LAB DISTRIBUTED BIG DATA ANALYTICS (SANA RDF READER-PARSER)

Nayef Fayez Roqaya- Abdul Majeed Moustafa Alkattan- Emad Bahrami Rad

University of Bonn, Master Computer Science SS17

ABSTRACT

SANSA-RDF Reader is an abstraction library for reading and processing different types of RDF formats and integrating Semantic Data Technologies with Apache Spark technologies. In this report we are considering an extension to this library with three different types of RDF serialization formats. Specifically, RDF/XML, N-Quads and Turtle. Then, we will be discussing the different techniques and their drawbacks that could be used for implementing these readers for these formats.

***Index Terms*—** Turtle, parser, tokens, blocks, big data, distributed, Spark, Hadoop, dependencies, parallel processing

1. INTRODUCTION

Data Serialization formats are a way of sending and receiving Data in the Semantic Web Technologies spectrum, but one of the key challenges in Semantic Web technologies is having huge datasets which can not fit in the memory of a single machine; therefore, parallel processing have to be used in order to process big data, which is by definition huge. Apache spark is one of the prevalent parallel data processing frameworks which helps in exploiting the idea of parallel processing on multiple machines to process big data, yet this framework does not have any specific functions for reading different types of RDF serialization formats, and as a result does not allow us to use the huge amounts of RDF datasets, which are available all over the web, so an extension for the framework has to be considered in order to read and process these formats and utilize the huge amounts of RDF data, but since these formats have structures imposed on them we will be considering a solution that fits Apache Spark's programming model.

2. PROBLEM DEFINITION

There are many data serialization formats for representing information on the web, and Resource Description Framework (RDF) is one of the most important data models for achieving this goal and it is a standard for data interchange on the Web. In this report we consider four types of RDF serialization:Turtle, RDF/XML, N-Quad and Json-LD. Regarding the turtle format, to be able to clarify the parsing problem with turtle file format, we have to explain the meaning and structure of turtle file as RDF serialization format. Turtle file takes an extension(ttl= Terse RDF Triple Language). The key idea is that there are three types of RDF Terms: the first is a blank node that enables describing the characteristics of an entity that does not need to have a name. The second is a literal; this term describes a data value which does not have any special identity. The third term is RDF URI Reference that is used to identify resources uniquely [1]. On the one hand, Turtle file may contain only one specific keyword, which is Prefix. On the other hand, there are many different possible structures in turtle grammar that use(,), (;) and (.) to represent blocks [6] of text that have a meaning in a specific context, but, as far as we know, there is no specific approach to get blocks that contain one or more triple from a turtle file; moreover, there are no keywords in turtle that corresponds to RDF/XML and json-LD, which have keywords that specify start, end blocks, yet there are logic dependencies between the lines in a dataset, and we have considered this point carefully when we used spark because the data structures in spark are distributed, so we can not specify or predicate the location of data inside the partitions of a spark dataset. As a result, we have to find preserve the dependency between the file lines to ensure data integrity, yet we can not arbitrarily divide a Turtle file to parallel chunks in the same way that Apache spark reads a text file; moreover, when considering RDF/XML format, we have found that the format depends on tags(rdf:Description), (/rdf:Description) for imposing structure on blocks [2]. Also, in this format there are logic dependencies between the lines of the file and we have to find a method to preserve the structure in the same way as Turtle. Furthermore, Json-LD format has the exact same problem, but using different keywords (@context, @graph), so we can not follow sparks approach in parsing this format because, yet again, we have to keep the logic dependencies and find an approach to block json-ld statements. Lastly, N-Quads is a line-based, concrete serialization format for RDF. N-Quads statements are a sequence of RDF terms representing the subject, predicate, object and graph label of an RDF Triple, which basically denotes the graph that the triple belongs to. The terms are separated by space and the sequence of terms in terminated by a . and a new line. Here is the structure of an N-Quads statement (subject,predicate,object,graph). The subject and graph can be an IRI [1] or a blank node; moreover, The predicate can only be

an IRI, but the object can be either an IRI, a blank node or a literal. This format files do not have specific keywords or prefixes, so we can use the parser that Jena provides to parse this format directly as there are no logic dependences between lines and we can consider each line as a separate block. Finally, Big Data, which is often characterized by the so called 3Vs: Volume (the amount of data), Variety (different types of data), and Velocity (the speed of input). Often comes as an RDF serialization format, so supporting these formats is necessary for interacting with large scale data effectively, which is one of the priorities of the big data analysis. These problems are of big interest in the big data spectrum because large scale data is the main component of big data applications.

