**Big Data Spring 2018**

**Project 4**

Nicolas Eldering

Yifan Chen

**Member 1: Nicolas Eldering**  
My teammate and I agree that I handled 50% of the overall project. My specific tasks included:

* Task 1: Worked on part 2 and 4
* Task 2: worked on half the documentation

**Member 2: Yifan Chen**  
My teammate and I agree that I handled 50% of the overall project. My specific tasks included:

* Task 1: Worked on part 1 and 3
* Task 2: worked on half the documentation

**Part 1:**

1) What is the total number of rows in this dataset?

h1bDF = spark.read.csv('/FileStore/tables/h1b\_kaggle.csv', header=True, inferSchema=True)

h1bDF.cache()

print(h1bDF.count())

***Result: 3002458***

2) How many different employers filed for H-1B visa?

# firstly we do filter to get the certified status and then do select and distinct on “EMPLOYER\_NAME”

print(h1bDF.select("EMPLOYER\_NAME").distinct().count())

***Result:*** ***236015***

3) Show 10 different employers ordered by employer name in descending order.

from pyspark.sql.functions import desc

diffEmployer = h1bDF.select("EMPLOYER\_NAME").distinct()\

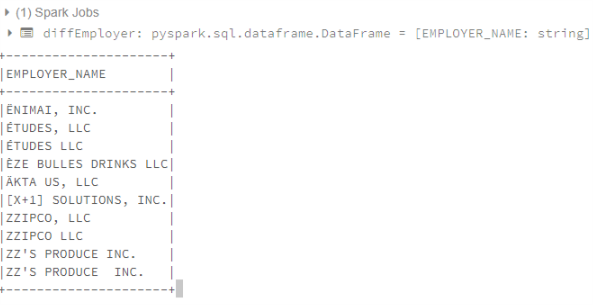
.sort(desc("EMPLOYER\_NAME"))\

.limit(10)

# using truncate = False to get the entire “EMPLOYER\_NAME”, Otherwise, if the name is too long, it would be omitted.

diffEmployer.show(truncate=False)

***Result:***



**Part 2:**

1) How many visa petitions are there for each ‘case status’ (e.g., 1000 withdrawn,

100 denied)? Order results by the number of visa petitions (i.e., count) in

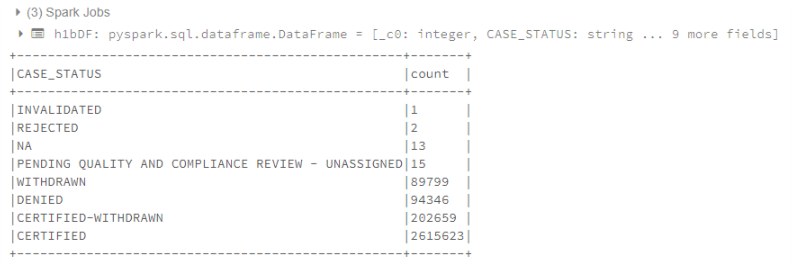
ascending order. Note, ascending order is the default, thus, there is no need to

specify that.

h1bDF = spark.read.csv('/FileStore/tables/h1b\_kaggle.csv', header=True, inferSchema=True)

h1bDF.select('CASE\_STATUS').groupBy("CASE\_STATUS").count().orderBy('count').show(truncate=False)

***Result:***



2) Repeat the above query to perform in-memory processing. Databricks will display

execution time for each ‘cell’, take a screenshot of the cell including the running

times for both in-memory and on-disk.

h1bDF\_cache = spark.read.csv('/FileStore/tables/h1b\_kaggle.csv', header=True, inferSchema=True)

h1bDF\_cache.cache()

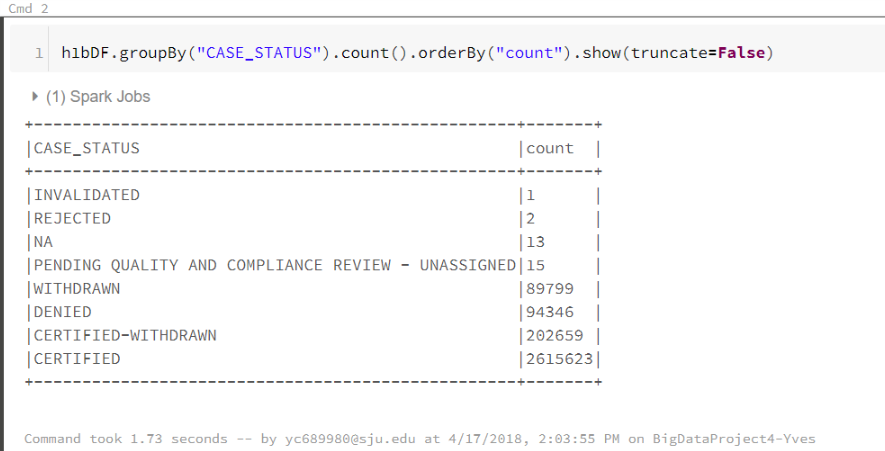
# because of the active(lazy) cache.

