Universidade Nova de Lisboa

Faculdade de Ciências e Tecnologia

Processamento de Streams

Taxi trips from NYC - Spark Streaming

Autores:
Lucas Fischer
Joana Martins
Mário Gomes

Professor:
Nuno Preguiça

Maio de 2019

1 Introdução

Este projecto pretende responder um conjunto de questões sobre as viagens de Táxis em Nova Iorque, a partir do processamento do *dataset* ACM DEBS 2015 Grand Challenge [3], utilizando Spark Streaming. Este *dataset* contém os registos do conjunto de viagens de táxi em Nova Iorque que incluem, entre outras variáveis: o identificador do táxi (*medallion*), a data e hora de recolha e largada do passageiro (*pickup_datetime* e *dropoff_datetime*), distância e duração das viagens (*trip_time_in_secs* e *trip_distance*), latitude e longitude dos pontos de início e final de viagem (*pickup_longitude* e *pickup_latitude*, *dropoff_longitude* e *dropoff_latitude*) e o preço da viagem e a gorjeta (*fare_amount* and *tip_amount*.

2 System design

Este projecto apoia-se em duas *frameworks* principais: o Apache Spark [2] e o Apache Kafka [1]. O Kafka é uma plataforma distribuída de *streaming*[1] e é usada neste trabalho para produzir uma *stream* de eventos a partir do ficheiro CSV que contém os registos das viagens de táxis. O Spark é uma *framework* para computação distribuída que possui uma API especificamente desenhada para o processamento de *streams* de eventos de forma escalável, com taxas de transferência elevadas e tolerância a falhas [2].

Assim, no nosso sistema foi utilizado o Kafka como produtor de eventos (lidos a partir do ficheiro que contém o *dataset* [3]) e o processamento dos eventos para responder às questões colocadas é feito em PySpark num jupyter-notebook.

Para processamento das *streams* de eventos tomou-se partido quer dos *Resilient distributed datasets* em Spark e do Spark SQL, que permite utilizar a linguagem SQL para escrever de forma mais expressiva as *queries*.

2.1 Setup

O *setup* do nosso *script* onde são executadas as *queries* começa com a inicialização do contexto do Spark Streams:

```
sc = SparkContext("local[2]", "KafkaExample")
ssc = StreamingContext(sc, 5)

# Lazily instantiated global instance of SparkSession
def getSparkSessionInstance(sparkConf):
    if ("sparkSessionSingletonInstance" not in globals()):
        globals()["sparkSessionSingletonInstance"] = SparkSession \
            .builder \
            .config(conf=sparkConf) \
            .getOrCreate()
    return globals()["sparkSessionSingletonInstance"]
```

e de uma *stream* directamente a partir do Kafka:

```
lines = KafkaUtils.createDirectStream(ssc, ["debs"], \
                {"metadata.broker.list": "kafka:9092"})
```

A primeira parte do processamento da stream criada é comum a todas as *queries* e consiste em mapear a *stream* em RDD (que são as linhas do ficheiro de *logs* neste caso). Seguidamente, filtram-se aquelas que correspondem a viagens fora da grelha do problema (*outliers*), utilizando a função por nós implementada para o efeito `filter_lines`:

```
lines = lines.map(lambda tup: tup[1])
filtered_lines = lines.filter(lambda line: filter_lines(line))
```

Uma vez filtrados os dados, aplica-se a cada RDD a *query* pretendida, como por exemplo (para a *Query1*:

```
filtered_lines.foreachRDD(lambda time, rdd: process(time, rdd, "Query1"))
```

onde a função `process` cria um `dataframe` com os *records* e aplica nele a *query* pretendida, de acordo com o argumento `query_name`:

```
def process(time, rdd, query_name):

    print("========= %s =========" % str(time))

    try:
        # Get the singleton instance of SparkSession
        spark = getSparkSessionInstance(rdd.context.getConf())

        # Convert RDD[String] to RDD[Row] to DataFrame
        rowRdd = rdd.map(lambda line: create_row_df(line))

        df = spark.createDataFrame(rowRdd)

        # This if block switches between the different queries
        if query_name == "Query1":

            [...]

        elif query_name == "Query2":
```

```
            [...]

        [...]

    except:
        traceback.print_exc()
```

