

Text Mining using Spark

Objective: -

The objective of this assignment is to mine Wikipedia XML dump using Apache Spark. We also analyzed the performance by changing default configuration and different techniques to process the XML file.

Problem Statement: -

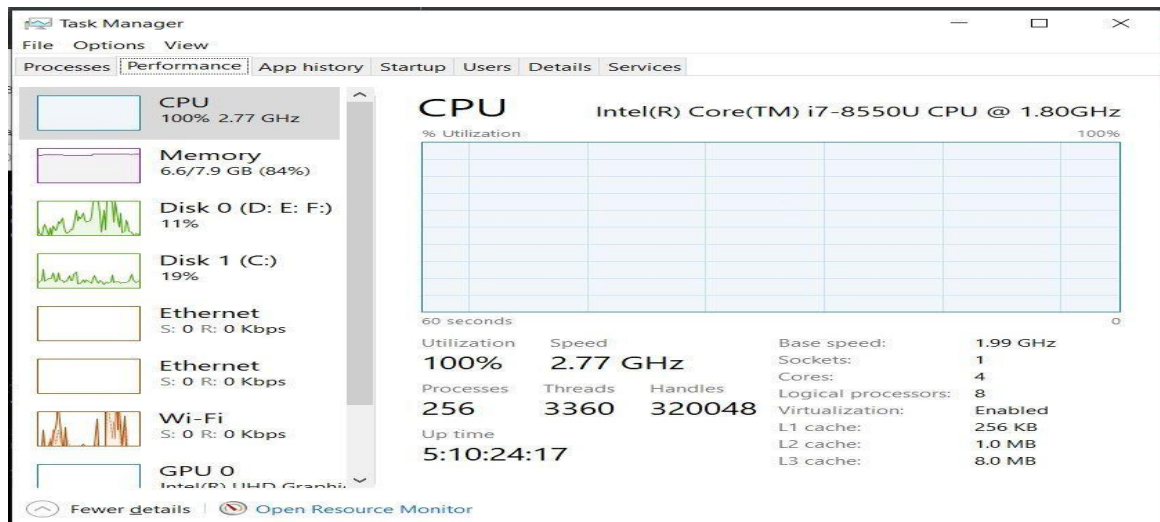
- Determine the total number of minor revision (<minor /> : tags) available in the dataset.
- List out page title and id of all pages that have at most five URL links in the text field.
- List out all the contributors with more than one contribution along with revision id. The list of revision id must be sorted in the descending order of timestamp.

Input Dataset: -

1. sample_data.xml (256 MB)
2. wiki_data_dump_32GB.xml (32 GB)

System Configuration: -

- Processor: - Intel® Core™ i7-8550HQ CPU @ 1.80GHz
- Installed memory (RAM): - 8.00 GB (7.9 GB)
- System type: - 64-bit Operating System, x64-based processor
- CPU status while Spark tasks getting executed:



Task Manager Snap

XML Data Source for Apache Spark using databricks:-

- A library for parsing and querying XML data with Apache Spark, for Spark SQL and Data Frames.
- This package supports to process format-free XML files in a distributed way, unlike JSON data source in Spark restricts in-line JSON format.

Requirements: -

- This library requires Spark 2.2+.
- Python environment (Canopy), JDK (1.8.0_211 on local system), Spark 2.2+ and winutils(Windows binaries for Hadoop versions)
- com.databricks:spark-xml_2.11:0.5.0 jar (download and put it in jar folder)

Procedure: -

- To process the XML data, first we need to convert XML data into data frame using below command.

```
read.format('com.databricks.spark.xml').options(rootTag='page').options(rowTag='page').load('file')
```

- This command will convert the XML data into data frames and generate a schema. We will register a temporary table using this schema to process the data.
- There are two different ways to process this data.
 - o Dataframe API
 - o HiveContext
- In this assignment, we will use SQLContext to process the data.
- The schema structure of the 32GB file:

```
>>> df.printSchema()
root
|-- id: long (nullable = true)
|-- ns: long (nullable = true)
|-- redirect: struct (nullable = true)
|   |-- _VALUE: string (nullable = true)
|   |-- _title: string (nullable = true)
|-- restrictions: string (nullable = true)
|-- revision: struct (nullable = true)
|   |-- comment: struct (nullable = true)
|   |   |-- _VALUE: string (nullable = true)
|   |   |-- _deleted: string (nullable = true)
|   |-- contributor: struct (nullable = true)
|   |   |-- _VALUE: string (nullable = true)
|   |   |-- _deleted: string (nullable = true)
|   |   |-- id: long (nullable = true)
|   |   |-- ip: string (nullable = true)
|   |   |-- username: string (nullable = true)
|   |-- format: string (nullable = true)
|   |-- id: long (nullable = true)
|   |-- minor: string (nullable = true)
|   |-- model: string (nullable = true)
|   |-- parentid: long (nullable = true)
|   |-- sha1: string (nullable = true)
|   |-- text: struct (nullable = true)
|   |   |-- _VALUE: string (nullable = true)
|   |   |-- _space: string (nullable = true)
|   |-- timestamp: string (nullable = true)
|-- title: string (nullable = true)
```