h1bDF\_cache.count()

h1bDF\_cache.select('CASE\_STATUS').groupBy("CASE\_STATUS").count().orderBy('count').show(truncate=False)

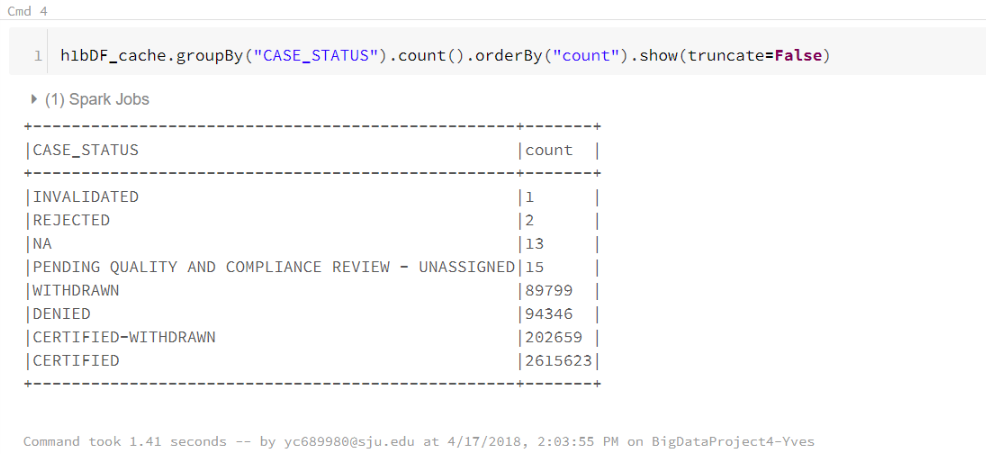
***Screenshot:***

On-disk:



It’ll takes us 1.73 seconds.

In-memory:



It’ll takes us 1.41 seconds.

So in-memory really saves us time compare to on-disk.

3) Use Python, R, or Scala to create a new data-frame (RDD) that contains employer names only. Then, iterate (i.e., loop) through the newly created data-frame to print all rows of data/employer names, and count/print number of rows. Hint, a transformation returns a new data-frame. Be patient, this might take a while.

# after we do collect() we can do iteration to our dataFrame (RDD)

nameDF = h1bDF.select('EMPLOYER\_NAME').rdd

print("Rows number: "+str(nameDF.count()))

for row in nameDF.collect():

print row.EMPLOYER\_NAME

***Result:***

Rows number: 3002458



**Part 3:**

1) How many visa petitions were denied in 2016?

**Spark SQL:**

h1bDF = spark.read.csv('/FileStore/tables/h1b\_kaggle.csv', header=True, inferSchema=True)

h1bDF.cache()

# Create a VIEW to perform SQL queries

h1bDF.createOrReplaceTempView("h1bView")

# cache table

spark.catalog.cacheTable("h1bView")

%sql

select

count(\*) as Denied2016

from

h1bView

where

YEAR='2016' and CASE\_STATUS='DENIED'

**Spark core:**

h1bDF.filter("YEAR='2016'").filter('CASE\_STATUS="DENIED"').count()

***Result:*** ***9175***

2) Show different employers in California and Pennsylvania who have certified visa petitions in 2013. Order results by employer names in ascending order.

**Spark SQL:**

# we use “like” to get the rows which contain “CALIFORNIA” or “PENNSYLVANIA” in column WORKSITE.

%sql

select

distinct EMPLOYER\_NAME

from

h1bView

where

YEAR='2013' and CASE\_STATUS='CERTIFIED' and WORKSITE REGEXP 'CALIFORNIA|PENNSYLVANIA'

order By

EMPLOYER\_NAME

**Spark core:**

h1bDF.filter("YEAR='2013' and CASE\_STATUS='CERTIFIED' and WORKSITE REGEXP 'CALIFORNIA|PENNSYLVANIA'").select("EMPLOYER\_NAME").distinct().orderBy("EMPLOYER\_NAME").show()

***Result:***



3) Is it possible to perform SQL queries directly on RDD? Justify your answer.

No, it’s not possible to perform SQL quires directly on RDD. RDD offers us low-level functionality and control. So it doesn’t support SQL queries, while DataFrame and DataSet all support SQL queries, because Dataset and DataFrame support rich semantics, high-level abstractions, and domain specific APIs, including SQL queries. When our data is unstructured, such as media streams or streams of text, we need to use rdd. When we use rdd, we don’t care about imposing a schema, such as columnar format, while processing or accessing data attributes by name or column. In other words, without schema we cannot directly access data attributes by column name. That’s why we cannot perform SQL queries directly on RDD.

