University of Southern Denmark

Big Data and Data Science Technologies, (e20) - Project

MSc in Software Engineering - 1st semester

Deadline for submission: December 23rd 2020

BDDST - Project

Student:
Kasper Møller Nielsen
kaspn17@student.sdu.dk
06/10-1998
Exam no.: 453489

Student:
Rasmus Bang Poulsen
rapou15@student.sdu.dk
25/01-1995
Exam no.: 426796

Student:
Rasmus Emil Albrektsen
raalb17@student.sdu.dk
07/02-1995
Exam no.: 453517

Student:
Troels Zink Kristensen
tkris17@student.sdu.dk
30/04-1997
Exam no.: 455427

Lector:
Fisayo Caleb Sangogboye
fsan@mmmi.sdu.dk

Lector:
Jakob Hviid
jah@mmmi.sdu.dk

Abstract

The aim of this project is to create a flow of big data using concepts and technologies introduced by the course Big Data and Data Science Technologies. The requirements of the project include data selection, insights to be created from the datasets, an architecture containing the chosen technologies including Hadoop Distributed File System (HDFS), two Apache Spark libraries, and a method to visualize the processed data.

The project topic is provided by Marjan Mansourvar and is about analyzing COVID-19 related tweets posted on Twitter. The desire is to gain insights by comparing the number of COVID-19 related tweets with the number of reported COVID-19 cases from the same country and investigate possible tendencies. The data is collected from Kaggle as historical data, and therefore not as live data from Twitter.

Several technologies have been implemented in order to fetch the data from Kaggle and visualize it in line graphs and heat maps on a web page. The used technologies are Flume for data ingestion. Kafka is used as a transport bus between Flume and Spark Streaming. Processing of the data is done with Spark SQL; thereafter the processed data is saved on the HDFS. These files are loaded into a Hive database.

Using the data saved on the Hive database, visualized line graphs and table heat maps depict possible tendencies showing occurrences of spikes in daily amount of tweets and reported COVID-19 cases relatively close to each other.

Preface

This project is the work of Kasper Møller Nielsen, Rasmus Bang Poulsen, Rasmus Emil Albrektsen and Troels Zink Kristensen.

The project is associated with the course: Big Data and Data Science Technologies in MSc in Software Engineering on the 1st semester in autumn 2020 at the University of Southern Denmark in Odense. The course covers 10 ECTS-points.

The project started September 4th and was concluded on December 23rd with the submission of this report.

Thanks to the topic owner, Marjan Mansourvar, for guidance and help regarding the topic.

Reading Guide:

Literature references are shown with $^{[x,p/c.y]}$ (literature number, page/chapter number) and refers to the reference list in section 10. Code specific expressions are formatted using the `menlo font`. An overview of the submission can be found in section 11.
The PDF is interactive, which means that it is possible to click on references to chapters, figures and code examples to jump directly to them.

Table of Contents

Table of Figures

Table of Code Examples

1 Introduction

This project is composed of multiple technologies introduced in the course: Big Data and Data Science Technologies. A dataset had to be selected and collected for analysis. This dataset had to be used together with Spark and a Hadoop Distributed File System (HDFS) cluster. The data is meant to be ingested into the system, where it will be analyzed and stored.

The selected topic is *Idea 1* composed by Marjan Mansourvar. This topic is concerning the COVID-19 (COrona VIrus Disease 2019) pandemic, which is in full effect at the moment of writing. More specifically, this topic is centered around the use of COVID-19 tweets from Twitter. The dataset is collected from Kaggle, which has tweets from a certain time period.

COVID-19 is much worse than first anticipated. Countries around the world have different approaches to how the virus is handled. The aim of this project is to examine the number of COVID-19 tweets in a certain country, and compare them to the number of reported cases within the same time frame. Figure 1 visualizes the massive number of total reported cases compared to the population of each country in the world. The figure was collected December 16th.[1]

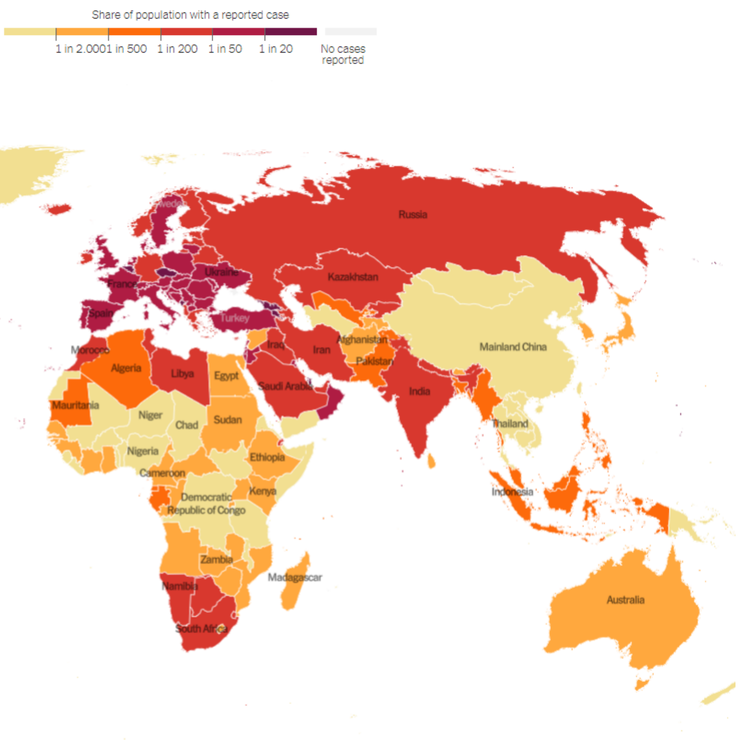


Figure 1: *Number of cases per inhabitant for every country in the world*

2 Data Analysis Background

This section describes the reasoning behind the choice of topic for the project, as well as the sought out solution to the specific problem presented by this project.

2.1 Data Background

The reasoning behind choosing to work with the specific data and topic, centres around the current worldwide events transpiring as this report is written, and to help investigate trends that could create insights into the outbreaks occurring around the world.

The COVID-19 virus has proven that it can easily spread with a rapid pace, making it a global pandemic. The outbreaks are most often determined when the government of a country has gathered the daily amount of positive tests for the virus. Using data gathered about COVID-19 related tweets, would allow to investigate if a possible correlation between possible outbreaks and surges in posted tweets is present.

2.2 The Problem and Solution

The specific problem to be solved is determining a rising numbers of reported COVID-19 cases in specific countries using COVID-19 related tweets. With this approach, the intention is to create separate table heat maps and line graphs that can be used to investigate any correlation. The problem statements is as follows:

"How can you track the rising number of reported cases within a country using tweets with hashtags related to COVID-19?".

This project aims to solve the problem through the use of a dataset from Kaggle. A query runs daily for the collection of tweets related to COVID-19. These tweets will be used along with the country of origin, so that they can be counted towards the specific statistic for each country.

2.3 Data Insights

The tweets from the dataset contain two important characteristic of information, which are the date of the tweet and the country code of the tweet author. These two will be the foundation for the insights acquired from the specific dataset. Using the date and country code, it is intended to analyze frequencies of COVID-19 related tweets occurring in a specific time frame for each available country. This is then compared to the frequency of reported COVID-19 cases in the same country using information on reported cases gathered from an API.

Once the initial goals of tracking individual countries have been achieved, it is intended to create heat maps that will show surges in both outbreaks and reported cases within a specific time frame. The heat maps will be table heat maps, showing some countries, which are known to have surges in the number of reported cases. Using this strategy, the ambition is to create a map that can, within a chosen time frame, show where COVID-19 is surging globally, and then be compared to the heat map showing the amount of COVID-19 related tweets happening concurrently.

3 Technologies

This section contains general descriptions of the technologies used within this project.

3.1 Apache Hadoop

A distributed file system (DFS) is a file system stored on a server. However, the data is accessed and processed as if it was stored locally on the client's computer. Files are shared between users as they would have been on a local machine.
Apache Hadoop is a DFS used for data storage and data analysis, and it is open-source. Hadoop uses a series of computers to handle large amounts of data, which is referred to as big data. The data storage part of Hadoop is called the Hadoop Distributed File System (HDFS), and the data analysis part has several methods.[2,c.1]

3.1.1 Hadoop Distributed File System

The Hadoop Distributed File System differs from other DFS in several ways. The HDFS is designed to run on commodity hardware and at the same time being highly fault-tolerant. This makes it a relatively cheap solution compared to other DFSs. Furthermore, the HDFS is perfectly suitable for large datasets, or what is referred to as big data. Files on the HDFS can be huge in size, such as terabytes. The system can consist of thousands of servers, where each machine has small parts of the data. As this is commodity hardware, it is possible that at least one machine (or more) is faulty; however, HDFS has been designed with these problems in mind. The Hadoop Distributed File System uses horizontal scaling, by having many machines using cheap commodity hardware, instead of vertical scaling, which is fewer machines with expensive high-end hardware. Horizontal scaling is cheaper than vertical, especially if a machine stops working.