Dataframe Schema

a) Dataframe API: -

This approach allowed us to read XML file in Apache Spark using Dataframe API. There are different methods which can be used to get the desired output. We used collect and count methods.

collect(): - This method brings all the entries to the driver as a List therefore if there isn't enough memory you may get several exceptions (this is why it's not recommended to do a collect if you aren't sure that it will fit in your driver), besides it requires to transfer much more data.

count(): - This method sums the number of entries of the RDD for every partition, and it returns an integer consisting in that number, hence the data transfer is minimal.

There are two ways to read the XML file directly.

SparkSQL

SparkSQL – DSL (Domain Specific Language) way

1) SparkSQL: -

```
sqlContext.sql("select COUNT(revision.minor) from XMLTABLE where revision.minor='').collect()

sqlContext.sql("select id,title from XMLTABLE where (LENGTH(revision.text._VALUE) -
LENGTH(REPLACE(revision.text._VALUE,'http','')))/4 > 0 and (LENGTH(revision.text._VALUE) -
LENGTH(REPLACE(revision.text._VALUE,'http','')))/4 < 6").collect()

sqlContext.sql("select revision.id AS 'Revision id',revision.contributor.id AS 'Contributor id', revision.contributor.username AS
'Contributor Name', revision.timestamp from XMLTABLE where revision.contributor.id in (select revision.contributor.id from
XMLTABLE group by revision.contributor.id having COUNT(revision.contributor.id)>1) order by
revision.contributor.id,revision.timestamp desc").collect()
```

Query 1, Query 2 and Query 3 using collect() method

```
1) output_data = sqlContext.sql("select revision.minor as 'counts' from XMLTABLE where revision.minor=''")
   output_data.select('counts')
   output_data.toDF("counts").count()

2) output_data = sqlContext.sql("select id,title from XMLTABLE where (LENGTH(revision.text._VALUE) - LENGTH(REPLACE(revision.text._VALUE,'http','')))/4 > 0 and
(LENGTH(revision.text._VALUE) - LENGTH(REPLACE(revision.text._VALUE,'http','')))/4 < 6")
   output_data.select('id','title')
   output_data.toDF('id','title').count()

3) output_data = sqlContext.sql("select revision.id AS 'Revision id',revision.contributor.id AS 'Contributor id', revision.contributor.username AS 'Contributor Name',
revision.timestamp from XMLTABLE where revision.contributor.id in (select revision.contributor.id from XMLTABLE group by revision.contributor.id having
COUNT(revision.contributor.id)>1) order by revision.contributor.id,revision.timestamp")
   output_data.select('Revision id','Contributor id','Contributor Name','timestamp')
   output_data.toDF('Revision id','Contributor id','Contributor Name','timestamp').count()
```

Query 1, Query 2 and Query 3 using count() method

2) SparkSQL – DSL way: -

Number of minor tags can be counted using DSL way as follow:

df.select('revision.minor').filter(df['revision.minor']!='None').count()

```
>>> df.select('revision.minor').filter(df['revision.minor']!='None').count()
3710
>>>
```

SparkSQL – DSL way (Query 1)- 256MB Sample File

b) HiveContext: -

- In this approach, we first create a hive table and then access this hive table using HiveContext in Apache Spark.

