MASTER M2 MOSIG – UGA

LAB SPARK

Analyzing Data with Spark

Authors:
Abdallah ADWAN
Li MOU

Submitted to:
Assoc Prof. Thomas ROPARS

January 15, 2021

Contents

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("platfrom_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 totheir 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):

$$total\_of\_down - time = \sum ts\_of\_ADD(0) + (-\sum ts\_of\_REMOVE(1)) \tag{3}$$

where ts stands for timestamp

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

$$total\_of\_down - time = maxTime + \sum ts\_of\_ADD(0) + (-\sum ts\_of\_REMOVE(1)) \tag{4}$$

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)
```

$$Percentage\_of\_Power\_lost = \frac{Total\_lost\_Energy\_Consumption}{Ideal\_Total\_Energy\_Consumption} \times 100\% \tag{5}$$

```
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 maintanance is:",
    round((lostComputationCapacity / idealComputationCapacity) * 100, 2),
    "%",)
```

The percentage of computational power lost due to maintanance 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 avarage 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 avarage number of tasks per job is: 38

0.6 Question 4

0.6.1 What can you say about the relation between the scheduling class of a job, the scheduling class of its tasks, and their priority?

Since the question is asking about the scheduling classes of job and tasks, and their priority. By reference, we are notified that the job is split into tasks for processing, but we have not validated it on this data set given, so we need to ignore it for some moments.

In dataset, the scheduling classes of tasks and jobs co-exist in the table of task_events, however, the scheduling classes of jobs is existing in the table of job_events. And these two tables share the column information of job_id. So, we will work based on these two tables and try to join them together by job_id.

```
jobEventsSchema = StructType(
    [
        StructField("timestamp", IntegerType(), True),
        StructField("missing_info", StringType(), True),
        StructField("job_id", IntegerType(), True),
        StructField("event_type", IntegerType(), True),
        StructField("user_name", StringType(), True),
        StructField("job_scheduling_class", IntegerType(), True),
        StructField("job_name", StringType(), True),
        StructField("logical_job_name", StringType(), True),
    ]
)

taskEventsSchema = StructType(
    [
        StructField("timestamp", IntegerType(), True),
        StructField("missing_info", StringType(), True),
        StructField("job_id", IntegerType(), True),
        StructField("task_index", IntegerType(), True),
        StructField("machine_id", IntegerType(), True),
        StructField("event_type", IntegerType(), True),
        StructField("user_name", StringType(), True),
        StructField("task_scheduling_class", IntegerType(), True),
        StructField("priority", IntegerType(), True),
        StructField("cpu_request", FloatType(), True),
        StructField("memeory_request", FloatType(), True),
        StructField("disk_space_request", FloatType(), True),
        StructField("machine_restrictions", BooleanType(), True),
    ]
)

jobEventsDf = spark.read.schema(jobEventsSchema).csv("../data/job_events/*.csv.gz")
taskEventsDf = spark.read.schema(taskEventsSchema).csv("../data/task_events/*.csv.gz")

job_scheduling_class_Df = (
    jobEventsDf.select("job_id", "job_scheduling_class").distinct()
```

```
    .where(F.col("job_id").isNotNull() & F.col("job_scheduling_class").isNotNull())
    .sort("job_id")
)

task_scheduling_class_and_priority_Df = (
    taskEventsDf.select("job_id", "task_scheduling_class", "priority").distinct()
    .where(F.col("task_scheduling_class").isNotNull() & F.col("priority").isNotNull() &
        F.col("job_id").isNotNull())
    .sort("job_id")
)
```

We Load tasks_events and job_events into dataframes and preprocess the data to remove rows that have null values in columns of interest and rearrange the order of rows according to job_id.

```
new = job_scheduling_class_Df.join(
    task_scheduling_class_and_priority_Df, ["job_id"]
).sort("job_id")
D1 = new.toPandas()
```

We join dataframes of job_events and task_events together. To visualize the joining result, we convert it into pandas. We found that scheduling class of the tasks share the identical value as that of the job they belong, and it demonstrates and validates the fact that job is split into tasks.

So, to compute the correlation between scheduling class and priority, we can use the table of task_event only and drop the column of job_id first, then compute such correlation value based on the columns of task_scheduling_class and priority by the function of pyspark.sql.corr.

