Spark Project on Demographic Analysis of Turkey

Xiao Lulu*
School of Data Science
Fudan University
17307130304@fudan.edu.cn

Abstract

This is a report on how to use Spark to handle massive amounts of data by operating on specific data. In python or other supported languages, by using some operators, we can tell Spark what we wish to do to compute with data, and as a result, it can construct an efficient query plan for execution. In this project, we not only use low-level RDD API , but also express the same query with high-level DSL operators and the DataFrame API. This report contains experimental analysis, discussion of proposed methodology, errors raised in the process and the corresponding solutions.

1 About the dataset

The dataset contains some Turkish citizenship information which helps to develop an understanding of the age , sex, and geographical distribution of a population. This leak contains the following information for 49,611,709 Turkish citizens. The leaked database only includes adults of 18 or older and do not includes deceased citizens as of 2009.

It takes me a long time to load the data as a unsuitable method is used. At first I attempt to load the data_dump.sql file into the postgresql database and then use jdbc to connect the database and get a dataframe. However, some errors are raised that it exceeds the memory and excesses response time. It's probably because it loads the dataframe at once into the memory and there is not sufficient space to hold it. It's a better choice to load the data into hive. To simplify the process, I decide to parse the data manually. Firstly read the file 'data_dump.sql' in text format and get a rdd. Then use map transformation to operate on each row and finally convert it to a dataframe. The first row is shown below(only part of the columns are shown). And the total count 49611709 is in line with reality.

```
spark = (SparkSession
.builder
.appName("mernis")
.getOrCreate())
# load data
file_path = '/root/myfile/mernis/data_dump.sql'

data = spark.sparkContext.textFile(file_path). \
filter((lambda line: re.findall('^\d{6}', line))). \
map(lambda line: line.split('\t')[:-1])

schema = "uid STRING, national_identifier STRING, first STRING, last STRING, " \
"mother_first STRING, father_first STRING, gender STRING, birth_city STRING, " \
```

*Use footnote for providing further information about author (webpage, alternative address)—*not* for acknowledging funding agencies.

33rd Conference on Neural Information Processing Systems (NeurIPS 2019), Vancouver, Canada.

```
"date_of_birth STRING, id_registration_city STRING, id_registration_district " \
"STRING, " \
"address_city STRING, address_district STRING, address_neighborhood STRING, " \
"street_address STRING, door_or_entrance_number STRING"

df = spark.createDataFrame(data, schema)
total_count = df.count()
print(total_count)
df.show(1)
```

RESULT

```
|   uid|national_identifier|   first|    last|mother_first|father_first|gender|
+------+-------------------+--------+--------+------------+------------+------+
|291990|        23480340824|NESLIHAN|  ZENGIN|      ZEYCAN|       OSMAN|     K
+------+-------------------+--------+--------+------------+------------+------+
only showing top 1 rows
49611709
```

2 Easy part

2.1 The oldest men among all Turkish citizens

Considering that the column birth_of_data is in character format so it can't be compared directly. Noticing that the date format in the sample data is like 'd-M-yyyy', I tried to convert it using the following command:

```
>  df_format_date = res.withColumn('date_of_birth', to_date('date_of_birth',
'dd/mm/yyyy'))
```

Raise an error:

```
at org.apache.spark.api.python.BasePythonRunner$WriterThread.run(PythonRunner
.scala:232)
Caused by: java.time.format.DateTimeParseException: Text '13/12/95' could not
be parsed at index 6
```

The reason is that there exists non-standard dates in the data, such as '13/12/95', which cannot be resolved. Therefore, a UDF is used to standardize the date format, which is unified into "dd_mm_yyyy", and then converted into date type. Since the data was collected in 2009, '01/01/2009' is taken as the cutoff date to calculate the age. '13/12/95' is converted to '0095-12-13' which is obviously a wrong date so I set these ages to null.

```
# Convert the date to a standard format, dd/mm/yyyy and add 0s in front of
the strings.
def format_date(line):
    li = line.split('/')
    if len(li[2]) == 4 and 0 < len(li[1] <= 2 and 0 < len(li[1])) <= 2:
        return li[2] + '-' + li[1].zfill(2) + '-' + li[0].zfill(2)
    else:
        return 'null'

    format_date_udf = udf(format_date, returnType=StringType())

    df.createOrReplaceTempView('citizens')
    res = df.withColumn("date_of_birth", format_date_udf(df["date_of_birth"]))
    df_format_date = res.withColumn('date_of_birth', to_date('date_of_birth'))
    df_format_date.show(10)
```