When I did:

h1bDF = spark.read.csv('/FileStore/tables/h1b\_kaggle.csv', header=True, inferSchema=True).rdd

h1bDF.select("EMPLOYER\_NAME")

, error “AttributeError: 'RDD' object has no attribute 'select'” shows up, which means in rdd, we cannot directly access data attributes by column name.

**Part 4:**

1） Since SOC\_NAME column does not specify what sort of data it contains, change the column name to OCCUPATION\_CODE for a coherent column name. Take a screenshot of the Schema, to verify that the column name has changed. Hint, use DF.printSchema() to view column names and data types.

from pyspark.sql.types import StructType, StructField, IntegerType, StringType, DoubleType

from pyspark.sql.functions import min, max, avg

# we change PREVAILING\_WAGE to DoubleType to do calculation conveniently.

struct\_schema = StructType([

StructField("INDEX", IntegerType()),

StructField("CASE\_STATUS", StringType()),

StructField("EMPLOYER\_NAME", StringType()),

StructField("OCCUPATION\_NAME", StringType()),

StructField("JOB\_TITLE", StringType()),

StructField("FULL\_TIME\_POSITION", StringType()),

StructField("PREVAILING\_WAGE", DoubleType()),

StructField("YEAR", StringType()),

StructField("WORKSITE", StringType()),

StructField("lon", StringType()),

StructField("lat", StringType())

])

# header: header = true means we will use the first row as the header of our dataFrame.

# inferSchema: infers the input schema automatically from data. It requires one extra pass over the data. If None is set, it uses the default value, false. Here, we will use our own schema so let’s set it false.

# schema: an optional pyspark.sql.types.StructType for the input schema.

h1bDF = spark.read.format('csv')\

.schema(struct\_schema)\

.option("inferSchema", "false")\

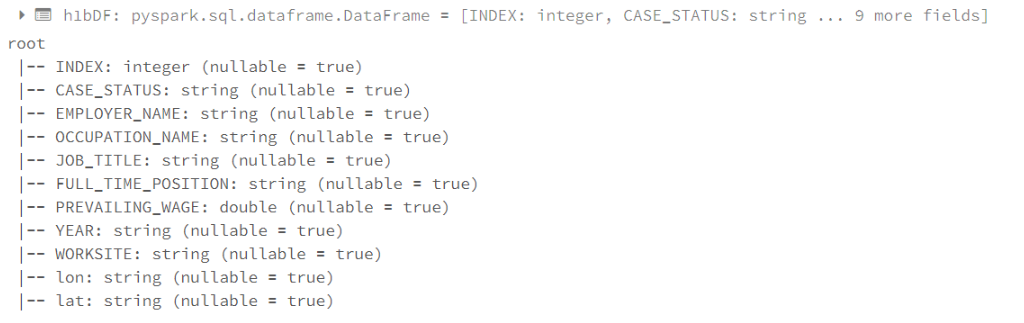
.option("header","true")\

.load("/FileStore/tables/h1b\_kaggle.csv")

h1bDF.cache()

h1bDF.printSchema()

***Screenshot:***



2) Show the lowest, highest, and average wage/salary of all filed visa petitions?

print("average wage")

h1bDF.select(avg("PREVAILING\_WAGE").alias("Average")).show()

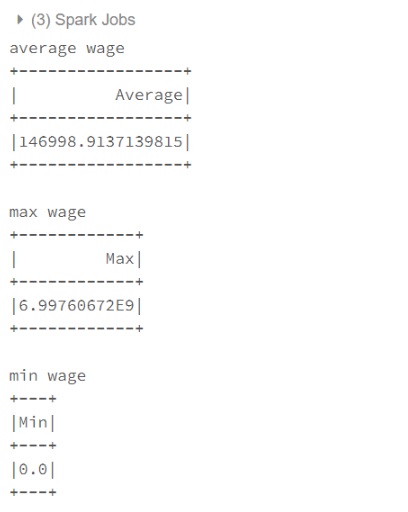
print("max wage")

h1bDF.select(max("PREVAILING\_WAGE").alias("Max")).show()

print("min wage")

h1bDF.select(min("PREVAILING\_WAGE").alias("Min")).show()

***Result:***



3) Print an appropriate message if there are wages that are greater than 80,000. If yes, print how many, else, print “none”.

# filter selects the PREVAILING\_WAGE>80000 rows in h1bDF.

highPayCount = h1bDF.filter(h1bDF.PREVAILING\_WAGE > 80000).count()

if highPayCount > 0:

print("Quanitity = " + str(highPayCount))

else:

print("none")

***Result:* *Quanitity = 789875***