university of groningen

faculty of mathematics and natural sciences

Research Internship

Scala Library for guarding assertions in Spark Pipelines

M. Lykiardopoulos
S3490890
m.p.lykiardopoulos@student.rug.nl

1 Project Description

The main scope of this project is to design and deploy a Scala Library for guarding assertions in Spark Pipelines. A Spark Pipeline is defined as a sequence of stages, and each stage is either a Transformer or an Estimator. A Transformer is an abstraction that includes feature transformers and learned models. Technically, a Transformer implements a method $transform()$, which converts one DataFrame into another, generally by appending one or more columns[2]. An Estimator abstracts the concept of a learning algorithm or any algorithm that fits or trains on data. Technically, an Estimator implements a method fit(), which accepts a DataFrame and produces a Model, which is a Transformer[2]. These stages are run in order and the input Dataframe is transformed as it passes through each stage.

The main idea of the Project is that the user using the aforementioned Library would be able to check if the assertions that he or she is interested about are valid or not. The library would take as input a Dataframe which comes from previous stages of Spark Pipelines, will examine if the assertions are valid or not. If they are valid it will return the Dataframe that took as input in order to continue to the next Stages. If the assertion is not valid then it will throw an exception, reporting the error.

2 Problem Solution

For our implementation of the library we used the Scala programming language because it is the most suitable with Spark. One of the features the Scala language provides us with is the concept of implicit classes. By using *implicit class* we make the class primary constructor available for implicit conversions when the class is in scope. An *implicit class* named $DataFrameassertions$ was defined inside an object named *implicit*. Inside the *class* we defined a number of building blocks and each of these checks for different kind of assertions. These assertions are:

1. At most n entries which satisfy the expression given by the user.
2. At least n entries which satisfy the expression given by the user.
3. The mean value of a Dataframe column belongs to a range of values that the User provides.
4. The standard deviation of a Dataframe column belongs to a range of values that the User provides.

5. The covariance between two Dataframe columns belongs to a range of values that the User provides.
6. The correlation between two Dataframe columns belongs to a range of values that the User provides.

A dataset containing 3229 records was used in order to check the building blocks that our Library contains and evaluate the results we get. The building blocks are the following:

1. `at_most`: The goal of this building block is to examine if there are at most n records in a Dataframe which satisfy an expression given by the user. This building block takes as inputs an Integer n and an expression which is of type String and returns a Dataframe. The Datframe is being filtered with the expression given by the user and counts the number of entries which satisfy the expression. The number of counts is stored in a variable named *count*. If we count more than n occurrences in the resulting Dataframe, we throw an exception, otherwise a success message is printed. The code of the implementation is shown in Listing 1:

```
def at_most( n : Int, expression : String) : DataFrame = {
  val count = df.filter(expression).count
   if(count > n) {
     throw new Exception(s"Assertion error found, count found")
   }
     println(s"Success, count found" )
     return df
}
```

Listing 1: Scala Method

2. `at_least`: The goal of this building block is to examine if there are at least n records in a Dataframe which satisffy an expression given by the user. This building block is almost similar with to previous one and the only thing that is chaneged is the *if* statement. The Datframe is being filtered with the expression given by the user and counts the number of entries which satisfy the expression. The number of counts is stored in a variable named *count*. If we count less than n occurrences in the resulting Dataframe, we throw an exception, otherwise a success message is printed. The code of the implementation is shown in Listing 2:

```
def at_least(n : Int, expression : String) : DataFrame = {
   val count = df.filter(expression).count
   if(count < n) {
     throw new Exception(s"Assertion error found, count found")
   }
     println(s"Success, count found" )
     return df
}
```

Listing 2: Scala Method

3. `mean_s`: The goal of this building block is to examine if the mean of a Dataframe column is included into a range of values that the user passes as input arguments in the building block. This method takes as inputs a column name: `col_name` which is of type String, and two Double values which are named *start* and *end*. The Dataframe column is selected and and we calculate the mean value of the selected column. After that we examine if the mean value belongs to the range of values that the user gave as inputs. If the mean value belongs to the range of values we print a success message and the Dataframe is returned. Otherwise an exception is thrown. The code of the implementation is shown in Listing 3:

```
def mean_s(col_name: String, start : Double, end : Double) : DataFrame = {
   val mvalue = df.select(mean(df(col_name))).head().getDouble(0)
   if (mvalue >= start && mvalue <= end) {
   println(mvalue)
   return df
   }
   throw new Exception(s"Assertion error found, $mvalue found")
 }
```

Listing 3: Scala Method

4. `stddev_s`: The goal of this building block is to examine if the standard deviation of a Dataframe column is included into a range of values that the user passes as input arguments in the building block. This method takes as inputs a column name: `col_name` which is of type String, and two Double values which are named *start* and *end*. The Dataframe column is selected and and we calculate the standard deviation value of the selected column. After that we examine if the standard deviation value belongs to the range of values that the user gave as inputs. If the value belongs to the range of values we print a success message and the Dataframe is returned. Otherwise an exception is thrown. The code of the implementation is shown in Listing 4:

```
  def stddev_s(col_name : String, start : Double, end : Double) : DataFrame =
     {
   val stddev_val = df.select(stddev(df(col_name))).head().getDouble(0)
   if (stddev_val >= start && stddev_val <= end){
     println(stddev_val)
     return df
   }
    throw new Exception(s"Assertion error found, $stddev_val found")
 }
```

Listing 4: Scala Method

5. `cov_s`: The goal of this building block is to examine if the covariance between two Dataframe columns is included into a range of values that the user passes as input arguments in the building block. This method takes as inputs two column names: `col_name1`, `col_name2` which are of type String, and two Double values which are named *start* and *end*. The Dataframe columns are selected and we calculate the covariance value of the selected columns. After that we examine if the covariance value belongs to the range of values that the user gave as inputs. If the value belongs to the range of values we print a success message and the Dataframe is returned. Otherwise an exception is thrown. The code of the implementation is shown in Listing 5:

```
  def cov_s(col_name1 : String, col_name2 : String, start : Double, end :
      Double) : DataFrame = {
    val cov_val = df.select(covar_pop(col_name1,col_name2)).head().getDouble(0)
    if (cov_val >= start && cov_val <= end){
      println(cov_val)
     return df
    }
     throw new Exception(s"Assertion error found, cov_val found")
  }
```

