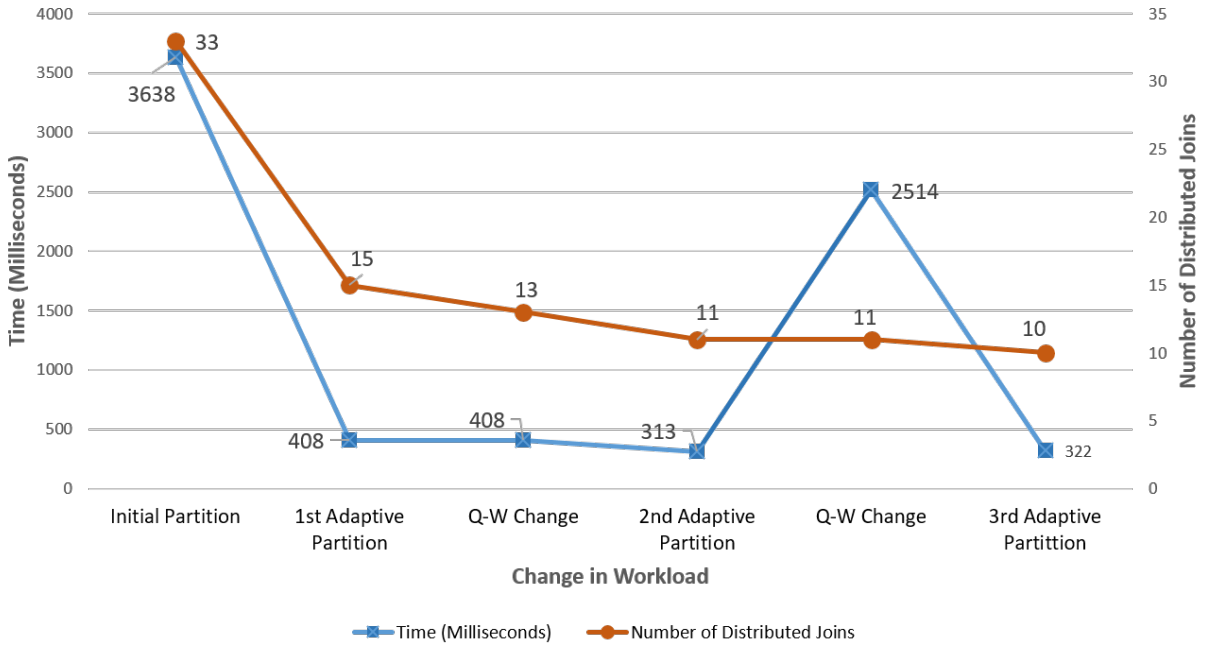

Figure. 6.11: Performance comparison when BSBM query workload changes

Table. 6.4: LUBM Initial Partition

PN 1	PN 2	PN 3	PN 4
name advisor undergraduateDegreeFrom teachingAssistantOf **type**	takesCourse **type**	publicationAuthor teacherOf telephone mastersDegreeFrom subOrganizationOf **type**	emailAddress memberOf doctoralDegreeFrom worksFor degreeFrom headOf imports researchInterest **type**

Table. 6.5: LUBM 1^{st} Adaptive Partition

PN 1	PN 2	PN 3	PN 4
name telephone worksFor **type**	takesCourse teacherOf **type**	publicationAuthor mastersDegreeFrom advisor degreeFrom headOf imports teachingAssistantOf **type**	emailAddress memberOf doctoralDegreeFrom undergraduateDegreeFrom researchInterest subOrganizationOf **type**

6.2 LUBM Results

We used another benchmark to evaluate our system's performance, the Lehigh University Benchmark (LUBM) [45]. It consists of a university domain ontology, synthetic data and a set of test queries. We generated a total of 1562711 triple using the benchmark and divided it among four processing nodes. To test our system, we select five queries from the queries that LUBM provides. They are mentioned in Appendix B and are numbered similarly to the LUBM query-set[2].

[2] `http://swat.cse.lehigh.edu/projects/lubm/queries-sparql.txt`

Table. 6.6: LUBM partition based on the predicate-object value

PN 1	PN 2	PN 3	PN 4
name telephone worksFor doctoralDegreeFrom undergraduateDegreeFrom type:ResearchAssistant type:TeachingAssistant type:UndergraduateStudent type:Employee type:AssociateProfessor	takesCourse teacherOf **type:GraduateStudent** **type:Student** type:Course type:Faculty	publicationAuthor mastersDegreeFrom advisor degreeFrom headOf imports teachingAssistantOf type:FullProfessor type:Lecturer type:Ontology type:GraduateCourse type:ResearchGroup type:Department type:University type:researchInterest	emailAddress memberOf subOrganizationOf **type:GraduateStudent** **type:Student** type:Person

Initial Partition

Similar to the BSBM benchmark, we used predicate based partitioning to generate the initial partitions. We randomly assigned a predicate to a processing node and all the triples with that predicate were assigned to the same processing node.

Rewriting Queries According To Initial Partition

Five selected queries from the LUBM query-set were added to the query-workload. The queries were re-written into federated queries as the data has to be accessed from multiple nodes. All the re-written queries are mentioned in Appendix B.1. After analyzing the queries, it was observed that almost all the queries required the triples with the predicate value *rdf:type*. At first, all the triples with *rdf:type* predicate value was kept in a single node. However, it introduced a lot of distributed joins which eventually increased the running time of the queries significantly. The SPARQL processor terminated most of the queries as they were taking excessive time to execute. Eventually, all the triples with the predicate value

rdf:type were replicated in every node to reduce the number of distributed joins. This brought down the running time of the queries. However, there was an extensive overhead because of data replication. The initial partition with replicated *rdf:type* triples (in bold) is shown in Table 6.4.

1^{st} Adaptive Partition

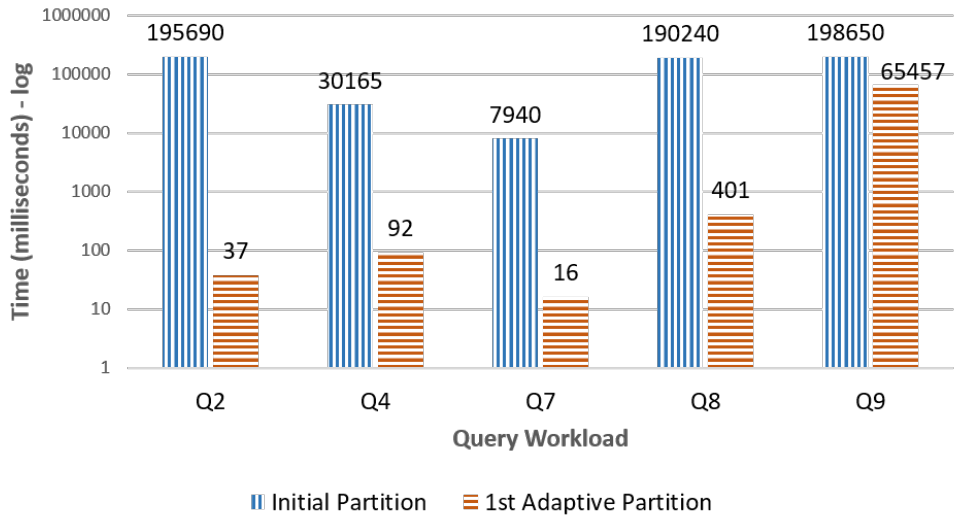

Figure. 6.12: Query run-time comparison between Initial and 1^{st} Adaptive Partition

After analyzing all the queries in the query workload, the triples were re-shuffled among the nodes while maintaining the load balance. The data which was accessed together was kept in a single triple store to reduce the total number of distributed joins in the queries. The new partition was called the 1^{st} adaptive partition and is shown in Table 6.5. Figure 6.12 shows the comparison of the running time for every query in the query workload between Initial and 1st adaptive partition. Similarly, Figure 6.4 depicts the change in the number of distributed joins for the queries when the triples are shuffled. It can be observed that the running time of the queries was improved remarkably. This happens because the 1st adaptive partition tries to make the data local to a single node of every query.

