RDF Data Compression Techniques in a Highly Distributed Context

Abakar Bouba

Matriculation number: 2825222

March 20, 2019

Master Thesis

Computer Science

Supervisors:

Prof. Dr. Jens Lehmann
Dr. Damien Graux
Mr. Gezim Sejdiu

INSTITUT FÜR INFORMATIK III

RHEINISCHE FRIEDRICH-WILHELMS-UNIVERSITÄT BONN

Declaration of Authorship

I, Abakar Bouba, declare that this thesis, titled "RDF Data Compression Techniques in a Highly Distributed Context," and the work presented therein 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 of the main sources of help.
- Where the thesis is based on work done in collaboration with others, I have made clear exactly what was done by others and what I have contributed myself.

Signed: _____

Date: 13.03.2019

Acknowledgements

First and foremost, I am very much indebted to Almighty Allah, the most Merciful without whose patronage and blessing this master's thesis would not have been successfully completed.

With this note, I extend my deepest gratitude and appreciation for my supervisors, Dr. Damien Graux and Mr. Gezim Sejdiu, for their continuous support of my thesis as well as for their patience, enthusiasm, encouragement and immense knowledge. They provided me with great guidance, technical support, and constructive suggestions. One could not have asked for more perfect supervisors for my master's thesis. I must also express my gratitude on the enormous time they have spent in assessing the various phases of my project work. Their commentaries, criticisms, and observations through the various stages of my work have contributed significantly in augmenting both the quality and content of my work.

Besides my supervisors, I want to take a moment to thank Prof. Dr. Jens Lehmann and Dr. Hajira Jabeen for letting me pursue my master's thesis at the Smart Data Analytics (SDA) group at the University of Bonn (at the Department of Computer Science).

This note would not be complete without a mention of my family, alongside their unwavering support, without which this thesis would not have been possible.

Abbreviations

RDD . . . . . Resilient Distributed Datasets

SANSA . . . . . Semantic Analytics Stack

RDF . . . . . Resource Description Framework

SDA . . . . . Smart Data Analytics

SPARQL . . . . . SPARQL Protocol and RDF Query Language

HDFS . . . . . Hadoop Distributed File System

W3C . . . . . World Wide Web Consortium

HTML . . . . . Hypertext Markup Language

RDBMS . . . . . Relational Database Management System

RAM . . . . . Random Access Memory

CPU . . . . . Central Processing Unit

RLE . . . . . Run Length Encoding

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

PBs . . . . . PetaBytes

LUBM . . . . . Lehigh University Benchmark

BSBM . . . . . Berlin SPARQL Benchmark

MR . . . . . MapReduce

NFS . . . . . Network File System

Hadoop . . . . . Highly Archived Distributed Object Oriented Programming

Contents

List of Tables

List of Figures

Listings

Abstract

The ever-increasing volume of data generated through online shopping, cameras, Facebook, YouTube, banks, and other online platform is posing substantial storage challenges. The optimization of disk space for storing such a volume of data is, consequently, of optimum importance. To use less disk space while saving such a volume of data, one must compress data to a reduced size that will take up less space. This process might be achieved through data compression. Data compression is a cheap process of saving disk space and can be applied to all data formats, whether text, image, sound, or video.

The primary goal of this thesis is to implement a compression technique on a dataset for SANSA Stack [1] by using the Resource Description Framework (RDF) dictionary approach and then by comparing and testing our system with the standard dataset (normal dataset). To tackle these areas, we have used Spark and Scala to build a scalable and parallel distributed system. We designed and implemented a compression system that has one main feature—optimizing the space on the disk. The idea of RDF data compression is to reduce data size by maintaining a single copy for each record and by vertically partitioning a dataset. As a result, our compression technique provides an excellent compression ratio, from 70% to 82%.

Keywords: RDF, Data Compression, Semantic Web, Lehigh University Benchmark, Berlin SPARQL Benchmark, Evaluation.

Chapter 1

Introduction

This chapter looks into the resource description framework, or RDF as it is known [2] is a simplified scheme that structures and links data [2, 3]. It represents a model of knowledge that uses the properties of S, P and O, i.e. the subject, the predicate, and finally the object. To elaborate, we understand the subject as the resource that is being described, while the property is the predicate, and the value linked to it is known as the object. The popularity of this framework has fueled development when it comes to RDF stores – this typically has a significant role in the data web. RDF stores will typically support its storage, and can help find relevant infrastructure to link to it with SPARQL Protocol And RDF Query Language (SPARQL) query interfaces [4]. Even though the elevated number of RDF in presence can be conducive for semantic processes, it's also going to lead to RDF store bottlenecks [2, 5]. The normal volume of an RDF dataset is extremely large-sometimes crossing in terabytes. Storing, processing, and analyzing such a dataset require high-configuration machines or a distributed system. Spark library provides a distributed solution with the open source, general-purpose, distributed, cluster-computing framework called Apache Spark. Spark provides noticeable performance benefits: for example, some-times hundred times faster over traditional processing methods when data is completely fit in memory and access frequently. As data volume increases, Spark spills off the data which can't be handle into a persistent storage that increases the disc Input and Output (IO) and read/write time, which, in turn, causes a higher query-processing time. Hence, if the dataset can be compressed and can thus reduce the data size, we could see a drastic reduction in storage, memory requirements and processing times. There is also an opportunity to reduce the size of the RDF dataset. The RDF dataset contains many redundant records (in subject, predicate, and object). Sometimes, redundant records occupy around 80% of the space. Storing these redundant records impacts the performance of our system. Due to the inability of the RDF semi-structured format to fit into the relational model solutions that have been proposed for RDF stores, native solutions have been designed to provide solutions to the problem. These native solutions have not been able to take care of scalability and space requirement issues. This is the reason that scalability has presented itself as a significant problem, and has restricted many prominent RDF-powered applications because when it comes to larger scale deployments the traditional solutions are no longer viable [6]. If we could avoid or re-place redundant records with small-sizes, values, then data size would be reduced drastically, and the performance time would be sped up as a small dataset can easily fit into memory. Accordingly, there is need to engineer an appropriate compression technique so as to reduce the storage space taken up by RDF stores. Compressed vertical partitioning for efficient RDF management is the

use of a vertical partitioning model in RDF dictionaries of compressed spaces. [7, 8] proposed an RDF dictionaries compression technique that makes it possible RDF triples to be replaced in the long terms with short IDs that reference those long terms. This technique will allow one to compress a huge dataset and to reduce its scalabilities problems.

1.1 Questions

The following questions have been set as guidelines to fulfill this aim.

1. What is the efficiency of RDF dictionary compression techniques compare to other compression techniques?
2. How fast is the query processing over compressed data with compare to the normal data (Normal graph database)?

1.2 Objectives

Scalability, fault-tolerance processing, and storage engines are required to process and analyze a huge dataset in an efficient manner. The huge volumes of these datasets have been termed big data. They are so large and complex that traditional data-processing software or systems cannot handle them [9]. These normal solutions cannot take care of scalability and space-requirement issues. For semantic analysis, huge web datasets are generated [1] and are converted into a standard RDF format, containing a subject, predicate, and object (S, P, O). The main objective in this research is to reduce the space taken by RDF stores in SANSA [1]. Hence, this project addresses the scalability and space requirement issues by reducing the space required for organizing and storing RDF data [7]. Another alternative way to address the scalability problem is to distribute the data across clusters and to process it with an efficient in-memory cluster-computing engine such as Apache Spark [10]. We, accordingly, propose a compression system that partitions a large RDF dataset and saves the results to disk. The output reduces the size to nearly 82% of the disk.

1.3 Thesis Structure

This study's structure is explained henceforth. The Background Chapter includes some background knowledge about technologies used throughout the thesis. In the Related Work Chapter, an overview of the related work that has been implemented in the past is provided. In the Approach Chapter, the approach used for the RDF compression technique is discussed. And also the insights about our system are provided and code examples of our system are mentioned. In the Evaluation Chapter, our implementation is evaluated in a distributed cluster. In the Conclusions and Future Work Chapter, I conclude with a perspectives to future work.

[1] https://lod-cloud.net/

Chapter 2

Background

2.1 The Semantic Web

The semantic web refers to W3C's vision of a web of linked data [1]. It was built as a common framework that allows data to be shared and reused across application boundaries and enables people to create data stores on the web, to build vocabularies, to write rules for handling data, to use RDF as a flexible data model, and to use an ontology to represent data semantics [11]. Linked data is empowered by technologies such as RDF and SPARQL. The semantic web can also be interpreted as an enhancement of the current web, in which information has a well-defined meaning,and it thus better enables machines and humans to interact [11]; furthermore, it facilitates the communication of information in a format set by the W3C's and makes machines perceive hyperlinked information.

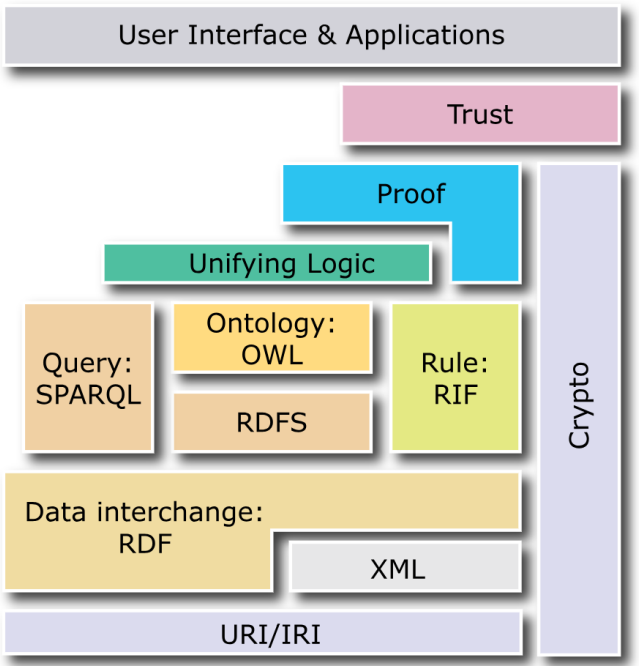

Figure 2.1: Semantic Web Architecture [12]

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

From the the figure 2.1, we can see how data such as XML, OWL, RDF,and OWL are involved.

2.1.1 RDF Model

The RDF is a typical model that allows data exchange online. It was created as metadata for a data model originally. However, in time, it has become a method to model information or conceptualize something that is to be executed on the web, via multiple data serialization formats and syntax notations [13]. The following are some of the most popular serialization formats:

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

The Figure 2.2 below show a simple example of an RDF triples:

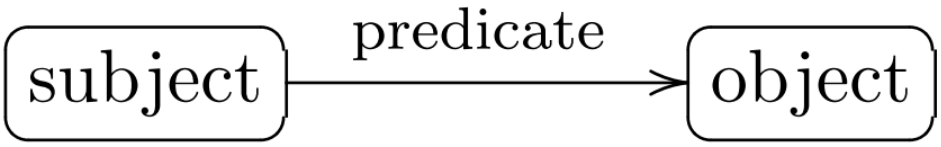

Figure 2.2: A Simple Example of an RDF Triple

As Figure 2.2 illustrates, subjects and objects are represented as nodes, while predicates are represented as arcs.

N-Triples

`N-Triples`[9] is a line-based and plain text format for encoding an RDF graph. The general form of each triple can be described as $< Subject >< Predicate >< Object >$, which is separated by whitespace and terminated by '.' after each triple [14].

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

1. `The subject` contains URIs and empty nodes.
2. `The predicate` contains URIs.
3. `The object` contains URIs, empty nodes, and literals

We can describe URIs, empty 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. `Empty nodes or blank nodes` are used to represent 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 [15] is a query language of the semantic web that is used to fetch unstructured, structured, and semi-structured data. For example, one might examine the following example of query RDF data in a SPARQL base on chain pattern from a LUBM benchmark:

Listing 2.1: SPARQL Query Example for the Graph in figure 2.3

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

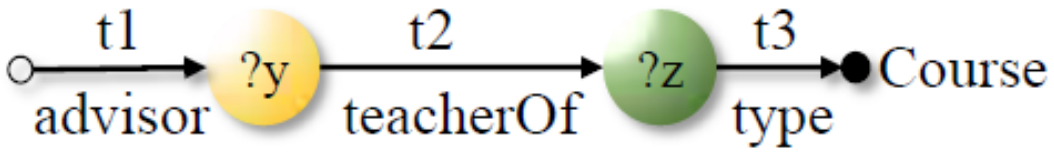

Figure 2.3: Query RDF Data in SPARQL [16]

2.2 Big Data

In a digital world, tons of structured and unstructured data are generated every day and are later analyzed for business purposes. This data can be generated by all kinds of devices such as mobile devices, sensors, the Internet, social networks, computer simulations, and satellites [17]. The massive volumes of these datasets has been termed big data. This data is so large and complicated that traditional data-processing systems cannot deal with them [9]. In recent years, many researchers have proposed techniques for reducing the volume of data when storing it. We can cite here, "Data Compression" by [7, 18], and many more.

Big data is beyond storage capacity and processing. Hence, Big data is define as `3Vs` [10]: volume of data, variety of data, and velocity of data.

1. `volume of data.` Data is collected from organizations or companies through business transactions, social media, information from sensors, airlines, and so on. [19].
2. `Variety of data.` "Data streams unparalleled speed of velocity and have improved in a timely manner. Sensors and smart metering are driving the need to deal with the overflow of data in real-time operations."
3. `velocity of data.` Data is found in many different forms, and can be categorized based on structures i.e. quantifiable such as numeric data, or unstructured such as text-based documents, emails, videos, audio files, financial transactions, and so forth [19].

