A Scalable SPARQL Query Engine Over Large-Scale Compressed RDF Data Using SANSA

David Ohahimere Ibhaluobe

Matriculation number: 2776780

October 19, 2019

Master Thesis

Computer Science

Supervisors:

Prof. Dr. Jens Lehmann
Damien Graux
Gezim Sejdiu

INSTITUT FÜR INFORMATIK III

RHEINISCHE FRIEDRICH-WILHELMS-UNIVERSITÄT BONN

Declaration of Authorship

I, David Ohahimere Ibhaluobe, declare that this thesis, titled "A Scalable SPARQL Query Engine Over Large-Scale Compressed RDF Data Using SANSA", and the work presented in it are my own. I confirm that:

- This work was done wholly or mainly while in candidature for a research degree at this University.
- Where any part of this thesis has previously been submitted for a degree or any other qualification at this University or any other institution, this has been clearly stated.
- Where I have consulted the published work of others, this is always clearly attributed.
- Where I have quoted from the work of others, the source is always given. Except for such quotations, this thesis is entirely my own work. I have acknowledged all main sources of help.
- Where the thesis is based on work done by myself jointly with others, I have made clear exactly what was done by others and what I have contributed myself.

Signed: ____

Date: ____

Acknowledgements

First, I give glory to God for making this thesis a success. My profound gratitude goes to my esteem supervisors Dr. Damien Graux, Mr. Gezim Sejdiu and Prof. Dr. Jens Lehmann for their guidance and contributions towards the success of this project.
I am very grateful to my wife Agatha Ibhaluobe for her unwavering support, encouragement and love. Thank you for taking charge of our home and caring for our children while I concentrated to finish this thesis. Special thanks to my lovely children Ruth and Joshua for their complete understanding during those times that I was absent when they needed me.
I owe a lot to my parents and my in-law for their warm encouragement, love and support, I am indeed blessed to have you in my life.
I sincerely appreciate my friend and colleague Abakar Bouba who worked on the same project SANSA, for his guidance during this thesis.
Words fail me to sincerely thank you my brother from another mother Adonis Sheriff and his lovely wife Fatu, you have been a source of emotional, academic, psychological and financial support since I started this program. Thank you for offering your shoulder to be cried on when I needed to, I am so grateful.
To the one big Daddy in my home Ralf Uphoff and the entire Uphoff family, how can I thank you enough? Thank you for being a shoulder to lean on and a source of emotional, academic, psychological and financial support. To my brother Dr. Glenn Aliu and his family, I am sincerely grateful for your encouragements, prayers and support. Thank you for being a blessing to my whole family.
I lack words to thank the Wardenbach family, dear Miquette and Uwe, thank you for being you.
I am immensely grateful to all my friends (Dr. Isi and family, Dr. Kingsley, Dr. Peter, Mr. Isaiah, Dr. Bisi, just to mention a few) for their support during this thesis.
To my spiritual family at the American Protestant Church, the Gospel Ambassadors choir; you were all a blessing to me and my family. I couldn't have asked for more.
I am indeed very grateful; God bless you all.

Abbreviations

RDD	Resilient Distributed Datasets
SANSA	Semantic Analytics Stack
RDF	Resource Description Framework
SDA	Smart Data Analytics
SPARQL	SPARQL Protocol and RDF Query Language
HDT	Header Dictionary Triples
HDFS	Hadoop Distributed File System
W3C	World Wide Web Consortium
RDBMS	Relational Database Management System
LOD	Linked Open Data
HTTP	Hypertext Transfer Protocol
URI	Uniform Resource Identifier
SQL	Structured Query Language
IRI	Internationalized Resource Identifier
BGP	Basic Graph Pattern
XML	Extensible Markup Language
LUBM	Lehigh University Benchmark
JVM	Java Virtual Machine
API	Application Programming Interface
S,P,O	Subject, Predicate and Object
API	Application Programming Interface
IO	Input/Output

LUBM	Lehigh University Benchmark
BSBM	Berlin SPARQL Benchmark
MR	MapReduce
Hadoop	Highly Archived Distributed Object Oriented Programming

Contents

List of Tables

List of Figures

Listings

Abstract

The fast-growing rate of the web of data from various sources poses a storage and retrieval problem. These data are compiled in the form of statements; subject, predicate, and object, which invariable form an RDF graph. SPARQL Query Language is one of the query languages to query RDF data. Querying this RDF (Resource Description Framework) graph with millions of triples may pose a retrieval bottleneck. Consequently, optimizing disk space to store such a quantity of data is of optimal significance. Compressing this data will optimally save disk space for storing this large amount of data. Data compression technique is a way to achieve this. Data compression becomes a cost-effective way of saving disk space. This is why compression techniques are of utmost importance to enable easy retrieval of the RDF data.

The main focus of this thesis project is to build a scalable SPARQL Query Engine evaluator with SPARQL-to-SQL translator on the compressed RDF data using direct mapping technique; which can handle simple and complex SPARQL queries in SANSA [1] for effective retrieval. This compressed RDF data are in a dictionary ID format. Our system in SANSA will be able to query this dictionary value data and translate the SPARQL queries to SQL queries. We evaluate the scalability of the query engine by its capacity to scale big compressed datasets.

Keywords: RDF, Query-Engine, Large-Scale, SPARQL, SQL, Direct-Mapping, Sparklify, Compression, Scalability

Chapter 1

Introduction

As the Data Web continues to grow, the capacity to retrieve and exchange data through a Resource Description Framework, RDF, is becoming highly essential [2]. Performance and scalability are a major issue in this scenario, and their resolution is heavily related to the efficient storage and retrieval of semantic data [3]. Currently, scientists and businesses are increasingly interested in the RDF data format [2]. RDF is the standard language for the representation of information on the Semantic Web, the evolution of the World Wide Web [1] aimed at providing Well-founded infrastructure to publish, share and query structured data [4] and modelling information in the form of triples (subject, predicate and object) to allocate a single system for structuring and linking data. SPARQL is the W3C RDF query recommendation [2] [5]. It is a graphic-matching language, built on top of the triple models, i.e. RDF triples, in which each subject, predicate, or object can be a variable. This means that eight separate triple models are possible in SPARQL (i.e. variables are preceded by a symbol in the pattern ?): "(S,P,O), (S,?P,O), (S,P,?O), (S,?P,?O), (?S,P,O), (?S,?P,O), (?S,P,?O), and (?S,?P,?O)" [6]. SPARQL generates more complicated queries (generally referred to as Basic Graph Patterns, BGPs) by joining triple pattern sets, resulting in unnecessary intermediate outcomes processing for SPARQL queries. For example, we can use the SPARQL query to retrieve documents published by Albert Einstein from the local RDF in Listing 1 below:

Listing 1.1: A SPARQL Query to get all the written articles by Albert Einstein [7]

```
SELECT ?title WHERE {
? article  <hasTitle> ? title .
? article  <hasAuthor> ?author.
?author  <hasName> Albert Einstein.
}
```

The above Listing Listing 1.1 Shows a simple SPARQL query where the dot shows each of the joins. The entire query pattern can be seen in the RDF dataset as a graph pattern which requires matching. It may also include predicates that enhance the complexity of query assessment.

Typically, a Relational Database Management System (RDBMS) enables various databases,

[1] https://onlinelibrary.wiley.com/doi/full/10.1111/tgis.12496

[2] https://www.w3.org/TR/rdf-primer/

each with various tables with field columns and information rows. SQL (structured query language) is the language used to access the information in these tables [3]. Although compression techniques on RDF data reduce storage space by creating an N-triples dictionary of (S, P, O) in the form of IDs rather than string; which in turn makes it possible to write efficient queries to query the already compressed RDF data. For the compressed RDF data (dictionaries) to be scalable, the system should be able to manage a straightforward, effective and complicated SPARQL query for easy retrieval.
The relational databases are still the most used despite the advent of the semantic web due to its elevated efficiency in handling a large quantity of data through semantic filters, and it is, therefore, necessary to establish a link between the two heterogeneous structures in order to bridge the divide between them [8]. On this note lies why the goal of this thesis which is to design a SPARQL Query Engine with SPARQL-to-SQL paradigm over the compressed RDF data using direct mapping technique Which can manage SPARQL queries that are easy and complicated [9]. Our system should be capable of handling simple queries with features like Simple SELECT, SELECT with single WHERE Condition and query with Multiple FILTER and complex queries with features like SELECT with Nested FILTER (filter predicates like comparison operators: ($<=$, $<$, $=$, $>$, $>=$), logical operators: (!, &&, ||) and arithmetic operators: ($+$, $-$)) and FILTER functions (STRLEN, SUBSTR, STRENDS, CONTAINS, RAND, IN, NOT IN, STRBEFORE, STRAFTER, and REPLACE) [4], DISTINCT, GROUP_BY, OPTIONAL and aggregate functions: COUNT($*$) [5] and then convert the SPARQL query to SQL.

1.1 Questions

The question below has been set as a guideline to achieve this aim.

1. How Scalable is a SPARQL query engine (evaluator) over compressed data?
2. How efficient is SPARQL-to-SQL re-writer over compressed data?
3. How accurate is the SPARQL-to-SQL translator over compressed data?

1.2 Objectives

The objective of this thesis is to build a Query Engine using SPARQL-to-SQL paradigm over the compressed RDF data upon the Query Layer of SANSA Stack to enable faster and efficient retrieval of the compressed RDF data; which is the main objective of our system SANSA-Stack (Query Layer).

[3]https://medium.com/metaphorical-web/from-sql-tables-to-sparql-graphs-4aa2dda65a46
[4]https://en.wikibooks.org/wiki/SPARQL/Expressions_and_Functions
[5]https://github.com/SmartDataAnalytics/Sparqlify/supported-sparql-language-features

1.3 Thesis Structure

The rest of this chapter looks into the pre-process that gave way to the designing of our SPARQL query engine SPARQL-to-SQL on a compressed RDF data (dictionaries). The following chapters are organized as follows. Chapter two, which is the Background of study, focuses on the presentation of the terms and methods used in this thesis to familiarize the reader with the platform on which we are working. Chapter three Related Work shortly discusses the current work concerning SPARQL query engines and SPARQL-to-SQL translator and compression of RDF N-triple data. Chapter four Approach provides extensive data on how to implement the solution, and our system code sample is mentioned. Our application is evaluated in a distributed two nodes cluster in chapter five Evaluation. Chapter six Conclusions and Future Work concludes the report with a view to future research work by summarizing execution.

Chapter 2

Background

2.1 Semantic Web

The semantic web is the W3C's vision of a web of linked data [1]. It is a network of data connected in such a manner that machines can efficiently process on a worldwide scale [10, 11]. It was developed as an accepted structure that enables data to be shared and reused across system boundaries and allows individuals (e.g. people, organization, etc.) to develop such as web-based data stores, build vocabulary, write information processing laws, use RDF as a flexible information model and use ontology to portray data semantics. RDF and SPARQL are the technologies that powered linked data [12]. The semantic web can be described as an enhancement to the current web in which information has a well-defined meaning and therefore makes it easier for machines and humans to communicate; furthermore, it also enables the communication of information in a semantic web format and makes machines perceive hyperlinked information.

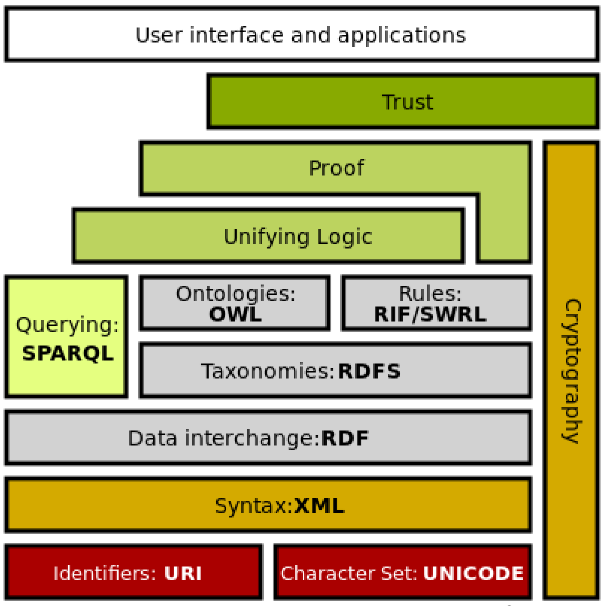


Figure 2.1: Semantic Web Architecture [2][13]

[1]https://www.w3.org/standards/semanticweb/

. We can see from figure 2.1 the involvement of data like RDF, OWL and XML.

2.1.1 RDF Model

At the core of an RDF lies a model to represent named properties and their corresponding values [3]. In other words, it has become an approach to model information or conceptualizes any domain to be executed on the web; via different data serialization formats and syntax notation. Below are some of the most popular serialization formats:

1. `N3` [4], which is a text format with advanced features beyond RDF.
2. `N-Quads` [5], which is a superset of N-Triples for serializing multiple RDF graphs.
3. `JSON-LD` [6], a JSON-based serialization.
4. `N-Triples` [7], which is a text format focusing on simple parsing.
5. `RDF/XML` [8], which is an XML-based syntax that was the first standard format for serializing RDF.
6. `RDF/JSON` [9], which is an alternative syntax for expressing RDF triples through the use of a simple JSON notation.
7. `Turtle` [10], which is a text format focusing on human readability.

Figure 2.2 below show a simple example of an RDF triple:

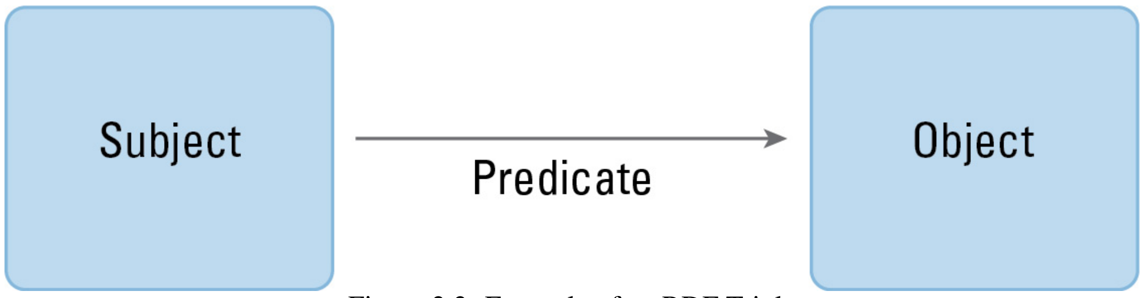


Figure 2.2: Example of an RDF Triple

As illustrated in Figure 2.2 above; subjects and objects are depicted as nodes while predicates as arcs [14].

N-Triples `N-Triples`[11] is a simple text and line-based format for RDF graph encoding. The general form of each triple can be described as $<Subject><Predicate><Object>$, which is separated by white space and terminated by '.' after each triple.

1. The `subject` contains URIs and blank nodes.

[2]https://en.wikipedia.org/wiki/File:Semantic_web_stack.svg
[3]https://www.w3.org/TR/WD-rdf-syntax-971002/
[4]http://www.w3.org/DesignIssues/Notation3.html
[5]https://www.w3.org/TR/n-quads/
[6]https://www.w3.org/TR/json-ld/
[7]https://www.w3.org/TR/n-triples/
[8]https://www.w3.org/TR/rdf-syntax-grammar/
[9]http://www.w3.org/blog/SW/2011/09/13/the-state-of-rdf-and-json/
[10]https://www.w3.org/TR/turtle/
[11]https://www.w3.org/TR/n-triples/

2. The `predicate` contains URIs.
3. The `object` contains URIs, blank nodes, and or literals

We describe URIs, blank nodes, and literals as follows:

1. `URIs` provide an easy and extensible method for resource identification. This can be used for anything – examples may include places, people or even animals.
2. `Blank nodes or for anonymous resources` that are not assigned. For example, no literal or no URI is given.
3. Strings, Booleans, data, and more are values that are identified through `literals`.

2.1.2 SPARQL

SPARQL [6] is a query language used in semantic web to retrieve structured, semi-structured and unstructured data. SPARQL can query RDF data through graph pattern with their conjunctions and disjunctions. The findings of the SPARQL query can be either result sets or RDF graphs [12]. For instance, one might consider the following example of query RDF data in a SPARQL base on a chain pattern from a LUBM benchmark [15]:

Listing 2.1: Example SPARQL Graph Query [16] in figure 2.3

```
SELECT * WHERE {
?x   advisor   ?y  .
?y   teacherOf   ?z  .
?z   type  Course  }
```

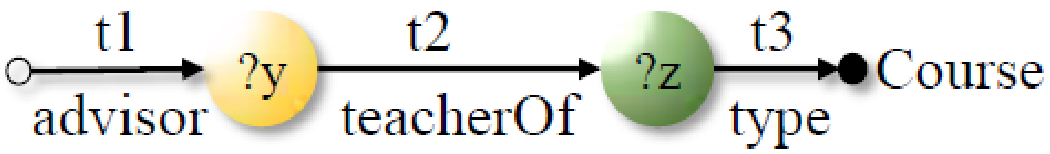


Figure 2.3: Query RDF Data in SPARQL [17]

2.2 SQL

SQL is a unique programming language that is designed for database management and is used by a variety of applications and organizations [13]. It is especially useful when handling structured data where there are relations between individual data entities/variables and based initially on relational algebra and tuple relational calculus. SQL includes many different kinds of

[12] https://www.w3.org/TR/rdf-sparql-query/
[13] https://www.khanacademy.org/computing/computer-programming/sql

statements which can informally be classified as subsidiary languages: Data Query Language (DQL), Data Definition Language (DDL) and Data Manipulation Language (DCL) [14]. The SQL range comprises information query, information manipulation (insert, update and delete), information definition (schema development and alteration) and data access control [15].