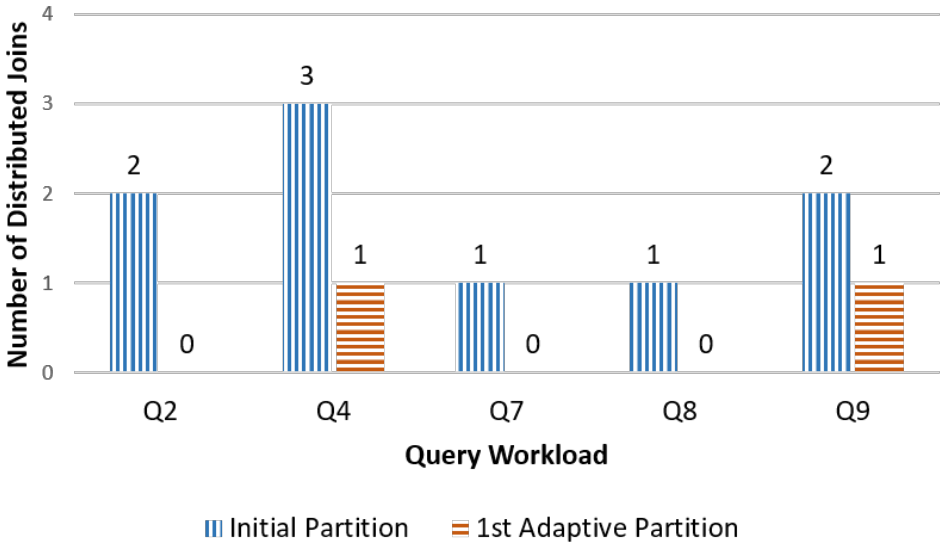

Figure. 6.13: Graphical representation of the change in the number of distributed joins from Initial to 1^{st} Adaptive Partition

Reducing The Replication Of Triples

As discussed earlier, all the triples with the predicate value *rdf:type* in the triple store of every processing node. This incured an excessive data replication overhead as almost 40% of the data was replicated.

After further analyzing the queries, it was observed that not all the *rdf:type* triples were used during the query evaluation in every processing node. For instance, consider the query shown in Figure **??**. Notice that only the triples with the predicate value *rdf:type* and the object value *ub:Professor* was used by the query. Similarly, Table 6.7 depicts the *type:Object* values corresponding to the processing node in which they were used by the queries for the evaluation of the results. This shows that the triples can further be divided according to both predicate and the object values. This is called P-O (predicate-object) division of triples which is shown in Table 6.6. Notice that only the triples with the object value of

Table. 6.7: List of the Object values that are processed by the queries in the corresponding nodes.

Object Value	**Processing Node**
ub:Department	PN_4
ub:GraduateStudent	PN_4, PN_2
ub:University	PN_4
ub:Professor	PN_1
ub:Student	PN_4, PN_2
ub:Course	PN_2
ub:Faculty	PN_2
ub:Publication	PN_3
ub:Person	PN_4
ub:ResearchGroup	PN_4

ub:GraduateStudent and *ub:Student* were required to be replicated in processing node PN_2 and PN_4. This reduces the replication of the triples significantly while there is no effect in the running time of any of the query in the query-workload. Figure 6.14 shows the reduction in the replicated triples after P-O division.

2^{nd} Adaptive Partition

To show how the content of each node was adapted according to the change in queries, we change the query workload. The Q2 was removed, and a new query shown in Figure 6.17 was added to the query workload. All the queries were analyzed again, and if possible, the content of the nodes was changed to reduce the number of distributed joins and increase the average performance of the query workload. This is called the 2^{nd} adaptive partitioning. The newly introduced query was re-written according to the new partitioning. Figure 6.15 compares the run-time of the queries, before and after the 2^{nd} adaptive partitioning. Only the queries that are affected by the shuffling of the triples are included. Similarly, Figure 6.16 shows that the number of distributed joins have decreased. From these graphs, it can be

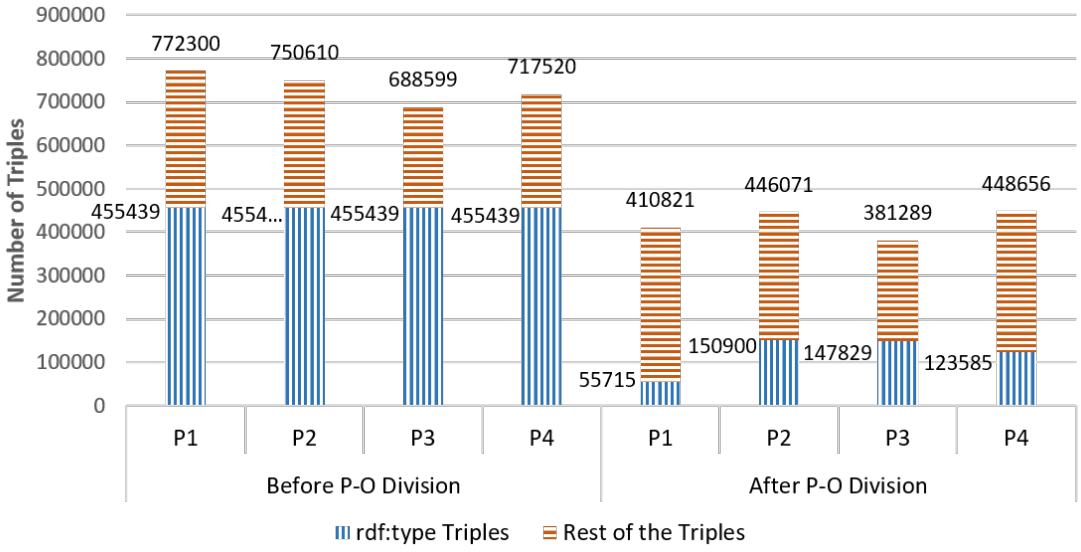

Figure. 6.14: Graphical representation of the change in the number of distributed joins from Initial to 1^{st} Adaptive Partition

concluded that as the queries are changed, the content of the nodes also has to be changed to increase the data locality. Observe that the run-time of Q8 is increased due to the new partition. However, the overall performance of the entire query workload is enhanced.

3^{rd} Adaptive Partition

The query workload was again changed to show how the system adapts to it. A new query shown in Figure 6.19 was added while Q4 was removed from the query-workload. It can be seen that the new query introduce a new distributed which was eliminated by transferring all the triples with the predicate value *ub:undergraduateDegreeFrom* to PN_3. This is called the 3rd Adaptive Partition. This makes all the data that was accessed by the new query local to PN_3 and it reduced the running time significantly as shown in Figure 6.18. There are no other queries in the query workload which are affected by 3rd Adaptive Partitioning .