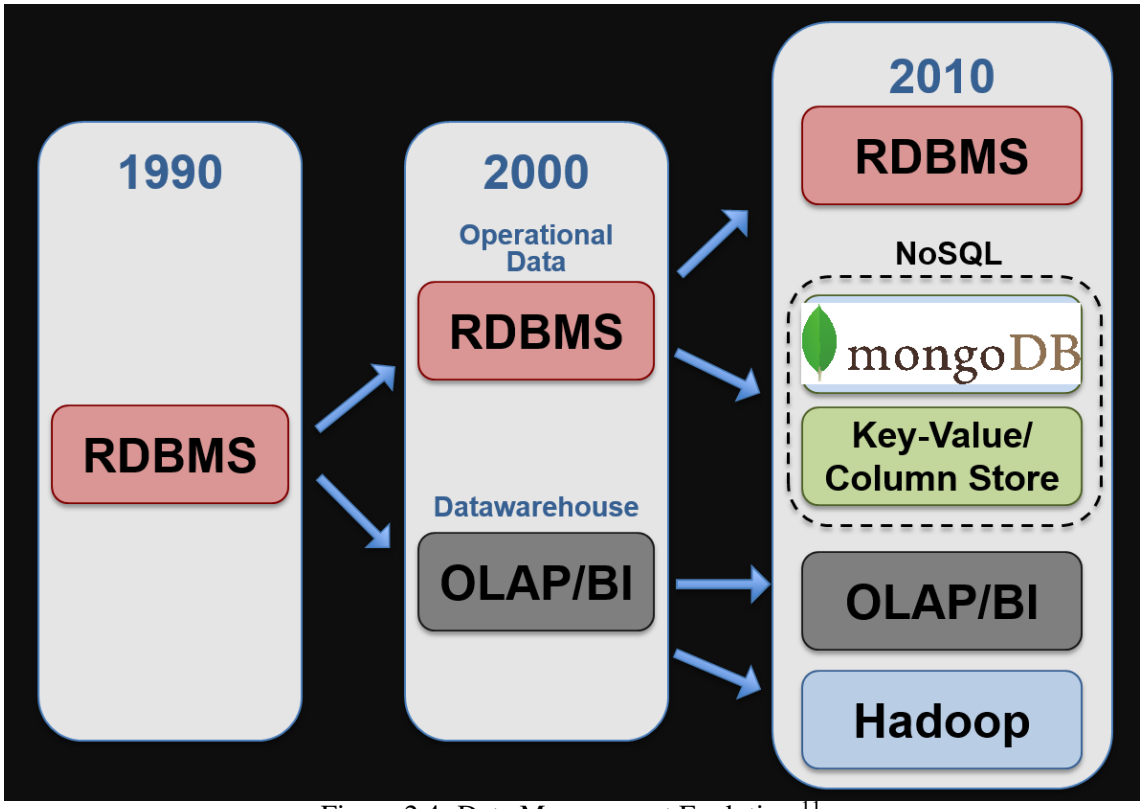

Figure 2.4: Data Management Evolution [11]

The storage proved to be a big problem, however, new technologies like Hadoop MR and Apache Spark has reduced the issues.

[10] https://intellipaat.com/tutorial/hadoop-tutorial/big-data-overview/
[11] http://practicalanalytics.co

Hadoop

Hadoop is an open source software framework technology that helps to store, access and gain significant resources from big data in large distributed files of data over many computers at low cost, with an extraordinary level of fault tolerance and important scalability [20].
Hadoop Features Include:

1. `Distributed.` A cluster in Hadoop is made up of several machines connected together.
2. `Scalable.` New machines can be added in order to process data fast.
3. `Fault tolerant.` In case any machine fails, the machine can be replaced since many machines are working together.
4. `Open source.` It is overseen by Apache Software Foundation. For storage, we used the Hadoop Distributed File System (HDFS), and for processing, we used MapReduce, which allows computations on that data.

The Figure 2.5 shows the Hadoop architecture with its components; its composed of the HDFS and the Map-Reduce. The HDFS It is used for storing and processing of huge datasets, while MapReduce is a way of splitting a computation task to a distributed set of files. A Hadoop cluster consists of one master node and multiple slave nodes; the master node includes a data node. The name node, job tracker and task tracker; the slave node works as a data node and task tracker; and the job tracker handles the job scheduling [20].

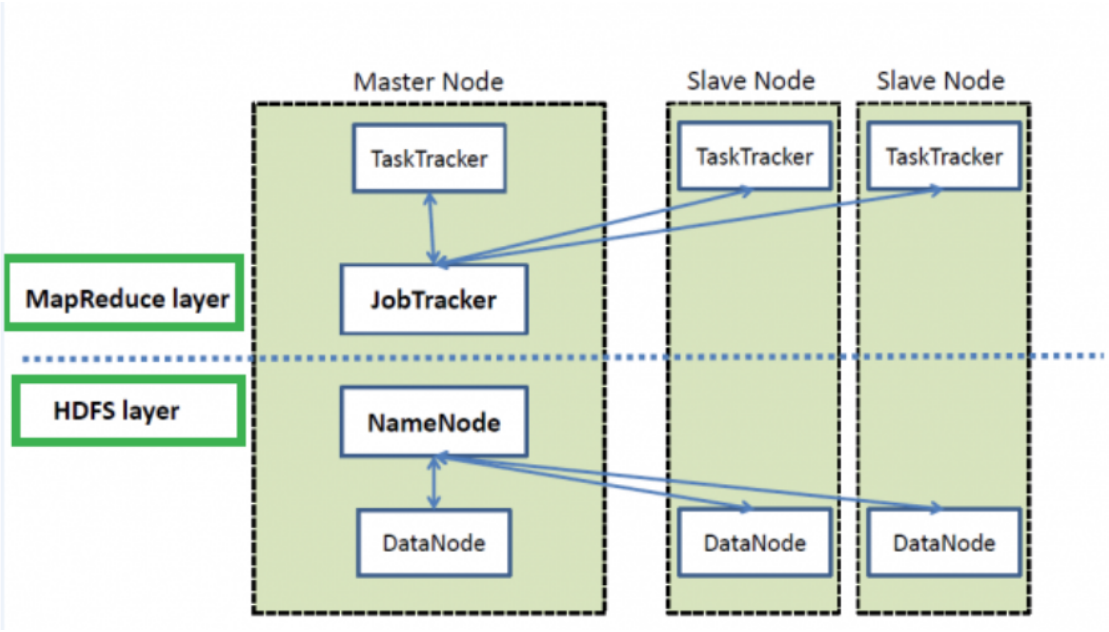

Figure 2.5: Hadoop Architectures [20]

HDFS

HDFS is designed to be executed through commodity hardware. It is used for storing and processing of huge datasets with a cluster (that is, a group of machines in a LAN) of commodity hardware. It differs from others because it is extremely 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 copies of a block prevent the loss of data due to a failure of a particular node.

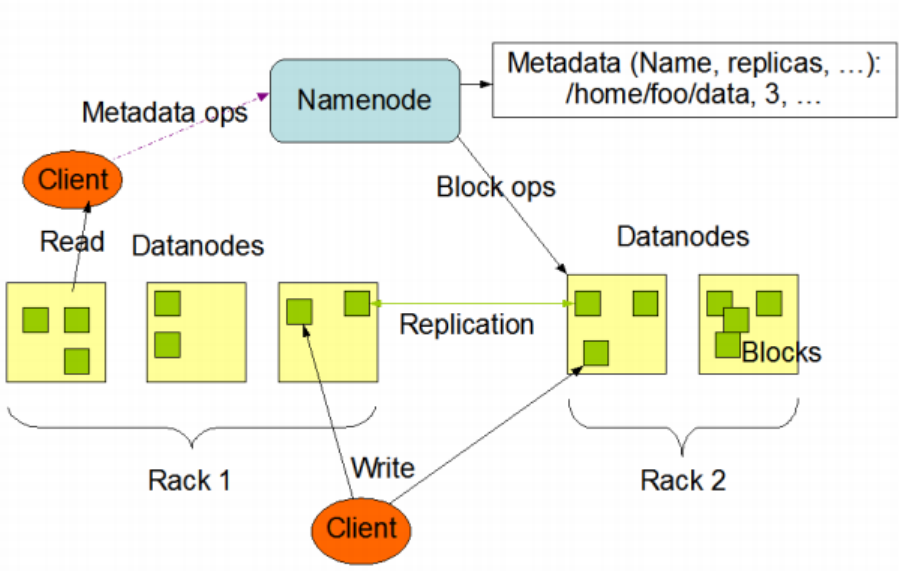

Figure 2.6: HDFS Architecture [21]

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. Smaller blocks provide more parallelism during processing. By having multiple copies of a block, loss of data due to a failure of a node is prevented. Features of HDFS [22]:

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 rebalancing`. The HDFS rebalancing feature rebalances the data uniformly across all data nodes in the cluster.

MapReduce

MapReduce is a way of splitting a computation task to a distributed set of files such as HDFS. We can define it as a programming model for data processing. It consists of a job tracker and multiple task trackers. MapReduce works on structured and unstructured data from HDFS. It is mainly designed to run in a parallel environment to facilitate or accelerate the computation on large data. This works simultaneously by creating a set of independent tasks for any given job. This results in better reliability and improved speed. Hadoop can execute a MapReduce program in various languages, such as Ruby, Java, Python, and many more. MapReduce is divided into two phases:

1. `The map phase.` This where data is processed for the first time. This 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) [12] 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.

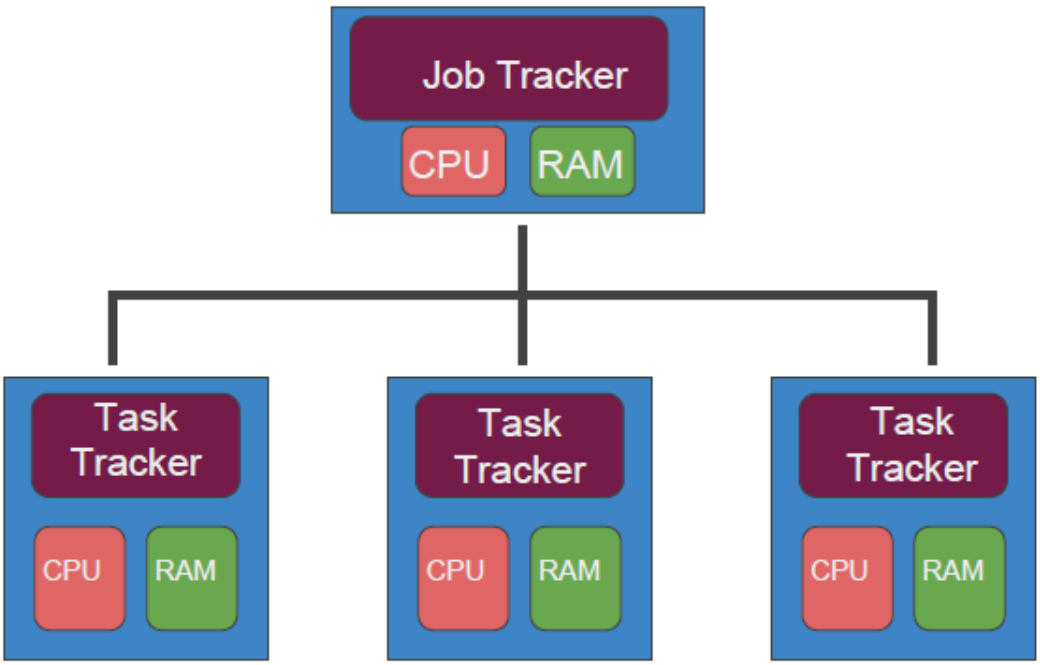

Figure 2.7: Architecture HDFS [13]

[12] https://en.wikipedia.org/wiki/Attribute-value_pair
[13] https://en.wikipedia.org/wiki/Attribute-value_pair

Figure 2.7 shows how the job tracker sends the code to run on the task trackers and then how the task trackers allocate the CPU and memory for the tasks and thus monitor the tasks on the worker nodes.

2.3 Data Compression

Data compression is the process of modifying, encoding, or converting the data structure of large datasets in such a way that data consumes less space on disk or memory. It can be applied to all data formats (text, images, sound, and video). In compression, there are two types of techniques [23]: lossless and lossy compression.