2.3 Jena

Jena [16] "is a Java framework for the development of Semantic Web application. It provides a programmatic environment for RDF, RDFS, OWL, SPARQL and includes a rule-based inference engine" [18]. Jena offers APIs for multiple format RDF parsing and RDF writing in different formats are supported. The Jena package also provides ARQ, the query engine that implements SPARQL 1.1 Specifications.

```
bin/sparql --data=/path/to/rdf/data.rdf --query=/path/to/query/file.rq
```

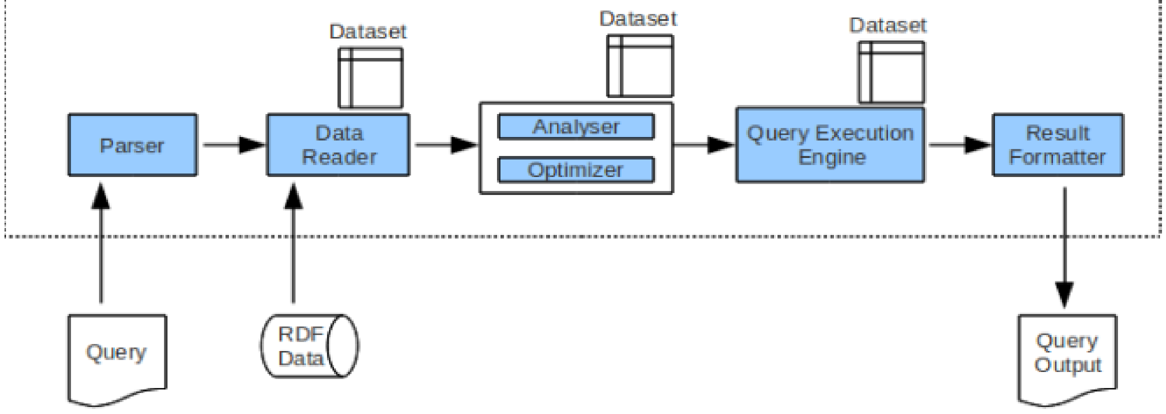


Figure 2.4: The Architecture Overview [19]

2.3.1 Jena ARQ Parser

Jena provides the argument parser that parses the interface arguments and sets the rest of the `context` program for execution [17]. Jena supplies an argument parser [19] which passes through the interface arguments and set the rest as a `context` object. The global object is known as a `context` that can be used by other classes to set and read data on the context level.
The `QueryFactory` class parses, validates and populates the `Query` object. Encapsulation of all properties and methods about a query structure is what is known as the `Query` class. A SPARQL query is what is known as the internal representation of a `Query`. The Query holds the information of a projected variable list, basic graph pattern and the query operators like

[14] https://en.wikipedia.org/wiki/SQL
[15] https://howlingpixel.com/i-en/C-O-U-N-T-R-Y
[16] https://jena.apache.org/
[17] https://jena.apache.org/documentation/javadoc/arq/org/apache/jena/riot/RDFParser.html

`DISTINCT`, `FILTER`, `LIMIT`, `OPTIONAL`, `ORDER BY` and `GROUP BY` which are features implemented in our SPARQL-to-SQL query engine. We used the Jena ARQ OpVisitor interface for parsing SPARQL to SQL, which gives us more options to walk through SPARQL query and do not need to force-parse the SPARQL queries in the case of manual `String Manipulation`. It saves us the hassle of string manipulation for parsing SPARQL to SQL [19].

2.4 Big Data

As the world continues in its fast pace of digitization, gobs of structured and unstructured data are generated every single day, which are later analyzed for business purposes. This data are generated by all kinds of devices such as the Internet, sensors, social networks, mobile devices, computer simulations, and satellites [20, 14]. The enormous volumes of these datasets are what is known as big data. This data is so massive and complicated that traditional data-processing systems cannot deal with them [18] [14]. Many researchers have suggested techniques in recent years for reducing data volume when it is stored.
Big Data goes beyond storage capabilities and processing. Hence, Big Data is define as `3Vs` [19]: volume of data, variety of data, and velocity of data.

1. `Volume of data:` Data is gathered from organizations or companies through business transactions, social media, information from sensors, airlines, etc.
2. `Variety of data:` Data streams have unparalleled speed and enhanced in a timely way. Sensors and intelligent metering drive the need to deal with information overflow in real-time operations.
3. `Velocity of data:` Data is discovered in many distinct forms and can be classified based on structures such as numeric or unstructured data such as text-based papers, emails, videos, audio files, financial transactions, etc.

The storage posed a big problem; however, new technologies like Hadoop MR and Apache Spark has reduced the issues [21].

[18] `https://elysiumpro.in/big-data-analytics-projects/`
[19] `https://intellipaat.com/tutorial/hadoop-tutorial/big-data-overview/`

Hadoop [20] is an open-source framework technology that helps to store, access and obtain vital resources from massively distributed big data files of data over many computer systems at low cost, with an extraordinary level of fault-tolerance and significant scalability.
Features of Hadoop Include:

1. `Distributed.` A Hadoop cluster consists of several connected machines together.
2. `Scalable.` In order to process data fast, new machines can be added.
3. `Fault tolerant.` If any machine fails, since many machines work together, the machine can be replaced.
4. `Open Source.` Apache Hadoop is an open-source project [21]. This means that its code can be modified under business requirements. We used the Hadoop Distributed File System (HDFS) for storage and MapReduce for data processing [22].

The Figure 2.5 illustrates the parts of the Hadoop architecture; its structure of the HDFS and the Map-Reduce. The HDFS is used to store and process big datasets, while MapReduce is a way to split a computation job into a distributed collection of files. A Hadoop cluster consists of a single master node and several slave nodes; a data node is included in the master node. The name node, the job tracker and the task tracker; the slave node acts as a data node and a task tracker; and the work tracker manages the work schedule [22].

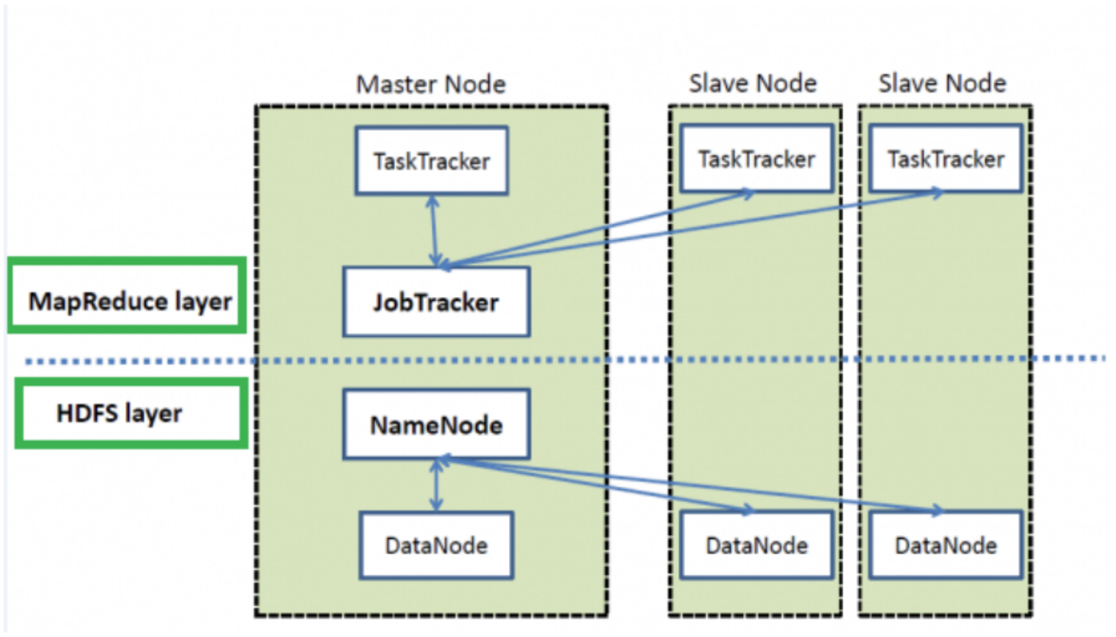


Figure 2.5: Hadoop Architectures [14]

HDFS

A distributed file system for commodity hardware execution is the Hadoop File System (HDFS) [22]. It is used for storing and processing of massive datasets with a cluster (that is, a group of machines in a LAN) of commodity hardware. It differs from others because it is

[20] `https://hadoop.apache.org/`
[21] `https://data-flair.training/blogs/features-of-hadoop-and-design-principles/`
[22] `http://web.mit.edu/mriap/hadoop/hadoop-0.13.1/docs/hdfs_design.pdf`

exceptionally fault-tolerant, and can sustain itself on cheaper hardware. Moreover, it is suitable for an application with substantial datasets. HDFS uses, by default, blocks of data with a size of 128 MB. Each of these blocks is replicated three times. Multiple versions of a block prevent data loss owing to a specific node failure.

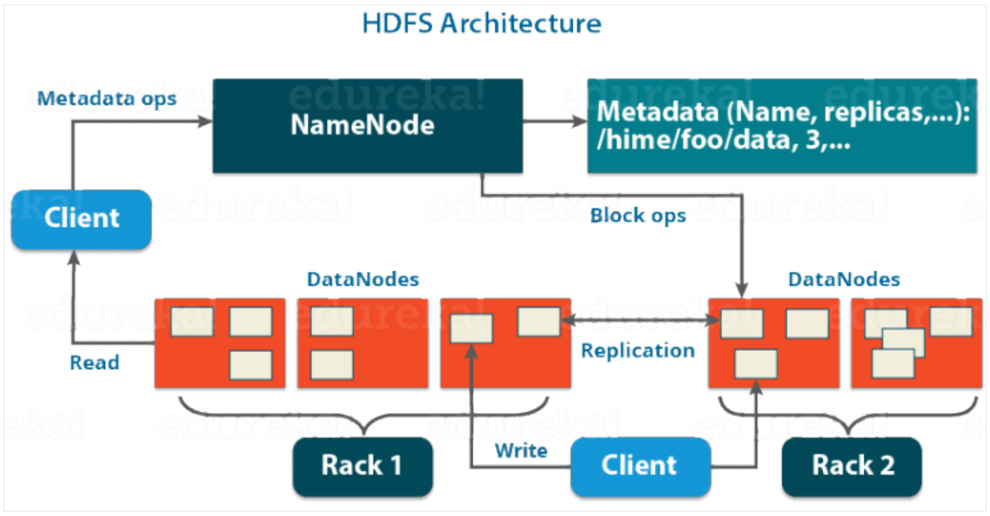


Figure 2.6: HDFS Architecture [23, 14]

Figure 2.6, describes how replications are achieved when the size of the block of data exceeds the maximum size (which is 128 MB). HDFS has a larger block size to bring down the amount of time required to read the complete file. During processing, smaller blocks provide more parallelism. When multiple copies of a block are available, data loss due to node failure is prevented. HDFS features below [24, 14]:

1. `NFS access.` By using this feature, HDFS can be mounted as part of the local file system, and users can upload, browse, and download data on it.
2. `High availability.` This feature is done by creating a standby Name Node.
3. `Data integrity.` When blocks are stored on HDFS, computed checksums are stored on the data nodes as well. Data is verified against the checksum.
4. `Caching.` Caching of blocks on data nodes is used for high performance. Data nodes cache the blocks in an off-heap cache.
5. `Data encryption.` HDFS encrypts data at rest once enabled. Data encryption and decryption happen automatically without any changes to the application code.
6. `HDFS re-balancing.` The HDFS re-balancing feature re-balances the data uniformly across all data nodes in the cluster.

MapReduce [25] is a programming model that provides an easy way to parallel complex tasks. MapReduce is influenced by a functional programming language that provides maps and reduces primitives. MapReduce has an environment that handles the issue of task allocation, fault tolerance, location of information over a distributed file system and provides an abstraction to achieve programming logic in Map and Reduce techniques [19]. This model is similar to a parallelization split, and aggregation, where a task to be performed on a dataset, is managed by running the job on a data split block in parallel and then aggregating the results of all functions to provide the final solution. The map resembles the split phase, and the reduce resembles the aggregation. Hadoop can run MapReduce programs in a variety of languages, such as Python, Ruby, Java. MapReduce is split into the following two phases:

1. `The map phase:` Is where data is processed for the first time, which is the time when all the complex logic and business rules are specified.
2. `The reduce phase:` Known as the second part of processing, here processing such as summation is specified. Each phase has (key, value) [23] pair; the first and the second phase are composed of an input and output(I/O). We can define our map function and reduce the function.

Listing 2.2: Map Reduce concept

```
Map
(k1, v1)   > l i s t (k2, v2 )

Reduce
(k2, l i s t ( v2 ) )   > l i s t (k3 , v3 )
```

From the listing 2.2 above, the Map and Reduce functions have rigid agreements. The map receives an input key / value pair and generates an intermediate key / value pair list [24]. The MapReduce framework ensures that all keys resulting from the map stage output are grouped and given as input to the Reduce function, which recognizes the key and the value list as input, and lists key/value pairs as output. The MapReduce framework automatically parallelizes programs written in this style. MapReduce environment framework does not expect any unique parallel cluster; a simple cluster of commodity machines can be used to create a MapReduce cluster [19].

[23]https://en.wikipedia.org/wiki/Attribute-value_pair
[24]https://dataweb.infor.uva.es/projects/hdt-mr/

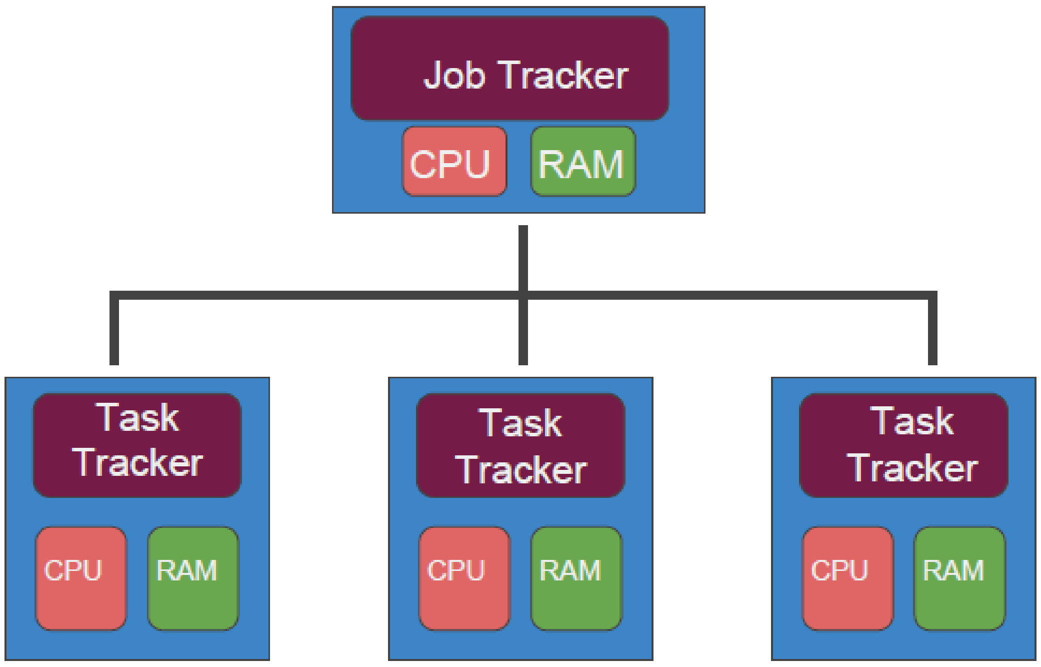


Figure 2.7: Architecture HDFS [25]

Figure 2.7 shows how the job tracker sends code for the Task Tracker to run, and then how CPU and memory are assigned to the tasks tracker and therefore, the worker nodes are monitored.

2.5 Data Compression

Data compression reduces the number of bits necessary for representing the data [26]. Data compression is the process of modifying, encoding or converting the data structure of large datasets; such that the data consumes less disk or memory space. It can be used in all formats of data (text, pictures, sound and video). There are two types of techniques [26, 14] in compression: lossless and lossy compression.