Master and Worker

The master and worker relationship is a well-known architecture used in a variety of systems. In the cluster of an HDFS-architecture, the master and workers are referenced to as the NameNode and DataNodes. There is only one NameNode, and this node is responsible for managing and regulating the namespace of the file system and file access by clients. All other nodes in the cluster usually have one DataNode attached to it. These are responsible for managing the storage on the nodes.[2,c.3]

3.2 Apache Spark

Apache Spark is defined by Bill Chambers & Matei Zaharia as:

> ”...a unified computing engine and set of libraries for parallel data processing on computer clusters.”[3,p.14]

The three main components that define Apache Spark are: unified, computing engine and libraries, and they are described in the following three sections.

3.2.1 Unified

Apache Spark is unified because it gathers multiple types of data processing under the same roof. Everything from data loading, SQL-queries on the data, to machine learning, while also allowing the user to create their own data processing tools. Apache Spark will automatically combine multiple processing tasks under one scan of the data, decreasing the execution overhead.[3,c.1]

3.2.2 Computing Engine

Apache Spark is only designed for data loading and data processing, and not for permanent data storage. This is where Spark differs from e.g. the Hadoop Distributed File System, where as mentioned in section 3.1.1, is both a storage system and a computation engine. It can be used together with Hadoop and many other storage systems like Apache Kafka, see section 3.4.[3,c.1]

3.2.3 Libraries

Spark supports the use of the built-in libraries as well as third-party libraries. Two of these built-in libraries are Spark Streaming and Spark SQL, which are described in sections 3.2.4 & 3.2.5, respectively.[3,c.1]

3.2.4 Spark Streaming

The Spark Streaming library from Apache Spark is a technology that enables processing of live data streams. Live data is data that is fetched directly from a source, meaning that the technology will continuously fetch data from the source. However, Spark Streaming does not fetch the data directly, but through a data ingestion technology. Kafka is used in this project to ingest data into Spark Streaming. A discretized stream (DStream) is created from Spark Streaming, which consist of a continuous stream of data, provided by Kafka. This stream of data is called RDDs, which are immutable and distributed datasets.[4]

3.2.5 Spark SQL

Spark SQL is an SQL-technology within Apache Spark, which enables configuration of structured data into data frames. This data can be used in the HDFS. Spark SQL can be used as standard SQL, in which certain values from a table can be fetched.[5]

3.3 Flume

Apache Flume is a service that focuses on delivering data by streaming it to its clients, where this data could be live data. Some of the major components that are relevant to this project are: source, sink and channel. Source is where the data that Flume streams to the sinks originates from. There are many types of sources, and they can be anything from reading data sent to a port Flume is listening on, data retrieved from Twitter's API for specific hashtags, or data sent from Kafka. The sink is where Flume puts the data, and therefore referred to as the counterpart of a source. A sink could be an HDFS or Kafka. The channel is the connection between the source and the sink, and it defines the durability of the data delivery, as it can be configured to have the data in memory or in mass storage.[6]

3.4 Kafka and Zookeeper

Kafka is a publisher/subscriber system where publishers produce events with a topic, and subscribers can subscribe to these topics and then consume the events from the topics. In the middle, a number of brokers relay events from the publishers to all topic subscribers.[7] Zookeeper is a service for managing configurations and synchronization between applications. At the time of writing this project, running Zookeeper is necessary in order to run Kafka, but in the future this will no longer be the case.[8]

3.5 Hive

Apache Hive is a data warehouse component used with Hadoop to process the data received by the HDFS. Hive is a non-traditional RDBMS (Relational Database Management System). RDBMS is complex to work with, when a HDFS is used, as traditional data warehouses use SQL to extract the data. However, Hive was developed to handle the complexity with Hadoop, as with Hive it is possible to write HQL-queries (Hive Query Language) to extract data from the HDFS.[2,c.17] In this project the data has to be visualized in a certain way. It could be a graph, where the data is acquired by writing queries with Hive on the HDFS.

4 Architecture

The overall architecture of the system can be seen in Figure 2. It depicts the data flow from one end to the other, going left to right. The data flow starts from Flume ingesting data into Kafka. Kafka forwards the data to Spark, which does computations over the data, and puts it into the HDFS. The data is then retrieved from the HDFS using Hive, and provided to a web page used for visualization of the data.

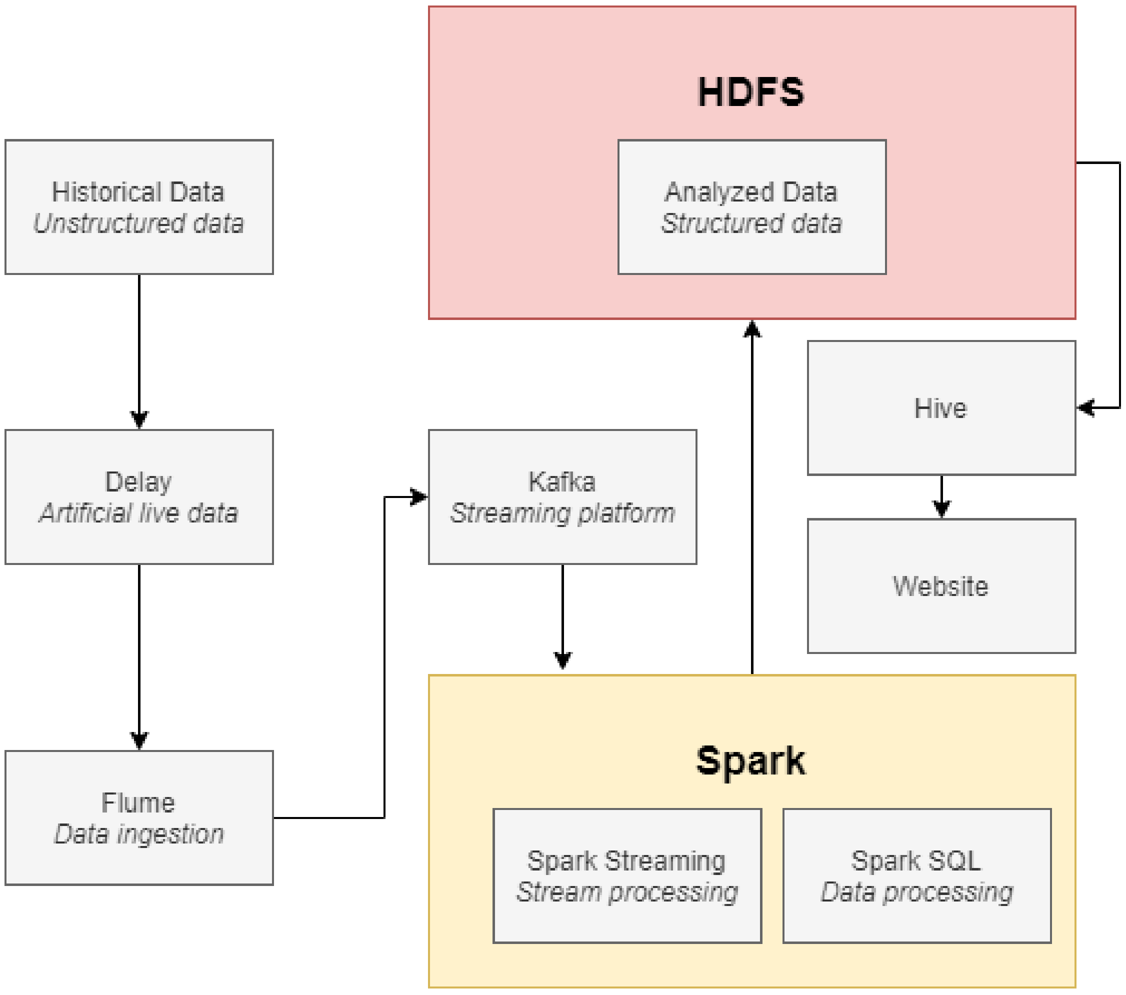


Figure 2: *The overall architecture*

Flume - Historical data is downloaded from Kaggle, where a dataset of tweets related to COVID-19 has been found.[9] Due to this being historical data, and not live data, a delay is implemented in a Python script to make artificial live data. The script feeds the data to Flume as a tweet in JSON-format. Afterwards, Flume sends the tweets to Kafka as a producer using a specific topic.

Kafka - The JSON-objects from Flume are received in the Kafka broker and relayed to consumers subscribed to the specified topic. This is further sent to Spark Streaming, which acts as a consumer.