3. APPROACH

3.1. Sequences Vs. Parallel programming approaches

On the one hand, the parallel approach depends on using Apache spark for saving the data in distribute dataset such as RDD, DataFrames or Graphx, and then applying a parsing algorithm for every record in this data structure to get the results in a parallel from partitions. In this case, the performance is improved and the execution time is decreased significantly. On the other hand, when considering the sequential approach, we compare to Apache jena or RDF4j approach that reads a complete dataset into an internal model and then apply the parsing step on the resulting model without distributing the model to any other worker. In other words, this approach gives a result on a single machine , With one exception that is N-QUAD format, which has every line contains all the necessary information. In Turtle file, the tokens represent associated concepts and we cannot distribute the lines of data set inside the RDD directly in the same way of the word counting example, because we will lose the sequence of tokens that give a block. Figure 1 illustrates the logic mistake in distributed the line inside an RDD comparing with the figure 2 which

3.2. Solution approach

We found a simple and clever approach to block the data before we apply the parsing by spark. This solution depends on dividing the content of the file into blocks and each block represent one complete statement.

3.2.1. RDF/XML solution

With existing tags in RDF/XML, we can block the lines that from a block of meaning in a specific context into when a line that start with the string (rdf:Description) and a line after it end with the string (/rdf:Description). In this case, when the blocks are distributed inside the spark data structure, we will not have a problem in logic dependencies because each block will be parsed independently.

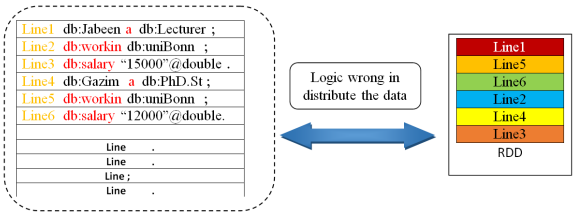

Fig. 1. Logic mistake in distributing the lines inside the RDD /Turtle

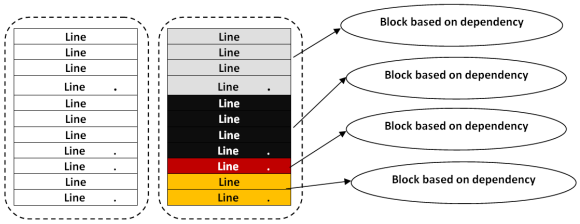

Fig. 2. Blocking the lines based on logic dependencies/Turtle

3.2.2. Json-LD solution

Json-Ld also has a structure, and we can get the(@context)json object from the file, which is basically prefixes, and get the main data object that is in(@graph). In spark, there is a function for reading json files, but this function has a limitation because it cannot read the json-Ld file without appending all the lines in the file into one line. This issue need to be solved, but until now there is no clear documentation about this issue or how can we avoid the appending step.

3.2.3. Turtle solution

In turtle files there are no keywords such as RDF/XML or Json-LD so we cannot build a regex for catching the file depending on keywords, but there is a golden rule in turtle files. Each triple is finished by a dot (.), so we investigate this idea for getting the contents of a file to completeblocks where every block may contain (;) or (,) that will contribute in generating more triples from this block. The figure2 illustrates blocking approach in turtle file and how we kept the dependencies by using this approach. Lastly, spark will be applied on the blocks of text that have a meaning in a specific context and we will not have any problem when we distributed the blocks to one of sparks data structure.