```
correlation = (
    task_scheduling_class_and_priority_Df.drop("job_id")
    .agg(
       F.corr(F.col("task_scheduling_class"), F.col("priority"))
    ).alias("correlation")
)
```

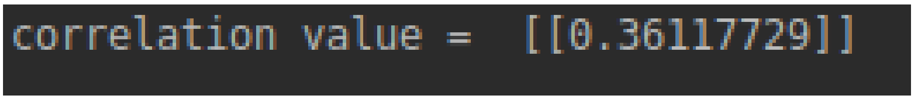


Figure 2: Correlation value

The computed correlation value is about 36.1%. It means the scheduling class has positive correlation with priority but not strong.

0.7 Question 5

0.7.1 Do tasks with low priority have a higher probability of being evicted?

Since the question is asking the whether the tasks with low priority have a higher probability of being evicted, what we are going to do is to count the number of tasks for each level of priority first and count the number of evicted tasks for each level of priority, then we can compute the probability of being evicted for each level of priority.

$$P_{priority\_x}(being\_evicted) = \frac{number\_of\_evicted\_tasks\_for\_x}{total\_number\_of\_tasks\_for\_x} \tag{6}$$

```
taskEventsSchema = StructType(
    [
       StructField("timestamp", IntegerType(), True),
       StructField("missing_info", StringType(), True),
```

```
        StructField("job_id", IntegerType(), True),
        StructField("task_index", IntegerType(), True),
        StructField("machine_id", IntegerType(), True),
        StructField("event_type", IntegerType(), True),
        StructField("user_name", StringType(), True),
        StructField("task_scheduling_class", IntegerType(), True),
        StructField("priority", IntegerType(), True),
        StructField("cpu_request", FloatType(), True),
        StructField("memeory_request", FloatType(), True),
        StructField("disk_space_request", FloatType(), True),
        StructField("machine_restrictions", BooleanType(), True),
    ]
)
```

We load tasks_events into dataframe and preprocess the data to remove rows that have null values in columns of interest.

```
taskEventsDf = (
    spark.read.schema(taskEventsSchema)
        .csv("../data/task_events/*.csv.gz")
        .where(F.col("event_type").isNotNull() & F.col("priority").isNotNull())
)
```

```
task_Per_priority_Df = (
    taskEventsDf.select(
        F.concat_ws("_", F.col("job_id"), F.col("task_index")).alias("task_id"),
        "priority",
    )
        .groupBy("priority")
        .count()
        .withColumnRenamed("count", "number of tasks per priority")
        .sort("priority")
)
```

We compute the number of tasks under each certain priority level. To do it, we create the column which contains the specific the task_id by concatenating the column of job_id and task_index together. Then we groupBy priority and count the total number of task for each level of priority.

```
evicted_tasks_Per_priority_Df = (
    taskEventsDf.select(
        F.concat_ws("_", F.col("job_id"), F.col("task_index")).alias("task_id"),
        "event_type",
        "priority"
    )
        .where(F.col("event_type") == 2)
        .groupBy("priority")
        .count()
        .withColumnRenamed("count", "number of evicted tasks per priority")
        .sort("priority")
)
```

We compute the number of evicted tasks for each level of priority. To do it, we create the column which contains the specific the task_id by concatenating the column of job_id and task_index together. And then we filter the rows with $event\_type = 2$. Then we groupBy priority and count the total number of evicted task for each level of priority.

```
new = task_Per_priority_Df.join(
    evicted_tasks_Per_priority_Df, ["priority"]
).sort("priority")
```

We join the dataframes of task_Per_priority_D and evicted_tasks_Per_priority_D together and rearrange the order of rows by the column of priority.