Listing 5: Scala Method

6. `corr_s`: The goal of this building block is to examine if the correlation between two Dataframe columns is included into a range of values that the user passes as input arguments in the building block. This method takes as inputs two column names: `col_name1`, `col_name2` which are of type String, and two Double values which are named *start* and *end*. The Dataframe columns are selected and and we calculate the correlation value of the selected columns. After that we examine if the correlation value belongs to the range of values that the user gave as inputs. If the value belongs to the range of values we print a success message and the Dataframe is returned. Otherwise an exception is thrown. The code of the implementation is shown in Listing 6:

```
    def corr_s(col_name1 : String, col_name2 : String, start : Double, end :
        Double) : DataFrame = {
    val cor_val = df.select(corr(col_name1,col_name2)).head().getDouble(0)
    if (cor_val >= start && cor_val <= end){
      println(cor_val)
      return df
    }
     throw new Exception(s"Assertion error found, cor_val found")
  }
```

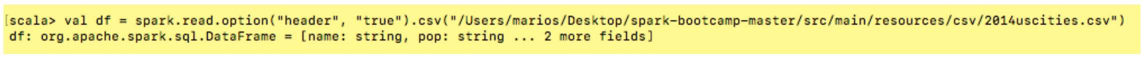
Listing 6: Scala Method

3 Starting Procedure

We evaluated our building blocks on a Dataframe with four Columns and 3229 records. We used a data set, containing data about United States cities. These data are: the name of the city, the population, the longitude and the latitude. In order to be able to work with the data, we first need to read the data into a Spark Dataframe, using the command of Listing 7. The Dataframe is stored in a value called *val df*.

```
val df = spark.read.option("header","true").csv("/path-to-csv-file/2014uscities.csv")
```

Listing 7: Scala Reading the Dataframe



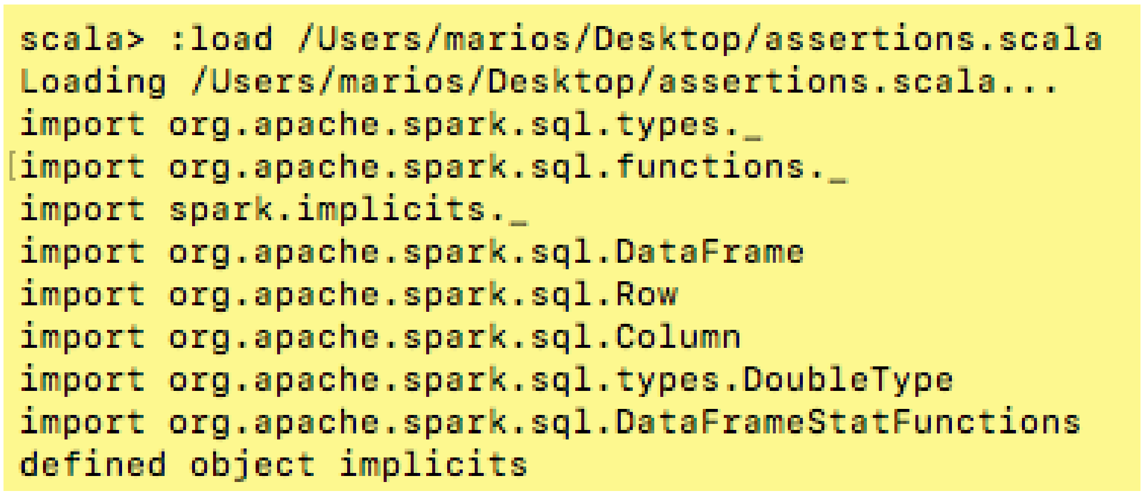
```
[scala> val df = spark.read.option("header", "true").csv("/Users/marios/Desktop/spark-bootcamp-master/src/main/resources/csv/2014uscities.csv")
df: org.apache.spark.sql.DataFrame = [name: string, pop: string ... 2 more fields]
```

Figure 1: Reading the Dataframe.

When the above step is finished, we have to load our Scala file into Spark using the command of Listing 8.

```
:load /path/to/scalacode/assertions.scala
```

Listing 8: Scala Loading our Scala code into Spark.



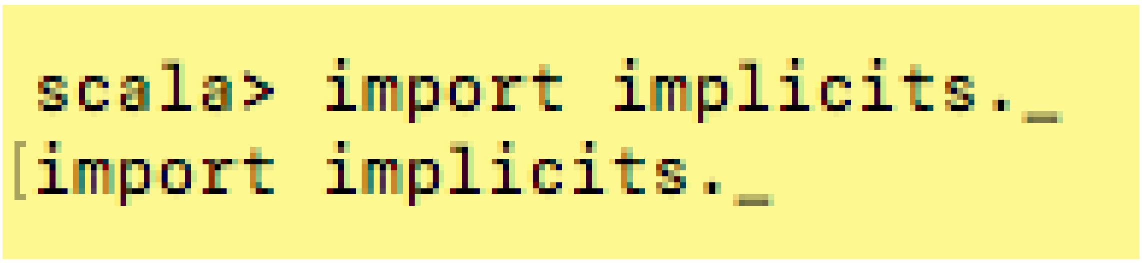
```
scala> :load /Users/marios/Desktop/assertions.scala
Loading /Users/marios/Desktop/assertions.scala...
import org.apache.spark.sql.types._
[import org.apache.spark.sql.functions._
import spark.implicits._
import org.apache.spark.sql.DataFrame
import org.apache.spark.sql.Row
import org.apache.spark.sql.Column
import org.apache.spark.sql.types.DoubleType
import org.apache.spark.sql.DataFrameStatFunctions
defined object implicits
```

Figure 2: Loading our Scala code into Spark.

