Data Intensive Computing - Project Report.

Egill Friðriksson Gard Aasness Maximilian Georg Kurzawski

October 21, 2023

0 Introduction

This is the project report for the final project of the course "Data Intensive Computing" at KTH Royal Institute of Technology. You can find the github link with all of the code **here**. If you want to look at the deployed versions **here** is the Frontend deployed on a server.

0.1 Introduction

This project is a data pipeline that takes data from an endpoint provided by **straeto.is**, stores it in a database, and then displays the data in a frontend. The data contains information, (location, next stop, etc.) about the busses in Iceland. The project is divided into 7 parts:

1 Deployment

The deployment works as follows:

1. The datagatherer is being deployed as a docker container on a server,
2. The Kafka cluster is set up as a service on the same server; it will only be restarted with the updated configuration,
3. The Flink-Kafka-Consumer is being deployed as a docker container on the same server,
4. The PostgreSQL script is only being copied on the server, it is not being executed (We assume that no changes are made to the database structure),
5. The BatchDataClean is currently not being deployed,
6. The Webserver and Frontend are being combined into one docker container and also deployed.

2 Data Gathering

The data gathering part is implemented in the `data_gatherer.py` script. The script is run with the following command:

```
python3 data_gatherer.py
```

The script will gather data from the following sources:

- **straeto.is**

The xml-data returned by the API will be taken literally and stored as a string in the Kafka Cluster.

3 Kafka Cluster

To enable the asynchronous communication between the different parts of the project, a Kafka cluster is used. The cluster is running on a single machine and is started with the following commands:

```
zookeeper-server-start.sh Kafka/zookeeper.properties

kafka-server-start.sh Kafka/server.properties
```

This project only provides the `server.properties` for the Kafka cluster, as well as the `zookeeper.properties` for the Zookeeper server.

The Kafka cluster is configured to run on port 9092, and the Zookeeper server is configured to run on port 2181. Feel free to change this if you want a different configuration.

4 Data Stream Processing

The data stream processing part is implemented in the Flink-Kafka-Consumer. It takes strings from the Kafka cluster and parses them into a case class. The case class is then used to insert the data into a PostgreSQL database. The Flink-Kafka-Consumer is started with the following command:

```
sbt run
```

The data structure of the case class is as follows:

```
case class BusData(
  dev: String, time: String,
  lat: String, lon: String,
  head: String, fix: String,
  route: String, stop: String,
  next: String, code: String,
  fer: String)
```

What the different fields mean can be found in the **Straeto API documentation**.

Notes:

- We get a timestamp for each individual bus, but we only take the timestamp of the xml-response. That allows us to easier check the data for validity, but it also means that we lose some precision. We could have taken the timestamp of each individual bus, but that would have made the data validation more complicated.
- The additional fields "dev" and "fer" are not documented in the Straeto API documentation. Dev stands for "device" and fer stands for "ferry".

5 Data Storage

The data storage part is implemented in the PostgreSQL database. The SQL script we used to create the database can be found in the file `db.sql`. To initialize the database, run the following command:

```
psql -U <username> -f db.sql
```

6 (Optional and NOT FINISHED) Data Deletion

The data deletion part is implemented in the BatchDataClean. It is a Work in progress Spark script that is supposed to take the elements from the DB and delete the ones that are older than X hours/ A certain timestamp. The script is started with the following command:

```
sbt run
```

7 Data Visualization

The data visualization part is implemented in two parts: The frontend and the Webserver.

7.1 Webserver

The Webserver is implemented in the `webserver.py` script. It is a simple Flask server that serves the data from the PostgreSQL database. Essentially, it is a REST-API that returns the data in JSON format. The Webserver is started with the following command:

```
python webserver.py
```

The API currently has one endpoint: "/bus/getData". It takes no parameters and returns the latest data for all buses in the database.

Example return:

```
{
  "data": [
    [
      "Sat, 14 Oct 2023 17:47:55 GMT",
      "2-D",
      "2",
      "64.121208350",
```

```
      "-21.898571667"
    ],
  ]
}
```

7.2 Frontend

The frontend is implemented as a simple JavaScript application. It gets the data from the Webserver and displays it on a map.

8 Further Work

Ideas for further work:

8.1 Testing

The whole system is more or less a prototype that, during development, has not been tested enough. We have observed crashes and found fixxes for several problems which could have been avoided with proper test-driven development.

8.2 Monitoring

Currently, if a docker container crashes or otherwise gets shut down, the idea is to re-trigger the workflow to deploy a new container or manually connect to the server and run the command to restart the container. A monitoring tool as simple as a python script that runs every n seconds and reports the state if a container has failed could help reducing down time.

8.3 Data Deletion

We added the data deletion part because we wanted to gather knowledge about using spark and it's batch processing. The script that we build is not finished but we still wanted to elaborate on the idea behind it. We are aware that we could have solved the "issue" of getting rid of the old data by a simple sql statement, but we still wanted to experiment with having the batch processing in place that clears up the data. Turns out this is not a good idea. Extracting the data from the DB to "filter" it and then based on the filter either reenter or delete the obsolete ones is not very usefull.

8.4 Frontend

First of all, the current setup gathers data way beyond the scope of the frontend. The frontend is only able to display the most recent data. Therefor we gather a lot of data but are only actually able to display the data.

An idea of ours that, due to timeconstraints, is not yet implemented would be to extend the API to return all available timestamps, then returning this to the frontend so that the frontend can make specific calls to the API requesting the busses for this timestamp.

8.5 Data Gatherer

The data gatherer and the flink component sometimes crashed without a reasonable error message. After turning on debug logging we discovered a problem where the XML-messages could not be properly parsed. We tried a lot of different approaches and after enabling an initial parsing on the data gatherer we noticed that the data sometimes contained either incomplete or empty messages from the straeto endpoint. We are not sure wether or not this is a mistake on our end or on theirs, but the current solution is an initial parsing of the xml-response and a discarding of the message if not properly parseable.

9 Conclusion

Overall we really enjoyed working on this project. It gave us a lot of use full knowledge as well as a place to combine all the skills that we learned.