Real-Time Credit Cards Transactions Analysis

Project Report

Bianchi Edoardo - 20740

June 16, 2022

Course Code: 73033

Contents

1 Introduction

Buying products and services online is an extremely common practice nowadays. With the advent of new means for electronic payment, making transactions comfortably from a smartphone or PC is a valid, convenient and fast option. The number of services and products available online far exceeds the number of those available in any physical store, and the merchants themselves have the ability to expand the number of customers. Precisely for these reasons the number of transactions that take place in real time is very high. The purpose of this project is to build a system that analyzes transaction data in the form of a stream and proposes a summary dashboard of the analysis conducted.

The repository of the project can be found on GitHub at https://github.com/EdoWhite/RealTime-CreditCard-Transactions-Analysis. Tag: v1.0.0 ”Final Release”.

2 Application Domain

Since the invention of money in the 7th century BC, the exchange of paper, metal, and other forms of currency has been the most convenient way to pay for everyday purchases in real time. Even with the advent of credit cards, cash is still used for convenient and instantaneous exchanges. However, many transactions no longer take place in person, and in some cases certain items are not available in physical stores: in these cases, other means of payment are required.

Actually the idea of "instant payment" is not new - with cash, for instance, we can pay in real-time - but here the change is in the mean used to pay: not cash anymore. Smart devices and online services represent the main reason behind the adoption of electronic real-time payments, available 24/7 from everywhere.

For these reasons, we can understand how large the volume of data generated by electronic transactions is, and that by its very nature, it is a continuous, real-time stream. We can also understand how sensitive this data is: By analyzing the transactions performed by an individual, it is possible to build an accurate profile of the individual himself. The transactions can describe a person's behavior, needs, and desires. In this analysis we will try to understand how people use electronic payments means.

3 Data Sources Description

3.1 Dataset Description

As previously reported, data relating to transactions (online and otherwise) performed by users represent sensitive information, which could lead to a violation of privacy. Precisely for these reasons, finding real-time streams of real transactions is hard if not impossible. The same is true for real transactions performed in the past (sources not in real time). To overcome this lack of data and still be able to develop an application, we will use a transaction dataset generated by a data generator.

The dataset in question is presented in *csv* format and can be found on *Kaggle*. This dataset contains more than one million transactions records with related information. The original version of the data contains several attributes, but only a few will be considered for the purpose of the project. In particular, the attributes that will be considered are:

1. **ID**: Represent the transaction unique identifier number, not particularly useful in the final analysis.
2. **category**: Represents the category to which the transaction in question belongs. In particular, there are 14 different types of categories.
3. **amt**: The transaction amount.
4. **gender**: Represents the gender of the holder of the credit card used for the transaction.
5. **city**: Represents the city of the holder of the credit card used for the transaction.
6. **state**: Represents the state of the holder of the credit card used for the transaction.
7. **city_pop**: Represent the city population of the city of the credit card holder.
8. **job**: Represents the job of the holder of the credit card used for the transaction.
9. **unix_time**: Represents the UNIX time in which the transaction occurred. Unix time represents the number of seconds elapsed from 00:00:00 UTC, 1 January 1970.

3.2 Data Generation

As we reported previously, data is generated by a data generator. Accordingly with the creator of the dataset hosted on *Kaggle*, the generator used is presented on GitHub in a repository named *Sparkov Data Generation* and was created for a project called *Sparkov Project*, *"A Markov-Chain based fraud detection system based in Spark"*.

According to the code, in order to generate data, first is necessary to create a customer file and then this file can be used to generate the actual transactions. The generator uses a pre-defined list of merchants and transaction categories. The customer file, containing customer-related data, is generated using a Python library called *Faker*. After the generation of the customers data, it is necessary to select a *profile*, a file that contains some behavioural attributes proper to demographics aspects of the customers. These attributes are defined in terms of minimum and maximum transactions per day, distribution of transactions across days of the week and normal distribution properties (mean, standard deviation) for amounts in various categories. According to the selected profile, the transactions are generated again using *Faker*. An example of profile is named *adults_2550_female_urban.json* and represent behaviours of adult females in the age range of 25-50 who are from urban areas.

