Scalable RDF Clustering

Pratik Kumar Agarwal

Matriculation number: 3056132

November 2, 2019

Master Thesis

Computer Science

Supervisors:

Prof. Dr. Jens Lehmann
Dr. Hajira Jabeen

Institut für Informatik III

Rheinische Friedrich-Wilhelms-Universität Bonn

Declaration of Authorship

I, Pratik Kumar Agarwal, declare that this thesis, titled "Scalable RDF Clustering", 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

I am highly obliged and grateful and would like to thank my supervisor Dr. Hajira Jabeen for her constant guidance and support for the successful completion of my thesis work. I would like to thank Prof. Dr. Jens Lehmann for providing me the opportunity to do my thesis at Smart Data Analytics (SDA) group at the University of Bonn. And I would like to than Mr. Gezim Sejdiu for his guidance in using the spark cluster and regarding the queries and optimisation of code and for guiding me in preparing the raw dataset. Last but not the least I would like to thank all my colleagues who supported me in this work.

Contents

1 Introduction **1**
- 1.1 Motivation . . . 1
- 1.2 Objective . . . 3
- 1.3 Contributions . . . 3
- 1.4 Thesis Structure . . . 4

2 Preliminaries **5**
- 2.1 Semantic Web . . . 5
- 2.2 Big Data . . . 8
  - 2.2.1 Importance of Big Data: . . . 10
- 2.3 Apache Spark . . . 10
  - 2.3.1 Spark Resilient Distributed Datasets (RDDs) . . . 13
  - 2.3.2 Spark DataFrames . . . 14
  - 2.3.3 Hadoop Distributed File System (HDFS) . . . 15
- 2.4 Clustering . . . 16
- 2.5 Why Clustering ? . . . 17

3 Related Work **18**
- 3.1 K-Means Clustering . . . 18
- 3.2 K-Medoid Clustering . . . 19
- 3.3 Hierarchical Clustering . . . 21
- 3.4 Power iteration clustering (PIC) . . . 24
- 3.5 Spectral Clustering . . . 25

4 Implementation **26**
- 4.1 Architecture . . . 26
  - 4.1.1 Locality Sensitive Hashing . . . 27
  - 4.1.2 MinHash for Jaccard Distance . . . 28
  - 4.1.3 Approximate Similarity Join . . . 29
  - 4.1.4 Need for GraphFrames . . . 30
  - 4.1.5 Why GraphFrames ? Why not GraphX ? . . . 31

4.2 Silhouette 33
4.3 Challenges and Improvement 35

5 Evaluation **36**
5.1 Experimental Setup 36
5.1.1 Datasets 37
5.2 Preliminary Results 37
5.2.1 Analysis of BSBM + LUBM dataset 39
5.2.2 Analysis of DBpedia dataset 42
5.2.3 Analysis of BIRT dataset 45

6 Conclusion **48**

List of Figures

1.1 LOD Cloud[1] 2

2.1 Semantic Web Stack[2] 6
2.2 A RDF Triple 7
2.3 An example of RDFS 7
2.4 Velocity of data 8
2.5 5Vs of Big data[5] 9
2.6 Apache Spark Ecosystem[7] 11
2.7 Spark Cluster Overview[8] 12
2.8 HDFS Architecture[11] 15

3.1 Dendrogram for Hierarchical clustering. 21

4.1 Approach 26
4.2 Algorithm Steps 29
4.3 Graph with 3 connected components[2] 31
4.4 Using GraphFrames to find clusters. 32
4.5 Using GraphX to find clusters. 32
4.6 To add Graphframe package. 33
4.7 Calculating Silhouette score when used GraphFrames approach. 34
4.8 Calculating Silhouette score when used GraphX approach. . . 34
4.9 Architecture Pipeline 35

5.1 Execution time 38
5.2 Result of BSBM+LUBM Dataset 39
5.3 Result of Dbpedia Dataset 43
5.4 Result of BIRT Dataset 46

6.1 SANSA Stack[1] 48

List of Tables

2.1 RDD Transformations 14
2.2 RDD Actions 14

4.1 Input to our model 27
4.2 Output from pre-processing step 27
4.3 Clusters formed from minHash LSH. 30
4.4 Output from pre-processing step 35

5.1 Evaluation Results for different datasets. 37
5.2 HashingTF VS CountVectorizer 38
5.3 BSBM + LUBM dataset analysis between different classes. . . 40
5.4 BSBM + LUBM dataset analysis between Product Feature and Product Type. 41
5.5 BSBM + LUBM dataset analysis between Producer and Vendor. 42
5.6 Analysis for Sports players. 44
5.7 Analysis for Baseball players. 45
5.8 Analysis of different classes of BIRT dataset. 46
5.9 Analysis of different classes of BIRT dataset. 47

Abstract

Web is becoming rich in data. Some of the source from where these data are originating is via Blogs, Youtube, Twitter, Emails, E-Commerce, Banking, sensors and Internet of Things. But these data are structured in a very poorly fashion. The content of the web is becoming heterogeneous in nature both in terms of content and structure. We can say that these data are human readable data but our motive is to draw inferences from these data which is only possible if we make them machine accessible. Clustering is considered an important task to organise these data and draw meaning inferences from these data. In this work we are presenting a clustering approach which can be applied on Knowledge graphs. We have explored the possibility of applying Locality Sensitive Hashing on knowledge graphs. Given the size of linked data we show that this approach can be effective and scalable in comparison to other clustering approaches like Hierarchical clustering, K-Means clustering, K-Medoid clustering in discovering different communities that are defined by the link structure of the graph. The experimental results on different types of Linked Data sources shows that the proposed method is scalable and efficient.

Keywords: Linked Data, Semantic Web, Big Data, Clustering, Locality Sensitive Hashing, GraphFrames, Silhouette

Chapter 1

Introduction

1.1 Motivation

Traditionally, during the time of the hypertext web, hyperlinks allowed users to traverse from one documents to another to gather sufficient information. Based on the user query search engines used to analyze the structure of links between the documents and give results to the users. But unfortunately, the same principle has not been applied to data. All the documents are in HTML data format but the problem with this format is that it doesn't provide flexibility to enable individual entities that are present in that document to get connected with similar kinds of entities via links. But in recent years there has been a huge evolution in the Web. The World Wide Web now has lowered the restriction of data publication which has resulted in a tremendous growth in the amount of data that are now available publicly on the web. The velocity of data generation is so huge that if we look at the data generated through internet we will find that in 60 seconds approximately 3.8 million google search queries are fired, 4.5 Million YouTube videos are being viewed, 2.1 Million snaps are being created and approximately 41.6 Million messages are being sent via Messenger and WhatsApp etc. Web has now matured to a space where both documents and data are now linked and we call it as *Linked Data (LD)*. These LD are represented in graphical format and follows a set of simple principles which we call as Linked Data Principle. Data from heterogeneous domains like Books, People, Companies, Movies, Music, Scientific research, Drugs etc. are now connected on a global data space which ultimately forms a Linked Open Data (henceforth LOD[1]) cloud. LOD cloud as per March 2019 is shown in Figure 1.1.

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

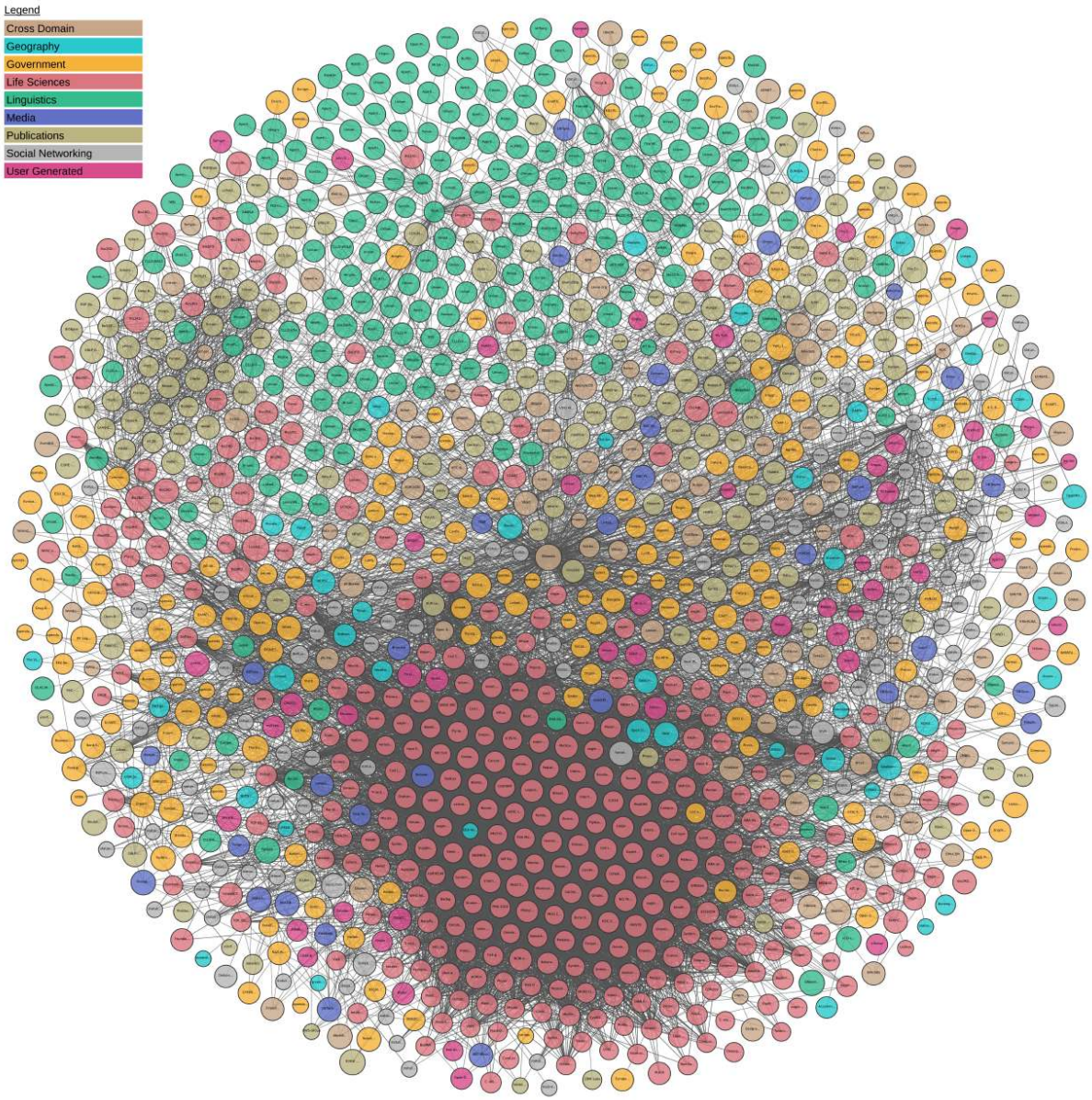

Figure 1.1: LOD Cloud[1]

The English version of DBpedia[2] categorize 4.58 million things. Out of the description of 38.3 million things from all the localized versions of DBpedia, 23.8 million descriptions exist in its English version. These data go along with semantic web standards like Resource Description Framework (henceforth RDF) and Resource Description Framework Schema (henceforth

[2]`https://wiki.dbpedia.org/about`

RDFS) and are represented in the form of triples. As the size of LOD cloud is growing at an alarming speed and so does the amount of triples and so does the need to consume this data. If the data is not processed immediately it may be lost forever. Hence we need an approach through which we can efficiently organise these data and draw meaning inferences.

Heterogeneity in the data set resulted in the collection of instances from various domains. We can see this from Figure 1.1; Instances from domains like Media, Government, Geography, Life Science, Publications, etc. are all interconnected among themselves hence separating them into their respective groups or cluster them became an important task in the area of Semantic Web so that we can focus only on the specific cluster based on our interest. And Clustering will also enable the possibilities of finding hidden patterns from the data because we are unaware of its structure.