Spark - Spark Streaming carries out stream processing, and Spark SQL fetches and processes the relevant information from each JSON-object, which is tweet_id, date and country_code. This is written to a .csv-file and sent to the HDFS.

HDFS - The .csv-files are saved on the HDFS, where the data is spread onto the datanodes.

Hive - The .csv-files are put into a table on the Hive database to be acquired from the web page.

Web Page - The data from the Hive database is collected, along with numbers of COVID-19 cases from the specific countries from an external API, and are visualized on the web page in line graphs and table heat maps.

5 Implementation

In this section, the implementation of the architecture is explained in detail regarding the data flow. Code examples and configuration files are provided to gain a better understanding of the implementations.

5.1 Flume

As briefly mentioned in section 4, the data ingested into Flume is sent from a Python script that reads the data from files on the local hard drive, which are downloaded from the dataset on Kaggle. The Python script is described in section 5.1.1, and the configuration of Flume is described in section 5.1.2

5.1.1 Data Ingestion

The Python script is executed on node0 of the cluster, just like Flume, and reads the data from the directory called "dailies", which contains the files with the tweets for each day. The script reads each file, with the extension .tsv, and iterates through each line, adding the tab separated data to a JSON-data-template, see Code example 1. This is necessary because Flume expects to receive the JSON-data in the specific format from line 1-6, where the the body tag represents the tweet data. When the data is added to the JSON-format, it is sent as a POST-request to Flume. This means that the script works as an HTTP-source for Flume.

```
1   json_format = [{
2       "headers" : {
3           "timestamp":"", "host":""
4           },
5       "body":""
6       }]
7   root_dir = 'dailies/'
8   json_body = {
9           "tweet_id":"", "date":"", "time":"", "lang":"", "country_code":""
10          }
```

Listing 1: *JSON data format*

5.1.2 Flume Configuration

Flume is configured to have an HTTP-source as a source and Kafka as a sink. This is shown on line 10 and 14 in Code example 2, which depicts the configuration file for Flume. Flume listens to port 8989 for any HTTP-requests, see line 12. Because Flume receives JSON from the source, it is necessary to define the timestamp, host and host name using interceptors, if these are not already set in the Python script, see line 5-8. The topic that

Kafka uses to know where to send the data to is defined on line 15, and the IP and port of the Kafka service is defined on line 18. The channel type is of `memory`, which means that it is saved in memory, and thus has the risk of memory loss, should it crash. The channel has a capacity of 1000 events. This is defined on line 22 and 23.

```
 1  kafkatest.sources = http-source
 2  kafkatest.sinks = kafka-sink
 3  kafkatest.channels = ch3
 4
 5  kafkatest.sources.http-source.interceptors = i1 i2
 6  kafkatest.sources.http-source.interceptors.i1.type = timestamp
 7  kafkatest.sources.http-source.interceptors.i2.type = host
 8  kafkatest.sources.http-source.interceptors.i2.hostHeader = hostname
 9
10  kafkatest.sources.http-source.type =
     ↪  org.apache.flume.source.http.HTTPSource
11  kafkatest.sources.http-source.channels = ch3
12  kafkatest.sources.http-source.port = 8989
13
14  kafkatest.sinks.kafka-sink.type = org.apache.flume.sink.kafka.KafkaSink
15  kafkatest.sinks.kafka-sink.kafka.topic = test-events
16
17  kafkatest.sinks.kafka-sink.channel = ch3
18  kafkatest.sinks.kafka-sink.kafka.bootstrap.servers = 10.123.252.225:9092
19  kafkatest.sinks.kafka-sink.kafka.flumeBatchSize = 5
20  kafkatest.sinks.kafka-sink.request.timeout.ms = 120000
21
22  kafkatest.channels.ch3.type = memory
23  kafkatest.channels.ch3.capacity = 1000
24
25  kafkatest.sources.http-source.channels = ch3
26  kafkatest.sinks.kafka-sink.channel = ch3
```

Listing 2: *Flume configuration file*

5.2 Kafka

As mentioned in the previous section, Flume is used to ingest the data as the producer for Kafka. Kafka has to be set up to run with a consumer as well. This consumer is Spark Streaming.

However, before the technologies are able to produce and consume, the services ZooKeeper and Kafka have to be started with the terminal commands shown below.[10]

- `zookeeper-server-start.sh config/zookeeper.properties`
- `kafka-server-start.sh config/server.properties`

A topic also has to be created; this is done using the command below. It creates a topic called `test-events` within the Kafka broker.

- `bin/kafka-topics.sh --create --topic test-events --bootstrap-server localhost:9092`

A direct stream (DStream), which is a continuous sequence of RDDs, is created to connect Kafka with Spark Streaming. The streaming context `ssc` is initialized within Spark Streaming; this is explained in section 5.3.[11] Kafka is set up to listen on the `test-events` topic within the broker as seen in Code example 3. Afterwards, the incoming data is processed with a `lambda`-function, which saves `tuple[1]` into `tuple`. This is done to only use the relevant data from Flume, as an array with two elements is received, where the first element is empty.

```python
dks = KafkaUtils.createDirectStream(ssc, ["test-events"],
 ↪  {"metadata.broker.list":"10.123.252.225:9092"})
data = dks.map(lambda tuple: tuple[1])
...
data.foreachRDD(lambda rdd: sendRecord(rdd))
```

Listing 3: *Kafka integration*

For each RDD received from the producer, a function `sendRecord()` is called, which handles all the processing of the data.
Code example 3 is part of a script that consumes data from Flume with Kafka and Spark Streaming. The data within `sendRecord()` is processed using Spark SQL. This will be further explained later on. This entire script called `kafka-spark-streaming.py` will be referenced to as the *consumer-script*; the entire file can be seen in Code example 12 in the Appendix.

5.3 Spark

Spark Streaming enables processing of live data streams across the nodes, as explained in section 3.2.4. In order for Spark Streaming to receive data from Kafka, a .jar-package has to be submitted together with Spark, when running the consumer-script.[12] See line 1 in Code example 4.

As seen in Code example 4, a `SparkContext` is created, which is used to connect to the Spark cluster that can be used to create RDDs.[13]. Furthermore, within the `SparkContext` `sc` a `StreamingContext ssc` is created to initialize the streaming service together with a batch interval of two seconds. In the bottom of the consumer-script, the `ssc` is started to run the streaming context parallel with Kafka to listen to the topic. The streaming context listens every two seconds (defined by the batch interval), and awaits termination by the user.

```
os.environ['PYSPARK_SUBMIT_ARGS'] = '--jars /home/hadoop/spark-streaming-kafka-0-8-assembly_2.11-2.4.7.jar --conf spark.sql.catalogImplementation=hive pyspark-shell'
...
sc = SparkContext(appName="PythonSparkStreamingKafka")
ssc = StreamingContext(sc,2)
...
ssc.start()
ssc.awaitTermination()
```

Listing 4: *Spark Streaming integration*

The technologies to produce and consume the data are now initialized and configured. It is now possible to process and edit the data with Spark SQL. First, a `SparkSession` has to be initialized in order to create data frames containing the data. See Code example 5.

Spark Streaming receives data from Kafka, which is configured in a JSON-format with five columns. Only three of them are relevant for the visualization later on, which are `tweet_id`, `date` and `country_code`. Therefore, a data frame is created with these three columns and corresponding rows.

Within the `sendRecord()`-function, a lot of functionality is implemented. The consumed `rdd` is an array with one item in JSON-format. The `tweet_id`, `date` and `country_code` are extracted from the item in the RDD on line 11-14.

