UNIVERSITEIT VAN AMSTERDAM

Big Data - Lab Assignment 2

Group 19

Jasper Adema & Daphnee Chabal
(UvA ID: 11879394) (UvA ID: 11200898)

UNIVERSITEIT VAN AMSTERDAM

Question One

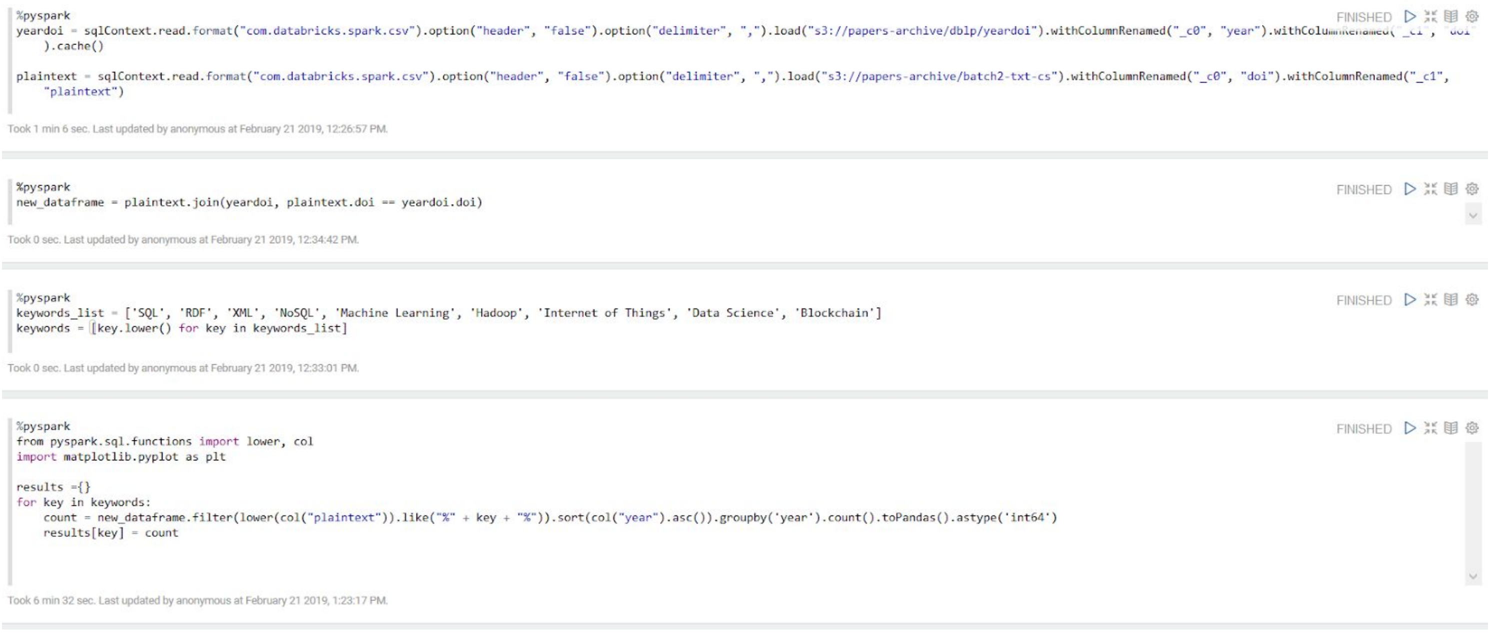
A. Both information on the article's publishing year and reference code, doi, and the text of the articles were loaded into two Spark DataFrames. Both data frames were then merged into one, called `new_dataframe`. For given `keywords`, which were transformed to lowercase words, a `count` dataframe filtered the text of the articles which contained the keywords, sorted the articles depending on their year of publication, and counted how many articles mentioned each keyword for a given year, while returning the year and the count per year. The results were stored as a dictionary named `results`, where the `keywords` are the key, and `count` is associated dataframe for that key.