2.2 Q1: Find the top 10 most frequent routes during the last 30 minutes.

Na secção de código associada à *query* 1, é criado um *dataframe* com as diferentes rotas e a sua frequência a partir de várias transformações sobre o *dataframe* contendo os *records* das viagens lidos:

```
def process(time, rdd, query_name):
    print("========= %s =========" % str(time))
    try:
        [...]

        # This if block switches between the different queries
        if query_name == "Query1":

            # Obtain the unsorted list of route frequencies by
            # grouping by a route (a pickup_cell and a dropoff_cell) and a window of 30 minutes
            # then aggregate the values into a count (in this case counting fare amount for no
                particular reason)
            route_freqs = df\
                        .groupBy(df.pickup_cell,\
                                 df.dropoff_cell,\
                                 window(df.pickup_dt, '30 minutes'))\
                        .agg(count(df.fare_amount).alias("NumTrips"))

            # Then select only the desired columns and sort descendingly the NumTrips variable
                and limit it to 10 values
            # to get the 10 most frequent routes
            most_freq_routes = route_freqs\
                        .select(route_freqs.pickup_cell, route_freqs.dropoff_cell,
                            route_freqs.NumTrips)\
                        .sort(desc("NumTrips"))\
                        .limit(10)

            most_freq_routes.show()

        elif query_name == "Query2":

        [...]

    except:
        traceback.print_exc()
```

Começa-se por fazer um `groupBy` que faz o agrupamento dos resultados que se encontram numa janela de 30 minutos utilizando o tempo `df.pickup_dt` por `pickup_cell` e `dropoff_cell`. Cada par `pickup_cell,dropoff_cell` distinto constitui uma rota pelo que o `groupBy` anterior faz um agrupamento dos *records* por rota. Finalmente, usando o `agg` obtém-se o *dataframe* com uma coluna extra designada por `"NumTrips"` com o número de *records* para cada rota.

Para obter a lista das 10 rotas mais frequentes basta agora seleccionar as colunas de interesse, `pickup_cell`, `dropoff_cell` e `NumTrips`, e ordenar por esta última, i.e. por número de viagens (`sort(desc("NumTrips"))`). Finalmente, com `limit(10)` extraem-se as 10 rotas mais frequentes.

2.2.1 Evaluation

Exemplo do início de *output* da *query* 1:

```
========= 2019-06-18 16:53:15 =========
+-----------+------------+--------+
|pickup_cell|dropoff_cell|NumTrips|
+-----------+------------+--------+
|    153.164|     155.160|       2|
|    161.159|     161.159|       2|
|    156.167|     155.163|       2|
|    156.164|     162.158|       2|
|    152.169|     153.168|       2|
|    157.162|     157.162|       2|
|    160.160|     157.165|       2|
|    154.164|     153.164|       2|
|    157.162|     155.163|       1|
|    160.171|     153.169|       1|
+-----------+------------+--------+
```

2.3 Q2: Identify areas that are currently most profitable for taxi drivers.

Para a *query* 2, a criação do *dataframe* com os *records* processa-se mais uma vez como descrito na secção 2.1.

As áreas mais rentáveis são aquelas em que a (*profitability*) é maior, sendo que esta é dada pelo lucro numa área nos últimos 15 minutos dividido pelo número de táxis vazios nessa mesma área nesse intervalo de tempo.

