ΕΘΝΙΚΟ ΜΕΤΣΟΒΙΟ ΠΟΛΥΤΕΧΝΕΙΟ

ΣΧΟΛΗ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ ΚΑΙ ΜΗΧΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ

ΜΑΘΗΜΑ: **Προχωρημένα Θέματα Βάσεων Δεδομένων**
ΕΞΑΜΗΝΟ ΜΑΘΗΜΑΤΟΣ: **9ο**

Όνομα: Αποστόλου Αθανάσιος
Εξάμηνο: 15ο
Αριθμός Μητρώου: 03112910

Εξαμηνιαία Εργασία

Θέμα 1ο
Υλοποίηση SQL ερωτημάτων για αναλυτική επεξεργασία δεδομένων

Διάλεξα να υλοποιήσω το 1ο θέμα από την εξαμηνιαία εργασία.

Αρχικοποίηση

Έχουμε φτιάξει τα VMs στο https://cyclades.okeanos-knossos.grnet.gr και τα έχουμε εγκαταστήσει java, python, hadoop 2.7.7 και spark 2.4.4 σύμφωνα με τις οδηγίες.

Η public ip του master είναι 83.212.76.14
Μπορούμε να τσεκάρουμε το hdfs από την διεύθυνση http://83.212.76.14:50070/
και το spark από την διεύθυνση http://83.212.76.14:8080/

Εκτέλεση

Αρχικά, έχουμε κατεβάσει το αρχείο yellow_trip_data.zip στον master και το έχουμε κάνει extract στον φάκελο που δουλεύουμε.
Τοποθετούμε τα csv αρχεία μας στο hdfs με τις εντολές:

```
hdfs dfs -put ./yellow_tripdata_1m.csv /
hdfs dfs -put ./yellow_tripvendors_1m.csv /
```

Δημιουργούμε 6 αρχεία python ως εξής:
Q1-MR.py
Q1-SQL.py
Q1-SQL-PARQUET.py
Q2-MR.py
Q2-SQL.py
Q2-SQL-PARQUET.py

που αντιστοιχούν στις μεθόδους Map Reduce με RDD API, SparkSQL σε αρχεία κειμένου και SparkSQL σε αρχεία parquet για τα ερωτήματα Q1 και Q2 αντίστοιχα. Κάθε script το τρέχουμε με την εντολή:

```
time spark-submit ./src/Q1-MR.py 2>&1 | tee ./logs/Q1-MR/master.log
```

όπου το όνομα αλλάζει για κάθε script. Έτσι μπορούμε να μετράμε τον συνολικό χρόνο εκτέλεσης καθώς και να αποθηκεύουμε το output του master.
Κάθε script τυπώνει το αποτέλεσμα στο τέλος στην γραμμή εντολών και το γράφει και πάνω στο hdfs (π.χ. για το Q1-MR.py το γράφει στο αρχείο /Q1-MR-out). Τα αποτελέσματα τα παίρνουμε στο directory **output** στο τοπικό σύστημα αρχείων με τις εντολές:

```
hdfs dfs -get ./Q1-MR-out/ ./output
hdfs dfs -text ./Q1-MR-out/* > ./output/Q1-MR-out.txt
```

Όπου η 1η εντολή παίρνει τα αρχεία όπως είναι σπασμένα στο hdfs σύστημα αρχείων μέσα στον φάκελό τους, ενώ η 2η εντολή τα ενώνει και τα δείχνει σαν τελικό αρχείο το οποίο σώζουμε στο **Q1-MR-out.txt.** Παρόμοια δουλεύουμε και για τα άλλα ονόματα αρχείων.

Από τον slave παίρνουμε τα logs των workers από τον φάκελο **/home/user/spark-2.4.4-bin-hadoop2.7/work/** και τα τοποθετούμε στον φάκελο logs.

Έχουμε δημιουργήσει επίσης το script **pqrquet.py** το οποίο κάνει μετατροπή των csv αρχείων σε parquet και τα γράφει πάνω στο hdfs.

Για ευκολία (και backup) όλα τα παραπάνω προστίθενται στο git repository μου https://gitlab.com/ThanosApostolou/db-project

Για το ερώτημα Q2 με parquet αρχεία αναγκαστήκαμε να αλλάξουμε την μνήμη των workers σε 768 MB γιατί ορισμένες φορές (όχι πάντα) παρουσίαζε out of memory error.