```
@udf(returnType=FloatType())
def compute_Evicted_event_Per_priority(tasks, evicted_tasks):
    prob = evicted_tasks / tasks
    return prob


Evicted_task_Per_priority_probability_Df = (
    new.withColumn("probability of evicted_event",
                   compute_Evicted_event_Per_priority(F.col("number of tasks per priority"),
                                                      F.col("number of evicted tasks per
                                                          priority")))

).drop("number of tasks per priority", "number of evicted tasks per priority")
DataFrame_To_pandas_0 = Evicted_task_Per_priority_probability_Df.toPandas()
```

We define a user defined function to compute the probability of being evicted for each level of priority in the joining dataframe we just generate. Finnally, to visualize it, we convert it into pandas.

priority	probability of evicted_event
0	0.070198
1	0.040574
2	0.012956
3	0.042394
4	0.000941
6	0.000204
8	0.003642
9	0.003410
10	0.005256

Figure 3: Probability of being evicted for each level of priority

We can find that in tasks with lower priority have more chance to become being evicted except for the priority level of 4 and 6.

0.8 Question 6

0.8.1 In general, do tasks from the same job run on the same machine?

Since the question is asking whether tasks from the same job run on the same machine. So we will work the table of taskEvents. Sicne we have demonstrated the fact that the job is split into tasks for processing in previous question, this questions means we evaluate the presumption that a specific is processed on a single machine.

```
taskEventsSchema = StructType(
    [
        StructField("timestamp", IntegerType(), True),
        StructField("missing_info", StringType(), True),
        StructField("job_id", IntegerType(), True),
        StructField("task_index", IntegerType(), True),
        StructField("machine_id", IntegerType(), True),
        StructField("event_type", IntegerType(), True),
        StructField("user_name", StringType(), True),
        StructField("task_scheduling_class", LongType(), True),
        StructField("priority", IntegerType(), True),
        StructField("cpu_request", FloatType(), True),
```

```
        StructField("memeory_request", FloatType(), True),
        StructField("disk_space_request", FloatType(), True),
        StructField("machine_restrictions", BooleanType(), True),
    ]
)

taskEventsDf = (
    spark.read.schema(taskEventsSchema)
        .csv("../data/task_events/*.csv.gz")
        .where(F.col("machine_id").isNotNull() & F.col("job_id").isNotNull())
)
```

We load tasks_events into dataframe and preprocess the data to remove rows that have null values in columns of interest.

```
machine_job_Df = (
    taskEventsDf.select("machine_id", "job_id").distinct()
        .groupBy("job_id").count()
        .withColumnRenamed("count", "No_different_machine_ids_for_each_job_id")
        .sort("job_id")
)
DataFrame_To_pandas_0 = machine_job_Df.toPandas()
DataFrame_To_pandas_0.plot.bar(x="job_id", y="No_different_machine_ids_for_each_job_id")
```

We select the column of machine_id and job_id from taskEvenDf dataframe, then filter out the repeated rows using distinct(), then we groupBy job_id and count the total number of different machines operated for each specific job_id. To visualize it, we convert it into pandas and plot out.

```
job_on_single_machine_Df = (
    machine_job_Df.where(F.col("No_different_machine_ids_for_each_job_id")==1)
)

print('percentage of jobs only run on single machine:',
     job_on_single_machine_Df.count()/machine_job_Df.count()*100)

frame = plt.gca()
# hide x-axis
frame.axes.get_xaxis().set_visible(False)

