**Starbucks Caffeine to Calorie Analysis**

Tasks done:

1. Retrieved a dataset from Kaggle.
2. Load chosen dataset into Hive. Then, performed HiveQL queries for getting
   1. Average calories by categories
   2. Calorie-to-Caffeine Ratio by Beverage Preparation
3. Converted the Hive queries into RDD MapReduce programs using Spark RDD API using map, filter and reduceByKey functions.

**RDD Programming**

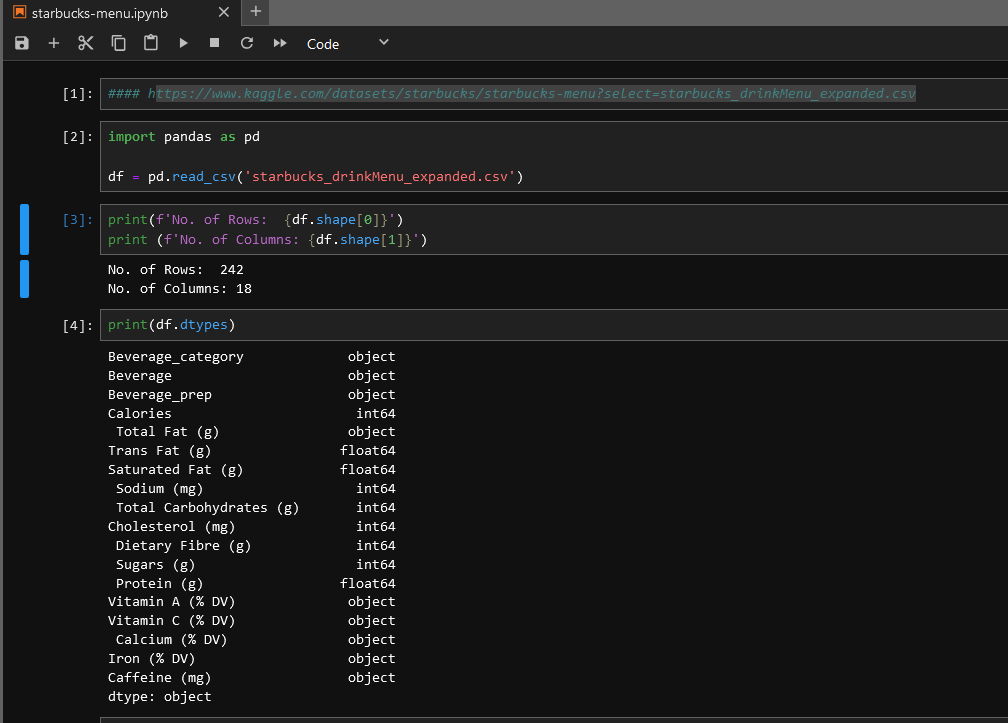
1. **Getting data from Kaggle**

* We are using Nutrition facts for Starbucks Menu from Kaggle:

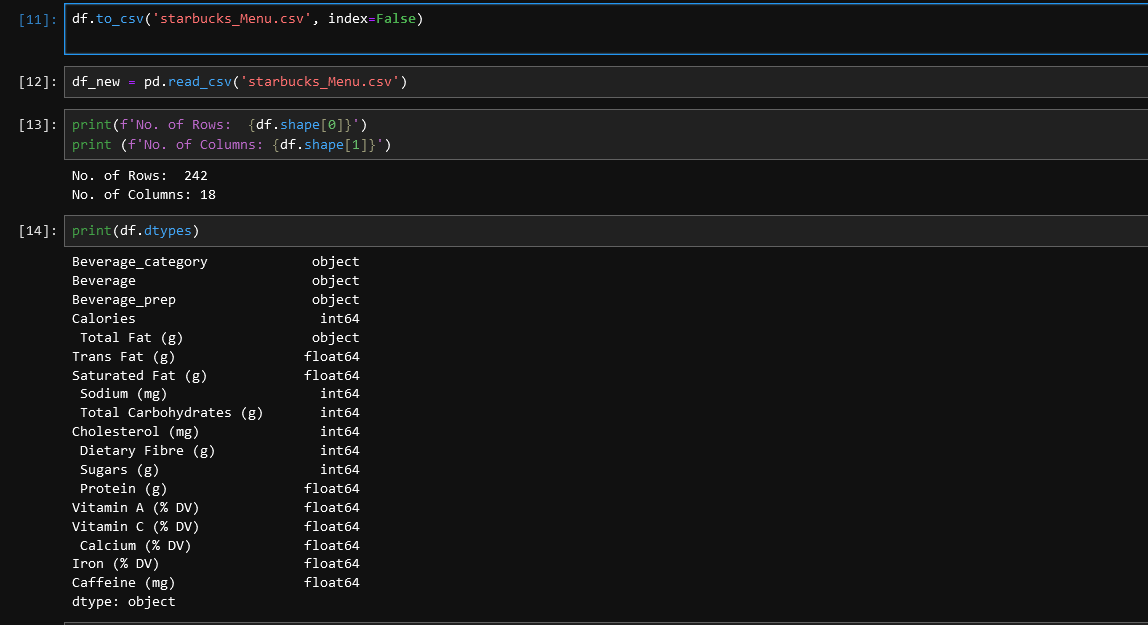
<https://www.kaggle.com/datasets/starbucks/starbucks-menu?select=starbucks_drinkMenu_expanded.csv>