As reported by the creator of the dataset, transactions have been generated across all profiles and then merged together to create a more realistic representation of real transactions. The transactions are generated in a time interval that ranges from 1 Jan 2019 to 31 Dec 2020.

3.3 From a Dataset to a Real-Time Stream

In order to simulate a real-time stream of data we make use of a custom Python file, called *producer.py* that "converts" the tabular data into a stream-like source, sending json messages to a *Kafka* topic. What this script does is reading line by line the transactions from the dataset, selecting the attributes of interest, parsing this attributes in a json structure and send this json structure to *Kafka* respecting the timestamp attached to each transaction.

In this way we simulate transactions happening in real time and, at the same time, we respect the original time attribute attached to each transaction.

It is important to point out that the speed at which the data is output is not particularly high if the timestamp associated with the transactions is respected. In a real application one would expect a much faster flow of data. To simulate this situation, it is possible to increase the speed at which data is emitted; obviously the timestamp is no longer respected, but a somewhat more realistic situation can be tested.

4 Technologies and Overall Architecture

4.1 Technologies & System Architecture

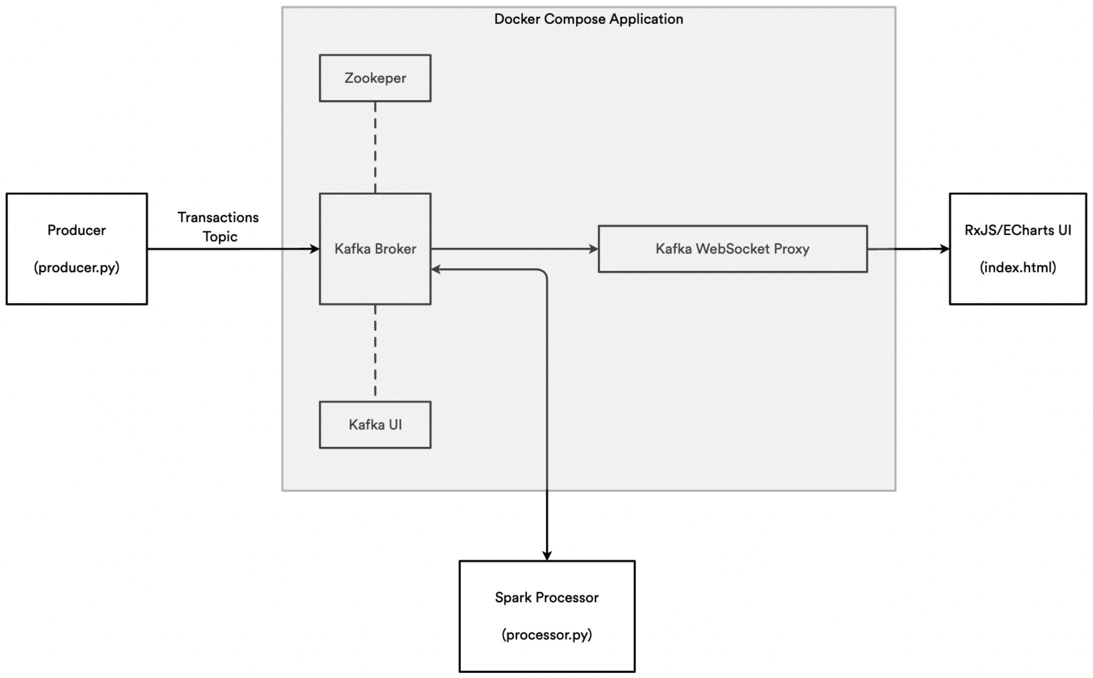

Figure 1: Overall system architecture. Image by the author.

In this section we present the technologies used to develop the system and describe the general architecture of the application, presented in Fig. 1, introducing all the components and their purpose.

In particular, some components are managed using the *Docker* Compose platform. *Docker* Compose is a tool used in order to build and run multi-component *Docker* applications. To use Compose, a YAML file must be created and this file represents the configuration of the application's services. Then, with a single command, we create and start all the services from our configuration. *Docker* Compose works with *images*, an image is basically a file used to execute code in a *Docker* container.

Docker is used to set up and run *Kafka Docker Image*, *ZooKeeper Docker Image*, *Kafka UI Docker Image* and *Kafka Websocket Proxy Docker Image*. This components are used to manage the streams of data.