plt.show()
```

We do count the percentage of jobs who have only single machine to process as well.

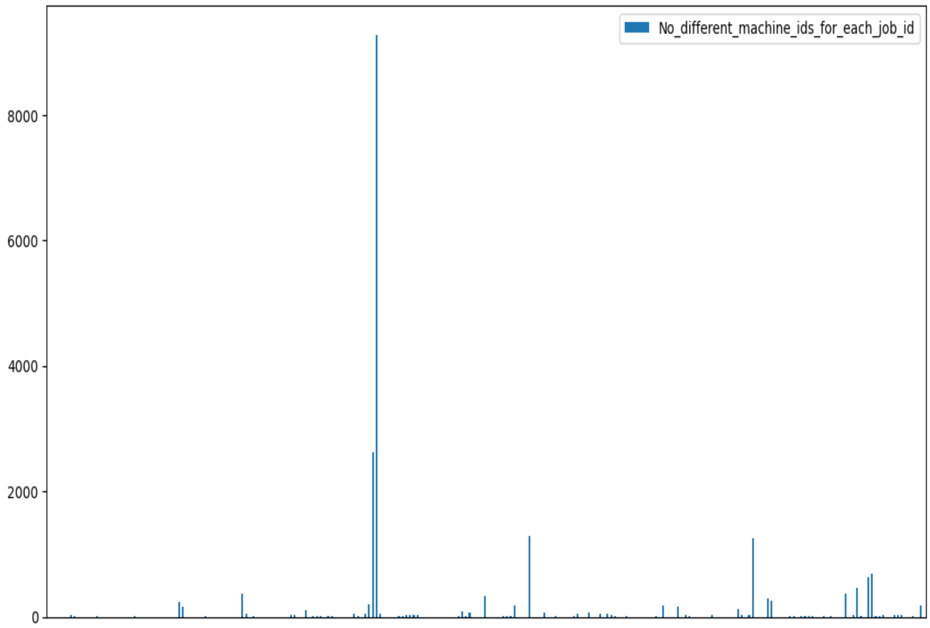


Figure 4: Caption

The percentage of jobs only run on single machine: 16.95%. It means for most of jobs, it would be split into tasks and then distribute those tasks to different machines.

0.9 Question 7

0.9.1 Are the tasks that request the more resources the one that consume the more resources?

In the context of question, we need to compare the resources requested and consumed in reality for the task. And since the resource requests represent the maximum amount of CPU, memory, or disk space a task is permitted to use. So we need to cite the tables "TaskEvent" and "TaskUsage".

In the table of "TaskEvent" which contains request information, that are "CPU request","memory request" and "disk space request". However, in the table of "TaskUsage", the information on resources consumption only contains "cpu_rate" and "canonical_memory_usage".

So, we will select "CPU request" and "memory request" from table of "TaskEvent" and select "cpu_rate" and "canonical_memory_usage" from table of "TaskUsage".

```
# Create the taskUsageSchema
taskUsageSchema = StructType(
    [
       StructField("start_time", LongType(), True),
       StructField("end_time", LongType(), True),
       StructField("job_id", StringType(), True),
       StructField("task_index", StringType(), True),
       StructField("machine_id", StringType(), True),
       StructField("cpu_rate", FloatType(), True),
       StructField("canonical_memory_usage", FloatType(), True),
       StructField("assigned memory usage", FloatType(), True),
       StructField("unmapped page cache", FloatType(), True),
       StructField("total page cache", FloatType(), True),
       StructField("maximum memory usage", FloatType(), True),
```

```
        StructField("disk I/O time", FloatType(), True),
        StructField("local disk space usage", FloatType(), True),
        StructField("maximum CPU rate", FloatType(), True),
        StructField("maximum disk IO time", FloatType(), True),
        StructField("cycles per instruction", FloatType(), True),
        StructField("memory accesses per instruction", FloatType(), True),
        StructField("sample portion", FloatType(), True),
        StructField("aggregation type", BooleanType, True),
        StructField("sampled CPU usage", FloatType, True)
    ]
)

# 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),
    ]
)

taskEventsDf = (
    spark.read.schema(taskEventsSchema)
    .csv("../data/task_events/*.csv.gz")
    .where(F.col("cpu_request").isNotNull() & F.col("memeory_request").isNotNull()) )
```

Load tasks_events into dataframe and preprocess the data to remove rows that have null values in columns of interest.

```
taskUsageDf = (
    spark.read.schema(taskUsageSchema)
    .csv("../data/task_usage/*.csv.gz")
    .where(F.col("cpu_rate").isNotNull() & F.col("canonical_memory_usage").isNotNull()))
```

Load task_usage into dataframe and preprocess the data to remove rows that have null values in columns of interest.