I first select all men from the dataframe. Because the number of oldest men may not only one, we'd better find the earliest date of birth and than go through each row to find the exact men corresponding to it.

```
# filter out male
def oldest_men():
    male = df_format_date.filter(expr(""" gender='E' """))
    male.createOrReplaceTempView('male')
    oldest_men = spark.
    sql('SELECT UID, date_of_birth FROM male
        'WHERE date_of_birth = '
        '(SELECT min(date_of_birth) FROM male)' )
    print("The oldest men:", oldest_men.collect())
```

Unlike showing first few rows, it takes some time to output the result as the entire dataset will be read. Finally we get the result:

RESULT

```
> oldest_man()
The oldest men: [Row(uid='32198722', national_identifier='52552749852',
first='CELIL', last='UNAL', mother_first='HAYRIYE', father_first='MEHMET',
gender='E', birth_city='BARTIN', date_of_birth=datetime.date(1329, 9, 20),
id_registration_city='BARTIN', id_registration_district='AMASRA',
address_city='BARTIN', address_district='AMASRA', address_neighborhood=
'ALIOBASI KOYU', street_address='MERKEZ SOKAK',door_or_entrance_number='66')]
```

2.2 Count the most common letters in all names

It's a little complicated to solve such a problem only with dataframe, compared to which rdd is more flexible. Firstly merge the last name and first name into a string, then convert it into a list of character, and use the flatmap() method to get an object composed of characters from all iterators. Considering there are some blank characters, we can use isalpha() method to filter out valid characters. Rather than sql, I use max() method of rdd to find the most common letters of full names.

```
def letter_most_name():
    full_name_rdd = df.select(lower('first'), lower('last')).rdd
    res = full_name_rdd. \
        flatMap(lambda line: list((line[0] + line[1]))). \
        map(lambda letter: (letter, 1)). \
        reduceByKey(lambda a, b: a + b). \
        filter(lambda x: x[0].isalpha()). \
        repartition(1). \
        max(lambda x: x[1])
    print("The letters that appear most frequently are: ")
    print(res)
```

RESULT

```
> letter_most_name()
The letters that appear most frequently are: ('a', 82319942)
```

2.3 The age distribution of the population in the country grouping by age groups (0-18, 19-28, 29-38, 39-48, 49-59, >60).

We can define a udf age_group to classify the ages according to different groups(0-18,19-28,29-38,39-48,49-59,>60). Apply months_between() method to the current time "2009-12-31" and the

date of birth of each row and then divide the results by 12. Thus we can roughly get the age column and map it into corresponding age group to get the age group column.

Sometimes the absolute number does not directly reflect the age distribution of the population, while the proportion can more clearly show the composition of the population.

```
def age_group(age):
    if not isinstance(age, float):
        return 'null'
    if 0 < age <= 18:
        return '0-18'
    elif 18 < age <= 28:
        return '19-28'
    elif 28 < age <= 38:
        return '29-39'
    elif 38 < age <= 48:
        return '39-48'
    elif 48 < age <= 59:
        return '49-59'
    elif 59 < age < 200:
        return '>60'
    else:
        return 'null'

age_udf = udf(age_group, returnType=StringType())

def age():
    df_age = df_format_date.withColumn('age', (round(months_between(
        to_date(lit('2009-12-31')), col('date_of_birth')) / 12, 2)).
        cast('float'))
    print(df_age.schema)
    df_age.show()
    df_age_group = df_age.withColumn('age_group', age_udf(df_age['age']))
    dist = df_age_group.groupBy('age_group'). \
        agg(count('*').alias('total_number'), round((count('*') /
        total_count), 3).\
        alias('proportion'))
    dist.show()
```

RESULT

```
> age()
+---------+------------+----------+
|age_group|total_number|proportion|
+---------+------------+----------+
|    29-39|    12239293|     0.247|
|    49-59|     7827432|     0.158|
|    19-28|    11520654|     0.232|
|    39-48|     9688217|     0.195|
|      >60|     8265129|     0.167|
|     NULL|       70984|     0.001|
+---------+------------+----------+
```