[25]https://en.wikipedia.org/wiki/Attribute-value_pair
[26]https://searchstorage.techtarget.com/definition/compression

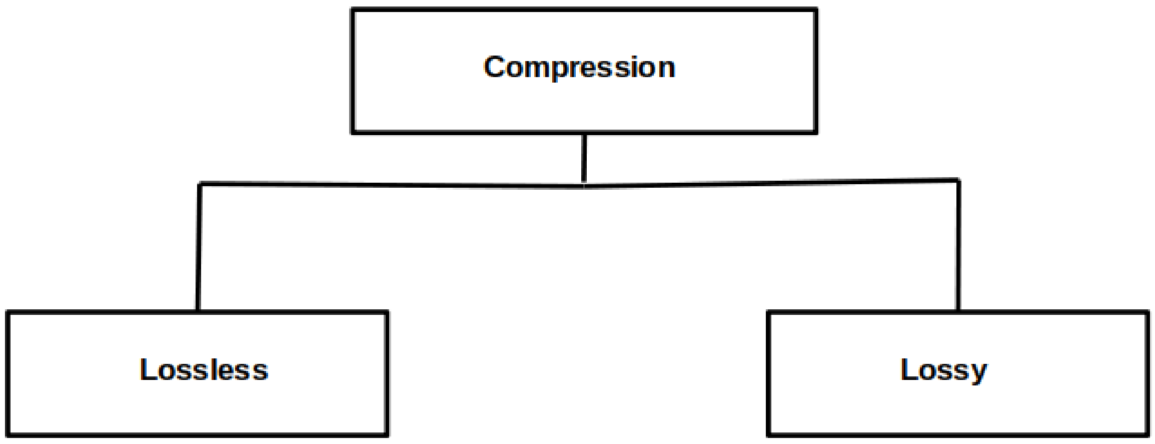


Figure 2.8: Types of Data Compression

1. `Lossless compression` [26], uses compressed data to recreate the original data using data compression algorithms. The compressed data is the same as the previous version before compression, i.e. bit for bit, no data loss. Lossless compression is frequently used to compress "executable code, text files and numeric data, as programs that process such data do not tolerate error in the data" [27].
2. `Lossy compression` is also known as compression that is irreversible [26, 14] where the original data is thrown away, and an incorrect approximation is used during data encoding. Precisely, it is used when working with images, videos or audio files.

2.6 HDT

HDT (Header, Dictionary, Triples) [27] is a robust RDF data structure and binary serialization model that keeps large datasets compressed to save room while preserving search and browsing activities without prior decompression [28]; making it an optimal model for storing and sharing RDF datasets on the Web.

Below are a few facts about HDT:

1. `The file size is smaller` than other RDF serialization formats, which implies less cost to the supplier of bandwidth, but also less time to download for users.
2. `The HDT file is indexed already`. Users of RDF dump want to do something useful about the data. By using HDT, the file will be downloaded and browsed/questing in a reasonable amount of time rather than spending time using parsing and indexing tools they never remember setting up and tuning.

[27]`http://ecomputernotes.com/computer-graphics/basic-of-computer-graphics/data-compression`

[28]`http:/www.rdfhdt.org/what-is-hdt/`

3. `High performance querying`. The bottleneck database is usually slow access to the disk. HDT's internal compression techniques make it possible to store most of the information (or even the entire dataset) n the main memory, which is several orders of magnitude faster than the disk. [27].
4. `Highly concurrent`. HDT is read-only, and queries per second can be displayed with different threads.

2.6.1 Dictionary

The main objective of the dictionary is to contribute to compactness by assigning a unique dataset ID to each component [27, 14]. Therefore, using the dictionary implies two principal and minimum operations [29]:

(a) `locate(element)`: returns the distinctive identifier of the element when it appears in the dictionary.

(b) `extract(id)`: returns the component ID in the dictionary if it occurs.

The dictionary is divided into sections based on whether the term is used in the subject, predicate, or object roles [27, 14]. In semantic web data, however, it is quite common that the URI appears in one of the triple as a subject and another triple as an object. We can use 4-section in collecting the triples called shared subject-object to avoid repeating these terms twice in subjects and sections of objects [27, 14].

Figure 2.9 below shows the 4-section dictionary operation and how the IDs of the corresponding terms are assigned. Each section is lexicographically sorted, and for each term, correlative IDs are assigned from 1 to n [27]. It should be a noted that the shared Subject-Object part uses a lower range of IDs for subjects and objects; e.g. if there exist m terms that are interchangeably played as subject and object, all respective IDs x such that x < = m belong to this shared section [27].

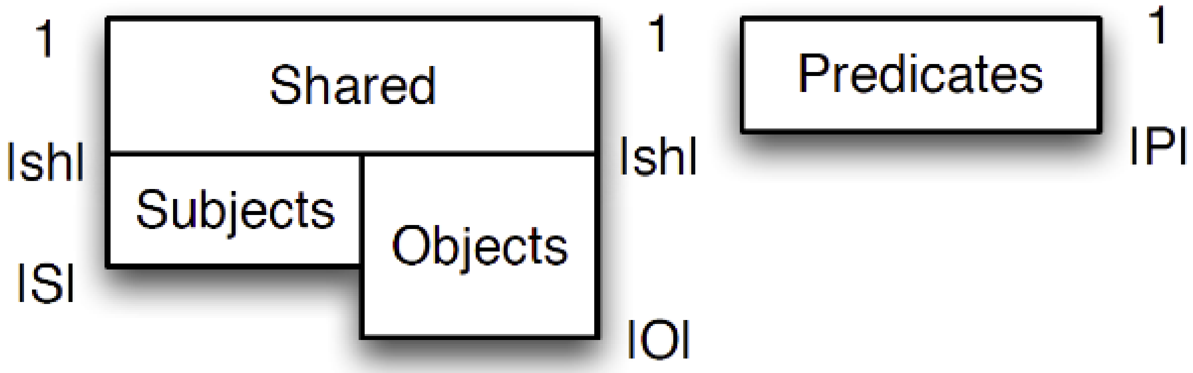


Figure 2.9: Organization of HDT dictionaries in four parts [27]

HDT allows well-defined techniques of dictionary representation [30]. Catalogue of terms can be managed by everyone in different ways as quickly as they perform the compulsory

[29] http://www.rdfhdt.org/technical-specification/
[30] http://www.rdfhdt.org/technical-specification/

locate/extract operations. They can also initiate additional enhanced activities such as full-text searches, prefix searches, or even periodic search expression.

2.6.2 Triples

HDT recommends a term known as BitmapTriples (BT); Triple encoding, which arranges the data in such a way that uses the advantage of graph redundancy to retain the data compactly. Also, this encoding can be easily mapped into a data structure that allows performing basic retrieval operations [27].

BT requires triples, such as subject-predicate-object (SPO), to be sorted in a specific order. BT can manage all triple orders [27].

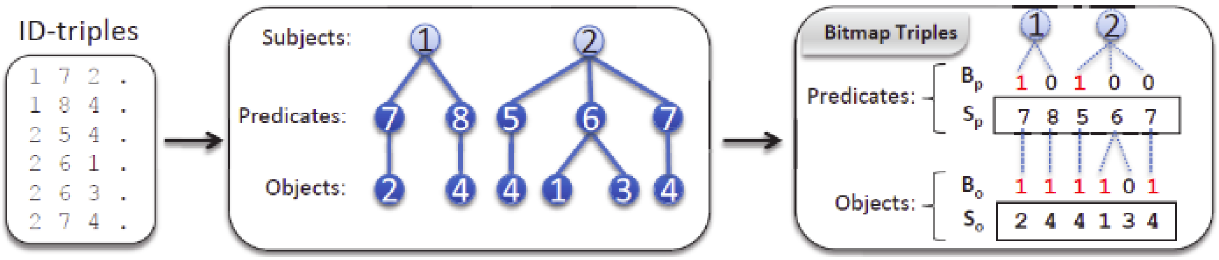


Figure 2.10: Description of Bitmap Triples.

From the Figure 2.10 Therefore, the very first tree represents all triples rooted in the subject as 1; the second tree reflects all triples rooted in the subject as two and so on [27]. Each tree consists of three stages: the root represents the subject, the second stage lists all predicates concerning the subject, and lastly, the leaves represent all objects for their path (subject, predicate). Predicate stages and concentrations of objects are also sorted [27].

Then the model of the BT forest is physically encoded layer by layer. At the subject top layer, which is the correlation list of one to the total number of corresponding assigned IDS subject in the dictionary and the triple SPO order. Therefore, the subjects do not need to be encoded and remain implicit, that is; the first evidence of spatial saving of space [27].

“The predicate layer is stored using I a series of predicate IDs (Sp); ii) a bit sequence (Bp) comprising one bit per component in Sp: A 1 bit suggests that the predicate is the first child of its parent subject in the tree; the remaining siblings are labelled with 0. This enables knowing which subject is associated with each predicate by counting the amount of 1’s in the bitmap up to that place, or even finding the range of predicates connected with the $n-th$ subject by placing the $n-th$ and $(n+1)-th$ 1 in the bitmap” [27] [31]. Similarly, a series of object IDs (So) together with a bit sequence (Bo) are used to store the object layer. Each 1 bit reveals the first object of each parent’s predicate, permitting traversals up and down in the tree [27].

[31]http://www.rdfhdt.org/technical-specification/

2.6.3 Querying HDT-encoded datasets

The SPARQL [28] [32] triple patterns are query atoms for fundamental RDF retrieval. That is, all triples comprising the model (s;p;o) (where s, p and o may be variable) must be acquired directly from the encoding of the triples. Once mapped to the main memory, there is a possibility to access an HDT-encoded dataset directly. However, it does not immediately support all kinds of triple patterns in this framework; it is limited to SPO, SP?, S?? and ??? queries [27]. As a result, once the HDT-encoded data is loaded into the memory hierarchy, the representation is slightly enriched with additional succinct data structures to support the existing triple patterns to be effectively resolved. HDT-FoQ: HDT Querying Focused is on the final fully queriable representation [27].

2.7 Scala and Spark

2.7.1 Scala

Scala is a programming language for general purposes [33]. "It was designed by Martin Odersky in the early 2000s at the Ecole Polytechnique Federale de Lausanne (EPFL), in Switzerland [34]. Scala source code is designed to be compiled into Java bytecode to run on a Java virtual machine (JVM); " Java libraries can be used straight in Scala [29]. Unlike Java, Scala has many functional programming features [35]. In the latest years, demand for Scala has increased dramatically due to Apache Spark [36]. Scala is the world's fourth most demanded programming language. Some of the biggest businesses like LinkedIn, Twitter, FourSquare and more use it extensively.

2.7.2 Spark

Apache Spark [30] is one of the recent techniques to handle big data rapidly and easily. It is an open-source project on Apache that was first published in February 2013 and exploded in popularity owing to its velocity and ease of use. It was developed at UC Berkeley's AMPLab [24]. Spark is considered today to be a flexible alternative to MapReduce[14].

Spark can use the data stored in various formats:

- HDFS
- Cassandra
- Other formats

Some of the Spark features are [24, 14]:

(a) `Easy development.` Multiple native APIs such as Java, Scala, R, and Python.

[32] https://www.w3.org/TR/rdf-sparql-query/

[33] https://www.scala-lang.org/

[34] https://en.wikipedia.org/wiki/Martin_Odersky

[35] https://en.wikipedia.org/wiki/Java_bytecode

[36] https://www.infoworld.com/article/3216144/spark/the-rise-and-predominance-of-apache-spark.html

(b) `Optimized performance.` Caching, optimized shuffle, and catalyst optimizer.

(c) `High-level APIs.` Data frames, data sets, and data sources APIs.

SparkSQL

Spark SQL is Apache Spark's module for working with structured data [37]. Using SQL or a familiar DataFrame API [38], SparkSQL allows for query structured information within Spark programs. Usable in Scala, Python, R and Java. Executing SQL queries is one use of SparkSQL.

Spark RDDs

Resilient distributed datasets (RDDs) are a distributed abstraction of memory that enables programmers to conduct fault-tolerant in-memory computations on a huge cluster. It allows effective data reuse in a wide spectrum of applications and fault-tolerant, parallel data structures. Spark RDDs are generated by transforming data into stable storage using data stream operators such as map, group-by, filter; they can also be stored in memory through parallel operations [14].

- `Resilient.` Resilient means that if data in memory is lost, it can be recreated or recomputed.
- `Distributed.` It distributes data across clusters.
- `Datasets.` Initial data can come from a file.

There are two kinds of activities that the RDD supports (it is also used for any computational data processing)[14]:

(a) `Transformation.` This implies changing a dataset from one dataset to another dataset. Any alteration of data leads to a transformation: for instance, by multiplying numbers, adding a number, adding two distinct datasets, or joining two datasets. The transformation utilizes the following functionality [14]:
  - `map()`, is used to convert the dataset into a different one based on our logic. For example, if we want to multiply a number by 2, we can use RDD.map$(x > x2)$.
  - `flat map()` is used for our dataset.
  - `filter()` should filter only the desired element.
  - `distinct()` should give only the distinct elements from the dataset.
  - `union()` should join two datasets.

(b) `Action.` This mean doing computation on our dataset.
  - First
  - Collect
  - Count
  - Take

[37] https://spark.apache.org/sql/

[38] https://spark.apache.org/docs/latest/sql-programming-guide.html

Four primary features of a resilient distributed dataset (RDD) are [14]:

(a) A distributed information collection.
(b) Fault-tolerant.
(c) It can handle parallel operation.
(d) It can use many data sources.

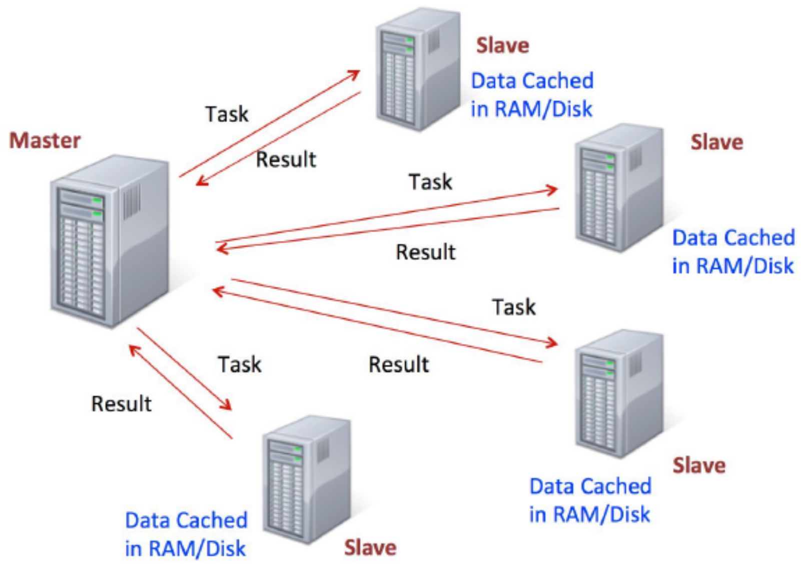


Figure 2.11: Overview of Spark RDDs [31, 14]

2.8 Local Versus Distributed Systems

Local Systems

A local process utilizes a single cluster's computing resources; this implies that it will take more time to process information if a large dataset is operation [14].

Distributed System

The computer resources can be accessed on a variety of networked computers, which is comparable to a local system with incredibly fast processing. Hadoop distributes a dataset through various clusters or computers. Distributed machines also benefit from easy scaling (more machines can be added). They also include fault tolerance, which means that if one machine fails, the entire network can continue to function [14]. However, if a computer comes down in a local cluster, the entire system will go down [14].

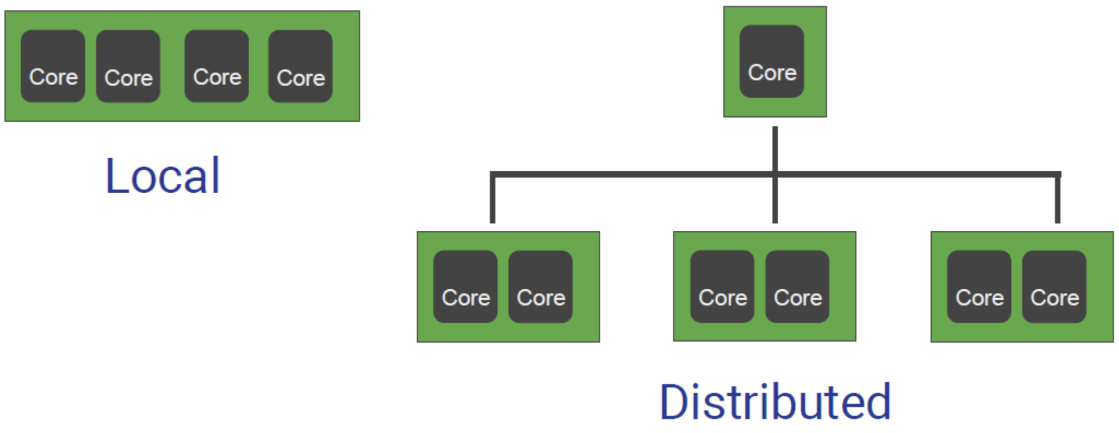


Figure 2.12: View of the Local and Distributed Systems [31, 14]

2.9 Hadoop Versus Spark

Both `Hadoop` and `Spark` are big data frameworks. We found that when processing a big dataset, Spark is faster than Hadoop. However, a word-count instance can also be compared with the Scala programming language by using Hadoop and Spark [14].