Other important components are the producer and the processor, in addition to the graphical interface proposed to the final user. The producer and the processor are defined as Python files that must be started from a terminal. While the producer is used to simulate a stream of data, the processor is more advanced and responsible for the actual analysis conducted on the data coming.

The processor uses *Spark* in order to process the stream of data. *Spark* is an open-source data processing framework known for parallelism and fault tolerance properties. In particular, we make use of an extension named *Spark Structured Streaming*.

Apache *Spark Structured Streaming* is an extension of the Spark API and the Spark SQL engine that allows real-time data processing from various sources including *Kafka*, supporting batch and streaming workloads. The processed data can be managed in different ways, in this case we push back to *Kafka*, in new topics. The main *Spark Structured Streaming* idea is a Discretized Stream that is a stream of data divided into small batches. *Spark Structured Streaming* can work with other *Spark* components like MLlib, used to perform machine learning tasks, and Spark SQL, that is used in this project to perform analytical queries on streaming data.

Spark Structured Streaming allows the developer to simply write queries, completely hiding the actual stream management: the user writes the query, Spark updates the answer. Regarding the update of the answer, we can personalize *what* is the output and *when* to output. *What* is the output can be defined specifying the *Output Mode* property: "complete" to output the whole answer every time, "update" to output changed rows and "append" to output only the new rows; regarding *when* to output we can specify the *Trigger* property in the form of a time attribute, for example one minute. No trigger means to output as fast as possible. In this project the trigger is 3 seconds and the output mode is *update* for all the queries except for the temporal queries that uses *append* mode.

Regarding fault-tolerance, *Spark Structured Streaming* utilizes a *Checkpoint* system use to track the progress of the queries in a persistent store, that can be used to restore states in case of failure. Out-of-order or late data can be managed with *Watermarking*: It is a kind of sliding threshold that determines when old states should be discarded. Data older than the watermark is discarded because it is designated as "too late". *Watermarking* allows us to specify how late data can be. In this project, the watermark is 2 minutes.

Spark Structured Streaming provides also practical functions to deal with complex data facilitating parsing and serializing operations, *from_json* and *to_json* does exactly this.

The graphical interface provides a dashboard updated in real time used to show the results of the analysis conducted on the data streams. In particular, these results are shown in the form of summary graphs and "badges". This interface is created using *RxJS*, *ECharts* and the *Bootstrap* framework. *RxJS* is a reactive extensions library for JavaScript and Apache *ECharts* is an open source JavaScript visualization library that provides customizable and reactive charts. *Bootstrap* is an open-source CSS framework directed at responsive design and front-end web development.

All these components are used together and are indispensable for the correct functioning of the application and may be a good idea to create a custom Python virtual environment in order to have a dedicated environment to run the application with all the required and necessary additional libraries.

Each of the above cited component has a fundamental purpose:

1. **Producer**: This Python file is used to convert a csv file in a stream of data, simulating a real time source. Sends out json data to a *Kafka* topic, named "Transactions".

2. **Kafka + ZooKeeper + Kafka UI**: This components are used together in order to store topics into *Kafka*. The stored topics are both original topics and topics resulting from processing steps. *Kafka UI Docker Image* provides a practical user interface, useful to monitor all the topics. The *Kafka UI Docker Image* dashboard can be accessed from *http://localhost:28080/*.

3. **Spark**: The *Spark* framework is used to analyze the stream of data and create new topics. These new topics represent the results of the query. In particular, we make use of the *Spark Structured Streaming* extension and the Spark SQL language. The results of the queries are pushed back to *Kafka* in new topics, one for each query.

4. **Kafka WebSocket proxy**: This component is a WebSocket proxy that allows JavaScript code to produce and consume messages from topics in *Kafka*.

5. **RxJS + ECharts + Bootstrap**: This components are used in order to build the final dashboard presented to the user. In particular, we use *RxJS* to get data from the Kafka Proxy and update the page in real time, *Bootstrap* to build the page structure and pagination, Apache *ECharts* to generate the proposed visualizations. The dashboard can be accessed opening the *index.html* file.

