



MASTER M2 MOSIG – UGA

LAB SPARK

Analyzing Data with Spark

Authors:

Abdallah ADWAN

Li MOU

Submitted to:

Assoc Prof. Thomas ROPARS

January 2, 2021

Contents

0.1	Data Formats Used in Working	3
0.2	Packages to be imported for Spark Environment	3
0.3	Question 1	3
0.3.1	What is the distribution of the machines according to their CPU capacity?	3
0.4	Question 2	4
0.4.1	What is the percentage of computational power lost due to maintenance (a machine went offline and reconnected later)?	4
0.5	Question 3	6
0.5.1	On average, how many tasks compose a job?	6

0.1 Data Formats Used in Working

Dataframe: DataFrame is an RDD-based immutable distributed data set officially introduced in Spark 1.3, similar to the two-dimensional table of a traditional database, in which data is organized and stored in the form of columns. If you are familiar with Pandas, it is very similar to Pandas DataFrame. We use the DataFrame to extract the column(s) information and consecutively do the transformations until we obtain the expected format of data, over which we convert it into RDD format for further distributed computation.

RDD: Spark revolves around the concept of a resilient distributed dataset (RDD), which is a fault-tolerant collection of elements that can be operated on in parallel. There are two ways to create RDDs: parallelizing an existing collection in your driver program, or referencing a dataset in an external storage system, such as a shared filesystem, HDFS, HBase, or any data source offering a Hadoop InputFormat.

0.2 Packages to be imported for Spark Environment

```
from pyspark import SparkContext
from pyspark.sql.session import SparkSession
from pyspark.sql import functions as F
# The functions subclass provides almost all functions in SQL

from pyspark.sql.types import *
# Defines the data type of each column in the DataFrame, which is basically synchronized
# with the data type in SQL, and is generally used to specify the table structure
# schema when the DataFrame data is created

# start spark with 1 worker thread
sc = SparkContext("local[1]")
# SparkContext is the entrance of Spark, which is equivalent to the main function of the
# application, here we use the local mode

sc.setLogLevel("ERROR")

spark = SparkSession(sc)
# Provides a session environment for subsequent spark operations, specifically, receives a
# SparkContext object as input, and creates the main entrance of Spark SQL
```

0.3 Question 1

0.3.1 What is the distribution of the machines according to their CPU capacity?

```
# machine_events schema
machineEventsSchema = StructType(
    [
        StructField("timestamp", LongType(), True),
        StructField("machine_id", StringType(), True),
        StructField("event_type", StringType(), True),
        StructField("platform_id", StringType(), True),
        StructField("capacity_cpu", FloatType(), True),
        StructField("capacity_memory", FloatType(), True),
    ]
)
# Detailed data structure information is provided in the DataFrame, so that SparkSQL can
# clearly know which columns are contained in the data set, and what are the names and
```

```

types of each column

# create machine_events dataframe
machineEventsDf =
    spark.read.schema(machineEventsSchema).csv("../project/data/machine_events/*.csv.gz")
# Create dataframe by reading csv file

cpuCapacityCountDf = machineEventsDf.select("machine_id",
    "capacity_cpu").distinct().where(F.col("capacity_cpu").isNotNull()).groupBy("capacity_cpu").count()

```

Here we select column 'machine id' and 'capacity cpu' from machineEventsDf dataframe; then return a new dataframe containing the distinct elements in this dataframe; then filter out rows whose 'capacity cpu' is not null; finally group by 'capacity cpu', forming a grouped format data, and count number of 'machine id' under each certain 'capacity cpu'.

```

cpuCapacityCountDf = cpuCapacityCountDf.toPandas()
# convert it into pandas data format and show
cpuCapacityCountDf.head()

```

	capacity_cpu	count
0	0.25	126
1	0.50	11659
2	1.00	798

Figure 1: Distribution of the machines according to their CPU capacity

0.4 Question 2

0.4.1 What is the percentage of computational power lost due to maintenance (a machine went offline and reconnected later)?

```

# create machine_events dataframe
machineEventsDf=spark.read.schema(machineEventsSchema).csv("../data/machine_events/*.csv.gz")
.where(F.col("capacity_cpu").isNotNull())
# And remove the row whose 'capacity_cpu' is null

# sort out the max timestamp for the further computation of power
maxTime = machineEventsDf.select(F.max("timestamp").alias("max")).collect()[0]["max"]

```

Energy Consumption Calculation

Actually the energy consumption is mainly due to the CPU. Someone may consider the energy consumption by RAM, however, since of the computer random access memory architecture working mechanism, RAM energy is also influenced by data transfer between disks, but we have no idea about it. So:

$$Energy_Consumption_for_each_machine = normalized_CPU_capacity \times each_total_time \quad (1)$$

and

$$Total_Energy_Consumption = \sum_{machine_id} Energy_Consumption_for_each_machine \quad (2)$$

```

eventsPerMachineDf = machineEventsDf.where(F.col("event_type") != "2").select(
    "machine_id",

```

```

"capacity_cpu",
F.struct("event_type", "timestamp").alias("event_type_timestamp"))
# remove the row whose 'event_type' is not "2"

```

Here we create another dataframe namely 'eventsPerMachineDf' based on dataframe machineEventsDf since we would like to generate a new column including information of event_type and capacity_cpu.