Output:-

Whole output files attached along with source code but to have the count of such tags please refer the screenshot for Query1 (total number of minor revision) for reference:

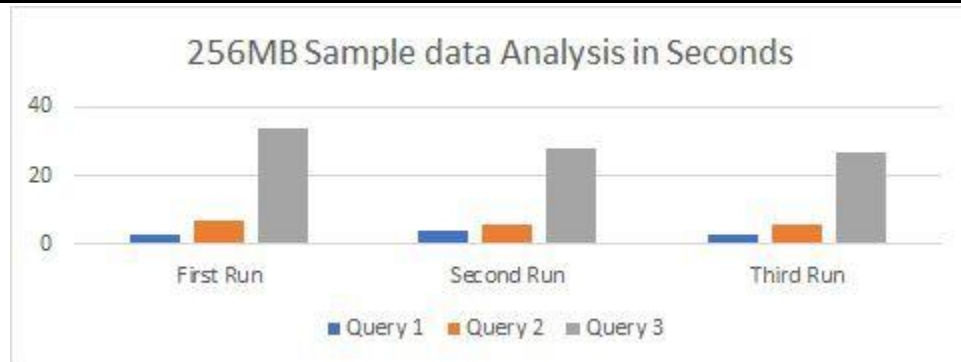
```
>>> df.registerTempTable("XMLTABLE")
>>> output_data = sqlContext.sql("select revision.minor as `counts` from XMLTABLE where revision.minor=''")
>>> output_data.select('counts')
DataFrame[counts: string]
>>> output_data.toDF('counts').count()
3346414
>>>
```

Query 1 count (32GB)

Data Visualization: -

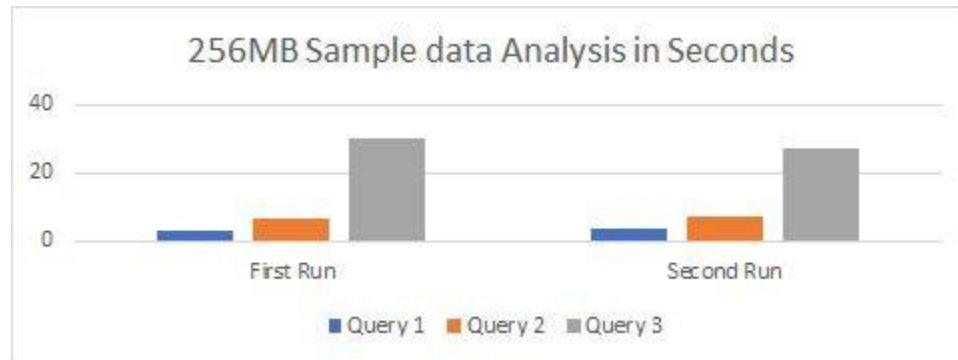
- 1) Comparing running time in seconds for all three queries for three runs. Using collect().

	First Run	Second Run	Third Run	Average
Query 1	3	4	3	3.3
Query 2	7	6	6	6.3
Query 3	34	28	27	29.7



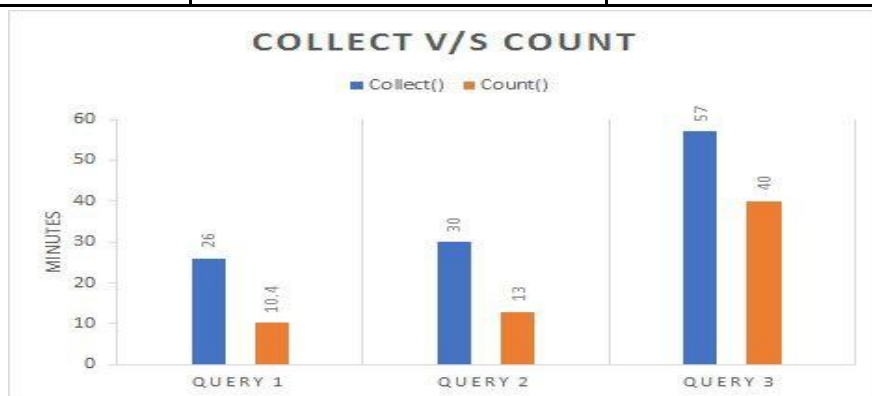
- 2) Running time in seconds for 256MB data using count().

	First Run	Second Run	Average
Query 1	3.3	3.95	3.6
Query 2	6.9	7.1	7
Query 3	30.5	27.1	28.8



3) Average running time in minutes collect() v/s count() for 32GB file.