Assim, para encontrar as áreas mais rentáveis para os taxistas é necessário primeiro determinar o número de táxis vazios em cada área, que é dado pelos táxis que tiveram um *dropoff* nessa área nos últimos 30 minutos:

```
# Get the number of empty taxis in an area by obtaining the number of taxis that
# had a dropoff in a given cell in the last 30 minutes
empty_taxis = df\
            .groupBy(df.dropoff_cell,\
                     window(df.dropoff_dt, "30 minutes"))\
            .count().alias("NumEmptyTaxis")
```

Ao *dataframe* inicial com os *records* das viagens é aplicada uma janela de 30 minutos e feito o `groupBy` também por `dropoff_cell`. A contagem dos táxis nestas condições é então adicionada ao *dataframe* inicial numa coluna designada por `"NumEmptyTaxis"`. O *dataframe* assim obtido é designado por `empty_taxis`.

O lucro de uma dada área é dado pela média das receitas das viagens (soma do preço da viagem com a gorjeta) para iniciadas nessa área nos últimos 15 minutos. Assim, é necessário aplicar uma janela de 15 minutos e fazer o `groupBy` por `pickup_cell`.

```
# Get the total profit of an area which is the average
# of fare amount plus tip for a given area in the last 15 minutes
profit = df\
        .groupBy(df.pickup_cell,\
                 window(df.pickup_dt, "15 minutes"))\
        .agg(avg(df.fare_amount + df.tip_amount).alias("AreaProfit"))
```

Seguidamente, faz-se uma agregação e adiciona-se ao *dataframe* inicial o valor da média da receita das viagens iniciadas numa dada `pickup_cell`, que é guardada na coluna designada por `"AreaProfit"`. O *dataframe* assim obtido é designado por `profit`.

Para obter a *profitability* para uma dada área é necessário dividir a média das receitas das viagens iniciadas nessa área nos últimos 15 minutos pelo número de táxis vazios nessa mesma área. Assim, torna-se necessário fazer um `join` das streams com a condição `empty_taxis.dropoff_cell == profit.pickup_cell`.

```
# Join the two streams together on the commun column (the cell of pickup or dropoff)
joined_dfs = empty_taxis.join(profit, empty_taxis.dropoff_cell == profit.pickup_cell)

# Creating a temporary view of this stream naming it 'joined_dfs'
joined_dfs.createOrReplaceTempView('joined_dfs')
```

A partir da nova *stream* `joined_dfs`, pode-se agora calcular a `Profitability` e ordenar as áreas por ordem decrescente desta aplicando uma *query* SQL. Esta começa por seleccionr as colunas `pickup_cell, AreaProfit, count` e calcular a `Profitability` como `AreaProfit/count`. Seguidamente filtram-se as áreas para as quais o número de viagens é maior que zero e ordenam-se por ordem decrescente de `Profitability`:

```
sql_query = """
SELECT pickup_cell, AreaProfit/count AS Profitability
FROM joined_dfs
WHERE count > 0
ORDER BY Profitability DESC
"""
```

A instrução que aplica a *query* `sql_query` e faz o output das áreas ordenadas por `Profitability` encontra-se nas linhas:

```
profitability = spark.sql(sql_query)
profitability.show()
```

2.3.1 Evaluation

Exemplo do início de *output* da *query* 2:

```
========= 2019-06-18 17:00:30 =========
+-----------+------------------+
|pickup_cell|     Profitability|
+-----------+------------------+
|    189.185|              46.5|
|    190.186| 45.23333333333333|
|    182.160|              36.5|
|    158.153|              27.0|
```

```
|    153.164|22.873333333333335|
|    154.160|              21.0|
|    157.160|          19.40375|
|    182.173|              19.0|
|    162.155|             17.25|
|    170.163|             16.25|
|    157.159|             16.05|
|    155.158|              15.5|
|    156.166|            13.925|
|    154.165|              13.6|
|    156.160|              13.5|
|    155.163|             12.75|
|    160.150|             12.25|
|    156.169|12.166666666666666|
|    155.168|              12.0|
|    155.167|             11.75|
+-----------+------------------+
only showing top 20 rows
```

