Spark Project - 3

The Spark Dataframes

In this project, we will try to create and work with Spark's Dataframe objects.

Spark is well known for its ability to process large data sets. Spark DataFrames combine the scale and speed of Spark with the familiar query, filter, and analysis capabilities of Python's pandas. Unlike pandas, which can only run on one computer, Spark can use distributed memory (and disk when necessary) to handle larger data sets and run computations more quickly.

Spark DataFrames allow us to modify and reuse our existing pandas code to scale up to much larger data sets. They also have better support for various data formats. We can even use a SQL interface to write distributed SQL queries that query large database systems and other data stores.

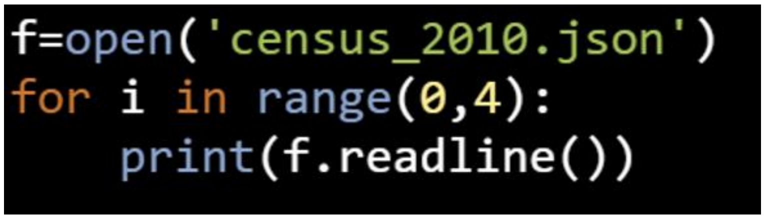
About the dataset:

The JSON file census_2010.json contains the data from the 2010 U.S. Census. It has been taken from the Machine Learning Library of UC Irvine:
https://archive.ics.uci.edu/ml/datasets/Census+Income

It has the following columns:

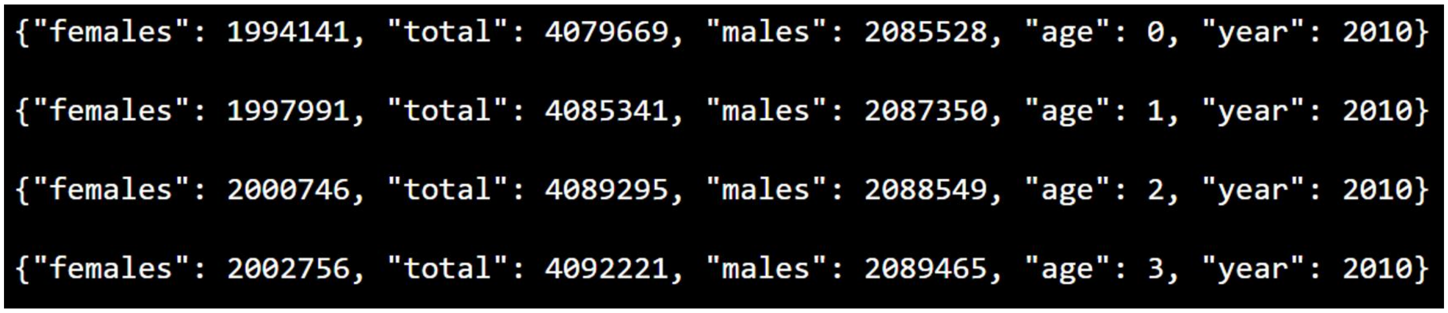
- age - Age (year)
- females - Number of females
- males - Number of males
- total - Total number of individuals
- year - Year column (2010 for all rows)

1. We will first just check the first few lines of the json file.



```python
f=open('census_2010.json')
for i in range(0,4):
    print(f.readline())
```

The output that we will be in JSON format of Key: Value pairs in documents:



```
{"females": 1994141, "total": 4079669, "males": 2085528, "age": 0, "year": 2010}

{"females": 1997991, "total": 4085341, "males": 2087350, "age": 1, "year": 2010}

{"females": 2000746, "total": 4089295, "males": 2088549, "age": 2, "year": 2010}

{"females": 2002756, "total": 4092221, "males": 2089465, "age": 3, "year": 2010}
```

2. RDD is essentially a list of tuples with no enforced schema or structure of any kind. An RDD can have a variable number of elements in each tuple, and combinations of types between tuples.

This makes RDDs very useful for representing unstructured data like text. Without them, we'd need to write a lot of custom Python code to interact with such data.

To use the familar DataFrame query interface from pandas, however, the data representation needs to include rows, columns, and types. Spark's implementation of DataFrames mirrors the pandas implementation, with logic for rows and columns.

The Spark SQL class is very powerful. It gives Spark more information about the data structure we're using and the computations we want to perform. Spark uses that information to optimize processes.

To take advantage of these features, we'll have to use the SQLContext object to structure external data as a DataFrame, instead of the SparkContext object.

We can query Spark DataFrame objects with SQL, which we'll explore in the next mission. The SQLContext class gets its name from this capability.