1.2 Objective

Data available to us is huge and the fact that there are no constraints imposed by the RDF model on the structure of the data makes it difficult to understand the graph patterns and hence, forming a meaningful query over such source is challenging. Therefore it is extremely important for us to find a way to understand the structure of the data and group them.

In this work we are presenting a clustering[3] approach called Locality Sensitive Hashing which can be applied to LD. By looking at the instance level data from RDF triples this approach will group these data into clusters based on their predicate similarity. Many existing researches in Clustering have high time and space complexity and considering the size of data our goal is to develop an efficient and scalable clustering approach. As our local machines and storage space are not capable enough to handle big data so we need a distributed environment where we can process the data in parallel in order to achieve efficiency and scalability.

1.3 Contributions

1. Deciding the value K for clusters is a very common problem for almost all the clustering algorithms. We have eliminated the need to define K.

2. For graph computation we have explored the possibility of applying

[3] `https://spark.apache.org/docs/latest/mllib-clustering.html`

GraphFrames instead of GraphX and successfully integrated it in this thesis work which ultimately gave us a better result without increasing the complexity of the model.

3. We have developed a generic approach using MinHash Locality Sensitive Hashing technique which can be applied on any Knowledge Graphs which has instance level information for entities.

4. Our approach increased the scalability of Clustering task.

5. Our approach is completely build on a distributed framework i.e Apache Spark using Scala programming language.

1.4 Thesis Structure

There are a total of 6 chapters in this work. The remainder of this document is structured as follows. Chapter 2 will discuss various key terminologies and basic concepts that are required for a complete understanding of this work. Chapter 3 will talk about various researches which are already been done in this field. Chapter 4 will give a detailed description of the approach which we have used to perform Clustering. Chapter 5 will talk about the validity and scalability of our work by performing an extensive evaluation of various Linked Data Sources and by analyzing the results. And finally, Chapter 6 will conclude this work.

Chapter 2

Preliminaries

In this chapter we will talk about various topics which are the base of this work and are important to understand for the complete understanding of this thesis work. This chapter will give a brief description about Semantic Web, Big Data, Apache Spark and about Clustering.

2.1 Semantic Web

Semantic Web is a web of data. It allows people to store data on the web through the internet, e-commerce, Internet Of Things or by any other means without any restrictions. Generally, semantics refers to the study of "meaning". So the main objective of the semantic web is to scrutinize the meaning of the data available on the web. Earlier the data available on Web was only human-readable and web-only took care of the syntax of the data. Semantic Web helped in combining semantics to these data and since the web data is heterogeneous in nature it helped in retrieving intelligent inferences from these data. The nucleus of the Semantic Web is the Linked Data. According to the World Wide Web Consortium (W3C)[1], "The Semantic Web provides a common framework that allows data to be shared and reused across application, enterprise, and community boundaries". The semantic web extends the World Wide Web through standards given by W3C[1]. The term Semantic Web was devised by Tim Berners-Lee for a web of data that machines can manage.

Semantic Web Stack[2] shown in Figure 2.1 describes the architecture of Semantic Web. The middle layer of the Semantic Web Stack contains standardized Semantic Web technologies. Each layer of the stack helps in making

[1] https://en.wikipedia.org/wiki/Semantic_Web

[2] https://en.wikipedia.org/wiki/Semantic_Web_Stack

the web information machine-understandable.

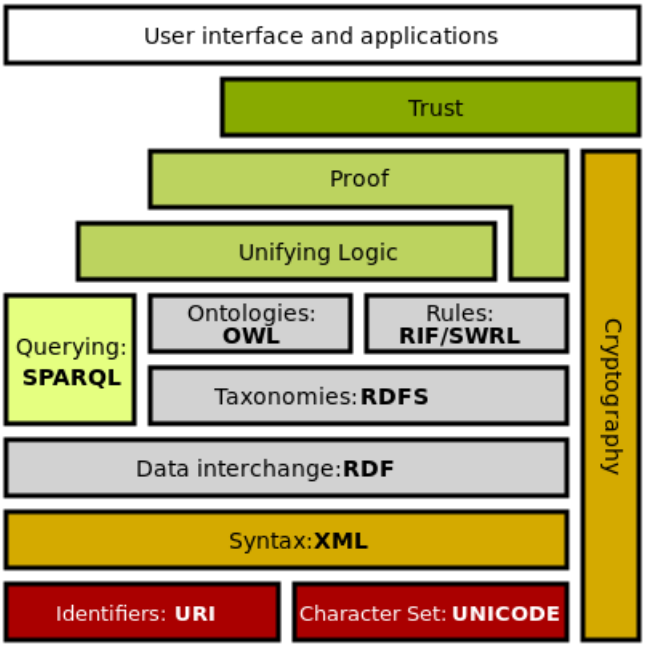

Figure 2.1: Semantic Web Stack[2]

RDF : The Semantic Web stack is built on RDF. RDF provides the foundation for Linked Data and is responsible for representing data in the form of *triples.* A RDF *triple* is of the form:

$$(subject,\ predicate,\ object) \in (R \cup B) \times R \times (R \cup B \cup L) \qquad (2.1)$$

where, R is the resources which is represented in the form of Uniform Resource Identifiers (URIs) or Internationalized Resource Identifier (IRIs), B represents Blank Nodes and L represents Literals.

- **URIs:** It gives a unique name to resources and helps them identified globally.
- **Blank Node[3]:** It represents a resource in RDF graph which does not have a URI or literal. Such resources are also called anonymous resources.
- **Literals:** It is generally represented in the form of a string. Resources that do not have a valid identification is represented in the form of literals. *For Example:* Name of a person, $"1"^{\wedge\wedge}xs : integer$.

[3] `https://en.wikipedia.org/wiki/Blank_node`

Figure 2.2 shows an example of a triple.

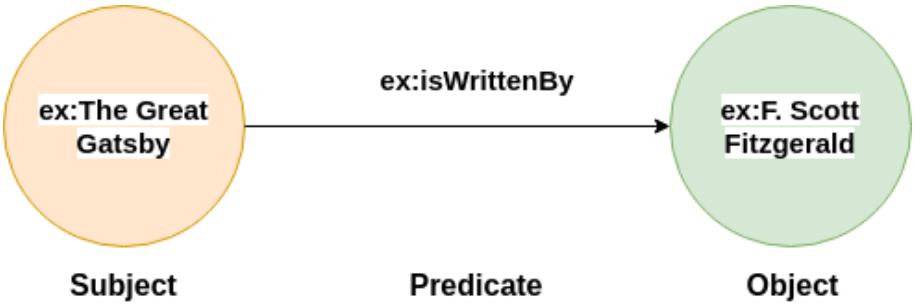

Figure 2.2: A RDF Triple

The set of RDF triples forms an RDF graph. Hence it provides the flexibility to represent information in the form of a graph.

RDFS : RDFS is responsible for providing basic vocabulary for RDF. It might be possible that some triples are useless and do not make any sense like Cinema — AlbertEinstein —> 2012. RDFS allows to define classes, properties and restrict their use. With RDFS it's possible to create hierarchies of classes and properties. Figure 2.3 defines a class for the Subject.

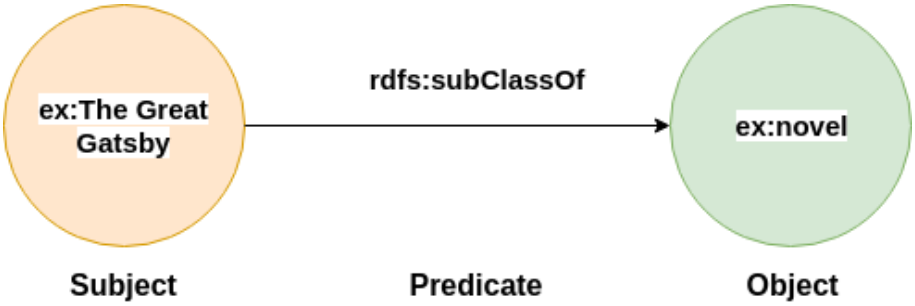

Figure 2.3: An example of RDFS

OWL: It stands for Web Ontology language. It extends RDFS. More advanced constraints like cardinality, restrictions in applying specific values, constraints in relation like transitivity and range, constraints in data types etc. are added by OWL for RDF statements. It brings reasoning power to the semantic web.

SPARQL: It's the query language for RDF. It fetches information for semantic web applications. Queries can also include RDFS and OWL state-

ments to make them robust and meaningful.

RIF: Stands for Rule Interchange Format. We can think of a rule as IF-THEN statement. IF somethings holds THEN the conclusion. Rules can be used to infer somethings from the data which is not directly given.

2.2 Big Data

Big data refers to a substantial amount of data. The data are so huge and complex that we cannot process them on our local machines. The data can be structured or unstructured. Often Big Data is expressed as three Vs i.e Velocity, Volume and Variety.

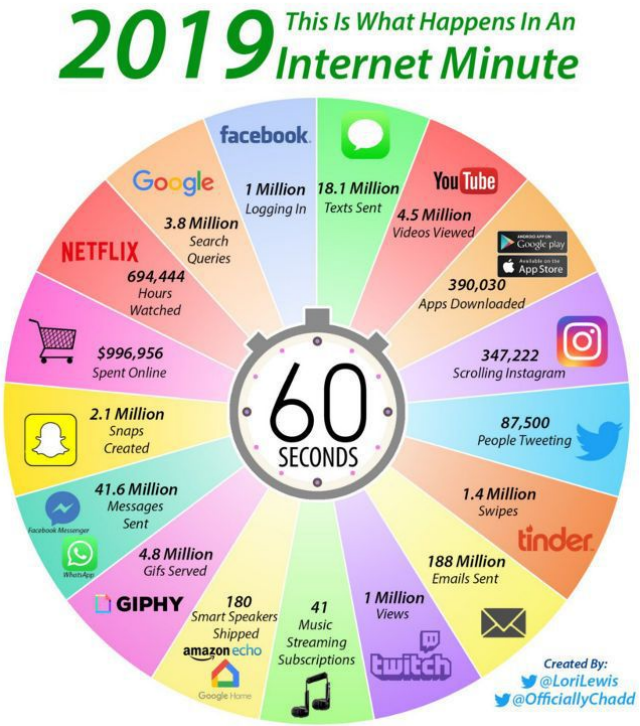

Figure 2.4: Velocity of data

Velocity: Refers to the rapidity at which data is received. For Example: Data generated from sensors, RFID tags needs to be operated in near real time otherwise information might be lost forever. Figure 2.4 shows what happens in an internet minute and how much data is being generated by us.

Volume: Refers to the size of data. It might be hundreds of terabytes or even more. Data can be crowded from numerous sources like mobile phones, sensors, web-pages, automobiles, social media, business transactions, or machine to machine interaction from Internet Of Things, etc.

Variety: Refers to the different data formats that are available today. It's no more the case where you can easily store the structured data into a database. Now data can be audio, video, email, text which are unstructured and semi-structured in nature and hence require additional pre-processing in order to extract meaningful information from them.

Value and Veracity have also emerged as the two more V's of Big Data[4].
Value[4]: Finding value from so much of the data is extremely essential. We need to be insightful to recognize the hidden patterns from the data and predict the behavior. Asking the right questions from the data is equally important to get meaningful insight.

Veracity: Refers to the noise and abnormality in the data. Noisy data needs extra effort like cleaning in order to develop insights. Noise can be generated due to experimental error, natural error, sampling error, data entry loss, data processing error etc. Figure 2.5[5] shows 5Vs of Big Data.