2.4 The number and the proportion of male and female

Group the date row by gender and use agg() to count the the number of the male and female.

```
def male_female():
    num_male_female = df.select('gender').groupBy('gender').agg(count('*').
    alias('total'))
    num_male_female.show()
    num_male_female.createOrReplaceTempView('tc')
    male_num = spark.sql("""SELECT t.total FROM tc t WHERE t.gender='E'""")
    .collect()
    female_num = spark.sql("""SELECT t.total FROM tc t WHERE t.gender='K'""")
    .collect()
    print('The number of males'
    'in the country is{}, The number of females in the country is{}'.
    format(male_num[0][0], female_num[0][0]))
    ratio = male_num[0][0] / female_num[0][0]
    print('The female to male ratio is ', ratio)
```

Group the date row by gender and use agg() to count the the number of the male and female. By sql we can finally get the result: By sql we can finally get the result:

RESULT

```
> male_female()
+------+--------+
|gender|   total|
+------+--------+
|     K|25077226|
|     E|24534483|
+------+--------+
The number of males in the country is 24534483, The number of males in the
country is 25077226
The female to male ratio is 1.0221216399791264
```

The number of male is 25077226 which is a little larger than the number of female, just like most other countries. Female to male ratio is about 1.02 which is very close to 1, which is well-balanced.

One thing to note is that the type returned by spark.sql() is still dataframe. After using the collect() method, the result is a list containing Row(). To get the value of numeric type when referencing the result, need to index the list and row.

2.5 Months with the highest male and female birth rate in the country

Split the complete dataframe into two parts, one of which contains all the rows of male and the other contains female. Then count the number of the months of birth separately.

```
def birth_rate():
    df1 = df_format_date.select('gender', 'date_of_birth')
    df_month = df1.
        select('gender', month('date_of_birth').
        alias('birth_month'))
    male = df_month.
        filter(expr("""gender = 'E'""")). \
        groupBy('birth_month'). \
        agg(count('*').alias('count')). \
        rdd. \
        max(lambda line: line[-1])

    female = df_month.
        filter(expr("""gender = 'K'""")). \
```

```
        groupBy('birth_month'). \
        agg(count('*').alias('count')). \
        rdd. \
        max(lambda line: line[-1])
    print("The month with the highest male birth rate:", male)
    print("The month with the highest female birth rate:", female)
```

RESULT

```
> birth_rate()
The month with the highest male birth rate Row(birth_month=1, count=3912665)
The month with the highest male birth rate Row(birth_month=1, count=3911255)
```

2.6 street with the most residential population

```
def street():
    df_stree = df.
        select('street_address').
        groupBy('street_address'). \
        agg(count('*').alias('total'))
    df_street.createOrReplaceTempView('df_street')
    res = spark.sql('select street_address, total as total from df_street t \
    where total = (select MAX(total) from df_street)')
    res.show()
```

When counting the number of resident population of each street, I ignore the city and district where the street lies in. So if there are some streets of the same name, the number will be added altogether.

RESULT

```
+--------------+-------+
|street_address|  total|
+--------------+-------+
| KOYUN KENDISI|4748581|
+--------------+-------+
```

3 Normal part

3.1 The most common surnames of male and female.

```
def last_name():
    df_n1 = df.select('gender', 'last')
    male = df_n1.filter("""gender = 'E'""")
    male.groupBy('last'). \
        agg(count('*'). \
        alias('total')). \
        orderBy('total', ascending=False). \
    show(10)

    female = df_n1.filter("""gender = 'K'""")
    female.groupBy('last'). \
        agg(count('*'). \
        alias('total')). \
        orderBy('total', ascending=False). \
    show(10)
```

The format of the problem is similar to E5 so we can do the same operations. Separately count the 10 most common surnames among male and female. The result listed below is only for men.

```
> last_name()
+--------+------+
|    last| total|
+--------+------+
|  YILMAZ|352338|
|    KAYA|244272|
|   DEMIR|231289|
|   SAHIN|201958|
|   CELIK|199622|
+--------+------+
only showing top 5 rows
+--------+------+
|    last| total|
+--------+------+
|  YILMAZ|355954|
|    KAYA|244100|
|   DEMIR|230428|
|   SAHIN|202155|
|   CELIK|199330|
+--------+------+
only showing top 5 rows
```

3.2 What is the average age of citizens in each city? Analyze the population aging degree of each city, and judge whether the current city is in an aging society.

select out the column address_city and age then group by the cities. Use avg() method to calculate the average age.