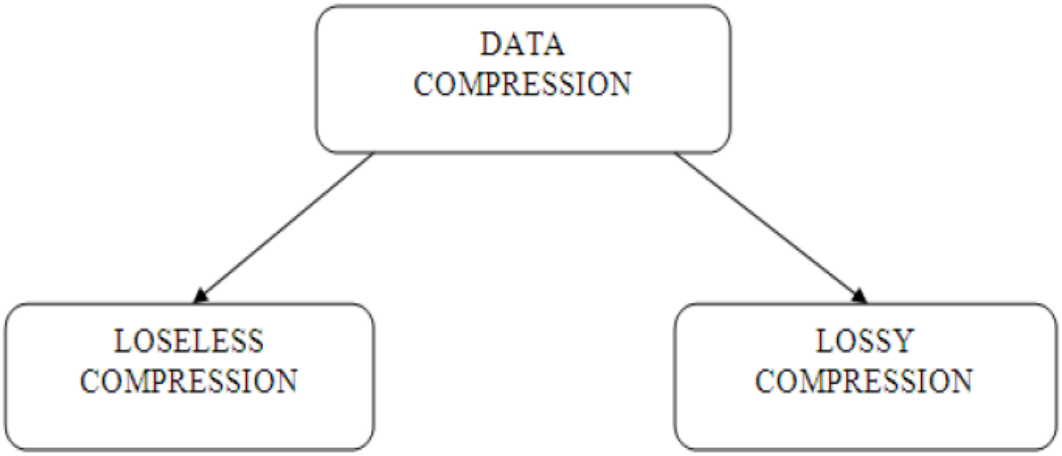

Figure 2.8: Types of Data Compression

1. `Lossless compression` [23], uses compressed data to reconstruct the original data through use of data compression algorithms. The compressed version is the exact replica of the original, bit for bit. Lossless compression is most commonly used to compress text and data files.
2. `Lossy compression is also called compression that is irreversible` [23]. This is where the original data is discarded and an inexact approximation is used during data encoding. This is normally used when working on images, video or sound files.

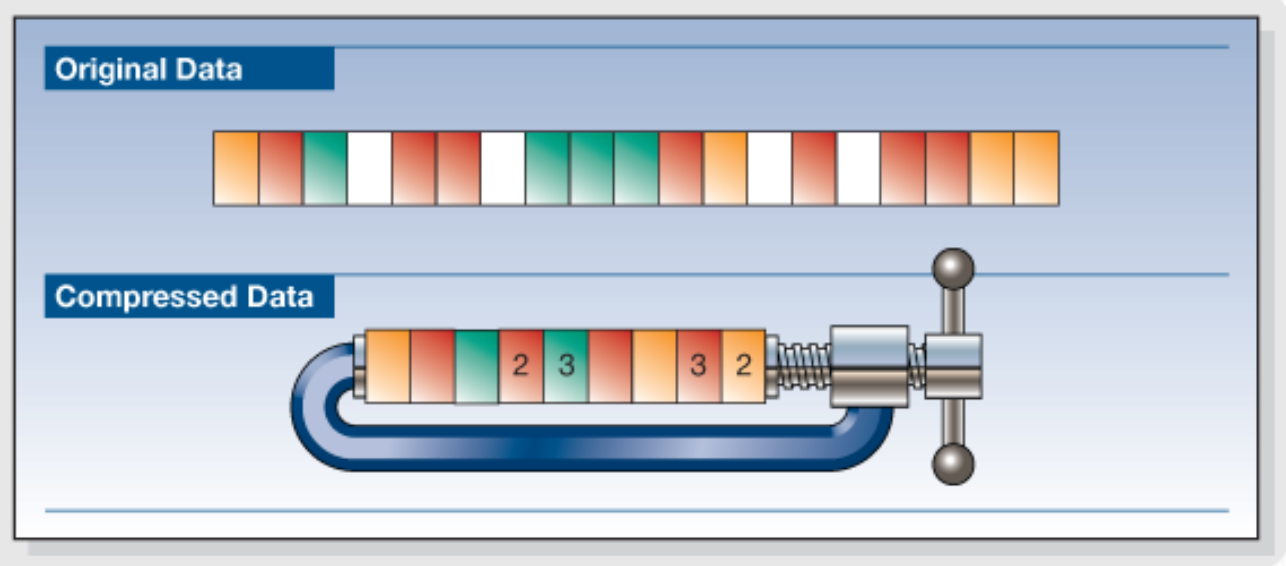

Figure 2.9: Basic Compression Techniques[14]

Figure 2.9 shows how data is reduced from the original size to the size of compressed data after getting rid of duplicate data. This is one way to optimize space.

2.4 Scala and Spark

2.4.1 Scala

Scala [24] is a general-purpose programming language. "It was designed by Martin Odersky in the early 2000s at the Ecole Polytechnique Federale de Lausanne (EPFL), in Switzerland.[15] Scala source code is intended to be compiled to Java bytecode so as to run on a Java virtual machine (JVM)"; Java libraries may be used directly in Scala. Unlike Java, Scala has many features of functional programming. [16] Scala demand has dramatically risen in recent years because of Apache Spark. [17] Scala is the fourth most demanded programming language in the world; It is widely used by some of the largest companies such as Twitter, LinkedIn, FourSquare, and more.

2.4.2 Spark

Apache Spark [25] is one of the latest technologies for quickly and easily handling big data. It is an open source project on Apache, which was first released in February 2013 and has exploded in popularity due to its ease of use and speed. It was created at the AMPLab at UC Berkeley [22]. Today, Spark is considered a flexible alternative to MapReduce.
Spark can use data stored in a variety of formats:

- HDFS

[14] https://clementandjohn.weebly.com/data-compression.html
[15] https://en.wikipedia.org/wiki/Martin_Odersky
[16] https://en.wikipedia.org/wiki/Java_bytecode
[17] https://www.infoworld.com/article/3216144/spark/the-rise-and-predominance-of-apache-spark.html

- Cassandra
- Other formats

Some Spark features are [22]:

1. `Easy development.` Multiple native APIs such as Java, Scala, R, and Python.
2. `Optimized performance.` Caching, optimized shuffle, and catalyst optimizer.
3. `High-level APIs.` Data frames, data sets, and data sources APIs.

Spark RDDs

Resilient distributed datasets (RDDs) are a distributed memory abstraction that allow programmers perform in-memory computations on a huge clusters in a fault-tolerant way. It enables efficient data reuse in a broad range of applications and fault-tolerant, parallel data structures. Spark RDDs are created by transforming data in stable storage through the use of data flow operators such as map, group-by, filter; in addition, they can be cached in memory across parallel operations.

- `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 types of operations that an RDD supports (it is also used for any computation in data):

1. `Transformation.` This means modifying a dataset from one dataset to another dataset. Any data modification leads to a transformation: for example, by multiplying numbers, adding a number, adding two different datasets, or joining two datasets. The transformation uses the following functions:
    - `map()`, is used to transform the dataset into another based on our logic. For instance, 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.
2. `Action.` This mean doing computation on our dataset.
    - First
    - Collect
    - Count
    - Take

A resilient distributed dataset (RDD) has four main features:

1. A distributed collection of data.

2. It is fault-tolerant.
3. It has the ability to handle parallel operation.
4. It has the ability to use many data sources.

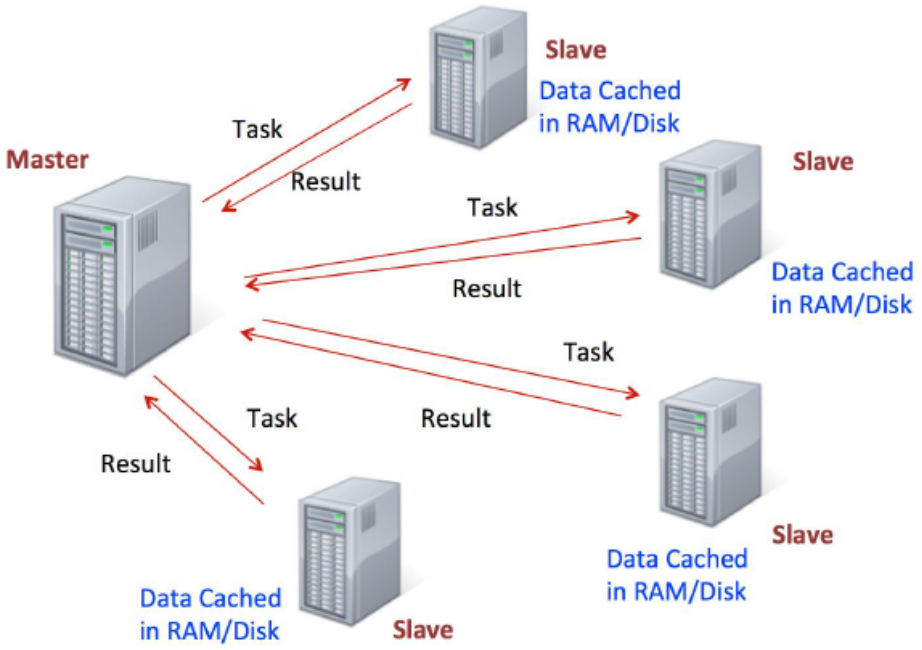

Figure 2.10: Overview of Spark RDDs [26]

2.5 Local Versus Distributed Systems

Local Systems

A local process uses the computation resources of a single cluster. This means that if one has to run a huge dataset, it will take more time to process data.

Distributed System

This has access to the computational resources across a number of machines connected through a network. This can be compared to a local system, where the processing is extremely fast. Hadoop will distribute a dataset across several clusters or computers. Distributed machines also have the advantage of easily scaling (one can add more machines). They also include fault tolerance, which means that if one machine fails, then the whole network can continue to operate. However, in a local cluster, if one computer goes down, then the whole system will go down.

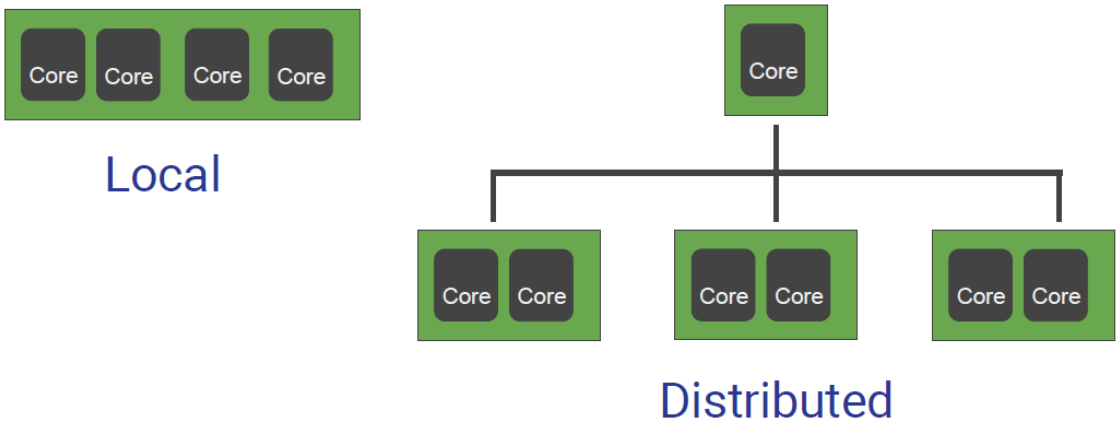

Figure 2.11: View of the Local and Distributed Systems [26]

2.6 Hadoop Versus Spark

Both `Hadoop` and `Spark` are frameworks of big data. We have noticed that Spark is considerably faster than Hadoop when processing a large dataset. Nevertheless, one can also compare the implementation of a word-count example through the use of Hadoop and Spark with the Scala programming language.

Table 2.1: Apache Spark Versus Hadoop [27]

Apache Spark Versus Hadoop	
Apache Spark	**Hadoop**
Easier to program and has no abstraction requirement.	Hard to program and needs abstractions.
Written in scala.	Written in Java.
Developers can perform streaming, batch processing, and machine learning, all in the same cluster.	Here, It is used for generating reports that help find responses to historical queries.
In-built interactive mode.	No in-built interactive mode, except tools such as Pig and Hive.
Programmers can edit the data in real-time through Spark Streaming.	Enable one to process a batch of save data.

Table 2.1 explains the basic difference between Apache Spark and Hadoop. Figure 2.12 provides a view of both frameworks. One should explain here the memory-based computation in disk and memory.

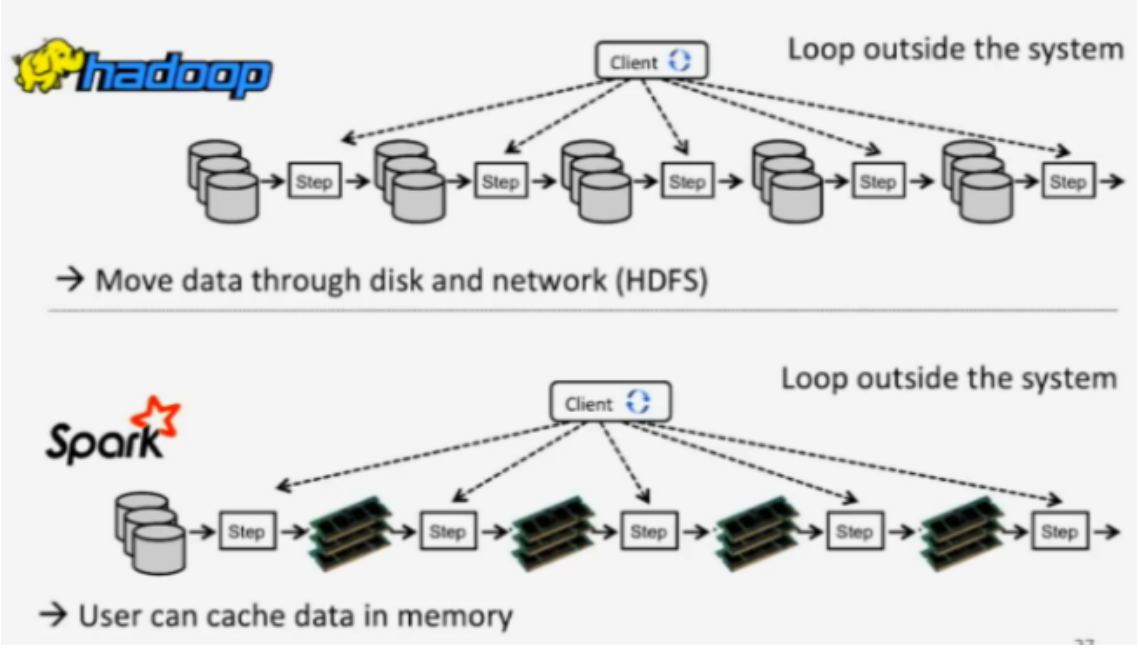

Figure 2.12: View of Hadoop Versus Spark [28]

Chapter 3

Related Work

RDF data compression plays a huge role in big data applications. With the increase of data providers who are publishing their data in RDF format, the size of RDF data is increasing every second [29]. Accordingly, to efficiently manage the scalability of an RDF dataset is becoming an issue. However, there are various existing compression techniques that can assist in reducing the size of an RDF dataset. We have such system that allows us to compress RDF datasets throught the use of different compression techniques. We are going to enumerate and discuss some of those compression techniques in short. In this chapter, we focus on previous work which is related to the problem stated in the Introduction.