The last step is to import the object named *implicits*, which was already defined by loading our scala code, using the command of Listing 9.

```
import implicits.
```

Listing 9: Scala Importing the defined object.

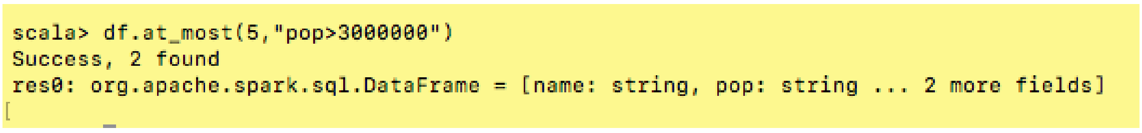


```
scala> import implicits._
[import implicits._
```

Figure 3: Importing the defined object.

4 Evaluation

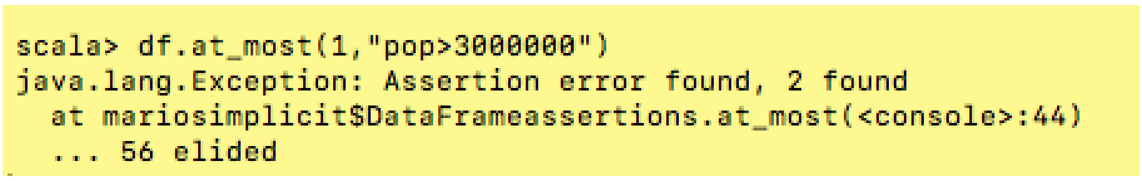
When the above steps are finished, we are ready to use our Scala code in order to check possible assertions in Spark Pipelines. In the following images we will look for possible assertions on our test Dataframe. First we try to examine if there are at most 5 cities that their population is greater than 3 million. As you can observe in the following image there are only 2 cities that have a population greater than 3 million.



```
scala> df.at_most(5,"pop>3000000")
Success, 2 found
res0: org.apache.spark.sql.DataFrame = [name: string, pop: string ... 2 more fields]
[
```

Figure 4: At most 5 cities with $pop > 3000000$.

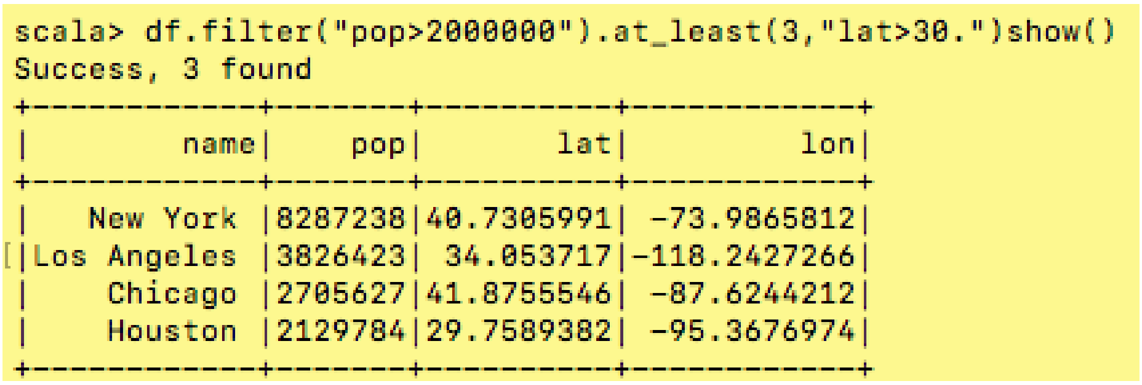
However, if we change the assertion query and search for at most 1 city which has population greater than 3 million we will get an Exception and the process will be terminated.



```
scala> df.at_most(1,"pop>3000000")
java.lang.Exception: Assertion error found, 2 found
  at mariosimplicit$DataFrameassertions.at_most(<console>:44)
  ... 56 elided
```

Figure 5: At most 1 city with $pop > 3000000$.

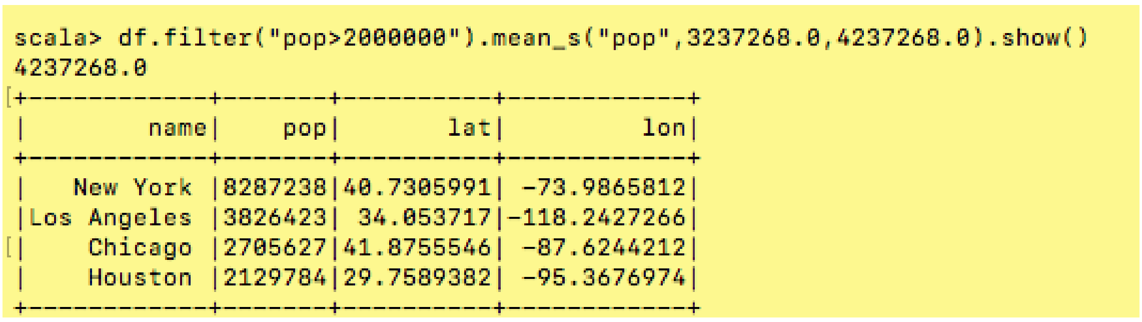
We will follow almost the same but a little more complex procedure in order to check the rest of our building blocks in our code. We will now first *filter* our Dataframe in order to find the cities that their population is greater than 2 million, and we will search in the filtered Dataframe to check if there are *at least* 3 cities which their latitude is greater than 30. Finally we will print the Dataframe in order to be sure that we get the appropriate results as you can see in the image below.



```
scala> df.filter("pop>2000000").at_least(3,"lat>30.")show()
Success, 3 found
+------------+-------+----------+------------+
|        name|    pop|       lat|         lon|
+------------+-------+----------+------------+
|   New York |8287238|40.7305991| -73.9865812|
[|Los Angeles |3826423| 34.053717|-118.2427266|
|    Chicago |2705627|41.8755546| -87.6244212|
|    Houston |2129784|29.7589382| -95.3676974|
+------------+-------+----------+------------+
```

Figure 6: Checking if the filtered Dataframe has at least 3 cities with $lat > 30$.