The code was as follows:

```
%pyspark
yeardoi = sqlContext.read.format("com.databricks.spark.csv").option("header",
"false").option("delimiter",
",").load("s3://papers-archive/dblp/yeardoi").withColumnRenamed("_c0",
"year").withColumnRenamed("_c1", "doi").cache()
plaintext = sqlContext.read.format("com.databricks.spark.csv").option("header",
"false").option("delimiter",
",").load("s3://papers-archive/batch2-txt-cs").withColumnRenamed("_c0",
"doi").withColumnRenamed("_c1", "plaintext")
new_dataframe = plaintext.join(yeardoi, plaintext.doi == yeardoi.doi)
keywords_list = ['SQL', 'RDF', 'XML', 'NoSQL', 'Machine Learning', 'Hadoop',
'Internet of Things', 'Data Science', 'Blockchain']
keywords = [key.lower() for key in keywords_list]
from pyspark.sql.functions import lower, col
import matplotlib.pyplot as plt

results ={}
for key in keywords:
    count = new_dataframe.filter(lower(col("plaintext")).like("%" + key +
"%")).sort(col("year").asc()).groupby('year').count().toPandas().astype('int64')
    results[key] = count
```

On Zeppelin, the code was submitted as follows, taking a little over 7 minutes to run:



```
%pyspark
yeardoi = sqlContext.read.format("com.databricks.spark.csv").option("header", "false").option("delimiter", ",").load("s3://papers-archive/dblp/yeardoi").withColumnRenamed("_c0", "year").withColumnRenamed("_c1", "doi"
    ).cache()

plaintext = sqlContext.read.format("com.databricks.spark.csv").option("header", "false").option("delimiter", ",").load("s3://papers-archive/batch2-txt-cs").withColumnRenamed("_c0", "doi").withColumnRenamed("_c1",
    "plaintext")
```

Took 1 min 6 sec. Last updated by anonymous at February 21 2019, 12:26:57 PM.

```
%pyspark
new_dataframe = plaintext.join(yeardoi, plaintext.doi == yeardoi.doi)
```

Took 0 sec. Last updated by anonymous at February 21 2019, 12:34:42 PM.

```
%pyspark
keywords_list = ['SQL', 'RDF', 'XML', 'NoSQL', 'Machine Learning', 'Hadoop', 'Internet of Things', 'Data Science', 'Blockchain']
keywords = [key.lower() for key in keywords_list]
```

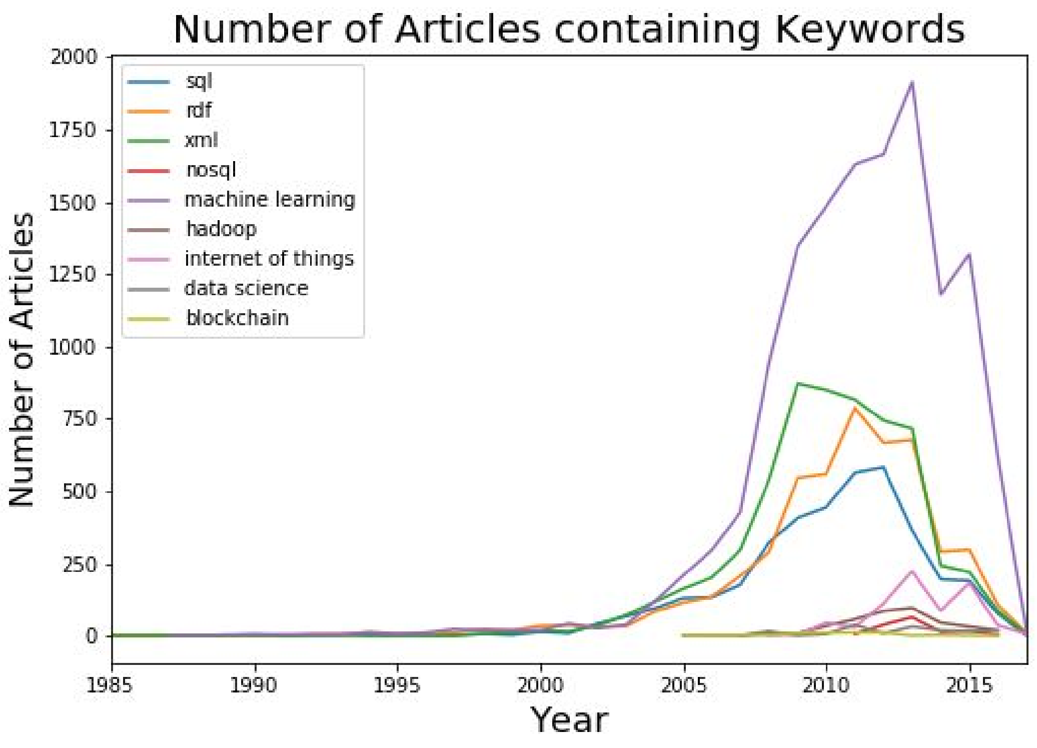
Took 0 sec. Last updated by anonymous at February 21 2019, 12:33:01 PM.

```
%pyspark
from pyspark.sql.functions import lower, col
import matplotlib.pyplot as plt

results ={}
for key in keywords:
    count = new_dataframe.filter(lower(col("plaintext")).like("%" + key + "%")).sort(col("year").asc()).groupby('year').count().toPandas().astype('int64')
    results[key] = count
```

Took 6 min 32 sec. Last updated by anonymous at February 21 2019, 1:23:17 PM.