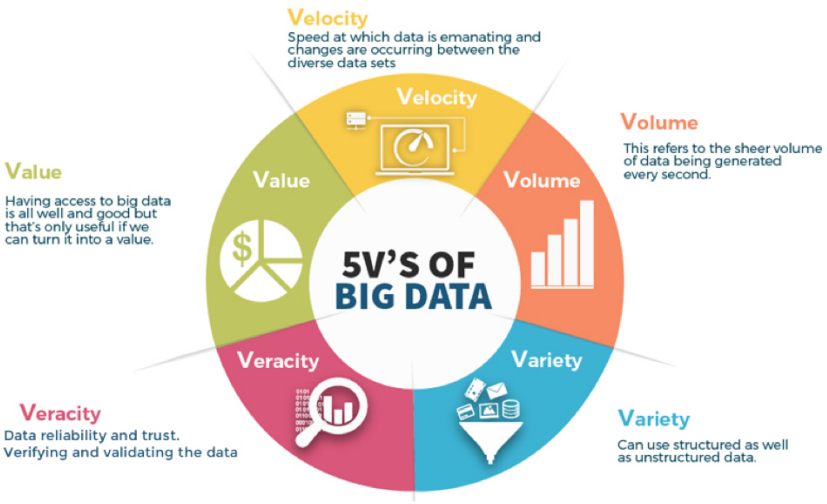

Figure 2.5: 5Vs of Big data[5]

[4]https://www.oracle.com/big-data/guide/what-is-big-data.html
[5]https://www.techentice.com/the-data-veracity-big-data/

But why is Big Data Important? In the following section we will see some of the importance of Big Data.

2.2.1 Importance of Big Data:

- Freedom of taking data from any source for its analysis.
- Big Data gives us more and detailed information through which we can know in a better way about the behavior of the target and can be more confident in answering the questions. Which ultimately helps in smart decision making.
- New products can be developed based on the insights.
- Provides us with real time data which can be helpful for various industries like logistics and health care.
- Different optimizations can be offered in order to improve the quality.
- Root cause of any failure can be identified. Like in case of sensor data if we have data of every second we can find the exact reason for failure.
- Companies can provide better customer service based on their buying habits.

But to process Big data is a challenging task because of the limited storage space and processing speed, hence the need for distributed computing. Various open-source distributed frameworks like Hadoop, Apache Spark etc.are available which can be used to leverage the power of parallel processing and to process the data in a scalable manner. Spark is known for it's performance and proficiency in processing Big Data. In this work we have used Apache Spark framework for processing the data with Hadoop Distributed File System (HDFS) as a storage unit. In the following sections, we will talk about Apache Spark and HDFS.

2.3 Apache Spark

"Apache Spark is a lightning-fast unified analytics engine for big data and machine learning"[6]. Figure 2.6[7] shows the Apache Spark Ecosystem.

[6] https://databricks.com/spark/about

[7] https://www.learningjournal.guru/article/apache-spark/apache-spark-introduction/

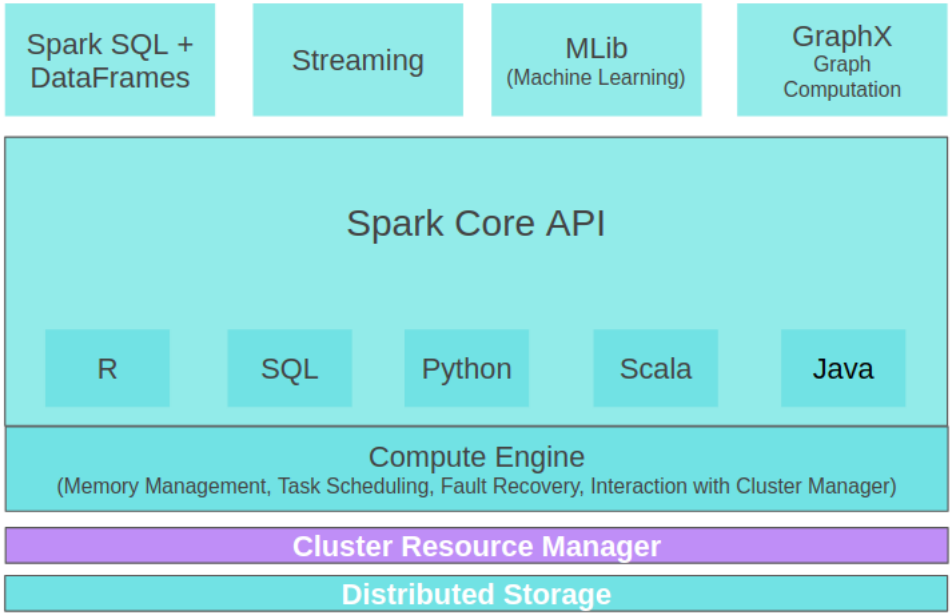

Figure 2.6: Apache Spark Ecosystem[7]

Spark ecosystem is divided into three broader categories.

- Cluster Manager and Distributed Storage
- Spark Core
- Spark Libraries

Cluster Manager and Distributed Storage : Spark does not has it's integral cluster resource manager and a distributed storage. In Spark parallel operation runs on various computer node. Cluster Managers like Apache YARN, Mesos, Kubernetes or it's own standalone cluster manager can be used for Spark. And storage systems which can be used are HDFS, Cassandra File system, Google Cloud storage etc. Figure 2.7, taken from spark's official documentation[8], gives an overview of the spark cluster architecture.

[8]`https://spark.apache.org/docs/latest/cluster-overview.html`

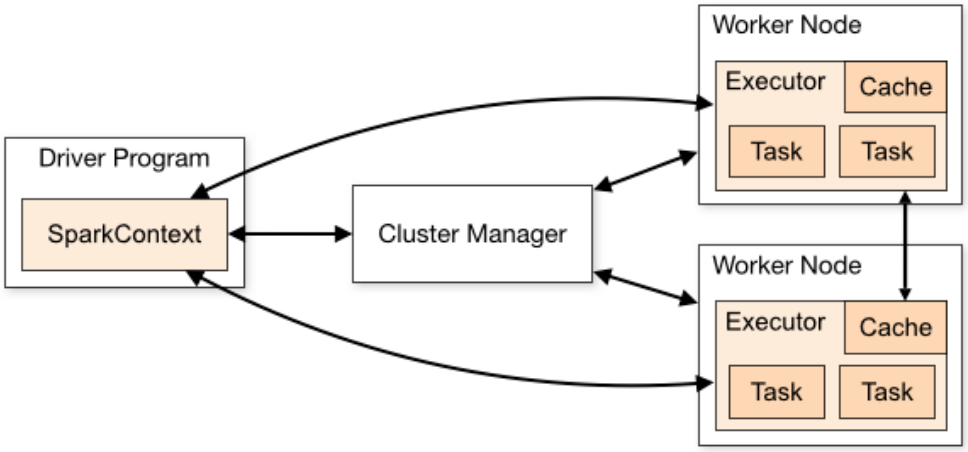

Figure 2.7: Spark Cluster Overview[8]

An object of SparkContext which is present into the main program or driver program initiates the running of spark application as unrestricted set of processes on the cluster. There are various cluster managers like Spark's own standalone cluster manager, YARN or Mesos to which SparkContext can connect which provides resources for the application. There are different Worker nodes in the cluster and spark procure executors on nodes. SparkContext will send the tasks to the executors to run. For each application there is their own executor which means we are unable to share the data across various spark application unless we use an external storage.

Spark Core : At its core Spark has 2 main modules: "Compute Engine" and "Core APIs". As visible from the figure, Compute Engine provides functionalities like memory management, task scheduling, fault recovery and helps in communicating with the lower level. Hence, it's the compute engine which is responsible for managing all the job which makes it the most important part of the ecosystem.

Core APIs like Resilient Distributed Datasets (RDDs) and Dataframes are available for different programming languages like Scala, Java, Python, and R. We will talk about these core APIs in more detail in the upcoming section.

Spark Libraries : Spark Libraries depends upon Core APIs to attain distributed processing. Different Spark libraries are:

- **Spark SQL**

  To process structured data. The data is stored in DataFrames and

basic SQL operations can be performed on the data.

- **Streaming**

  Some applications requires the stream of data to be processed in real-time. Spark Streaming is responsible for the processing of real-time data. Uses Spark core's fast scheduling capability to accomplish stream analytics.

- **MLib**

  It is a machine learning library which helps in making predictions and statistical analysis from the data through various machine learning algorithms like clustering, classification and regression.

- **GraphX**

  GraphX is a graph processing framework which processes graphs at scale.

2.3.1 Spark Resilient Distributed Datasets (RDDs)

RDD[9] is one of the core APIs and a fundamental data structure of Spark. It is a collection of objects which are distributed across worker nodes and handled in parallel. Two ways in which RDD can be created are; Firstly, by distributing a collection of object in the driver program and secondly by loading a dataset from external storage like HDFS, HBase etc. There are two kinds of operations which are supported by RDDs.

- **Transformation** : It contains certain functions which creates a new RDD from an existing RDD. Example: map().
  Table 2.1 shows different transformations which can be applied.

- **Action** : It performs a computation on the RDD and returns the value to the main program. Example: reduce()
  Table 2.2 shows different actions which can be performed.

One of the reasons why Spark runs efficiently is that all the transformations are lazy. They do not process immediately when applied, instead, they keep on stacking the operations applied on the dataset and starts computing when an action is called. For all the transformations Spark maintains a lineage graph and executes it when an action is called. The reason for maintaining a lineage graph is if some partition is lost it can easily be recovered.

[9] `https://spark.apache.org/docs/latest/rdd-programming-guide.html`

Transformations	Description
map	Returns a new RDD which is formed by applying a function to all the elements of a given RDD.
filter	Returns a new RDD containing only the filtered elements on which applied function retured true.
flatMap	Similar to map, but flattens the results.
union	Returns a new RDD containing the union of elements of two RDDs.
distinct	Return a new RDD containing only the distinct elements.
intersection	Returns a new RDD containing only the common elements of two RDDs.
reduceByKey	When input data is in key-value pair, it reduces the result based on key.
join	Joins two dataset based on a common feature.

Table 2.1: RDD Transformations

Table 2.1 above, describes some of the transformations which are frequently used. And Table 2.2 below, describes few actions which are frequently used. In this work we have mostly used these set of transformations and actions.

Actions	Description
reduce	Aggregates the elements based on the function applied.
count	Counts the total number of elements present in the dataset.
foreach	Applies the defined function on each element of the dataset.

Table 2.2: RDD Actions

2.3.2 Spark DataFrames

Another core API of Spark is DataFrames[10]. A Dataframe is like a table in a Relational Database Management System or it can also be thought of as similar to dataframe in Python. It has columns with specific names and provides operations like filter, group and all the normal operations which can be performed on tables with using queries. A Dataframe can also be created from an RDD. It is a newer API than RDDs. LIke RDD, it is also immutable. But unlike RDD, data here is stored in tables.

So when we are dealing with unstructured data such as torrent of texts it's better to use RDD and if we need a schema than Dataframe can be used.

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

With Dataframes the processing of large data becomes easier.

2.3.3 Hadoop Distributed File System (HDFS)

HDFS[11] is a distributed file system which is designed especially to store large data in a distributed fashion. A file is break into many blocks of specific size and gets stored in the data nodes. In order to make the system robust and fault tolerant the data is also duplicated in different data nodes. Figure 2.8 shows the basic architecture of HDFS.