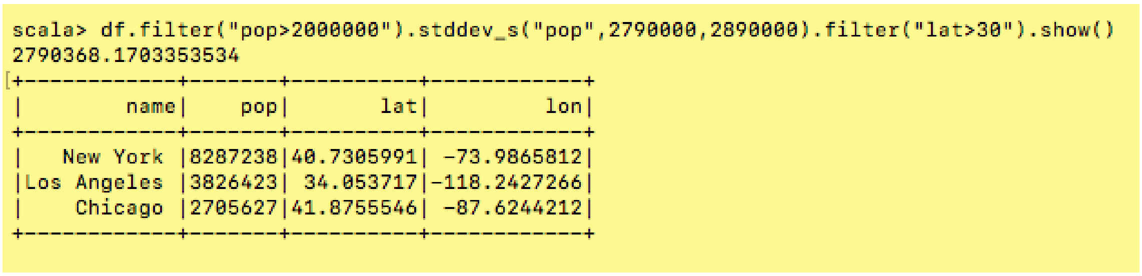
The next building block is the `mean_s` which examines if the mean value of a Dataframe column belongs to a range of values that the user passes into the method as input arguments. As it can be seen in the following image the method takes as input a Column name and two *Double* values. It is also examines if the assertion is true on the Dataframe that comes from the previous method which in our case is the *filter* method.



```
scala> df.filter("pop>2000000").mean_s("pop",3237268.0,4237268.0).show()
4237268.0
[+------------+-------+----------+------------+
|        name|    pop|       lat|         lon|
+------------+-------+----------+------------+
|   New York |8287238|40.7305991| -73.9865812|
|Los Angeles |3826423| 34.053717|-118.2427266|
[|    Chicago |2705627|41.8755546| -87.6244212|
|    Houston |2129784|29.7589382| -95.3676974|
+------------+-------+----------+------------+
```

Figure 7: Checking if the Column "pop" of the filtered Dataframe has a mean value between 3237268.0 and 4237268.0.

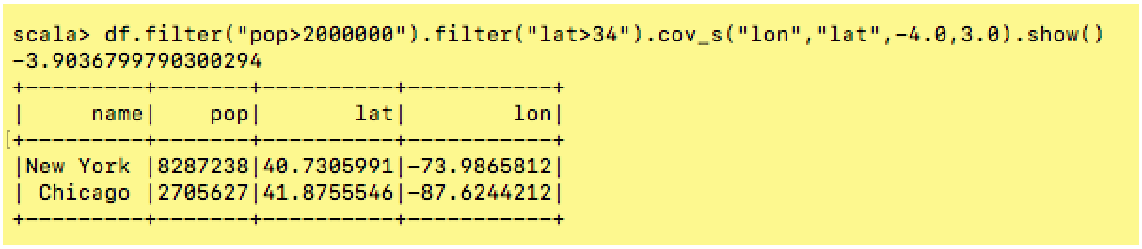
The following building block named `stddev_s` calculates the standard deviation of a Dataframe column and examines if the standard deviation value belongs to a range of values that the user gives as an input. First the Dataframe is being filtered, then we use the `stddev_s` method in order to guard our assertion and finally we filter again the Dataframe with another condition and show it.



```
scala> df.filter("pop>2000000").stddev_s("pop",2790000,2890000).filter("lat>30").show()
2790368.1703353534
[+------------+-------+----------+------------+
|        name|    pop|       lat|         lon|
+------------+-------+----------+------------+
|   New York |8287238|40.7305991| -73.9865812|
|Los Angeles |3826423| 34.053717|-118.2427266|
|    Chicago |2705627|41.8755546| -87.6244212|
+------------+-------+----------+------------+
```

Figure 8: Checking if the Column "pop" of the filtered Dataframe has a standard deviation value between 2790000 and 2890000.

The following building block of our code named `cov_s` calculates the covariance between two Dataframe columns in a Dataframe and examines if the covariance value belongs to a range of values that the user gives as an input. As it can be seen in the image below we first filter the Dataframe in order to take only the cities with $pop > 2000000$ and $lat > 34$ and then we examine if the covariance between the Dataframe columns lon and lat belongs to the range of values that we gave as inputs. Results can seen in the image below.



```
scala> df.filter("pop>2000000").filter("lat>34").cov_s("lon","lat",-4.0,3.0).show()
-3.9036799790300294
+---------+-------+----------+-----------+
|     name|    pop|       lat|        lon|
[+---------+-------+----------+-----------+
|New York |8287238|40.7305991|-73.9865812|
| Chicago |2705627|41.8755546|-87.6244212|
+---------+-------+----------+-----------+
```

Figure 9: Checking if the Columns "lon" and "lat" of the filtered Dataframe has a covariance value between -4.0 and 3.0.

The last building block of our code, named `corr_s` calculates the correlation between two Dataframe columns in a Dataframe and examines if the correlation value belongs to a range of values that the user gives as an input. We filter the Dataframe to find the cities that their population is greater than 2 million($pop > 2000000$) and then we examine if the correlation value of the Dataframe columns lon and lat belongs to the range of values that the user gives as inputs. Finaly we print the Dataframe. The results can seen in the image below.