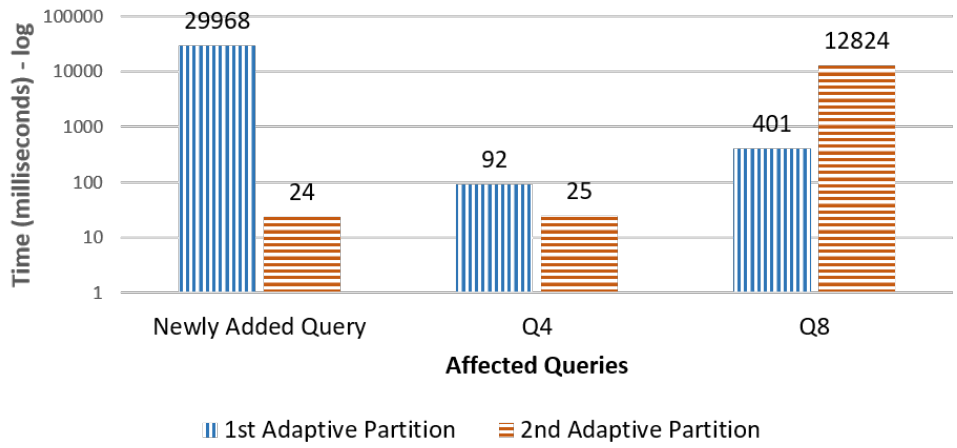

Figure. 6.15: Query run-time comparison between 1^{st} and the 2^{nd} Adaptive Partition

Final Analysis

As explained in the BSBM benchmark results, the system was able to adapt to the query workload changes. The repartitioning of the triples was performed whenever required to reduce the number of distributed joins and eventually reduce the run-time of each query in the query workload. The performance gain is depicted in Figure 6.20. It can be observed that whenever the query workload was changed, re-partitioning of the triples was performed which leads to the performance gain.

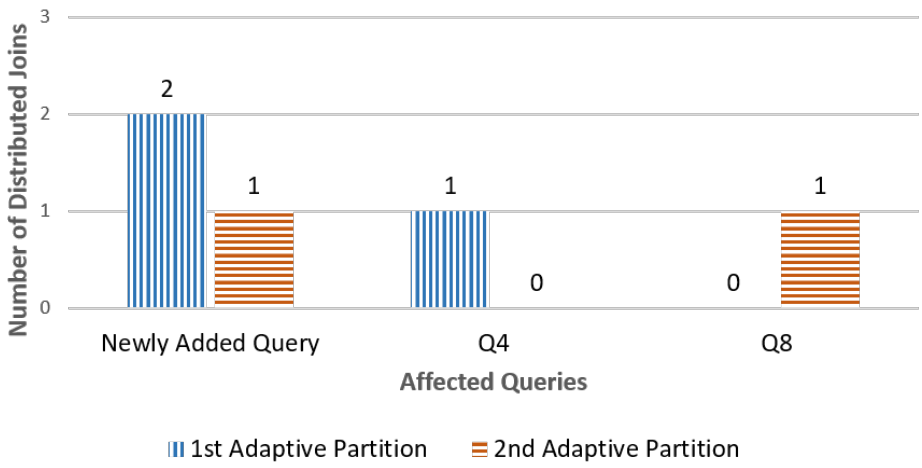

Figure. 6.16: Graphical representation of the change in the number of distributed joins from 1^{st} to 2^{nd} Adaptive Partition

Added Query

```
select distinct ?pub
from <http://localhost:8890/sparql/bsbm>
where{
SERVICE<http://172.17.151.172:8890/sparql/bsm> {?s bsbm:product
<http://www4.wiwiss.fu-
berlin.de/bizer/bsbm/v01/instances/dataFromProducer32/Product1465>.}
?s dc:publisher ?pub}
```

Re-written Query

```
select distinct ?pub
from <http://localhost:8890/sparql/bsbm>
where{
?s bsbm:product http://www4.wiwiss.fu-
berlin.de/bizer/bsbm/v01/instances/dataFromProducer32/Product1465>.
?s dc:publisher ?pup}
```

Figure. 6.17: Newly Added Query

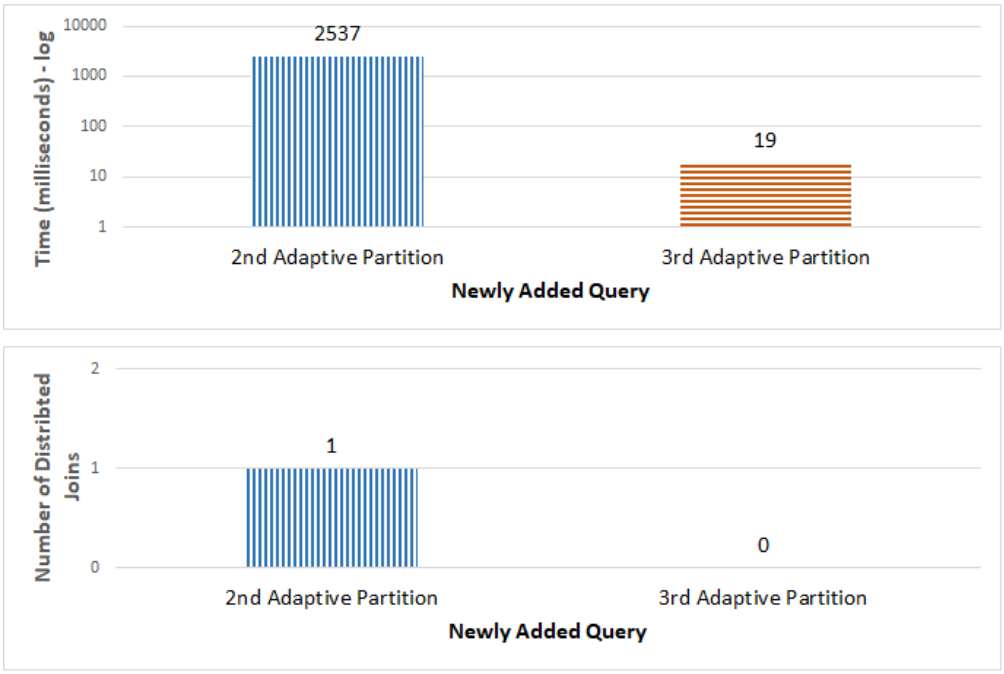

Figure. 6.18: Query run-time and distributed joins comparison between 2^{nd} and the 3^{rd} Adaptive Partition

Added Query

```
select ?product ?productFeature ?comment from <http://localhost:8890/sparql/bsbm>
where {
?s rdf:type <http://www4.wiwiss.fuberlin.de/bizer/bsbm/v01/instances/ProductType110>.
?product bsbm:productFeature ?productFeature.
SERVICE<http://172.17.151.170:8890/sparql/bsbm> {?productFeature rdfs:comment ?
comment}} limit 100
```

Re-written Query

```
select ?product ?productFeature ?comment from <http://localhost:8890/sparql/bsbm>
where {
?s rdf:type <http://www4.wiwiss.fuberlin.de/bizer/bsbm/v01/instances/ProductType110>.
?product bsbm:productFeature ?productFeature.
?productFeature rdfs:comment ?comment
} limit 100
```

