Waseem Tannous: 207866328
Zinat Abo Hgool: 206721714
Julian Farraj: 208155580

Cloud Architecture HW3 - Kafka

Build and run:

In order to run the multiple components that we developed using multiple languages, we dockerized each element for an easier way of running them. To run the code:

- Extract files from zip file.
- cd to the folder.
- Run command: docker-compose pull
- Run command: docker-compose build
- Run command: docker-compose up -d
- Wait about 20 seconds after the “up” command is finished working so that all the services are up and connected successfully. This needs a couple of seconds.
- Open http://localhost:3000 in your browser.

To shut down the app, run the command “docker-compose down”.

Kafka and Zookeeper:

Kafka and Zookeeper were run using docker containers pulled from the official images from dockerhub. Kafka was given multiple environment variables including one that specifies the topics we want to create, partitions, and a replication factor. In our use case, we created two topics:

- “recentChange” topic with one partition and a replication factor of one. This is used to store all the messages received from Wikipedia. It has a string key which is the URL/language of the message and a JSON value that contains the message itself.
- “output” topic with one partition and a replication factor of one. This is used to store the statistics and all the data that the stream outputs. It has a string key which is the URL/language of the message and a JSON value that contains relevant data and metadata about the type of stats that this message contains.

In production, we think that it is better to change the partition number and replication factor of each topic so that it would be easier to save the data into different segments of the topic and filter them according to some other key while also having multiple redundant copies of each record for increased reliability in case of failure. Also, it would be a good idea to create multiple brokers of Kafka for increased redundancy.

Producer:

A small python program that consumes the messages from the “recentChange” topic in Wikipedia, reorders and keeps only relevant data that will be used in the processing stream.

Stream Workers:

In order to consume these messages, we used the Kafka Streams framework in java. The stream was split to be able to process and create multiple stats. We used a combination of filters, key-value maps, and grouping techniques to reorganize the data for each specific use. Also, we introduced windowing in order to create stats across multiple timeframes (last hour, day, week, month). In some cases, we joined tables of different stats that have the same timeframe to have a more rich output of the stream. In the end, all messages created had metadata about the type of stats that each message contained which included the timeframe, language, bot/user filtering, and the data itself. finally, the messages were published to the “output” topic to be saved.

DataViewer and Server:

We decided to take the project a step beyond the scope of the project’s requirements and create a website that shows the stats produced by the stream in real-time. We used React to create a client which we called DataViewer. The logic is to consume the messages from Kafka, store them in the component’s state and render the new state with the updated stats. To be able to do that, we had to create a server that consumes these messages from Kafka and sends them to the client using Server-Sent-Events. The server was implemented using Flask and contains a Kafka-consumer object that subscribes to the Wikipedia “recentChange” topic, consumes the messages, and sends them to the client. In React, we connected to the server and when a new message arrives, it is stored in the state which triggers a re-render event on the component with the new data.

A minor detail: In the DataViewer, we only rendered the data of the “www.wikidata.org” URL. This can be changed by deleting the if-statement from the “App.js” file line 19. We choose to do so as there are lots of stats being produced which sometimes creates a delay in the rendering and some other issues related to the huge number of messages received. A relatively powerful computer will have no problem displaying all stats of all URLs. We chose to do it this way to be able to easily run the app across all PCs regardless of how powerful they are.

<u>Question 2.d:</u>

Our application is obviously bound by the internet connection between our pc and Wikipedia's server. In a better deployment, we would have chosen to run our app locally on the server in which the Wikipedia events are hosted to eliminate any connection issues and interrupts and create all stats offline on the server.