2.4 Q3: The city wants to be alerted whenever the average idle time of taxis is greater than a given amount of time (say 10 minutes)

Nesta *query*, começamos por criar tuplos estruturados com as variáveis registadas nos *logs* das viagens, utilizando o método `create_row`:

```
filtered_lines.foreachRDD(lambda time, rdd: process(time, rdd, "Query3"))
structured_lines = filtered_lines.map(lambda line: create_row(line))
```

Considera-se que um táxi se encontra disponível quando efectuou pelo menos uma viagem na última hora. Assim, para determinar os táxis disponíveis, começou-se por aplicar uma janela de 1 hora e criar uma nova *stream* de pares chave-valor em que a chave é o `medallion`, que identifica o condutor, e o valor é um tuplo com os elementos `pickup_dt, dropoff_dt, 1`.

O tempo durante o qual um táxi se encontra vago (*idle time*) é dado pela diferença entre o tempo de *dropoff* de uma viagem (`dropoff_dt`) e o tempo de *pickup* da viagem seguinte (`pickup_dt`). Para conseguirmos determinar este intervalo de tempo usamos o método `minutes_between` e filtramos os valores positivos para obter apenas os casos cujo tempo `pickup_dt` foi posterior ao tempo `dropoff_dt`.

```
# Calculating the minutes between the pickup_dt and dropoff_dt
# Then filtering the negative values (occurs when pick_dt is before dropoff_dt)
avg_idle_times = avg_idle_times \
                 .mapValues(lambda tup: (minutes_between(tup[1], tup[0]), 1)) \
                 .filter(lambda tup: int(tup[1][0] > 0))
```

Seguidamente, é feito um `reduceByKey` por `medallion` e são acumulados os tempos em que os táxis se encontram vagos e as unidades (que assim dão o número total de intervalos em que os táxis se encontram vagos). A duração média dos intervalos de tempo em que os condutores não se encontram a realizar viagens pode então ser calculada dividindo os valores acumulados anteriormente descritos, um pelo outro. Obtêm-se assim RDDs com pares cuja chave é o `medallion` e o valor é a média dos tempos em que o táxi está vago.

```
# Summing the idle times to then make the average for each medallion
# Then calculating the average idle time for each medallion
avg_idle_times = avg_idle_times \
                 .reduceByKey(lambda acc, elem: (acc[0] + elem[0], acc[1] + elem[1])) \
                 .mapValues(lambda tup: tup[0] / tup[1])
```

Finalmente, para se efectuar um alerta quando a média dos tempos em que os táxis se encontram desocupados excede os 10 minutos, filtra-se o *stream* de RDDs anteriores para valores superiores a este limite.

```
# Retaining only the values that have an average idle team of over 10 %minutes
# Then emmiting an alert message with the idle time of that medallion
avg_idle_times = avg_idle_times \
                 .filter(lambda tup: tup[1] > 10) \
                 .mapValues(lambda idle_time: f"Idle time alert: {idle_time} minutes idle")

avg_idle_times.pprint()
```

Quando se obtêm RDDs em que o tempo vago médio é superior a 10 minutos, faz-se uma transformação do elemento valor do RDD, que passa a ser uma *string* com a mensagem de alerta e o tempo médio de desocupação dos táxis.

2.4.1 Evaluation

Exemplo do início de *output* da *query* 3:

```
-------------------------------------------
Time: 2019-06-18 17:02:50
-------------------------------------------
('01D8C877762B42B4F375E1449019CC79', 'Idle time alert: 57.0 minutes idle')
('095FB08B5C968BC1BFA396E4A297741D', 'Idle time alert: 58.0 minutes idle')
('0A2B0D09CC814C4DFE90079A20E490E3', 'Idle time alert: 57.0 minutes idle')
('0A7E2DB544D098553DDEF21DE361DD95', 'Idle time alert: 57.0 minutes idle')
('0DC4DDFEC71A53DBB96C8557177EAD68', 'Idle time alert: 57.0 minutes idle')
('109B228195BF72453608B04226A8603F', 'Idle time alert: 57.0 minutes idle')
('185DB975730A99E17D05E4BFDA3888F2', 'Idle time alert: 57.0 minutes idle')
('2F0CF7DA9D342867A73A387748175701', 'Idle time alert: 57.0 minutes idle')
('313331ED5239D808BCA7444CA77FC69F', 'Idle time alert: 57.0 minutes idle')
('3B1DD51648DEB1B7C11DCBDAC23A8FFB', 'Idle time alert: 57.0 minutes idle')
...
```

