Fraud Detection System

Davide Calzà[1], Marta Faggian[2], Denis Gurabardhi[3]

[1]davide.calza@studenti.unitn.it

[2]marta.faggian@studenti.unitn.it

[3]denis.gurabardhi@studenti.unitn.it

https://github.com/martafaggian/BDT2023-fraud-detection

Abstract

The project develops an intelligent fraud detection platform that would ingest transactional data from account banks, credit, debit and prepaid cards and investment portfolios. Such a system will process real time data in order to perform statistical analyses to flag suspicious activities to warn banks. The basic idea was to develop a system that could be used by a cooperative of banks to monitor the movements of their user accounts.

Keywords

Fraud detection, Kafka, Redis, Cassandra, Flink, Docker

I. Introduction

The project's main object is to create an intelligent fraud detection platform that gathers transactional data from various sources and processes it in real-time to identify and flag suspicious activity. The platform was initially intended for use by a cooperative of banks to monitor their user accounts and prevent fraudulent activities. Due to the unavailability of real streaming transactional data, both real and simulated, offline sources derived from existing fraud detection datasets were used to simulate actual fraudulent activities and test the performance of the algorithms and models implemented within the platform.

II. System Model

A. System architecture

The system architecture (see **Fig. 1**) consists of a pipeline that processes data from different source CSV files. Each file is streamed in parallel and the streamers statuses are stored in a Redis cache, which serves as a tracker. The streamed data is transmitted to Kafka, which acts as a message broker, and Flink reads the messages from Kafka in a streaming fashion for further processing.

Within the Flink component of the system architecture, there are multiple parsers specifically designed to handle the data received from Kafka. These parsers play a crucial role in extracting the relevant information from the incoming data stream. They analyze the data, identify key data points, and transform them into a standard target format that is suitable for further processing. The extracted information can be used for two main purposes: fraud detection and balance updates. For fraud detection, the processed data is sent to Flink's stream processing pipeline, where specialized models and algorithms are applied to detect suspicious activities and potential fraudulent transactions. The transformed data can also be utilized to update account balances. By leveraging the parsed information, the system can perform calculations and adjustments to the account balances based on the received transactions.

Outside of the Flink processes, the system architecture includes two important steps for data storage and retrieval. First, Redis is utilized to store certain data related to accounts for fast lookup and retrieval. The data is stored in Redis using a predefined format that includes fields such as account, bank ID, user ID and type of transaction. This format ensures that the data is organized and easily accessible for subsequent operations. Furthermore, the processed data is loaded into a Cassandra database that provides reliable storage and retrieval capabilities for large volumes of data. The data is stored in Cassandra, enabling efficient querying and retrieval based on various criteria. The Cassandra database serves as a robust and reliable storage solution for the system's data.

The visual representation of the data is provided through Grafana, where the cooperative of banks can access and analyze the data. Grafana allows for the creation of customized dashboards and offers powerful visualization and analytics capabilities. For end-users, Flask and Dash frameworks are utilized to provide web interfaces for data visualization and analysis. These frameworks enable users to interact with the data, access their account information, and perform various actions.

B. Technologies

Our project required several needs that made us choose some technologies over others, guided by research on existing fraud detection pipelines and analysis of their respective use cases.

We decided to use **Docker** because its speed allows for faster development, testing, and deployment of applications. Docker's portability ensures easy movement of applications while maintaining performance. The scalability feature enables deployment across various servers and cloud platforms, with the flexibility to adjust the scale as needed. Rapid delivery is achieved through standardized container formats, providing a reliable and consistent environment for efficient code movement between development, test, and production systems[1].

We used **Redis** because it provides a data structures server, allowing multiple processes to query and modify shared data structures. It offers fast performance with non-volatile storage, efficient memory usage, and essential database features such as replication, durability, clustering, and high availability. Its in-memory nature enables quick data retrieval, reducing the need to access slower storage systems. Redis is well-suited for processing large amounts of distributed data and serves as an effective caching solution to improve system performance and scalability. Moreover, Redis follows a key-value data model, which simplifies data storage and retrieval.[2]

We have chosen **Kafka** because it is a distributed platform for stream processing and messaging. Its low latency and high throughput make it ideal for real-time data pipelines and applications. Kafka's cluster-based architecture allows for easy scalability and reliable data transfer between systems. With its storage layer functioning as a distributed transaction log, Kafka serves as a crucial component for processing data streams in enterprise infrastructures[3].