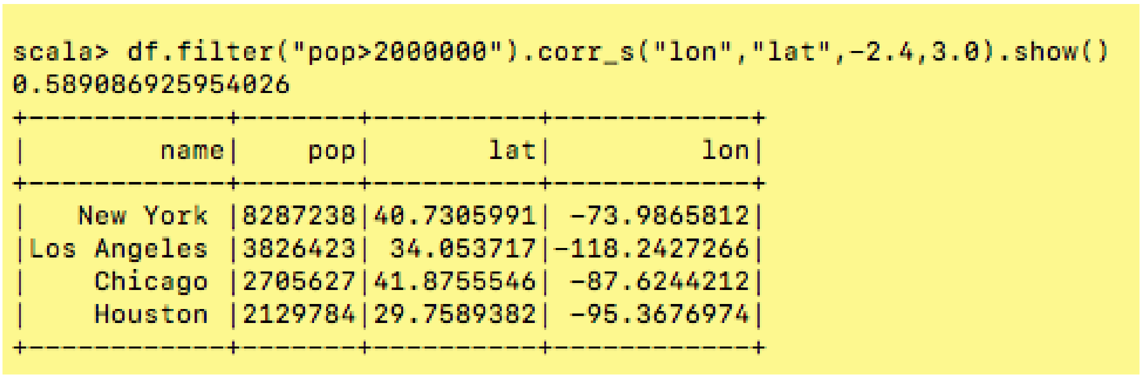

```
scala> df.filter("pop>2000000").corr_s("lon","lat",-2.4,3.0).show()
0.589086925954026
+------------+-------+----------+------------+
|        name|    pop|       lat|         lon|
+------------+-------+----------+------------+
|    New York|8287238|40.7305991| -73.9865812|
|Los Angeles |3826423| 34.053717|-118.2427266|
|     Chicago|2705627|41.8755546| -87.6244212|
|     Houston|2129784|29.7589382| -95.3676974|
+------------+-------+----------+------------+
```

Figure 10: Checking if the Columns ”lon” and ”lat” of the filtered Dataframe has a correlation value between -2.4 and 3.0.

4.1 Spark Cluster Evaluation

So far, we evaluated our results running Spark Standalone, using spark-shell. The next step was to evaluate that our code works properly also in a cluster. We deployed a local cluster of Spark with four workers and one master on our development machine, which is a MacBook Air late 2017 a core i5 processor at 1.8Ghz and 8 cores.

4.1.1 Setting up the Cluster

In order to set up the cluster we configured Spark as shown in Listing 10. With this configuration four workers are started, each having access to two cores and 2Gb of memory. Listing 10 shows the added parameters in the $spark - env.sh$ file which is inside the $conf$ directory of Spark.

```
SPARK_WORKER_CORES=2
SPARK_WORKER_INSTANCES=4
SPARK_WORKER_MEMORY=2g
```

Listing 10: Spark Cluster Parameters

4.1.2 Starting the Cluster

When these parameters are changed the next step is to start the cluster and communicate with the master, using the spark-shell. The commands we used can seen in Listings 11 and 12.

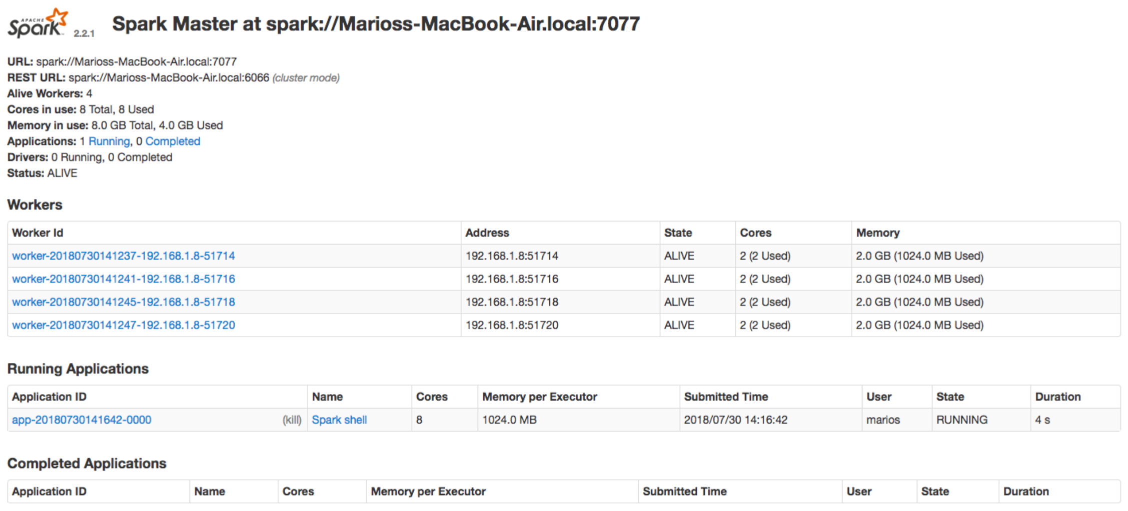
```
./sbin/start-all.sh
```

Listing 11: Spark Cluster starting the workers and the master

```
./bin/spark-shell --master spark://"Master IP Adress"
```

Listing 12: Spark Cluster connecting the Spark-shell with the master

The following image shows the Spark Cluster Web User Interface.



Spark Master at spark://Marioss-MacBook-Air.local:7077

URL: spark://Marioss-MacBook-Air.local:7077
REST URL: spark://Marioss-MacBook-Air.local:6066 *(cluster mode)*
Alive Workers: 4
Cores in use: 8 Total, 8 Used
Memory in use: 8.0 GB Total, 4.0 GB Used
Applications: 1 Running, 0 Completed
Drivers: 0 Running, 0 Completed
Status: ALIVE

Workers

Worker Id	Address	State	Cores	Memory
worker-20180730141237-192.168.1.8-51714	192.168.1.8:51714	ALIVE	2 (2 Used)	2.0 GB (1024.0 MB Used)
worker-20180730141241-192.168.1.8-51716	192.168.1.8:51716	ALIVE	2 (2 Used)	2.0 GB (1024.0 MB Used)
worker-20180730141245-192.168.1.8-51718	192.168.1.8:51718	ALIVE	2 (2 Used)	2.0 GB (1024.0 MB Used)
worker-20180730141247-192.168.1.8-51720	192.168.1.8:51720	ALIVE	2 (2 Used)	2.0 GB (1024.0 MB Used)