6. **Docker**: The *Docker* Compose is used to manage all the components, except for the producer and the processor Python files that must be started from the command line, by hand.

4.2 Workflow & Processing Operations

In this subsection we provide a description of the workflow, that is how this application works and what are the involved processing phases. Please note that the proposed Command Line Interface (CLI) commands may vary depending on the platform used. It is also important pointing out that the platform on which this application was developed and tested is *osx-arm64* and it may be necessary to install / remove additional libraries to make the application work on other platforms.

The first step is to clone the GitHub repository and make sure to have all the requisites necessary to use the application. To simplify this operation we can create a new virtual environment and install the required libraries. The *conda_env.txt* file contains all the required libraries in a format that is ready to install in an Anaconda Virtual Environment.

This operation can be done from the (CLI) with the following command:

```
# navigate to the cloned folder
$ conda create --name kafka-env --file conda_env.txt
$ conda activate kafka-env
```

The second operation to be performed is starting the *Docker* composer, which will take care of loading the components needed by the application. Also this operation is done from the CLI, with the following command:

```
(kafka-env) $ docker-compose up
```

Next, we have to launch the producer, which is in charge of transforming the transactions file into a data stream that is sent to *Kafka*. It is possible to specify how fast the data is forwarded to *Kafka* with the *speed* attribute: if set to "1" (default) the transactions will be issued respecting the timestamp attached to them, with higher values, the emission speed will increase. Useful option in debugging to see graphs updating very often. Another option that can be applied regards the *Kafka* server Bootstrap Server setting (default: localhost:29092).

Start the producer from the CLI using:

```
(kafka-env) $ cd bin/
(kafka-env) $ python producer.py
```

Start the producer at faster speed:

```
(kafka-env) $ cd bin/
(kafka-env) $ python producer.py --speed 10
```

Start the producer with a specific Bootstrap Server:

```
(kafka-env) $ cd bin/
(kafka-env) $ python producer.py --boostrap-server localhost:29092
```

After transaction data arrives at *Kafka*, it is the processor's job to process and execute operations. The processor must be started via CLI, using the command:

```
(kafka-env) $ cd bin/
(kafka-env) $ python processor.py
```

Again, a parameter related to the *Kafka* Bootstrap Server can be set, as with the producer, in the same way.

The processor's job is to read data from the topic created by the producer and perform analytical queries on the data. The results of the queries are sent back to *Kafka* in new topics. Each new topic that is created, corresponds to a query and correspond also to a particular visualization in the final dashboard. The queries are defined using *Spark* SQL and the result must be formatted as json.

The main structure of the queries, very similar to each others, is constructed in the following way:

```
SELECT <new_topic_name> AS topic,
    <messages_key> AS key,
    to_json(named_struct(
        <attribute>, <value>,
        ...
        )) AS value
FROM <table>
```

Aggregation functions, windowing operations, counts and averages can be added to the basic structure. An example of a complete query with windowing and grouping looks like the following:

```
SELECT 'trans_last_minute' AS topic,
            '' AS key,
            to_json(named_struct(
            'category', category,
            'start', window.start,
            'end', window.end,
            'transactions', COUNT(*),
            'amount', SUM(amount)
            )) AS value
FROM trans
GROUP BY category, WINDOW(ts, '1 minute', '2 seconds')
```

Finally, to view the dashboard we can simply open the *index.html* file in the browser. Note that there may be differences in page rendering depending on the browser you use; the application was tested using the *Firefox* browser.

To stop the application, we can simply close producer and processor and run the following command from the CLI:

```
(kafka-env) $ docker-compose down -v
```