A `count` has been implemented in such a way that rows are added to the `df` data frame 1000 times before it is saved to the HDFS in a .csv-file. The data frame is reset after it is saved.

```
1   sparkSession = SparkSession.builder.appName("SparkToHDFSWrite").getOrCreate()
2   ...
3   columns = ['tweet_id', 'date', 'country_code']
4   df = sparkSession.createDataFrame([("1","2","3")], columns)
5   count = 0
6   countName = 0
7   def sendRecord(rdd):
8       try:
9           ...
10          json1 = json.loads(rdd.take(rdd.count())[0])
11          tweetId = str(json1["tweet_id"])
12          date = str(json1["date"])
13          countryCode = str(json1["country_code"])
14          row = [(tweetId, date, countryCode)]
15
16          if count == 0:
```

```
                df = sparkSession.createDataFrame(row, columns)
                count += 1
            elif count < 1000:
                newRow = sparkSession.createDataFrame(row, columns)
                df = df.union(newRow)
                count += 1
            else:
                ...
                currentDate = date
                df = sparkSession.createDataFrame(row, columns)
                count = 0;
                countName += 1

        except Exception as e:
            traceback.print_exception(*sys.exc_info())
            pass

data.foreachRDD(lambda rdd: sendRecord(rdd))
...
```

Listing 5: *Spark SQL integration*

5.4 HDFS

All .csv-files are saved on the HDFS. In this way, the files are distributed across every node connected to the HDFS. The path to the folder *twitterFiles* on the HDFS containing the .csv-files has a unique name, `countName`, to distinguish each file, see Code example 6. This creates a folder with two files. One being a .csv-file and a file that indicates if the operation was a success. It is the reason why lines 6-10 are implemented. The desired file in the folder is retrieved and added to `fileList2`, where it is decoded and stripped off unwanted characters.

```
1   ...
2   else:
3       print("Writing to HDFS")
4           path = "hdfs://10.123.252.225:9000/user/hadoop/twitterFiles/" +
            ↪   str(countName)
5           df.coalesce(1).write.csv(path)
6           path2 = "/user/hadoop/twitterFiles/" + str(countName)
7           fileList = subprocess.Popen("hdfs dfs -ls " + path2 + " |  awk
            ↪   '{print \$8}'", shell=True, stdout=subprocess.PIPE,
            ↪   stderr=subprocess.STDOUT)
8           fileList2 = []
9           for line in fileList.stdout.readlines():
10              fileList2.append(line.decode("utf-8").rstrip())
11      ...
```

Listing 6: *Saving files to the HDFS*

5.5 Hive

The Hive database is created within the Hive command line interface (HCLI), and is called `twitterdb`. A table called `tweets` within the Hive database is created with three columns corresponding to the data frame from Spark SQL in the previous section. These columns are `tweet_id`, `date` and `country_code`. Code example 7 shows the command for creating the table.

```
sparkHive.sql("CREATE TABLE IF NOT EXISTS tweets (tweet_id STRING, `date`
↪  STRING, country_code STRING) ROW FORMAT DELIMITED FIELDS TERMINATED BY
↪  ',' LINES TERMINATED BY '\n' STORED AS TEXTFILE")
```

Listing 7: *Creating a table on the Hive database*

In order to use Hive, it has to be configured in a certain way. This can be seen in Code example 8. Hive is configured together with Spark Streaming and Kafka on line 1, with the command `catalogImplementation=hive`. This command has to be set in order to use Spark SQL cooperatively with HQL (Hive Query Language). Lines 3-4 configures the connection to the Hive database. Furthermore, the Spark warehouse is connected with the Hive warehouse as well. In this way, it is possible to make HQL-queries within SQL to load data into the table on the Hive database. The Thrift Hive Server is configured in order to allow remote clients to connect to the Hive database; this has to be done, because Hive is running on node1 and this script is running on node2.[14] If the Thrift Hive Server was not configured, it would default to a local Hive database on node2 instead of `twitterdb`.

```
1  os.environ['PYSPARK_SUBMIT_ARGS'] = '--jars
   ↪  /home/hadoop/spark-streaming-kafka-0-8-assembly_2.11-2.4.7.jar --conf
   ↪  spark.sql.catalogImplementation=hive pyspark-shell'
2  ...
3  sparkHive = SparkSession.builder.master("local").appName("SparkToHive")
4  .config("spark.sql.warehouse.dir",
   ↪  "/user/hive/warehouse").config("hive.metastore.uris",
   ↪  "thrift://10.123.252.223:9083").enableHiveSupport().getOrCreate()
5  ...
6  sparkHive.sql("LOAD DATA INPATH
   ↪  \'hdfs://bddst-group8-node0.uvm.sdu.dk:9000" + fileList2[-1] + "\' INTO
   ↪  TABLE tweets")
```

Listing 8: *Hive integration*

When the .csv-files have been saved into the HDFS, it is possible to load them into the table on the Hive database. As the files are not stored locally on the datanode, but on the HDFS, the files have to be loaded as seen on line 6. The host name in conjunction with the last filename within `fileList2` is used to load the file into the table `tweets`.

5.5.1 Loading Data from the Hive Database

In order to visualize the data in graphs and heat maps on the web page, the data from the Hive database has to be fetched and saved into new files. This will result in simple acquisition for the web page to use. A new script is created to handle this, as can be seen in Code example 9. The entire script can be seen in Code example 13 in the Appendix.

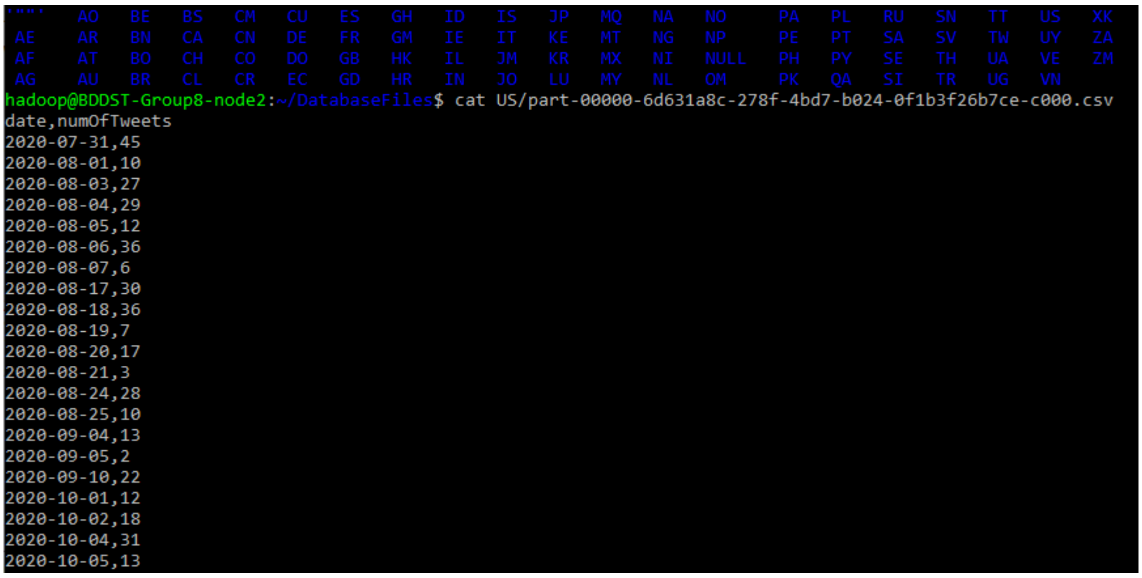
The necessary configurations for acquiring data from the Hive database are the same as in Code example 8. A new data frame, `df`, is created by selecting everything from `tweets` where the `country_code` is not equal to `NULL` or an empty string. Furthermore, it is ordered by `country_code` and `date`. See line 2.

A new data frame, `df_new`, is created with two columns containing `date` and `numOfTweets`. Every tweet is counted for every day for each country. Each country is saved within a separate folder. Every row within the data frame `df` is iterated through to separate the countries and the dates within the countries. If the next `country_code` is the same as the one before, a row will be added for each new date. If the next `date` is the same as the one before, the `numOfTweets` will be incremented by one. When a new `country_code` is received, then the file is written locally on node2.

```
1  ...
2  df = sparkHive.sql("SELECT * FROM tweets WHERE (country_code != 'NULL' AND
   ↪  country_code != '') ORDER BY country_code, date")
3  ...
4  columns = ['date', 'n_tweets']
5  df_new = sparkSession.createDataFrame([("date","numOfTweets")], columns)
6  country_temp = ""
7  date_temp = ""
8  count_tweets = 0
9
10 for row in df.rdd.collect():
11     country = str(row["country_code"])
12     date = str(row["date"])
13     if country_temp == "":
14         country_temp = country
15         count_tweets += 1
16     elif country_temp == country:
17         ...
18     else:
19         ...
20         df_new.coalesce(1).write.csv("/home/hadoop/DatabaseFiles/" +
           ↪  country_temp)
21         df_new = sparkSession.createDataFrame([("date","numOfTweets")],
           ↪  columns)
22         country_temp = country
```

Listing 9: *Loading data from Hive database*

Figure 3 shows the folder called `DatabaseFiles` containing folders for every `country_code`. Within every folder, two files are located as explained before in section 5.4. The .csv-file within a folder contains the required data.



```
'""'   AO   BE   BS   CM   CU   ES   GH   ID   IS   JP   MQ   NA   NO     PA   PL   RU   SN   TT   US   XK
 AE    AR   BN   CA   CN   DE   FR   GM   IE   IT   KE   MT   NG   NP     PE   PT   SA   SV   TW   UY   ZA
 AF    AT   BO   CH   CO   DO   GB   HK   IL   JM   KR   MX   NI   NULL   PH   PY   SE   TH   UA   VE   ZM
 AG    AU   BR   CL   CR   EC   GD   HR   IN   JO   LU   MY   NL   OM     PK   QA   SI   TR   UG   VN
hadoop@BDDST-Group8-node2:~/DatabaseFiles$ cat US/part-00000-6d631a8c-278f-4bd7-b024-0f1b3f26b7ce-c000.csv
date,numOfTweets
2020-07-31,45
2020-08-01,10
2020-08-03,27
2020-08-04,29
2020-08-05,12
2020-08-06,36
2020-08-07,6
2020-08-17,30
2020-08-18,36
2020-08-19,7
2020-08-20,17
2020-08-21,3
2020-08-24,28
2020-08-25,10
2020-09-04,13
2020-09-05,2
2020-09-10,22
2020-10-01,12
2020-10-02,18
2020-10-04,31
2020-10-05,13
```

Figure 3: *The .csv-file for the United States of America*

5.6 Web Page

The web page is used for the visualization of the data and is run locally, communicating with the remote cluster for the retrieval of data. The web page is created with a simple HTML-setup, using `<div>` to separate each item on the page. Also found on the web page, is a button, "Search date", that is used once two dates are selected. These two dates are the starting date and the end date for the wanted time period. Also possible to select, is a specific country wanted for line graph visualizations.