Μεθοδολογία MapReduce

Τα δεδομένα μας είναι της μορφής
yellow_tripdata_1m.csv:
{trip_id, datetime_start, datetime_finish, longitude_start, latitude_start, longitude_finish, latitude_finish, cost}
yellow_tripvendors_1m.csv:
{trip_id,vendor_id}

Παρουσιάζονται σε ψευδοκώδικα οι MapReduce διαδικασίες για κάθε query ξεχωριστά

Για το Q1:
Ποια είναι η μέση διάρκεια διαδρομής (σε λεπτά) ανά ώρα έναρξης της διαδρομής; Ταξινομείστε το αποτέλεσμα με βάση την ώρα έναρξης σε αύξουσα σειρά:

Φτιάχνουμε την MapReduce διαδικασία:

MapReduce1: Παίρνουμε τις χρήσιμες πληροφορίες από κάθε αρχείο και παρουσιάζουμε το αποτέλεσμα ταξινομημένο ως προς starthour

```
# όπου tripdata δίνουμε το αρχείο yellow_tripdata_1m.csv
map (fileid, tripdata) :
    foreach line in tripdata:
        # φτιάχνουμε λίστα από την κάθε γραμμή του csv και παίνουμε τις πληροφορίες που χρειαζόμαστε
        # κάνουμε πάντα τις αντίστοιχες μετατροπές τύπων γιατί όλα διαβάζονται ως strings αρχικά
        record = line.split(“,”)
        datetime_start = datetime(record[1])
        datetime_finish = datetime(record[2])
        starthour =  hour(time(datetime_start))
        # υπολογίζουμε την διάρκεια της κάθε διαδρομής
        duration = (datetime_finish – datetime_start) in minutes
        emit (starthour, duration)
        # στέλνουμε ως key την ώρα εκίνησης και ως value την διάρκεια
        # η map πάντα ταξινομεί τα αποτελέσματα ως προς το key, δηλαδή το starthour

# η duration_list έχει όλα τα duration των trip με ίδιο starthour
reduce(starthour, duration_list):
```

```
    sum=0
    count=0
    # υπολογίζουμε το άθροισμα όλων των duration και το πλήθος τους
    foreach d in duration_list:
        sum += d
        count++
    # υπολογίζουμε τον μέσο όρο
    duration_avg = sum / count
    emit (starthout, duration_avg)
    # το αποτέλεσμα είναι ταξινομημένο ως προς το starthour λόγω της ιδιότητας του map να ταξινομεί ως προς το key
```

Για το Q2:
Ποιες είναι οι 5 πιο γρήγορες κούρσες που έγιναν μετά τις 10 Μαρτίου και σε ποιους vendors ανήκουν:

Θεωρούμε ότι όταν ζητάει τις κούρσες που έγιναν μετά τις 10 Μαρτίου εννοεί αυτές που ξεκίνησαν μετά τις 10 Μαρτίου. Αν εννοούσε αυτές που τελείωσαν μετά τις 10 Μαρτίου (ακόμα και αν ξεκίνησαν νωρίτερα) τότε απλά αντί για startdate υπολογίζουμε το finishdate παρομοίως και χρησιμοποιούμε αυτό.

Φτιάχνουμε 2 MapReduce διαδικασίες:

MapReduce1: Παίρνουμε τις χρήσιμες πληροφορίες από κάθε αρχείο και φτιάχνουμε το tripdata join tripvendors ως προς το trip_id

```
# όπου tripdata είναι το αρχείο yellow_tripdata_1m.csv
map (fileid, tripdata) :
    # ορίζουμε ένα tag που θα ξεχωρίζει τα δεδομένα
    tag=”trip”
    # φτιάχνουμε λίστα από την κάθε γραμμή του csv και παίνουμε τις πληροφορίες που χρειαζόμαστε
    # κάνουμε πάντα τις αντίστοιχες μετατροπές τύπων γιατί όλα διαβάζονται ως strings αρχικά
    foreach line in tripdata:
        record = line.split(“,”)
        datetime_start = datetime(record[1])
        datetime_finish = datetime(record[2])
        startdate= date(datetime_start)
        # αν η κούρσα έγινε μετά τι; 10 μαρτίου
        if startdate > 10 Μαρτίου :
            # παίρνουμε τα υπόλοιπα στοιχεία και
            # υπολογίζουμε την ταχύτητα του trip ως απόσταση προς διάρκεια βάσει των τύπων
            trip_id = record[0]
            duration = (datetime_finish – datetime_start) in minutes
            φ1 = double(record[4])
            φ2 = double(record[6])
            Δφ = φ2 – φ1
            Δλ = double(record[5]) - double(record[3])
            a = sin(Δφ/2)^2 +cos(φ1) cos(φ2) sin(Δλ/2)^2
            c = 2 atan2 (sqrt(a),sqrt(1-a))
            d = 6371 c
            speed = d / duration
            # φτιάχνουμε την λίστα info με πρόθεμα το tag και μετά τις πληροφορίες που θέλουμε (δηλαδή το speed)
            # και την στέλνουμε ως value με κλειδί το trip_id
            info = [tag, speed]
            emit (trip_id, info)

# tripvendors είναι το αρχείο yellow_tripvendors_1m.csv
map (fileid, tripvendors) :
    # ορίζουμε ένα tag που θα ξεχωρίζει τα δεδομένα
    tag=”vendor”
    # φτιάχνουμε λίστα από την κάθε γραμμή του csv και παίνουμε τις πληροφορίες που χρειαζόμαστε
```

```
    # κάνουμε πάντα τις αντίστοιχες μετατροπές τύπων γιατί όλα διαβάζονται ως strings αρχικά
    foreach line in tripvendors:
        record = line.split(“,”)
        trip_id = record[0]
        vendor_id = record[1]
        info =[tag, vendor_id]
        emit (trip_id, info)
        # στέλνουμε ως κλειδί το trip_id και ως value το info

# η LIST_V είναι η λίστα που περιέχεις λίστες τόσο με trip info αλλά και vendor info
# που έχουν το ίδιο trip_id
reduce (trip_id, LIST_V) :
    table_id = 1
    table_list = []
    foreach i in LIST_V with i[0]==”trip” :          # i λίστα με tag “trip”
        foreach j in LIST_V with j[0]==”vendor” :    # j λίστα με tag “vendor”
            # φτιάχνουμε το record ως merge του key και των λιστών χωρίς τα tags
            # και το βάζουμε στο table_list
            record = [trip_id] merge i.tail() merge j.tail()
            table_list.append(record)
    emit (table_id, table_list)
# φτιάξαμε το join των 2 προηγούμενων map.
# το table_id είναι απλά κάποιο τυχαίο id για τον νέο πίνακα που φτιάξαμε, θα μπορούσμε να χρησιμοποιήσουμε και null
# Το table_list είναι λίστα που περιέχει όλα τα record
# Τα record είναι λίστα της μορφής [trip_id, speed, vendor_id]
```

MapReduce2: Κάνουμε ταξινόμηση του καινούριου πίνακα και παίρνουμε τα 5 γρηγορότερα trips

```
# η map εκτελείται στο αποτέλεσμα της προηγούμενης reduce
map (table_id, table_list) :
    foreach record in table_list:
        trip_id = record[0]
        speed = record[1]
        neg_speed = - speed
        vendor_id = record[2]
        info = [trip_id, vendor_id]
        emit (neg_speed, info)
# Για κάθε record στέλνω ως κλειδί την αρνητική ταχύτητα και ως value τις λίστα με τις υπόλοιπες πληροφορίες
# Εκμεταλεύομαι ότι η map πάντα ταξινομεί τα αποτελέσματα ως προς το key πριν τα στείλει στην reduce
# Στέλνουμε αρνητική ταχύτητα για να ταξινομηθούν κατά φθίνουσα σειρά

# εκτελείται στην προηγούμενη map
reduce (neg_speed, info_list) :
    count = 0
    while (count < 5) :
        # η foreach χρειάζεται στην περίπτωση που έχουμε πολλαπλές διαδρομές με ακριβώς ίδια ταχύτητα
        foreach info in info_list:
            speed = - neg_speed
            trip_id = info[0]
            vendor_id = info[1]
            emit (trip_id, (vendor_id, speed))
            count++
# Τα στοιχεία είναι ταξινομημένα με φθίνουσα σειρά
# Στέλνουμε τα 5 πρώτα στοιχεία σε όποια μορφή θέλουμε
# εδώ επιλέξαμε ως key το trip_id και ως value την τούπλα (vendor_id, speed)
```

Σημειώνεται ότι με το spark rdd api θα χρησιμοποιήσουμε μεθόδους όπως filter(), sort(), limit() οπότε η υλοποίηση σε κώδικα θα διαφέρει αρκετά από τους παραπάνω ψευδοκώδικες. Φυσικά με το Spark SQL api χρησιμοποιούμε ακόμα πιο high level λογική οπότε οι ομοιότητες είναι λίγες.