Table 2.1: Apache Spark Versus Hadoop [32, 14]

Apache Spark Versus Hadoop	
Apache Spark	**Hadoop**
It is easier to program and [33] has no necessity for abstraction.	Hard to program and needs abstractions.
Written in Scala.	Written in Java.
Developers in the same cluster can execute streaming, batch processing, and machine learning.	The generation of reports is used here to help find answers to historical queries [33].
Interactive mode built-in [33].	No interactive mode included, except for tools like Pig and Hive [33].
Programmers can edit the information via Spark Streaming in real time [33].	Enable a batch of saved data to be processed [33].

Table 2.1 explains the fundamental distinction between Apache Spark and Hadoop. Figure 2.13 below provide a perspective of both frameworks. The memory-based computation in disk and memory should be explained here.

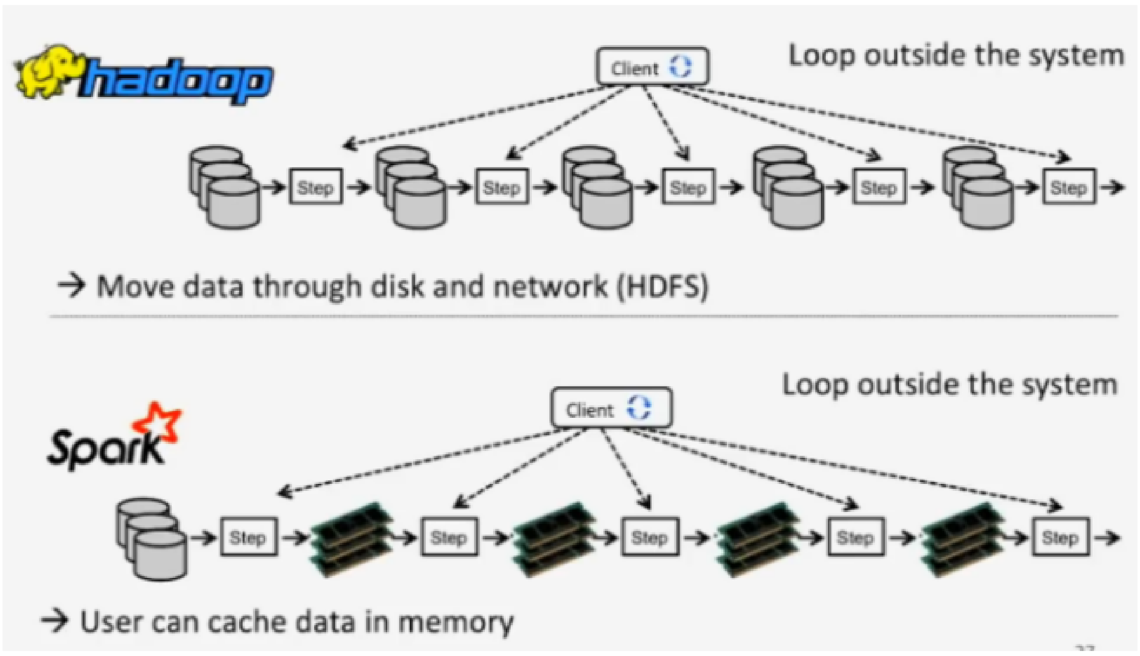


Figure 2.13: View of Hadoop Versus Spark [34, 14]

Chapter 3

Related Work

The size of RDF data increases every second as data providers publish their data in RDF format. Compression of RDF data plays a huge role in big data applications [14]. In this chapter, we give an overview of the work related to our work. There are already existing approaches for SPARQL engine over RDF data, but few works are done over compressed RDF data like HDT MapReduce [35], Fast Search of Semantic Data on Compressed Indexes [36]). Here we are going to look at some of these work done concerning our work.

3.1 SPARQL Query Engine Techniques and Compression Techniques

3.1.1 Header Dictionary Triple

HDT [27] is a robust RDF data structure and binary serialization model that keeps large datasets compressed to save room while preserving search and browsing activities without prior decompression [1]; making it an optimal model for storing and sharing RDF datasets on the Web. Our approach in designing the SPARQL-to-SQL query engine is drawn from HDT technology which enables us to build projection fields on the compressed index value (subject hdt, predicate hdt and object hdt).

3.1.2 HDT-MR

HDT-MapReduce [35] enhances the HDT-java library by implementing MapReduce as a computational framework for large HDT serialization. HDT-MR works in linear time with dataset size and has proven to be capable of serializing datasets up to 4.42 billion triples, maintaining HDT compression and retrieval functionality [2]. We are mentioning this approach for robust information reason, but it is outside the scope of this work.

[1]http:/www.rdfhdt.org/what-is-hdt/

[2]https://github.com/rdfhdt/hdt-mr

3.1.3 Fast Search of Semantic Data on Compressed Indexes

This is a `trie-based index layout techniques` [3] reducing its representation room to improve efficiency [36]. In order to support (symmetrically) all three selection models with one or two wildcard symbols, the index materializes three distinct triple permutations. By leveraging well-engineered compression methods, demonstrate that this structure is already as compact as the literature's most space-efficient competitor, and on average for all selection models is 2 ÷ 4X faster [36].

3.1.4 RDFMatView: "Indexing RDF Data for SPARQL Queries"

RDFMatView [7] is based on materialized queries which are offline. RDFMatView SPARQL query indexes [4], a cost model for evaluating their potential impact on query performance and a rewriting algorithm for using SPARQL query indexes. It also creates and compares different techniques for integrating these indexes into an existing SPARQL query engine. Preliminary findings indicate that RDFMatView strategy can drastically reduce the processing time of queries compared to normal query processing [7].

3.1.5 Sesame

Sesame [12]: RDF and RDF Schema Generic Architecture is an architecture that is used in both RDF and RDF schema to store and explicitly query large volumes of metadata effectively. Sesame design and execution are autonomous of any specific storage device. Therefore, sesame can be applied in several storage systems, such as relational databases, triple stores or object-oriented databases, without changing the query engine or other functional modules. Sesame supports competitiveness control, autonomous transport of RDF and RDFS data, as well as an RQL query engine, an RDF query language that provides RDF semantics native help [12].

3.1.6 RDF-3X

RDF-3X [37]: "a RISC-style Engine for RDF" is SPARQL Implementation that accomplishes exceptional performance by seeking RISC-style architecture with streamlined architecture and very well-designed, purist data structures and activities. RDF-3X's capture points are:

(a) A generic RDF storage and indexing solution that removes the need for tuning to the physical design.

(b) A powerful yet easy query processor, which uses fast fusion, joins to the greatest extent possible, and

(c) A query optimizer to select optimal join orders using a cost model based on a statistical synopsis for complete join paths.

RDF-3X presents index compression to decrease Hexastore's use of space. RDF-3X produces its indexes with three rows over a single table and then stores them in a compressed cluster [14].

[3] `https://arxiv.org/abs/1904.07619`

[4] `https://edoc.hu-berlin.de/handle/18452/3141`

3.1.7 Dcomp

`Dcomp` [38, 14]: approach target a common attribute where there is an enormous amount of duplicates. Then it focuses on eliminating redundant triples. To store a distinctive value, Dcomp generates a subject, predicate, and object dictionary. "All triples in the dataset can then be rewritten by changing the terms with their matching ID. Hence, the original dataset is now modelled over the dictionary created and the resultant ID-triples representation". Dictionaries are managed in a dataset according to their role:

- A common `S`, and `O` organize all conditions in the dataset that perform subject roles and object roles and mapped to the range $[1, |SO|]$.
- `S` arranges all subjects which does not play the part of an object and are mapped to the range $[|SO|+1, |SO|+|S|]$.
- `O` arranges all objects which does not play a role in the subject and are mapped to the range $[|SO|+1, |SO|+|O|]$.
- `P` predicates are all mapped to $[1, |P|]$.

3.1.8 k2-Triples

`k2-Triples` [6, 14]: This is a method used for compressed, self-indexed constructions that were initially developed for web graphs, proposed a K2 model depicting pairs in sparse binary matrices (subject, object) that are efficiently indexed in compressed space through k2-tree structures achieving the most compressed representations in terms of the cutting-edge baseline.

3.1.9 Huffman Coding

`The Huffman coding` 3.1: is one of the most common methods that is used to remove redundant data, targeting each character's count frequency input. The primary task of this method is to substitute or assign shortcodes to a more commonly occurring symbol (i.e. a redundant string or symbol) and longer codes where the occurrence is less frequent. One instance we can consider is a dataset "that uses only the characters A, B, C, D & E" [39]. Each character must have a weight depending on how often it is used before a pattern is assigned.
The example on figure 3.1 explains this approach.

Character	A	B	C	D	E
Frequency	17	12	12	27	32

Table 3.1: Huffman Coding [39]

3.1.10 Lempel-Ziv Compression

`The Lempel-Ziv` [39, 14] to save data space, the method depends on recurring patterns which are based on generating an indexed dictionary utilizing a string of compressed symbols. The lowest sub-string is obtained from the remaining uncompressed version under the algorithm unless it can be discovered in the dictionary. The algorithm then keeps a copy of this sub-string in the dictionary, which makes it a new entry and gives it a value within the index.

3.1.11 RLE

`RLE` [39, 14]: This is a method that provides excellent data compression with numerous runs of the same value. It substitutes sequences of the same symbols from a dataset. The primary logic behind this method is to replace repeating items with one symbol occurrence, followed by the number of occurrences.

Chapter 4

Approach

This thesis project follows the work of Abakar Bouba [14] thesis work; who worked on “RDF Data Compression Techniques in a Highly Distributed Context”. Before we talk about our implementation; which is an extension of his contribution, we would like to talk about his work and the framework where both projects are running on (SANSA-Stack [1]) to give a broad overview linking to our implementation.

4.1 SANSA-Stack Architecture

4.1.1 Introduction

The work here uses SANSA [1], an open source [1] *engine for information flow processing* to perform distributing computation on a enormous scale across RDF datasets. It offers distribution of data, communication and faults tolerance to manipulate huge RDF graphs and apply machine learning techniques to scale information. SANSA’s main idea is to combine the distributed computer frameworks in Spark and Flink with the semantic technology stack. [2] [1].

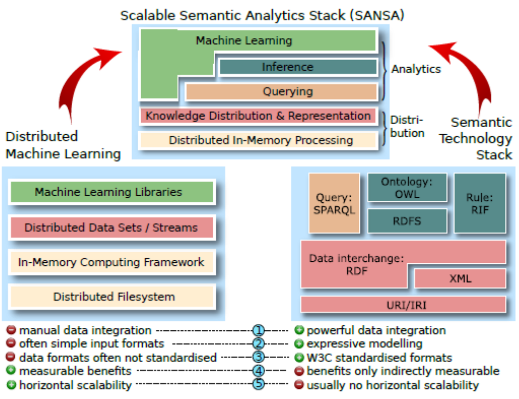

Scalable Semantic Analytics Stack (SANSA)
Machine Learning
Inference
Querying
Analytics
Knowledge Distribution & Representation
Distributed In-Memory Processing
Distri-bution
Distributed Machine Learning
Semantic Technology Stack
Machine Learning Libraries
Distributed Data Sets / Streams
In-Memory Computing Framework
Distributed Filesystem
Query: SPARQL
Ontology: OWL
RDFS
Rule: RIF
Data interchange: RDF
XML
URI/IRI
manual data integration
often simple input formats
data formats often not standardised
measurable benefits
horizontal scalability
1
2
3
4
5
powerful data integration
expressive modelling
W3C standardised formats
benefits only indirectly measurable
usually no horizontal scalability

Figure 4.1: A SANSA Framework overview combining Distributed Analytics (left) and Semantic Technologies (right) into a Scalable Semantic Analytics Stack (top of the diagram) [1]

[1] https://github.com/SANSA-Stack

[2] http://sansa-stack.net/

Our implementation, which we called Scalable SPARQL query engine over large-scale compressed RDF, is deployed over a SANSA-Stack project. There are currently five layers in the SANSA Stack: RDF, Query, Inference, Machine Learning, and OWL layer. We used the query for this project [3]. However, the RDF Compression [14] which is also deployed on SANSA project used the RDF [4].

SANSA-RDF
SANSA-RDF [5], holds the bottom layer of the SANSA Stack, a read / write data layer, by enabling users to read and write n-triple, N-Quad, RDF / XML, and Turtle format. It also support Jena interfaces [6] for processing RDF data.

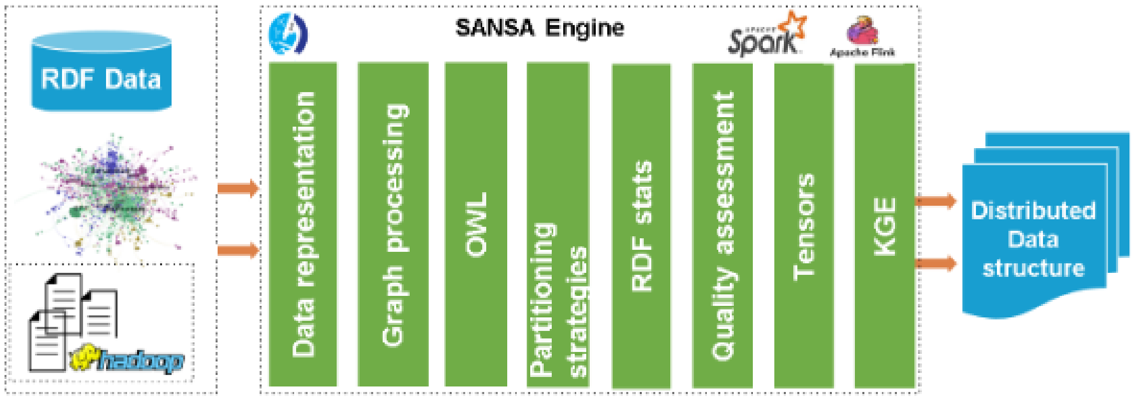


Figure 4.2: An Overview of SANSA-RDF Engine [7]

SANSA-QUERYING
SANSA-Querying [8], an RDF graph is a significant source for extracting and searching information from the related underlying data. SPARQL takes the description as a query and returns it as a set of bindings or as an RDF graph. SANSA offers three SPARK representation formats, namely; Resilient Distributed Datasets (RDD), Spark GraphX and Spark Data Frames, which is graph parallel computing. In our implementation, we used Jena ARQ OpVisitor [9] interface for parsing SPARQL to SQL on the querying layer on figure 4.3.

[3]https://github.com/SANSA-Stack/SANSA-Query
[4]https://github.com/SANSA-Stack/SANSA-RDF
[5]http://sansa-stack.net/introduction/
[6]https://jena.apache.org/
[7]http://sansa-stack.net/libraries/#RDF_OWL_API
[8]http://sansa-stack.net/introduction/
[9]https://jena.apache.org/documentation/query/index.html

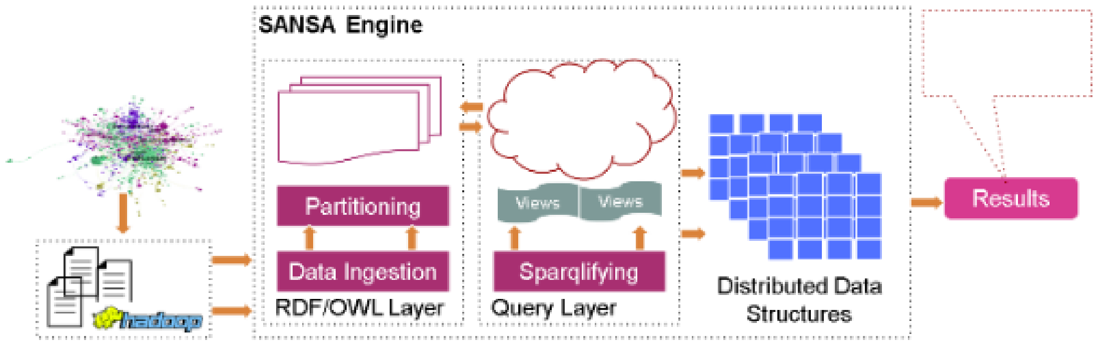


Figure 4.3: An Overview of SANSA-Querying Engine [10]

4.2 System Design of RDF Data Compression Techniques in a Highly Distributed Context

The RDF compression method focuses on decreasing an RDF dataset's space requirement by eliminating or replacing small value duplicate records and facilitating queries on top of compressed records [14]. It must be assured that the significance of a record stays intact, and by reversing the process, it should be feasible to recreate the same records. RDF compression combines the methods of vertical partitioning and star schema to represent and store data. For each attribute that includes the distinctive values zipped with an index, it produces a dimension table. In other words, there is a single central fact table which contains the subject, predicate, and object reference index numbers pointing to value in a dimension table. A dimension table only stores distinctive values which render it small enough for in-memory datasets to continue and process. Their system's major contributions are as follows:

- Using a dictionary strategy to compress RDF data.
- To use Spark RDDs to store the results of a query in memory (RAM).

