Lab Assignment 3 **Big Data**
Due Date: 22/02/2019 Prof. P. Boncz
Jeroen Schmidt (122808560) & Dimitris Michailidis (12325929) TA: Dean De Leo
Group 10

Part 1 - Computer Science Research Trends Over the Years

The goal of this exercise is to use Spark to investigate the computer science research trends over the year. To achieve this we created an AWS cluster and used PySpark and Zeppelin to perform the operations. The below script counts the number of published papers in which each search term appears per year. It then plots it using matplotlib. Figure 1 shows the results over the years.

The script's process is the following:

- load yeardoi and plaintext from s3, store them in spark data frames.
- join the two dataframes on `doi` field.
- convert the `plaintext` field into lowercase.
- for each search term, create a list of dictionaries, in the following format: $[\{search\_term : [\{year : count\}, ..]\}, ..]$
- sort the dictionaries by year.
- plot the results using matplotlib.pyplot.

```
from pyspark.sql.functions import lower, col, sum
from matplotlib.pyplot import figure

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")

# Join yeardoi and plaintext into a DataFrame
df_papers = plaintext.join(yeardoi, how="inner", on="doi")
# Convert 'plaintext' to lower case
df_papers = df_papers.withColumn("plaintext",lower(col("plaintext")))

# Create the count dictionaries
search = ["sql", "rdf", "xml", "nosql", "machine learning", "hadoop",
"internet of things", "data science", "blockchain"]

counts = {}
for term in search:
    temp = df_papers.filter("plaintext like '% " + term + " %'")
        .groupBy('year').count()
    counts[term.replace(' ', '_')] = map(lambda row: row.asDict(), temp.collect())
```

```
# Sort dictionaries by year
sorted_by_year = {}
for key, value in counts.items():
    sorted_by_year[key] = sorted(value, key=lambda x: int(x['year']))

# Plot the results
figure(figsize=(20,10))
for key, value in sorted_by_year.items():
    plt.plot([data['year'] for data in value],
             [data['count'] for data in value],
             '-o',
             label=key)
    plt.legend()

plt.show()
```

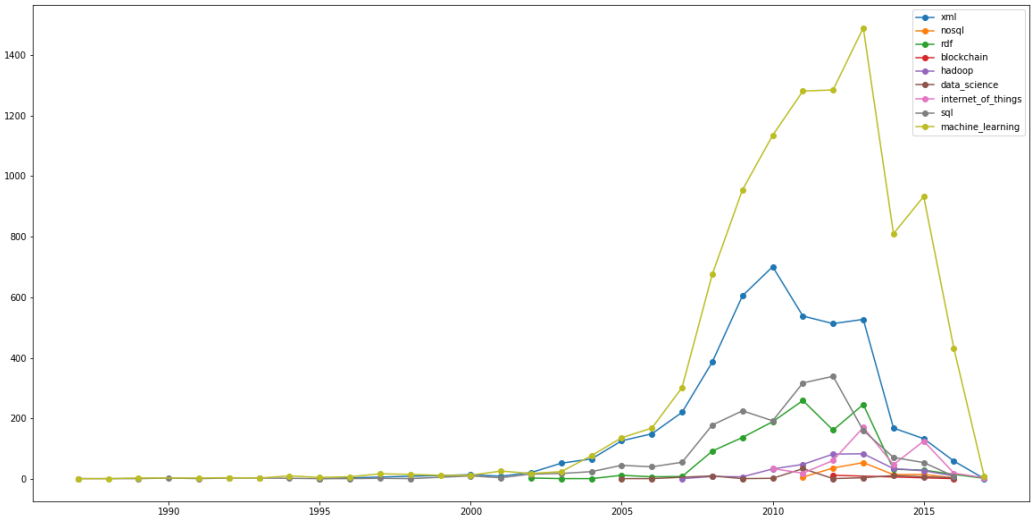

Figure 1: Research trends of computer science papers over the years

Part 2 - Fundamental Papers for each Term

The goal of this exercise is to use Spark to find the first occurrence of a term in the corpus of academic papers; we searched both the papers title and plaintext. The assignmnet placed an emphasis on performing this task within the spark framework and we were able to adhere to that.

The first occurances of each term are plotted in figure 2. The csv file can also be found within our S3 bucket. The solution can be found on the next page. An outline of our solution is as follows:

1. Join the data containing the paper title and year of publication with the plaintext data.
2. Concatenate the plaintext and title columns into the plaintext coloum. This was done so that we can search a single coloum for the occurrence of tags with out worrying about it happening in either column.
3. Load the tag data into spark
4. normalise the text such that everything is in lowercase
5. Collect the tag data and then broadcast it as a set, which will later be used
6. declare a UDF which will be used to compare the plaintext to all the tags in the broadcast variable. If there is a match, then the udf will output the tag, which will be added as a new value to the dataframes tag coloum
7. Perform a ranked window function over tags sorter by year. Once the ranks have been determined we can then take the top rank for each tag to determine the first instance the tag appeared.
8. Repartition the dataframe to 1 partition and then write it to s3 as a csv.

```
from pyspark.sql.functions import lower, udf, col , explode, rank, concat_ws
from pyspark.sql.types import StringType, ArrayType
from pyspark.sql.window import Window

#import paper information
from pyspark.sql.functions import lower, col, concat, lit

df = spark.read.json("s3://bigdatacourse2019/dblp-2019-02-01")
                    #.withColumnRenamed("title",lower(col("title")))

#import tags to aggrigate on
tags = spark.read.csv("s3://bigdatacourse2019/dblp_tags.txt")\
                    .withColumnRenamed("_c0","tag")\
                    .withColumn("tag",lower(col("tag"))).cache()

body = 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")

# join plain text to title dataframe. Also combine the title and
# plaintext into one coloum so that we can search both the title and plaintext by
```

```
# refering to one coloumn
df = df.join(body,how="inner",on="doi")
df = df.withColumn("plaintext",concat(col("title"), lit(" "), col("plaintext")))\
    .withColumn("plaintext",lower(col("plaintext")))



#collect the tags on a single machine and broadcast it to the cluster
tags_list = tags.select("tag").rdd.flatMap(lambda x: str.lower(x)).collect()
keywords = sc.broadcast(set(tags_list))

#define a UDF inorder to link tag to words in string
def check_keywork(row):
    output = []

    for x in keywords.value:
        if x in row:
            output.append(x)

    return output

    if len(output) == 0:
        return None

contains = udf(lambda row: check_keywork(row), ArrayType(StringType()))

#build dataframe with tags identified
has_tag = df.withColumn("tag",explode(contains(col("title")))).cache()


#windows over the tags and rank according to the year in asc order,
# take the top 1 to find the first occurance.
window = Window.\
              partitionBy('tag')\
              .orderBy(df['year'].asc())

first_tag = has_tag.withColumn("rank",rank().over(window))\
        .filter("rank == 1")

# transform the list object of authors into a string and then write to s3,
# it is repartitioned to 1 so that a single csv is written
first_tag\
.withColumn("authors",concat_ws(",","authors")))\
.select("tag","year","authors","title")\
.repartition(1)\
.write.csv('s3://bd-group10/lab3/q2.csv')
```

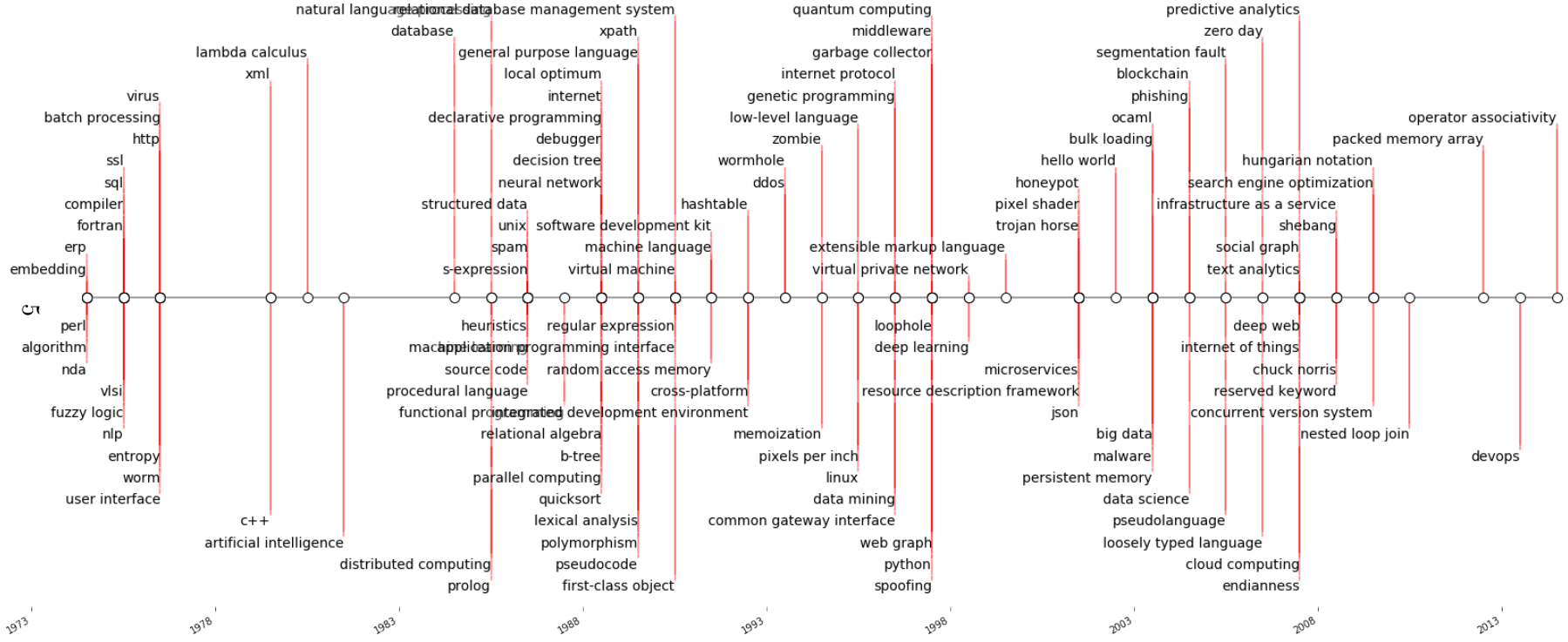

Figure 2: First Occurance of Each Key Word

Part 3 - Answering Questions about Spark and SparkSQL

In this section we explain the differences, advantages and disadvantages of RDDs, DataFrames and SparkSQL.