Flink's stream-first approach provides low latency, high throughput, and real-time processing, making it well-suited for our use cases. Its ability to manage memory independently and automatically optimize tasks based on data characteristics simplifies development and improves performance. Furthermore, Flink seamlessly integrates with other technologies in our ecosystem, ensuring compatibility and smooth operation. By effectively handling data integration and stream processing, make Flink better than Spark in building scalable and robust streaming data pipelines[4].

Cassandra's Column-Family data model, which supports structured, semi-structured, and unstructured data, aligns well with our requirements. Cassandra's ability to handle large data volumes and distribute data across multiple data centers provides scalability and efficiency. Its peer-to-peer architecture eliminates the need for master nodes, ensuring high availability and scalability. The replication strategies offered by Cassandra, along with durability through synchronous and asynchronous replication, provide fault tolerance and data consistency. The use of indexing improves data retrieval efficiency[5].

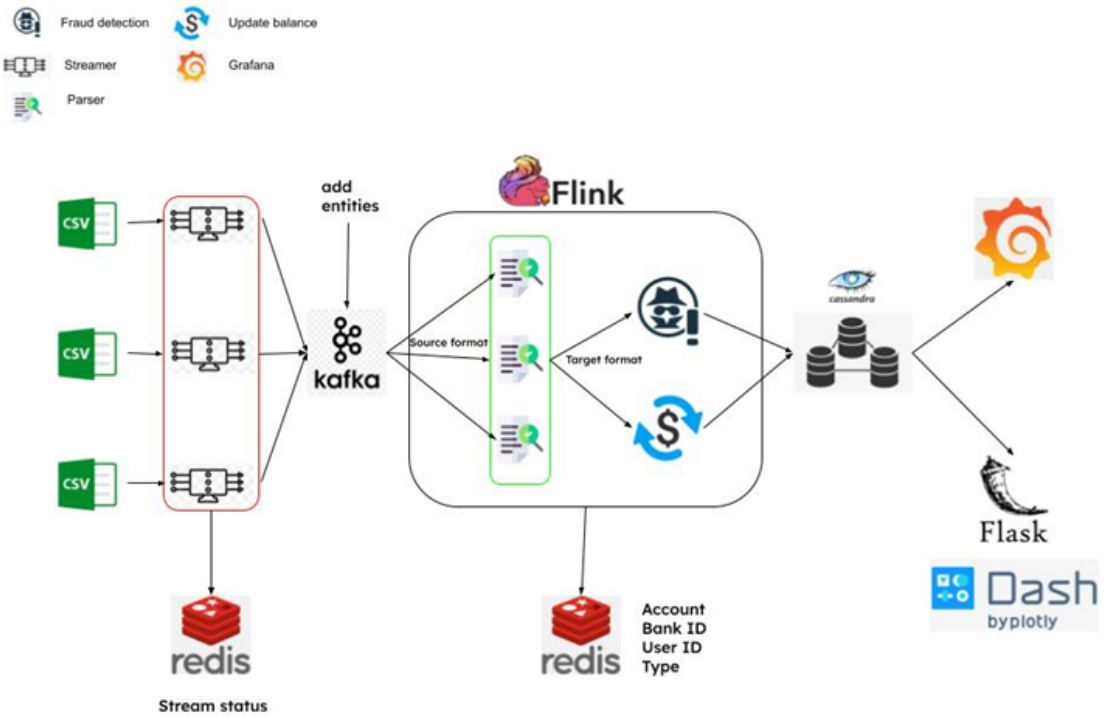

Fig. 1 Pipeline design

[1] (Rad, Bhatti, and Ahmadi 2017)
[2] (https://github.com/redis/redis, last visited 07/06/23
[3] (Shree et al. 2017)
[4] (Gurusamy et al. 2017)
[5] (Abramova and Bernardino 2013)

Grafana enables powerful real-time data visualization and analytics, as well as being useful for client alerting. It supports querying, visualization, and exploration of data from all the sources that we used, transforming time-series data into compelling graphs.
Flask and **Dash** offer flexibility and scalability for web development. On one hand, Flask is a lightweight, micro WSGI web application framework. On the other hand, Dash is an open-source Python framework for building analytical web applications. It is built on top of Flask and other frameworks, and allows the creation of web interfaces for data visualization and analytics using Plotly's powerful tools.