Running Applications

Application ID	Name	Cores	Memory per Executor	Submitted Time	User	State	Duration
app-20180730141642-0000 (kill)	Spark shell	8	1024.0 MB	2018/07/30 14:16:42	marios	RUNNING	4 s

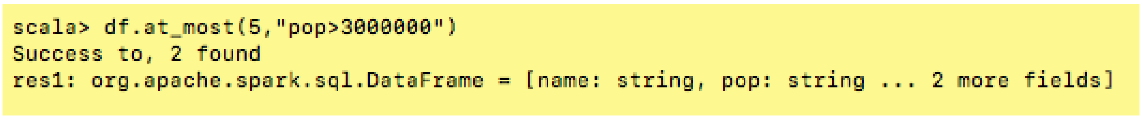
Completed Applications

Application ID	Name	Cores	Memory per Executor	Submitted Time	User	State	Duration

Figure 11: Spark Cluster Web User Interface.

4.1.3 Evaluation on the Cluster

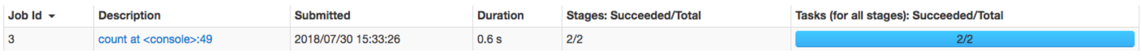
In order to be sure that our assertions library functions well when running on Spark in Cluster mode, we ran the same evaluation as we did for the Standalone case.



```
scala> df.at_most(5,"pop>3000000")
Success to, 2 found
res1: org.apache.spark.sql.DataFrame = [name: string, pop: string ... 2 more fields]
```

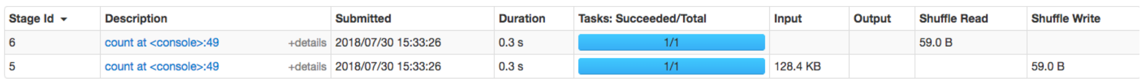
Figure 12: Spark Cluster at most 5 cities with $pop > 3000000$.

On the images below also we present the Stages for this specific job from the Spark UI. As it can be seen from the times the stages for the Job run in parallel.



Job Id ▾	Description	Submitted	Duration	Stages: Succeeded/Total	Tasks (for all stages): Succeeded/Total
3	count at <console>:49	2018/07/30 15:33:26	0.6 s	2/2	2/2

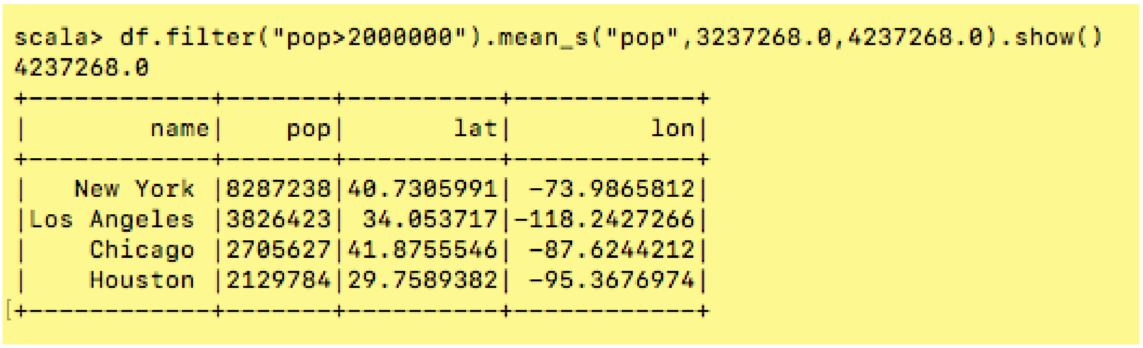
Figure 13: Spark Cluster at most 5 cities with $pop > 3000000$.



Stage Id ▾	Description		Submitted	Duration	Tasks: Succeeded/Total	Input	Output	Shuffle Read	Shuffle Write
6	count at <console>:49	+details	2018/07/30 15:33:26	0.3 s	1/1			59.0 B	
5	count at <console>:49	+details	2018/07/30 15:33:26	0.3 s	1/1	128.4 KB			59.0 B

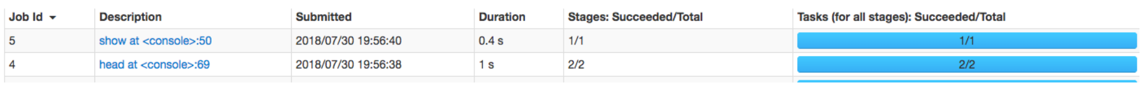
Figure 14: Spark Cluster at most 5 cities with *pop* $>$ 3000000.

The next building block that we will test is the `mean_s`, using the same sequence of orders as we did in Section 4. Results can be shown in the images below.



```
scala> df.filter("pop>2000000").mean_s("pop",3237268.0,4237268.0).show()
4237268.0
+------------+-------+----------+------------+
|        name|    pop|       lat|         lon|
+------------+-------+----------+------------+
|   New York |8287238|40.7305991| -73.9865812|
|Los Angeles |3826423| 34.053717|-118.2427266|
|    Chicago |2705627|41.8755546| -87.6244212|
|    Houston |2129784|29.7589382| -95.3676974|
+------------+-------+----------+------------+
```

Figure 15: Spark Cluster mean value between 3237268.0 and 4237268.0.



Job Id ▾	Description	Submitted	Duration	Stages: Succeeded/Total	Tasks (for all stages): Succeeded/Total
5	show at <console>:50	2018/07/30 19:56:40	0.4 s	1/1	1/1
4	head at <console>:69	2018/07/30 19:56:38	1 s	2/2	2/2

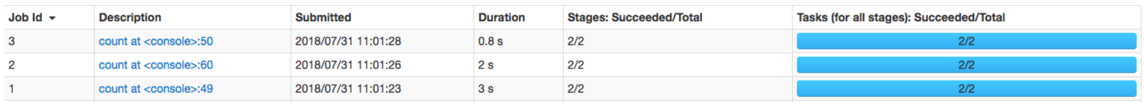
Figure 16: Spark Cluster mean value between 3237268.0 and 4237268.0.