```
resourcesRequestedPerTaskDf = (
    taskEventsDf.select(
        F.concat_ws("_", F.col("job_id"), F.col("task_index")).alias("task_id"),
        "cpu_request",
        "memeory_request",
    )
    .groupBy("task_id")
    .agg(
        F.max("cpu_request").alias("cpu_request"),
        F.max("memeory_request").alias("memeory_request") ) ).distinct()
```

Since we need to compare the resources requested and consumed in reality in terms of the task. However, the task_id is not existing in tables. Since the work arrives at a cell in the form of jobs and a job is comprised of one or more tasks, each of which is accompanied by a set of resource requirements used for scheduling (packing) the tasks onto machines. We can obtain the information of task_id by concatenating the “job_id” and “task_index” tht can form an unique identifier for the task_id.

To get the resources requested by each task. We use the dataframe corresponding to "task_events" we again derive a new column "task_id" from "job_id" & "task_index" columns and select "cpu_request" and "memory_request" columns.

Since the resource requests represent the maximum amount of CPU, memory, or disk space a task is permitted to use, we need to take the maximum value for each unique "task_id", we can apply groupBy function with respect to "task_id" in collaboration of aggregation function by which we perform pyspark.sql.function.max() to extract the maximum value in "cpu_request" and "memory_request" for each unique "task_id".

Finally, we call distinct() on the dataframe to remove duplicate rows.

```
resourcesConsumedPerTaskDf = (
    taskUsageDf.select(
        F.concat_ws("_", F.col("job_id"), F.col("task_index")).alias("task_id"),
        "cpu_rate",
        "canonical_memory_usage",
    )
    .groupBy("task_id")
    .agg(
        F.sum("cpu_rate").alias("total_cpu_usage"),
        F.sum("canonical_memory_usage").alias("total_memory_usage"), ) ).distinct()
```

To get the resources consumed by each task, it is very similar to the way to obtain the resources requested by each task.

Since the "CPU usage - sampled" field is only provided in version 2.1 and later and it is the average CPU usage during a one-second period that is picked uniformly at random from the 5-minute measurement period, we need to take the average value for each unique "task_id". we can apply groupBy function with respect to "task_id" in collaboration of aggregation function by which we perform pyspark.sql.function.sum() to extract the maximum value in "cpu_request" and "memory_request" for each unique "task_id", and we will take their averages in further processing step.

Up to this point, we have a dataframe that has columns "task_id", "total_cpu_usage" and "total_memory_usage.

Finally, we call distinct() on the dataframe to remove duplicate rows.

```
requestedConsumedDf = resourcesRequestedPerTaskDf.join(
    resourcesConsumedPerTaskDf, ["task_id"]
)
# Joining data
```

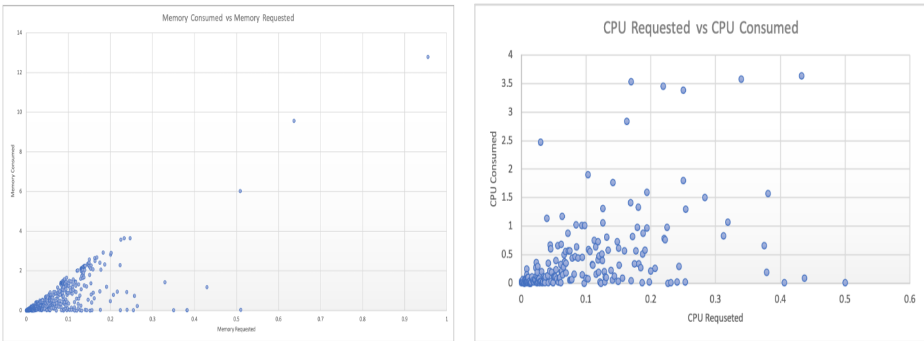
Next we joined the two dataframes in terms of "task_id" that hold the total resources consumed and the resources requested by each task.

```
memoryRequestedConsumedDf = (
    requestedConsumedDf.select("task_id", "memeory_request", "total_memory_usage")
    .groupBy("memeory_request")
    .agg(F.avg("total_memory_usage").alias("avg_memory_usage"))
    .sort("memeory_request") )
```

Here we generate the table of memoryRequestedConsumedDf based on the table of requestedConsumedDf. Since we need to observe the memory consumed that is expressed by "total_memory_usage" with respect to the the "memory request", at this point, the "task_id" has no point. So, we can perform groupBy function with respect to "memory request" in collaboration of aggregation function by which we perform pyspark.sql.function.avg() to take the mean value of memory usage per "memory request".