Figure. 6.19: Newly Added Query

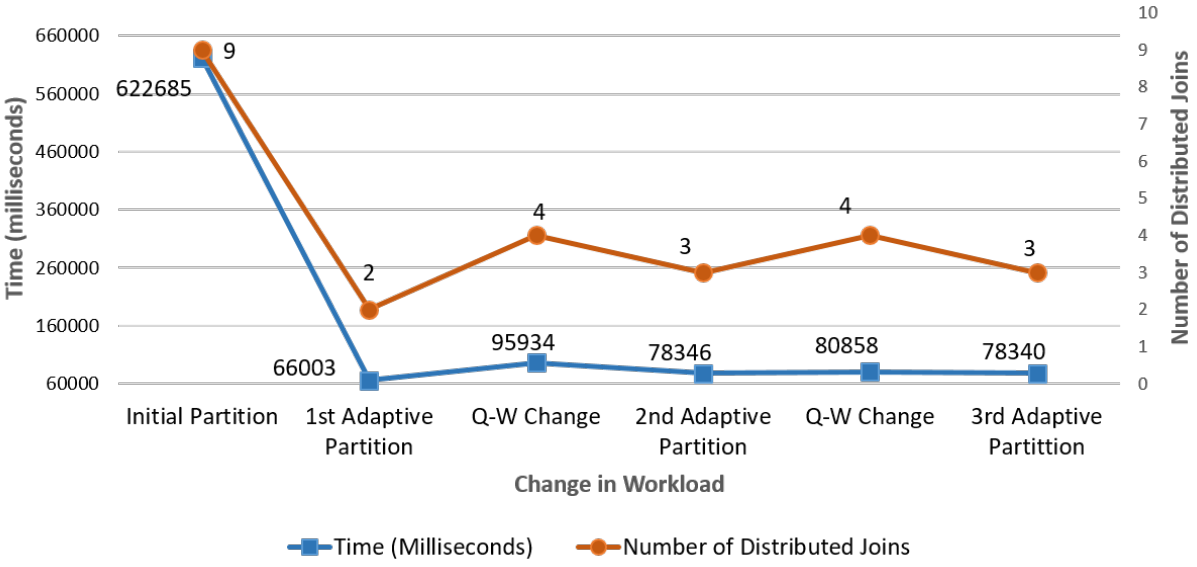

Figure. 6.20: Performance comparison when the LUBM query workload changes

CHAPTER 7

CONCLUSION

In this thesis, we have proposed RePart, a distributed SPARQL query processing system that adaptively partitions the RDF triples according to the query workload. It aims to reduce the number of distributed joins in a distributed query execution that eventually leads to a smaller run-time for the queries and better performance. This thesis is a part of a bigger research, and the major contribution is to develop an underlying system that transfers the triples from one processing node of a cluster to another. We implement an efficient system to facilitate communication between the various components of RePart using HTTP protocol. We develop a RDF-Metadata Notation that provides a profound description of the type of triples that exists in the triple store of each processing nodes. Partition Manager uses this information to decides whether a re-partitioning of triples is required. It generates RScript, which encodes the details about the shuffling of the triples. RScript is given as an input to the RScript executor, which facilitates the transfer. For evaluations, we used two benchmarks, LUBM and BSBM. They provide synthetic RDF data and a query set to test the systems. Our experiments provided a study of the effect of query-workload adaptive partitioning. We showed that as the query workload is changed, it is necessary to change the partition to make data local as much as possible for each query. The results depict a significant increase in the performance of the queries. We are also able to reduce the replication of the data while maintaining the smaller run-time for the queries.

BIBLIOGRAPHY

[1] B. DuCharme, *Learning SPARQL: querying and updating with SPARQL 1.1.* " O'Reilly Media, Inc.", 2013.

[2] "The official site for the estate of marshall mcluhan," https://www.marshallmcluhan.com/, accessed: 2010-09-30.

[3] B. DuCharme, *Learning SPARQL.* O'Reilly Media, Inc., 2011.

[4] "World wide web consortium (w3c)," https://www.w3.org/.

[5] C. Kale, "Rio: Restful interface to ontology," Master's thesis, 2011.

[6] "Rdf - semantic web standards," https://www.w3.org/RDF/.

[7] R. Al-Harbi, I. Abdelaziz, P. Kalnis, N. Mamoulis, Y. Ebrahim, and M. Sahli, "Adaptive partitioning for very large RDF data," *CoRR*, vol. abs/1505.02728, 2015. [Online]. Available: http://arxiv.org/abs/1505.02728

[8] M. T. Özsu, "A survey of RDF data management systems," *CoRR*, vol. abs/1601.00707, 2016. [Online]. Available: http://arxiv.org/abs/1601.00707

[9] I. Abdelaziz, R. Harbi, Z. Khayyat, and P. Kalnis, "A survey and experimental comparison of distributed sparql engines for very large rdf data," *Proc. VLDB Endow.*, vol. 10, no. 13, pp. 2049–2060, Sep. 2017. [Online]. Available: https://doi.org/10.14778/3151106.3151109

[10] A. Potter, B. Motik, Y. Nenov, and I. Horrocks, "Distributed rdf query answering with dynamic data exchange," in *The Semantic Web – ISWC 2016*, P. Groth, E. Simperl,

A. Gray, M. Sabou, M. Krötzsch, F. Lecue, F. Flöck, and Y. Gil, Eds. Cham: Springer International Publishing, 2016, pp. 480–497.

[11] "Linkeddata - w3c wiki," https://www.w3.org/wiki/LinkedData.

[12] R. Harbi, I. Abdelaziz, P. Kalnis, N. Mamoulis, Y. Ebrahim, and M. Sahli, "Accelerating sparql queries by exploiting hash-based locality and adaptive partitioning," *The VLDB Journal*, vol. 25, no. 3, pp. 355–380, 2016.

[13] L. Galárraga, K. Hose, and R. Schenkel, "Partout: A distributed engine for efficient RDF processing," *CoRR*, vol. abs/1212.5636, 2012. [Online]. Available: http://arxiv.org/abs/1212.5636

[14] K. Hose and R. Schenkel, "Warp: Workload-aware replication and partitioning for rdf," in *Data Engineering Workshops (ICDEW), 2013 IEEE 29th International Conference on*. IEEE, 2013, pp. 1–6.

[15] S. Yang, X. Yan, B. Zong, and A. Khan, "Towards effective partition management for large graphs," in *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*. ACM, 2012, pp. 517–528.

[16] M. Hammoud, D. A. Rabbou, R. Nouri, S.-M.-R. Beheshti, and S. Sakr, "Dream: distributed rdf engine with adaptive query planner and minimal communication," *Proceedings of the VLDB Endowment*, vol. 8, no. 6, pp. 654–665, 2015.

[17] B. Quilitz and U. Leser, "Querying distributed rdf data sources with sparql," in *The Semantic Web: Research and Applications*, S. Bechhofer, M. Hauswirth, J. Hoffmann, and M. Koubarakis, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 524–538.

[18] P. Peng, L. Zou, M. T. Özsu, L. Chen, and D. Zhao, "Processing sparql queries over distributed rdf graphs," *The VLDB Journal*, vol. 25, no. 2, pp. 243–268, Apr. 2016. [Online]. Available: http://dx.doi.org/10.1007/s00778-015-0415-0

[19] B. Quilitz and U. Leser, "Querying distributed rdf data sources with sparql," in *The Semantic Web: Research and Applications*, S. Bechhofer, M. Hauswirth, J. Hoffmann, and M. Koubarakis, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 524–538.

[20] G. Aluç, M. T. Özsu, and K. Daudjee, "Workload matters: Why rdf databases need a new design," *Proceedings of the VLDB Endowment*, vol. 7, no. 10, pp. 837–840, 2014.

[21] "Resource description framework - wikipedia," https://en.wikipedia.org/wiki/Resource_Description_Framework#Serialization_formats.

[22] "Ontologies - w3c," https://www.w3.org/standards/semanticweb/ontology.

[23] "Rdfs - semantic web standards," https://www.w3.org/2001/sw/wiki/RDFS.

[24] "Dcmi: Home," http://dublincore.org/.

[25] "Owl - semantic web standards," https://www.w3.org/OWL/.

[26] "Sparql query language for rdf," https://www.w3.org/TR/rdf-sparql-query/.

