Smart flight comparer considering alternative transportation

Serghei Socolovshi (serghei@kth.se) and Angel Igareta (alih2@kth.se)

Data Intensive Computing

Project Report

Stockholm, October 2020

1 Introduction

Nowadays, there are several websites that help finding the best route to fly from one place to another, such as Kayak, SkyScanner or Google Flights. However, none of these platforms take into account alternate routes with other methods of transport, such as travelling by boat, bus or train.

In this project the aim is to work with different data streaming APIs to implement a Spark Streaming application that, reading from Kafka data stream the origin, destination, departure date, price and time ranges, returns the optimal combination of routes, in terms of time and price.

2 Tools

The tools utilized for this project would include:

- Spark Streaming: an extension of the core Spark API that enables scalable, high-throughput, fault-tolerant stream processing of live data streams.
- Kafka: a distributed, topic oriented, partitioned, replicated commit log service used to publish and subscribe to streams of data.

3 Data

The data is retrieved from the following APIs:

- GeoDB Cities API [1]: A service that provides basic information about cities and countries around the world. It allows to constrain and sort the data in various ways, such as distance or maximum population. The data, however, cannot be considered reliable, since the provided information sometimes can be contrasted by the facts found on the Internet.
- SkyScanner Flight Search API [3]: Cloud service offered through the API portal RapidAPI that offers different endpoints to retrieve information about the airports in different cities, and possible connections among the airports. However, the data that can be extracted present some problems:
  - In case the flight is not direct, no transit stops are indicated.
  - There is no information about the departure or arrival time and flight duration.
  - The returned data is not always reliable. To collect airports through the API a query has to be specified, but no coordinates, resulting in the return of airports from other countries with similar names.
  - Complex and obsolete JSON structure, which requires a lot of processing to extract the necessary information.
  - If the flight is not available on the selected date, API still returns options for close dates.
  - Limited number of requests per minute.
- Meinfernbus-REST API [2]: Unofficial Public Flixbus API, containing the information about the bus stations in the cities and all the connections contained in the Flixbus database. It is possible to recover all the necessary information from the API replies, starting from departure and arrival stations to scheduled times and prices. The problems encountered with the data received from the API are the following:
  - Limited coverage of Flixbus, because the company is not operating in all the European countries. In order to solve this, it would be necessary to integrate other public transportation APIs.
  - Different name formats that make it difficult to recognize bus stops in the airports indicated by SkyScanner API.

4 Developed Applications

4.1 Search Engine

The Search Engine aims to act as a front-end to collect information about the trip the user aspires to do. First, it prompts the required information by console specifying the expected format and showing some example results. This information is encoded in a JSON with custom properties and it is sent through a producer to a Kafka topic with the key `input-<user_id>`. After this, a Spark Streaming Context is created along with a Kafka consumer, which is subscribed to the exact same topic name but only processing the keys that start with `output-<user_id>`, so the results are custom to only the user with `<user_id>`. Finally, the top n routes if found are displayed in the console.

4.2 Trip Generator Back-end

This second application is in charge of processing the input received in a custom Kafka Topic, collecting data from the previously stated APIs implement the idea, and finally returning the top n routes for the user `<user_id>`. To achieve this, the following steps were performed:

- Collect the client request from the Kafka Topic that contains `<src, departureDate, dst, priceRange, timeTravelRange>`
- Look for the nearest cities from the coordinates stored in `src` and `dst` in GeoDB Cities API. The nearest cities will become the origin and the destination points, while the furthest would be considered as transit points.
- Search for the transport hubs in the before-mentioned cities, such as airports and bus stations, using SkyScanner and FlixBus APIs. To achieve this, as these stations contained different names, a cluster of stations in the same city had to be manually performed.
- Find connections among the generated clusters for the `departureDate`, considering all the stations contained in every cluster.
- Create a graph, where the cities with the relative transport hubs are considered as nodes, and the routes are the edges. These edges contain a certain time and price, which later will be filtered taking into account the input price range and travel time range.
- Look for the optimal path from the origin to the destinations, in terms of time and/or price.

5 Results

The first image shows how the user can insert the data required to perform the query. The query is then encoded as a JSON and sent to the Kafka topic with the key `input-<user_id>`.

```
Welcome to CheapTravelling Search Engine! You are the user 956329276
    Enter source coordinates in the ISO-6709 format: ±DD.DDDD,±DDD.DDDD. Example: +59.3493243,+18.0707407
+59.3493243,+18.0707407
    Enter destination coordinates in the ISO-6709 format: ±DD.DDDD,±DDD.DDDD. Example: +55.6761,+12.5683
+55.6761,+12.5683
    Enter desired departure date in the format: yyyy-mm-dd. Example: 2020-10-29
2020-10-29
    Enter price range in the format: min,max. Example: 20,150
20,150
    Enter time duration range in hours in the format: min,max. Example: 0,24
0,24
Sending to Kafka: ProducerRecord(topic=input-travel, partition=null, key=input-956329276, value={"src": "+59.
```