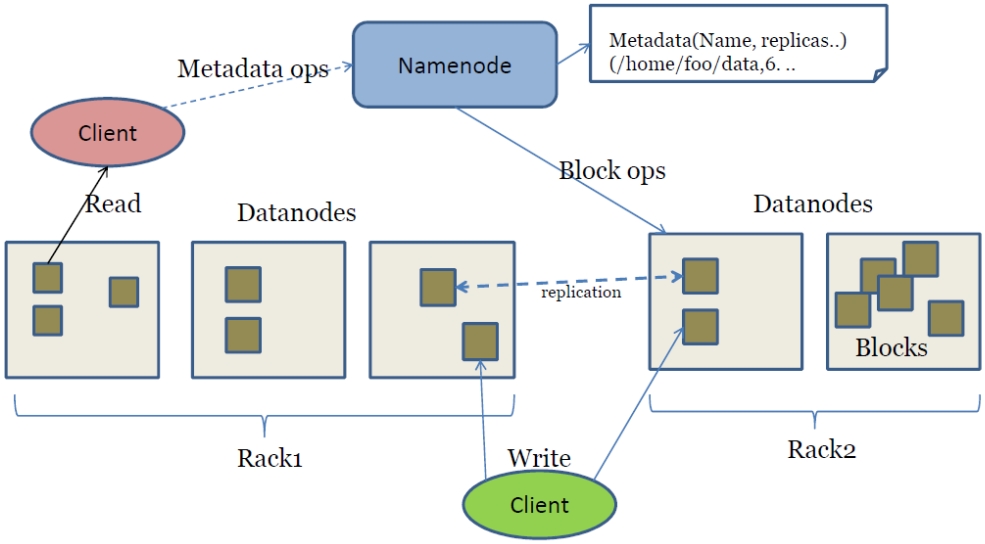

Figure 2.8: HDFS Architecture[11]

HDFS has a master/slave architecture. Namenode which acts as a master manages the file and control their access by clients. Operations like opening a file, closing a file, renaming files or directories all are taken care by Namenode. As visible from the above architectural diagram the Datanodes handles read and write requests from clients and also provides replication in blocks based on the instructions provided by Namenode. In this work we are using HDFS to store our files.

Goals[12] of HDFS are:

- **Fast recovery from hardware failures** : Since HDFS consists of cluster of nodes, there is a chance of failure of any node. Detecting

[11] `https://searchenginedeveloper.wordpress.com/2013/09/18/hadoop-distributed-file-system/`

[12] `https://www.ibm.com/analytics/hadoop/hdfs`

faults and fast recovery of data from that node is an important task of HDFS.

- **Access to streaming data** : HDFS is designed for batch processing rather than real-time use, so more emphasis is given on high throughput rates of data.
- **Large data sets** : HDFS can accommodate large data sets varying from gigabytes to terabytes in size. It can increase to large number of nodes in a cluster.
- **Portability** : HDFS is portable to multiple hardware platforms and compatible with different operating systems.

2.4 Clustering

Clustering is an Unsupervised Machine Learning technique. The unsupervised technique is used to draw inferences and find hidden patterns from a dataset. The basic idea of clustering is, given a data set consisting of a set of elements, group the elements into clusters such that elements in the same group are of similar kind or exhibit similar behavior or pattern. Elements should have minimum intra-cluster distance and maximum inter-cluster distances.

$$A_{inter} = \frac{\sum_{E_1,E_2 \in E_{pair-inter}} (1 - f(E_1, E_2))}{|E_{inter}|} \tag{2.2}$$

$$A_{intra} = \frac{\sum_{E_1,E_2 \in E_{pair-intra}} f(E_1, E_2)}{|E_{intra}|} \tag{2.3}$$

In the above equations E_1 and E_2 are the pair of elements which belongs to the set of inter-cluster and intra-cluster Elements E and f(E_1,E_2) = 1 if E_1 and E_2 are in the same predicted cluster, 0 otherwise.

The similarity of these elements are found based on their features. In this work we are considering *predicates* of the entities to find the similarity between them and we will group them based on their features a.k.a predicates. Similarity between individuals i.e f(E_1,E_2) in Equation 2.2 and 2.3 is generally found using Jaccard similarity [1] or Cosine Similarity.

$$Jaccard(E_1, E_2) = \frac{|E_1 \cap E_2|}{|E_1 \cup E_2|} \in [0, 1] \tag{2.4}$$

Here, E_1 and E_2 are the pair of candidate and Jaccard similarity is checked by taking fraction of their predicates which are common and union of their predicates.

$$Cosine(E_1, E_2) = \frac{V_1.V_2}{||V_1||.||V_2||} \in [0, 1] \tag{2.5}$$

Here, Cosine similarity between two entities can be found by using there respective feature vector. V_1 and V_2 are the feature vectors for E_1 and E_2.

Note: Higher the value more similar they are.

Example: Different scientific papers can be grouped into clusters based on their subject matter or based on the citations.

2.5 Why Clustering ?

We saw how big is LD. We are dealing with heterogeneous and noisy Linked Data which has no specific standard representation, hence clustering task becomes more important as it will help us to simplify bigger datasets. Instead of looking at the complete dataset at once we can now look only at a specific cluster based on our area of interest and draw the analysis.

Example: As in the case of our scientific paper example, if we want papers related to Big Data then we only have to look at the cluster which has papers related to Big Data.

The work presented in this thesis is focusing on presenting a scalable and generic clustering approach that can be applied to Knowledge Graphs and can give results in efficient time. To the best of our knowledge, this is the first scalable approach where Big Data is of concern. First, we will talk about various past researches that are done in this field in our next chapter before diving into our method of implementation.

Chapter 3

Related Work

In this chapter, we focus on previous work that is related to the problem stated in Introduction. Previous researches on this domain are restricted to perform clustering only on a subset of RDF data. Clustering algorithms like K-Means clustering, K-Medoid clustering and Hierarchical clustering have been used in the past. We will talk about these methods in detail in the following section.

3.1 K-Means Clustering

K-Means Clustering is a unsupervised machine learning technique which is used to find groups in the data such that elements within a group/ cluster exhibits same kind of behaviour. The number of groups/ cluster is represented by K. Let I $\in \{x_1, x_2, x_3, ..., x_n\}$ be the set of instances or data points or nodes of a graph.

Algorithm:

1. Select K cluster centers. (This could be any instance of the data.)
2. **Repeat**
3. **E-Step:** Assign each instance to closest center.
4. **M-Step:** Compute center of the formed cluster.
5. **Until** quality of the cluster does not improve anymore.
6. *Optional:* Adjust K and again start from Step 2.
7. Return formed cluster.

In case of RDF graphs, if two entities are strongly related then they will have more similar types of connections in comparison to other entities. *For Example:* The clustering of *subjects* can be done by looking at their *properties*. But there are various drawbacks in this approach.

Drawbacks:

- We don't know the value of K in advance. So it is basically a hit and trial method.
- The solution is dependent on the starting point i.e the cluster center which we choose initially.
- When clusters are of different sizes this approach will not give accurate clusters.
- It is not scalable as distance is calculated between every points.

3.2 K-Medoid Clustering

K-Medoid clustering works almost same as K-Means clustering except one thing that medoids are taken as centers of the cluster instead of mean. The medoid can be calculated as:

$$m = medoid(C) = argmin_{x \in C} \sum_{j=1}^{n} d(x, x_j) \tag{3.1}$$

where, cluster $C = \{x_1, x_2, ..., x_n\}$ and d is the distance. The individual who has the lowest distance among others will be selected as medoid.

[2, 3] used this technique to perform clustering. The work presented in [2] applied this technique to performed Graph Clustering. Graph clustering means finding K disjoint partition of vertices of the graph where K is the desired number of clusters. [2] performed the task to cluster similar kinds of vertices together based on some features by two approaches. The first approach used was the k-medoids algorithm and the second approach used was the Girvan-Newman method which is based on edge-betweenness centrality. And the clustering performance was measured by considering the pairwise intra-cluster accuracy [2] and pair-wise inter-cluster accuracy defined in Equation 2.2 and 2.3.

In the **Graph k-medoids** approach, based on the number of hops between nodes all the nodes of the graphs are assigned to the nearest *center* and the process continues until the cluster converges. Apart from the disadvantages stated above for k-means clustering, this approach had several other problems.
Disadvantage: The distance between various nodes could be the same with different cluster medoids and in that case, a cluster is randomly selected which could be wrong. This shows that graph distance is highly sensitive to the edges. Adding a single shortcut to the graph can affect various nodes and hence the clustering performance. It is not scalable as calculating the pairwise node distances will take $O(|V^3|)$ operations with $O(|V^2|)$ space complexity.

Girvan-Newman algorithm was based on edge removal from the graph to achieve clustering. Edges were ranked based on their betweenness and the edge with the highest score was removed. Then the connected components will be the clusters. The process continues until desired number of connected components are achieved. But this approach takes $O(|V||E|)$ operations to calculate edge betweenness for links. Hence it was also not scalable.

In order to overcome this limitation of complexity, Network structure indices (NSIs) has been used into these methods. NSI is a technique for indexing graph structure and efficiently finding shortest paths. The index has a set of node annotations [2] combined with a distance measure. NSIs provided fast approximation of the distances and can be paired with those algorithms to find shortest path between nodes. The DTZ - distance to zone - index was used. DTZ created d random partitions called dimensions and each dimensions has z random partitions called *zones*. Distance between each node and zones across each dimensions are stored by DTZ's annotations. It needs $O(|E|zd)$ time and $O(|V|zd)$ space. z and d being $<< |V|$ exact graph distances can be calculated very quickly. But there is a trade-off, if the number of random partitions are less the nodes will become closer to many medoids and if partitions are more then DTZ distance estimates will approach exact graph distance. k-medoid used this technique when assigning nodes to clusters and while calculating medoids. And Girvan-Newman algorithm used this technique to approximate edge betweenness centrality. Though more accurate clusters were found but the average run time was still large even for moderate-sized graphs, hence it was also not scalable.

The work done in [3] assumes that the resources and their relationships are defined in some generic ontology language. and that there are sufficient

features that could discriminate really different individuals. The clustering algorithm grouped these individuals into clusters based on their feature-set (F). Two advantages of selecting medoid has been reported in this paper [3]. First, it has no limitations on attributes types. Second, medoids are unaffected by outliers.

3.3 Hierarchical Clustering

Let I $\in \{x_1, x_2, x_3, ..., x_n\}$ be the set of instances or data points or nodes of a graph.

The hierarchical clustering of I is a sequence $C_1, C_2, ..C_n$ of partitions of I such that $C_1 = \{\{x_1\}\{x_2\}...\{x_n\}\}$ which means that there are n-clusters because all the instances are kept separately. C_2 has n-1 cluster and at the end C_n will have 1 cluster i.e $\{\{x_1, x_2, .., x_n\}\}$. In order to visualise the hierarchical clustering a dendrogram is created which gives you a binary tree representation of hierarchical clustering. Figure 3.1 shows the structure of the dendrogram.

$$\mathcal{C}_5 = \{\{x_1, x_2, x_3, x_4, x_5\}\}$$
$$\mathcal{C}_4 = \{\{x_1, x_2, x_3, x_4\}, \{x_5\}\}$$
$$\mathcal{C}_3 = \{\{x_1, x_2\}, \{x_3, x_4\}, \{x_5\}\}$$
$$\mathcal{C}_2 = \{\{x_1, x_2\}, \{x_3\}, \{x_4\}, \{x_5\}\}$$
$$\mathcal{C}_1 = \{\{x_1\}, \{x_2\}, \{x_3\}, \{x_4\}, \{x_5\}\}$$

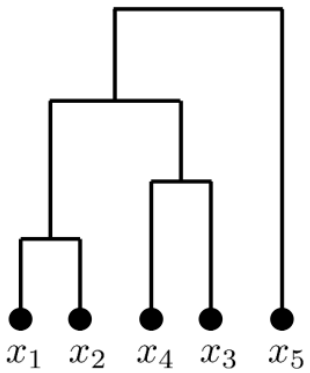

Figure 3.1: Dendrogram for Hierarchical clustering.