Since we need the information of time spent by maintenance, only the event types ADD (0) and REMOVE (1) are needed, so we filter out the event type of UPDATE (2) at the beginning. As the monitoring of event type is for each machine (machine_id), we select the column of "machine_id".

Because the power consumption is proportional to number of CPU, and we know each machine has different number of CPUs (as indicated by "capacity_cpu"), we select the column of "capacity_cpu" as well.

Because the power consumption is proportional to active time of machine, i.e. time between events ADD (0) and REMOVE (1). And we use pyspark.sql.functions.struct to create a new struct column namely 'event_type_timestamp' containing "event_type" and "timestamp". And this new struct column is the ROW type, that is a row in the DataFrame, the fields can be accessed like attributes.

```

eventsPerMachineDf = eventsPerMachineDf.groupBy("machine_id", "capacity_cpu").agg(
    F.collect_list("event_type_timestamp").alias("events"))

```

We group by "machine_id" and "capacity_cpu" for the purpose of a newly aggregated column 'event_type_timestamp' to have time information of events for each machine_id. And pyspark.sql.functions.collect_list is an aggregation function that returns a list of objects with duplicates, then rename it as "events".

Down-Time calculation

The calculateDowntime function is performed on column of "events" row by row. The downtime can be computed as:

For the case(s) whose final event is ADD(0):

$$total_of_down - time = \sum ts_of_ADD(0) + (- \sum ts_of_REMOVE(1)) \quad (3)$$

where ts stands for timestamp

For the case(s) whose final event is REMOVE(1):

$$total_of_down - time = maxTime + \sum ts_of_ADD(0) + (- \sum ts_of_REMOVE(1)) \quad (4)$$

where ts stands for timestamp

```

@udf(returnType=LongType())
def calculateDowntime(machineEvents):
    totalDowntime = 0
    global maxTime
    for event in machineEvents:
        if event[0] == "0":
            totalDowntime += int(event[1])
        elif event[0] == "1":
            totalDowntime -= int(event[1])
        if event[0] == "1" and event is machineEvents[-1]:
            totalDowntime += maxTime
    return totalDowntime

```

Here we defined a UDF (user defined function) which takes the list of (timestamp, event_type) pairs of each machine and calculates the total down time for that machine (sum of time period between events of type 1 (remove) and type 0 add).

```

totalDowntimePerMachineDf = (

```

```

eventsPerMachineDf.withColumn("total_down_time",
    calculateDowntime(F.col("events"))).where(F.col("total_down_time").isNotNull())
# Since we compute the downtime, we reject the case that do not have downtime (downtime = 0)

```

$$Percentage_of_Power_lost = \frac{Total_lost_Energy_Consumption}{Ideal_Total_Energy_Consumption} \times 100\% \quad (5)$$

```

idealComputationCapacity =
    machineEventsDf.where(F.col("capacity_cpu").isNotNull()).rdd.map(lambda x:
        x["capacity_cpu"] * maxTime).sum()

```

We convert the dataframe into RDD (Resilient Distributed Datasets) for the purpose of faster computation by the advantage of distributed computation; We then use map method of RDD to return a new RDD by applying a function to each element of this RDD; Finally we sum over all elements.

```

lostComputationCapacity = totalDowntimePerMachineDf.rdd.map(lambda x: x["capacity_cpu"] *
    x["total_down_time"]).sum()

print( "The percentage of computational power lost due to maintainance is:",
    round((lostComputationCapacity / idealComputationCapacity) * 100, 2),
    "%",)

```

The percentage of computational power lost due to maintainance is: 0.24 %.

0.5 Question 3

0.5.1 On average, how many tasks compose a job?

```

# Create the taskEventsSchema
taskEventsSchema = StructType(
    [
        StructField("timestamp", LongType(), True),
        StructField("missing_info", StringType(), True),
        StructField("job_id", StringType(), True),
        StructField("task_index", StringType(), True),
        StructField("machine_id", StringType(), True),
        StructField("event_type", StringType(), True),
        StructField("user_name", StringType(), True),
        StructField("scheduling_class", StringType(), True),
        StructField("priority", StringType(), True),
        StructField("cpu_request", FloatType(), True),
        StructField("memeory_request", FloatType(), True),
        StructField("disk_space_request", FloatType(), True),
        StructField("machine_restrictions", FloatType(), True),
    ]
)

# create task_events dataframe
taskEventsDf = spark.read.schema(taskEventsSchema).csv("../data/task_events/*.csv.gz")
# Create dataframe by reading csv file

numOfTasksPerJobDf = taskEventsDf.select(
    "job_id",
    "task_index"
).distinct().groupBy("job_id").count()

```

Here we select column 'job id' and 'task_index' from taskEventsDf dataframe; then return a new dataframe containing the distinct elements in this dataframe; finally group by 'job id', forming

a grouped format data, and count number of 'task_index' under each certain 'job_id'

```
avgNumOfTasksPerJob =  
    numOfTasksPerJobDf.select(F.mean("count").alias("avg")).collect()[0]["avg"]  
  
print("The average number of tasks per job is: ", round(avgNumOfTasksPerJob))
```

We select a column from numOfTasksPerJobDf frame such that it is created by applying the pyspark.sql.functions.mean on the extracted aggregation column "count" of dataframe numOfTasksPerJobDf and name this column as "avg", and finally we extract the mean result.

The average number of tasks per job is: 38