3.2.4. N-QUAD solution

N-Quad format does not have keywords and there is no structure dependencies between lines, so we can usJena parser approach with spark to directly parse the file.

3.3. Limitations of solution

In the cases of Turtle, RDF/XML and JSON-LD, we can not use spark before blocking the lines, so these cases can not be managed by spark as a result of sparks way of working, which is randomly distributing the lines to chunks, which we dont have any control on, that makes it impossible to keep the dependencies between lines.

3.4. Spark data structures

3.4.1. RDD (Resilient Distributed Datasets

In this section, we briefly review the three types of Spark data structures such as (RDD, Data Frames and Graphx). Resilient Distributed Datasets(RDD) is one of sparks data structures that is an immutable distributed collection of objects. Basically, each RDD can be divided into partitions that can be distributed to the nodes of a cluster. The most important properties of RDD are that RDD is read only, partitioned collection of records and fault-tolerant collection of elements which we can operated in parallel. There are two main ways for create the RDDs. The first way is referencing a dataset in an external storage as in SparkContext textfile method. The second way is parallelizing an existing collection in the driver program. The simple example in Fig3 which illustrates the general idea of RDD and how it works [5].

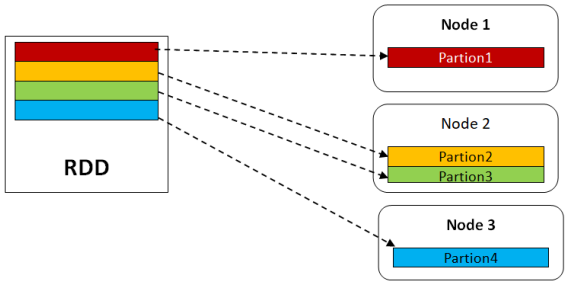

Fig. 3. RDD

3.4.2. DataFrames

Data frame is a distributed collection of data and takes the table (rows, columns) form which is illustrated in the figure 4 . We can create a Dataframe by three ways [3]:

- Using various data formats .For instance, loading the data from CSV.
- Specifying the schema programmatically.
- Loading the data from current rdd.

In this section, we take a closer look at the algorithm that is followed to Create dataframe [3]:

- Create tuples of data that include our data.
- Create RDD depending on the list of tuples in the previous step.
- Converting the tuples to rows
- Applying CreateDataFrame on RDD by helping sqlcontext

Fig. 4. Data Frames

3.4.3. Graphx

Graphx is one of the most important components in spark for graphs and graph-parallel computation. To grasp the idea of Graphx, we say that Graphx depends on extending an RDD by a new Graph abstraction. Graphx has many useful operations such as aggregateMessages, joinVertices and subgraph. GraphX supports representing vertices and edges especially when they are primal data types. The individual steps of the Graphx include: Create an RDD for the vertices, create an RDD for the edges then build an initial graph [4]. Figure5 illustrates the general idea of Graphx and how the vertices and edges tables have to be.

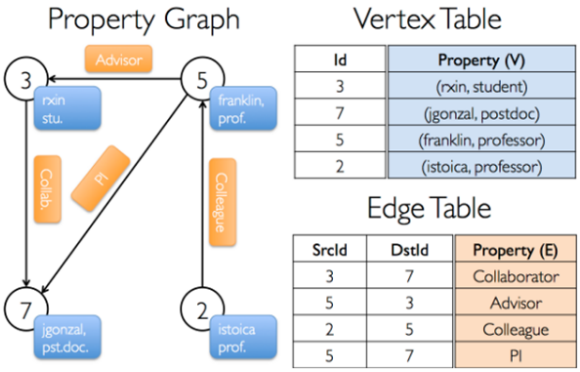

Fig. 5. Graphx

4. IMPLEMENTATION

Basically, our modules provides parser for (turtle , RDF/XML,Quad , Json-LD wich is not completed yet) rdf file and the Tables below illustrate the functionalities that are implemented in this project:

Turtle	
Function	**Explanation**
def readFile(fileName: String): ListBuffer[String] = { }	This function is responsible about read the input file.
def blockParse(block: String, prefixes: String): Array[String] = { }	This function is responsible about parsing the triples.
def prefixParse(line: String): Array[String] = { }	This function is responsible about collect the prefixes.
def Parsing_Turtle_File(TTLPath: String): Unit = {}	Parser from RDF4j .This function only to compare the results between parsing by spark and parsing by RDF4j.