The height of the dendrogram depends on the distance/similarity at which two clusters have merged. Any desired number of clusters can be obtained by cutting the dendrogram at a specific level which is unlike K-Means clustering where we have to initial the number of clusters in the beginning. But at which level we have to cut the dendrogram to get the best clusters is again a hit and trial method.

There are two main types of hierarchical clustering.

- Agglomerative
- Divisive

Agglomerative: This approach starts with all the instances as the singleton clusters and then each step merges the closest cluster until left with a single cluster. This approach is also called bottom-up approach.

Divisive: It is reverse of Agglomerative, it starts with one cluster and each step splits the cluster until we get singleton clusters. This approach is also called top-down approach.

In hierarchical clustering a similarity or distance matrix is maintained to keep track of merged or split clusters. The most commonly used hierarchical technique is Agglomerative clustering.

Algorithm: Agglomerative Clustering

1. $C_1 = \{\{x_1\}\{x_2\}...\{x_n\}\}$
2. k = 1
3. for(k=1; $|C_k| > 1$; k=k+1) do
4. find "closest" pair $C_i, C_j \in C_k$ with $i \neq j$.
5. $C_{k+1} = C_k \backslash \{C_i, C_j\}$.
6. $C_{k+1} = C_{k+1} \cup \{C_i \cup C_j\}$.
7. Return $C_1, ..., C_n$.

There are different techniques which calculates the inter-cluster distance to merge the closest pair of clusters or to find the distance between the clusters.

Single-Linkage: Calculates the distance between two cluster as minimum distance between clusters member.

Complete-Linkage: Calculates the distance between two cluster as maximum distance between clusters member.

Average-Linkage: Calculates the distance between two cluster as average distance between all the pairs of elements of the clusters.

Centroid-Distance: Calculates the distance between two cluster as distance between two cluster centroids.

Similarity between individuals is generally found using Jaccard similarity [1] or Cosine Similarity defined in Equation 2.4 and 2.5 respectively. And using Jaccard similarity we can calculate the cluster similarity [1] as:

$$cluster\_sim(U_i^m, U_j^m) = \frac{\sum_{E_a \in U_i^m} \sum_{E_b \in U_j^m} Jaccard(E_a, E_b)}{|U_i^m||U_j^m|} \in [0, 1] \quad (3.2)$$

Here, U_i^m, U_j^m are the clusters. A threshold t value is set to pass the similarity condition. Equation 3.2 is used when Average-Linkage technique is used. This equation can be easily modified for other techniques as well based on their definition.

Research work described in [1, 4, 5, 6] has used this approach for clustering. [1, 4] used this technique to detect groups among a pool of individuals which are defined as a set of predicates. Similarity measures such as Jaccard Similarity, Cosine Similarity, Sorensen Similarity have been applied on the pairs of individuals to put them into clusters. In [4] clusters are evaluated using F-measure and the purity of a cluster is identified by taking the ratio of the superior class in the cluster by the size of cluster. Whereas Silhouette coefficient was used to determine the quality of the clusters in [1]. It also annotates the clusters based on the majority of *rdf:type* of the elements within a cluster. And in case *type* information is unavailable then classes are assigned with 'unknown' label. It also describes the structural schema of the data by making an Entity-Relationship (ER) diagram where attributes of the entities were the predicates and in order to show relationship between entities they considered predicates which were pointing to other resources rather than literals.

[5] put in each cluster nodes incident to each link/ predicate contained in partitions. It also uses Jaccard similarity between links to build a dendrogram and used Single Linkage agglomerative hierarchical clustering. Link clustering method has been used for graph clustering. In particular, only the similarity between those pairs of links has been computed that share a node.

The work done in [6] divided the schema graph into disjoint sub-graphs. In this work they are comparing only the neighbor nodes means each node is

compared with a set of nodes that are neighbors of that node. Elements of every cluster pair are combined when the similarity between the two clusters exceeds a given threshold. Intra-clustering similarity at each level has been calculated by averaging the similarity between each pair of data within a cluster to select the best level to cut the dendrogram.

But there are various drawbacks of this clustering approach.
Drawbacks:

- A similarity matrix is used which has quadratic space complexity i.e $O(n^2)$ where 'n' is the total number of elements, because the size of the matrix depends on the size of the dataset.
- Time complexity is also high as most of the time it is cubic.
- It tends to break large clusters.
- Sensitive to outliers.
- It is not scalable.

There are clustering algorithms which are particularly about Graph Clustering. In graph clustering, K partitions of vertices of a graph are made based on the similarities in their properties where K is the desired number of clusters.

3.4 Power iteration clustering (PIC)

Power iteration clustering[1] is a graph clustering technique which is efficient and scalable. This approach is included in *spark.mllib* library which works using GraphX. The edges property between the vertices of the graph is denoted by a non-negative similarity value. This similarity between the vertices can be found using Jaccard Similarity or Cosine Similarity from Equation 2.4 and 2.5 . A normalized similarity matrix is formed based on the similarity score between the nodes. This algorithm quantifies a pseudo eigenvector of the formed similarity matrix using power iteration[2] method and uses it to form cluster of vertices. But this approach also contains certain hyper-parameters which needs to tested based on hit and trial and can be considered as the

[1] `https://spark.apache.org/docs/latest/mllib-clustering.html#power-iteration-clustering-pic`

[2] `https://en.wikipedia.org/wiki/Power_iteration`

drawback of this approach.
Drawbacks:

- Need to define K i.e number of clusters beforehand.
- Need to define the maximum iteration for power method.
- Select random vertices to start the process.
- The convergence rate might be slow.

3.5 Spectral Clustering

In Spectral Clustering [7] a similarity matrix $S = (s_{ij})$ is formed from graph G where $(s_{ij})_{i,j=1...n}$ denotes the similarity value between two nodes i and j. Again the similarity can be calculated based on Cosine or Jaccard similarity. Hence the degree d of a node i can be found as $d_i = \sum_j s_{i,j}$. A graph Laplacian matrix[3] L is formed using $L = D - S$ where D is the diagonal matrix of degrees of all the nodes and S is the similarity matrix. Then this clustering algorithm proceed with an eigenvalue decomposition[4] of the Laplacian matrix and K non zero smallest eigen values are selected. The value of K depends upon the number of clusters we want. These K eigen values forms an eigen vector which acts as a starting point for the K-Means clustering algorithm. Then with the help of k-Means clustering algorithm desired clusters are formed. Hence this approach also cannot be considered as scalable.

The work done in [8] based on the idea of spectral clustering trained an AutoEncoder named *GraphEncoder* using deep learning techniques and performed graph clustering. They showed that GraphEncoder works better than spectral clustering and is more efficient and the complexity is much lower than the later and works fine with the large datasets.

[3] `https://en.wikipedia.org/wiki/Laplacian_matrix`
[4] `https://en.wikipedia.org/wiki/Eigendecomposition_of_a_matrix`

Chapter 4

Implementation

In this chapter, we discuss in detail the implementation of our approach. In this work we have used minHash Locality Sensitive Hashing (LSH) technique. The reason behind choosing this technique is that this method is a data-independent method, we don't need to specify any k value like in the case of k-means clustering and in hierarchical clustering where we have to decide at which level cutting the dendrogram will give better clusters. Because LSH hashes similar items into same bucket with higher probability we can use it for data clustering. We shall see in the evaluation chapter if this technique proved to be scalable or not. This work is done in Scala which is Spark's native programming language.

Before diving into the technical details about the LSH algorithm let's first see the overall architecture of our approach in the following section.

4.1 Architecture

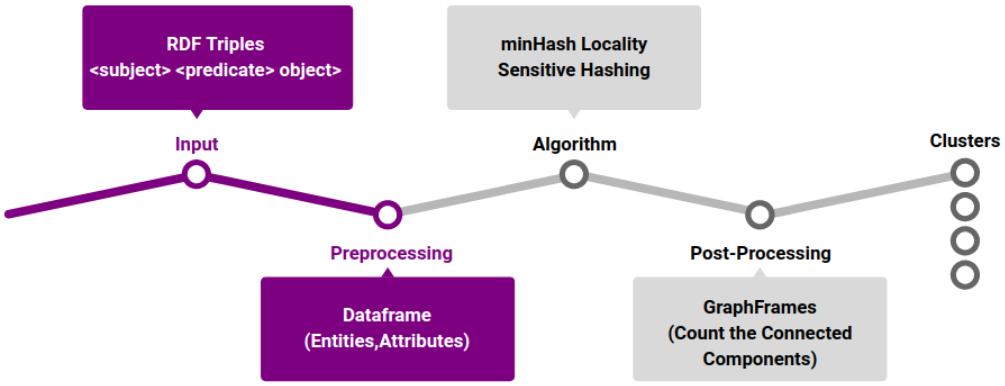

Figure 4.1: Approach

Input: As shown in Figure 4.1 the input to our model is a RDF triples which contains subject, predicate and object as shown in Table 4.1.

Triples	Subject	Predicate	Object
T1	The Great Gatsby	isWrittenBy	F.Scott Fitzgerald

Table 4.1: Input to our model

Pre-processing: In this step, we are filtering all the triples which contain blank nodes and literals which are not in English language. Hence our RDF triple will be *(subject, predicate, object)* $\in$ *IRIs* $\times$ *IRIs* $\times$ *Literals*. In this step we are creating a dataframe as shown in Table 4.2 in which for every subject we are taking there predicates and objects as attributes and passing this dataframe to our algorithm.

Entities	Attributes
Subject1	(Predicates,Objects)
Subject2	(Predicates,Objects)

Table 4.2: Output from pre-processing step

Clustering Algorithm: The dataframe from the pre-processing step is given to MinHash LSH algorithm. Let's understand the algorithm.

4.1.1 Locality Sensitive Hashing

Locality Sensitive Hashing (LSH) [9] hashes elements in the buckets using some functions where similar elements lands in the same bucket. It is considered as an important class of hashing technique. Elements in different buckets are considered different from each other.
Formally we can define LSH as follows.

In a metric space *(M, d)*, where M is a set and d is a distance function on M, an LSH family is a family of functions h that satisfy the following properties:

$$\forall p, q \in M \tag{4.1}$$

$$d(p, q) \leq r1 \Rightarrow Pr(h(p) = h(q)) \geq p1 \tag{4.2}$$

$$d(p, q) \geq r2 \Rightarrow Pr(h(p) = h(q)) \leq p2 \tag{4.3}$$

This LSH family is called (r1, r2, p1, p2)-sensitive. In Spark, different LSH families are implemented in separate classes (e.g., MinHash), and APIs for feature transformation, approximate similarity join and approximate nearest neighbor are provided in each class.

LSH defines a false positive as a pair of distant input features (with d(p,q)≥r2) which are hashed into the same bucket, and defines a false negative as a pair of nearby features (with d(p,q)≤r1) which are hashed into different buckets.

There are two LSH algorithms, first is 'Bucketed Random Projection for Euclidean Distance' and second is 'MinHash for Jaccard Distance'. Since our method requires Jaccard distances we have used MinHash algorithm.

4.1.2 MinHash for Jaccard Distance

As described in Apache Spark documentation[10], "MinHash is an LSH family for Jaccard distance where input features are sets of natural numbers. Jaccard distance of two sets is defined by the cardinality of their intersection and union:

$$d(A, B) = 1 - \frac{|A \cap B|}{|A \cup B|} \tag{4.4}$$

But as we have large set of elements computing Jaccard distance for every pair is not efficient and will cost us both in terms of time and space. MinHash on the other hand provides us an approximation of this measure where every large set is represented by some small fixed size representation. Hence adds scalability to the task. MinHash applies a random hash function g to each element in the set and take the minimum of all hashed values:

$$h(A) = min_{a \in A}(g(a)) \tag{4.5}$$

The input sets for MinHash are represented as binary vectors, where the

vector indices represent the elements themselves and the non-zero values in the vector represent the existence of that element in the set. Both dense and sparse vectors are supported but sparse vectors are proposed for efficiency. *For example, Vectors.sparse(10, Array[(2, 1.0), (3, 1.0), (5, 1.0)])* means there are 10 elements in the space. This set contains elem 2, elem 3 and elem 5. All non-zero values are treated as binary "1" values".