5 Functionalities

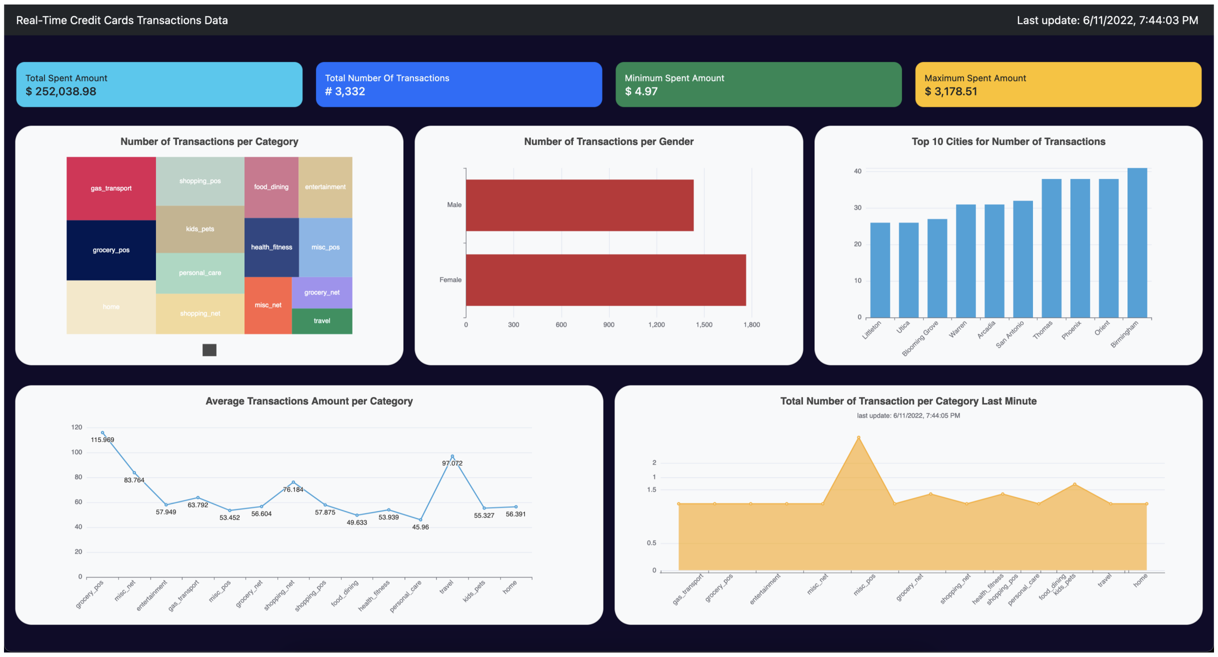
Real-Time Credit Cards Transactions Data
Last update: 6/11/2022, 7:44:03 PM
Total Spent Amount
$ 252,038.98
Total Number Of Transactions
# 3,332
Minimum Spent Amount
$ 4.97
Maximum Spent Amount
$ 3,178.51
Number of Transactions per Category
Number of Transactions per Gender
Male
Female
Top 10 Cities for Number of Transactions
Littleton
Utica
Blooming Grove
Warren
Arcadia
San Antonio
Thomas
Phoenix
Orient
Birmingham
Average Transactions Amount per Category
Total Number of Transaction per Category Last Minute
last update: 6/11/2022, 7:44:05 PM

Figure 2: The final dashboard presented to the user. Image by the author.

The system is used to compute analytical queries on transactions data. The results are then proposed to the final user in a dashboard. This dashboard is made up of different types of graphs and plots, populated in real-time.

All system functionalities are grouped and accessible from the GUI, which in this case is passive, i.e., the user cannot change its properties.

In applications such as this, the role of the GUI is critical: simplicity and clarity in a modern, dynamic design. The proposed interface is shown in Fig. 2. The structure shares elements common to web pages, such as the navigation bar, and elements proper to dashboards.

The basic structure has three rows, on the first one are shown numerical data related to:

- **Total Spent Amount**: Represent the total amount of all transactions, useful for understanding the volume of money moving over time.
- **Total Number of Transactions**: Represents the total number of transactions over time, which is useful for understanding the volume of transactions in terms of the amount of payments.

- **Minimum Spent Amount**: Represent the minimum amount spent by a person.
- **Maximum Spent Amount**: Represent the maximum amount spent by a person.

The last two numerical information can be useful in order to define a sort of threshold regarding the amount of a transaction.

On the second row there are three visualizations, specifically:

- **Number of Transactions per Category**: Useful to understand in which categories people tend to buy most frequently.
- **Number of Transactions per Gender**: Useful to understand who is more likely to spend, among men and women, note that no age groups are considered.
- **Top 10 Cities for Number of Transactions**: This visualization shows the cities where most electronic transactions take place. It can be an indicator of how inclined merchants and customers are to use electronic means of payment.