This class allows us to read in data and create new DataFrames from a wide range of sources. It can do this because it takes advantage of Spark's powerful Data Sources API.

File Formats it can accept:

- JSON, CSV/TSV, XML
- Parquet, Amazon S3 (cloud storage service)

Big Data Systems where it is compatible to:

- Hive, Avro, HBase

SQL Database Systems where it is compatible:

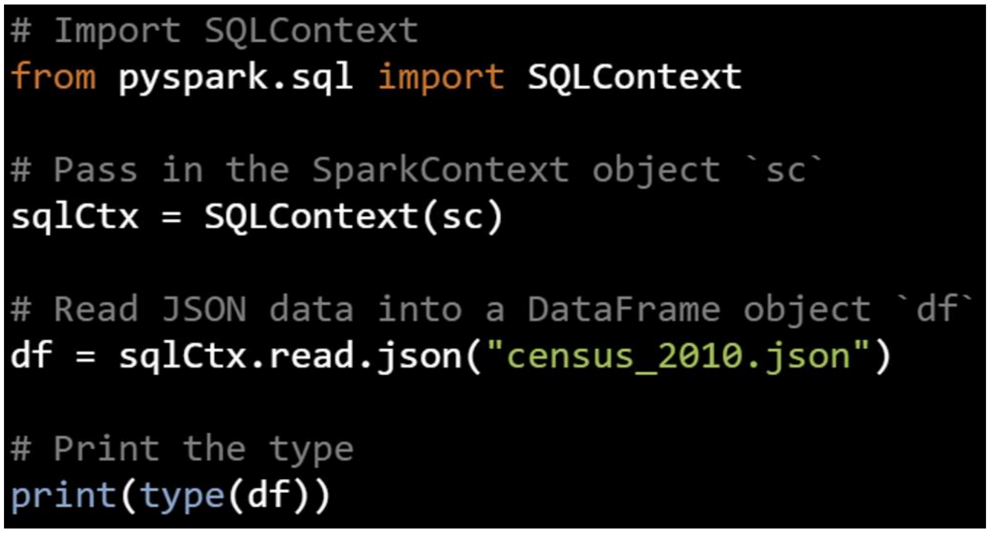
- MySQL, PostgreSQL

Data science organizations often use a wide range of systems to collect and store data, and they're constantly making changes to those systems. Spark DataFrames allow us to interface with different types of data, and ensure that our analysis logic will still work as the data storage mechanisms change.

After learning about Spark DataFrames, we will read in census_2010.json. This data set contains valid JSON on each line, which is what Spark needs in order to read the data in properly.

We will:

- Import SQLContext from the pyspark.sql library
- Instantiate the SQLContext object (which requires the SparkContext object (sc) as a parameter), and assign it to the variable sqlCtx
- Use the SQLContext method read.json() to read the JSON data set into a Spark DataFrame object named df
- Print df's data type to confirm that we successfully read it in as a Spark DataFrame



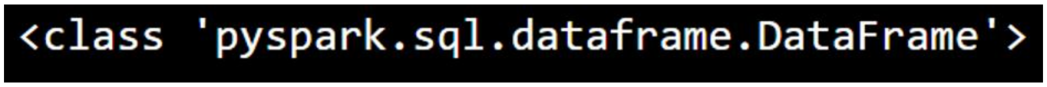
```python
# Import SQLContext
from pyspark.sql import SQLContext

# Pass in the SparkContext object `sc`
sqlCtx = SQLContext(sc)

# Read JSON data into a DataFrame object `df`
df = sqlCtx.read.json("census_2010.json")

# Print the type
print(type(df))
```

The output will show details of the resulting RDD:



```
<class 'pyspark.sql.dataframe.DataFrame'>
```

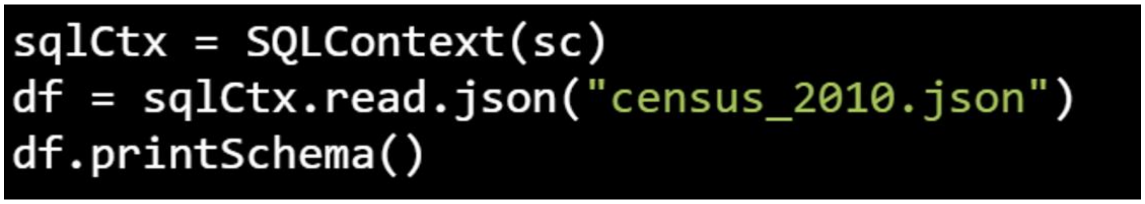
3. When we read data into the SQLContext object, Spark:

- Instantiates a Spark DataFrame object
- Infers the schema from the data and associates it with the DataFrame
- Reads in the data and distributes it across clusters (if multiple clusters are available)
- Returns the DataFrame object

We expect the DataFrame Spark created to have the following columns, which were the keys in the JSON data set:

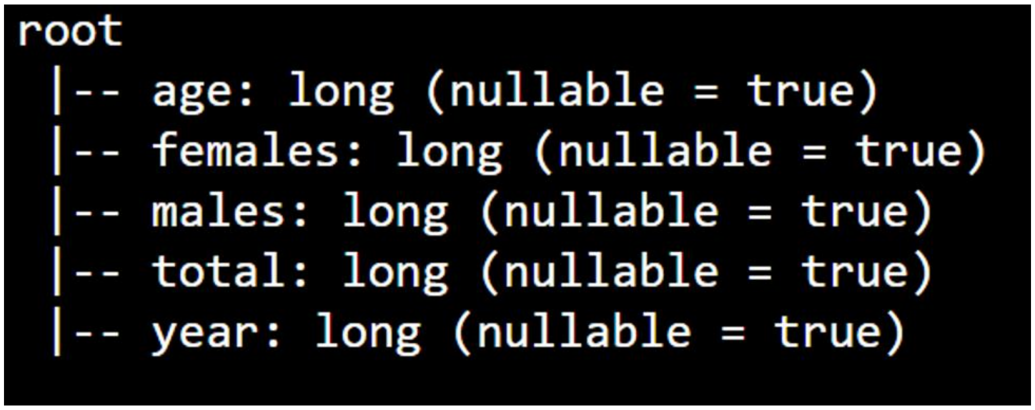
- age
- females
- males
- total
- year

Spark has its own type system that's similar to the pandas type system. To create a DataFrame, Spark iterates over the data set twice - once to extract the structure of the columns, and once to infer each column's type. We will use one of the Spark DataFrame instance methods to display the schema for the DataFrame we're working with. For this we will use the printSchema() method.



```
sqlCtx = SQLContext(sc)
df = sqlCtx.read.json("census_2010.json")
df.printSchema()
```

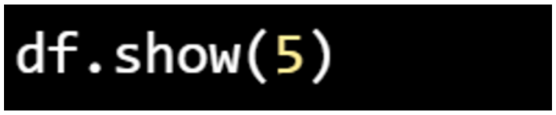
As the output we will see the schema of our dataframe:



```
root
 |-- age: long (nullable = true)
 |-- females: long (nullable = true)
 |-- males: long (nullable = true)
 |-- total: long (nullable = true)
 |-- year: long (nullable = true)
```

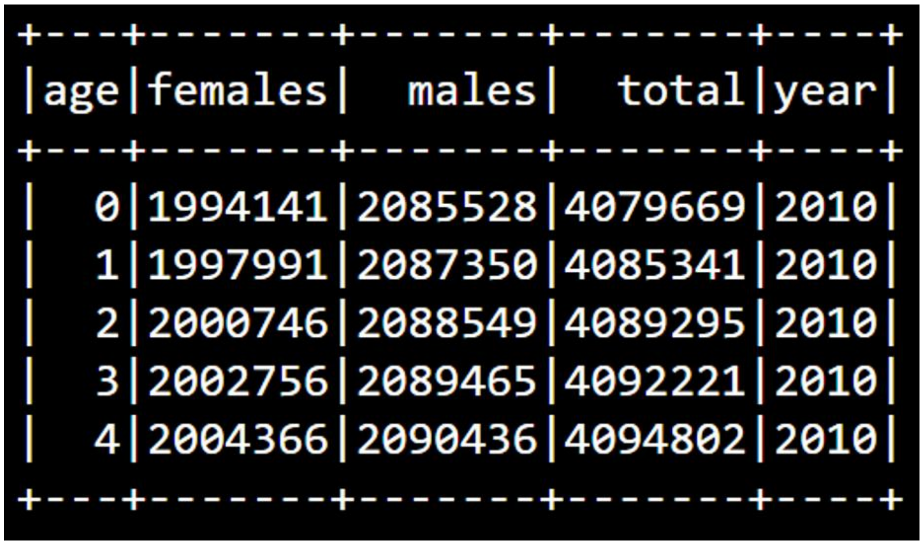
4. Unlike pandas DataFrames, however, Spark DataFrames are immutable, which means we can't modify existing objects. Most transformations on an object return a new DataFrame reflecting the changes instead. Spark's creators deliberately designed immutability into Spark to make it easier to work with distributed data structures.