Note: Empty sets cannot be transformed by MinHash, therefore any input vector have to have at least one non-zero entry".

So once the algorithm receives its input, as shown in Figure 4.2, it vectorizes the attributes either by HashingTF or CountVectorizer method, then hashes of feature vectors are created and finally with a threshold value similarity between hashes has been found using approximate similarity join method.

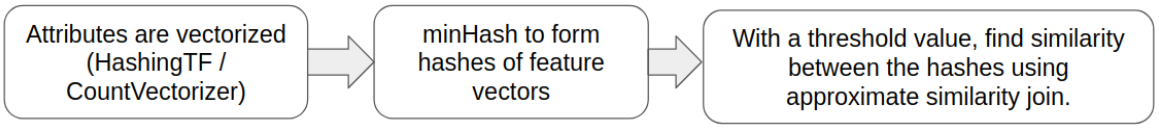

Figure 4.2: Algorithm Steps

4.1.3 Approximate Similarity Join

Approximate similarity join[1] needs two datasets. It then compares the entities of these datasets and returns pairs of entities whose distance is less than a threshold (t) provided by the user. As in this work we are working with a single dataset at a time we are performing self-join. Approximate Similarity join gives us the flexibility of self-join as well. But because of the self-join duplicate pairs were formed so we have removed all the duplicate pairs from the formed dataframe. In the returned output it also adds a distance column which tells how close the paired entities are. But for this clustering task we don't really need this distance column as we want to keep all the entities which share a common entity together in a single cluster. Hence we have dropped this column. The output is our clusters. Table 4.3 gives an idea about the way the clusters were formed.

[1] `https://spark.apache.org/docs/2.1.0/ml-features.html#approximate-similarity-join`

Cluster	Elements in cluster
1	(S1, S2, S3, S4)
2	(S2, S5, S8)
3	(S5, S9, S11)
4	(X1, Y1, Z1)
5	(Y1, Y2)

Table 4.3: Clusters formed from minHash LSH.

Post-processing: After receiving the output from minHash LSH, shown in Table 4.3, we realized the need for adding a post-processing step to our model. We used GraphFrames in our post-processing step.

4.1.4 Need for GraphFrames

The output which we were getting from minHash LSH was like the format given in below Table 4.3. From the above table we can see that LSH algorithm was giving us 5 clusters but if we look closely we will find out that S2 is present in cluster 1 and cluster 2, and S5 is present in cluster 2 and cluster 3, that means we can combine cluster 1, cluster 2, cluster 3; according to rule of Association which says that if A -> B and B -> C then A -> C. Similarly, we can combine cluster 4 and cluster 5 because of the common element Y1.

Hence, ideally there should be only 2 clusters but we were getting 5 clusters. There were two ways in which we could have solved this problem.

1. Manual Inspection of results:

This is the first method in which we employ a human annotator to combine the rows which follow the above relation. But this approach is extremely challenging and time-consuming because the size of our data is huge. And as we saw above that instead of 2 clusters 5 clusters were formed so which this huge amount of data the number of clusters will be large and checking the elements of each cluster will be highly infeasible. Hence we needed a faster and scalable approach. We used Graphframes to solve this problem.

2. Using Graphframes:

We used GraphFrames and counted the connected components of the graph. The total number of connected components will be our desired number of

clusters.

Connected components[2]: A graph is divided into subgraphs. In each subgraph two vertices are connected to each other by an edge between them and connected to no other vertices from other subgraphs. A vertex with no incident edges is itself a component. A graph that is itself connected has exactly one component, consisting of the whole graph. Figure 4.3 shows the connected components of a graph.

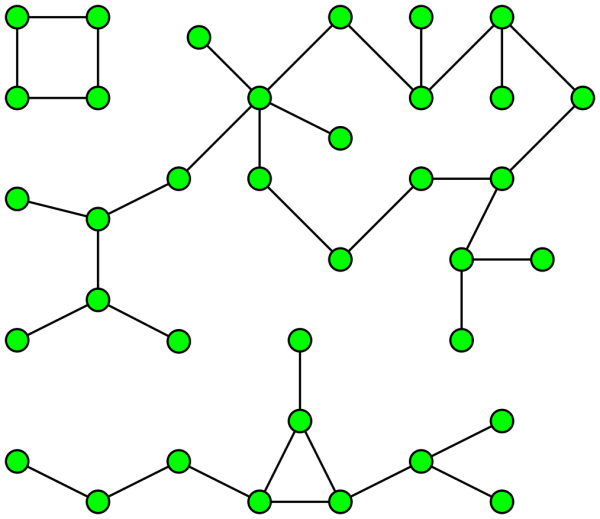

Figure 4.3: Graph with 3 connected components[2]

4.1.5 Why GraphFrames ? Why not GraphX ?

Now, one question which might come to our mind is why graphframes and why not graphx ? GraphFrames is a graph library that is based on dataframe. It is said that GraphX is to RDDs as GraphFrames is to DataFrames. GraphFrames is benefited from the scalability and high performance of dataframes. Graphframes includes all the GraphX features and provides some extended functionalities as well like dataframe based serialization, various high expressive graph queries like node degree.

To create graphframes we need a vertex dataframe and an edge dataframe. The vertex dataframe contains all the unique elements and edge dataframe stores the information about which elements are linked together. The vertex dataframe must have a column named *id* and the edge dataframe must have

[2]`https://en.wikipedia.org/wiki/Component_(graph_theory)`

two columns named *src* and *dsc*, which means source and destination of the edge. And since with graphframes we can represent a graph in the form of a dataframe it becomes easy to make powerful queries on the dataframe. Figure 4.4 shows how GraphFrames can be used to find clusters via code-snippet.

```
val col_1 = entities_dataframe.select("src")
val col_2 = entities_dataframe.select("dst")
val vertexDF = col_1.union(col_2).distinct().toDF("id")
val g = GraphFrame(vertexDF, entities_dataframe)
g.persist()
val connected_components = g.connectedComponents.run()
```

Figure 4.4: Using GraphFrames to find clusters.

Because LSH gives the output in a dataframe using GraphFrames proved to be a better choice than using GraphX. GraphX works with RDDs. Therefore if we had used GraphX we need to convert our dataframe to RDD which was an extra overhead and time-consuming. And as we are dealing with Big Data converting a bigger dataframe into RDD is highly inefficient and may takes ages to convert. And GraphX also demands a lot of pre-processing to find the connected components in a graph. Figure 4.5 shows via code-snippet how we can find connected components using GraphX.

```
def run(spark: SparkSession, entity_columns: RDD[(String, String)],
        featuredData_Df: DataFrame) {
  val col_1 = entity_columns.map(f => f._1)
  val col_2 = entity_columns.map(f => f._2)
  val entities_for_vertex = col_1.union(col_2).distinct
  val indexVertexID = (entities_for_vertex).zipWithIndex()
  val vertices: RDD[(VertexId, String)] = indexVertexID.map(f => (f._2, f._1))

  val tuples = entity_columns.keyBy(_._1).join(indexVertexID).map({
    case (k, ((s1, s2), l)) => (s2, (l, s1))
  })

  val edges: RDD[Edge[String]] = tuples.join(indexVertexID).map({
    case (k, ((si, p), oi)) => Edge(si, oi, p)
  })

  val graph = Graph(vertices, edges)
  val ccGraph = graph.connectedComponents()
  val clusters = ccGraph.vertices.map(f => {
    val key = f._2.toString()
    val value = f._1.toString()
    (key, value)
  }).reduceByKey(_ + ',' + _).map(f => (f._2))
  println("Total Number of clusters are: ", clusters.count())
```

Figure 4.5: Using GraphX to find clusters.

As we can see from these figures that GraphX asks for a pre-processing step of its own to find clusters whereas with GraphFrames we can directly use the dataframe for the same. GraphFrames helped us in reducing the complexity of our approach and made our model more scalable. Hence we have used GraphFrames instead of GraphX.

Note: Currently, we have to add GraphFrames as a separate package into our project and it is still not a part of core Apache Spark. However, it is compatible with all the 1.6+ versions of Spark. Figure 4.6 shows how we can add this package in our Spark applications. We can also add the jar file directly into our IDE. Different versions of GraphFrames can be found from the link in the foot note[3].

```
> $SPARK_HOME/bin/spark-shell --packages graphframes:graphframes:0.7.0-spark2.4-s_2.11
```

Figure 4.6: To add Graphframe package.

Now, we have made our clusters. But how do we decide if clusters best groups the data? For that, we need to find the quality of the clusters formed.

4.2 Silhouette

Usually, the validity of the clustering is based on two concepts: (a) *Compactness:* how close the elements of a cluster are, and (b) *Separation:* how distinct a cluster is from another cluster. We are using silhouette coefficients (SC) for our evaluation[1].

$$sil(i) = \frac{b(i) - a(i)}{max(b(i), a(i))} \in [-1, 1] \tag{4.6}$$

For each individual, we calculate SC which tells us how similar the individual is to other individuals in its own cluster compared to the representatives of the second nearest cluster. In equation 4.7 let i be that individual in a cluster C, a(i) is the average dissimilarity of i with other individuals in C and b(i) is the average dissimilarity of i with representatives of the closest cluster to C. SC ranges from -1 to 1 where -1 is considered as bad cluster structure, 0 is considered as indifferent and 1 is considered as good cluster structure i.e good assignments of individuals. If the SC > 0.7 we can consider it as strong cluster structure and if SC > 0.5 we can say its a reasonable cluster

[3] `https://spark-packages.org/package/graphframes/graphframes`

structure. Hence we can calculate the average score for the entire cluster based on sil(i) $\forall i \in C$.

$$Average\_score(C) = \frac{\sum_{i=1}^{n} sil(i)}{n} \in [-1, 1] \tag{4.7}$$

Figure 4.7 shows via code-snippet how we can calculate the silhouette score when we used GraphFrames approach.

```
var silhoutte_input = connected_components_.join(featured_Data, "entities")
val evaluator = new ClusteringEvaluator().setPredictionCol("prediction").
  setFeaturesCol("features").setMetricName("silhouette")
val silhouette = evaluator.evaluate(silhoutte_input)
println(s"Silhouette for LSH = $silhouette")
```

Figure 4.7: Calculating Silhouette score when used GraphFrames approach.

If we had used GraphX approach then calculating the silhouette score again involves a lot of overhead. We had to create a separate dataframe for silhouette with specific column names, had to maintain a set of values and had to perform several join operations which will increase the complexity and reduce the scalability and efficiency of our approach.

```
val schema = StructType(
  StructField("ids", LongType, true) ::
    StructField("prediction", IntegerType, true) :: Nil)
var initialDF = spark.createDataFrame(spark.sparkContext.emptyRDD[Row], schema)
val all_id = indexVertexID.toDF("entity", "ids").select("ids")

set_.map(f => {
  val id_entity = f.toDF("ids")
  val clu = id_entity.withColumn("prediction", lit(i))
  i += 1
  initialDF = initialDF.union(clu)

})
val id_prediction_entity = initialDF.join(indexVertexID.
  toDF("entities", "ids"), "ids")
val silhoutte_input = id_prediction_entity.join(featuredData_Df, "entities").
  drop("attributes", "words", "filtered_words")
val predictions = silhoutte_input.drop("entities", "ids")
val evaluator = new ClusteringEvaluator()
val silhouette = evaluator.evaluate(predictions)
println(s"Silhouette for LSH = $silhouette")
```

Figure 4.8: Calculating Silhouette score when used GraphX approach.