The dataset has the nutritional information for Starbucks’ drink items. We will be looking into the calorie and caffeine content of the drinks.

We use Jupyter Notebook for data exploration and cleaning.



* We have performed some data cleaning and exported the clean dataset to a new csv file.

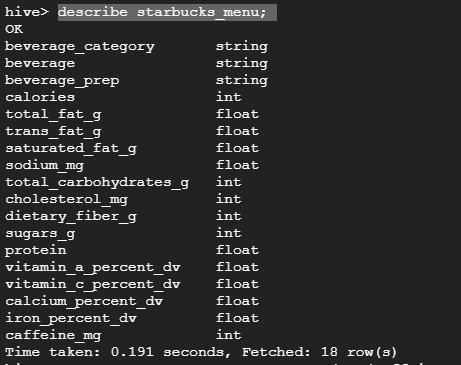


* We load the cleaned csv file to our GCP cluster:



1. **Loading data to Hive and analysis**

From our cluster we have copied our data to Hadoop file system and created a table in Hive. Table name is **starbucks\_menu**.



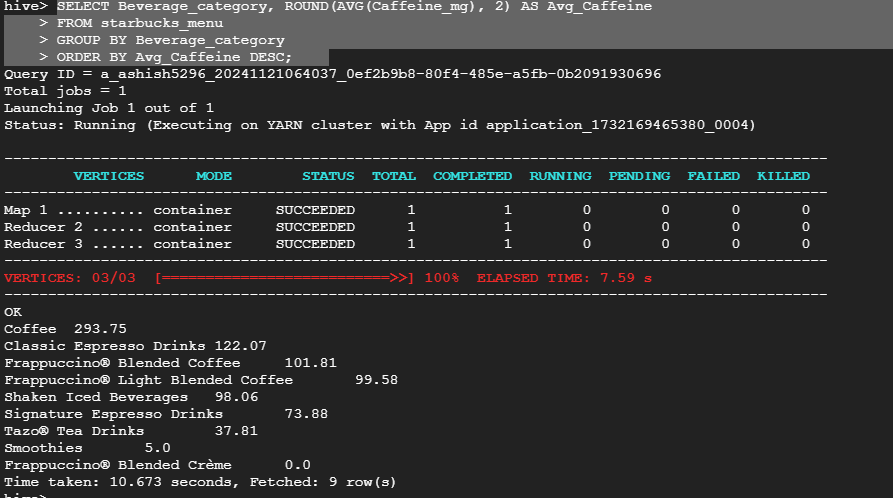
Then we run the following query:

* **Query 1: Average calorie by category**

SELECT Beverage\_category, ROUND(AVG(Caffeine\_mg), 2) AS Avg\_Caffeine  
FROM starbucks\_menu

GROUP BY Beverage\_category

ORDER BY Avg\_Caffeine DESC;