The following screenshot presents a partial log of how the back-end processes the query.

```
Raw {"src": "+59.3493243,+18.0707407","departureDate":"2020-10-29","dst":"+55.6761,+12.5683", "priceRange": [100,200],"timeTravelRange"
Fetching cities of +59.3493243,+18.0707407
Processing cities of +59.3493243,+18.0707407
Fetching places of Stockholm
Fetching places of Uppsala
Fetching places of Upplands Väsby
Fetching places of Västerås
Fetching places of Örebro
Fetching cities of +55.6761,+12.5683
Processing cities of +55.6761,+12.5683
Fetching places of Copenhagen
Fetching places of Malmö
Fetching places of Aarhus
Fetching places of Rostock
Fetching places of Rostock
Creating graph
Initializing graph edges...
Initializing graph nodes...
Initializing graph routes...
Graph
**Nodes**
    City(Stockholm,,true,Set(Place(13628,Arlanda Airport T5,FlixBus,Stockholm), Place(40830.0,Stockholm Arlanda,SkyScanner,Stockholm),
    City(Malmö,,false,Set(Place(75145.0,Palma - Majorca,SkyScanner,Malmö)))
    City(Aarhus,,false,Set(Place(6118,Aarhus C,FlixBus,Aarhus), Place(29788,Aarhus Vest (Tilst),FlixBus,Aarhus)))
    City(Copenhagen,,true,Set(Place(2186,Copenhagen,FlixBus,Copenhagen), Place(45336.0,Copenhagen,SkyScanner,Copenhagen)))

**Edges**
    *Outcoming routes from node Stockholm*
        (3978:Stockholm Cityterminalen:FlixBus:Stockholm) ==(58.8€:15.0h)=> (2186:Copenhagen:FlixBus:Copenhagen)
        (13628:Arlanda Airport T5:FlixBus:Stockholm) ==(75.8€:16.0h)=> (2186:Copenhagen:FlixBus:Copenhagen)
        (42881.0:Stockholm Bromma:SkyScanner:Stockholm) ==(248.0€:0.0h)=> (45336.0:Copenhagen:SkyScanner:Copenhagen)
        (71119.0:Stockholm Skavsta:SkyScanner:Stockholm) ==(86.0€:0.0h)=> (45336.0:Copenhagen:SkyScanner:Copenhagen)
        (40830.0:Stockholm Arlanda:SkyScanner:Stockholm) ==(54.0€:0.0h)=> (45336.0:Copenhagen:SkyScanner:Copenhagen)
```

Lastly, the final output of the application is presented, which shows the top 5 routes according to the price. Unfortunately, it was not possible to perform such optimization with a precise time, because SkyScanner API was not providing correct data regarding the flight duration.

```
Kafka result for user 956329276 query
Routes found to travel from Stockholm to Copenhagen:
    Route 1 | Stops: Stockholm Arlanda => Copenhagen | Total Price: 54.0 EUR | Total Time: 0.0 hours
    Route 2 | Stops: Stockholm Cityterminalen => Copenhagen | Total Price: 58.8 EUR | Total Time: 15.0 hours
    Route 3 | Stops: Arlanda Airport T5 => Copenhagen | Total Price: 75.8 EUR | Total Time: 16.0 hours
    Route 4 | Stops: Stockholm Skavsta => Copenhagen | Total Price: 86.0 EUR | Total Time: 0.0 hours
    Route 5 | Stops: Stockholm Skavsta => Palma - Majorca => Copenhagen | Total Price: 102.0 EUR | Total Time: 0.0 hours
```

6 How to run code

6.1 Preparation

As Kafka uses ZooKeeper to maintain the configuration information, the ZooKeeper server needs to be started followed by the Kafka server.

```
zookeeper-server-start.sh $KAFKA_HOME/config/zookeeper.properties
kafka-server-start.sh $KAFKA_HOME/config/server.properties
```

Next, a Kafka topic with `<topic_name>` needs to be created, where all the messages will be exchanged.

```
kafka-topics.sh --create --zookeeper localhost:2181 --replication-factor 1
--partitions 1 --topic <topic_name>
```

Regarding the applications, first the trip generator needs to be executed from the root of the source project containing `build.sbt`. This can be done with the utility *sbt*.

```
sbt "run <topic_name>" // Select Trip Generator
```

Finally, the search engine needs to be executed following the same procedure as in the last step but choosing Search Engine. After executing it, the trip details will be prompted.

References

[1] Geodb cities api. *https://rapidapi.com/wirefreethought/api/geodb-cities/details*.

[2] Meinfernbus-rest. *https://github.com/juliuste/meinfernbus-rest*.

[3] Skyscanner api documentation. *https://skyscanner.github.io/slate/*.