4.3 Challenges and Improvement

It turned out that considering both the predicates and objects was not a good idea and the results were not good because of two main reasons. First, some of the predicates had multiple object values and second, predicates like Description, Abstract, Text, and Comment contains long text which was misleading. Hence, we made a small change in our pre-processing step.

Preprocessing: As shown in Table 4.4, instead of taking both the predicates and objects we took only the predicates based on the idea that different classes will have different features set.

Entities	**Attributes**
Subject1	(Predicates)
Subject2	(Predicates)

Table 4.4: Output from pre-processing step

The overall architecture of our approach is shown in Figure 4.9.

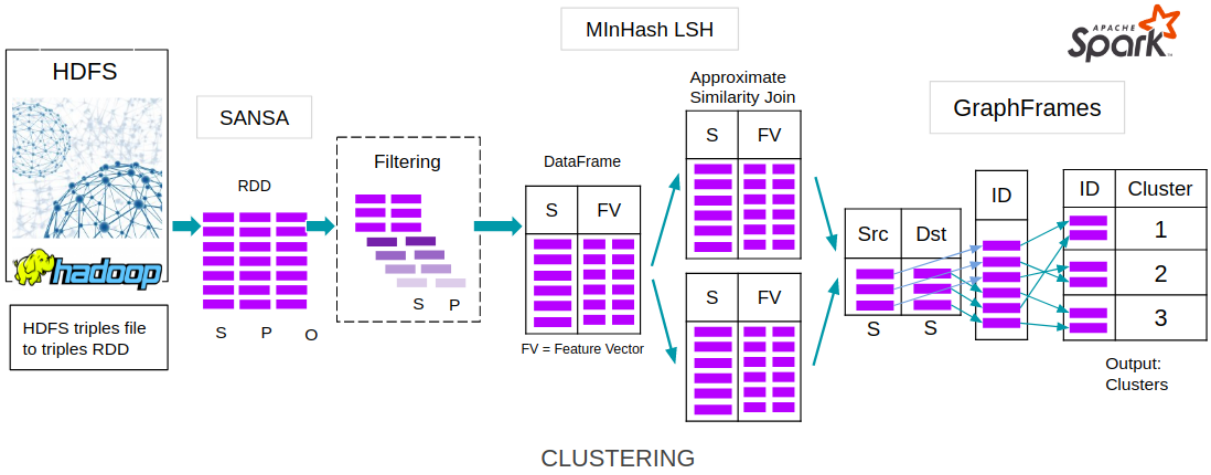

Figure 4.9: Architecture Pipeline

We will see in our evaluation chapter how much score did we get from using LSH, which will tell us about the quality of the formed clusters i.e how good our clusters are. The code is available at the github link[4].

[4] `https://github.com/agarwalpratikkumar/Scalable-RDF-Clustering`

Chapter 5

Evaluation

Now as we are familiar with the complete procedure of our algorithm, in this chapter we will be evaluating our implementation on different datasets and with this evaluation we would be able to justify if our approach is scalable or not. In this chapter, we will also evaluate the running time between HashingTF and CountVectorizer method and will find out which one is faster and why. We will do error analysis which will tell us the reason behind bad clusters. Before diving into the evaluation results, in the following section, we will know about the environment in which we have tested our implementation. We have tested our approach on a Spark cluster[1], The following section talks about the configuration details of the cluster.

5.1 Experimental Setup

To evaluate the performance of our approach which is based on Spark and Scala, we deployed it on the Spark cluster. We have used Spark Standalone Mode[2], which is a simple cluster manager which makes it easy to set up a cluster, with Spark version 2.2.1 and Scala with version 2.11.11. We ran our experiments on a private cluster of Smart Data Analytics (SDA) group of the University of Bonn which consists of 4 servers. Servers have a total of 256 cores [11], and each server has Xeon Intel CPUs at 2.3GHz, 256GB of RAM and 400GB of disks space, running Ubuntu 16.04.3 LTS (Xenial) and connected in a Gigabit Ethernet[3]. Each Spark executor is assigned a memory of 250GB.

[1] `https://en.wikipedia.org/wiki/Computer_cluster`

[2] `https://spark.apache.org/docs/latest/spark-standalone.html`

[3] `https://en.wikipedia.org/wiki/Gigabit_Ethernetnetwork`

5.1.1 Datasets

We have evaluated our results on different datasets. The information about the dataset can be found in Table 5.1. DBpedia dataset can be downloaded from DBpedia, the link to download the BIRT dataset is provided in the footnote and BSBM dataset can be downloaded using following commands:

wget http://downloads.sourceforge.net/project/bsbmtools/bsbmtools/bsbmtools-0.2/bsbmtools-v0.2.zip

unzip bsbmtools-v0.2.zip

cd bsbmtools-0.2/

To generate the dataset of specific size:
./generate -fc -s nt -fn BSBM_20GB -pc 233368

5.2 Preliminary Results

Dataset	Category	Size	# of Triples	Time Taken (HashingTF)	Silhouette Score
BIRT[4]	Very Small	82.6 KB	4565	1 Min.	**1.0**
DBpedia	Small	24.3 MB	79534	1 Min.	**0.11**
BSBM + LUBM	Small	402 MB	2005135	7 Min.	**0.88**
BSBM	Medium	3 GB	12005557	30 Min.	0.89
BSBM	Large	20 GB	81980472	60 Min.	0.88

Table 5.1: Evaluation Results for different datasets.

We have stated earlier that if the SC > 0.7 we can consider it as a strong cluster structure and if SC > 0.5 we can say it's a reasonable cluster structure. After trials we fixed our threshold to 0.40. Looking at SC score in Table 5.1 we can say that the results are pretty good but there are three things that is worth analysing in detail.

1. Why the silhouette score for BSBM + LUBM dataset is 0.88 and why is it not 1.0 ?

[4]https://www.eclipse.org/birt/documentation/sample-database.php

2. Why the silhouette score of Dbpedia dataset is so small i.e 0.11 ?

3. Why we got the silhouette score for BIRT dataset as perfect 1.0 ?

In Table 5.1 we have mentioned the time taken by the HashingTF method only, the time taken by the HashingTF method was half the time taken by the CountVectorizer method. Figure 5.1 compares the time taken by both HashingTF and CountVectorizer. Table 5.2 compares both the techniques and shows us the reason behind this.

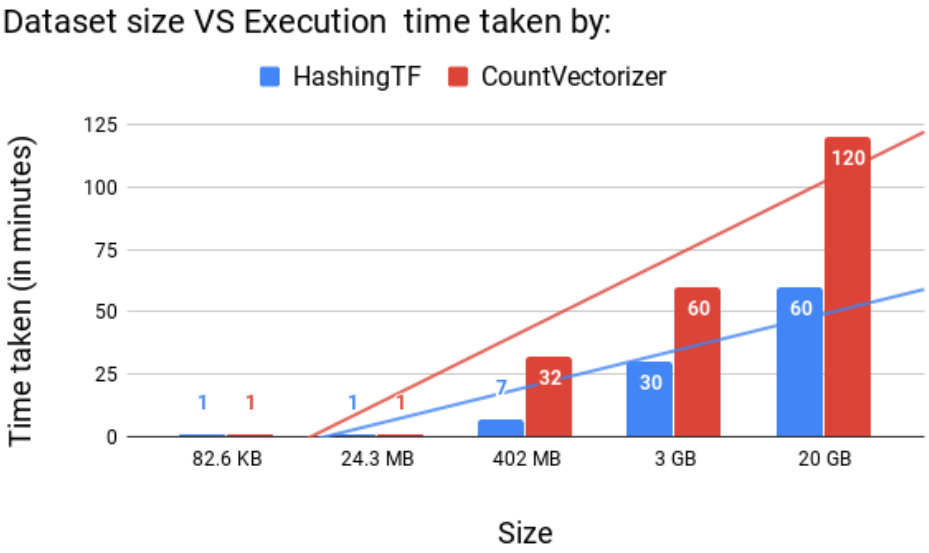

Figure 5.1: Execution time

HashingTF	**CountVectorizer**
It scans the data only once.	Scans data twice: once for building model and again for transformation.
Does not require any additional storage.	Needs extra space equal to number of unique features.
Back conversion is not possible.	Works well when we want to convert back the numerical feature vector to text.

Table 5.2: HashingTF VS CountVectorizer

5.2.1 Analysis of BSBM + LUBM dataset

In this dataset, there are total 19 classes, namely, Department, Full Professor, Associate Professor, Assistant Professor, Lecturer, Offer, Publication, Research Group, Graduate Course, Course, Reviewer, Undergraduate Students, Graduate Students, Product Features, Product Type, Producer, Vendor, Product, and Review. Figure 5.2 shows the result or clusters formed from this dataset.

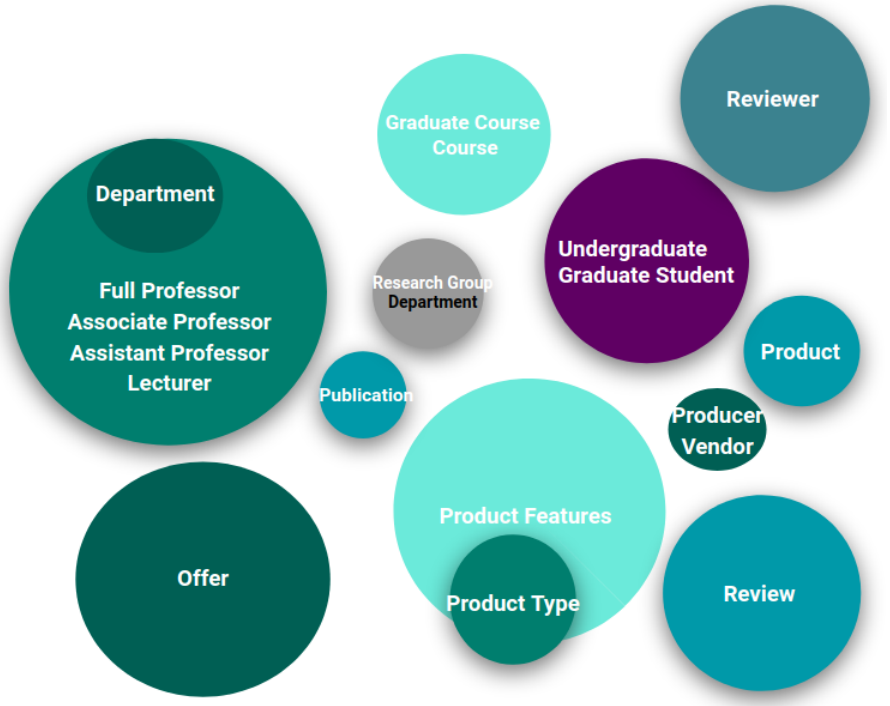

Figure 5.2: Result of BSBM+LUBM Dataset

We have seen in Table 5.1 that the silhouette score for this dataset was 0.88. so our concern is why it was not 1.0. On looking at the clusters, shown in Figure 5.2, we can see that *Department* is put into the same cluster with all the *Professors* and it is also mixed with the cluster of *Research Group*. And similarly, some of the *Product Types* are mixed with the cluster of *Product Features* and some of them are clustered separately. *Producer* and *Vendor* are also kept into the same cluster. So these things could also be the cause of the problem. Rest all the other classes are perfectly kept into their cluster separate from each other. All the *Undergraduate* and *Graduate students* are kept into the same clusters and similarly all the *courses* are also kept together in the same cluster so this is fine.

Table 5.3 shows the reason why *Department* was mixed with *Professors* and *Research Group*. Because the features of *Department* were clashing with the features of *Professors* and *Research Group* our algorithm mixed the *Department* class with the other classes.