4.2.1 RDF Dictionary Compression Architecture

To implement the RDF dictionary compression method, Spark and not Hadoop was used. Using Spark RDDs; maintain compressed data in-memory; Spark will split the data into a disk if there is no memory available to suit all the data that will speed up the process of collecting and processing compressed RDF data effectively [14].

RDF data Compression

The approached below was used for the system:

(a) RDF files were loaded through the SANSA API; then the RDF was transformed and distributed across the cluster node in Spark RDD. SANSA utilizes the RDF data

[10] http://sansa-stack.net/libraries/#RDF_Query_API

model to represent triple graphs with S, P and O. The RDF dataset may include many graphs and note information on each, enabling any of SANSA's decreased layers (querying) to generate queries involving more than one graph of information [1]. The key dataset is developed as the main construction block for Spark based on an RDD data structure. RDDs function as record databases in memory that could be performed with other bigger clusters concurrently [40, 14].

(b) The compressed RDD was subsequently maintained into an HDFS that offers a scalable, tolerance of failure. With the RDF dictionary compression algorithm, partitioned RDD was compressed and persisted in the distributed Hadoop cluster. The HDFS ensured that the data were uniformly split throughout the cluster node in order to guarantee balanced input and output.

(c) Instead of loading the RDF files sequentially from a local file system on the application (re)start, the application loaded the RDF in memory (RAM) from HDFS cluster nodes where each spark executor is responsible for reading from one or more HDFS nodes [14].

The method of compression is as follows:

(a) Read the data frame of the raw RDF dataset. Use Spark to distribute the RDF dataset vertically.

(b) Extract each field's distinctive values. Zipping that unique index value eliminates duplicate data and reduces space. Each value is subsequently referenced by its index number rather than its value.

(c) Generate a data table from the raw data frame input by replacing the real value with the respective reference index number.

(d) This creates a central fact table which stores reference index numbers; these refer to dictionary tables (subject, predicate, and objects dictionaries).

(e) The fact table guarantees that relationships between values are untouched and that a raw dataset can be recreated [14].

4.2.2 RDF Data Compression Algorithm

The compression algorithm uses both RDF vertical partitioning and star schema characteristics. The compression algorithm divides the dataset vertically; as a result, we receive N partitions for an N-triple dataset — one partition for each triple [14]. The suggested system is implemented over the SANSA project, which includes five parts: an implementation of RDF data compression; data loader; query engine (where the application is located); record scheme. It was observed that the data had too many duplicates after evaluating the dataset. This problem was targeted in the compression technique by storing a single value in a dictionary, which means that all the duplicate value were removed and stored only one record copy. Each unique subject, predicate, and object value is stored in the dictionary once (key, value) by assigning a unique number to a string value [14]. Transformed RDD triples contain only the distinctive number that corresponds to them. The method of compression takes the following steps to compress the data:

4.3 Our Architectural Approach

Our Scalable SPARQL Query Engine on top of the compressed RDF Data [14] focuses on SPARQL query evaluator with SPARQL-to-SQL conversion using direct mapping of the index value (dictionaries). The query engine can query large compressed RDF datasets using direct mapping. In direct mapping technique on the compressed index value (hdt schema), the resulting SQL translation directly reflects the name of database schema element such that neither structure nor vocabulary is changed. In other words, the results output of the SPARQL queries should be the same with the results output of the SQL queries. The SPARQL-to-SQL translator was designed through defining projection fields on top of the compressed index values of subject hdt, object hdt and predicate and then; the query conditions like `getWhereCondition()`, `getDistinct()` etc. On the following section, we are going to explain the architectural overview of our approach:

- Input Data must be in RDF File.
- RDF file will be converted and compressed with HDT based approach defined in SANSA Spark package.
- Convert SPARQL queries that works on RDF dataset should be work on compressed/Indexed dataset.
- Same result should be produced with Both the query.
- Input SPARQL query must have only three columns subject, predicate and object (?S ?P ?O).
- Where condition should have a condition in below order
  (a) Subject Condition.
  (b) Predicate Condition.
  (c) Object Condition.

4.4 Architecture

Our system design, map the SPARQL query directly on the compressed index dataset (subject hdt, predicate hdt and object hdt) and convert the SPARQL query to SQL and also execute the SQL query on the compressed index dataset.

4.5 Modules

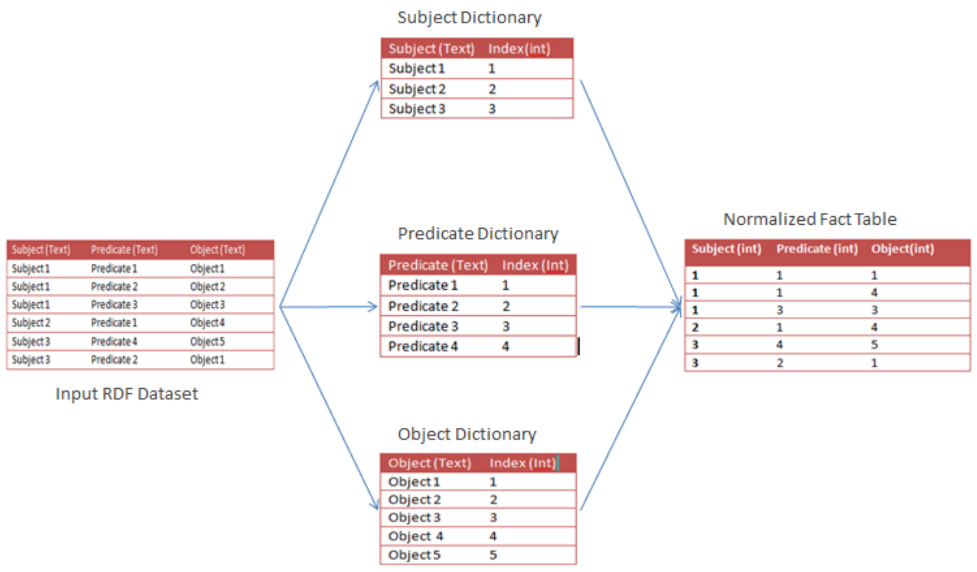
The entire work is divided into three module

4.5.1 RDF Data Compression Module

This is an existing component under SANSA [1] stack [11]. It implemented a compression system that has a feature like optimizing the space on the disk by avoiding duplicates [14]. The idea of RDF data compression is to reduce the data size by maintaining a single copy for each record

[11] https://github.com/SANSA-Stack

and using the vertically partitioning technique on the dataset. As a result, the compression technique provides an excellent compression ratio, from 70% to 82%.



Subject (Text)	Predicate (Text)	Object (Text)
Subject1	Predicate1	Object1
Subject1	Predicate2	Object2
Subject1	Predicate3	Object3
Subject2	Predicate1	Object4
Subject3	Predicate4	Object5
Subject3	Predicate2	Object1

Subject (Text)	Index(int)
Subject1	1
Subject 2	2
Subject 3	3

Predicate (Text)	Index (Int)
Predicate 1	1
Predicate 2	2
Predicate 3	3
Predicate 4	4

Object (Text)	Index (Int)
Object 1	1
Object 2	2
Object 3	3
Object 4	4
Object 5	5

Subject (int)	Predicate (int)	Object(int)
1	1	1
1	1	4
1	3	3
2	1	4
3	4	5
3	2	1

Figure 4.4: An Overview of the RDF Data Compression [14]

Scala and Spark software frame are used for the implementation of the system.

Record Schema Module

The record schema module offers the intermediate dataset with schema-related data [14]. Two kinds of schema are return:

Dictionary schema

(a) INDEX (Numeric)

(b) NAME (String)

We have the subject, predicate and object dictionary from the dictionary schema, where each dictionary will have the name and index number as shown in the figure 4.5 [14].

Fact table schema A central table containing only the numerical values of subject, predicate, and object Which enables us to maintain relations between the three dictionaries (See the dictionary schema below).

(a) SUBJECT_INDEX (Numeric)

(b) PREDICATE_INDEX (Numeric)

(c) OBJECT_INDEX (Numeric)

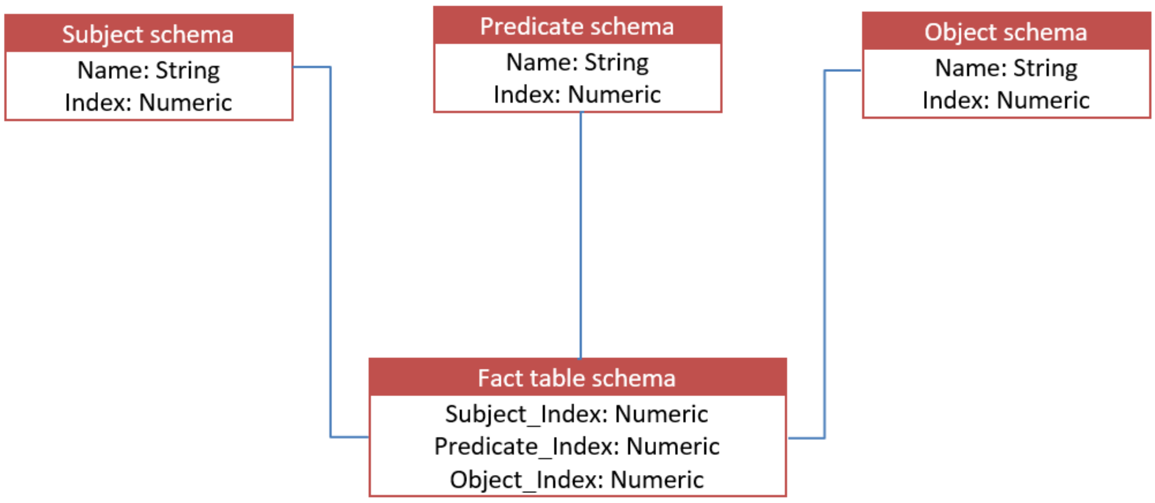


Figure 4.5: Visualization of Record Schema [14]

4.5.2 SPARQL to SQL Translator

The key component of our query engine accepts SPARQL query and returns the SparkSQL query. It has leveraged Apache Jena framework's OpVisitor that splits the SPARQL queries into various components like projection fields, Where condition, Count/Distinct, filter functions and arguments etc. As of now below, functionalities of SPARQL is supported. The component is written in a way that it could be easily extended with minimum code.

- Simple Select without condition.
- Select with single Where Condition.
- Select Query with Multiple filters.
- Select with Distinct.
- Select with single Filter.
- It supports below filter functions
  (a) STRLEN
  (b) SUBSTR
  (c) STRENDS
  (d) CONTAINS
  (e) RAND
  (f) STRBEFORE
  (g) STRAFTER
  (h) REPLACE
- Filter Logical and comparison operators supported
  (a) AND
  (b) OR
  (c) Greater than ($>$)

(d) Less Than ($<$)

(e) Greater than Equal to ($>=$)

(f) Less Than Equal to ($<=$)

(g) Equals to ($=$)

SPARQL to SQL Translation Code

Listing 4.1: Translation Technique using Spark and Scala

```

package net.sansa_stack.examples.spark.hdt

import ...

// Read the RDF Data Frame and convert into the compressed Data Frame creating
    Subject_hdt, Object_hdt and Predicate_hdt with column side S,O,P containing the
    index number.
class TripleOpsQuery
{
  val log=LoggerFactory.getLogger("TripleOpsQueryy")

  def parseValue(value:String): String ={
    value.replace("<","").replace(">","")
  }

  // Function convert SPARQL Projection fields to SQL Projection
  def getProjectionFields () = {
    var result = ""

    for (i <- 0 to queryScanner.varList.size() - 1) {
      val name=queryScanner.varList.get(i).getVarName

      if(name.equalsIgnoreCase("S")){
        result +=s"${TripleOps.SUBJECT_TABLE}.name as subject, "
      }
      else if(name.equalsIgnoreCase("O")){
        result +=s"${TripleOps.OBJECT_TABLE}.name as object, "
      }
      else if(name.equalsIgnoreCase("P")){
        result +=s"${TripleOps.PREDICATE_TABLE}.name as predicate, "
      }
    }
    //remove extra comma at the end of the triples
    result.reverse.replaceFirst(",", "").reverse
  }

  // Function to convert SPARQL WHERE Conditions to SQL.
```

```
def getWhereCondition(): String = {
  var tempStr = ""
  for(i <- 0 to queryScanner.whereCondition.size()-1)
  {
    if(!queryScanner.subjects.get(i).toString().toLowerCase().contains("?s")){
      tempStr += s" ${TripleOps.SUBJECT_TABLE}.name='${queryScanner.subjects.get(i
          )}' and"
    }
    if(!queryScanner.objects.get(i).toString().toLowerCase().contains("?o")){
      tempStr += s" ${TripleOps.OBJECT_TABLE}.name='${queryScanner.objects.get(i)}'
          and"
    }
    if(!queryScanner.predicates.get(i).toString().toLowerCase().contains("?p")){
      tempStr += s" ${TripleOps.PREDICATE_TABLE}.name='${queryScanner.predicates.
          get(i)}' and"
    }
  }
  tempStr=tempStr.reverse.replaceFirst("dna","").reverse
  if(tempStr.length>5) {s" where (${tempStr})" }
  else {""}

  // Function to convert SPARQL DISTINCT Conditions to SQL.
  def getDistinct():String = {
  if(queryScanner.isDistinctEnabled){

    var groupBy=""
    for (i <- 0 to queryScanner.varList.size() - 1) {
      if (queryScanner.subjects.contains(queryScanner.varList.get(i))) {
        groupBy += s"${TripleOps.SUBJECT_TABLE}.name, "
      }
      else if (queryScanner.objects.contains(queryScanner.varList.get(i))) {
        groupBy += s"${TripleOps.OBJECT_TABLE}.name, "

      } else if (queryScanner.predicates.contains(queryScanner.varList.get(i))) {
        groupBy += s"${TripleOps.PREDICATE_TABLE}.name, "
      }
    }
    "group by "+ groupBy.reverse.replaceFirst(",", "").reverse
  }
  else
  {
    ""
  }
}

  // Function to convert SPARQL FILTER Condition to SQL.
  def getFilterCondition(): String ={
  var strCondition =""
```

```
    for( i <- 0 to queryScanner. filters . size ()-1)
    {
      strCondition += FilterCondition .getHDTFilter(queryScanner. filters . get(i))
      println (" Condition Processed: "+ strCondition )
    }

    strCondition = strCondition . reverse . replaceFirst ("dna","") . reverse
    if ( strCondition . length>5) s"where ${ strCondition }" else ""
  }

  // Important function that convert the SPARQL Query to SQL
  def getQuery(queryStr : String ) ={

    queryScanner. reset
    // val queryStr="SELECT ?resource WHERE { ?resource ?x ?age . FILTER (?age >= 24)}
        "
    val query = QueryFactory. create ( queryStr )
    val op = Algebra.compile(query)
    OpWalker.walk(op, queryScanner)
    val result =s" select ${ getProjectionFields () }from ${TripleOps.HDT_TABLE} inner
        join ${TripleOps.SUBJECT_TABLE} on ${TripleOps.HDT_TABLE}.s=${TripleOps.
        SUBJECT_TABLE}.index" +
      s" inner join ${TripleOps.OBJECT_TABLE} on ${TripleOps.HDT_TABLE}.o=${
          TripleOps.OBJECT_TABLE}.index" +
      s" inner join ${TripleOps.PREDICATE_TABLE} on ${TripleOps.HDT_TABLE}.p=$
          {TripleOps.PREDICATE_TABLE}.index" +
      s" ${getWhereCondition()} ${ getFilterCondition () } ${ getDistinct () }"
    result
  }

  // To check the SPARQL conversion to SQL Accuracy
  def execute(spark:SparkSession, rdfTriple : RDD[org.apache.jena.graph. Triple ] , query:
      String ): Unit ={

    var queryops=new TripleOpsQueryNew()

    var df=spark. sql (queryops.getQuery(query))
    val count= rdfTriple . sparql (query) . count ()
    println (s"SparQL Query : ${query}")
    println ("Spark SQL: "+queryops.getQuery(query))
    println ("SparQL Query Count: "+ count)
    println (s"Spark SQL Count: ${df.count() }")

  }
```

The codes are explained below [12].

(a) `Line 7` Read the RDF data frame and convert it into the Compressed data frame by creating Subject_hdt, Object_hdt and Predicate_hdt with column S.O.P containing the index values.

(b) `Line 16` We create the SPARQL projection fields to SQL projection which is user-defined, i.e., the user define what they want to see based on (?S, ?O and ?P).

(c) `Line 32` Removes extra comma at the end of the triples

(d) `Line 36` We defined the function here to convert SPARQL WHERE conditions to SQL to be able to execute this feature. Where there is a dot (.) tells the system that there is a WHERE condition between the two triples. Splitting all conditions by (.) and evaluate conditions one by one.