Αποτελέσματα

Τα αποτελέσματα από τα ερωτήματα Q1 και Q2 με τις διάφορες μεθόδους βρίσκονται στο **output.zip.** Τα εναποθέτω και παρακάτω:

Q1-MR-out.txt

HourOfDay	*AverageTripDuration*
00	*14.0177937374*
01	*13.9750698989*
02	*13.0356355927*
03	*13.3222825205*
04	*13.7998579311*
05	*13.2755832212*
06	*12.4874202376*
07	*13.3950064185*
08	*14.6275045434*
09	*14.6701064198*
10	*14.6579391697*
11	*14.9358212219*
12	*15.1308813229*
13	*15.5539187332*
14	*16.5231383808*
15	*30.2234986321*
16	*17.2130720694*
17	*16.5108256544*
18	*15.2904537486*
19	*14.221208806*
20	*13.5758996364*
21	*13.5108553274*
22	*14.2317976256*
23	*13.9584711252*

Q1-SQL-out.txt

00	*14.014732218495666*
01	*13.97102690299322*
02	*13.031247770806884*
03	*13.198687908261553*
04	*12.75128390710552*
05	*13.274742036787812*
06	*12.487023011525073*
07	*13.395006418527371*
08	*14.626919176562348*
09	*14.668773991637424*
10	*14.657253383754934*
11	*14.93487388589339*
12	*15.129612377454897*
13	*15.552627873251884*
14	*16.5220609090768*
15	*30.21154817908388*
16	*17.211297781030606*
17	*16.509717117907787*
18	*15.289086069641007*
19	*14.219330238513901*
20	*13.574330702255745*
21	*13.509544933658104*
22	*14.23033952575753*
23	*13.956139401824828*

Q1-SQL-PARQUET-out.txt

00 14.014732218495363
01 13.971026902993286
02 13.031247770806862
03 13.19868790826156
04 12.751283907105702
05 13.27474203678772
06 12.487023011525174
07 13.395006418527243
08 14.626919176562463
09 14.668773991637512
10 14.657253383755128
11 14.934873885893436
12 15.129612377454777
13 15.552627873251815
14 16.52206090907684
15 30.21154817908411
16 17.211297781030705
17 16.509717117907883
18 15.289086069641018
19 14.219330238513681
20 13.574330702255642
21 13.509544933658093
22 14.230339525757714
23 13.956139401824828

Q2-MR-out.txt

Ride	*Speed*	*Vendor*
352187835406	*440082.563054*	*2*
352187723573	*428884.636875*	*2*
352187849372	*427312.299821*	*2*
326417878976	*403592.212801*	*2*
309237832112	*387160.858061*	*2*

Q2-SQL-out.txt

352187835406	*440082.5630542829*	*2*
352187723573	*428884.6368750969*	*2*
352187849372	*427312.29982091463*	*2*
326417878976	*403592.21280119184*	*2*
309237832112	*387160.85806124855*	*2*

Q2-SQL-PARQUET-out.txt

352187835406	*440082.5630542829*	*2*
352187723573	*428884.6368750969*	*2*
352187849372	*427312.29982091463*	*2*
326417878976	*403592.21280119184*	*2*
309237832112	*387160.85806124855*	*2*