Class	**Features**
Department	Type **Name** **subOrganisationOf**
Research Group	Type **subOrganisationOf**
Full Professor Associate Professor Assistant Professor Lecturer	Type **Name** TeacherOf UnderGraduateDegreeFrom MasterDegreeFrom DoctoralDegreeFrom WorksFor EmailID Telephone ResearchInterests

Table 5.3: BSBM + LUBM dataset analysis between different classes.

Table 5.4 shows the details about *Product Types* and *Product Features*. All the features of *Product Types* are same as features of *Product Features* except one i.e *subClassOf* feature hence all those entities which were having this extra feature were kept in different cluster and others were combined with *Product Features*.

Class	Features
Product Feature	Type Label Comment Publisher Date
Product Type	Type Label Comment **subClassOf** Publisher Date

Table 5.4: BSBM + LUBM dataset analysis between Product Feature and Product Type.

We saw that *Producer* and *Vendor* were kept in the same cluster. Table 5.5 gives the details about *Producer* and *Vendor*. Because all the features of both these classes were the same, these classes were kept in the same cluster.

Class	Features
Producer	Type Label Comment homepage Country Publisher Date
Vendor	Type Label Comment homepage Country Publisher Date

Table 5.5: BSBM + LUBM dataset analysis between Producer and Vendor.

Hence, the silhouette score was not 1.0. And we can say that if the feature set is different then the entities will cluster independently and if there are similarities beyond the specific threshold then they will be clustered together.

5.2.2 Analysis of DBpedia dataset

This dataset contains triples of sports players. Players from different games, namely, *Football, Basketball, Badminton, Baseball, Boxing, Cricket, Swimming, and Tennis* were taken. We know from Table 5.1 that our algorithm performs miserably in this dataset. Figure 5.3 shows the clusters formed for this dataset.

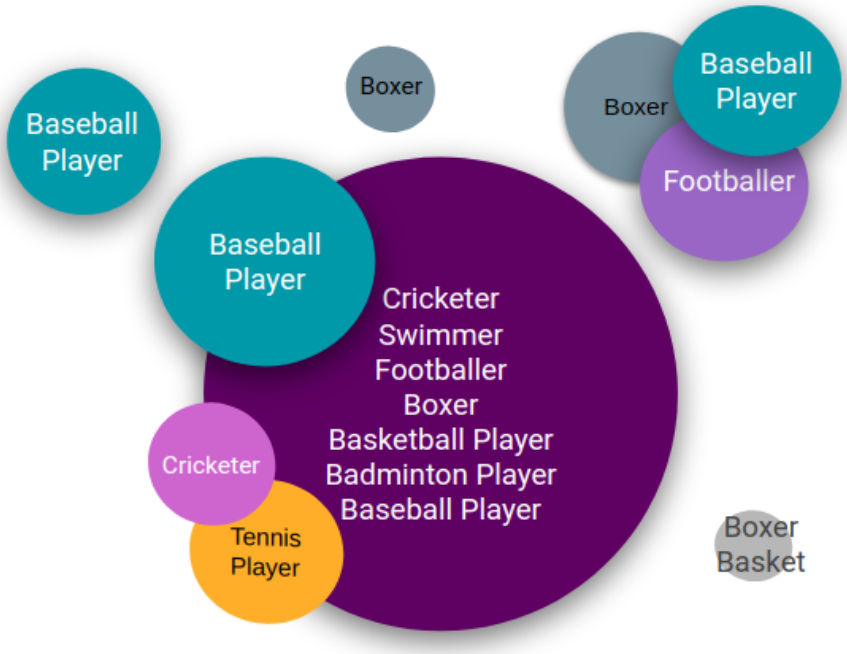

Figure 5.3: Result of Dbpedia Dataset

On looking at the above figure we will realize that all the classes are mixed. What I mean is if we look at the above figure we will find that for example, Baseball Players are present inside four clusters; they have their separate cluster and some of them are mixed with Boxers and Footballers and some of them are mixed with all the players. The similar case we can see for Boxers. Some of the boxers have their cluster some of them are mixed with basketball players and some are mixed with footballers and baseball players. And we can see that there is this bigger cluster in the image which has all the players from all the different games. So our major question is why is this happening like this? On analyzing the results we found that this is happening because of two main reasons. Firstly, there are a lot of overlapping features among all the players. Table 5.6 shows some of the features which were present in almost all the groups of players of all sports.

Players	Features
	Country
Boxer	Name
Footballer	Birthdate
Basketball Player	Type
Baseball Player	Description
Badminton Player	Abstract
Swimmer	WikiPageWikiLink
Cricketer	WikiPageExternallink
	WikiPageID

Table 5.6: Analysis for Sports players.

And very few sports-specific features were present like for tennis players features like *WimbeldonMixedResult, WimbeldonResult, WimbeldonDoubleResult, FrenchOpenDoubleResult, FrenchOpenResult*. Similarly for other sports. So because of this, our algorithm kept all the players in the same bucket as they were sharing a lot of features among themselves. And the second reason is that there was a non-uniform feature set among players from the same game. Table 5.7 shows the non-uniformity among players. We can see from the table that the first player had the only *Type* feature and the second player had some more features like *Hypernym, WikiPageRedirects, Type, isPrimaryTopicOf* and the third player had a different set of features. So because of such differences in their features, players from the same sport got clustered separately. And this kind of behavior was present among all the sports players.

Baseball Players	Features
Player 1	Type
Player 2	Hypernym WikiPageRedirects Type isPrimaryTopicOf
Player 3	WikiPageWikiLink isPrimaryTopicOf wasDerivedFrom Subject Depicton Thumbnail owlsameAs Comment Type Abstract Hypernym

Table 5.7: Analysis for Baseball players.

Hence the reason for that bigger cluster which contains players from all the sports categories. And with the same reason players with common features formed their small clusters as well. The result was so messy and so many that the clusters shown in Figure 5.3 is just the result from observing hundred results. So we got approximately ten clusters from observing just the top hundred results, but otherwise, there were lots and lots of small clusters and medium-size clusters. That was the reason why the silhouette score was just 0.11 as shown in Table 5.1.

5.2.3 Analysis of BIRT dataset

We saw that we got a silhouette score of 1.0 on this dataset. This was the smallest dataset that we used for our evaluation. Figure 5.3 shows the clusters formed for this dataset.

Figure 5.4: Result of BIRT Dataset

Different classes which were taken into consideration from this dataset were: *Employees, Customers, Offices, Orders, Products.* All the classes are perfectly kept into individual clusters. Table 5.8 and 5.9 shows some of the features of different classes present in the dataset.

Class	**Features**
Orders	orderNumber orderDate shippedDate status customerNumber
Products	productCode productName productVendor productDescription quantityInStock buyPrice

Table 5.8: Analysis of different classes of BIRT dataset.

Class	Features
Employees	employeeNumber lastName firstName extension email officeCode reportsTo jobTitle
Customers	customerNumber customerName phone addressLine1 city postalCode
Offices	officeCode city phone addressLine1 country postalCode territory

Table 5.9: Analysis of different classes of BIRT dataset.

The reason why we got perfect clusters in this dataset is that all the classes had their private set of features. Hence our algorithm could able to cluster them accurately.

Chapter 6

Conclusion

In this work, we have successfully clustered different RDF data in a scalable manner. We used minHash LSH with HashingTF and CountVectorizer to implement our approach and concluded that HashinfTF outperforms CountVectorizer in terms of speed. Our algorithm is fast and works efficiently but generates misleading results if the feature sets of different classes are identical. We saw why manual inspection of the results was not a good idea and hence concluded the need for GraphFrames and found the number of clusters by counting the number of connected components of the graph formed by GraphFrames.

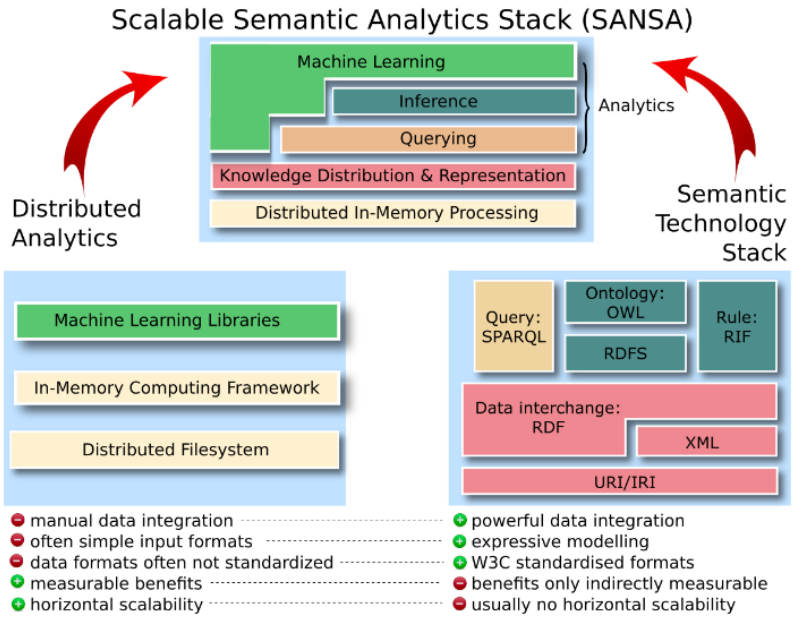

Figure 6.1: SANSA Stack[1]

This work will be added as a new RDF Clustering approach to the Machine Learning layer of Scalable Semantic Analytics Stack (SANSA)[1] stack shown in Figure 6.1. SANSA is a processing engine that provides distributed processing of Big Data especially RDF Data.

We have only considered the instance knowledge of the entities so this work can be extended to also take the schema knowledge into account. On considering the schema knowledge we can also develop the structural hierarchy of the entities and Entity-Relationship (ER) diagram that can be drawn to show the relationship between various sources. We can extend this work to other RDF datasets also like Wikidata, Yago, etc. Comparative analysis of our approach with other clustering approaches like K-Means clustering, Hierarchical clustering and Power iteration clustering and can get more insights about the performance of our approach.

[1] `http://sansa-stack.net/`

Bibliography

[1] K. Christodoulou, N. W. Paton, and A. A. Fernandes, "Structure inference for linked data sources using clustering," *Springer-Verlag Berlin Heidelberg 2015*, 2015.

[2] M. J. Rattigan, M. Maier, and D. Jensen, *Graph Clustering with Network Structure Indices*. 2007.

[3] N. Fanizzi and C. d. Amato, *A Hierarchical Clustering Method for Semantic Knowledge Bases*. 2007.

[4] S. Eddamiri, E. M. Zemmouri, and B. Asmaa, *An improved RDF data Clustering Algorithm.* 2018.

[5] S. Giannini, *RDF Data Clustering.*

[6] A. Algergawy, S. Massmann, and E. Rahm, *A Clustering-based Approach For Large-scale Ontology Matching.*

[7] https://www.cs.cmu.edu/ aarti/Class/10701/readings/Luxburg06_TR.pdf, *Spectral Clustering.*

[8] F. Tian, B. Gao, Q. Cui, E. Chen, and T.-Y. Liu, "Learning deep representation for graph clustering.," *Proceedings of the Twenty-Eighth AAAI Conference on Artificial Intelligence.*, 2014.

[9] https://spark.apache.org/docs/2.1.0/ml-features.htmllocality-sensitive hashing, *Locality Sensitive Hashing.*

[10] https://spark.apache.org/docs/2.1.0/ml-features.htmlminhash-for-jaccard distance, *MinHash for Jaccard Distance.*

[11] J. Lehmann, H. Jabeen, G. Sejdiu, and R. Dadwal, *Divided we stand out! Forging Cohorts for Numeric Outlier Detection in large scale knowledge graphs (CONOD).*