We will check out the pandas dataframe df using the show() method.



```
df.show(5)
```

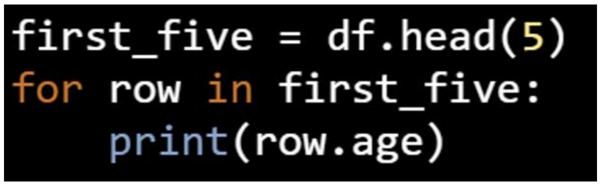
With the output being shown in the form of a table:



```
+---+-------+-------+-------+----+
|age|females|  males|  total|year|
+---+-------+-------+-------+----+
|  0|1994141|2085528|4079669|2010|
|  1|1997991|2087350|4085341|2010|
|  2|2000746|2088549|4089295|2010|
|  3|2002756|2089465|4092221|2010|
|  4|2004366|2090436|4094802|2010|
+---+-------+-------+-------+----+
```

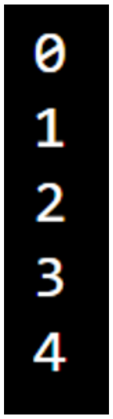
5. The head() method in Spark returns a list of row objects. Spark needs to return row objects for certain methods, such as head(), collect() and take().

We can access a row's attributes by the column name using dot notation, and by position using bracket notation with an index. So, using the head() method, we will get the first five rows in the DataFrame as a list of row objects into first_five. Then we will print the age value for each row object in first_five.



```
first_five = df.head(5)
for row in first_five:
    print(row.age)
```

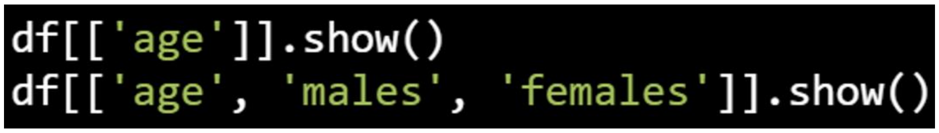
The output will be presented in a column form:



```
0
1
2
3
4
```

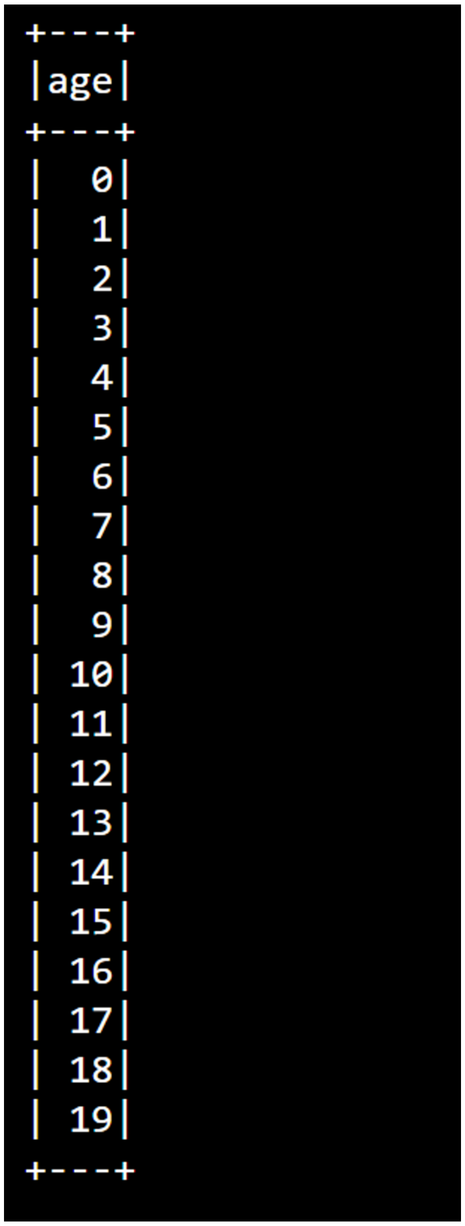
6. We can use the same kind of notation to get a particular column in spark as we did in Python i.e. passing the name of the column as a string while calling the dataframe.

For the current dataframe, we will get the age, males and females columns and display the using show method directly.