Since two indicators(60 and 65) must be combined to determine the aging of an area, we can group by city name and count the number of citizens over 60 and 65 years old. Then judge whether the city is in an aging society. The result is represented by a value of type bool.

```
df_age = df_age.withColumn('gt_60', (col('age') >= 60).cast('int')). \
withColumn('gt_65', (col('age') >= 65).cast('int'))
df_age.createOrReplaceTempView('df_age')
res1 = spark.sql("""
    select address_city, count(*) as count, avg(age) as avg_age,
    sum(gt_60)/count(*) as gt_60, sum(gt_65) / count(*) as gt_65
    from df_age group by address_city
""")
res2 = res1.selectExpr('address_city', 'count', 'round(avg_age,3) as avg_age',
    'round(gt_60,3) as gt_60', 'round(gt_65,3) as gt_65')
res3 = res2.withColumn('aging', (col('gt_60') > 0.1) | (col('gt_65') > 0.07))
res3.show(5)
```

RESULT

```
+------------+-------+-------+-----+-----+-----+
|address_city|  count|avg_age|gt_60|gt_65|aging|
+------------+-------+-------+-----+-----+-----+
|       ADANA|1390497| 40.957| 0.13|0.085| true|
|     TRABZON| 533719| 43.671|0.189|0.139| true|
|     DENIZLI| 666304| 43.311|0.175|0.123| true|
|   BALIKESIR| 847038| 45.978|0.222|0.159| true|
|     BILECIK| 141691| 44.575|0.201|0.151| true|
```

```
+------------+-------+-------+-----+-----+-----+
only showing top 5 rows
```

3.3 Among the 10 most populous cities, the top two months in which the most people are born

We first have to find the top 10 most populous cities. Count the number of surnames grouping by the name of city and month of birth, sort the number of each month in each city, and then select the two months with the largest number of births. The window() method can be used to sort the birth months of each city separately. Finally select the first two months and merge the final results.

```
def city10():
    df4 = df_format_date.select('address_city', month('date_of_birth').
    alias('birth_month'))
    df4.createOrReplaceTempView('df4')
    # 10
    top10 = spark.sql(""" select address_city, count(*) as total
    from df4 group by address_city order by total desc limit 10""")
    top10.createOrReplaceTempView('top10')
    # 10
    res1 = spark.sql("""
        select t1.address_city as address_city, t1.birth_month,
        count(*) as total from df4 t1
        where t1.address_city in (select address_city from top10)
        group by t1.address_city, t1.birth_month
    """)
    res1.show()
    window = Window.partitionBy(res1['address_city']).orderBy(res1['total']
        .desc())
    res1.
        select('*', rank().
        over(window).
        alias('rank')).
        filter(col('rank') <= 2).show()
```

RESULT

```
+------------+-----------+-------+----+
|address_city|birth_month|  total|rank|
+------------+-----------+-------+----+
|       ADANA|          1| 275752|   1|
|       ADANA|          3| 134666|   2|
|    ISTANBUL|          1|1229999|   1|
|    ISTANBUL|          3| 883712|   2|
+------------+-----------+-------+----+
```

3.4 Count the top three surnames in each of the country's 10 most populous cities, and analysis whether surnames are associated with the city in which they are located.

The first question is similar to N3, so I won't repeat it here.

```
def last_top3():
    df5 = df_format_date.select('address_city', 'last')
    df5.createOrReplaceTempView('df5')
    # Top 10 most populous cities
    top10 = spark.sql("""
        select address_city, count(*) as total
```

```
        from df5 group by address_city order by total desc limit 10
    """)
    top10.createOrReplaceTempView('top10')
    # Number of each surname in the 10 most populous cities
    res1 = spark.sql("""
        select t1.address_city as address_city, t1.last
        as last,
        count(*) as total from df5 t1
        where t1.address_city in (select address_city from top10) group by
        address_city, last
    """)
    # res1.show()
    res1.createOrReplaceTempView('res1')
    # 1
    window = Window.
        partitionBy(res1['address_city']).
        orderBy(res1['total']
        .desc())
    res2 = res1.
        select('*', rank().
        over(window).alias('rank')).
        filter(col('rank') <= 3)
    res2.show()
```

RESULT

```
+------------+------+------+----+
|address_city|  last| total|rank|
+------------+------+------+----+
|       ADANA|YILMAZ| 16209|   1|
|       ADANA|  KAYA| 13170|   2|
|       ADANA| DEMIR| 11536|   3|
|    ISTANBUL|YILMAZ|139022|   1|
|    ISTANBUL|  KAYA| 87269|   2|
|    ISTANBUL| DEMIR| 78160|   3|
|       BURSA|YILMAZ| 27375|   1|
+------------+------+------+----+
```