(e) `Line 56` We defined the function to convert SPARQL DISTINCT conditions to SQL. The system can handle SPARQL queries with this feature and convert the queries to SQL.

(f) `Line 56` We created the SPARQL FILTER conditions to SQL here to be able to handle most of SPARQL filter features like SELECT with Nested FILTER (filter predicates like comparison operators: (<=, <, =, >, >=), logical operators: (!, &&, | |) and arithmetic operators: (+, −)) and FILTER functions (STRLEN, SUBSTR, STRENDS, CONTAINS, RAND, IN, NOT IN, STRBEFORE, STRAFTER, and REPLACE)

(g) `Line 56` Here we created the most important function getQuery that performs the actual SPARQL to SQL conversion using `INNER JOIN` from `TripleOps.SUBJECT_TABLE` on `(TripleOps.HDT_TABLE).s` appending it to `(TripleOps.OBJECT_TABLE).index`. Doing the same for object and predicate as well.

(h) `Line 109` We created a function here to check for the accuracy of SPARQL-to-SQL conversion through SPARQL Query and SQL Query count and also the time it took to execute each query. The number of SPARQL Query count should be the same with the number of SQL Query count.

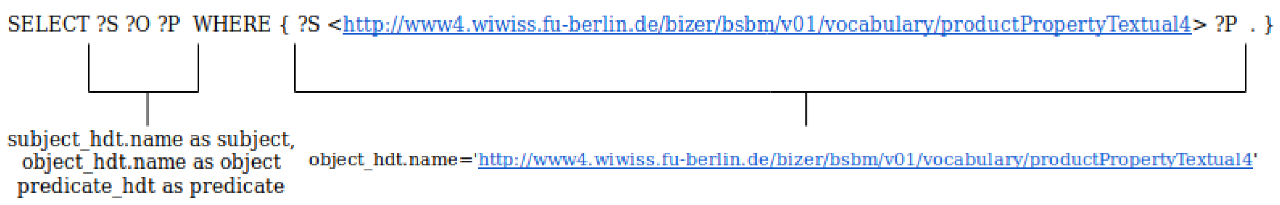
Below is one of the examples of SPARQL-to-SQL translation query

Listing 4.2: Example: Select Query with Where Condition

```

SELECT ?S ?O ?P WHERE
{
  ?S <http://www4.wiwiss.fu-berlin.de/bizer/bsbm/v01/vocabulary/
        productPropertyTextual4> ?P .
}
```

[12] https://github.com/ibhadavid/SANSA-Examples


SELECT ?S ?O ?P WHERE { ?S <http://www4.wiwiss.fu-berlin.de/bizer/bsbm/v01/vocabulary/productPropertyTextual4> ?P . }
subject_hdt.name as subject,
object_hdt.name as object
predicate_hdt as predicate
object_hdt.name='http://www4.wiwiss.fu-berlin.de/bizer/bsbm/v01/vocabulary/productPropertyTextual4'

Listing 4.3: Query Result

```

Resut Query:

select  subjects_hdt .name as  subject ,  objects_hdt .name as  object ,  predicates_hdt .name
    as  predicate  from hdt  inner  join  subjects_hdt  on hdt.s=subjects_hdt .index  inner
    join  objects_hdt  on hdt.o=objects_hdt .index  inner  join  predicates_hdt  on hdt.p=
    predicates_hdt .index   where (  objects_hdt .name='http :// www4.wiwiss.fu—berlin.de/
    bizer/bsbm/v01/vocabulary/ productPropertyTextual4 '  )  and   1=1
```

4.5.3 Query Engine

The query engine is the entry point of the project, which accepts input data source and queries to be executed on the data source. As an outcome, it returns the result of SPARQL Query count and the SQL Query count and their runtime.
The system accepts the N-triple dataset and query, then validate the query if it is a SPARQL query. Apply compression technique (vertical compression technique on the dataset) to remove redundancy, optimized the disk space and compressed the dataset to 70-80%. The SPARQL query is applied on the compressed dataset, and the SPARQL to SQL translator converts the SPARQL queries to SQL queries. The SQL queries are executed on the compressed dataset, and the result of the SQL count is validated with the result of the SPARQL query count. Below is the query engine processing flow:

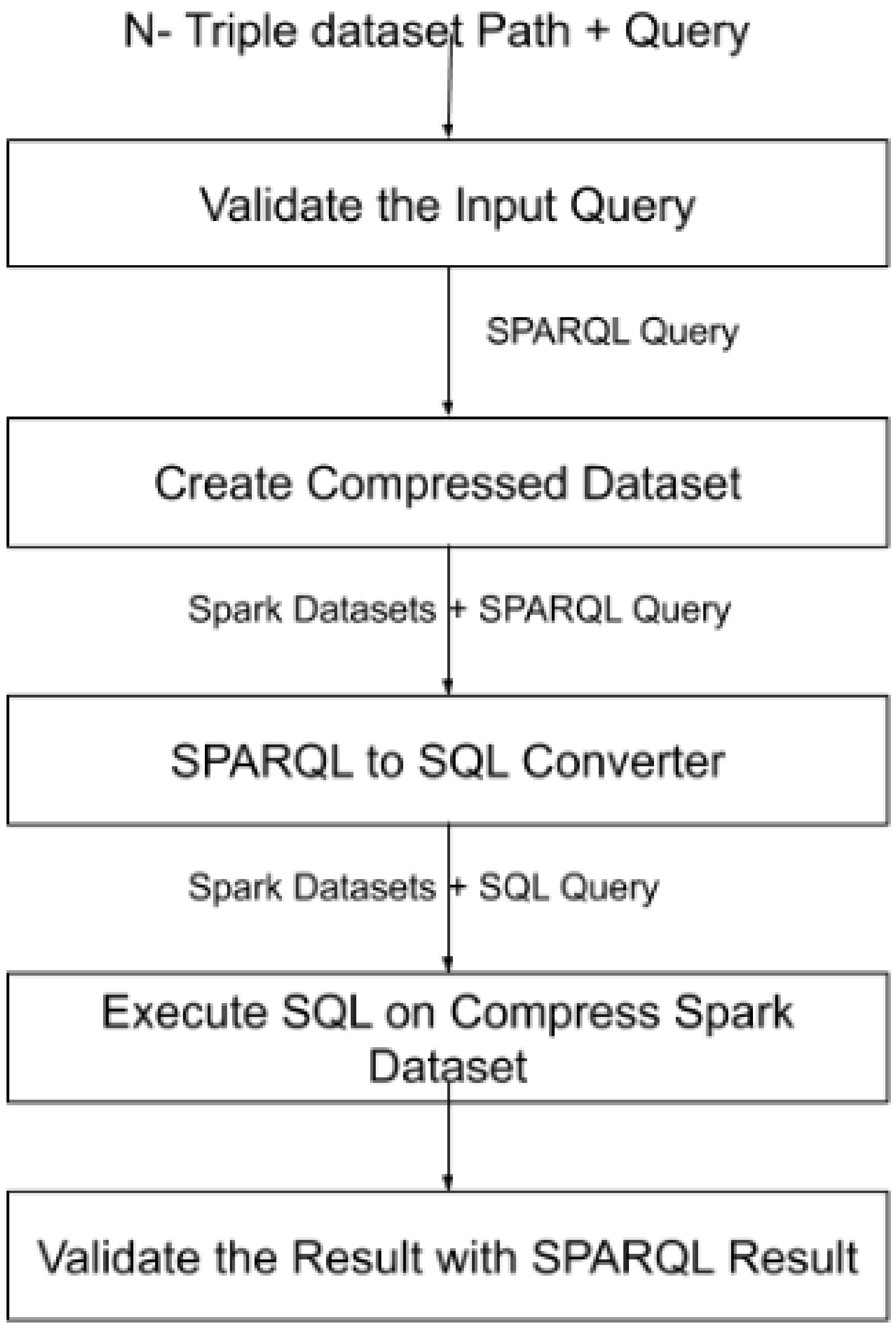

N- Triple dataset Path + Query
Validate the Input Query
SPARQL Query
Create Compressed Dataset
Spark Datasets + SPARQL Query
SPARQL to SQL Converter
Spark Datasets + SQL Query
Execute SQL on Compress Spark Dataset
Validate the Result with SPARQL Result

Figure 4.6: Query Engine Steps

Chapter 5

Evaluation

We implemented and design SPARQL-to-SQL Query Engine over the compressed RDF Data using direct mapping techniques as already discussed in chapter 4. To be able to evaluate our system overall efficiency, we ran some simple and complex SPARQL Query over the compressed RDF datasets based on the SPARQL features we implemented in the SPARQL query engine to ascertain the answer to the research question below:

(a) How Scalable is SPARQL query engine (evaluator) over compressed data?

(b) How efficient is SPARQL-to-SQL re-writer over compressed data?

(c) How accurate is the SPARQL-to-SQL translator over compressed data?

To understand the efficiency of the SPARQL-to-SQL Query Engine on the compressed RDF data, and better explain the research question, we take a look at the following points below:

- We compare the SPARQL Query count to that of the SQL Query count and record the results on the compressed RDF data.
- We record the time it took for the query to run for the compressed RDF data on different data sizes to ascertain scalability for both Sparklify [41] and SparkSQL.

5.1 Configuration of the Cluster

To evaluate how scalable with Spark and Scala our SPARQL-to-SQL Query Engine is, we deployed the system on the cluster of Smart Data Analytic [1] that has the following configuration to test for scalability:

- Three servers with a total of 256 cores
  - 1 cluster manager and 2 worker nodes.
- `Operating system`: 16.04.4 LTS.
- `Scala version`: 2.11.8.
- `Spark version`: 2.3.1.
- `Memory`: 1.70TB.

[1] http://sda.tech/

5.2 Methodology

Our evaluation criteria for the compressed RDF data by our SPARQL query engine with a two nodes cluster is in term of the accuracy of the query count for Sparklify and SparkSQL queries. The results of the queries were evaluated consecutively on the different datasets. All the queries were run automatically on the query engine over the compressed data, and the counts were recorded to test dataset scalability. The SPARQL Query Engine over the compressed data was verified by comparing the Sparklify query count over the SQL query count; the ability to scale from small of 204.6MB datasets to large datasets of up to 21.5GB. We query the compressed RDF data and record the individual query count to Sparklify and SQL query count and their runtime. This step is repeated three times, and the mean query time values are recorded.

5.3 The System

Our SPARQL query engine with SPARQL-to-SQL translator is a scalable system built with Scala and Spark. We compare our SPARQL Query Engine system mainly by the basis of being able to query the compressed RDF dictionary datasets and convert SPARQL queries to SQL queries in SANSA. Our system main feature is to convert SPARQL queries to SQL queries and to measure the runtime for scalability on different data sizes.

5.4 Experimental Setup

5.4.1 Benchmarks

To evaluate our system, we used two kinds of the dataset; Berlin SPARQL Benchmark (BSBM) [42] and Lehigh University Benchmark (LUBM) [43]. To analyze our system, we assume the datasets into three categories; small medium and large dataset:

(a) BSBM [42] is a very well-known benchmark, built for e-commerce, in which many products are available through various companies, and users of these products can review them. This benchmark enables us to produce the three datasets as follows:

  i. 841,931 triples [`216.1MB`]
  ii. 12,005,557 triples [`3.1 GB`]
  iii. 81,980,472 triples [`21.5 GB`]

(b) LUBM [43]: The domain ontology in LUBM benchmark are characterized in terms of publications, Universities, course-work and different department groups. The following datasets are generated with the help of the above benchmark:

  i. 1,163,215 triples [`204.6 MB`]
  ii. 17,705,050 triples [`3.1 GB`]
  iii. 69,080,764 triples [`12.3 GB`]

We performed five types of SPARQL queries in each of the benchmarks above to be able to evaluate our system, and these queries were converted to SQL as well (see accuracy tables 5.4 and 5.5).

5.4.2 Benchmarks Queries

We evaluated the SPARQL query engine with five different SPARQL queries on each of the benchmark.

BSBM Benchmark

(a) `Query one with` *`COUNT(*)`* `feature and the equivalent converted SQL Query: Compressed Datasets:`

```
SPARQL Query:
 SELECT (COUNT(*) AS ?A) WHERE { ?S ?P ?O }

 SQL Query:
  select  count(*) from hdt inner join subjects_hdt on hdt.s=subjects_hdt.
     index inner join objects_hdt on hdt.o=objects_hdt.index inner join
     predicates_hdt on hdt.p=predicates_hdt.index where 1=1 and 1=1
```

Listing 5.1: Query on the Compressed Data

Listing 5.1 above is a query that will return all values with As in subject, predicate and object triple.

(b) `Query two with` *`DISTINCT`* `feature and the equivalent converted SQL Query: Compressed Datasets:`

```
SPARQL Query:
  SELECT DISTINCT ?S ?O ?P WHERE { ?S ?P ?O }

 SQL Query:
   select  subjects_hdt.name as subject,  objects_hdt.name as object,
       predicates_hdt.name as predicate from hdt inner join subjects_hdt
       on hdt.s=subjects_hdt.index inner join objects_hdt on hdt.o=
       objects_hdt.index inner join predicates_hdt on hdt.p=predicates_hdt.
       index  where 1=1 and 1=1 group by subjects_hdt.name, objects_hdt.
       name, predicates_hdt.name
```

Listing 5.2: Query on the Compressed Data

Listing 5.2 above is a query that will return all distinct values of subject, predicate and object triple.

(c) `Query three with` *`OPTIONAL`* `feature and the equivalent converted SQL Query: Compressed Datasets:`

```
SPARQL Query:
 SELECT ?s ?p WHERE {
 ?s ?p <http://www4.wiwiss.fu−berlin.de/bizer/bsbm/v01/instances/
     dataFromProducer2/Product92> .
 OPTIONAL { ?s <http://www4.wiwiss.fu−berlin.de/bizer/bsbm/v01/vocabulary/
     productPropertyTextual1 > ?o }
}

SQL Query:
  select subjects_hdt.name as subject, predicates_hdt.name as predicate
     from hdt inner join subjects_hdt on hdt.s=subjects_hdt.index inner
     join objects_hdt on hdt.o=objects_hdt.index inner join predicates_hdt
      on hdt.p=predicates_hdt.index where ( objects_hdt.name='http://
     www4.wiwiss.fu−berlin.de/bizer/bsbm/v01/instances/dataFromProducer2/
     Product92' or ( predicates_hdt.name='http://www4.wiwiss.fu−berlin.
     de/bizer/bsbm/v01/vocabulary/productPropertyTextual1 ' )) and 1=1
```

Listing 5.3: Query on the Compressed Data

Listing 5.3 above is a query that will return all product92 except the product92 properties.

(d) `Query four with` *`PREFIX`* `feature and the equivalent converted SQL Query: Compressed Datasets:`

```
SPARQL Query:
   PREFIX foo: <http://www4.wiwiss.fu−berlin.de/bizer/bsbm/v01/instances/>
       PREFIX hoo: <http://www4.wiwiss.fu−berlin.de/bizer/bsbm/v01/
      vocabulary/> SELECT ?S ?O ?P WHERE { foo:ProductType2 ?P ?O .}
      limit 10

SQL Query:
   select subjects_hdt.name as subject, objects_hdt.name as object,
      predicates_hdt.name as predicate from hdt inner join subjects_hdt
      on hdt.s=subjects_hdt.index inner join objects_hdt on hdt.o=
      objects_hdt.index inner join predicates_hdt on hdt.p=predicates_hdt.
      index where ( subjects_hdt.name='http://www4.wiwiss.fu−berlin.de/
      bizer/bsbm/v01/instances/ProductType2' ) and 1=1
```

Listing 5.4: Query on the Compressed Data

(e) `Query five with` *`WHERE with FILTER`* `feature and the equivalent converted SQL Query: Compressed Datasets:`

```
SPARQL Query:
   SELECT ?S ?O ?P WHERE { ?S ?P ?O . FILTER ( STRLEN(?S) >= 40 ) . }

SQL Query:
    select  subjects_hdt .name as subject ,  objects_hdt .name as object ,
        predicates_hdt .name as predicate from hdt inner join subjects_hdt
        on hdt.s=subjects_hdt.index inner join objects_hdt on hdt.o=
        objects_hdt.index inner join predicates_hdt on hdt.p=predicates_hdt
        .index  where 1=1  and  length( subjects_hdt .name) >= 40
```

Listing 5.5: Query on the Compressed Data

(f) `Query six with` *`WHERE without FILTER`* `feature and the equivalent converted SQL Query: Compressed Datasets:`

```
SPARQL Query:
    SELECT ?S ?O ?P WHERE { ?S ?P ?O }

SQL Query:
     select  subjects_hdt .name as subject ,  objects_hdt .name as object ,
         predicates_hdt .name as predicate from hdt inner join subjects_hdt
         on hdt.s=subjects_hdt.index inner join objects_hdt on hdt.o=
         objects_hdt.index inner join predicates_hdt on hdt.p=
         predicates_hdt.index  where 1=1  and  1=1
```

Listing 5.6: Query on the Compressed Data

To analyze the scalability of our system, our dataset have been classified into three classes and they are generated from BSBM and LUBM datasets; ranging from `204.6 MB` to `21.5 GB`.