The web page uses a JavaScript-file for logic, called `main.js`. This file contains all calls and handling needed for the web page. The web page uses Node.js and Browserify, for the convenience of adding libraries like the ones required to use Vega for visualization (using Node.js), and then bundling them up with Browserify to be used with browser-side JavaScript.

To visualize the data, a library called `vega-embed` is used. This allows for inserting JSON-formatted Vega-"specs" into the HTML from a JavaScript file. The use of specs can be seen in Code example 10, where a line graph is created to depict the number of new tweets per day for a selected time period. The functions that create specifications to embed on the web page, receive `data`, once called, which is a JSON-array holding objects that are to be used for the visualization.

```
function embedTweetsLineGraph(data) {
    var spec = {
        "width": 400,
        "height": 400,
        "data": data,
        "mark": "line",
        "encoding": {
            "x": {"title": "Date", "timeUnit": "utcmonthdate", "field":
            ↪  "date", "type": "ordinal"},
            "y": {"title": "Tweets posted", "field": "numOfTweets",
            ↪  "type":"quantitative"}
        }
    }
    embed('#visTweetsLineGraph', spec);
}
```

Listing 10: *Creating a spec for embedding a Vega-view*

Both heat maps and line graphs are created once the button on the web page is clicked. The button has a listener that calls the `searchButtonClicked()` function in `main.js`.

5.6.1 Collecting Data Through a Node.js Server

Trouble was experienced when collecting data for visualization. Connecting with Hive and trying to retrieve data from the cluster, proved difficult to achieve within the time frame. At first there were issues using local forwarding, and then later on when using different Node.js libraries for requesting Hive-data. As a backup plan, implemented due to the lack of time, a decision was made to extract data using a Node.js server called `dirwalker.js`. This server is connected to from the web page, using the following URL through local port forwarding when establishing the SSH connection:
`/getTwitterData/SelectedStartDate/SelectedFromDate/CountryCode`
Once the server receives a request, a function called `getCountryTweetData()` is executed with the parameters given in the URL; this function is shown in Code example 11.

```
function getCountryTweetData(countryCode, fromDate, toDate) {
    var d1 = Date.parse(fromDate);
    var d2 = Date.parse(toDate);
    var data = [];
    const directoryPath = "DatabaseFiles";

    var countryFiles = fs.readdirSync(directoryPath + "/" + countryCode);
    var file = null;
    for(var i = 0; i < countryFiles.length; i++) {
        if(countryFiles[i].split('.').pop() === "csv") {
            file = countryFiles[i];
```

```
                break;
            }
        }

        let json = csvToJson.fieldDelimiter(',').getJsonFromCsv(directoryPath +
         ↪  "/" + countryCode + "/" + file);
        for(var j = 0; j < json.length; j++) {
            console.log("INSIDE FOR LOOP");
            var d = Date.parse(json[j].date);
            if(d >= d1 && d <= d2) {
                data.push(json[j]);
            }
        }
        return data;
    }
```

Listing 11: *Function that returns JSON data for a specific country code*

The function `getCountryTweetData()` takes the parameters `countryCode, fromDate` and `toDate`, which are the dates selected on the web page for collecting data in between. The `countryCode` is the ISO-standard country code, e.g. DK for Denmark, which is sent from the web page after a country has been chosen there as well. Once the function is called, it parses the dates in to `Date` objects, which are used for comparison of a collected tweet date and the ones needed. This is to filter out unwanted data. The function uses `fs.readdirSync()` to find the files in a folder named after the specific country code for a country. Thereafter, it finds the file with the .csv extension, since this file holds the correct data. The `let json` variable is used to store JSON-data converted from the earlier mentioned .csv-file. After this, the date-comparison is used to send back the correct data.

6 Data Visualization

This section describes what was used to visualize the data, as well as what was visualized, and how.

6.1 Vega

For the visualization of the data, Vega was selected. Vega is easy to integrate with Node.js in combination with HTML, and was therefore selected as the best option. Vega refers to itself as ”visualization grammar”, where in JSON-format it is declared how a graph should be displayed. Code example 10 in section 5.6 shows how a specification is defined, when creating a ”view” for a line graph.[15]

6.2 Line Graphs

Two line graphs were used for the visualization of data, one created using the Twitter data, and the other using data on confirmed cases of COVID-19. The two line graphs are shown in Figure 4 and Figure 5, respectively, below. The Twitter data is gathered from the cluster, and the data on confirmed cases is gathered from an API called ”COVID19API”, which sources data from John Hopkins CSSE.[16]

6.2.1 Line Graph of Daily Tweets Posted

The line graph in Figure 4 shows how many COVID-19 related tweets have been posted daily for the United States of America in the period of October 1st 2020 to October 8th 2020.

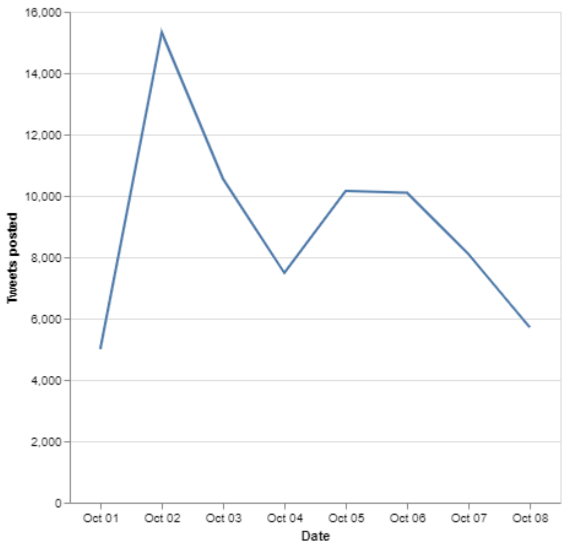


Figure 4: *Line graph of tweets in USA*

6.2.2 Line Graph of Daily Confirmed COVID-19 Cases

The line graph in Figure 5 below, shows daily confirmed COVID-19 cases for the United States of America in the period of October 1st 2020 to October 8th 2020.

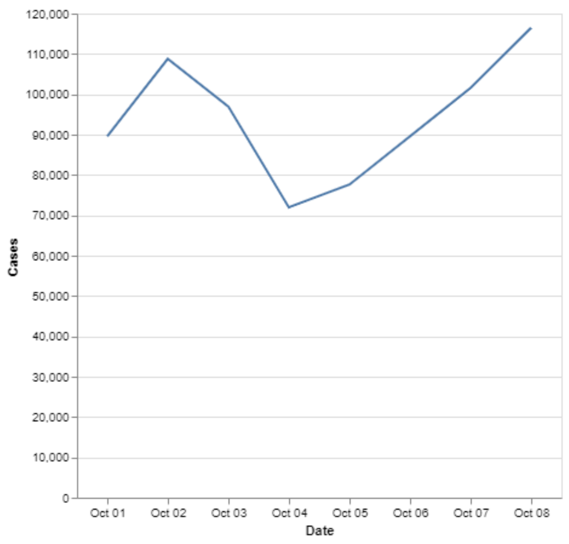


Figure 5: *Line graph of cases in USA*

6.3 Table Heat Maps

Two table heat maps were chosen to visualize a possible overall correlation tendency between COVID-19 cases and tweets posted for a given time frame. Both heat maps use population data gathered from an API called "World Population API".[17] The heat map of confirmed cases also uses the previously mentioned "COVID19API".