III. Implementation

During the implementation phase of the project, several key components were developed, including data modelling, data generation and streaming, stream processing, data parsing, fraud detection, balance update, saving to Cassandra, entity addition, and dashboard creation. Each component was designed to operate as a separate service within its own Docker container, providing modularity and scalability. The entire prototype can be easily launched using a single *start.sh* bash script, streamlining the deployment process. Essential parameters for managing the workflow, such as host and port configurations for accessing the various services, are included in the *config.yaml* described in details in the GitHub repository's README: https://github.com/martafaggian/BDT2023-fraud-detection/.

A. Data Modeling

A preliminary phase was the data modeling procedure, following the workflow proposed by Cassandra.[6] A **conceptual data model** was ideated, with the aim of highlighting the entities present in the model and their relationships between them. Afterwards, a **query model** was created, by analyzing the possible user stories of the end system. The two models were then merged into a **logical model,** which defines the final tables present in the model. Finally, a **physical model** was derived from the logical, by integrating the types for each variable. Initially, *UUID* and *date* types were assigned to specific variables. However, Flink was not able to bind its own types to them; therefore, they were assigned as *TEXT* type instead.
Moreover, a critical phase should have been the refinement and optimization of the physical model through partition calculations and bucketing. However, this step was skipped since an estimate of the data flow was required, and being this a prototype, we did not have such knowledge.

[6](https://cassandra.apache.org/doc/latest/cassandra/data_modeling/index.html , last visited 9/06/2023)

B. Data Generation & Streaming

As previously mentioned in Section I, we relied on offline data sources. In order to simulate a real data stream, some streaming utilities were included. In particular, the files contained in the *stream* library allow for the handling of multiple streams executed in parallel on separate threads. This enhances the scalability by allowing the presence of multiple and different sources and streams. Each streamer saves its state in a key stored in a Redis cache, allowing for its starting, disabling, enabling and interrupting (scripts *stream_[start/enable/disable/interrupt].sh*) by simply changing the value of the respective key. Streams continuously send transactional data to distinct Kafka topics.

C. Stream Processing

The processing of the data streams are performed by a Flink job which retrieves the transactional data from the Kafka topics, parses them, applies fraud detection and in parallel updates the relative account's balance. Finally, data is saved in the Cassandra database. Code related to the stream processing can be found in the *pipeline* folder.

1. Data Parsing

In order to allow multiple data sources, a parsing procedure is necessary as a first step of the processing phase. This procedure is intended to convert the format of the different sources to the target format. This enhances the scalability by allowing multiple formats. Therefore, if a new source will be needed, only the development of a relative parser will be needed. JSON configuration files in the *parsers* folders are needed, in order to indicate source and target data types. During this process, as transactions may contain only information about the account, a Redis cache has been integrated with the objective to provide a quick lookup solution for retrieving account-related information such as account type, user ID, and bank ID. The Redis database containing the accounts information is populated from Cassandra's *accounts* table at the start of the processing phase.

2. Fraud Detection

After a transaction is parsed, a fraud detection algorithm is performed. The output should be a boolean indicating if the transaction is fraudulent or not For simplicity, a simple threshold is set on the transaction amount, above which the transaction is set as fraud. A fraud confidence percentage is also returned, in this case set just to 0 for transactions below the threshold

and 1 for the ones above. This step can be easily integrated by using unsupervised techniques provided by Flink ML, or using existing techniques such as Benford's Law[7].

3. *Balance Update*

In parallel to the fraud detection procedure, the balance of the account relative to the input transaction is updated. This is done by analyzing the *balance_after* key of the incoming message, often used in fraud detection datasets. A possibly more robust alternative would be to look at the transaction amount, at the direction of the transaction (inbound or outbound) and updating the balance accordingly by performing a select query.

4. *Save to Cassandra*

The final parsed transaction is finally saved in Cassandra using the Cassandra sink of PyFlink. This has been one of the most challenging aspects of the entire implementation, since no indication was found on how to integrate it in python, neither in GitHub examples nor in the official documentation.