Resilient Distributed Datasets (RDDs)

RDDs are the building blocks behind Spark. The idea behind them is that, in certain big data operations, keeping data in memory can improve performance by an order of magnitude, compared to traditional computing frameworks(like MapReduce or Dryad)[1]. RDDs perform parallel processes on data in the distributed computers' memory, allowing for the reuse of intermediate results in a pipeline.

The functionality of RDDs(Figure 3) is achieved using three kinds of operations.

- *transformations*: operations on stable data storage or other RDDs. Examples include map, filter and join.
- *actions*: operations that return a value to the pipeline or save the results to the disk. Examples include count, collect and save.
- *persist*: operation to indicate which RDDs to be reused in future operations. Spark keeps those in memory or partially stores them to disks if there is no RAM available.

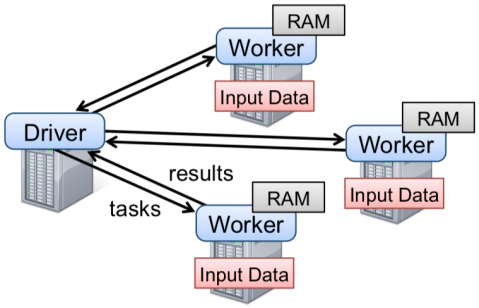

Figure 3: Spark runtime that enables the effective use of RDDs[1]

Advantages:

- They do not need to be materialized at all times. RDDs store all information on how they where derived from other datasets in a property called `lineage`.
- Provide fault tolerance without needing replication of data across the cluster. Through their lineage, RDDs can easily reconstruct themselves in case of a partition failure (note that only the failed partitions need reconstruction, not all of them).
- Slow nodes are replicated in MapReduce as backup tasks.
- Use the disks automatically when necessary. If there is not enough RAM for the RDDs to operate, they store those partitions to the disk instead.

Disadvantages:

- Not suitable for asynchronus operations. RDDs work well with batch operations that aim to do the same operation on all elements of a dataset. If an application makes multiple updates to a shared state, then RDDs are not suitable.

- Java Garbage Collection: RDDs are stored in memory as JVM objects, therefore they employ Java's garbage collection process, which can be slow.

Spark DataFrames

A Spark DataFrame is a high level abstraction framework that enables schema-based data manipulation in a similar context as relational databases. Essentially, a DataFrame is a distributed collection of rows with the same schema(RDD of row objects) and can be constructed from many structured sources(e.g. relational databases, other RDDs, Hive tables).

Advantages:

- Supports relational operators (select, where, groupby).
- Supports both primitive SQL types (integer, double, string, timestamp) and complex types (structs, arrays, maps).
- Provides a relational view of the data, enabling SQL manipulations.

Disadvantages:

- Not able to detect attribute errors. The compiler cannot detect if a column exists in a dataframe, it only catches it on runtime[1].
- When creating applications, domain objects cannot be created from the relational representations of the dataframe[1].
- It is slower than the low-level API that RDDs expose[2].

SparkSQL

SparkSQL is a module in Spark that enables the processing of structured data, usually stored in relational databases. Unlike the basic Spark RDD API, SparkSQL provide Spark with more information about the structure of the data and the computation being performed [3]. SparkSQL is just an API, meaning that during the execution of the processes, the same engine is used as in RDDs and DataFrames.

Queries built in SparkSQL are optimized by Catalyst. Catalyst employs advanced programming language features and behaves as an extensible query optimizer[4], making it easy to enhance the optimization techniques, as well as extend the them with new features.

Advantages[5]:

- Integrated in Spark. Enables quering structured data as an RDD.
- Loads and queries data from different datasources, via the creation of schema based RDDs.
- Highly scalable. As stated above, SparkSQL is an interface. At low level it takes advantage of the properties of RDDs and therefore its scalability.

Disadvantages[5]:

- Does not support the union type. Creation or reading of union fields is not available.
- No support for fixed length strings(char).

[1] https://intellipaat.com/blog/dataframes-rdds-apache-spark/
[2] https://community.hortonworks.com/articles/42027/rdd-vs-dataframe-vs-sparksql.html
[3] https://spark.apache.org/docs/2.3.0/sql-programming-guide.html
[4] https://databricks.com/glossary/catalyst-optimizer
[5] https://data-flair.training/blogs/spark-sql-tutorial/

References

[1] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J Franklin, Scott Shenker, and Ion Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation*, pages 2–2. USENIX Association, 2012.