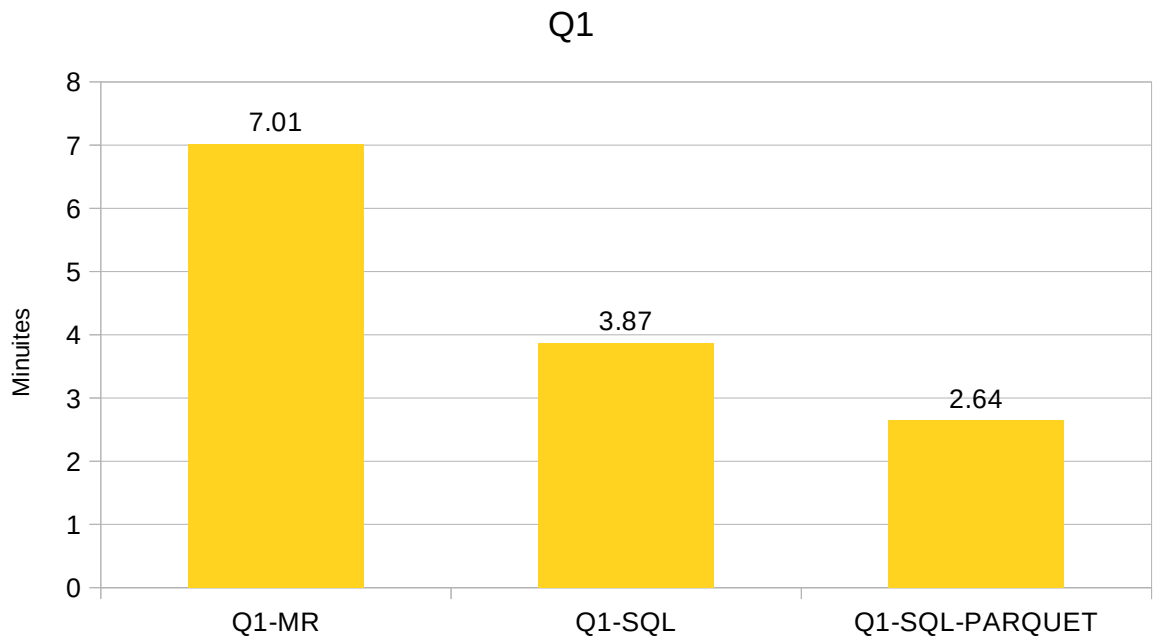
Χρόνοι

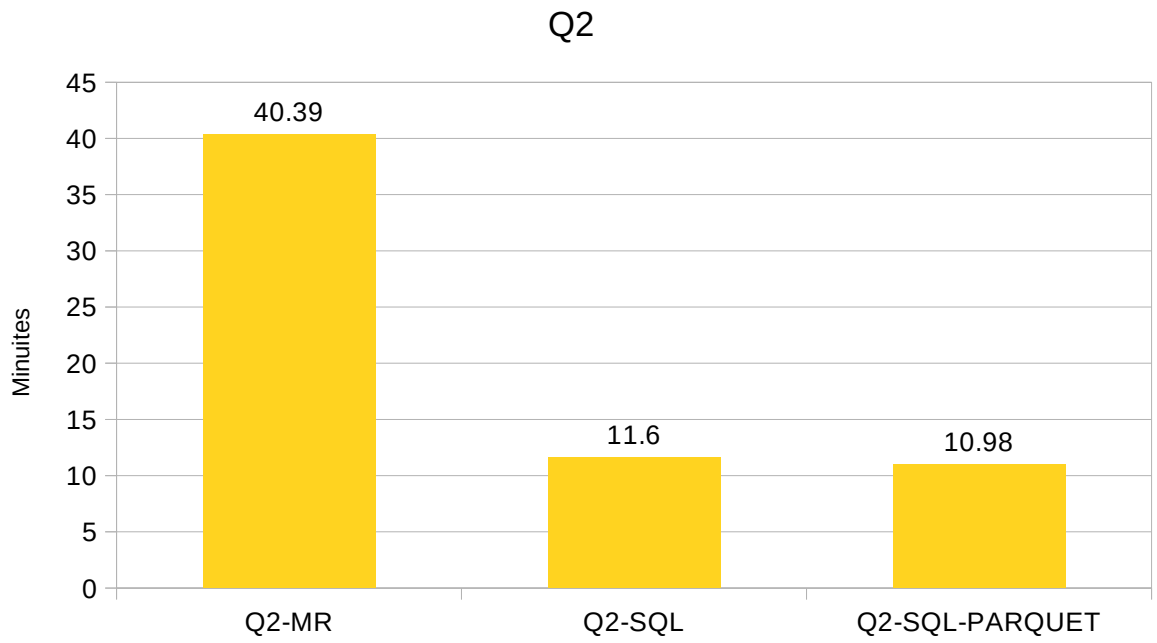
Έχουμε μετρήσει τους παρακάτω χρόνους από τις εκτελέσεις. Θυμίζουμε ότι το parquet.py που κάνει το transformation εκτελείται μόνο μια φορά και αποθηκεύει τα parquet αρχεία στο hdfs. Οπότε τα Q1-SQL-PARQUET και τα Q2-SQL-PARQUET δεν εκτελούν κάποιο transformation αλλά διαβάζουν απευθείας τα parquet αρχεία. Όλα τα scripts τυπώνουν στην οθόνη το αποτέλεσμα αλλά και το γράφουν στο hdfs, οπότε οι χρόνοι είναι λίγο αυξημένοι.

Q1-MR	**7m0.561s = 7.01m**

Q1-SQL	**3m51.900s =3.87m**
Q1-SQL-PARQUET	**2m38.123s = 2.64m**
Q2-MR	**40m52.543s = 40.39m**
Q2-SQL	**11m35.924s = 11.60m**
Q2-SQL-PARQUET	**10m58.816s = 10,98m**
parquet transformation	**2m27.195s = 2.47m**

Φτιάχνουμε τα αντίστοιχα διαγράμματα για τα Q1 και Q2:





Βλέπουμε ότι αν έχουμε πολλά και δύσκολα ερωτήματα τότε συμφέρει να μετατρέπουμε τα δεδομένα σε μορφή **parquet**, αφού ο χρόνος που χάνουμε για την μετατροπή είναι λιγότερος από αυτόν που κερδίζουμε κατά την εκτέλεση.