This results dictionary was then plotted into a line plot, so that each line shows the number of articles which contained a keyword per year.



The code submitted was as follows:

```
%pyspark
for key in keywords:
    plt.plot(results[key]["year"],results[key]["count"], label = str(key))
    plt.legend()
plt.xlim(1985,2017)
plt.title("Number of Articles containing Keywords", fontsize=20)
plt.xlabel("Year", fontsize=18)
plt.ylabel("Number of Articles", fontsize=16)

plt.show()
```

Question Two

For question two the objective was to write a single SQL query that would output the first 100 papers that mentioned a particular keyword. To construct one query we first separated it in four parts for testing purposes and later we chained together to one ugly long query.

Part one

The first part selects papers that contains the tags in their plaintext. To do this both the plaintext and tags needs to be lower chased.The query returns the tag and the document id.

```
SELECT tag, doi FROM plaintext JOIN tags ON LOWER(plaintext.plaintext) LIKE
CONCAT('%', LOWER(tags.tag), '%')
```

Part two

In the second part of the query the year, authors and title are appended to the data.

```
SELECT PART_ONE.tag, DBLP.year, DBLP.authors, DBLP.title FROM DBLP INNER
JOIN PART_ONE ON DBLP.doi = PART_ONE.doi)
```

Part three

In this part the papers are selected grouped by tag and first year appearance is returned. The number of returned records is limited to 100, as we only interested in first 100 appearances.

```
SELECT MIN(year) AS year, tag FROM PART_TWO GROUP BY tag ORDER BY year ASC
LIMIT 100
```

Part four
Finally the filtered year and metadata are joined and this should output the results.

```
SELECT PART_TWO.tag, PART_TWO.year, authors, title FROM PART_TWO JOIN
PART_THREE ON PART_TWO.year = PART_THREE.year AND PART_THREE.tag =
PART_THREE.tag
```

Code Run with Spark
Below is the query chained and how it was inputted in Zeppelin. For saving the output we use coalesce(1) to output one single CSV file.

```
%pyspark
from pyspark.sql.functions import lit, col, lower

#load all files
tags = sqlContext.read.format("com.databricks.spark.csv").option("header",
"false").option("delimiter",
",").load("s3://bigdatacourse2019/dblp_tags.txt").withColumnRenamed("_c0",
"tag").cache()
yeardoi = sqlContext.read.format("com.databricks.spark.csv").option("header",
"false").option("delimiter",
",").load("s3://papers-archive/dblp/yeardoi").withColumnRenamed("_c0",
"year").withColumnRenamed("_c1", "doi")
plaintext = sqlContext.read.format("com.databricks.spark.csv").option("header",
"false").option("delimiter",
",").load("s3://papers-archive/batch2-txt-cs").withColumnRenamed("_c0",
"doi").withColumnRenamed("_c1", "plaintext")
DBLP = sqlContext.read.json("s3://bigdatacourse2019/dblp-2019-02-01").cache()

#register temp tables
tags.registerTempTable("temp_tags")
yeardoi.registerTempTable("temp_years")
plaintext.registerTempTable("temp_text")
DBLP.registerTempTable("temp_DBLP")



#super query
output = sqlContext.sql("SELECT filter_on_doi.tag, filter_on_doi.year, authors,
title FROM (SELECT filter_on_tag.tag, temp_DBLP.year, temp_DBLP.authors,
temp_DBLP.title FROM temp_DBLP INNER JOIN (SELECT tag, doi FROM temp_text JOIN
temp_tags ON LOWER(temp_text.plaintext) LIKE CONCAT('%', LOWER (temp_tags.tag),
'%')) filter_on_tag ON temp_DBLP.doi = filter_on_tag.doi) filter_on_doi JOIN
(SELECT MIN(year) AS year, tag FROM (SELECT filter_on_tag.tag, temp_DBLP.year,
temp_DBLP.authors, temp_DBLP.title FROM temp_DBLP INNER JOIN (SELECT tag, doi
FROM temp_text JOIN temp_tags ON LOWER(temp_text.plaintext) LIKE CONCAT('%',
LOWER(temp_tags.tag), '%')) filter_on_tag ON temp_DBLP.doi =
filter_on_tag.doi)filter_on_doi GROUP BY tag ORDER BY year ASC LIMIT 100)
filter_on_year ON filter_on_doi.year = filter_on_year.year AND filter_on_doi.tag
=filter_on_year.tag")

#save query result
output.toPandas().coalesce(1).to_csv("s3a://group19bucket/Lab_03/output3/output.
csv", mode="overwrite", header=True)
```