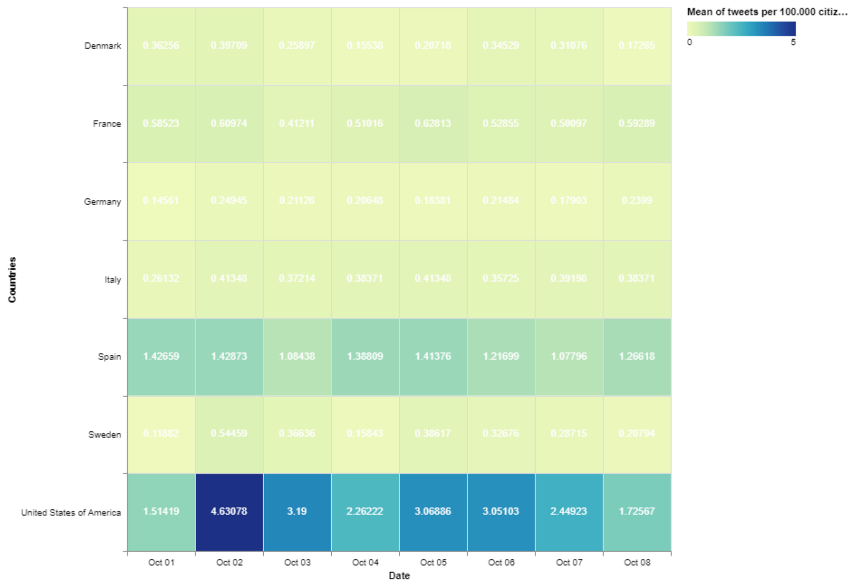
The data for the two heat maps have been scaled in terms of a country's population, and then multiplied by 100,000 to reflect either cases or tweets per 100,000 citizens. The numbers depicted for a country every day in the time period have therefore been calculated with the following equation:

$$\frac{\text{Daily tweets or cases}}{\text{Country population}} * 100,000$$

The heat maps display the number of cases or tweets per 100,000 citizens in the table as a mean of the overall numbers. For the heat maps, seven countries were chosen for data collection, since not all countries in the cluster data were possible to gather from "COVID19API" and "World Population API".

6.3.1 Table Heat Map of Daily Tweets

The table heat map below in Figure 6 shows the selected countries, and the mean of gathered daily tweets for the specific chosen time period.

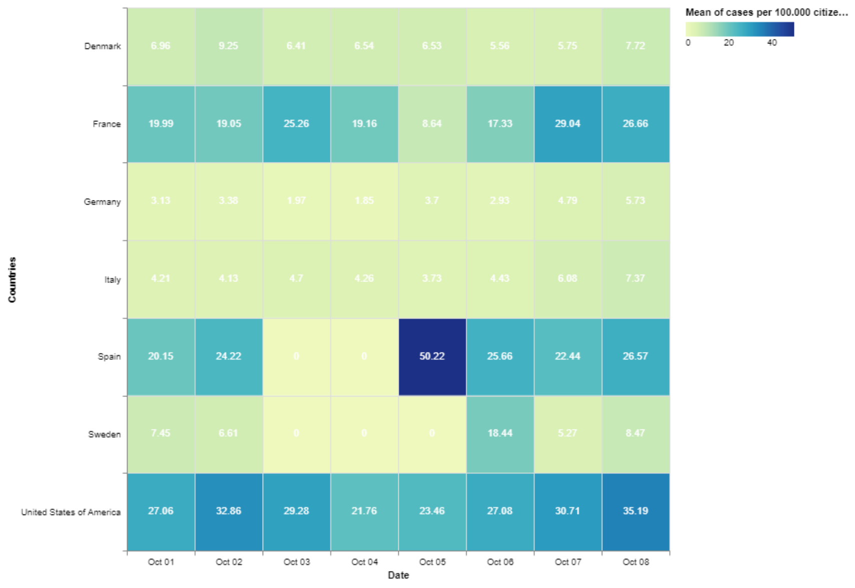


Countries	Oct 01	Oct 02	Oct 03	Oct 04	Oct 05	Oct 06	Oct 07	Oct 08
Denmark	0.36256	0.39709	0.25897	0.15538	0.20718	0.34529	0.31076	0.17265
France	0.58523	0.60974	0.41211	0.51016	0.62813	0.52855	0.50097	0.59289
Germany	0.14561	0.24945	0.21126	0.20648	0.18381	0.21484	0.17903	0.2399
Italy	0.26132	0.41348	0.37214	0.38371	0.41348	0.35725	0.39198	0.38371
Spain	1.42659	1.42873	1.08438	1.38809	1.41376	1.21699	1.07796	1.26618
Sweden	0.11882	0.54459	0.36636	0.15843	0.38617	0.32676	0.28715	0.20794
United States of America	1.51419	4.63078	3.19	2.26222	3.06886	3.05103	2.44923	1.72567

Figure 6: *Heat map of tweets*

6.3.2 Table Heat Map of Daily Confirmed COVID-19 cases

The table heat map in Figure 7, shows the selected countries, and the mean of daily cases reported for the chosen time period.



Countries	Oct 01	Oct 02	Oct 03	Oct 04	Oct 05	Oct 06	Oct 07	Oct 08
Denmark	6.96	9.25	6.41	6.54	6.53	5.56	5.75	7.72
France	19.99	19.05	25.26	19.16	8.64	17.33	29.04	26.66
Germany	3.13	3.38	1.97	1.85	3.7	2.93	4.79	5.73
Italy	4.21	4.13	4.7	4.26	3.73	4.43	6.08	7.37
Spain	20.15	24.22	0	0	50.22	25.66	22.44	26.57
Sweden	7.45	6.61	0	0	0	18.44	5.27	8.47
United States of America	27.06	32.86	29.28	21.76	23.46	27.08	30.71	35.19

Figure 7: *Heat map of cases*

Full scale of both tables can be seen on Figure 8 and Figure 9 in section 11.3 (Appendix).

7 Discussion

In this section the dataset and the implementation of the technologies used for the project are discussed in regards to how well the technologies are implemented, and how crashes will affect them. Alternative technologies to the ones chosen for the project are described, along with reasoning why they were not chosen.

7.1 Dataset

The dataset used has some inconsistencies both inside of the data, and with the range of the data. For the date 17-09-2020, there was a total of 624,873 tweets, where only 16,134 of them had a country code assigned. The range of the data used is the range of the first date to the last date, where the data is usable, which is from 26-07-2020 to 12-10-2020. The dataset has data all the way from 22-03-2020, but it does not contain a country code until 26-07-2020. The dataset has also not been updated with new data since the 12-10-2020, making that the last usable date.

The full dataset has five files for each day; three containing frequently used terms, one containing both tweets and retweets, and one only containing tweets. The latter one is referred to as "clean" by the creator. Only the "clean" data is used. This was decided due to the massive amounts of data that would come from not filtering the others out.

7.2 Technologies

7.2.1 Flume

The implementation of Flume was first intended to be outside of the cluster, but due to trouble of port forwarding from the cluster to local ports, it was moved to node0 on the cluster. The reasoning behind having Flume outside of the cluster, was guidance from the lecturer Jakob Hviid, that it would be the easiest way of doing it, without having any impact on its availability, other than the computer would have to be running when Flume is used.

Using a Python script with historical data as a placeholder for live Twitter data makes it necessary to tweak the configuration of Flume, should it be connected directly to Twitter's API later. Using the historical data is not only necessary because it provides an overview of the past, but also because the number of daily tweets exceed the monthly limit Twitter has for standard access to its API, which is 500,000.[9][18]

Flume could be configured to run on the two datanodes (node1 and node2) instead of the namenode (node0). All Flume requires to run is to have the source files along with the configuration file for Flume and Java installed. Doing this would allow for Flume to ingest even more data into the system, but would also require something that will make sure that the two Flume services do not ingest the same data into Kafka. This is more applicable for situations where the data comes from different places, and does not have the chance of being duplicated.

7.2.2 Spark

Spark Streaming has been implemented in such a way that the entire application is distributed across the cluster. This is done on line 1 within Code example 4 in section 5.3 with a .jar-package. It enables the functionality of receiving data from Kafka into Spark Streaming. This configuration was necessary to do for Python either in the file as in the code example or within the terminal console when executing the script. If Java or Scala were used, the .jar-package could have been included instead.

When the `tweet_id`, `date` and `country_code` are extracted from the tweet data, the tweets where the `country_code` are `NULL` could have been omitted. This would reduce what is stored on the HDFS, and make the retrieval of the usable data faster.

7.2.3 HDFS

The HDFS has been set up according to the configuration file handed out during the course: Big Data and Data Science Technologies. The HDFS was used in this project to store the .csv-files containing data ingested by Flume and processed by Spark. Furthermore, the Hive warehouse containing the used database is stored on the HDFS as well.
Nevertheless, the .csv-files could have been stored locally on the datanode instead of the HDFS to save writing time onto the file system. This could have been done, as the .csv-files are added to the Hive database directly afterwards. The Hive database is saved on HDFS to ensure faults if the server went down, as the database would be lost if it was saved locally and faults had occurred.