[27] "Sparql by example," https://www.w3.org/2009/Talks/0615-qbe/.

[28] "Sparqlendpoints - w3c wiki," https://www.w3.org/wiki/SparqlEndpoints.

[29] "Sparql 1.1 federated query," https://www.w3.org/TR/sparql11-federated-query/.

[30] "Redland rdf libraries," http://librdf.org/.

[31] A. Jena, "semantic web framework for java," 2007.

[32] "Openlink virtuoso," https://virtuoso.openlinksw.com/.

[33] "Rest - the short version," http://exyus.com/articles/rest-the-short-version/.

[34] H. Firth, P. Missier, and J. Aiston, "Loom: Query-aware partitioning of online graphs," *CoRR*, vol. abs/1711.06608, 2017. [Online]. Available: http://arxiv.org/abs/1711.06608

[35] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski, "Pregel: a system for large-scale graph processing," in *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*. ACM, 2010, pp. 135–146.

[36] T. Neumann and G. Weikum, "The rdf-3x engine for scalable management of rdf data," *The VLDB JournalThe International Journal on Very Large Data Bases*, vol. 19, no. 1, pp. 91–113, 2010.

[37] X. Zhang, L. Chen, Y. Tong, and M. Wang, "Eagre: Towards scalable i/o efficient sparql query evaluation on the cloud," in *Data engineering (ICDE), 2013 ieee 29th international conference on*. IEEE, 2013, pp. 565–576.

[38] A. Schätzle, M. Przyjaciel-Zablocki, S. Skilevic, and G. Lausen, "S2rdf: Rdf querying with sparql on spark," *Proceedings of the VLDB Endowment*, vol. 9, no. 10, pp. 804–815, 2016.

[39] D. J. Abadi, A. Marcus, S. R. Madden, and K. Hollenbach, "Scalable semantic web data management using vertical partitioning," in *Proceedings of the 33rd international conference on Very large data bases*. VLDB Endowment, 2007, pp. 411–422.

[40] G. Karypis and V. Kumar, "Metis–unstructured graph partitioning and sparse matrix ordering system, version 2.0," 1995.

[41] J. Huang, D. J. Abadi, and K. Ren, "Scalable sparql querying of large rdf graphs," *Proceedings of the VLDB Endowment*, vol. 4, no. 11, pp. 1123–1134, 2011.

[42] Y. Lu, A. Shanbhag, A. Jindal, and S. Madden, "Adaptdb: Adaptive partitioning for distributed joins," *Proc. VLDB Endow.*, vol. 10, no. 5, pp. 589–600, Jan. 2017. [Online]. Available: https://doi.org/10.14778/3055540.3055551

[43] A. Schwarte, P. Haase, K. Hose, R. Schenkel, and M. Schmidt, "Fedx: Optimization techniques for federated query processing on linked data," in *International Semantic*

Web Conference. Springer, 2011, pp. 601–616.

[44] O. Hartig and R. Heese, "The sparql query graph model for query optimization," in *European Semantic Web Conference*. Springer, 2007, pp. 564–578.

[45] Y. Guo, Z. Pan, and J. Heflin, "Lubm: A benchmark for owl knowledge base systems," *Web Semantics: Science, Services and Agents on the World Wide Web*, vol. 3, no. 2-3, pp. 158–182, 2005.

[46] C. Bizer and A. Schultz, "The berlin sparql benchmark," 2009.

APPENDIX A

BSBM Queries

A.1 Rewriting BSBM Queries Into Federated Queries According To The Initial Partition

```
SELECT DISTINCT ?product ?label
from <http://localhost:8890/sparql/bsbm>
WHERE {
?product rdfs:label ?label .
?product a <http://www4.wiwiss.fu-berlin.de/bizer/bsbm/v01/instances/ProductType10> .
?product bsbm:productFeature
<http://www4.wiwiss.fuberlin.de/bizer/bsbm/v01/instances/ProductFeature10>.
?product bsbm:productFeature <http://www4.wiwiss.fu-
berlin.de/bizer/bsbm/v01/instances/ProductFeature20> .
SERVICE <http://172.17.151.170:8890/sparql/bsbm> { ?product bsbm:productPropertyNumeric1
?value1 . }
FILTER (?value1 > 100)}
ORDER BY ?label
LIMIT 10
```

Figure. A.1: Re-written Federated Query 1 According To The Initial Partition: Runs on PN_3

```
SELECT ?label ?comment ?producer ?productFeature ?propertyTextual1 ?propertyTextual2 ?
propertyTextual3?propertyNumeric1 ?propertyNumeric2 ?propertyTextual4 ?propertyTextual5 ?
propertyNumeric4
WHERE {
SERVICE<http://172.17.151.172:8890/sparql/bsbm> {<http://www4.wiwiss.fu-
berlin.de/bizer/bsbm/v01/instances/dataFromProducer1/Product16> rdfs:label ?label . }
<http://www4.wiwiss.fu-berlin.de/bizer/bsbm/v01/instances/dataFromProducer1/Product16>
rdfs:comment ?comment .
SERVICE<http://172.17.151.171:8890/sparql/bsbm> {
<http://www4.wiwiss.fu-berlin.de/bizer/bsbm/v01/instances/dataFromProducer1/Product16>
bsbm:producer ?p . }
SERVICE <http://172.17.151.171:8890/sparql/bsbm> {
?p rdfs:label ?producer . }
<http://www4.wiwiss.fu-berlin.de/bizer/bsbm/v01/instances/dataFromProducer1/Product16>
dc:publisher ?p .
SERVICE <http://172.17.151.172:8890/sparql/bsbm> {
<http://www4.wiwiss.fu-berlin.de/bizer/bsbm/v01/instances/dataFromProducer1/Product16>
bsbm:productFeature ?f .
?f rdfs:label ?productFeature .}
<http://www4.wiwiss.fu-berlin.de/bizer/bsbm/v01/instances/dataFromProducer1/Product16>
bsbm:productPropertyTextual1 ?propertyTextual1 .
SERVICE <http://172.17.151.171:8890/sparql/bsbm> {
<http://www4.wiwiss.fu-berlin.de/bizer/bsbm/v01/instances/dataFromProducer1/Product16>
bsbm:productPropertyTextual2 ?propertyTextual2 .
<http://www4.wiwiss.fu-berlin.de/bizer/bsbm/v01/instances/dataFromProducer1/Product16>
bsbm:productPropertyTextual3 ?propertyTextual3 .}
{ <http://www4.wiwiss.fu-berlin.de/bizer/bsbm/v01/instances/dataFromProducer1/Product16>
bsbm:productPropertyNumeric1 ?propertyNumeric1 .
SERVICE <http://172.17.151.173:8890/sparql/bsbm>{
<http://www4.wiwiss.fu-berlin.de/bizer/bsbm/v01/instances/dataFromProducer1/Product16>
bsbm:productPropertyNumeric2 ?propertyNumeric2 .}
OPTIONAL {
SERVICE <http://172.17.151.171:8890/sparql/bsbm>{
<http://www4.wiwiss.fu-berlin.de/bizer/bsbm/v01/instances/dataFromProducer1/Product16>
bsbm:productPropertyTextual4 ?propertyTextual4 .}
}
OPTIONAL { <http://www4.wiwiss.fu-
berlin.de/bizer/bsbm/v01/instances/dataFromProducer1/Product16> bsbm:productPropertyTextual5 ?
propertyTextual5 }
OPTIONAL { <http://www4.wiwiss.fu-
berlin.de/bizer/bsbm/v01/instances/dataFromProducer1/Product16> bsbm:productPropertyNumeric4 ?
propertyNumeric4 }
}
```