Κώδικας

Τα scripts βρίσκονται στο **src.zip.** Τα εναποθέτω και εδώ:

parquet.py

```
from pyspark.sql import SparkSession
from pyspark.sql.types import *

if __name__ == "__main__":
    spark = SparkSession.builder.appName("parquet").getOrCreate()

    tschema = StructType([
        StructField("TripID", StringType(), False),
        StructField("StartDate", StringType(), True),
        StructField("FinishDate", StringType(), True),
        StructField("StartLongitude", StringType(), True),
        StructField("StartLatitude", StringType(), True),
        StructField("FinishLongitude", StringType(), True),
        StructField("FinishLatitude", StringType(), True),
        StructField("Cost", StringType(), True)
    ])
    tdf = spark.read.schema(tschema).csv("hdfs://master:9000/yellow_tripdata_1m.csv")
    tdf.write.mode("overwrite").parquet("hdfs://master:9000/yellow_tripdata_1m.parquet")

    vschema = StructType([
        StructField("TripID", StringType(), False),
        StructField("VendorID", StringType(), True)
    ])
    vdf = spark.read.schema(vschema).csv("hdfs://master:9000/yellow_tripvendors_1m.csv")
    vdf.write.mode("overwrite").parquet("hdfs://master:9000/yellow_tripvendors_1m.parquet")

    spark.stop()
```

Q1-MR.py

```
from __future__ import print_function
from pyspark import SparkContext
import sys
import datetime

def mapper (line) :
    line_list = line.split(",")
    start_date = line_list[1]
    sd = datetime.datetime.strptime(start_date, '%Y-%m-%d %H:%M:%S')
    finish_date = line_list[2]
    fd = datetime.datetime.strptime(finish_date, '%Y-%m-%d %H:%M:%S')
    duration = (fd - sd).total_seconds() / 60
    start_hour = start_date.split(" ")[1].split(":")[0]
    return (start_hour, (duration, 1))

def reducer (a, b) :
```

```
    return (a[0]+b[0],a[1]+b[1])

def mapper2 (line) :
    return ("" + line[0] + "\t\t" + str(line[1][0]/line[1][1]))

if __name__ == "__main__":
    sc = SparkContext(appName="Q1-MR")

    final_rdd = sc.parallelize(["HourOfDay\tAverageTripDuration"])
    text_file = sc.textFile("hdfs://master:9000/yellow_tripdata_1m.csv")
    mean_times = text_file.map(mapper).reduceByKey(reducer).sortByKey().map(mapper2)
    final_rdd = final_rdd.union(mean_times)

    final_rdd.saveAsTextFile("hdfs://master:9000/Q1-MR-out")
    for x in final_rdd.collect() :
        print (x)
```

Q1-SQL.py

```
from pyspark.sql import SparkSession
from pyspark.sql import Row
from pyspark.sql.types import *
from pyspark.sql.functions import *

if __name__ == "__main__":
    spark = SparkSession.builder.appName("Q1-SQL").getOrCreate()

    tschema = StructType([
        StructField("TripID", StringType(), False),
        StructField("StartDate", StringType(), True),
        StructField("FinishDate", StringType(), True),
        StructField("StartLongitude", StringType(), True),
        StructField("StartLatitude", StringType(), True),
        StructField("FinishLongitude", StringType(), True),
        StructField("FinishLatitude", StringType(), True),
        StructField("Cost", StringType(), True)
    ])
    trips = spark.read.schema(tschema).csv("hdfs://master:9000/yellow_tripdata_1m.csv")

    trips = trips.selectExpr("TripID",
            "date_format(cast(StartDate as Timestamp),'HH') as HourOfDay",
            "(cast(cast(FinishDate as Timestamp)as double) - cast(cast(StartDate as Timestamp)as double))/60 as
Duration")
    trips.createOrReplaceTempView("trips")

    result = spark.sql("SELECT HourOfDay, AVG(Duration) AS AverageTripDuration \
                    FROM trips \
                    GROUP BY HourOfDay \
                    ORDER BY HourOfDay")

    result.write.format("csv").mode("overwrite").options(delimiter='\t').save("hdfs://master:9000/Q1-SQL-out")
    result.show()
    spark.stop()
```