	Collect()	Count()
Query 1	26	10.4
Query 2	30	13
Query 3	57	40



Performance Tuning: -

- 1) Spark API's automatic optimization. Using the command `df.explain('True')`, we found out the optimization steps performed by Spark on RDD. Here `df` is the data frame variable.

```
>>> df.explain("true")
== Parsed Logical Plan ==
Relation[id#0L,ns#1L,redirect#2,restrictions#3,revision#4,title#5] XmlRelation(<function0>,Some(file:///SparkCourse/dump_32GB.xml),Map(rowtag -> page, roottag -> page, path -> file:///SparkCourse/dump_32GB.xml),null)

== Analyzed Logical Plan ==
id: bigint, ns: bigint, redirect: struct<VALUE:string, title:string>, restrictions: string, revision: struct<comment:struct<VALUE:string, _deleted:string>, contributor:struct<VALUE:string, _deleted:string, id:bigint, ip:string, username:string>, format:string, id:bigint, minor:string, model:string, parentId:bigint, sha1:string, text:struct<VALUE:string, _space:string, timestamp:string>, title: string
Relation[id#0L,ns#1L,redirect#2,restrictions#3,revision#4,title#5] XmlRelation(<function0>,Some(file:///SparkCourse/dump_32GB.xml),Map(rowtag -> page, roottag -> page, path -> file:///SparkCourse/dump_32GB.xml),null)

== Optimized Logical Plan ==
Relation[id#0L,ns#1L,redirect#2,restrictions#3,revision#4,title#5] XmlRelation(<function0>,Some(file:///SparkCourse/dump_32GB.xml),Map(rowtag -> page, roottag -> page, path -> file:///SparkCourse/dump_32GB.xml),null)

== Physical Plan ==
*(1) Scan XmlRelation(<function0>,Some(file:///SparkCourse/dump_32GB.xml),Map(rowtag -> page, roottag -> page, path -> file:///SparkCourse/dump_32GB.xml),null) [id#0L,ns#1L,redirect#2,restrictions#3,revision#4,t
title#5] PushedFilters: [], ReadSchema: struct<id:bigint,ns:bigint,redirect:struct<VALUE:string, title:string>,restrictions:string, revis...
```

Dag in Spark

Logic plan is the tree which represents both the schema and data. In the backend it automatically generates the Analysis plan and finally using analysis plan it comes up with the optimized logical plan. Analyzed logical plans usually go through couple of rules to get the task done. After that the optimized logical plan is produced. Lastly the optimized plan is converted to physical execution for further execution.

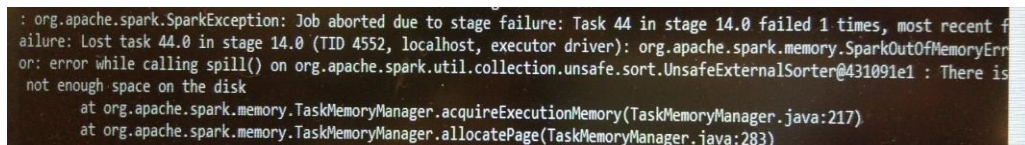
== Optimized Logical Plan ==

Relation[id#0L,ns#1L,redirect#2,restrictions#3,revision#4,title#5]

XmlRelation(<function0>,Some(file:///SparkCourse/dump_32GB.xml),Map(rowtag->page, roottag -> page, path -> file:///SparkCourse/dump_32GB.xml),null)

2) Using the count() method to process the XML:

- Primarily we used the collect() method which loaded all the data segments into the main memory and then started processing the queries. We got the consolidated result at the go but at the cost of memory.
- For small data test the issues was not noticed but when it came to 32GB file after almost 90% completion it threw error saying "There is no enough space on Disk".



```
: org.apache.spark.SparkException: Job aborted due to stage failure: Task 44 in stage 14.0 failed 1 times, most recent failure: Lost task 44.0 in stage 14.0 (TID 4552, localhost, executor driver): org.apache.spark.memory.SparkOutOfMemoryError: error while calling spill() on org.apache.spark.util.collection.unsafe.sort.UnsafeExternalSorter@431091e1 : There is not enough space on the disk
    at org.apache.spark.memory.TaskMemoryManager.acquireExecutionMemory(TaskMemoryManager.java:217)
    at org.apache.spark.memory.TaskMemoryManager.allocatePage(TaskMemoryManager.java:283)
```

Memory Issue

- This was due to the fact that it used to create almost equal size of temporary data while execution which is very much memory inefficient. We managed to run the program at last by cleaning up the disk.
- So now we used count() method which used to partition the whole data and then generate individual result which were then consolidated to generate the final output file.
- By this not only we resolved the memory issue but also achieved almost half execution time which was due to the fact that whole data was not being processed at the same time.
- Since we cannot afford to have very huge memory every time we found using count() to be the most efficient one.