Figure. A.2: Re-written Federated Query 2 According To The Initial Partition: Runs on PN_2

```
SELECT DISTINCT ?product ?label ?propertyTextual
from <http://localhost:8890/sparql/bsbm>
WHERE {{
?product rdfs:label ?label .
?product rdf:type <http://www4.wiwiss.fu
berlin.de/bizer/bsbm/v01/instances/ProductType11>.
?product bsbm:productFeature <http://www4.wiwiss.fu-
berlin.de/bizer/bsbm/v01/instances/ProductFeature12> .
?product bsbm:productFeature <http://www4.wiwiss.fu-
berlin.de/bizer/bsbm/v01/instances/ProductFeature17> .
SERVICE<http://172.17.151.170:8890/sparql/bsbm>{ ?product bsbm:productPropertyTextual1 ?
propertyTextual .
?product bsbm:productPropertyNumeric1 ?p1 .}
FILTER ( ?p1 > 1 )}
UNION {
?product rdfs:label ?label .
?product rdf:type <http://www4.wiwiss.fu-berlin.de/bizer/bsbm/v01/instances/ProductType11>.
?product bsbm:productFeature
<http://www4.wiwiss.fuberlin.de/bizer/bsbm/v01/instances/ProductFeature12>.
?product bsbm:productFeature
<http://www4.wiwiss.fuberlin.de/bizer/bsbm/v01/instances/ProductFeature1841>.
SERVICE<http://172.17.151.170:8890/sparql/bsbm> { ?product bsbm:productPropertyTextual1 ?
propertyTextual . }
SERVICE<http://172.17.151.173:8890/sparql/bsbm> {   ?product bsbm:productPropertyNumeric2 ?p2
. }
FILTER ( ?p2> 50 )}}
ORDER BY ?label OFFSET 5
```

Figure. A.3: Re-written Federated Query 4 According To The Initial Partition: Runs on PN_4

```
SELECT DISTINCT ?product ?productLabel
WHERE {
?product rdfs:label ?productLabel .
FILTER (<http://www4.wiwiss.fu-berlin.de/bizer/bsbm/v01/instances/dataFromProducer1/Product25>
!= ?product)
<http://www4.wiwiss.fu-berlin.de/bizer/bsbm/v01/instances/dataFromProducer1/Product25>
bsbm:productFeature ?prodFeature .
?product bsbm:productFeature ?prodFeature .
SERVICE<http://172.17.151.170:8890/sparql/bsbm> {
<http://www4.wiwiss.fuberlin.de/bizer/bsbm/v01/instances/dataFromProducer1/Product25>
bsbm:productPropertyNumeric1 ?origProperty1 .
?product bsbm:productPropertyNumeric1 ?simProperty1 .}
FILTER (?simProperty1 < (?origProperty1 + 120) && ?simProperty1 > (?origProperty1 - 120))
SERVICE<http://172.17.151.173:8890/sparql/bsbm>{
<http://www4.wiwiss.fu-berlin.de/bizer/bsbm/v01/instances/dataFromProducer1/Product25>
bsbm:productPropertyNumeric2 ?origProperty2 .
?product bsbm:productPropertyNumeric2 ?simProperty2 .}
FILTER (?simProperty2 < (?origProperty2 + 170) && ?simProperty2 > (?origProperty2 - 170))}
ORDER BY ?productLabel LIMIT 5
```

Figure. A.4: Re-written Federated Query 5 According To The Initial Partition: Runs on PN_3

```
SELECT ?title ?text ?reviewDate ?reviewer ?reviewerName ?rating1 ?rating2 ?rating3 ?rating4
WHERE {
SERVICE <http://172.17.151.173:8890/sparql/bsbm> { ?review bsbm:reviewFor
<http://www4.wiwiss.fu-berlin.de/bizer/bsbm/v01/instances/dataFromProducer1/Product21> . }
?review dc:title ?title .
SERVICE<http://172.17.151.170:8890/sparql/bsbm>{ ?review rev:text ?text .
FILTER langMatches( lang(?text), "EN" ) }
?review bsbm:reviewDate ?reviewDate .
SERVICE <http://172.17.151.172:8890/sparql/bsbm> {   ?review rev:reviewer ?reviewer . }
?reviewer foaf:name ?reviewerName .
OPTIONAL { ?review bsbm:rating1 ?rating1 . }
OPTIONAL { ?review bsbm:rating2 ?rating2 . }
OPTIONAL { ?review bsbm:rating3 ?rating3 . }
OPTIONAL { ?review bsbm:rating4 ?rating4 . }
}
ORDER BY DESC(?reviewDate) LIMIT 20
```

Figure. A.5: Re-written Federated Query 8 According To The Initial Partition: Runs on PN_1

```
SELECT DISTINCT ?offer ?price
from <http://localhost:8890/sparql/bsbm>
WHERE {
SERVICE<http://172.17.151.172:8890/sparql/bsbm> {   ?offer bsbm:product
<http://www4.wiwiss.fuberlin.de/bizer/bsbm/v01/instances/dataFromProducer11/Product487> . }
?offer bsbm:vendor ?vendor .
?offer dc:publisher?vendor .
SERVICE<http://172.17.151.171:8890/sparql/bsbm> {    ?vendor bsbm:country
<http://downlode.org/rdf/iso-3166/countries#US> . }
SERVICE<http://172.17.151.173:8890/sparql/bsbm> {?offer bsbm:deliveryDays ?deliveryDays . }
FILTER (?deliveryDays <= 3)
SERVICE<http://172.17.151.173:8890/sparql/bsbm> {    ?offer bsbm:price ?price . }
?offer bsbm:validTo ?date .
FILTER (?date > 2007-01-04 )}
ORDER BY xsd:double(str(?price))
LIMIT 10
```

Figure. A.6: Re-written Federated Query 10 According To The Initial Partition: Runs on PN_2

A.2 Rewriting Queries Into Federated Queries According To The 1^{st} Adaptive Partition

```
SELECT DISTINCT ?product ?label
WHERE {?product rdfs:label ?label .
?product a <http://www4.wiwiss.fuberlin.de/bizer/bsbm/v01/instances/ProductType10> .
?product bsbm:productFeature <http://www4.wiwiss.fu-
berlin.de/bizer/bsbm/v01/instances/ProductFeature10>.
?product bsbm:productFeature
<http://www4.wiwiss.fuberlin.de/bizer/bsbm/v01/instances/ProductFeature20> .
?product bsbm:productPropertyNumeric1 ?value1 .
FILTER (?value1 > 100)
}
ORDER BY ?label
LIMIT 10
```