Q1-SQL-PARQUET.py

```
from pyspark.sql import SparkSession
from pyspark.sql import Row
from pyspark.sql.types import *
from pyspark.sql.functions import *

if __name__ == "__main__":
    spark = SparkSession.builder.appName("Q1-SQL-PARQUET").getOrCreate()
```

```
trips = spark.read.parquet("hdfs://master:9000/yellow_tripdata_1m.parquet")

trips = trips.selectExpr("TripID",
        "date_format(cast(StartDate as Timestamp),'HH') as HourOfDay",
        "(cast(cast(FinishDate as Timestamp)as double) - cast(cast(StartDate as Timestamp)as double))/60 as
Duration")
trips.createOrReplaceTempView("trips")

result = spark.sql("SELECT HourOfDay, AVG(Duration) AS AverageTripDuration \
                FROM trips \
                GROUP BY HourOfDay \
                ORDER BY HourOfDay")

result.write.format("csv").mode("overwrite").options(delimiter='\t').save("hdfs://master:9000/Q1-SQL-PARQUET-
out")
result.show()
spark.stop()
```

Q2-MR.py

```
from __future__ import print_function
from pyspark import SparkContext
import sys
import datetime
import math

def filter_date (line) :
    line_list = line.split(",")
    start_datetime = line_list[1]
    sd = datetime.datetime.strptime(start_datetime, '%Y-%m-%d %H:%M:%S')
    finish_datetime = line_list[2]
    fd = datetime.datetime.strptime(finish_datetime, '%Y-%m-%d %H:%M:%S')
    start_date = sd.date()
    the_date = datetime.datetime.strptime("2015-03-10", '%Y-%m-%d')
    the_date = the_date.date()
    return (start_date > the_date)

def map_ride_speed (line) :
    line_list = line.split(",")
    ride_id = line_list[0]
    start_datetime = line_list[1]
    sd = datetime.datetime.strptime(start_datetime, '%Y-%m-%d %H:%M:%S')
    finish_datetime = line_list[2]
    fd = datetime.datetime.strptime(finish_datetime, '%Y-%m-%d %H:%M:%S')
    duration = (fd - sd).total_seconds() / 60

    l_start = float(line_list[3])
    f_start = float(line_list[4])
    l_finish = float(line_list[5])
    f_finish = float(line_list[6])
    Df = f_finish - f_start
    Dl = l_finish - l_start
    a = math.sin(Df/2)**2 + math.cos(f_start) * math.cos(f_finish) * (math.sin(Dl/2)**2)
    c = math.atan2(math.sqrt(a), math.sqrt(1-a))
    R = 6371
    distance = R * c
    if (duration == 0) :
        speed = 0
    else :
        speed = distance / duration
    return (ride_id, speed)

def map_vendor_data (line) :
```

```
    line_list = line.split(",")
    ride_id = line_list[0]
    vendor = line_list[1]
    return (ride_id, vendor)

if __name__ == "__main__":
    sc = SparkContext(appName="Q2-MR")

    final_rdd = sc.parallelize(["Ride\t\tSpeed\t\tVendor"])
    faster_rides = sc.textFile("hdfs://master:9000/yellow_tripdata_1m.csv")
    faster_rides = faster_rides.filter(filter_date).map(map_ride_speed)
    faster_rides = faster_rides.sortBy(lambda line : line[1], False).take(5)
    rides_list = [ x[0] for x in faster_rides ]
    faster_rides = sc.parallelize(faster_rides)

    vendors = sc.textFile("hdfs://master:9000/yellow_tripvendors_1m.csv")
    vendors = vendors.map(map_vendor_data)
    vendors = vendors.filter(lambda line : line[0] in rides_list )

    faster_rides = faster_rides.leftOuterJoin(vendors).sortBy(lambda line : line[1][0], False)
    faster_rides = faster_rides.map(lambda line : line[0]+'\t'+ str(line[1][0])+'\t'+line[1][1])

    final_rdd = final_rdd.union(faster_rides)

    final_rdd.saveAsTextFile("hdfs://master:9000/Q2-MR-out")
    for line in final_rdd.collect() :
        print (line)
```