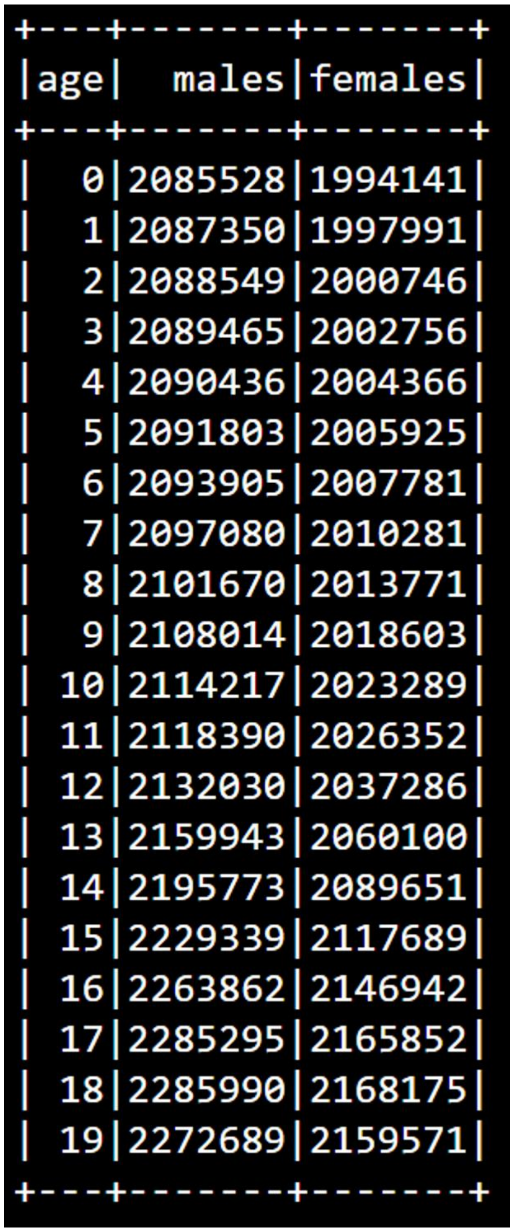


```
df[['age']].show()
df[['age', 'males', 'females']].show()
```

As output, we will get two tables, one after the other:

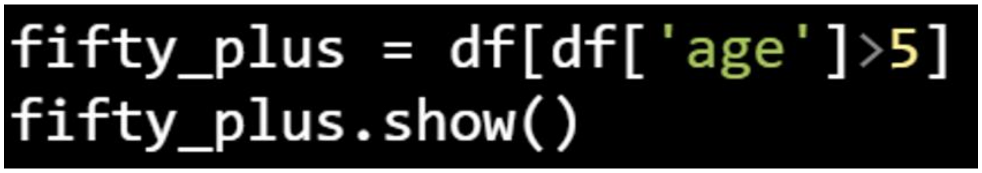


```
+---+
|age|
+---+
|  0|
|  1|
|  2|
|  3|
|  4|
|  5|
|  6|
|  7|
|  8|
|  9|
| 10|
| 11|
| 12|
| 13|
| 14|
| 15|
| 16|
| 17|
| 18|
| 19|
+---+
```



