NetID : iv447
University ID: N17385760

XBRL

The Extensible Business Reporting Language (XBRL) is an XML format for financial information that is more amenable to automatic processing than traditional financial information representations such as PDF, HTML and text documents. XBRL uses XML that is difficult to process. There are mainly two documents in an XBRL report : XBRL instance document (this is the actual marked up information) and one or more taxonomies. The taxonomies are the descriptive documents. These documents contains the labels used in the instance document. They also contain the metadata such as format and relationships between labels. Instance document can contain facts and they are values in conjunction with a context. Contexts are what give the values meaning. When we analyse XBRL as big data problem, we can assume a hypothetical case, where all companies in USA are sending their financial reporting to a central organization for processing. According to 2010 statistics there were 27.9 million small businesses. Also IRS received around 32 million non-farm business tax return. So clearly, processing XBRL is an important big data challenge.

The method that can be implemented for processing these xbrl is : sending all the messages to the cluster (that works in parallel) and wait for a response denoting that the process is complete.The XBRL documents can be submitted to different cluster nodes that run spark and the information can be processed parallely.

Pre Processing of XBRL documents :

Before submitting it to cluster node the XBRL document need to be preprocessed :
That can be done using a XBRLParser :

A java class can be created that could be used in spark. The name of this class can be XBRLParser and it’s purpose will be to parse a string into an XML document and store it within the object. The object exposes a method that counts tags depending on the input string it receives. Internally this XBRLParser will be using existing Java classes made for parsing XML.

Small Part of an xbrl document :

```
<numericContext id="Current_AsOf" precision="18" cwa="true">
```

```
  <entity>
    <identifier scheme="http://www.sampleCompany.com">Sample
    Company</identifier>
  </entity>
  <period>
    <instant>2002-12-31</instant>
  </period>
  <unit>
    <measure>iso4217:EUR</measure>
  </unit>
</numericContext>
```

Let's assume that this document is given to XML Parser, following java code will parse it :

```
import java.io.File;
import javax.xml.parsers>*;
import org.w3c.dom.*;

public class XBRLParser {
public static void main(String [] args) {
  try{
        File inputfile = new File ();
                                    DocumentBuilderFactory dbFactory =
DocumentBuilderFactory.newInstance();
        DocumentBuilder dBuilder = dbFactory.newDocumentBuilder();
        Document doc = dBuilder.parse(inputfile);
        doc.getDocumentElement().normalize();
        System.out.println ( "Root Element : " + node.getNodeName());
        ….....
        …….. }
  catch(Exception e) { e.printStackTrace(); } } }
```

We can make use of distributed computing services of apache spark and REST API. Rest APIs are inherently conducive to parallel processing as each call to API is completely independent of any other call to the same API (and this is very important in parallel processing). This fact, in conjunction with the parallel computing capability of spark, can be leveraged to create a solution that solves the problem by delegating the API call to Spark's parallel

workers. Under this approach, one can package a specification for how to call the API along with the input data, and pass that to Spark to divide the effort among its workers (and tasks). The output can be assembled in set-level abstractions supported by Spark (like Dataframes or Datasets**)** and passed back to the calling program. This approach not only helps you turn a sequential execution into a parallel one with the least coding effort, but also makes it much easier to analyze and transform the returned result with an easier data abstraction model to work with. The performance benefit that we can get is tremendous in this approach. This turns a problem that takes incremental time for computation (that increases linearly with the number of records to process), to one that is much more efficient and scales linearly on a much lower slope — number of records to process divided by the number of cores available to process them. Theoretically, one can make the process constant time by having enough cores to process all of the records at once. So, the tremendous number of XBRL documents can be processed parallely using this approach.

The approach can be summarized as follows:

1. We can first read different sets of parameter values (that have to be sent to target REST API) from a file/table (in our case it will be XBRL document which will be preprocessed using XBRLParser) to a Spark Dataframe (say Input Data Frame).
2. Then the Input Data Frame is passed to the REST Data Source.
3. The REST Data Source returns the results to another Dataframe, say Result Data Frame.
4. Now we can use Spark SQL to explore, aggregate, and filter the result using the Result Data Frame.