Q2-SQL.py

```
from pyspark.sql import SparkSession
from pyspark.sql.functions import udf
from pyspark.sql.types import *
import sys
import datetime
import math

def calculate_duration (sd, fd) :
    finish_datetime = datetime.datetime.strptime(fd, '%Y-%m-%d %H:%M:%S')
    start_datetime = datetime.datetime.strptime(sd, '%Y-%m-%d %H:%M:%S')
    duration=(finish_datetime - start_datetime).total_seconds() / 60
    return duration

def calculate_distance (ls, fs, lf, ff) :
    l_start = float(ls)
    f_start = float(fs)
    l_finish = float(lf)
    f_finish = float(ff)
    Df = f_finish - f_start
    Dl = l_finish - l_start
    a = math.pow(math.sin(Df/2),2) + math.cos(f_start) * math.cos(f_finish) * math.pow(math.sin(Dl/2),2)
    c = math.atan2(math.sqrt(a), math.sqrt(1-a))
    R = 6371
    distance = R * c
    return distance

def calculate_speed (sd, fd, ls, fs, lf, ff) :
    duration = calculate_duration(sd, fd)
    distance = calculate_distance(ls, fs, lf, ff)
    if (duration == 0) :
        speed = 0
    else :
        speed = distance / duration
```

```
    return speed

if __name__ == "__main__":
    spark = SparkSession.builder.appName("Q2-SQL").getOrCreate()

    tschema = StructType([
        StructField("TripID", StringType(), False),
        StructField("StartDate", StringType(), True),
        StructField("FinishDate", StringType(), True),
        StructField("StartLongitude", StringType(), True),
        StructField("StartLatitude", StringType(), True),
        StructField("FinishLongitude", StringType(), True),
        StructField("FinishLatitude", StringType(), True),
        StructField("Cost", StringType(), True)
    ])
    trips = spark.read.schema(tschema).csv("hdfs://master:9000/yellow_tripdata_1m.csv")
    vschema = StructType([
        StructField("TripID", StringType(), False),
        StructField("VendorID", StringType(), True)
    ])
    vendors = spark.read.schema(vschema).csv("hdfs://master:9000/yellow_tripvendors_1m.csv")

    speedUdf = udf(calculate_speed, DoubleType())
    trips = trips.filter("cast(StartDate as DATE) > cast('2015-03-10' as DATE)"). \
            withColumn("Speed", speedUdf("StartDate", "FinishDate", "StartLongitude",
                    "StartLatitude", "FinishLongitude", "FinishLatitude")). \
            select("TripID", "Speed").orderBy("Speed", ascending=False).limit(5). \
            join(vendors, "TripID", "left").orderBy("Speed", ascending=False)
    trips.write.format("csv").mode("overwrite").options(delimiter='\t').save("hdfs://master:9000/Q2-SQL-out")
    trips.show()
    spark.stop()
```

Q2-SQL-PARQUET.py

```
from pyspark.sql import SparkSession
from pyspark.sql.functions import udf
from pyspark.sql.types import *
import sys
import datetime
import math

def calculate_duration (sd, fd) :
    finish_datetime = datetime.datetime.strptime(fd, '%Y-%m-%d %H:%M:%S')
    start_datetime = datetime.datetime.strptime(sd, '%Y-%m-%d %H:%M:%S')
    duration=(finish_datetime - start_datetime).total_seconds() / 60
    return duration

def calculate_distance (ls, fs, lf, ff) :
    l_start = float(ls)
    f_start = float(fs)
    l_finish = float(lf)
    f_finish = float(ff)
    Df = f_finish - f_start
    Dl = l_finish - l_start
    a = math.pow(math.sin(Df/2),2) + math.cos(f_start) * math.cos(f_finish) * math.pow(math.sin(Dl/2),2)
    c = math.atan2(math.sqrt(a), math.sqrt(1-a))
    R = 6371
    distance = R * c
    return distance

def calculate_speed (sd, fd, ls, fs, lf, ff) :
    duration = calculate_duration(sd, fd)
    distance = calculate_distance(ls, fs, lf, ff)
```

```
    if (duration == 0) :
        speed = 0
    else :
        speed = distance / duration
    return speed

if __name__ == "__main__":
    spark = SparkSession.builder.appName("Q2-SQL-PARQUET").getOrCreate()

    trips = spark.read.parquet("hdfs://master:9000/yellow_tripdata_1m.parquet")
    vendors = spark.read.parquet("hdfs://master:9000/yellow_tripvendors_1m.parquet")

    speedUdf = udf(calculate_speed, DoubleType())
    trips = trips.filter("cast(StartDate as DATE) > cast('2015-03-10' as DATE)"). \
            withColumn("Speed", speedUdf("StartDate", "FinishDate", "StartLongitude",
                        "StartLatitude", "FinishLongitude", "FinishLatitude")). \
            select("TripID", "Speed").orderBy("Speed", ascending=False).limit(5). \
            join(vendors, "TripID", "left").orderBy("Speed", ascending=False)
    trips.write.mode("overwrite").csv("hdfs://master:9000/Q2-SQL-PARQUET-out")
    trips.show()
    spark.stop()
```