Figure. A.7: Re-written Federated Query 1 According To 1^{st} Adaptive Partition: Runs on PN_2

```
SELECT ?label ?comment ?producer ?productFeature ?propertyTextual1 ?propertyTextual2 ?
propertyTextual3?propertyNumeric1 ?propertyNumeric2 ?propertyTextual4 ?propertyTextual5 ?
propertyNumeric4
WHERE {
<http://www4.wiwiss.fu-berlin.de/bizer/bsbm/v01/instances/dataFromProducer1/Product16>
rdfs:label ?label .
SERVICE <http://172.17.151.170:8890/sparql/bsbm> {
<http://www4.wiwiss.fuberlin.de/bizer/bsbm/v01/instances/dataFromProducer1/Product16>
rdfs:comment ?comment .}
<http://www4.wiwiss.fu-berlin.de/bizer/bsbm/v01/instances/dataFromProducer1/Product16>
bsbm:producer ?p .
?p rdfs:label ?producer .
SERVICE <http://172.17.151.171> {
<http://www4.wiwiss.fuberlin.de/bizer/bsbm/v01/instances/dataFromProducer1/Product16>
dc:publisher ?p .}
<http://www4.wiwiss.fu-berlin.de/bizer/bsbm/v01/instances/dataFromProducer1/Product16>
bsbm:productFeature ?f .
?f rdfs:label ?productFeature .
<http://www4.wiwiss.fu-berlin.de/bizer/bsbm/v01/instances/dataFromProducer1/Product16>
bsbm:productPropertyTextual1 ?propertyTextual1 .
<http://www4.wiwiss.fu-berlin.de/bizer/bsbm/v01/instances/dataFromProducer1/Product16>
bsbm:productPropertyTextual2 ?propertyTextual2 .
<http://www4.wiwiss.fu-berlin.de/bizer/bsbm/v01/instances/dataFromProducer1/Product16>
bsbm:productPropertyTextual3 ?propertyTextual3 .
<http://www4.wiwiss.fu-berlin.de/bizer/bsbm/v01/instances/dataFromProducer1/Product16>
bsbm:productPropertyNumeric1 ?propertyNumeric1 .
<http://www4.wiwiss.fu-berlin.de/bizer/bsbm/v01/instances/dataFromProducer1/Product16>
bsbm:productPropertyNumeric2 ?propertyNumeric2 .
OPTIONAL {
<http://www4.wiwiss.fuberlin.de/bizer/bsbm/v01/instances/dataFromProducer1/Product16>
bsbm:productPropertyTextual4 ?propertyTextual4 }
OPTIONAL {
<http://www4.wiwiss.fuberlin.de/bizer/bsbm/v01/instances/dataFromProducer1/Product16>
bsbm:productPropertyTextual5 ?propertyTextual5 }
OPTIONAL {
<http://www4.wiwiss.fuberlin.de/bizer/bsbm/v01/instances/dataFromProducer1/Product16>
bsbm:productPropertyNumeric4 ?propertyNumeric4 }}
```

Figure. A.8: Re-written Federated Query 2 According To 1^{st} Adaptive Partition: Runs on PN_3

```
SELECT DISTINCT ?product ?label ?propertyTextual
WHERE {
{?product rdfs:label ?label .
?product rdf:type
<http://www4.wiwiss.fuberlin.de/bizer/bsbm/v01/instances/ProductType11>.
?product bsbm:productFeature
<http://www4.wiwiss.fuberlin.de/bizer/bsbm/v01/instances/ProductFeature12> .
?product bsbm:productFeature <http://www4.wiwiss.fu-
berlin.de/bizer/bsbm/v01/instances/ProductFeature17> .
?product bsbm:productPropertyTextual11 ?propertyTextual .
?product bsbm:productPropertyNumeric1 ?p1 .
FILTER ( ?p1 > 1 )}
UNION {
?product rdfs:label ?label .
?product rdf:type
<http://www4.wiwiss.fuberlin.de/bizer/bsbm/v01/instances/ProductType11>  .
?product bsbm:productFeature
<http://www4.wiwiss.fuberlin.de/bizer/bsbm/v01/instances/ProductFeature12>.
?product bsbm:productFeature
<http://www4.wiwiss.fuberlin.de/bizer/bsbm/v01/instances/ProductFeature1841> .
?product bsbm:productPropertyTextual11 ?propertyTextual .
?product bsbm:productPropertyNumeric2 ?p2 .
FILTER ( ?p2> 50 )}}
ORDER BY ?label
OFFSET 5
LIMIT 10
```

Figure. A.9: Re-written Federated Query 4 According To 1^{st} Adaptive Partition: Runs on PN_3

```
SELECT DISTINCT ?product ?productLabel
WHERE { ?product rdfs:label ?productLabel .
FILTER (<http://www4.wiwiss.fu-
berlin.de/bizer/bsbm/v01/instances/dataFromProducer1/Product25> != ?product
<http://www4.wiwiss.fuberlin.de/bizer/bsbm/v01/instances/dataFromProducer1/Product25>
bsbm:productFeature ?prodFeature .
?product bsbm:productFeature ?prodFeature .
<http://www4.wiwiss.fuberlin.de/bizer/bsbm/v01/instances/dataFromProducer1/Product25>
bsbm:productPropertyNumeric1 ?origProperty1 .
?product bsbm:productPropertyNumeric1 ?simProperty1 .
FILTER (?simProperty1 < (?origProperty1 + 120) && ?simProperty1 > (?origProperty1 - 120))
<http://www4.wiwiss.fu-berlin.de/bizer/bsbm/v01/instances/dataFromProducer1/Product25>
bsbm:productPropertyNumeric2 ?origProperty2 .
?product bsbm:productPropertyNumeric2 ?simProperty2 .
FILTER (?simProperty2 < (?origProperty2 + 170) && ?simProperty2 > (?origProperty2 - 170))
}
ORDER BY ?productLabel
LIMIT 5
```

Figure. A.10: Re-written Federated Query 5 According To 1^{st} Adaptive Partition: Runs on PN_3

```
SELECT ?title ?text ?reviewDate ?reviewer ?reviewerName ?rating1 ?rating2 ?rating3 ?rating4
WHERE {
?review bsbm:reviewFor <http://www4.wiwiss.fu-
berlin.de/bizer/bsbm/v01/instances/dataFromProducer1/Product21> .
?review dc:title ?title .
?review rev:text ?text .
FILTER langMatches( lang(?text), "EN" )
?review bsbm:reviewDate ?reviewDate .
SERVICE <http://172.17.151.171:8890/sparql/bsbm> { ?review rev:reviewer ?reviewer .
?reviewer foaf:name ?reviewerName . }
OPTIONAL { SERVICE <http://172.17.151.171:8890/sparql/bsbm> {?review bsbm:rating1 ?rating1 . }
}
OPTIONAL { ?review bsbm:rating2 ?rating2 . }
OPTIONAL { ?review bsbm:rating3 ?rating3 . }
OPTIONAL { ?review bsbm:rating4 ?rating4 . }}
ORDER BY DESC(?reviewDate) LIMIT 20
```

Figure. A.11: Re-written Federated Query 8 According To 1^{st} Adaptive Partition: Runs on PN_2

```
SELECT DISTINCT ?offer ?price
from <http://localhost:8890/sparql/bsbm>
WHERE {
SERVICE<http://172.17.151.172:8890/sparql/bsbm>{   ?offer bsbm:product <http://www4.wiwiss.fu-
berlin.de/bizer/bsbm/v01/instances/dataFromProducer11/Product487> . }
?offer bsbm:vendor ?vendor .
?offer dc:publisher ?vendor .
?vendor bsbm:country <http://downlode.org/rdf/iso-3166/countries#US> .
SERVICE<http://172.17.151.173:8890/sparql/bsbm>{
?offer bsbm:deliveryDays ?deliveryDays . }
FILTER (?deliveryDays <= 3)
?offer bsbm:price ?price .
SERVICE<http://172.17.151.170:8890/sparql/bsbm>{
?offer bsbm:validTo ?date .
FILTER (?date > 2007-01-04 )}
}
ORDER BY xsd:double(str(?price)) LIMIT 10
```

Figure. A.12: Re-written Federated Query 10 According To 1^{st} Adaptive Partition: Runs on PN_1