Dataset Description: Small Sized

Shown below is the small datasets sizes with its respective triples; `subjects`, `predicates` and `objects` in the table 5.1

Dataset	Size (MB)	Triples	Subjects	Predicates	Objects
BSBM	216.1	841,931	78,478	40	178,604
LUBM	204.6	1,163,215	183,426	18	137,923

Table 5.1: Small Dataset Description

Dataset Description: Medium Sized

Shown below are the sizes of the medium datasets with its respective triples; subjects, predicates, and objects as shown in table 5.2

Dataset	Size (GB)	Triples	Subjects	Predicates	Objects
BSBM	3.1	12,005,557	1,091,609	40	2,221,346
LUBM	3.1	17,705,050	2,781,793	18	2,070,590

Table 5.2: Medium Dataset Description

Dataset Description: Large Sized

Shown below is the large datasets sizes with its respective triples; `subjects`, `predicates` and `objects` in the table 5.3

Dataset	Size (GB)	Triples	Subjects	Predicates	Objects
BSBM	21.5	81,980,472	7,410,953	40	12,467,580
LUBM	12.3	69,080,764	1,084,7184	18	8,072,360

Table 5.3: Large Dataset Description

5.5 Experiments & Results

5.5.1 System Accuracy

The accuracy of our system will be our primary focus to measure the success of the project. We consider the result row count as primary success criteria. If the result of SQL on compressed data and result count of SPARQL matches, then we consider it as a success.

SPARQL-to-SQL Translation Accuracy for BSBM Datasets

The table Table 5.4 below show the accuracy of SPARQL-to-SQL conversion and the runtime for each query for BSBM datasets.

#SPARQL Query	#Our Approach Query Count	#Our Approach Query Time(second)	#Sparklify Query Count	#Sparklify Query Time(second)
Query 1	841931	2	841931	29
Query 2	87810	2	87810	10
Query 3	1	2	1	29
Query 4	841931	9	841931	33
Query 5	6	19	6	34

Table 5.4: SPARQL to SQL Translation Accuracy BSBM Datasets

SPARQL-to-SQL Translation Accuracy for LUBM Datasets

The table Table 5.5 below show the accuracy of SPARQL-to-SQL conversion and the run-time for each query for LUBMB datasets.

#SPARQL Query	#Our Approach Query Count	#Our Approach Query Time(second)	#Sparklify Query Count	#Sparklify Query Time(second)
Query 1	1163215	1	1163215	21
Query 2	232863	1	232863	9
Query 3	1	1	1	22
Query 4	1124616	11	1124616	23
Query 5	0	18	0	23

Table 5.5: SPARQL to SQL Translation Accuracy LUBM Datasets

5.5.2 Compressed Datasets Query Result

Five SPARQL queries were used to evaluate our system on BSBM and LUBM benchmark datasets ranging from simple SELECT to complex queries such as DISTINCT, COUNT, FILTER, etc.

On the BSBM Small Size Datasets

To determine the correctness (that is SPARQL count equals the SQL query count) and completeness (that is the query finish running all the time) of our SPARQL query engine over the compressed RDF Datasets described on table Table 5.1 in terms of scalability on the small BSBM datasets. We performed five queries, and the SparkSQL query count, as well as the Sparklify count, was recorded and their runtime.

SPARQL Query	Our Approach Time(second)	Sparklify Time(second)	Our Approach Query Count	Sparklify Query Count
Query 1	2	29	841931	841931
Query 2	2	10	87810	87810
Query 3	2	29	1	1
Query 4	9	33	841931	841931
Query 5	19	34	6	6

Table 5.6: Query Results on BSBM Small Dataset

The chart below shows the scalability of the system on small-sized datasets.

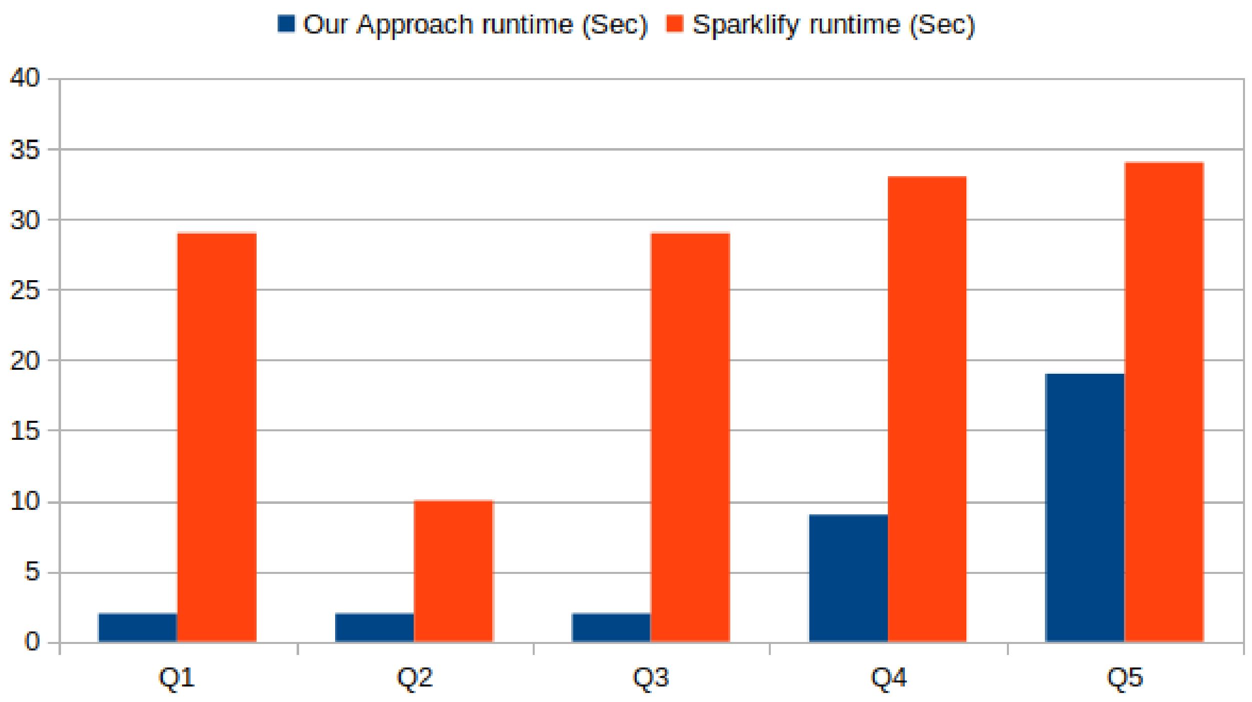


Figure 5.1: BSBM Query Runtime Table 5.1

On the LUBM Small Size Datasets

We performed the same five queries on the compressed LUBM RDF Dataset described on table Table 5.1 in terms of scalability. We recorded the Spark SQL query count as well as the Sparklify count and their runtime.

SPARQL Query	Our Approach Time(second)	Sparklify Time(second)	Our Approach Query Count	Sparklify Query Count
Query 1	1	21	1163215	1163215
Query 2	1	9	232863	232863
Query 3	1	22	1	1
Query 4	11	23	1124616	1124616
Query 5	18	23	0	0

Table 5.7: Query Results on LUBM Small Dataset

The chart below shows the scalability of the system on LUBM small-sized datasets.

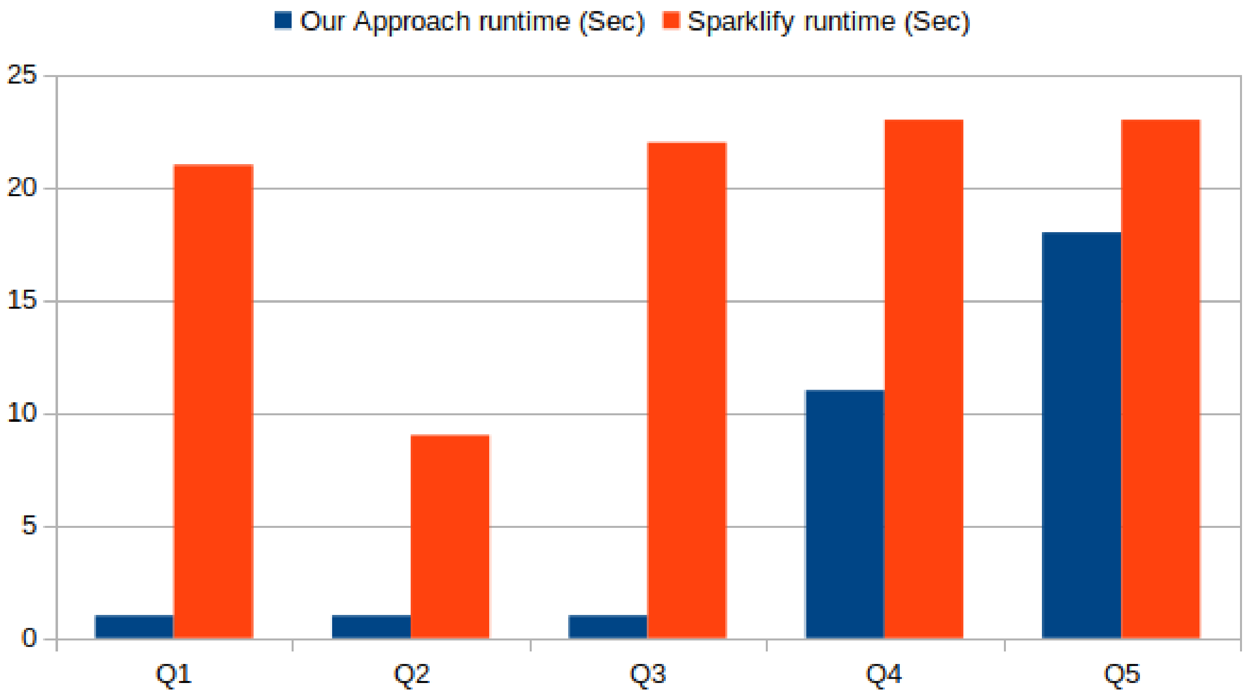


Figure 5.2: LUBM Query Runtime Table 5.1

On the BSBM Medium Size Datasets

We performed the same five queries on the compressed medium-sized BSBM RDF Dataset described on table Table 5.2 in terms of scalability. We recorded the Spark SQL query count as well as the Sparklify count and their runtime.

SPARQL Query	Our Approach Time(second)	Sparklify Time(second)	Our Approach Query Count	Sparklify Query Count
Query 1	5	267	12005557	12005557
Query 2	9	34	1226929	1226929
Query 3	7	272	1	1
Query 4	17	276	12005557	12005557
Query 5	52	281	6	6

Table 5.8: Query Results on BSBM Medium Dataset

The chart below shows the scalability of the system on BSBM medium sized datasets.

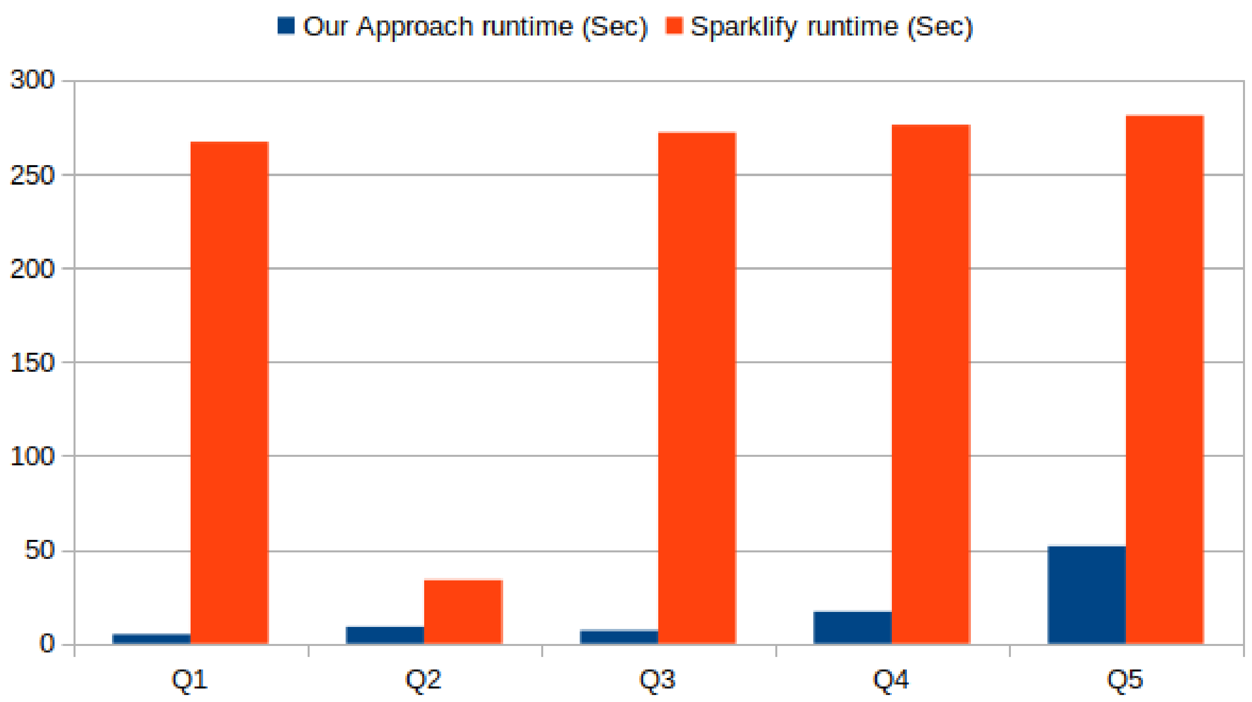


Figure 5.3: LUBM Query Runtime Table 5.2

On the LUBM Medium Size Datasets

We performed the same five queries on the compressed medium-sized LUBM RDF Dataset described on table Table 5.2 in terms of scalability. We recorded the Spark SQL query count as well as the Sparklify count and their runtime.

SPARQL Query	**Our Approach Time(second)**	**Sparklify Time(second)**	**Our Approach Query Count**	**Sparklify Query Count**
Query 1	10	172	17705050	17705050
Query 2	12	32	3547452	3547452
Query 3	12	165	1	1
Query 4	25	185	17103866	17103866
Query 5	54	179	0	0

Table 5.9: Query Results on LUBM Medium Dataset

The chart below shows the scalability of the system on LUBM medium sized datasets.

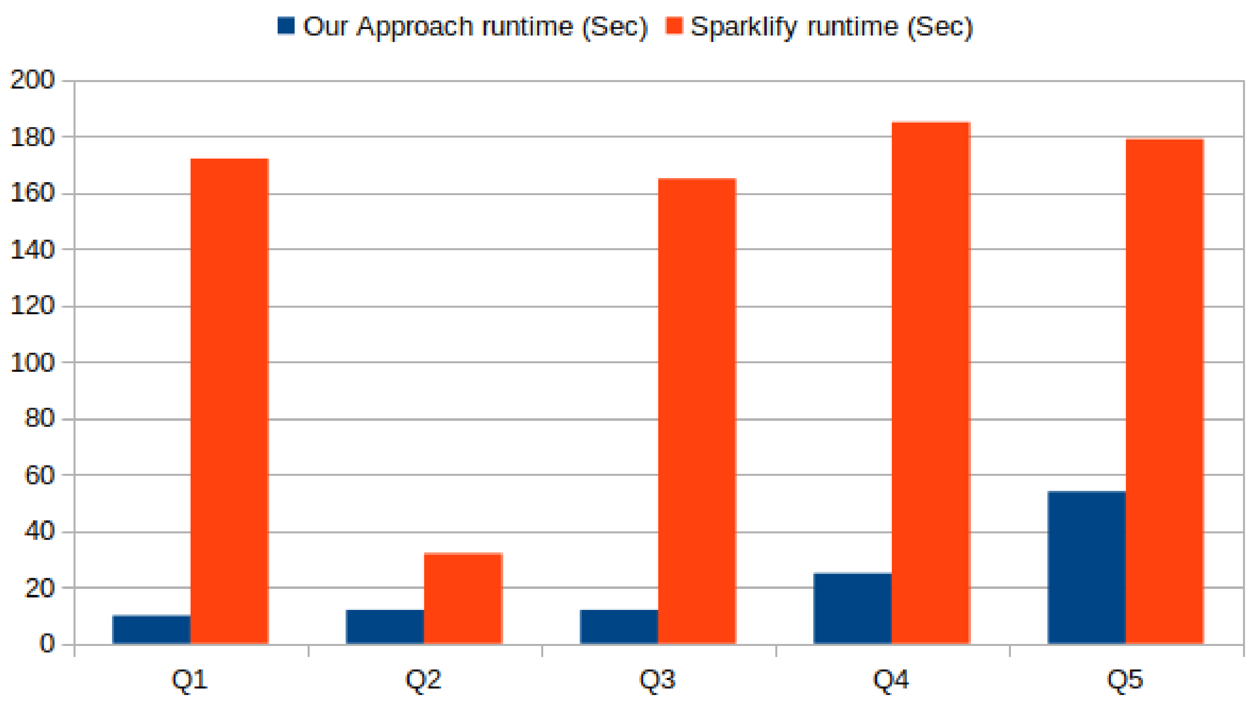


Figure 5.4: LUBM Query Runtime Table 5.2

On the BSBM Large Size Datasets

We performed the same five queries on the compressed large-sized BSBM RDF Dataset described on table Table 5.3 in terms of scalability. We recorded the Spark SQL query count as well as the Sparklify count and their runtime.