5 Parallelization Issue

One of the problems that came up during the deployment of the Project was that when we run a sequence of orders the Stages for all jobs are not fully parallelized. This behavior has as a result that the Stages for the previous command should be finished in order to start the Stages for the following one. For example if we give a sequence of orders like the following one on Listing 13 we can easily examine that each of the Stages in the Spark Job run in a sequence and not in parallel.

```
    df.at_most(65,"pop>300000").at_least(50,"pop>300000").count
```

Listing 13: Spark Cluster parallelization issue



Job Id ▾	Description	Submitted	Duration	Stages: Succeeded/Total	Tasks (for all stages): Succeeded/Total
3	count at <console>:50	2018/07/31 11:01:28	0.8 s	2/2	2/2
2	count at <console>:60	2018/07/31 11:01:26	2 s	2/2	2/2
1	count at <console>:49	2018/07/31 11:01:23	3 s	2/2	2/2

Figure 17: Spark Cluster Jobs times.

In order to solve this problem we tried two approaches that will be discussed in the following subsections. The first one was to change the Job Scheduling parameter in Spark from FIFO to FAIR. The second was to try to use Spark Streaming in order to reduce the batch processing time.

5.1 Changing Job Scheduling

By default, Spark's scheduler runs jobs in FIFO fashion. Each job is divided into "stages" (e.g. map and reduce phases), and the first job gets priority on all available resources while its stages have tasks to launch, then the second job gets priority, etc. If the jobs at the head of the queue don't need to use the whole cluster, later jobs can start to run right away, but if the jobs at the head of the queue are large, then later jobs may be delayed significantly. In order to change the Spark Job Scheduling from FIFO to FAIR we have to set up a new Spark Context and define the Spark Scheduler Mode to FAIR. A Spark Context is the main entry point for Spark functionality. A Spark Context represents the connection to a Spark cluster, and can be used to create RDDs, accumulators and broadcast variables on that cluster. Also we have to create a new Spark Session, which is the entry point to programming Spark with the Dataset and DataFrame API. The commands to change the Spark Scheduler mode and creating a new Spark Session can be found in Listings 14 and 15.

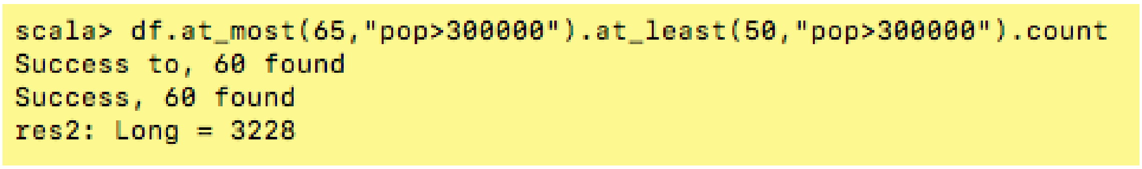
```
import org.apache.spark.SparkConf
import org.apache.spark.SparkContext
val conf = new SparkConf().setMaster("Master URL").setAppName("app-name")
conf.set("spark.scheduler.mode", "FAIR")
val sc = new SparkContext(conf)
```

Listing 14: Spark Cluster changing Scheduler Mode

```
import org.apache.spark.sql.SparkSession
val spark1=SparkSession
.builder.master("Master URL")
.appName("app-name")
.config("spark.sql.warehouse.dir", "./spark-warehouse")
.getOrCreate()
```

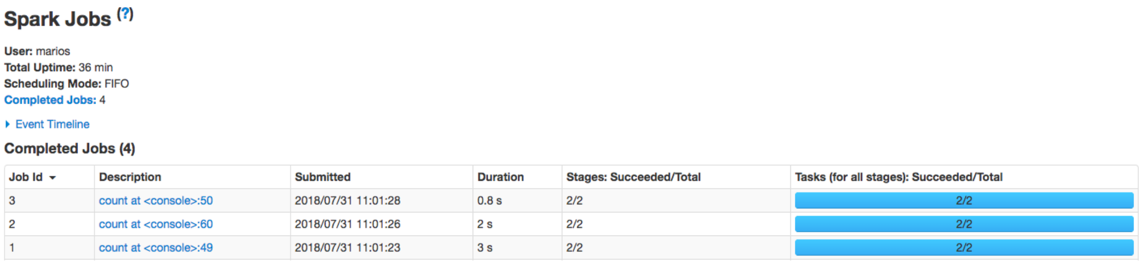
Listing 15: Spark Cluster creating a new Spark Session

When the above step is finished we are ready to use our code in the Spark Cluster with the Scheduler mode set up to FAIR. In order to check if the parallelization issue is solved we used the same sequence of orders as we did in the beginning of this Section. The results can be found in the following images.



```
scala> df.at_most(65,"pop>300000").at_least(50,"pop>300000").count
Success to, 60 found
Success, 60 found
res2: Long = 3228
```

Figure 18: Spark Cluster Jobs results.



Spark Jobs (?)

User: marios
Total Uptime: 36 min
Scheduling Mode: FIFO
Completed Jobs: 4

Event Timeline

Completed Jobs (4)

Job Id	Description	Submitted	Duration	Stages: Succeeded/Total	Tasks (for all stages): Succeeded/Total
3	count at <console>:50	2018/07/31 11:01:28	0.8 s	2/2	2/2
2	count at <console>:60	2018/07/31 11:01:26	2 s	2/2	2/2
1	count at <console>:49	2018/07/31 11:01:23	3 s	2/2	2/2

Figure 19: Spark Cluster Jobs times with scheduler mode set to FAIR.

As it can be examined from the above images the Jobs are not parallelized and we lead to the conclusion that there is a sequential dependency between the building blocks, so each of them should be finished and return the Dataframe that checked for possible assertions and then proceed to the next method. Furthermore Spark works with Pipelines so operations can not be fully parallelized.