A.3 Rewriting Query Into A Federated Queries According To The 2^{nd} Adaptive Partition

```
SELECT DISTINCT ?offer ?price
from <http://localhost:8890/sparql/bsbm>
WHERE {
?offer bsbm:product <http://www4.wiwiss.fu-
berlin.de/bizer/bsbm/v01/instances/dataFromProducer11/Product487> .
?offer bsbm:vendor ?vendor .
?offer dc:publisher ?vendor .
?vendor bsbm:country <http://downlode.org/rdf/iso-3166/countries#US> .
SERVICE<http://172.17.151.173:8890/sparql/bsbm>{
?offer bsbm:deliveryDays ?deliveryDays . }
FILTER (?deliveryDays <= 3)
?offer bsbm:price ?price .
SERVICE<http://172.17.151.170:8890/sparql/bsbm>{
?offer bsbm:validTo ?date .
FILTER (?date > 2007-01-04 )}}
ORDER BY xsd:double(str(?price))
LIMIT 10
```

Figure. A.13: Re-written Federated Query 10 According To 2^{nd} Adaptive Partition: Runs on PN_1

APPENDIX B

LUBM Queries

B.1 Rewriting LUBM Queries Into Federated Queries According To The Initial Partition

```
SELECT ?X, ?Y, ?Z FROM <http://localhost:8890/sparql/lubmInitial>
WHERE{
?X rdf:type ub:GraduateStudent .
?Y rdf:type ub:University .
?Z rdf:type ub:Department .
?X ub:memberOf ?Z .
SERVICE <http://172.17.151.172:8890/sparql/lubmInitial> {?Z
ub:subOrganizationOf ?Y . }
SERVICE <http://172.17.151.171:8890/sparql/lubmInitial> { ?X
ub:undergraduateDegreeFrom ?Y }
}
```

Figure. B.1: Re-written Federated Query 2 According To Initial Adaptive Partition: Runs on PN_4

```
SELECT ?X, ?Y1, ?Y2, ?Y3 FROM <http://localhost:8890/sparql/lubm>
WHERE
{?X rdf:type ub:Professor .
SERVICE <http://172.17.151.173:8890/sparql/lubmInitial>{?X ub:worksFor
<http://www.Department0.University0.edu> .
?X ub:emailAddress ?Y2 .}
?X ub:name ?Y1 .
SERVICE <http://172.17.151.172:8890/sparql/lubmInitial> { ?X
ub:telephone ?Y3 }

}
```

Figure. B.2: Re-written Federated Query 4 According To Initial Adaptive Partition: Runs on PN_1

```
SELECT ?X, ?Y FROM <http://localhost:8890/sparql/lubm>
WHERE{
?X rdf:type ub:Student .
?Y rdf:type ub:Course .
?X ub:takesCourse ?Y .
SERVICE <http://172.17.151.172:8890/sparql/lubm>{
<http://www.Department0.University0.edu/AssociateProfessor0>
ub:teacherOf ?Y.}
}
```

Figure. B.3: Re-written Federated Query 7 According To Initial Adaptive Partition: Runs on PN_2

```
SELECT ?X, ?Y, ?Z FROM <http://localhost:8890/sparql/lubmInitial>
WHERE
{?X rdf:type ub:Student .
?Y rdf:type ub:Department .
?X ub:emailAddress ?Z .
?X ub:memberOf ?Y .
SERVICE <http://172.17.151.172:8890/sparql/lubmInitial> {?Y
ub:subOrganizationOf <http://www.University0.edu>.}}
```

Figure. B.4: Re-written Federated Query 8 According To Initial Adaptive Partition: Runs on PN_8

```
SELECT ?X, ?Y, ?Z FROM <http://localhost:8890/sparql/lubmInitial>
WHERE
{?X rdf:type ub:Student .
?Y rdf:type ub:Faculty .
?Z rdf:type ub:Course .
?X ub:advisor ?Y .
SERVICE <http://172.17.151.172:8890/sparql/lubmInitial> { ?Y
ub:teacherOf ?Z . }
SERVICE <http://172.17.151.170:8890/sparql/lubmInitial> { ?X
ub:takesCourse ?Z }}
```

Figure. B.5: Re-written Federated Query 9 According To Initial Adaptive Partition: Runs on PN_1

B.2 Rewriting Queries Into Federated Queries According To The 1^{st} Adaptive Partition

```
SELECT ?X, ?Y, ?Z FROM <http://localhost:8890/sparql/lubm>
WHERE{
?X rdf:type ub:GraduateStudent .
?Y rdf:type ub:University .
?Z rdf:type ub:Department .
?X ub:memberOf ?Z .
?Z ub:subOrganizationOf ?Y .
?X ub:undergraduateDegreeFrom ?Y}
```

Figure. B.6: Re-written Federated Query 2 According To 1^{st} Adaptive Partition: Runs on PN_4

```
SELECT ?X, ?Y1, ?Y2, ?Y3 FROM <http://localhost:8890/sparql/lubmInitial>
WHERE
{?X rdf:type ub:Professor .
SERVICE <http://172.17.151.173:8890/sparql/lubm>{?X ub:emailAddress ?Y2 .}
?X ub:worksFor <http://www.Department0.University0.edu> .
?X ub:name ?Y1 .
?X ub:telephone ?Y3}
```

Figure. B.7: Re-written Federated Query 4 According To 1^{st} Adaptive Partition: Runs on PN_1

```
SELECT ?X, ?Y FROM <http://localhost:8890/sparql/lubm>
WHERE{
?X rdf:type ub:Student .
?Y rdf:type ub:Course .
?X ub:takesCourse ?Y .
<http://www.Department0.University0.edu/AssociateProfessor0>
ub:teacherOf ?Y.
}
```

Figure. B.8: Re-written Federated Query 7 According To 1^{st} Adaptive Partition: Runs on PN_2

```
SELECT ?X, ?Y, ?Z FROM <http://localhost:8890/sparql/lubm>
WHERE{
?X rdf:type ub:Student .
?Y rdf:type ub:Department .
?X ub:emailAddress ?Z .
?X ub:memberOf ?Y .
?Y ub:subOrganizationOf <http://www.University0.edu>.}
```

Figure. B.9: Re-written Federated Query 8 According To 1^{st} Adaptive Partition: Runs on PN_4

```
SELECT ?X, ?Y, ?Z FROM <http://localhost:8890/sparql/lubm>
WHERE{
?X rdf:type ub:Student .
?Y rdf:type ub:Faculty .
?Z rdf:type ub:Course .
SERVICE<http://172.17.151.172:8890/sparql/lubm>{ ?X ub:advisor ?Y.}
?Y ub:teacherOf ?Z .
?X ub:takesCourse ?Z}
```

Figure. B.10: Re-written Federated Query 9 According To 1^{st} Adaptive Partition: Runs on PN_2

B.3 Rewriting Query Into A Federated Queries According To The 2^{nd} Adaptive Partition

```
SELECT ?X, ?Y1, ?Y2, ?Y3 FROM <http://localhost:8890/sparql/lubm>
WHERE{
?X rdf:type ub:Professor .
?X ub:emailAddress ?Y2 .
?X ub:worksFor <http://www.Department0.University0.edu> .
?X ub:name ?Y1 .
?X ub:telephone ?Y3}
```

Figure. B.11: Re-written Federated Query 4 According To 2^{nd} Adaptive Partition: Runs on PN_4

```
SELECT ?X, ?Y, ?Z FROM <http://localhost:8890/sparql/lubm>
WHERE{
?X rdf:type ub:Student .
?Y rdf:type ub:Department .
SERVICE <http://172.17.151.171:8890/sparql/lubm>{  ?X ub:emailAddress ?Z . }
?X ub:memberOf ?Y .
?Y ub:subOrganizationOf <http://www.University0.edu>.}
```

Figure. B.12: Re-written Federated Query 8 According To 2^{nd} Adaptive Partition: Runs on PN_4