D. *Entities add*

An additional utility that is proposed is the possibility to add entities such as User, Account, Bank from a guided command line interface (*add.sh*). This sends the relative entity structure (modeled in the folder *model* and with JSON target type information in *parser* folder, as previously mentioned) to the Kafka broker, which is processed by a Flink job and saved into the Cassandra database.

E. *Dashboards*

Some final visualization tools have been proposed in order to provide two distinct point of views:
1) a Grafana dashboard intended for the use of the cooperative banks, providing some statistics and real-time overview of the incoming transactions. It could also integrate alerting functionalities (e.g., the sending of an email as soon as a fraud is flagged); unfortunately, the initial setup must be done by hand since it includes the authentication to the Cassandra and Redis data sources connectors, but a *grafana.json* model of the dashboard can be imported.
2) a flask and dash app intended for a user-based client, which can show the list of transactions for each distinct user. This can be scaled for the usage of a single app after a user login.

[7] https://www.tandfonline.com/doi/epdf/10.1198/tast.2009.0005?needAccess=true&role=button , last visited 4/06/2023

IV. Results

The final prototype represents a fully working pipeline including the phases of data streaming, parsing, fraud detection, accounts balances update, and final visualization. Additionally, side utilities were also deployed such as the possibility of adding new users, banks, or accounts.
The entire pipeline was finally deployed on a local old computer used as a server, showing the final Grafana dashboard.
The working demo can be reached at the link https://bdt.davidecalza.com/ with user ***guest*** and password ***BDT2023*** (for a glimpse of it, see **Fig. 2**).
For example, on this server, Flink needed a local installation because the relative docker container looped in the starting phase, probably due to low memory constraints. Only one Cassandra node was used for the same reason.
The intention of the project was to provide a system that was scalable, in order not to limit the implementation on this example, and the complete documentation of the code allows for further improvements.
By carefully searching for the entire GitHub repositories, this seems to be (one of) the first working examples of a Flink pipeline using Cassandra as a sink (besides the tests of the PyFlink library using a simple collection as a source). Therefore, it may result in a good contribution to the (Py)Flink community.

V. Conclusions

There are some further improvements not yet mentioned that are worth considering for future works on this project. Low RAM systems may encounter performance issues when handling resource-intensive tasks or running multiple processes concurrently. This can potentially impact the effectiveness of a fraud detection system. To address these issues, steps can be taken to optimize resource usage, such as implementing more efficient algorithms or reducing the memory footprint of the application. Additionally, hardware upgrades, such as increasing the amount of RAM, can also help alleviate these problems and enhance the system's capabilities. Furthermore, implementing general error checking and testing procedures, such as preventing the addition of already existing users, can help improve the reliability and accuracy of the system. There is also potential for better leveraging technologies such as Flink, Kafka, and Redis to enable distributed processing, which can improve the scalability and performance of the system.
This is because distributed processing involves dividing a large computational task into smaller subtasks that can be executed concurrently on multiple machines or nodes within a network. This approach can significantly improve the scalability of a system, as it allows the

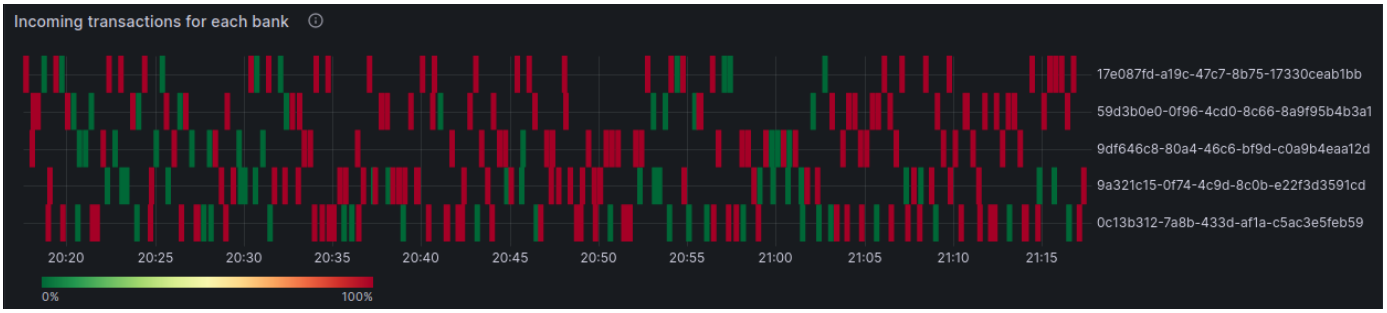

Fig. 2 Example of transaction data real-time monitoring

system to handle larger workloads by adding more machines or nodes to the network. As the workload increases, additional resources can be added to the system to maintain performance and prevent bottlenecks. This is in contrast to a single-node system, where the performance is limited by the resources of a single machine. By distributing the workload across multiple machines, a distributed system can achieve higher levels of scalability and performance.