2.5 Q4: Detect congested areas

Para detectar uma área congestionada (num dia), começa-se por aplicar uma janela temporal de um dia e faz-se um `groupBy` por `medallion`. Segue-se uma agregação em que é adicionada uma coluna com uma mensagem de alerta que depende do resultado da aplicação da *user defined function* `find_peak_udf` aos conjuntos de durações das viagens realizadas por um dado taxista (`collect_list(df.trip_time)`) e das correspodentes células de partida (`collect_list(df.pickup_cell)`).

```
# Obtain all rides for a taxi in a day and aggregate the values with the custom aggregation
    function
# described above
grouped = df.groupBy(df.medallion, window(df.pickup_dt, "1 days")) \
    .agg(find_peak_udf(collect_list(df.trip_time),
        collect_list(df.pickup_cell)).alias("alert_message"))
```

A função `find_peak_udf` procura o padrão associado a uma zona congestionada na lista de durações e devolve uma mensagem de altera e encontra-se registada nas linhas de código:

```
# Registering the user defined function
find_peak_udf = functions.udf(find_peak, T.StringType())
```

O código responsável pela procura do padrão definido por um pico seguido de três viagens com durações crescentes encontra-se nas seguintes linhas:

```
def find_peak(durations, cells):
    """
        Aggregation function that receives a list of durations and cells
        Searches in the list of durations for a pattern where a given duration is higher
        than the one before and after it, and the duration after it is followed by 3 rides of
            increasing duration
    """
    if(len(durations) == 0):
        return "No Congested Areas, and no durations"

    for idx, val in enumerate(durations):
        try:
            if idx != 0:
                if val > durations[idx - 1] and val > durations[idx + 1] and  durations[idx + 1]
                    <  durations[idx + 2] and durations[idx + 2] <  durations[idx + 3]:
                    return f"Alert, area {cells[idx]} is congested"
                else:
                    return "No Congested Areas"
            else:
                return "No congested Areas"
        except:
            return "No Congested Areas"
```

Quando este padrão é detectado na lista de durações, a mensagem de alerta retornada contém a célula correspondente onde ocorreu o congestionamento. A linha `grouped.select(grouped.alert_message).show()` realiza o *output* das mensagens de alerta.

2.5.1 Evaluation

Exemplo do início de *output* da *query* 4:

```
========= 2019-06-18 17:21:45 =========
+------------------+
|     alert_message|
+------------------+
|No congested Areas|
|No congested Areas|
|No congested Areas|
|No congested Areas|
|No congested Areas|
|No congested Areas|
|No congested Areas|
|No congested Areas|
|No congested Areas|
|No congested Areas|
+------------------+
```

2.6 Q5: Select the most pleasant taxi drivers

Os taxistas mais agradáveis são aqueles que possuem o maior total de gorjetas num dia. Assim, o primeiro passo é fazer um `groupBy` por `medallion` numa janela de 1 dia. Para cada `medallion`, e portanto cada taxista (assumindo que cada viatura é guiada sempre pela mesma pessoa), é feita a soma das gorjetas e esta é colocada numa nova coluna do *dataframe* designada por `"TotalTipAmount"`. O *dataframe* assim obtido é designado por `most_pleasant_unsorted`.

```
# Obtain an unsorted list of pleasant taxis by obtaining a list
# of taxis followed by the sum of the tips they obtained in a day
most_pleasant_unsorted = df\
                    .groupBy(df.medallion,\
                             window(df.pickup_dt, "1 days"))\
                    .agg(functions.sum(df.tip_amount).alias("TotalTipAmount"))
```

Para obter a lista ordenada dos taxistas mais agradáveis, é criado um novo *dataframe* (`most_pleasant`) a partir do anterior (`most_pleasant_unsorted`) onde são seleccionadas as colunas `medallion` e `TotalTipAmount` e é aplicado um ordenamento por esta última, de forma descendente.

