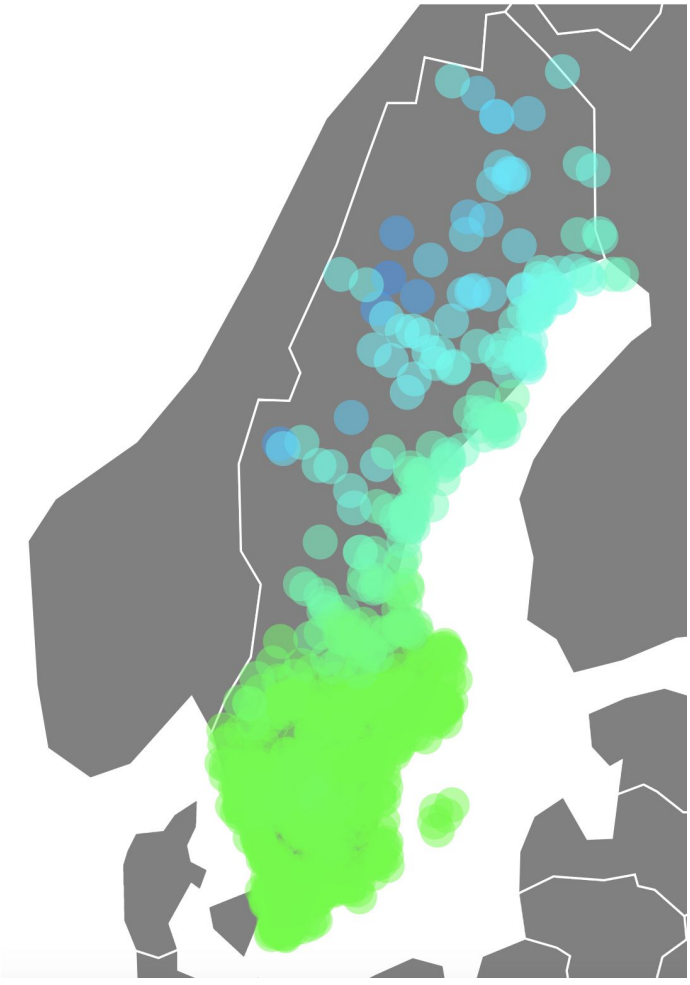
ID2221 - Data Intensive Computing

Project Report: SWEDEN-BIG-WEATHER

Course coordinator:
Amir H. Payberah

Group: Sigrún & Vincent
Sigrún Arna Sigurðardóttir <sasig@kth.se>
Vincent Paul Lohse <vplohse@kth.se>

25.10.2020



1 Introduction

Due to geographical or urban conditions, weather stations are not distributed perfectly equally across countries. For the same reasons, weather stations are not to be found on every inch on a given piece of land. When creating heat maps for countries, however, it is required to calculate a value for every single pixel of the map. Also, since heat maps are meant to show continuous color flows, the calculated temperature values cannot simply assume the temperature of the next weather station but must evaluate temperatures of different weather stations around it. Given live data from multiple weather stations in Sweden, we aim to create a live heat map, where every pixel represents a calculated temperature. We believe this is a good topic, since temperature data is available in abundance and is constantly changing, making it a good use-case for big-data streams.

2 How to run

It is possible to run the entire project using docker-compose. All steps to run the code are listed in the README of the repository. The application code repository including the Docker images can be found here: https://github.com/olapiv/sweden-big-weather

3 Implementation

We set up docker-compose for the following external services: Zookeeper, Kafka, Cassandra.

Then we implemented the following services: A Producer service (Producer.scala), A Spark Streaming service (Engine.scala), and a Node.js server (app.js, kafka.js), which would serve a D3 web-page (index.html) and also be listening to a websocket connection.

3.1 Producer.scala

In Producer.scala we get the weather data from a REST API provided by Open Weather. We then parse the JSON result returned from the endpoint to get the data we need to publish to the Kafka Broker.

3.2 Engine.scala

In Engine.scala we set up a connection to a Cassandra where we then create a keyspace and a table. The keyspace is called weather_keyspace and the table is called city_temps. The table consists of 3 columns of the type “double”: lat, lon and temperature. The lat & lon represent a composite primary key. We then set up a connection with Kafka via map and subscribe to the topic city-temperatures to be able to consume the messages from the topic. Each message that is consumed is then saved to Cassandra.

We then query Cassandra to fetch all recent weather-station temperatures for a fixed-size block around the coordinates of our most recent weather-station-update. Within this fixed size block, we build a grid, of which for every point on it we interpolate the temperature, using the temperatures of all weather station temperatures in the block. When we have calculated these temperatures we publish the results to a Kafka Broker.

Unfortunately, it turned out that this part was more difficult than expected. More specifically, we were not able to the grid temperatures in our blocks whenever we had a weather-station update. This was due to the following factors:

1) A DStream consists of multiple rdds. We wanted to perform calculations using Spark Datasets for every single temperature update. This meant that we had to do multiple calculations for every DStream.
2) Non DStream-related code is only executed once in Spark Streaming. So the only chance of doing something continuously is through DStream.map() or DStream.foreachRDD().
3) It is not possible to create rdds from a worker („nested rdds“). That meant that we could not load our data from Cassandra into a (distributed) Dataset/Dataframe within a DStream.map()/.foreachRDD() operation. When passing the SparkSession to the workers, we would receive a NullPointerException.

It seemed to us that this meant that we would have to query the data from Cassandra to each worker locally, using a local Cassandra session and neither a Spark DataFrame or DataSet (which would try to distribute the data). Without knowing what other open-source frameworks exist, we were quite sure that querying and analyzing the data with CSQL would be quite complicated. Furthermore, in real scenarios, this would potentially also overload the worker with data and defeat the point of having distributed nodes. We did not implement this, since we also felt that it would miss the purpose of the course.

3.3 app.js & kafka.js

Kafka’s protocol consists of binary TCP. Since browser clients do not allow raw TCP connections, unless packaged with HTTP, it was quite difficult finding a way to connect the browser to Kafka directly. We have therefore created a NodeJS server which forwards the Kafka messages to a browser client using a web socket. In Kafka.js we have a consumer that subscribes to the topic grid-temperatures (or city-temperatures).

3.4 index.html

D3 map that visualizes the temperature of every coordinate in our heat map via the data that is received through the WebSocket.

Conclusion

In conclusion, we see that we could have implemented a more simple pipeline. However, the learning process from this project has given us a great experience on how to set up a data pipeline using Kafka, Spark, and Cassandra.