In the initial design of the pipeline, an intermediate Redis Pub/Sub persistence layer was intended to be included between the parsing and fraud detection/balance update procedures, yet missing Redis connector for PyFlink did not allow us to include it. Therefore, this inclusion may improve the overall stability and robustness.

Considerations can also be made for the Flask/Dash application. Indeed, improving its performance by optimizing callbacks, implementing caching mechanisms, optimizing data loading, enhancing the user interface by bettering front-end components, improving error handling, testing and debugging, gathering user feedback, and staying updated with Dash's latest updates and best practices. Another improvement could be building the Dash application by implementing the structure with a more class-based approach.

Additionally, limitations to this project may refer to the timing performance. Therefore, a performance test should be executed to check whether a time optimization should be considered (for example, testing the actual time from transaction sending to the time instant it is saved into the database).

To conclude, another notable point to consider is that using synthetic data avoids problems regarding privacy and security of data and information, either of users and bank companies. Discussion about defining a system within general law and data protection frameworks such as the General Data Protection Regulation (GDPR) can be made. Specifically, although GDPR mentions fraud detection as a special case, it requires enterprises to design their solutions in such a way that the flow of personal data is protected by Technical and Organizational Measures (TOMs) at any point in time during the processing (namely, all concrete provisions taken by a company to guarantee the security of personal data.). This means that companies must ensure that all concrete provisions taken by a company to guarantee the security of personal data are satisfied while still effectively detecting and preventing fraudulent activities.

For further details about the whole development procedure, see the GitHub Repository README and the code documentation.

References

[1] Abramova, Veronika, and Jorge Bernardino. 2013. 'NoSQL Databases: MongoDB vs Cassandra'. In *Proceedings of the International C* Conference on Computer Science and Software Engineering*, Porto Portugal: ACM, 14–22. https://dl.acm.org/doi/10.1145/2494444.2494447 (June 9, 2023).

[2] Ajredini, Artan. 'Database Cashing In-Memory with Redis NoSQL Databases'.

[3] Gurusamy, Vairaprakash et al. 2017. 'The Real Time Big Data Processing Framework Advantages and Limitations'. *International Journal of Computer Sciences and Engineering* 5(12):305–12.https://www.ijcseonline.org/full_paper_view.php?paper_id=1621 (June 9, 2023).

[4] Laitos, Tietojenkäsittelytieteiden. 2015 'ADVANTAGES OF DOCKER', University of Jyväskylä.https://jyx.jyu.fi/handle/123456789/48029

[5] Rad, Babak Bashari, Harrison John Bhatti, and Mohammad Ahmadi. 2017. 'An Introduction to Docker and Analysis of Its Performance'.

[6] Shree, Rishika, Tanupriya Choudhury, Subhash Chand Gupta, and Praveen Kumar. 2017. 'KAFKA: The Modern Platform for Data Management and Analysis in Big Data Domain'.

Sitography

[1] 'Dash Documentation & User Guide | Plotly'. https://dash.plotly.com/ (June 9, 2023).

[2] 'Everything You Need to Know About Grafana -'.https://www.skedler.com/blog/everything-you-need-to-know-about-grafana/ (June 9, 2023).

[3] 'GitHub - Redis/Redis: https://github.com/redis/redis (June 9, 2023).

[4] 'Kaggle'.https://www.kaggle.com/ (June 9, 2023).

[5] 'Welcome to Flask — Flask Documentation (2.3.x)'. https://flask.palletsprojects.com/en/2.3.x/ (June 9, 2023).

[6]'Data Modeling | Apache Cassandra Documentation'. https://cassandra.apache.org/doc/latest/cassandra/data_modeling/index.html (June 9, 2023).