Finalmente, para obter apenas os dez taxistas mais agradáveis basta tomar as 10 primeiras linhas do *dataframe* `most_pleasant`, com a instrução `limit(10)`.

```
# Sort this previous value descendingly and limit the stream to just 10 outputs
# to obtain the 10 most pleasant taxi drivers
most_pleasant = most_pleasant_unsorted\
            .select(most_pleasant_unsorted.medallion, most_pleasant_unsorted.TotalTipAmount)\
            .sort(desc("TotalTipAmount"))\
            .limit(10)

most_pleasant.show()
```

2.6.1 Evaluation

Exemplo do início de *output* da *query* 5:

```
========= 2019-06-18 17:09:30 =========
+--------------------+--------------+
|           medallion|TotalTipAmount|
+--------------------+--------------+
|F605C5086D44454E5...|          13.0|
|0E5C91D3EEB96AB7E...|          13.0|
|6945300E90C69061B...|         12.75|
|75DA258524EB3B7B5...|          10.4|
|6F910ABF764B97720...|          10.4|
|0C3E065904E923C33...|          9.75|
|F0EC6BD4D30DA6399...|           7.3|
|9A0623180459F71E8...|           7.2|
|1DDB1255470A78637...|           7.0|
|127E9B842B2F64CFB...|           7.0|
+--------------------+--------------+
```

3 Conclusions

O desenvolvimento deste projecto permite-nos ter um conhecimento mais aprofundado no que toca a estas cinco questões referentes aos dados de Nova Iorque. Os dados são provenientes da *ACM International Conference on Distributed Event-Based Systems* [3] e referem-se a viagens de taxís realizadas na cidade de Nova Iorque durante o ano de 2013. Tendo este dataset uma dimensão elevada (cerca de 12 GB) foi utilizada uma versão reduzida (cerca de 350 MB) para a obtenção dos resultados presentes neste relatório.

Com a realização da primeira *query* foi possível obter uma listagem das 10 rotas mais percorridas por taxistas nesta cidade a cada trinta minutos, dado-nos o conhecimento de quais as áreas de maior afluência de taxis. Na segunda *query* obtém-se, através da identificação das zonas mais rentáveis, uma informação bastante útil para a maximização de lucro de um taxista. A terceira *query* em oposição à segunda *query* permite-nos obter um alerta seguido da zona onde os taxistas têm menos clientela, sendo assim possivelmente as áreas de menor interesse para os mesmos. Com a resolução da quarta *query* obtém-se informação útil sobre quais as zonas mais congestionadas na cidade de Nova Iorque. Finalmente na quinta *query* obtém-se a informação de quais os taxistas mais agradáveis para com os seus clientes, equacionando a soma total de gorjetas que o mesmo recebe no período de um dia.

A resolução deste projecto usando a ferramenta Spark [2] permite manusear com maior facilidade o grande volume de dados presente neste *dataset*, sendo assim uma ferramenta útil à análise de *Big Data*.

Referências

[1] Apache Software Foundation. Apache kafka - a distributed streaming platform, 2017. URL: https://kafka.apache.org.

[2] Apache Software Foundation. Apache spark - lightning-fast unified analytics engine, 2017. URL: https://spark.apache.org.

[3] Chris Whong. Acm debs 2015 grand challenge dataset, 2017. (ACM DEBS 2015 Grand Challenge). URL: http://www.debs2015.org/call-grand-challenge.html.