Turtle		
Function	**Input**	**Out put**
def readFile(fileName: String): ListBuffer[String] = { }	Path data set file	List buffer of lines
def blockParse(block: String, prefixes: String): Array[String] = { }	Blocks of facts and prefixes	Array of triples
def prefixParse(line: String): Array[String] = { }	Line from data set	Prefixes in the file
def Parsing_Turtle_File(TTLPath: String): Unit = {}	Path file	Triples

Fig. 6. Turtle Implementation Methods

RDF/XML	
Function	**Explanation**
readFile(fileName: String, prefix : String, url: String): ListBuffer[String] = {}	This function is responsible for reading the input file and dividing it to blocks
def prefixParse(line : String): Array[String] ={}	This function is responsible for finding the prefixes in a file
def blockParse(block : String, prefixes : String): Array[Triple] ={}	This function is responsible for parsing each and every block
loadRDDFromXML (sparkSession: SparkSession, path : String, tag: String, numPartitions : Int): RDD[Triple] ={}	This function loads an RDD from an XML file using blocking method
loadDataFrameFromXML(session: SparkSession, path: String, tag: String, numPartitions : Int): DataFrame ={}	This function loads a Dataframe from an XML file using blocking method

RDF/XML		
Function	**Input**	**Output**
readFile(fileName: String, prefix : String, url: String): ListBuffer[String] = {}	FileName, prefix, tag	List of blocks
def prefixParse(line : String): Array[String] ={}	line	Array of prefixes
def blockParse(block : String, prefixes : String): Array[Triple] ={}	Block, prefixes	Array of Triples
loadRDDFromXML (sparkSession: SparkSession, path : String, tag: String, numPartitions : Int): RDD[Triple] ={}	SparkSession, path, rowTag, numberOfPartitions	RDD of Triples
loadDataFrameFromXML(session: SparkSession, path: String, tag: String, numPartitions : Int): DataFrame ={}	SparkSession, path, rowTag, numberOfPartitions	DataFrame of Triples

Fig. 7. RDF/XML Implementation Methods

Quad	
Function	**Explanation**
def load(session: SparkSession, path: URI): RDD[Quad]={}	Reads an N-Quad file and loads it into RDD
def loadToDataFrame(sparkSession: SparkSession, rdd: RDD[Quad]): DataFrame={}	Creates a DataFrame from the given RDD
def loadToDataFrame(sparkSession: SparkSession, path: URI): DataFrame={}	Reads an N-Quad file and loads it into RDD
def apply(triples: RDD[Triple])	Creates the GraphX Graph from the given RDD of Triple
def makeGraph(triples: RDD[String])	Creates the GraphX Graph from the given RDD of String

Quad		
Function	**Input**	**Output**
def load(session: SparkSession, path: URI): RDD[Quad]	An instance of SparkSession and the path to the input file	RDD of Jena Quad class
def loadToDataFrame(sparkSession: SparkSession, rdd: RDD[Quad]): DataFrame	An instance of SparkSession RDD of Quad	DataFrame
def loadToDataFrame(sparkSession: SparkSession, path: URI): DataFrame	An instance of SparkSession and the path to the input file	DataFrame
def apply(triples: RDD[Triple])	RDD of Triple	LoadGraph.graph an instance of GraphX Graph
def makeGraph(triples: RDD[String])	RDD of String	LoadGraph.graph an instance of GraphX Graph