To measure whether there is a correlation between two discrete variables, we can use the chi-square test. One problem is how to completely convert the count column of the last name into a vector. I tried to expand the count column horizontally, that is, the city name as the row name, and all the last names in the 10 cities as the column name, and the null value is filled with 0. It has the advantage of avoiding the expansion of the last names that do not exist in the city. Then compress all the surname columns into a vector.

```
...
# Chi-squared test
# all the surnames in the top 10 city
all_last_name = res1.select('last').distinct().collect()
all_last_name = [row['last'] for row in all_last_name]
# conver a narrow table to a wide table
res = res1.groupBy('address_city'). \
    pivot('last', all_last_name). \
    agg(F.first('total', ignorenulls=True)). \
    fillna(0)
df_res = spark.createDataFrame(res, ['address_city', 'all_last_name'])
last_col_name = df_res.columns.remove('address_city')
assembler = VectorAssembler(
```

```
inputCols=last_col_name,
outputCol="vector_last_name")
vectorized_df = assembler.transform(df_res).select('address_city',
'vector_last_name')
```

But unfortunately, it warns me that this will be out of memory, so I must change my thinking for another strategy. I try to convert the surname count of each city directly into a list, but this means that I have to expand the non-existent surname and set it to 0. The join operation can achieve this. But this method involves loop and join operations, so the efficiency will be relatively low.

```
...
city10 = top10.select('address_city').take(10)
all_last_name = res1.select('last').distinct()
li = []
for city in city10:

# expand the non-existent surname and set it to 0
res2 = res1.\
    where(col('address_city') == city['address_city']).\
    select('last').\
    join(all_last_name,'last','right_outer').\
    fillna(0).\
    orderBy('last')

vector_last_name = res2.select('total').
    rdd.
    map(lambda x: x[0]).
    collect()
print(vector_last_name)
vector_last_name = Vectors.dense(vector_last_name)
res = [city['address_city'], vector_last_name]
li.append(res)

vectorized_df = spark.createDataFrame(li,['address_city',
    'vector_last_name'])
# Chi-square test
res = ChiSquareTest.test(vectorized_df, "vector_last_name ", "address_city")
res.show()
r = res.head()
print("pValues: " + str(r.pValues))
print("degreesOfFreedom: " + str(r.degreesOfFreedom))
print("statistics: " + str(r.statistics))

```

4 Summary

The main problems is exceeding time and out of memory.

```
> java.lang.OutOfMemoryError: Java heap space
> java.lang.OutOfMemoryErrorGC overhead limit exceeded
```

Typically, these two exceptions are caused by insufficient memory for executor or driver Settings. Driver Settings that are too small are relatively rare, but are usually caused by insufficient memory for executor. In either case, we can solve the problem by submitting commands or by specifying the size of driver-memory and executor-memory in Spark's configuration file.

```
> spark-submit --master yarn-cluster --class MAIN_CLASS --executor-memory 10G
```

```
--executor-cores 10 --driver-memory 2g --name APP_NAME
```

Due to the excessive content of the output result, only a small portion is shown here.

RESULT

```
+--------------------+--------------------+--------------------+
|             pValues|    degreesOfFreedom|          statistics|,
+--------------------+--------------------+--------------------+,
|[0.35571384759874...|[18, 0, 9, 9, 0, ...|[23.0000000000000...|,
+--------------------+--------------------+--------------------+,
```

In the process of solving the above problems, these methods are mainly used:

1. Conversion between dataframe and rdd. However, that this does have a cost and should be avoided unless necessary. When we want to precisely instruct Spark how to do a query and there is no proper operations we can do with a dataframe, we prefer to rdd. While if we want to ascertain the most efficient way to build a query and generate compact code, dataframe with Spark SQL is a better choice.
2. When querying a DataFrame, the following metheds can be adopted:
   - Create a temporary viem and use spark.sql("""expression""")
   - Use dataframe API without query statement, such as df.select(col)
   - Combine the two ways, such as df.filter(expr("""expression"""))
3. The flexibility of Spark allows for us to define their own functions as known as UDF. We need to declare a function that performs the operations defined.

We can tell if there is a relationship between these two variables based on the p-value combined with the degree of freedom.