* **Query 2: Calorie-to-Caffeine Ratio by Beverage Preparation**

*The goal is to find out which preparation style offers the "best energy boost"*

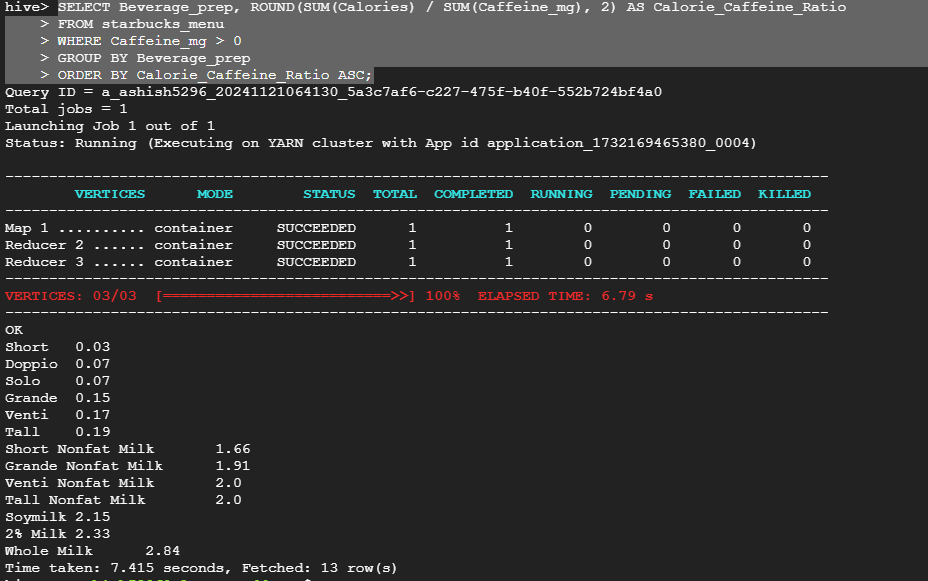
SELECT Beverage\_prep, ROUND(SUM(Calories) / SUM(Caffeine\_mg), 2) AS Calorie\_Caffeine\_Ratio

FROM starbucks\_menu

WHERE Caffeine\_mg > 0

GROUP BY Beverage\_prep

ORDER BY Calorie\_Caffeine\_Ratio ASC;



We can see that a Short has **3 calories per 100 mg of caffeine**.

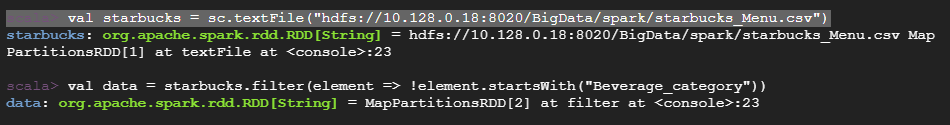
1. **Loading data to Spark and analysis**

* Loading the data from hdfs into spark:

val starbucks = sc.textFile("hdfs://10.128.0.18:8020/BigData/spark/starbucks\_Menu.csv")

* Removing the first row with headers

val data = starbucks.filter(element => !element.startsWith("Beverage\_category"))



* **For Query 1: Calculating Average Caffeine Content by Beverage Category**

// Split row line into array of strings using “,”

val splitData = data.map(line => line.split(","))

// Map each row to a tuple

val categoryCaffeine = splitData.map(fields => (fields(0), fields(17).toInt))

// Computing total caffine and count per category

val totalCaffeineAndCount = categoryCaffeine

.mapValues(caffeine => (caffeine, 1))

.reduceByKey((a, b) => (a.\_1 + b.\_1, a.\_2 + b.\_2))

// Calculating the average

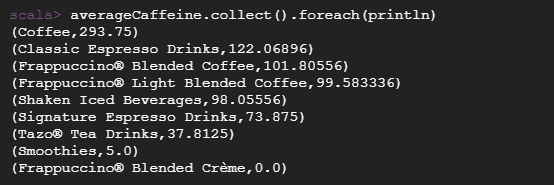
val averageCaffeine = totalCaffeineAndCount

.mapValues { case (totalCaffeine, count) => totalCaffeine.toFloat / count }

.sortBy(\_.\_2, ascending = false)

// Printing the results

averageCaffeine.collect().foreach(println)

filter 

* **For Query 2: Calculating the Calorie-to-Caffeine ratio**

Since we have already loaded our data to the memory we don’t have to do that step again, and we can take “data” value and filter required variables by performing split there

// filtering Rows with valid caffeine values

val filteredForRatio = data.map(line => line.split(",")).filter(fields => fields(17).toInt > 0)

// mapping to preparation (calories, caffeine)

val prepCaloriesCaffeine = filteredForRatio.map(fields => (fields(2), (fields(3).toInt, fields(17).toInt)))

// aggregating calories and caffeine per preparation style

val totalCaloriesAndCaffeine = prepCaloriesCaffeine

.reduceByKey((a, b) => (a.\_1 + b.\_1, a.\_2 + b.\_2))

// calculating the ration

val ratioByPrep = totalCaloriesAndCaffeine

.mapValues { case (totalCalories, totalCaffeine) => totalCalories.toFloat / totalCaffeine }

.sortBy(\_.\_2)

// printing the results

ratioByPrep.collect().foreach(println)