Problems with the query
Spark did not return an error while executing the query, so it can be assumed that the query is valid. Thus in theory this query should work, only problem is that it takes a zillion years to run. Therefore we can assume that is query lacks optimality, however we were unable to solve this issue before the assignment deadline. As well the saving of the query failed with the following error *`ImportError: The s3fs library is required to handle s3 files`*

Question Three

RDDs, Resilient Distributed Datasets, are Spark's earliest data abstraction framework, which consists of sets of datafiles or instructions which are distributed over several machines and which has built-in APIs to able access to and processing of the data. With an RDD, we can program how we want the data to be processed, or divided (via functional programming).

DataFrames is a data abstraction framework which can be composed by a group of RDDs, or other data types, organized in a tabular format of named columns and data types info. With a DataFrames, we cannot decide how the data will be processed, but only program what type of processing we would like to achieve, leaving the computational optimization in the hands of the API. They are Spark's smarter and more comprehensive version of Python's Pandas.

DataFrames perform such optimization via Custom Memory management, which stores the data efficiently in memory in binary format, and via Optimized Execution Plans (i.e. Catalyst), which will execute a command faster by only accessing the parts of the data necessary.

Thus, DataFrames are a level higher than RDDs. While RDDs can run faster than DataFrames, and they are the simple building blocks of Spark, DataFrames are make data easily readable thanks to the structured organization of data and are built for query optimization. Because of its simplicity, RDDs are faster for very simple data processing tasks, but loose in efficiency with task complexity involving several steps.

SparkSQL is Spark's interface. It executes SQL queries written in SQL syntax or HiveQL. It is thus at an advantage of a simpler more direct syntax for queries. It can read data from several formats. It can access both structured and unstructured data, and is portable in several languages. SparkSQL contains the same optimizer Catalyst as DataFrames, which RDDs lack, and which allow for time and memory efficient computations of tasks. It is at a disadvantage when it comes to real time processing as it has to first divide data into batches via RDDs, while other interfaces may work a little faster. Other disadvantages of SparkSQL is that is cannot create nor read a table containing union fields nor char fields.

References:

https://community.hortonworks.com/articles/42027/rdd-vs-dataframe-vs-sparksql.html
https://homepages.cwi.nl/~boncz/bigdatacourse/spark.shtml
https://www.quora.com/What-is-SparkSQL
https://stackoverflow.com/questions/31508083/difference-between-dataframe-dataset-and-rdd-in-spark

Error on exporting files

```
Traceback (most recent call last):
 File "/tmp/zeppelin_pyspark-4560870089834593051.py", line 367, in <module>
   raise Exception(traceback.format_exc())
Exception: Traceback (most recent call last):
 File "/tmp/zeppelin_pyspark-4560870089834593051.py", line 360, in <module>
   exec(code, _zcUserQueryNameSpace)
 File "<stdin>", line 1, in <module>
 File "/usr/local/lib64/python2.7/site-packages/pandas/core/generic.py", line 3019, in
to_csv
   escapechar=escapechar, decimal=decimal)
 File "/usr/local/lib64/python2.7/site-packages/pandas/io/formats/csvs.py", line 44, in
__init__
   path_or_buf, encoding=encoding, compression=compression, mode=mode
 File "/usr/local/lib64/python2.7/site-packages/pandas/io/common.py", line 212, in
get_filepath_or_buffer
   from pandas.io import s3
 File "/usr/local/lib64/python2.7/site-packages/pandas/io/s3.py", line 8, in <module>
   raise ImportError("The s3fs library is required to handle s3 files")
ImportError: The s3fs library is required to handle s3 files
```