3.1 Existing Compression Techniques

3.1.1 RDF3X

RDF3X [30]: introduces index compression so as to reduce the space usage of Hexastore. RDF3X creates its indexes over a single table with three columns and then stores them in a compressed clustered.

3.1.2 k2-Triples

`k2-Triples` [30]: This is a technique that is used for compressed, self-indexed structures, which was initially designed for web graphs. [30] proposed a K2 model that represents (subject, object) pairs in sparse binary matrices, that are efficiently indexed in compressed space via k2-tree structures achieving the most compressed representations in terms of a cutting edge baseline.

3.1.3 Dcomp

`The Dcomp` [7]: approach target a common attribute where the number of duplicates is huge. It then concentrates on eliminating redundant triples. Dcomp creates a subject, predicate and object dictionary in order to store a unique value. “All triples in the dataset can then be rewritten by changing the terms with their matching ID. Hence, the original dataset is now modeled over the dictionary created and the resultant ID-triples representation”. In Dcomp, dictionaries are

managed as per their role in a dataset [7]:

- A common `S`, and `O` organize all terms that play subject and object roles in the dataset. They are mapped onto the range [1, $|SO|$].
- `S` organizes all subjects that do not play an object role. They are mapped onto the range [$|SO|$+1, $|SO|$+$|S|$].
- `O` organizes all objects that do not play a subject role. They are mapped onto the range [$|SO|$+1, $|SO|$+$|O|$].
- `P` maps all predicates to [1,$|P|$].

3.1.4 Huffman Coding

`The Huffman coding` 3.1: is one of the most popular techniques for removing redundant data, targets the count frequency for each character in the input. The main job of this technique is to replace or assign short codes to a symbol that occur more frequently (that is, a redundant string or symbol) and longer codes where there is less frequent occurrence. An example that we can consider is that of a dataset that only employs the characters A, B, C, D & E. Each character must be given a weight based on how frequently it is used before it is assigned a pattern [31]. 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 [31]

3.1.5 Lempel-Ziv Compression

`The Lempel-Ziv` [31]: technique relies on recurring patterns to save data space. That is based on creating through a string of compressed symbols an indexed dictionary. The smallest sub-string is extracted under the algorithm if it cannot be found in the dictionary from the residual uncompressed version. A copy of this sub-string is then kept in the dictionary by the algorithm, which turns it into a fresh entry and provides it with a value within the index.

3.1.6 RLE

`RLE` [31]: is a technique that provides good compression of data containing many runs of the same value. It replaces sequences of identical symbols from a dataset. The main logic behind this process is the replacement of repeating items by one occurrence of the symbol followed by the number of occurrences.

Chapter 4

Approach

4.1 System Design

The RDF compression technique focuses on reducing the space requirement of an RDF dataset by eliminating or replacing duplicate records with a small value and by facilitate queries on top of compressed records. It must be ensured that meaning of a record remains intact, and it should be possible to recreate the same records by reversing the process. RDF compression combines the vertical partitioning and star schema approaches to represent and store the data. It generates a dimension table for each attribute that, contains the unique values zipped with an index. In other words, there is a single central fact table that, contains the reference index numbers of subject, predicate, and object pointing to value in a dimension table. A dimension table stores only unique values that make it small enough to persist and process datasets in-memory.
The main contributions of our system are as follows:

1. To compress RDF data by using a dictionary approach.
2. To process compressed RDF data by using a basic queries to output the query results.
3. To use Spark RDDs to store the results of a query in-memory (RAM).

4.2 RDF Dictionary Compression Architecture

To implement the RDF dictionary compression technique, we have used Spark and not Hadoop. We make use of Spark RDDs and then keep our compressed data in-memory; Spark will split the data into disk in case there is no memory available to fit all the data. This will speed up the process of fetching and processing compressed RDF data efficiently.

RDF data Compression
The following approach was used for our system.

1. We Loaded the RDF file through SANSA API; then we transformed and distributed the RDF in Spark RDD across the cluster node. SANSA [1] uses the RDF data model for representing graphs that consist of triples, with the S, P and O. The RDF dataset could include many graphs and note info on each one, making it possible for any of the lower layers of SANSA (querying) to make queries that involve information from more than one graph. Instead of directly dealing with RDF datasets, the target RDF datasets need

to be converted into an RDD of triples. The core dataset is created as per an RDD data structure, that is a primary building block for Spark. RDDs act as in-memory databases of records that could be conducted simultaneously with other larger clusters [32].

2. Later, the compressed RDD was persisted into an HDFS, which provides a scalable, fault-tolerance. Partitioned RDD was compressed with the RDF dictionary compression algorithm and persisted into the distributed Hadoop cluster. The HDFS ensured that the data was divided evenly across the cluster node in oreder to ensure balanced Input and Output processing.
3. On the application (re)start, instead of loading the RDF files sequentially from a local file system, the application loaded the RDF in memory (RAM) from the HDFS cluster nodes, where each spark executor is responsible for reading from one or more HDFS nodes.

Our compression technique works as follow:

1. Read the raw RDF dataset Data Frame. Vertically distribute the RDF dataset using Spark.
2. Extract the unique values of each field. Zipp that unique value with a unique index. In this way, replicate records are eliminated, and space is reduced. Later, each value is referred to by its index number instead of its value.
3. Generate a fact table from the input raw Data Frame by replacing the actual value with the corresponding reference index number.
4. This creates a central fact table that stores the reference index numbers; these refer to dictionary tables (subject, predicate, and object dictionaries).
5. The fact table ensures that relationships between the values are intact and can recreate a raw dataset.

4.3 SANSA Stack

4.3.1 Introduction

The work in this section uses SANSA [1], the data flow processing engine, for performing distributed computations over large-scale knowledge graphs; SANSA provides data distribution, communication, and fault tolerance for manipulating massive RDF graphs and for applying machine learning algorithms on the data at scale [1]. The main idea behind SANSA is to combine distributed computing frameworks in Spark and Flink with the semantic technology stack [1].

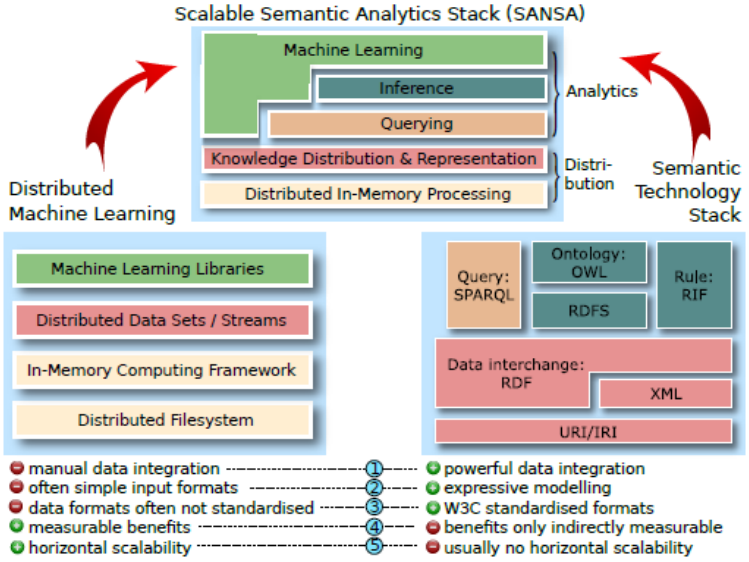

Figure 4.1: An Overview of the SANSA Framework, Combining Distributed Analytics (on the left side) and Semantic Technologies (on the right side) into a Scalable Semantic Analytics Stack (at the top of the diagram) [1]

Our application, which we have called RDF Dictionary Compression is implemented over a SANSA-Stack project. Currently, there are five layers in the SANSA Stack: the RDF, Query, Inference, Machine Learning, and OWL layer. For this project, we used the RDF [2].
In our system, we have provided an RDF data compression technique based on the dictionary approach that goes under SANSA-RDF.

[1] http://sansa-stack.net/
[2] https://github.com/SANSA-Stack/SANSA-RDF

SANSA-RDF
SANSA-RDF [3], occupies the bottom layer in the SANSA Stack which is a read/write data layer, by providing the facility for users to read and write RDF files of an n-triple, N-Quad, RDF/XML, and Turtle format; It also support Jena [4] interfaces for processing RDF data.

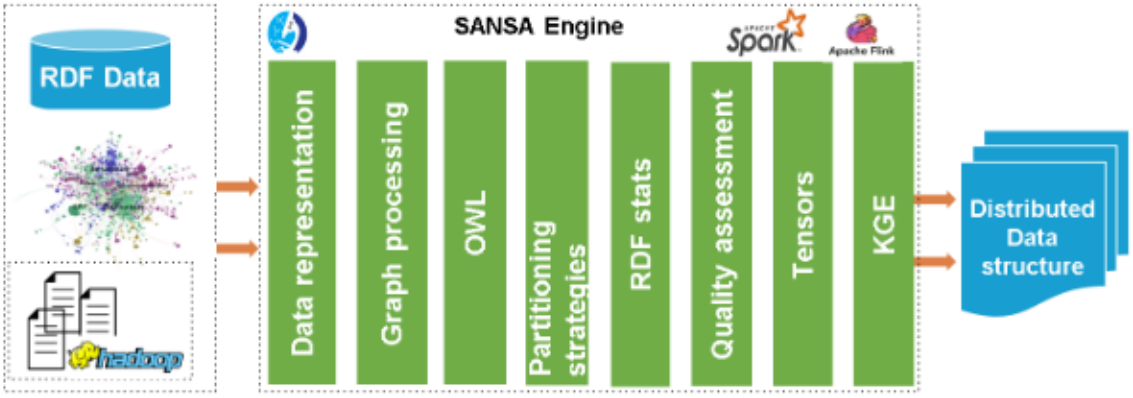

Figure 4.2: An Overview of SANSA-RDF Engine [5]

4.4 RDF Data Compression Algorithm

The compression algorithm leverages the features of both RDF vertical partitioning and star schema. Our compression algorithm splits the dataset vertically; for an N-triple dataset, accordingly, we obtain N partitions—a partition for each triple. Our proposed system is implemented over the SANSA project which, contains five components: an RDF data compression implementation; data loader; Query engine; record schema. I explain below briefly what each file is used for. After analyzing the dataset, we observed that the data had too many duplicates. In the proposed compression technique, we targeted this problem by storing a single value in a dictionary, which means that, we have eliminated all the duplicate value and store only a single copy of record. Each unique value of subject, predicate, and object are stores once in the dictionary (key, value) by assigning a unique number to a string value. Transformed RDD triples contains only the corresponding unique number.

[3] http://sansa-stack.net/introduction/
[4] https://jena.apache.org/
[5] http://sansa-stack.net/libraries/#RDF_OWL_API

Our compression technique performs the following steps to compress the data.

Algorithm 1 RDF Data Compression Algorithm

Result: It creates partitioned compressed dataset from N triple file, along with dictionaries for each triple(for subject, predicate, and object), stores distinct values zipped with unique index.