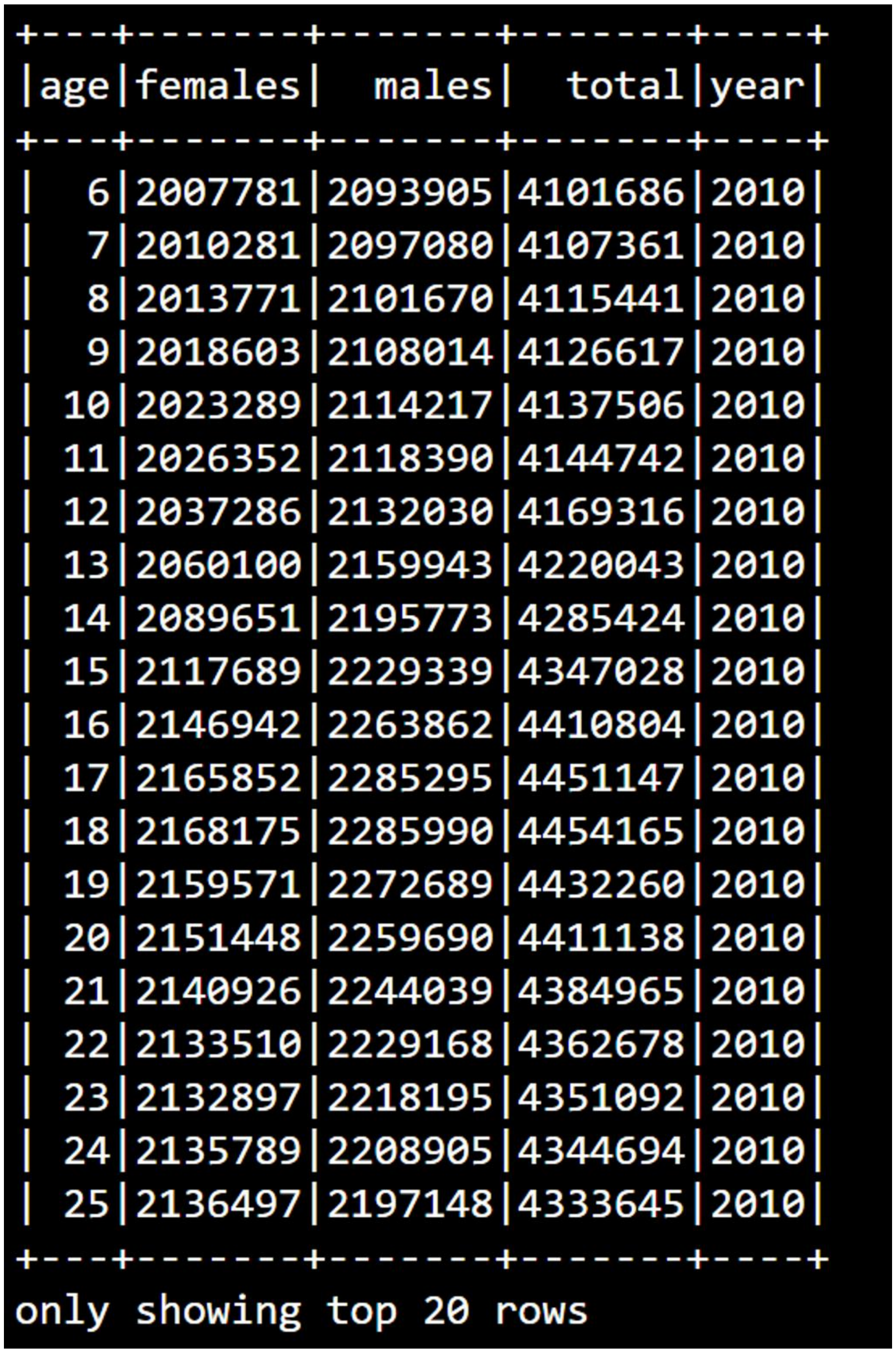
```
+---+-------+-------+
|age|  males|females|
+---+-------+-------+
|  0|2085528|1994141|
|  1|2087350|1997991|
|  2|2088549|2000746|
|  3|2089465|2002756|
|  4|2090436|2004366|
|  5|2091803|2005925|
|  6|2093905|2007781|
|  7|2097080|2010281|
|  8|2101670|2013771|
|  9|2108014|2018603|
| 10|2114217|2023289|
| 11|2118390|2026352|
| 12|2132030|2037286|
| 13|2159943|2060100|
| 14|2195773|2089651|
| 15|2229339|2117689|
| 16|2263862|2146942|
| 17|2285295|2165852|
| 18|2285990|2168175|
| 19|2272689|2159571|
+---+-------+-------+
```

7. In spark, just like in pandas, we use Boolean filtering to select only the rows we were interested in. We will use this functionality to select the rows where age is greater than 5, assign the resulting dataframe to fifty_plus and display it using show() method.



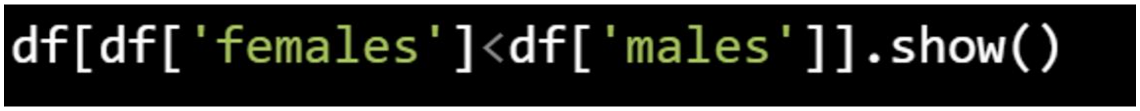
```
fifty_plus = df[df['age']>5]
fifty_plus.show()
```

We can see the that the age value of all the rows is greater than 5:



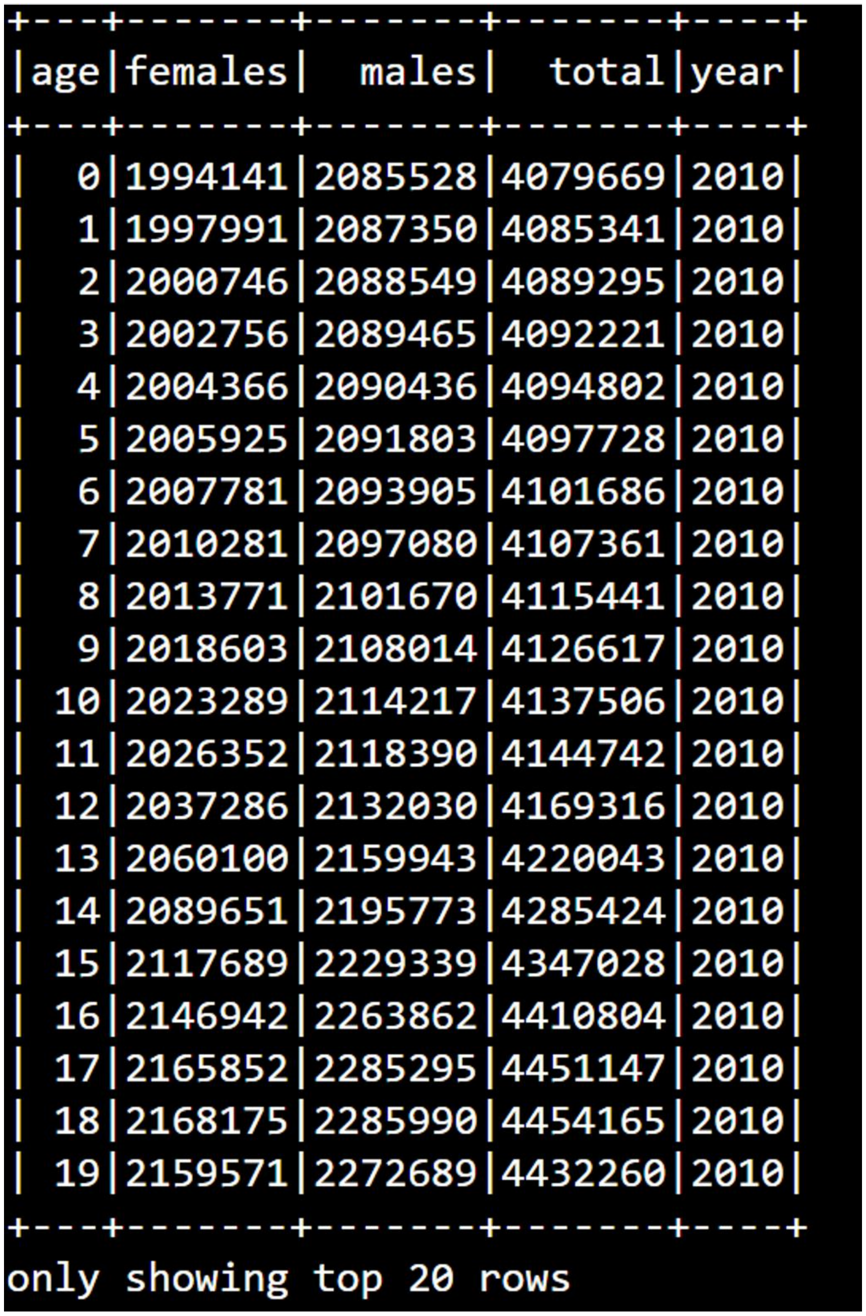
```
+---+-------+-------+-------+----+
|age|females|  males|  total|year|
+---+-------+-------+-------+----+
|  6|2007781|2093905|4101686|2010|
|  7|2010281|2097080|4107361|2010|
|  8|2013771|2101670|4115441|2010|
|  9|2018603|2108014|4126617|2010|
| 10|2023289|2114217|4137506|2010|
| 11|2026352|2118390|4144742|2010|
| 12|2037286|2132030|4169316|2010|
| 13|2060100|2159943|4220043|2010|
| 14|2089651|2195773|4285424|2010|
| 15|2117689|2229339|4347028|2010|
| 16|2146942|2263862|4410804|2010|
| 17|2165852|2285295|4451147|2010|
| 18|2168175|2285990|4454165|2010|
| 19|2159571|2272689|4432260|2010|
| 20|2151448|2259690|4411138|2010|
| 21|2140926|2244039|4384965|2010|
| 22|2133510|2229168|4362678|2010|
| 23|2132897|2218195|4351092|2010|
| 24|2135789|2208905|4344694|2010|
| 25|2136497|2197148|4333645|2010|
+---+-------+-------+-------+----+
only showing top 20 rows
```

8. We can compare the columns in Spark DataFrames with each other, and use the comparison criteria as a filter. For example, to get the rows where the population
of males exceeded females in 2010, we'd write the same notation that we would use in pandas.



```
df[df['females']<df['males']].show()
```

The output will show only those rows of the dataframe where the population of males exceeded the female population:

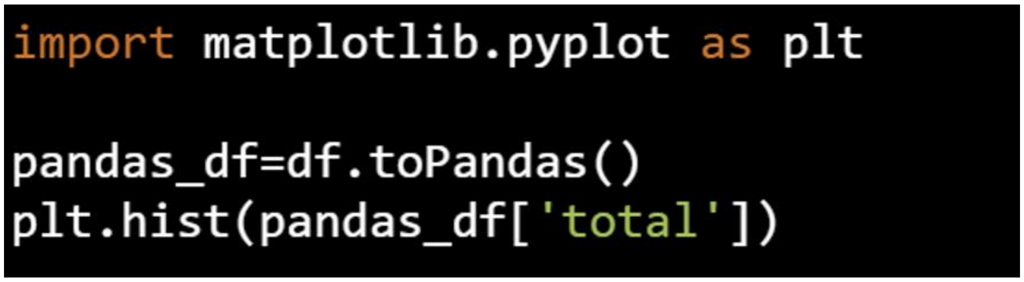


```
+---+-------+-------+-------+----+
|age|females|  males|  total|year|
+---+-------+-------+-------+----+
|  0|1994141|2085528|4079669|2010|
|  1|1997991|2087350|4085341|2010|
|  2|2000746|2088549|4089295|2010|
|  3|2002756|2089465|4092221|2010|
|  4|2004366|2090436|4094802|2010|
|  5|2005925|2091803|4097728|2010|
|  6|2007781|2093905|4101686|2010|
|  7|2010281|2097080|4107361|2010|
|  8|2013771|2101670|4115441|2010|
|  9|2018603|2108014|4126617|2010|
| 10|2023289|2114217|4137506|2010|
| 11|2026352|2118390|4144742|2010|
| 12|2037286|2132030|4169316|2010|
| 13|2060100|2159943|4220043|2010|
| 14|2089651|2195773|4285424|2010|
| 15|2117689|2229339|4347028|2010|
| 16|2146942|2263862|4410804|2010|
| 17|2165852|2285295|4451147|2010|
| 18|2168175|2285990|4454165|2010|
| 19|2159571|2272689|4432260|2010|
+---+-------+-------+-------+----+
only showing top 20 rows
```

9. The Spark Dataframe is relatively new as compared to Pandas dataframe, so its library is still limited. There is no easy way to create a histogram of a column’s data, or a line plot of values in two columns.

To handle some of these shortcomings, we can convert a Spark DataFrame to a pandas DataFrame using the toPandas()method. Converting an entire Spark DataFrame to a pandas DataFrame works just fine for small data sets. For larger ones, though, we'll want to select a subset of the data that's more manageable for pandas.

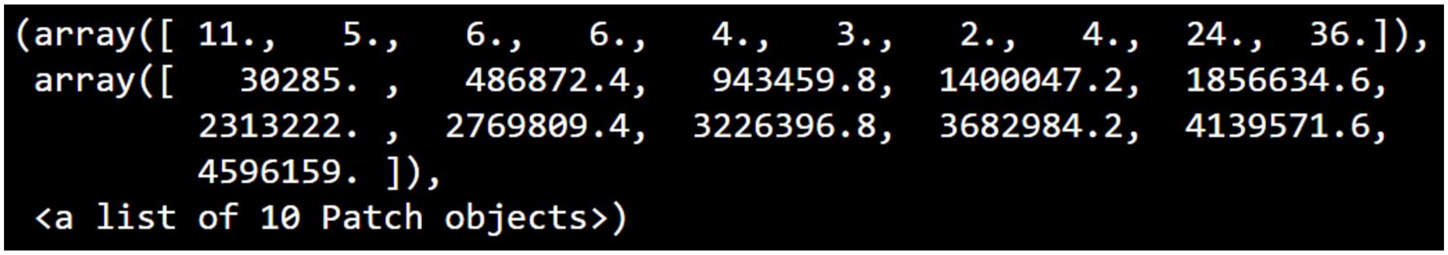
We will use the toPandas() method to convert Spark Dataframe to a pandas DataFrame and assign it to the variable pandas_df. Then we will, plot a histogram of the total column using hist() method in pandas.



```
import matplotlib.pyplot as plt

pandas_df=df.toPandas()
plt.hist(pandas_df['total'])
```

The output will consist of the conversion message and a histogram plot:



```
(array([ 11.,   5.,   6.,   6.,   4.,   3.,   2.,   4.,  24.,  36.]),
 array([   30285. ,   486872.4,   943459.8,  1400047.2,  1856634.6,
         2313222. ,  2769809.4,  3226396.8,  3682984.2,  4139571.6,
         4596159. ]),
 <a list of 10 Patch objects>)
```