7.2.4 Hive

The Hive database is stored on the HDFS. In order to connect to the Hive warehouse, a Thrift Hive Server was configured. In this way, it was possible to make HQL-queries on node2, while Hive being installed on node1. It also made it possible to enable HQL-queries from Spark SQL. If the Thrift Hive Server had not been configured, it would still be possible to use HQL, but it would not be possible to use the queries on the created database, as the specific Hive warehouse would not be known for this datanode. However, this configuration would only allow to connect to the Hive database, `twitterdb`, and load data into it. In order to do ordinary HQL-queries, such as `SELECT`, another configuration had to be implemented, as shown in line 1 (`hiveImplementation=hive`) in Code example 8.

As mentioned in section 7.2.2, the HDFS has tweets, where the `country_code` is `NULL`, stored on it. This data is unusable and have to be sorted out. Therefore a script called `hive-database.py` was created in order to remove the `NULL` values. The script also ordered the data in the table according to `country_code` and `date` to make it easier for the web page to fetch the data in a concise manner. The removal of the `NULL` values would not be necessary, if it was done earlier in the data flow.

The original plan was to fetch data directly from the Hive database into the web page in JavaScript. However, it was complex and could possibly have taken too much time to figure out. Therefore, that plan was changed. Instead the .csv-files created in the `hive-database.py` script was saved locally on node2 to fetch these files in JavaScript by using another technology called Node.js. In this way, the web page does not access the data from the Hive database in the HDFS directly, but instead locally through node2.

7.3 Alternative Technologies

Alternative technologies could have been implemented instead of the ones used in this project. Two alternatives are explained below.

7.3.1 Apache Storm

Apache Storm could have been used as an alternative to Spark Streaming if more time had been used to explore it. Storm uses micro-batch processing instead of batch processing. Furthermore, Storm is faster than Streaming and can be used across multiple languages besides Java, Scala and Python.[19]
However, Spark Streaming was the one that Jakob Hviid suggested to use, as it seemed more simple to implement. Time was already running out, which concluded that only one technology for stream processing was explored and implemented.

7.3.2 HBase

HBase is a an Apache technology that, like Hive, provides a way of storing data onto the HDFS. HBase stores its data as a NoSQL-store running on top of the HDFS and executes its operations in real-time, providing random access capabilities as the HDFS is not built to handle random read/write operations. This means that HBase is better suited for real-time querying than Hive and could have been used as an alternative.[20]

7.4 Web Page

The implementation of the web page is far from ideal; however, it was deemed a low priority since this is only the means of visualizing the data. The web page only runs locally, and all logic is placed within the `main.js` file, which has resulted in a huge file with many functions. To use the Vega library called `vega-embed`, which makes it possible to embed Vega views on the web page, Browserify was deployed. It allows the required libraries to be bundled up and used from browser-side JavaScript. This did, however, create difficulties in the form that communication between HTML and the `main.js` file was impossible without creating listeners in the code.

7.5 Visualization

7.5.1 Line Graphs

The use of two line graphs was decided with the intention that when comparing the two, it would be possible to track possible surges in both tweets and cases within the same time frame. As of now, one has to look at both of the line graphs to investigate this correlation. If this was to be improved upon, the data could instead be formatted in a way that would put both datasets in the same single line graph, containing two y-axes on either side of the graph. This would make it easier to investigate any correlation between datasets; however, it would require a way of scaling the data to be compared without compromising the integrity of the datasets upon comparison.

7.5.2 Table Heat Maps

The table heat maps are created with the purpose of showing comparable tendencies in the rise of reported COVID-19 cases and tweets posted. The issue with these heat maps is the fact that both use population of a country as a means of making the two comparable. This approach does not consider the number of people who are actually on Twitter in comparison to the total population, and this is the biggest threat to the validity of the comparison.

8 Conclusion

This project sought out to create a system capable of ingesting data from Twitter that could then store, analyze and visualize the data for the purpose of getting insights. These insights would possibly show a correlation between the number of tweets using specific hashtags in a country and that country's amount of reported COVID-19 cases within a given time period. To achieve this, an existing dataset containing tweets as historical data using COVID-19 related hashtags has been used to simulate live data by sending tweets to the system with a delay between each tweet. These tweets are then sent to a Flume service that ingests the data into the system by sending it through a Kafka broker to a consumer. The consumer uses Spark Streaming for handling the incoming data. Spark SQL is then used to put the data into data frames, which can then be stored on the HDFS and loaded into the Hive database, where it can then be fetched for visualization.

The given insights visualized in the line graphs and heat maps show a possible tendency between the number of COVID-19 cases and the number of tweets within the same country. According to the line graphs, countries like the United States of America, Spain and Denmark have a spike of COVID-19 cases occurring close in time to a spike of COVID-19 related tweets. The line graphs for the United States in Figure 4 and Figure 5 show this tendency. The heat maps depict that countries with a high number of cases has a fairly high number of tweets as well, which is especially visible in the United States of America, France and Spain (see Figure 6 and Figure 7).

9 Future Works

Historical data is used as the dataset in this project. A delay is implemented into the producer to simulate live data, meaning one tweet is sent every 0.1 seconds. Real live data could have been set up through the Twitter API, which would then result in data produced every time a user posts a tweet. Possible outbreaks from COVID-19 could then have been analyzed to see if the number of tweets would increase accordingly. This was not implemented; however, it could be added in the future with minor modifications to the Flume configuration. Although possible current outbreaks cannot be examined by using historical data, it still provided enough information to achieve the overall aim of the project.

This project uses a script, `hive-database.py`, to fetch and store data from the Hive database locally for the web page to use. This was done, as there was not enough time to figure out how to fetch the data directly from the Hive database in JavaScript to visualize on the web page, as the complexity of this integration seemed greater than first anticipated. As of now, a Node.js server called `dirwalker.js` collects the data to use on the web page. If it had been possible to do it directly in JavaScript, then the data flow could have been shortened by omitting the aforementioned script and server.

The visualization of the data has dependencies on external APIs for data regarding population and confirmed COVID-19 cases. This data could be stored on the HDFS, especially the data regarding COVID-19. The ”COVID19API” has certain features behind a pay wall; this required certain workarounds, slowing down the data collection. This would be solved by storing the data on the HDFS. It would also require another Flume service for fetching the data, and a new consumer for sorting the relevant data as well as storing it on the HDFS.

10 References

Notes

1. The New York Times, “Coronavirus World Map: Tracking the Global Outbreak,” The New York Times, 28-Jan-2020. [Online]. Available: https://www.nytimes.com/interactive/2020/world/coronavirus-maps.html. [Accessed: 16-Dec-2020].

2. T. White, Hadoop: the definitive guide. Sebastopol, CA: O’Reilly, 2015.

3. B. Chambers and M. Zaharia, Spark: the definitive guide: big data processing made simple. Sebastapol, California: O’Reilly Media, 2018.

4. “Spark Streaming Programming Guide,” Spark Streaming - Spark 2.4.0 Documentation. [Online]. Available: https://spark.apache.org/docs/2.4.0/streaming-programming-guide.html. [Accessed: 16-Dec-2020].

5. “Spark SQL amp; DataFrames,” Apache Spark. [Online]. Available: https://spark.apache.org/sql/. [Accessed: 16-Dec-2020].

6. “Apache Flume,” Apache Software Foundation. [Online].
Available: https://cwiki.apache.org/confluence/display/FLUME. [Accessed: 17-Dec-2020].

7. “Introduction,” Apache Kafka. [Online]. Available: https://kafka.apache.org/intro. [Accessed: 17-Dec-2020].

8. “Welcome to Apache ZooKeeper™,” Apache ZooKeeper. [Online]. Available: https://zookeeper.apache.org/. [Accessed: 18-Dec-2020].

9. Banda, Juan M., Tekumalla, Ramya, Wang, Guanyu, Yu, Jingyuan, Liu, Tuo, Ding, Yuning, Artemova, Katya, Tutubalin, Elena, Chowell, Gerardo, A large-scale COVID-19 Twitter chatter dataset for open scientific research - an international collaboration, vol. 29, Zenodo: Zenodo, 2018. [Dataset]. Available: https://www.kaggle.com/imoore/covid19-complete-twitter-dataset-daily-updates. [Accessed: December 16, 2020].

10. “Apache Kafka Quickstart,” Apache Kafka. [Online]. Available: https://kafka.apache.org/quickstart. [Accessed: 17-Dec-2020].

11. “Class DStream,” DStream (Spark 3.0.1 JavaDoc), 28-Aug-2020. [Online].
Available: https://spark.apache.org/docs/latest/api/java/org/apache/spark/streaming/dstream /DStream.html. [Accessed: 16-Dec-2020].