$dictSubject \leftarrow Dictionary()$
$dictObject \leftarrow Dictionary()$
$dictPredicate \leftarrow Dictionary()$
$compressedDataSet \leftarrow RDD[Row]$
while *next record* (s,p,o) *from* $N-triple\ file\ != null$ **do**
  if *s not exists in dictSubject* **then**
    $dictSubject \leftarrow dictSubject : +s$
  end
  if *p not exists in dictPredicate* **then**
    $dictPredicate \leftarrow dictPredicate : +p$
  end
  if *o not exists in dictObject* **then**
    $dictObject \leftarrow dictObject : +o$
  end
  $sIndex \leftarrow lookup(dictSubject, s)$
  $oIndex \leftarrow lookup(dictObject, o)$
  $pIndex \leftarrow lookup(dictPredicate, p)$
  $compressedDataSet \leftarrow compressedDataSet : +Row(sIndex, oIndex, pIndex)$
end

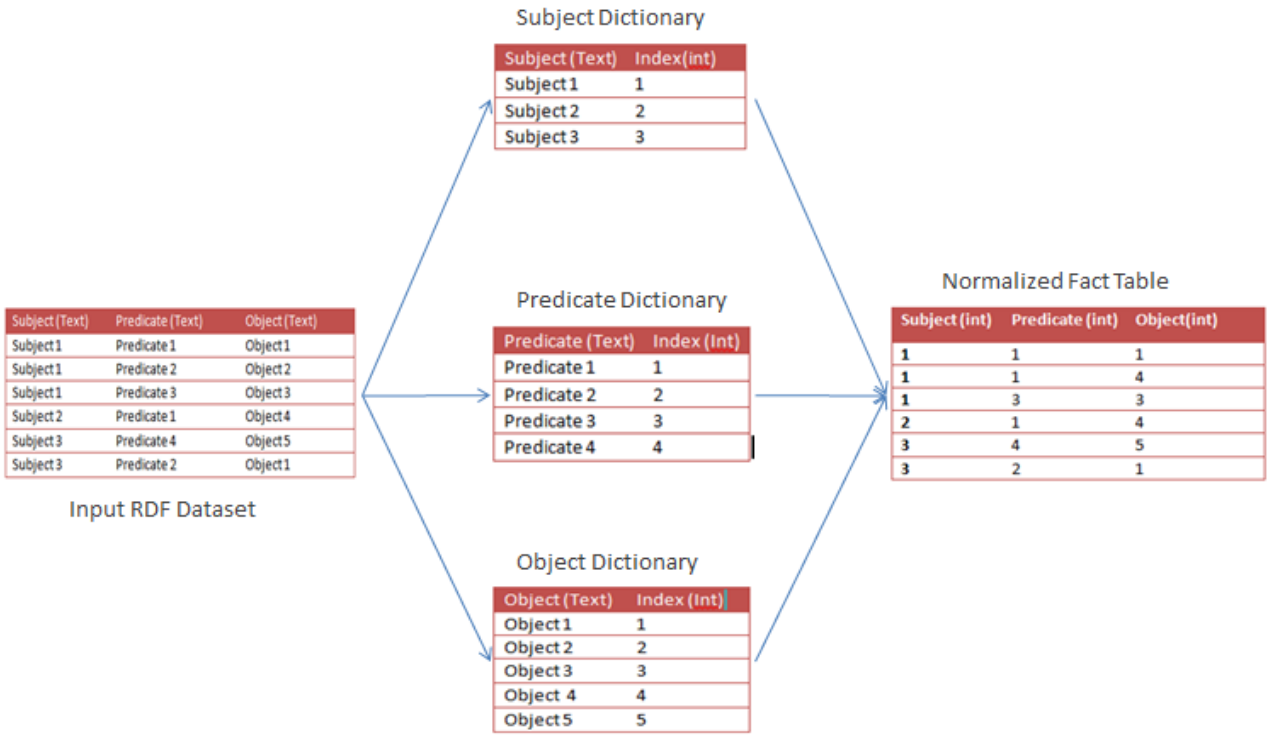

Figure 4.3: An Overview of the RDF Data Compression

Here, one should understand the implementation of our system through the framework of Scala and Spark software.

Listing 4.1: RDF Data Compression Code using Spark and Scala

```
1
2  val lang = Lang.NTRIPLES
3
4  // Read RDF dataset
5  val triples : RDD[graph.Triple]= spark.rdf(lang)(input)
6
7  // Create DataFrame
8  val triplesDF = spark.createDataFrame
9  ( triples .map(t=> Row(t.getSubject. toString ,
10 t.getObject. toString (),t. getPredicate .
11 toString ())),schemaTriple)
12
13 // Create the Dataframe for each dictionary .
14 val subjectDF= spark.createDataFrame( triples .
15 map(_.getSubject. toString ()). distinct ().
16 zipWithIndex().map(t=> Row(t._1,t._2)),
17 schema).cache();
18 val objectDF = spark.createDataFrame
19 ( triples .map(_.getObject. toString ()).
```

```
20  distinct () .zipWithIndex() .map(t=>
21 Row(t._1,t._2)),schema).cache();
22 val predicateDF = spark.createDataFrame
23 ( triples .map(_.getPredicate . toString ())
24 . distinct () .zipWithIndex() .map(t=>
25 Row(t._1,t._2)),schema).cache();
26
27 // Create Fact table for Triples .
28 val result = sql.sql(" select subject.index as s_index,
29 object.index as o_index, predicate.index as p_index from
30 triples join subject on triples . subject=subject.name"+
31 "join object on triples . object=object.name "+
32 "join predicate on triples . predicate=predicate.name");
33   }
```

SANSA provided us some functions such as getSubject, getPredicate, and getObject in our codes. This helped us to save some time and energy in not having to write those algorithms. Below is the explanation of the codes [6].

- `Line 4.` We read our input dataset that the user provided. After we read it, we stored it in a triple object.
- `Line 7.` We created a Data Frame (DF), so that we could perform a query on the dataset. We then created the row table which has three columns. The first column is Subject, the second is Object, and the third is Predicate.
- `Line 13.` Here, we created a dictionary, we apply a filter into the triple to get the distinct value of the subject, and then we zipped it with the index or numerical value. We did the same for the predicate and object.
- `Line 27.` We created a fact table for: triples, subject.index as s index, predicate.index as p index, and object.index as o index. For triples, we joined triples.object = object.name (that is to say, we compared the object with the object name), and we joined the predicate with predicate name. As a result, we obtained three columns of numerical values: the index of the subject, the index of predicate, and the index of the object. These results were stored in a dictionary result.

4.5 Query System for the Testing Purpose

We have applied the query system to compare the normal data used currently on SANSA and on our proposed system (the compression technique).
The query operation can be applied in two steps:

1. The query system applies the filter at the lowest level, that is to say, on the dictionary, which then returns a comparatively small result set. These multiple filter operations are applied simultaneously on different dictionaries at the same time.
2. This small result set is joined with the fact table to obtain an actual result.

[6] https://github.com/abakarboubaa

4.5.1 Record Schema

The record schema module provides the schema-related information to the intermediate dataset. It returns two kinds of schema:

Dictionary schema

- INDEX (Numeric)
- NAME (String)

From Dictionary schema, we have subject, predicate, and object dictionary where each dictionary will have the name and the index number as we can see figure 4.4.

Fact table schema

It's a centralized table that contains only numerical values of subject, predicate, and subject. It allows us to maintains the relationships between the three dictionaries (See Dictionary schema).

- SUBJECT_INDEX (Numeric)
- PREDICATE_INDEX (Numeric)
- OBJECT_INDEX (Numeric)

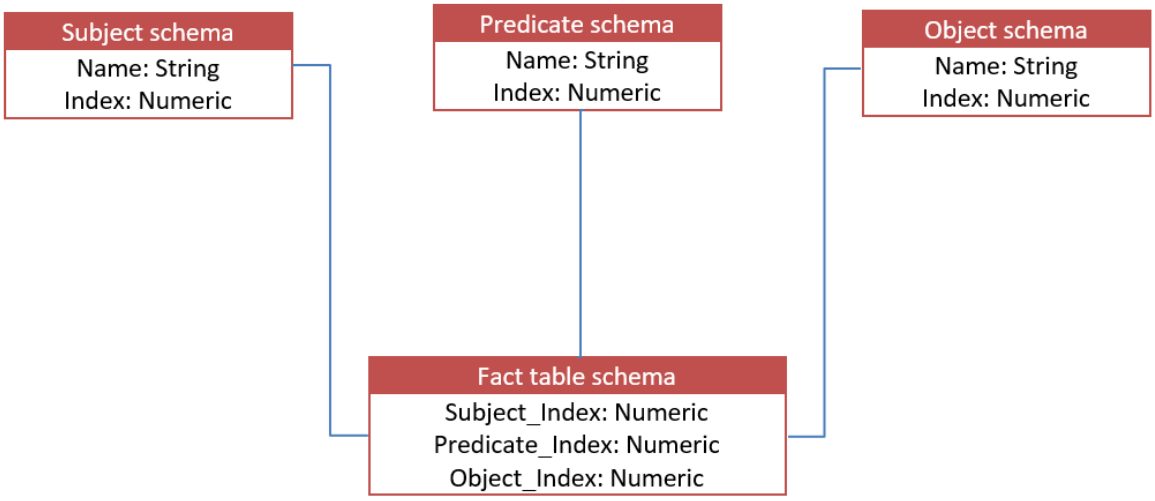

Figure 4.4: Visualization of Record Schema

Chapter 5

Evaluation

We have designed and implemented an RDF compression technique following the approach described in chapter 4. To evaluate the overall performance of our system, we applied a small query on top of the normal dataset and the compressed dataset.
We will cover the following research questions in our evaluation of the normal and the compressed dataset:

1. What is the efficiency of the RDF dictionary compression techniques compare to other compression techniques?
2. How fast is the query processing over compressed data compared to the normal data?

We assess the above research questions, explain them, and then observe some important facts. In order to better understanding RDF Dictionary Compression and to achieve a better solution for dealing with it, we considered the following points.

1. We recorded the compression time and the loading time.
2. We recorded the size of the compressed data.
3. We measured the execution time for each query on the normal and compressed data.
4. We recorded the size of the query results on the normal and compressed data.

5.1 Cluster Configuration

To assess the scalability of our RDF data compression system with Spark and Scala, we deployed our system on the cluster of the Smart Data Analytic [1], which consists of the following configuration:

- `Number of server:` three servers with a total of 256 cores
- `Operating system:` Ubuntu 16.04.5 LTS
- `Scala version:` 2.11.8
- `Spark version:` 2.3.1
- `Executor-memory:` 100 GB

[1] http://sda.tech/

5.2 Methodology

We have evaluated compressed data in term of the optimization of space, query time, and the result size in both compressed and normal data. The compression result and the queries portion were evaluated consecutively. The compression results were verified by comparing the input and the output data size, and queries were evaluated by comparing the query time on the compressed data with the query time on the normal data with the same dataset and the same query file. First, we compressed data, and then we recorded the output size and the compressed time for each dataset. This step was repeated three times, and we recorded the mean values of the three compression sizes and the compression times as well. Second, we proceeded with the queries on the compressed data, wherein we executed each file three times, and then we recorded the mean values of the three times. Finally, we executed queries on the normal data, where we repeated the procedure for the compressed data. It should be noted that the results obtained from the experiments are real-time results, without any extra noise.

5.3 Systems

We here compare our system based on data compression with the normal dataset in term of optimization of space. RDF data compression is a new scalable and parallel distributed system, which is built with Scala and Spark. Our RDF compression system aims to reduce data size by maintaining a single copy for each record and by vertically partitioning the dataset. A partitioned dataset with a small size makes in-memory processing possible for a large volume dataset such as in historical data processing. Our system has two main features. First, it compresses data through the dictionary approach, and second it applies a small query on top of the compressed data for the testing purpose.

5.4 Experimental Setup

5.4.1 Benchmarks

To compare our system, we used two types of dataset from the Lehigh University Benchmark (LUBM) [33] and the Berlin SPARQL Benchmark (BSBM) [34]. To test the scalability of our system, we sized this dataset into three categories: small, medium, and large-size dataset:

1. BSBM [34]. This is a well-known benchmark, which is built for e-commerce, wherein a bunch of products are on offer through different businesses and consumers can review them. This benchmark helps us to generate the following three datasets:
    - 841931 triples [`216.1MB`]
    - 12,005,557 triples [`3.1 GB`]
    - 81,980,472 triples [`21.5 GB`]

2. LUBM [33]: These are among the most famous benchmarks for semantic Web data. The ontology domain in LUBM benchmark defines universities, course-work, publishing as well as different group in a department.

   With the help of the above benchmark, We have generated the following datasets:

- 1,163,215 triples [`204.6 MB`]
- 17,705,050 triples [`3.1 GB`]
- 69,080,764 triples [`12.3 GB`]

In order to evaluate our system, We have performed three types of SPARQL queries in each of the above benchmarks, and we have converted also these queries to evaluate our compressed data.

5.4.2 Benchmark Queries

There are three type of queries that can be run on BSBM and LUBM datasets for the normal and the compressed data.