SPARQL Query	**Our Approach Time(second)**	**Sparklify Time(second)**	**Our Approach Query Count**	**Sparklify Query Count**
Query 1	35	14912	81980472	81980472
Query 2	39	162	8577793	8577793
Query 3	32	1490	1	1
Query 4	92	1536	81980472	81980472
Query 5	280	1479	6	6

Table 5.10: Query Results on BSBM Large Dataset

The chart below shows the scalability of the system on BSBM large sized datasets.

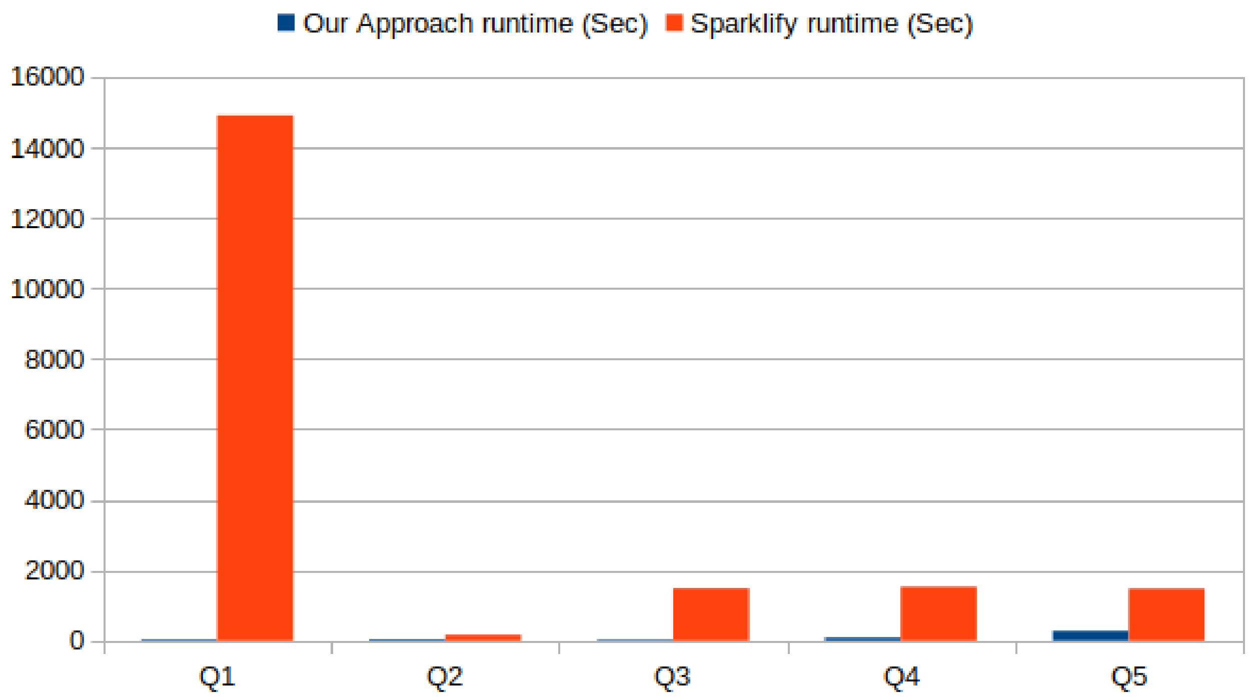


Figure 5.5: BSBM Query Runtime Table 5.3

On the LUBM Large Size Datasets

We performed the same five queries on the compressed large-sized LUBM RDF Dataset described on table Table 5.3 in terms of scalability. We recorded the Spark SQL query count as well as the Sparklify count and their runtime.

SPARQL Query	Our Approach Time(second)	Sparklify Time(second)	Our Approach Query Count	Sparklify Query Count
Query 1	25	454	69080764	69080764
Query 2	32	111	13839628	13839628
Query 3	24	462	1	1
Query 4	61	489	66731208	66731208
Query 5	171	458	0	0

Table 5.11: Query Results on LUBM Large Dataset

The chart below shows the scalability of the system on LUBM large sized datasets.

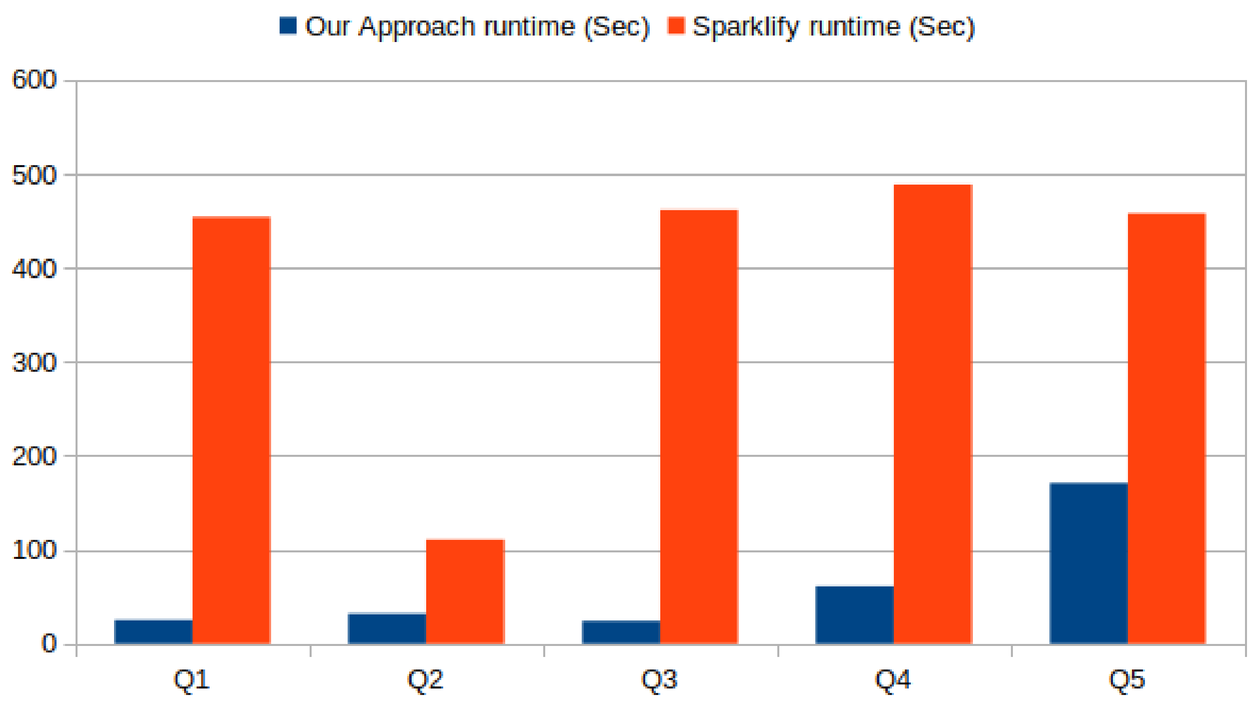


Figure 5.6: BSBM Query Runtime Table 5.3

Chapter 6

Conclusions and Future Work

This thesis project that is leveraged on SANSA framework was designed and implemented with Spark and Scala with the sole aim of developing a scalable SPARQL Query Engine on top of compressed data directly which can translate SPARQL queries to SQL queries. Our SPARQL Query Engine System scale well on the large Compressed data (index value) when we look at the query runtime in reference to the SPARQL-to-SQL translation. We observed this through the graphs that as we increase the compressed datasets from small (204.6MB) to medium (3.1GB) and to large (21.5GB) the query translation time becomes faster. This result is made possible because the HDT technique reduces the data size to 70%-80%, and we store the result in indexed and in-memory. So the query was directly applied on the indexed in-memory dataset which makes it fast.

For future work, we would like to suggest that:

(a) Compare the SANSA SPARQL Query Engine on compressed data with other SPARQL query engine (e.g. Sparklify, SPARQLGX and so fort).

(b) Integrate other SPARQL Query frontend endpoint for the SANSA SPARQL Query Engine on compressed RDF data to enable users to write SPARQL queries.

Bibliography

[1] Jens Lehmann, Gezim Sejdiu, Lorenz Bühmann, Patrick Westphal, Claus Stadler, Ivan Ermilov, Simon Bin, Nilesh Chakraborty, Muhammad Saleem, Axel-Cyrille Ngonga Ngomo, and Hajira Jabeen. Distributed Semantic Analytics Using the Sansa Stack. In *In International Semantic Web Conference (ISWC), Paper*, pages (147–155), Springer, Cham., 2017.

[2] Sangyoon Oh Minh Duc Nguyen, Min Su Lee and Geoffrey C. Fox. SPARQL Query Optimization for Structural Indexed RDF Data, 2014.

[3] Javier D. Fernández Miguel A. Martínez-Prieto and Rodrigo Cánovas. Querying RDF Dictionaries in Compressed Space. In *In Proceedings of the 27th Annual ACM Symposium on Applied Computing, Paper*, pages 340–347, March 26 - 30 2012.

[4] Nicholas Gibbins and Nigel Shadbolt. Resource Description Framework (RDF), 4 Feb 2009.

[5] Eric Prud'hommeaux and A Seaborne. Sparql query language for rdf, w3c recommendation: http://wifo5-03.informatik.uni-mannheim.de/bizer/berlinsparqlbenchmark, January 2008. Last accessed 31 August 20119.

[6] Javier D. Fernández Miguel A. Martínez-Prieto Sandra Álvarez García, Nieves Brisaboa and Gonzalo Navarro. Compressed Vertical Partitioning for efficient RDF management. In *In Proceeding of the 17th Americas Conference on Information Systems (AMCIS 2011), article*, 2011.

[7] Roger Castillo, Christian Rothe, and Ulf Leser. Rdfmatview: Indexing rdf data for sparql queries. In *Conference: 6th Int. WS on Scalable Semantic Web Knowledge Bases, At Shanghai, China*, December 2010.

[8] Nassima Soussi and Mohamed Bahaj. Semantics preserving sql-to-sparql query translation for nested right and left outer join. In *Journal of Applied Research and Technology 15*, page 504–512, 2017.

[9] Carsten Andreas Gerlhof André Eickler and Donald Kossmann. A performance evaluation of oid mapping techniques. In *VLDB*, 1995.

[10] Kevin R. Lawrence and Hongmei Chi. Framework for the design of web-based learning for digital forensics labs. In *ACM-SE 47 Proceedings of the 47th Annual Southeast Regional Conference Article No. 76*, page 4, 19 - 21 March 2009.

[11] Ahmed Elnaggar. The Semantic Web, November 2015.

[12] Frank Van Harmelen Jeen Broekstra and Arjohn Kampman. Sesame: A generic architecture for storing and querying rdf and rdf schema. In *The Semantic Web - ISWC 2002, First International Semantic Web Conference*, pages 54–68, 9-12 June 2002.

[13] Semantic web stack: https://en.wikipedia.org/wiki/. Last accessed 17 August 2019.

[14] Abakar Bouba. RDF Data Compression Techniques in a Highly Distributed Context, June 4 2019. Master of Science Thesis, Computer Science School of Informatics, University of Bonn.

[15] Georg Lausen Christoph Pinkel Michael Schmidt, Thomas Hornung. SP2Bench: A SPARQL Performance Benchmark. In *Conference paper to appear in Proc. ICDE'09*, 21 October 2008.

[16] R. Maharjan, Y. Lee, and S. Lee. Exploiting path indexes to answer complex queries in ontology repository. In *2009 International Conference on Computational Science and Its Applications*, pages 56–61, June 2009.

[17] Amann B. & Curé O. Naacke, H. Sparql graph pattern processing with apache spark. In *Proceedings of the Fifth International Workshop on Graph Data-management Experiences & Systems (p. 1). ACM.*, May 2017.

[18] Openjena. jena a semantic web framework for java: http://jena.apache.org/index.html, 2010. Last accessed 31 August 2019.

[19] Prasad Kulkarni. Distributed SPARQL Query Engine Using Mapreduce, 2010. Master of Science Thesis, Computer Science School of Informatics, University of Edinburgh.

[20] Parallel techniques for big data: https://bda2013.univ-nantes.fr/files/bda-2103-bigdata.pdf. Last accessed 18 November 2018.

[21] Wadhwani Khushboo and Dr. Yun Wang. Big data challenges and solutions. In *Big Data Challenges*, February 2017.

[22] M. N. Kalimoldayev, V. Siladi, M. N. Satymbekov, and L. Naizabayeva. Solving mean-shift clustering using mapreduce hadoop. In *2017 IEEE 14th International Scientific Conference on Informatics*, pages 164–167, November 2017.

[23] D. Borthakur. The hadoop distributed file system: Architecture and design. hadoop project website, 11(2007), 21.

[24] Venkat Ankam. Big data analytics, packt publishing ltd, 2016: https://www.packtpub.com/big-data-and-business-intelligence/big-data-analytics.

[25] Dean Jeffrey and Ghemawat Sanjay. Mapreduce: Simplified data processing on large clusters. *Communications of the ACM - 50th anniversary issue: 1958 - 2008*, 51(1):107–113, January 2008.

[26] D Marpe, G Blattermann, and J. Ricke. A two-layered wavelet-based algorithm for efficient lossless and lossy image compression.ieee transactions on circuits and systems for video technology, 10(7), 1094-1102, 2000.

[27] Javier D. Fernández, Miguel A. Martínez-Prieto, Claudio Gutiérrez, Axel Polleres, and Mario Arias. Binary rdf representation for publication and exchange (hdt). *Web Semantics: Science, Services and Agents on the World Wide Web*, 19:22–41, 2013.

[28] Prud'hommeaux E and Seaborne A. SPARQL Query Language for RDF: http://www.w3.org/tr/rdf-sparql-query/, 2008.

[29] Scala: A general purpose programming language: https://www.scala-lang.org/. Last accessed 3 October 2018.

[30] Apache spark:a apache spark an open source framework and a engine for large scale data processing: https://spark.apache.org/. Last accessed 20 September 2018.

[31] Scala and spark overview spark: https://pieriandata.com. Last accessed 17 October 2018.

[32] Apache spark vs hadoop: https://www.dezyre.com/article/hadoop-mapreduce-vs-apache-spark-who-wins-the-battle/83/. Last accessed 25 October 2018.

[33] Apache spark vs hadoop: https://www.dezyre.com/article/hadoop-mapreduce-vs-apache-spark-who-wins-the-battle/83. Last accessed 20 August, 2019.

[34] Jassem YAHYAOUI. Spark vs mapreduce in hadoop: https://yahyaouijassem.wordpress.com/2016/04/23/spark-vs-mapreduce-dans-hadoop/. Last accessed 17 October 2018.

[35] Giménez García and José Miguel. Scalable rdf compression with mapreduce and hdt. In *Extended Semantic Web Conference (ESWC)*, 2015.

[36] Giulio Ermanno Pibiri Raffaele Perego and Rossano Venturini. Compressed indexes for fast search of semantic data: arxiv.org > cs > arxiv:1904.07619v2, April 17th 2019.

[37] Thomas Neumann and Gerhard Weikum. Rdf-3x: a RISC-style engine for RDF. In *Proceedings of the VLDB Endowment, v.1, 647-659*, January 2008.

[38] A Martínez-Prieto, M, D Fernández, J, and s Cánovas, R. Compression of rdf dictionaries. In *Proceedings of the 27th Annual ACM Symposium on Applied Computing ACM*, pages 340–347, March 2012.

[39] P Ravi and A. Ashokkumar. A study of various data compression techniques. In *International Journal of Computer Science and Communication, Volume 6, Issue 2, Impact Factor 2.5,: www.csjournals.com*, April - September 2015.

[40] Han X. Interlandi M. Mardani S. Tetali S. D. Millstein T. D. & Kim M. Gulzar, M. A. Interactive debugging for big data analytics, June 2016.

[41] Claus Stadler, Gezim Sejdiu, Damien Graux, and Jens Lehmann. Sparklify: A Scalable Software Component for Efficient evaluation of SPARQL queries over distributed RDF datasets. In *Proceedings of 18th International Semantic Web Conference*, 2019.

[42] Christian Bizer and Andreas Schultz. The berlin sparql benchmark. *Int. J. Semantic Web Inf. Syst.*, 5:1–24, 04 2009.

[43] Yuanbo Guo, Zhengxiang Pan, and Jeff Heflin. Lubm: A benchmark for owl knowledge base systems. *SSRN Electronic Journal*, January 2005.