```
cpuRequestedConsumedDf = (
    requestedConsumedDf.select("task_id", "cpu_request", "total_cpu_usage")
    .groupBy("cpu_request")
    .agg(F.avg("total_cpu_usage").alias("avg_cpu_usage"))
    .sort("cpu_request") )
```

Here we generate the table of cpuRequestedConsumedDf in the same way as before.



(a) Memory consumed vs memory requested (b) CPU consumed vs CPU requested

Figure 5: Resources consumed vs Resources requested

0.10 Question 8

0.10.1 Is there a relation between the amount of resource consumed by tasks and their priority?

In Q8, we follow the same schema as Q7. In the context of question, we need to compare the resources consumed with their priority. In the table of "TaskEvents". there is "priority" information, and in the table of "TaskUsage", there is resources consumption information. Since the priority determines whether a task is scheduled on a machine. And tasks using more than their limit may be throttled (for resources like CPU) or killed, if this happens, one or more low priority task(s) may be killed. It means the priority could be reflected by CPU performance in some way. So, we take the "cpu_rate" from the table of TaskUsage.

```
taskEventsDf = (
    spark.read.schema(taskEventsSchema)
    .csv("../data/task_events/*.csv.gz")
    .where(F.col("priority").isNotNull())
)

taskUsageDf = (
    spark.read.schema(taskUsageSchema)
    .csv("../data/task_usage/*.csv.gz")
    .where(F.col("cpu_rate").isNotNull())
)
```

We load tasks_events and task_usage into dataframes and preprocess the data to remove rows that have null valuesin columns of interest.

```
priorityPerTaskDf = (
    taskEventsDf.select(
        F.concat_ws("_", F.col("job_id"), F.col("task_index")).alias("task_id"),
        "priority",
    )
    .groupBy("task_id")
    .agg(F.max("priority").alias("priority")) )
```

We can apply groupBy function with respect to "task_id" in collaboration of aggregation function by which we perform pyspark.sql.function.max() to extract the maximum value in "priority" for each unique "task_id" based on the table of "TaskEvents".

```
cpuConsumedPerTaskDf = (
    taskUsageDf.select(
```

```
        F.concat_ws("_", F.col("job_id"), F.col("task_index")).alias("task_id"),
        "cpu_rate",
    )
    .groupBy("task_id")
    .agg(
        F.sum("cpu_rate").alias("total_cpu_usage"),
    )
)
```

We can apply groupBy function with respect to "task_id" in collaboration of aggregation function by which we perform pyspark.sql.function.sum() to extract the total value in "cpu_rate" for each unique "task_id" to obtain "total_cpu_usage" based on the table of "TaskUasge".

```
priorityCpuConsumedDf = (
    priorityPerTaskDf.join(cpuConsumedPerTaskDf, ["task_id"])
    .groupBy("priority")
    .agg(F.mean("total_cpu_usage").alias("avg_cpu_consumption"))
    .sort("priority") )
```

Next we joined the two dataframes in terms of "task_id" that hold the total resources consumed and the priority by each task.

Since we need to observe the variation of esource consumed with respect to the "priority", we can apply groupBy function with respect to "task_id" in collaboration of aggregation function by which we perform pyspark.sql.function.mean() to extract the mean value in "total_cpu_usage". Finally we sort the table to be in ordered format in terms of the "priority".

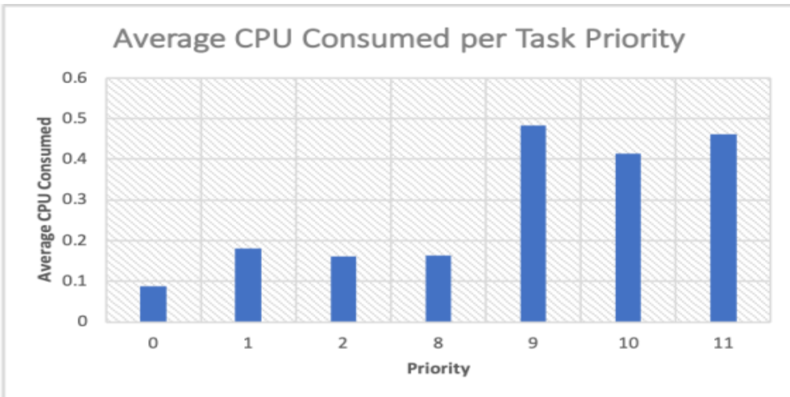


Figure 6: Average cpu vs consumed per task