BSBM Benchmark

1. `First query on the compressed dataset:`

   Listing 5.1 (below) shows a basis query that will return all "ProductFeature40".

```
1  select subject.name as x
2  from triple_fact join predicate on triple_fact.p=predicate.index and predicate
       .name = 'http :// www4.wiwiss.fu-berlin.de/bizer/bsbm/v01/vocabulary/
       productFeature'
3  join object on triple_fact.o=object.index and object.name = 'http :// www4.
       wiwiss.fu-berlin.de/bizer/bsbm/v01/instances/ProductFeature40'
4  join subject on triple_fact.s=subject.index
5    }
```

Listing 5.1: Query on the Compressed Data

2. `Second query on the compressed dataset:`

   Listing 5.2 (below) shows a simple query that will return all "ProductType2".

```
1  select subject.name as x
2  from triple_fact join predicate on triple_fact.p=predicate.index and predicate.
       name like '%subClassOf%'
3  join object on triple_fact.o=object.index and object.name = 'http :// www4.wiwiss.
       fu-berlin.de/bizer/bsbm/v01/instances/ProductType2'
4  join subject on triple_fact.s=subject.index
5    }
```

Listing 5.2: Query on the Compressed Data

3. `Third query on the compressed dataset:`

   Listing 5.3 (below) shows a simple query that will return all “ProductType3”. This query is more complex compared to listing 5.2 and 5.3.

```
select subject.name as x
from triple_fact join predicate on triple_fact.p=predicate.index and predicate.
    name = 'http :// www4.wiwiss.fu-berlin.de/bizer/bsbm/v01/vocabulary/
    productFeature'
join object on triple_fact.o=object.index and object.name = 'http :// www4.wiwiss.
    fu-berlin.de/bizer/bsbm/v01/instances/ProductFeature40'
join subject on triple_fact.s=subject.index
  }
```

Listing 5.3: Query on the Compressed Data

LUBM Benchmark

1. `First query on the compressed dataset:`

   Listing 5.4 (below) shows a simple query that returns publication Author.

```
select subject.name as s, predicate.name as p, object.name as o
  from triple_fact join subject on triple_fact.s=subject.index
    join object on triple_fact.o=object.index
    join predicate on triple_fact.p=predicate.index AND predicate.name = 'http
        :// www.lehigh.edu/~zhp2/2004/0401/univ-bench.owl#publicationAuthor'
```

Listing 5.4: Query on the Compressed Data

2. `Second query on the compressed dataset:`

   Listing 5.5 (below) shows a query that will returns "GraduateStudent" who takes course "GraduateCourse0".

```
select subject.name as s
  from triple_fact join subject on triple_fact.s=subject.index
    join predicate on triple_fact.p=predicate.index
    join object on triple_fact.o=object.index where
    ( predicate.name like '%#type%' and object.name like '%#GraduateStudent%')
        OR
    ( predicate.name like '%#takesCourse' AND object.name = 'http :// www.
        Department0.University0.edu/GraduateCourse0' )
    group by subject.name having count(*)=2
```

Listing 5.5: Query on the Compressed Data

3. `Third query on the compressed dataset:`
   Listing 5.6 (below) shows a query that will return all "UndergraduateStudent".

```
1
2     select   triple_fact .s as s
3       from  triple_fact
4           join  predicate  on  triple_fact .p=predicate .index
5           join  object  on  triple_fact .o=object .index   where
6           ( predicate .name like  '%#type%' and object .name like  '%#UndergraduateStudent
                %')
```

Listing 5.6: Query on the Compressed Data

To evaluate the scalability of our system, we have categorized the dataset into three classes. They are generated from the LUBM and BSBM datasets. The range of data varies from `204.6 MB` to `21.5 GB`.

Small Sized Dataset Descriptions

Dataset sizes with the respective triples, subjects, predicates, and objects as shown in table 5.1

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

Table 5.1: Small Dataset Description

Medium Sized Dataset Descriptions

Dataset sizes with his respective triples, subjects, predicates, and objects as shown in table 5.2

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

Table 5.2: Medium Dataset Description

Large Sized Dataset Description

Dataset sizes with his respective triples, subjects, predicates, and objects as shown in table 5.3

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

Table 5.3: Large Dataset Description

5.5 Experiments & Results

5.5.1 Experiments Results on the Normal and Compressed Data

For the Best Case results on the Compressed Data:

Small Sized Dataset

Dataset	Input Size (MB)	#Triples	Compressed Data (MB)	Compression Ratio	Compression Time (second)	Loading Time (second)
LUBM	204.6	1,163,215	36.7	82 %	11	10
BSBM	216.1	841,931	62.9	70 %	13	12

Table 5.4: Small-Sized Dataset Compression Results

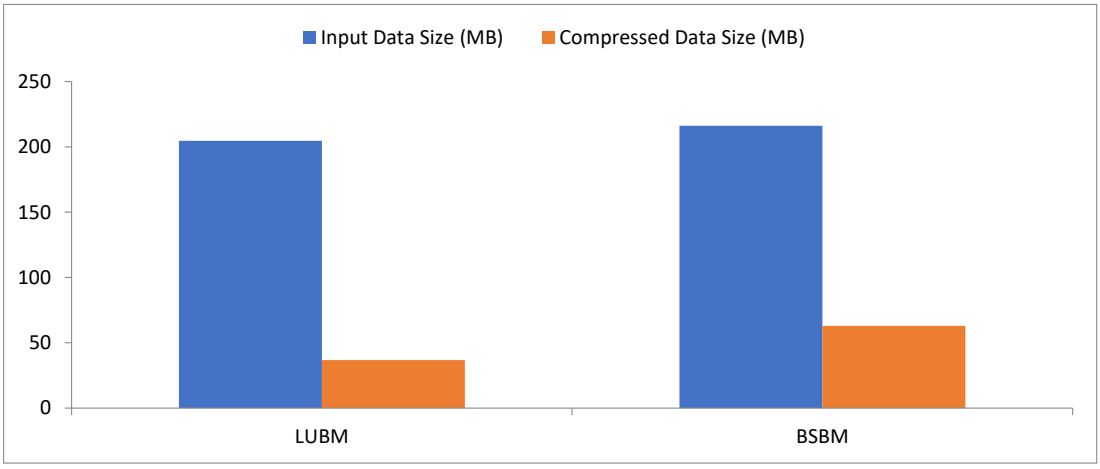

Figure 5.1: Compression Results

From the above chart and table, small-sized dataset consists of triples between 841,931 triples and 1,163,215 triples. The data compression ratio using the RDF Dictionary approach has a ranges from 70% to 82%; The compression time has a ranges from 11 to 13 seconds; The loading time has a range from 10 to 12 seconds.

Medium-Sized Dataset

Dataset	Input Size (MB)	#Triples	Compressed Data (MB)	Compression Ratio	Compression Time (second)	Loading Time (second)
LUBM	3100	17,705,050	616.9	80%	40	39
BSBM	3100	12,005,557	898.9	71%	43	42

Table 5.5: Medium-Sized Dataset Compression Results

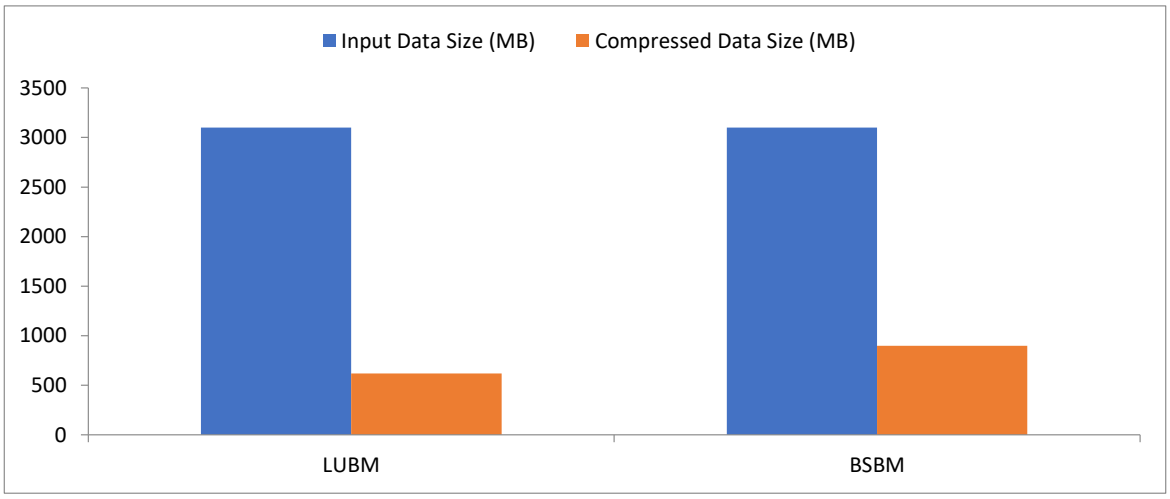

Figure 5.2: Compression Results on the Medium Dataset

From the above chart and table, the medium-sized dataset consists of triples between 12,005,557 triples and 17,705,050 triples. The data compression ratio using the RDF Dictionary approach has a ranges from 71% to 80%; The compression time has a ranges from 40 to 43 seconds; The loading time has a range from 39 to 42 seconds.

Large-Sized Dataset

Dataset	Input Size (GB)	#Triples	Compressed Data (GB)	Compression Ratio	Compression Time (second)	Loading Time (second)
LUBM	12.3	69,080,764	2.4	80%	92	91
BSBM	21.5	81,980,472	5.9	72%	223	222

Table 5.6: Large-Sized Dataset Compression Results

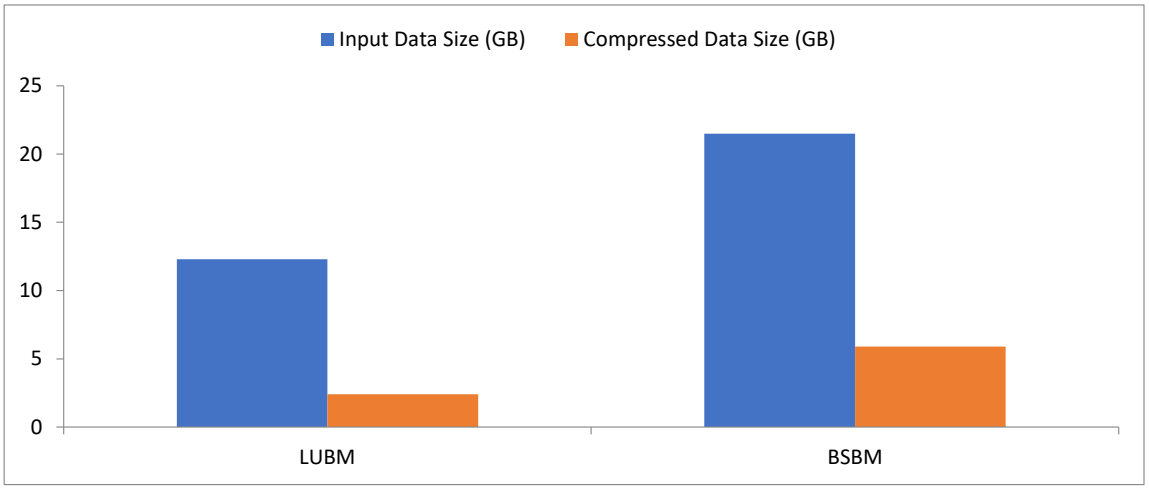

Figure 5.3: Compression Results on the Large Dataset

From the above chart and table, large-sized dataset consists of triples between `69,080,764 triples` and `81,980,472 triples`, the data Compression ratio using RDF Dictionary approach have a ranges from `72%` to `80%`; The compression time has a ranges from `92` to `223 seconds`; The loading time has a range from `91` to `222 seconds`.

Worst Case Results on the BSBM Data

The RDF dictionary algorithm focuses on eliminating duplicates data and persisting relationships in the centralized fact table. In the worst-case scenario, there are few or no duplicates records in the dataset; hence, it expected that the compressed data size is larger than the input data by N bytes, where N is the size of the fact table. To create the worst-case situation, we have generated several datasets that contain unique values only with our data generator class.

Small-Sized Dataset

Dataset	Input Size (MB)	#Triples	Compressed Data (MB)	Compression Ratio	Compression Time (second)	Loading Time (second)
LUBM	33.1	100	35.1	-5%	1	0.5
BSBM	33.1	100	35.1	-5%	1	0.5

Table 5.7: Worst-Case Compression Results on the Small Dataset

The compressed data volume is greater than the input dataset by the fact table size (which is around 2KB).

Medium-Sized Dataset

Dataset	Input Size (GB)	#Triples	Compressed Data (GB)	Compression Ratio	Compression Time (second)	Loading Time (second)
LUBM	1.4	5,000,000	1.6	-12 %	90	89
BSBM	1.7	5,000,000	1.9	-10 %	112	111

Table 5.8: Worst-Case Compression Results on the Small Dataset

The compressed data volume is greater than the input dataset by around 200MB). As data size increased, we observed a noticeable difference between the compressed data volume and the input volume.

Large-Sized Dataset

Dataset	Input Size (GB)	#Triples	Compressed Data (GB)	Compression Ratio	Compression Time (second)	Loading Time (second)
LUBM	14	50,000,000	16.3	-14 %	610	609
BSBM	16.8	50,000,000	19.2	-12%	720	719

Table 5.9: Worst-Case Compression Results on the Large Dataset

The compressed data volume is greater than the input dataset by around 2,300MB on the LUBM and 2,400MB on the BSBM.
Hence, in the worst case scenario, we have observed a 5% to 14% increase in space consumption.

5.6 Query Results on the Normal and Compressed Dataset

We performed three queries on the LUBM and BSBM datasets for the purpose of evaluation. The results are shown below.

On The Small Sized Dataset (LUBM)

In order to verify the correctness and the completeness of our system, we have performed three queries on the LUBM data. Table 5.10 below shows the processing time on the normal and the compressed dataset. We received the same number of result size in both the compressed and normal dataset; this means that we did not lose any data while compressing.