On the third and last row there are instead only two visualizations, in detail:

- **Average Transactions Amount per Category**: Useful for understanding in which categories people tend to spend more money.
- **Total Number of Transactions per Category, Last Minute**: Interesting to look at the number of transactions happening in a fixed time slot, could be also interesting to fix a different time slot, for example a day, and see how the value changes. This query make use of a sliding window of length *1 minute* and sliding interval of *5 seconds*.

6 Lessons Learned and Conclusions

The development of a project like this requires a wide set of knowledge due to the number of technologies and frameworks involved. The first thing i noticed is that combining together different frameworks and components, even using *Docker* Compose, is non trivial and may require lots of troubleshooting and trial & error.

Another important point regards the difficulty in finding real-time sources of data publicy and freely available. This is also associated with important privacy issues. The privacy aspect in such applications is very important and plays a fundamental role since data regards people.

We must also consider an ethical discourse: graphs and summaries presented in a dashboard can be misleading and affect one's view of reality. Impartiality and neutrality are essential to avoid influencing the end user's opinion. The same applies to the data streams used in the analysis: Incorrect, altered, inauthentic, corrupted, and inconsistent data can completely ruin an analysis. This aspect is fundamental for any kind of data-related product.

Talking about data and data streams, we need to specify how the complexity of the system can grow as the complexity of data grows: some sources of data can generate complex structures and the project may require working with more than one sources. Keeping under control a similar dimensionality and complexity can raise new difficulties and problems.

The speed at which data is produced presents another fundamental point. When working with real-time resources, there may be scenarios in which the volume of data produced is extremely high, and this poses another challenge: calculating computing power and defining requirements is not trivial, especially if the system will be used by multiple users who expect a particular service.

Another point concerns the design of the application's graphical user interface. The graphical user interface and user experience are extremely important because they are offered to the end user, usually a customer who is likely to pay for a service. The graphical user interface should be modern and functional, comprehensive and intuitive. Colors, icons, images, layout, and space must be carefully considered during development. Generally, the design is created by experts in human computer interaction and interface design.

All these problems and challenges for us as programmers only appear when we get our hands "dirty" and start developing. Even a theoretically noncomplex project like "a dashboard for visualizing transaction data in real time" involves non-trivial aspects and requires dealing with all the above issues.

Although I am satisfied with the result, there are aspects that can be improved or deepened: for example, more pages can be added to the dashboard specifically for different types of analysis.

It is also possible to add interactive visualizations that allow the user to explore different levels of depth and accuracy. Another improvement concerns the complexity of queries that could lead to more advanced analysis and results.

Although not in the scope of this project, analyzing and reporting fraud transactions in real-time can be another interesting addition to this dashboard.

In the end using a real data source would lead to more interesting and authentic results.

References

Apache Software Foundation. *ECharts*. URL: https://echarts.apache.org/en/index.html.

– *Kafka*. URL: https://kafka.apache.org/.

– *Spark*. URL: https://spark.apache.org/.

– *Spark Structured Streaming*. URL: https://spark.apache.org/docs/latest/streaming-programming-guide.html.

Bootstrap Core Team. *Bootstrap*. URL: https://getbootstrap.com/.

Docker Inc. *Docker*. URL: https://www.docker.com/.

– *Kafka Docker Image*. URL: https://hub.docker.com/r/confluentinc/cp-kafka.

– *ZooKeeper Docker Image*. URL: https://hub.docker.com/r/confluentinc/cp-zookeeper.

Joke2k. *Faker*. URL: https://github.com/joke2k/faker.

Kartik2112. *Kaggle*. URL: https://www.kaggle.com/datasets/kartik2112/fraud-detection.

Kpmeen. *Kafka Websocket Proxy Docker Image*. URL: https://kpmeen.gitlab.io/kafka-websocket-proxy/.

Namebrandon. *Sparkov Data Generation*. URL: https://github.com/namebrandon/Sparkov_Data_Generation.

– *Sparkov Project*. URL: https://github.com/namebrandon/Sparkov.

Provectus. *Kafka UI Docker Image*. URL: https://github.com/provectus/kafka-ui.

ReactiveX. *RxJS*. URL: https://rxjs.dev/.