5.2 Using Spark Streaming

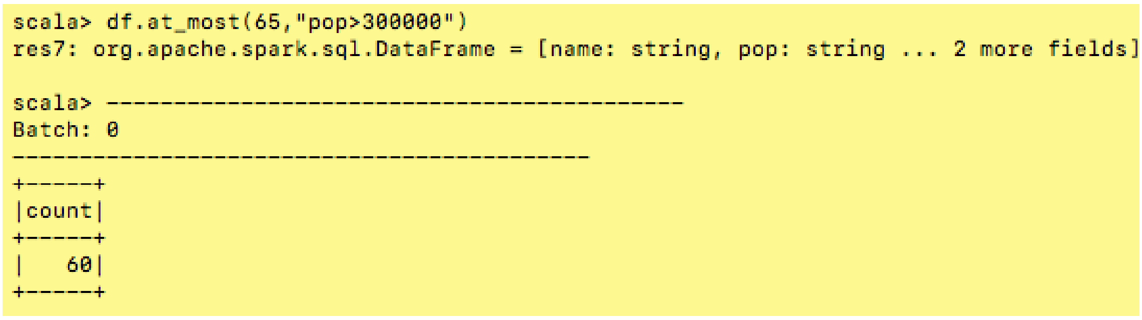
Our second approach in order to solve the parallelization problem was to use Spark Streaming in order to reduce the batch processing time. Spark Streaming is an extension of the core Spark API that enables scalable, high-throughput, fault-tolerant stream processing of live data streams. Data can be ingested from many sources like Kafka, Flume, Kinesis, or TCP sockets, and can be processed using complex algorithms expressed with high-level functions like map, reduce, join and window. Finally, processed data can be pushed out to filesystems, databases, and live dashboards[1]. As we examined the above solution we find out that there are some serious limitations with this approach. In addition, there are some Dataset methods that will not work on streaming Datasets. They are actions that will immediately run queries and return results, which does not make sense on a streaming Dataset[1].

These limitations cause some serious problems with our code, because we were able only to apply some transformations on the Streaming Dataframe, but not calculate and extract the values that our building blocks need in order to check for assertions on a Streaming Dataframe. When we tried any of these operations like *count*, *mean*, etc we get an analysis exception error that operation "X" is not a member of Streaming query. One possible solution to overcome this problem might be to transform the Streaming Data and then store them and guard for assertions. In Listing 16 you can find the modified code for the `at_most` bulding block and in the images below the results we get and the errors for the Streaming.

```
    import org.apache.spark.sql.types._
import org.apache.spark.sql.functions._
import spark.implicits._
import org.apache.spark.sql.DataFrame
import org.apache.spark.sql.Row
import org.apache.spark.sql.Column
import org.apache.spark.sql.types.DoubleType
import org.apache.spark.sql.DataFrameStatFunctions
import java.sql.Timestamp
import org.apache.spark.streaming.{Time, Seconds, StreamingContext}
import org.apache.spark.sql.streaming.StreamingQuery
import org.apache.spark.sql.streaming.OutputMode
import org.apache.spark.sql.streaming.Trigger
import org.apache.spark.Accumulator

object implicits{
  implicit class DataFrameassertions(df : DataFrame){
 def at_most( n : Int, expression : String) : DataFrame = {
   val df=spark.readStream.option("header", "true").schema(mschema).csv("path-to-
       csv-folder")
   val stream = df.filter(expression).groupBy().count().writeStream.outputMode("
       complete").format("console").start()
```

Listing 16: Spark Streaming

On the following image you can see that we get the total number of counts in a new Dataframe



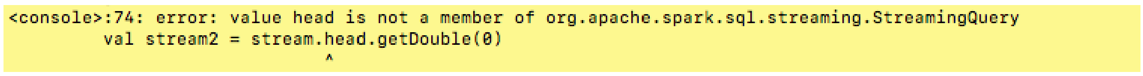
```
scala> df.at_most(65,"pop>300000")
res7: org.apache.spark.sql.DataFrame = [name: string, pop: string ... 2 more fields]

scala> -------------------------------------------
Batch: 0
-------------------------------------------
+-----+
|count|
+-----+
|   60|
+-----+
```

Figure 20: Spark Streaming total counts.

After that we tried to extract the value and store it to a new variable in order to compare it with the value given by the user. The code we added in order to make that and the error we get can be found on the Listing 17 and the Figure 21.

```
val stream2 = stream.head.getDouble(0)
```

Listing 17: Spark Streaming



```
<console>:74: error: value head is not a member of org.apache.spark.sql.streaming.StreamingQuery
         val stream2 = stream.head.getDouble(0)
                              ^
```

Figure 21: Spark Streaming total counts.

The same behavior is also observed with the rest of our building blocks. We end up to the conclusion that these necessary operations in order to guard assertions in Spark Streaming are not supported yet.

6 Discussion

The goal of the project was the deployment of a Scala Library for guarding assertions in Spark Pipelines. The significance of the Project is that with the code described above we are able to guard assertions in every possible Dataframe, by passing the appropriate input arguments into our Scala methods. By this procedure we expect to help Data Scientists examine possible assertions in Spark Pipelines and extract knowledge about their point of interest. Of course the aforementioned Research project is only the begining and a lot of work can be done in order to expand it. Possible future work can be done in order to add more building blocks in our code and based on these building blocks deploy a Domain Specific Language(DSL) that would be responsible for guarding assertions. Also the aforementioned parallelization problem can be examined in more details in the future.

References

[1] *Spark Documentation: Structured Streaming.* URL: https://spark.apache.org/docs/2.2.0/structured-streaming-programming-guide.html#operations-on-streaming-dataframesdatasets.

[2] *Spark Documentation: Transformers and Estimators.* URL: https://spark.apache.org/docs/2.2.0/ml-pipeline.html.