Fig. 8. N-QUAD Implementation Methods

5. EVALUATION

There are a lot of evaluation methods that can be used to evaluate the solutions proposed in the report, one of them is using asymptotic analysis to Analise the different algorithms implemented in our solution, but for us timing the algorithms on different datasets using the same machine works the best, as we can compare our result with the results of other popular parsers, which do not use parallel processing techniques such as RDF4j and Jena, and then considering the best implementation which could scale to large or even huge amounts of data. In our solution we are dividing the implementation into two parts that are firstly reading the RDF serialized file and secondly parsing the resulting collection. We have implemented both a sequential parsing solution in Apache Jena, and our own parser which is parallel and does not care in which node it's working on. As a result, we have found that the parallel implementation is more time efficient than its sequential counterpart; moreover, the parallel version could scale up to huge amounts of data that can run on one or multiple clusters. We show a table of the results in figure9 [!h]

Data set	Size	Execution Time Without spark	Output /triples	Execution Time with spark	Output /triples	PC/OS
Turtle / for Parsing function						
Open budget (with prefixes)	8.28 MB	1.321 s	85246	0.502 s	85246	Have to be considered
topical_concepts_unredirected_en (without prefixes)	25 MB	3.658 s	170173	0.554 s	170173	Have to be considered
Quad / for Parsing function						
Linked Open Vocabularies	39.5 MB	1.1362s	203484	0.0789 s	203484	Have to be considered
disambiguations_en_uris_de	48.6 MB	0.8462s	221286	0.0555s	221286	Have to be considered
RDF/XML/ for Parsing function						
STW.RDF	14.1MB	4.521 s	109340	0.969 s	109340	Have to be considered
Germany	1.5 MG	0.094 s	10001	0.803 s	10001	Have to be considered

Fig. 9. Experiments results for evaluation

6. PROJECT TIME LINE

Project timeline can be divided on weekly plan. It provides a table with complete (weekly) plan for the project. Additional to the plan, we illustrate a table of details that includes information about our respective roles and contributions to the project throughout semester; moreover, we have used the incremental software process to achieve the milestones that we have specified at the beginning of the lab.

Contribution	TTL	RDF/XML	Quad	Json -LD	RDD /DataFrames/GraphX
Nayef Roqaya	✓	Contributed in this task	Contributed in this task	Contributed in this task	All members of group
Abdul Majeed Alkattan	Contributed in this task	✓	Contributed in this task	Contributed in this task	All members of group
Emad Bahrami Rad	Contributed in this task	Contributed in this task	✓	Contributed in this task	All members of group

Fig. 10. Project respective roles

0.1.06	05.06.	12.06.	19.06.	26.06.
Preparing and reading References	Start Discussion the tasks with member group	Start analyzing the requirements	Start design the classes and functions	Start implementation
Preparing and reading References	Collecting the requirements	Design prototype	Update the prototype	Update the prototype
03.07.	10.07.	17.07.	20.7.	23 .7.
implementation	Meeting with Supervisor	Start test the results	Start evaluation	Prepare the presentation
Update the module (simple system)	Update the Module	Update the Module	Write the report	Finish

Fig. 11. Project Time Line

7. REFERENCES

[1] Jens Lehmann. Knowlage graph course, 2016. RDF serialization and SPARQL chapter, https://sewiki.iai.uni-bonn.de/teaching/lectures/kga/2016/start.

[2] Claudio Reggiani. Analysis of structured rdf/xml data using spark and graphx, 2016. RDF/XML data using Spark and GraphX, https://www.kuleuven-kulak.be/benelearn/papers/Benelearn_2016_paper_34.pdf.

[3] Apache Spark. Data frames, 2017. Data Structure:DataFrames, https://spark.apache.org/docs/latest/sql-programming-guide.html.

[4] Apache Spark. Graphx, 2017. Data Structure:Graphx, https://spark.apache.org/graphx/.

[5] Apache Spark. Resilient distributed datasets, 2017. Data Structure:Resilient Distributed Datasets, https://spark.apache.org/.

[6] World Wide Web Consortium (W3C). Rdf formates, 2017. RDF Format serializationProblem, https://www.w3.org/.