Queries	SparkSQL Time (second)	SparkSQL Result Size	Compression Query Time (second)	Compression Query Result Size	Loading Time(second)
Query-01 (Listing 5.4)	3	121,105	6	121,105	5
Query-02 (Listing 5.5)	3	4	6	4	5
Query-03 (Listing 5.6)	3	66,676	9	66,676	8

Table 5.10: Query Results on the LUBM Dataset

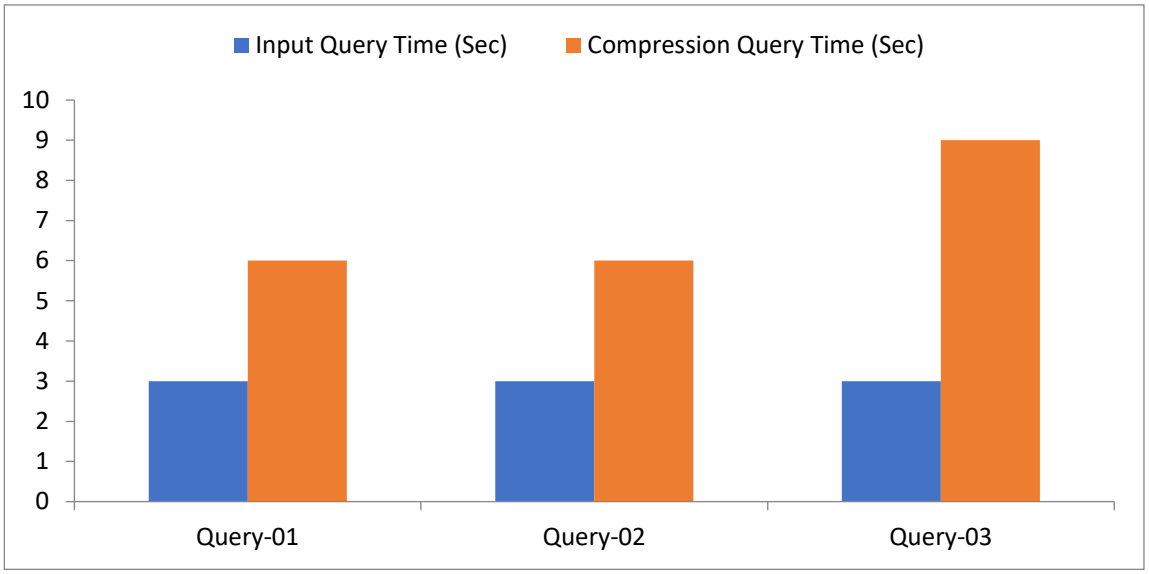

Figure 5.4: LUBM Query Runtime

From this chart, it can be observed that all three queries are faster on the normal data than the compressed dataset.

On The Small Sized Dataset (BSBM)

In order to verify the correctness and completeness of our system, we performed three queries on the BSBM data as well. Table 5.11 below shows the processing time on the normal dataset and the compressed dataset. We obtained the same number count (or result size) in both the compressed and normal dataset; this means that we did not lose any data while compressing.

Queries	SparkSQL Time (second)	SparkSQL Result Size	Compression Query Time (second)	Compression Query Result Size	Loading Time(second)
Query-01 (Listing 5.1)	3	115	7	115	6
Query-02 (Listing 5.2)	3	8	7	8	6
Query-03 (Listing 5.3)	3	47,902	10	47,902	9

Table 5.11: Query Results on the Small BSBM Dataset

From this chart, we can observe that, the query is faster on the compressed data than the normal dataset.

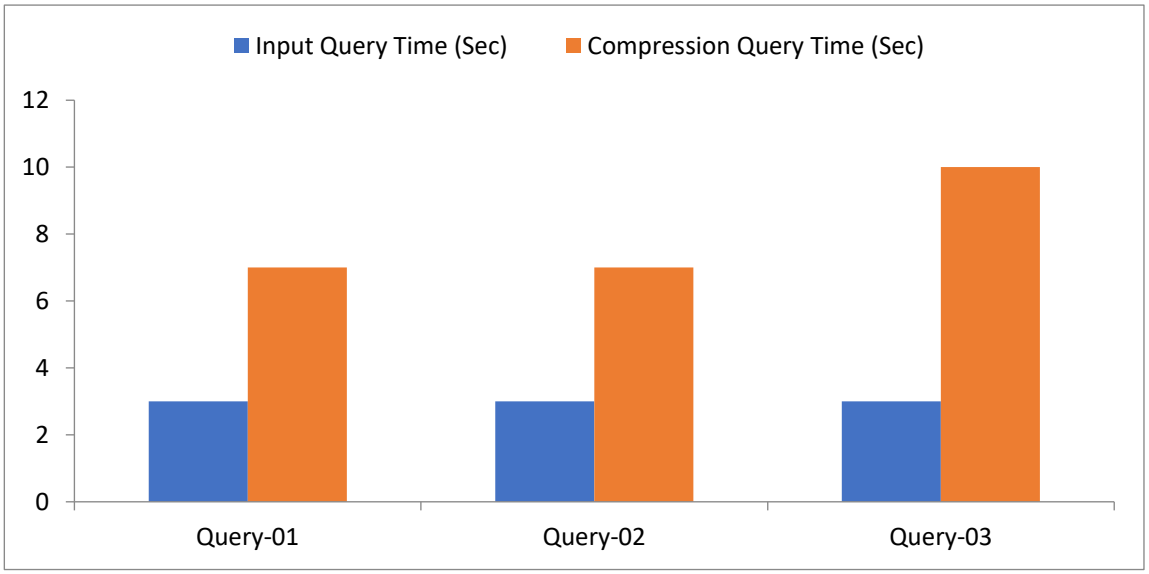

Figure 5.5: BSBM Query Runtime

This chart shows the query time on the BSBM dataset for both compressed and normal datasets. We can see that all three queries are faster on the normal data than the compressed data.

On The Medium Sized Dataset (LUBM)

For the testing purpose, we performed three queries on LUBM data. Table 5.12 below shows the processing times for the normal and the compressed dataset. We have obtained the same number result size for both the compressed and normal dataset, which means that we did not lose any data while compressing.

Queries	SparkSQL Time (second)	SparkSQL Result Size	Compression Query Time (second)	Compression Query Result Size	Loading Time(second)
Query-01 (Listing 5.4)	29	1,839,964	34	1,839,964	33
Query-02 (Listing 5.5)	42	4	39	4	38
Query-03 (Listing 5.6)	55	1,015,351	28	1,015,351	27

Table 5.12: Query Results on the Medium LUBM Dataset

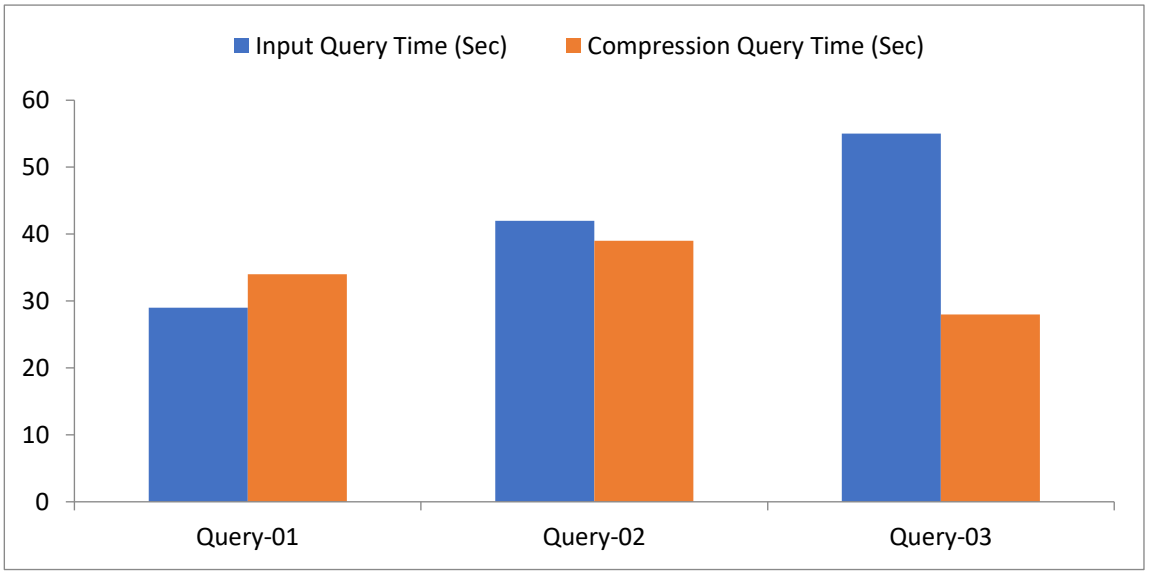

Figure 5.6: LUBM Query Runtime

This chart shows the query time on the LUBM dataset for both the compressed and normal datasets. The compressed data performed better on `query-02` and `query-03`.

On The Medium Sized Dataset (BSBM)

For the testing purpose, we performed three queries on the BSBM data. Table 5.13 below shows the processing time for the normal and the compressed dataset. We obtained the same number result size for both the compressed and normal dataset, which means that we did not lose any data while compressing.

Queries	SparkSQL Time (second)	SparkSQL Result Size	Compression Query Time (second)	Compression Query Result Size	Loading Time(second)
Query-01 (Listing 5.1)	28	12,33	27	1,233	26
Query-02 (Listing 5.2)	19	8	28	8	27
Query-03 (Listing 5.3)	40	725,753	43	725,753	42

Table 5.13: Query Results on the Medium BSBM Dataset

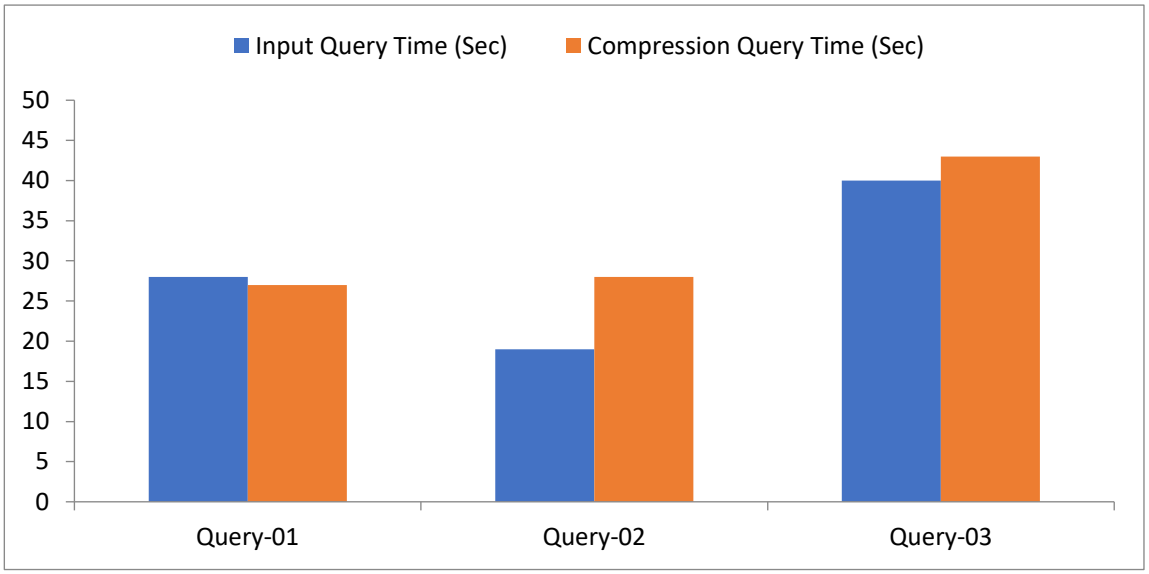

Figure 5.7: BSBM Query Runtime

This chart shows the query time on the BSBM dataset for both the compressed and normal datasets. The compressed data performed better on `query-01`.

On The Large Sized Dataset (LUBM)

In order to verify the correctness and the completeness of our system, we performed three queries on the LUBM data. Table 5.14 shows the processing times for the normal and the compressed dataset. We received the same number result size for both the compressed and normal dataset; this means that we did not lose any data while compressing

Queries	SparkSQL Time (second)	SparkSQL Result Size	Compression Query Time (second)	Compression Query Result Size	Loading Time(second)
Query-01 (Listing 5.4)	92	7,176,871	86	7,176,871	85
Query-02 (Listing 5.5)	62	4	95	4	94
Query-03 (Listing 5.6)	142	3,961,133	62	3,961,133	61

Table 5.14: Query Results on the Large LUBM Dataset

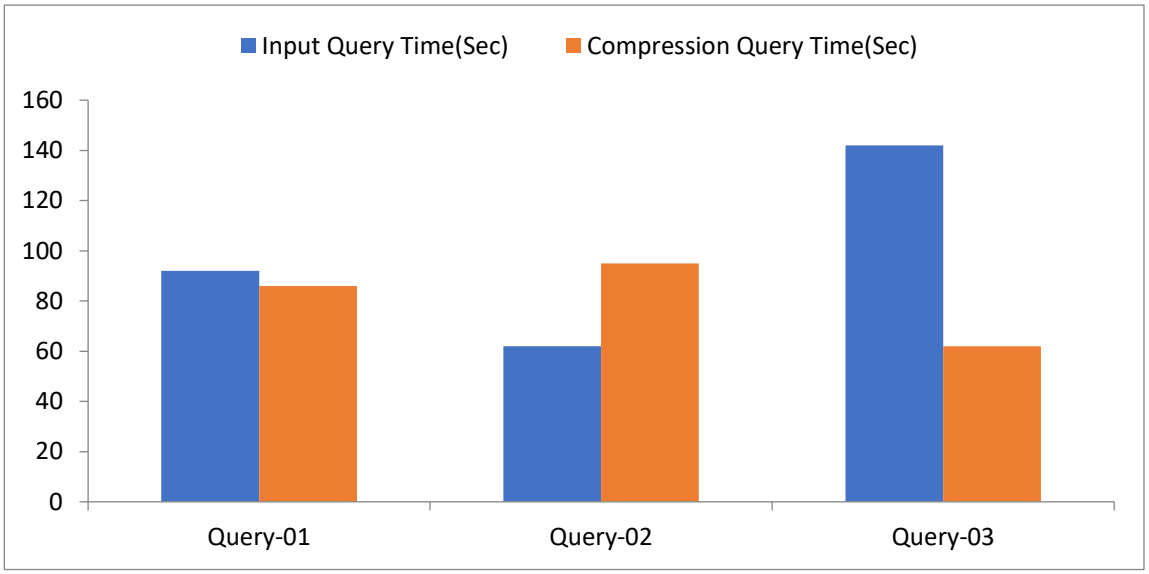

Figure 5.8: LUBM Query Runtime

This chart shows the query time on the LUBM dataset for both the compressed and normal datasets. The compressed data performed better on `query-01` and `query-03`.

On The Large Sized Dataset (BSBM)

In order to verify the correctness and completeness of our system, we performed three queries on the BSBM data as well. Table 5.15 shows the processing time for the normal dataset and the compressed dataset. We obtained the same number count (or result size) for both the compressed and normal dataset; this means that we did not lose any data while compressing.

Queries	SparkSQL Time (second)	SparkSQL Result Size	Compression Query Time (second)	Compression Query Result Size	Loading Time(second)
Query-01 (Listing 5.1)	150	7,728	99	7,728	98
Query-02 (Listing 5.2)	143	8	89	8	88
Query-03 (Listing 5.3)	193	4,534,530	136	4,534,530	135

Table 5.15: Query Results on the Large BSBM Dataset

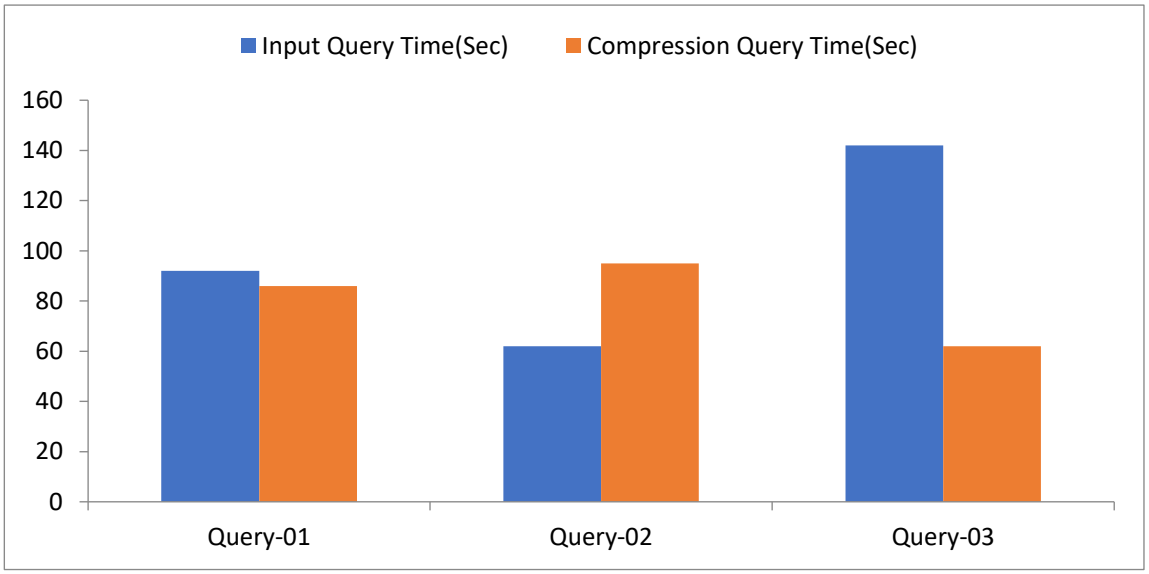

Figure 5.9: BSBM Query Runtime

This chart shows the query time for the BSBM dataset for both the compressed and normal datasets. The compressed data performed better on all three queries.

Figure 5.10, and 5.11 diagrams show the consolidated information of space and execution time for various benchmark dataset.
After compression, the data volume was reduced to 70% to 82%, and it reduced further as the volume of data increased (see table 5.1, 5.2 and 5.3). This reduction in space directly impacts the resource requirement of storage and processing. As the volume of data is reduced, the compressed data can be smoothly processed with a small cluster with limited configuration since the compressed data occupied less disk space, reduces storage cost and also reading and writing the compressed dataset takes less time than the normal dataset. Our RDF Dictionary Compression system gave a noticeable query results on a large volume of compressed data as we can see in a table 5.15. From the evaluation results, we can say that as compressed data volume is especially low, it is feasible to processed record in memory that improves the query performance.

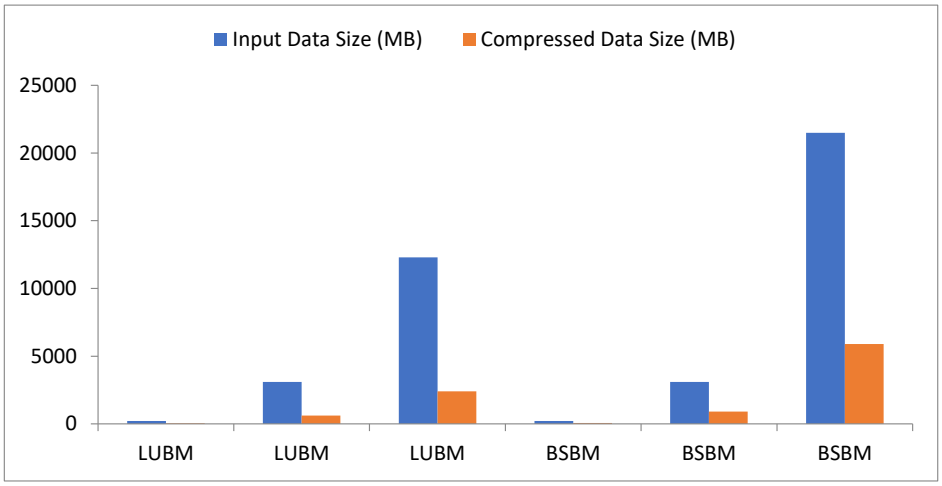

Figure 5.10: Space Consumption in Different Datasets

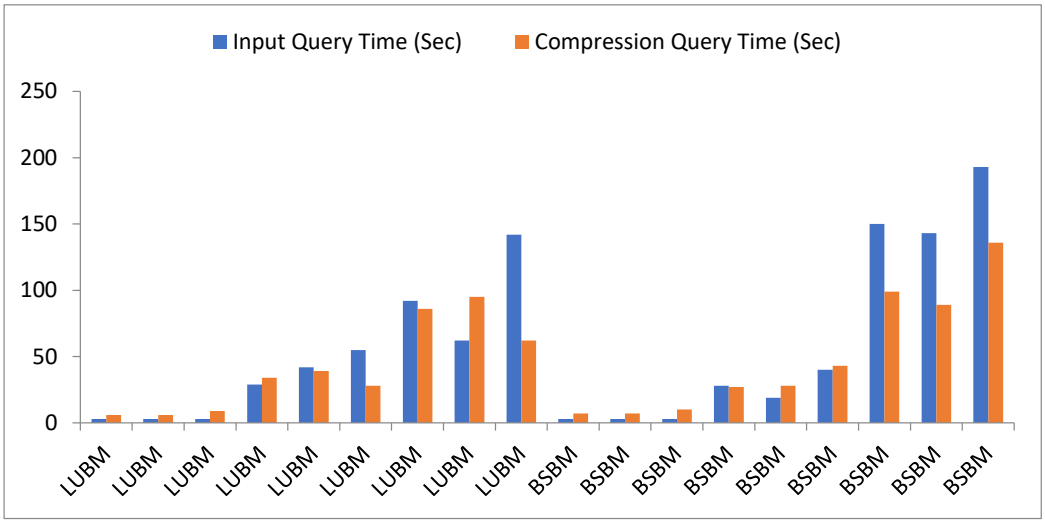

Figure 5.11: Comparison of Query Time in Different Datasets

Chapter 6

Conclusions and Future Work

This thesis was designed and implemented with Spark and Scala with the goal of developing a distributed scalable compression system that compresses a dataset in order to optimize the disk space and that allows for a query on top of the compressed data directly.
From the results of this research, it is clear that compression technique provides an excellent compression ratio from, 70% to 82%, and maintains the relationship between records; this makes it possible to recreate input records from the compressed data. In contrast, for a dataset with no duplicate record, the compressed data volume is higher than the input data (from 4% to 14%). Apart from data compression, the query system was designed for the testing purpose to query on compressed data directly and also to execute a query with a most optimized plan. In the LUBM and BSBM benchmark, the RDF data compression performed faster on the medium- and large-sized datasets than on the normal dataset; however, the small-sized dataset is two to three times faster on the normal dataset. Regarding overall performance based on the query results, we can conclude that the query system performs better on compressed data. Empirically has been showed a distributed system, especially Apache Spark, performs best when complete data is in the memory [35].
For future work, the following suggestion needs to be considered:

- To compare this system with other compression systems.
- To apply a compression algorithm such as gzip, snappy, or lzo on the stored result in order to further optimize the space requirement.
- To conduct evaluations with large datasets, on exceptionally large clusters.
- To implement a query system on the compressed data.

Bibliography

[1] J Lehmann, G Sejdiu, L Bühmann, P Westphal, C Stadler, ... Ermilov, I., and H. Jabeen. Distributed semantic analytics using the sansa stack. In *International Semantic Web Conference (pp. 147-155). Springer, Cham*, 2017, October.

[2] F Manola and E Miller. Rdf primer:w3c recommendation,http://www.w3.org/tr/rdf-primer/, 2004. Last accessed 3 October 2018.

[3] C Bizer, T Heath, and T Berners-Lee. Linked data-the story so far:int j semant web inf syst 5:1–22,2009.

[4] E Prud'hommeaux and A Seaborne. Sparql query language for rdf: W3c recommendation,http://www.w3.org/tr/rdf-sparql-query/, 2008. Last accessed 16 October 2018.

[5] J Huang, D Abadi, and K Ren. Scalable sparql querying of large rdf graphs.proc vldb endow 4(11):1123–1134,2011.

[6] Y Jing, D Jeong, and D Baik. Sparql graph pattern rewriting for owl-dl inference queries.knowl infsyst 20:243–262,2009.

[7] A Martínez-Prieto, M, D Fernández, J, and s Cánovas, R. Compression of rdf dictionarie. In *Proceedings of the 27th Annual ACM Symposium on Applied Computing (pp. 340-347). ACM*, 2012, March.

[8] Fernández J. D. & Cánovas R. Martínez-Prieto, M. A. Querying rdf dictionaries in compressed space. acm sigapp applied computing review, 12(2), 64-77,2012.

[9] Bigdata projects: https://elysiumpro.in/big-data-analytics-projects/. Last accessed 5 October 2018.

[10] M Zaharia, M Chowdhury, T Das, A Dave, J Ma, M McCauley, and I. Stoica. Resilient distributed datasets : A fault-tolerant abstraction for in-memory cluster computing. In *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation (pp. 2-2). USENIX Association*, 2012, April.

[11] Tim Berners-Lee, James Hendler, Ora Lassila, et al. The semantic web vol.284, no.5, pp.28-37,2001.

[12] Semantic web technologie: https://www.w3.org/2007/03/layercake.svg. Last accessed 6 October 2018.

[13] D. C.. Hazen. Rethinking research library collections: A policy framework for straitened times, and beyond,2010.

[14] Handbook of semantic web technologies: https://link.springer.com/content/pdf. Last accessed 05 November 2018.

[15] Sparql query language for rdf.https://www.w3.org/tr/rdf-sparql-query/. Last accessed 5 October 2018.

[16] 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.*, 2017, May.

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

[18] Gutierrez C. & Martínez-Prieto M. A. Fernández, J. D. Rdf compression: basic approaches. In *Proceedings of the 19th international conference on World wide web (pp. 1091-1092). ACM,*, 2010, April.

[19] Big data overview :https://intellipaat.com/tutorial/hadoop-tutorial/big-data-overview/. Last accessed 30 November 2018.

[20] Pazhaniraja N. Paul P. V.-Basha M. S. & Dhavachelvan P.. Saraladevi, B. Big data and hadoop-a study in security perspective. procedia computer science, 50, 596-601,2015.

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

[22] Venkat Ankam. Big data analytics.packt publishing ltd,2016.

[23] 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.

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

[25] 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.

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

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

[28] JASSEM yahyaoui. Spark vs mapreduce dans hadoop: https://yahyaouijassem.wordpress.com/2016/04/23/spark-vs-mapreduce-dans-hadoop. Last accessed 17 October 2018.

[29] Kyprianos K. & Stefanidakis M. Papadakis, I. Linked data uris and libraries: the story so far. d-lib magazine, 21(5/6),2015.

[30] Sandra Álvarez García and et al. Compressed vertical partitioning for efficient rdf management. knowledge and information systems 44.2: 439-474,2015.

[31] P Ravi and A. Ashokkumar. A study of various data compression techniques.ijcs journal 6.2,2015.

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

[33] Lehigh university benchmark to facilitate the evaluation of semantic web repositories in a standard and systematic way: http://swat.cse.lehigh.edu/projects/lubm/. Last accessed 15 November 2018.

[34] Berlin sparql benchmark: http://wifo5-03.informatik.uni-mannheim.de/bizer/berlinsparqlbenchmark. Last accessed 15 November 2018.

[35] A. G. Shoro and T. R. Soomro. Big data analysis: Apache spark perspective. global journal of computer science and technology,2015.