12. “Spark Streaming + Kafka Integration Guide,” Spark 2.2.0 Documentation. [Online].
Available: https://spark.apache.org/docs/2.2.0/streaming-kafka-0-8-integration.html. [Accessed: 16-Dec-2020].

13. “Class SparkContext,” SparkContext (Spark 3.0.1 JavaDoc), 28-Aug-2020. [Online].
Available: https://spark.apache.org/docs/latest/api/java/org/apache/spark/SparkContext.html. [Accessed: 16-Dec-2020].

14. “Apache Software Foundation,” HiveServer - Apache Hive - Apache Software Foundation. [Online]. Available: https://cwiki.apache.org/confluence/display/Hive/HiveServer. [Accessed: 17-Dec-2020].

15. “A Visualization Grammar,” Vega. [Online]. Available: https://vega.github.io/vega/. [Accessed: 26-Nov-2020].

16. A. Ngolo, A. Judd, and C. V. Woert, COVID19 API. [Online]. Available: https://covid19api.com/. [Accessed: 30-Nov-2020].

17. “World population API Documentation,” RapidAPI. [Online]. Available: https://rapidapi.com/aldair.sr99/api/world-population. [Accessed: 03-Dec-2020].

18. “Rate limits — Docs — Twitter Developer,” Twitter. [Online]. Available: https://developer.twitter.com/en/docs/twitter-api/rate-limits. [Accessed: 17-Dec-2020].

19. “What is Apache Storm? Comparison between Apache Storm vs Spark,” Intellipaat Blog, 08-Oct-2020. [Online]. Available: https://intellipaat.com/blog/what-is-apache-storm/. [Accessed: 17-Dec-2020].

20. M. Smallcombe, “Hive vs. HBase,” Xplenty, 27-Nov-2018. [Online]. Available: https://www.xplenty.com/blog/hive-vs-hbase/. [Accessed: 17-Dec-2020].

11 Appendix

11.1 Overview of Submission

This is an overview of the submitted files for this project. The .zip-file contains the source code of the data processing and the visualization software. The .zip-file is the downloaded GitHub repository as listed below. The GitHub repository contains four folders, which are Node0, Node1, Node2 and Website. The different scripts and configuration files mentioned in the report are within these folders, where the folder name corresponds to which node the files are located on (except Web site).

- BDDST_Report_Group8.pdf
- BDDST_Code_Group8.zip
- GitHub Repository: https://github.com/troelszink/BigDataProject

11.2 Scripts

```
import findspark
findspark.init('/etc/spark')
import pyspark
from pyspark import RDD
from pyspark import SparkContext
from pyspark.streaming import StreamingContext
from pyspark.streaming.kafka import KafkaUtils
from pyspark.sql import SparkSession
from pyspark.sql import HiveContext
import json
import os
import traceback
import sys
import subprocess

os.environ['PYSPARK_SUBMIT_ARGS'] = '--jars
↪ /home/hadoop/spark-streaming-kafka-0-8-assembly_2.11-2.4.7.jar --conf
↪ spark.sql.catalogImplementation=hive pyspark-shell'

sc = SparkContext(appName="PythonSparkStreamingKafka")
ssc = StreamingContext(sc,2)

sparkSession =
↪ SparkSession.builder.appName("SparkToHDFSWrite").getOrCreate()
sparkHive = SparkSession.builder.master("local").appName("SparkToHive")
.config("spark.sql.warehouse.dir",
↪ "/user/hive/warehouse").config("hive.metastore.uris",
↪ "thrift://10.123.252.223:9083").enableHiveSupport().getOrCreate()
sparkHive.sql("use twitterdb")

subprocess.Popen("hdfs dfs -rm -r -f /user/hadoop/twitterFiles/* |  awk
↪ '{print \$8}'", shell=True, stdout=subprocess.PIPE,
↪ stderr=subprocess.STDOUT)
```

```python
print(".csv-files deleted")
sparkHive.sql("DROP TABLE tweets")
sparkHive.sql("CREATE TABLE IF NOT EXISTS tweets (tweet_id STRING, `date` STRING, country_code STRING) ROW FORMAT DELIMITED FIELDS TERMINATED BY ',' LINES TERMINATED BY '\n' STORED AS TEXTFILE")
print("Database emptied")

dks = KafkaUtils.createDirectStream(ssc, ["test-events"], {"metadata.broker.list":"10.123.252.225:9092"})
data = dks.map(lambda tuple: tuple[1])

columns = ['tweet_id', 'date', 'country_code']
df = sparkSession.createDataFrame([("1","2","3")], columns)
currentDate = ""
count = 0
countName = 0
def sendRecord(rdd):
    try:
        global df, columns, currentDate, count, countName
        json1 = json.loads(rdd.take(rdd.count())[0])
        tweetId = str(json1["tweet_id"])
        date = str(json1["date"])
        countryCode = str(json1["country_code"])
        row = [(tweetId, date, countryCode)]

        if count == 0:
            df = sparkSession.createDataFrame(row, columns)
            count += 1
        elif count < 1000:
            newRow = sparkSession.createDataFrame(row, columns)
            df = df.union(newRow)
            count += 1
        else:
            print("Writing to HDFS")
            path = "hdfs://10.123.252.225:9000/user/hadoop/twitterFiles/" + str(countName)
            df.coalesce(1).write.csv(path)
            path2 = "/user/hadoop/twitterFiles/" + str(countName)
            fileList = subprocess.Popen("hdfs dfs -ls " + path2 + " |  awk '{print \$8}'", shell=True, stdout=subprocess.PIPE, stderr=subprocess.STDOUT)
            fileList2 = []
            for line in fileList.stdout.readlines():
                fileList2.append(line.decode("utf-8").rstrip())
            sparkHive.sql("LOAD DATA INPATH \'hdfs://bddst-group8-node0.uvm.sdu.dk:9000" + fileList2[-1] + "\' INTO TABLE tweets")
            df = sparkSession.createDataFrame(row, columns)
            count = 0;
            countName += 1

    except Exception as e:
        traceback.print_exception(*sys.exc_info())
        pass

data.foreachRDD(lambda rdd: sendRecord(rdd))
```

```

ssc.start()
ssc.awaitTermination()
```

Listing 12: *The entire script for the consumer:* ***kafka-spark-streaming.py***

```
import findspark
findspark.init('/etc/spark')
import pyspark
from pyspark.sql import SparkSession
from pyspark.sql import HiveContext
import os
import subprocess

os.environ['PYSPARK_SUBMIT_ARGS'] = '--jars
↪ /home/hadoop/spark-streaming-kafka-0-8-assembly_2.11-2.4.7.jar --conf
↪ spark.sql.catalogImplementation=hive pyspark-shell'

sparkSession =
↪ SparkSession.builder.appName("SparkToHDFSWrite").getOrCreate()
sparkHive = SparkSession.builder.master("local").appName("SparkToHive")
.config("spark.sql.warehouse.dir",
↪ "/user/hive/warehouse").config("hive.metastore.uris",
↪ "thrift://10.123.252.223:9083").enableHiveSupport().getOrCreate()
sparkHive.sql("use twitterdb")

subprocess.Popen("rm -r -f /home/hadoop/DatabaseFiles/* |  awk '{print
↪ \$8}'", shell=True, stdout=subprocess.PIPE, stderr=subprocess.STDOUT)
print(".csv-files deleted")

df = sparkHive.sql("SELECT * FROM tweets WHERE (country_code != 'NULL' OR
↪ country_code != '""') ORDER BY country_code, date")
df.show(50, False)

columns = ['date', 'n_tweets']
df_new = sparkSession.createDataFrame([("date","numOfTweets")], columns)
country_temp = ""
date_temp = ""
count_tweets = 0

for row in df.rdd.collect():
    country = str(row["country_code"])
    date = str(row["date"])
    if country_temp == "":
        country_temp = country
        count_tweets += 1
    elif country_temp == country:
        if date_temp == "":
            date_temp = date
            count_tweets += 1
        elif date_temp == date:
            count_tweets += 1
        else:
            count_tweets += 1
            row_new = [(date_temp, count_tweets)]
            df_row_new = sparkSession.createDataFrame(row_new, columns)
            df_new = df_new.union(df_row_new)
            date_temp = date
            count_tweets = 0
    else:
        count_tweets += 1
        row_new = [(date_temp, count_tweets)]
        df_row_new = sparkSession.createDataFrame(row_new, columns)
        df_new = df_new.union(df_row_new)
        date_temp = date
        count_tweets = 0

        df_new.coalesce(1).write.csv("/home/hadoop/DatabaseFiles/" +
        ↪   country_temp)
        df_new = sparkSession.createDataFrame([("date","numOfTweets")],
        ↪   columns)
        country_temp = country
```

Listing 13: *The entire script for loading from the database:* ***hive-database.py***

11.3 Figures

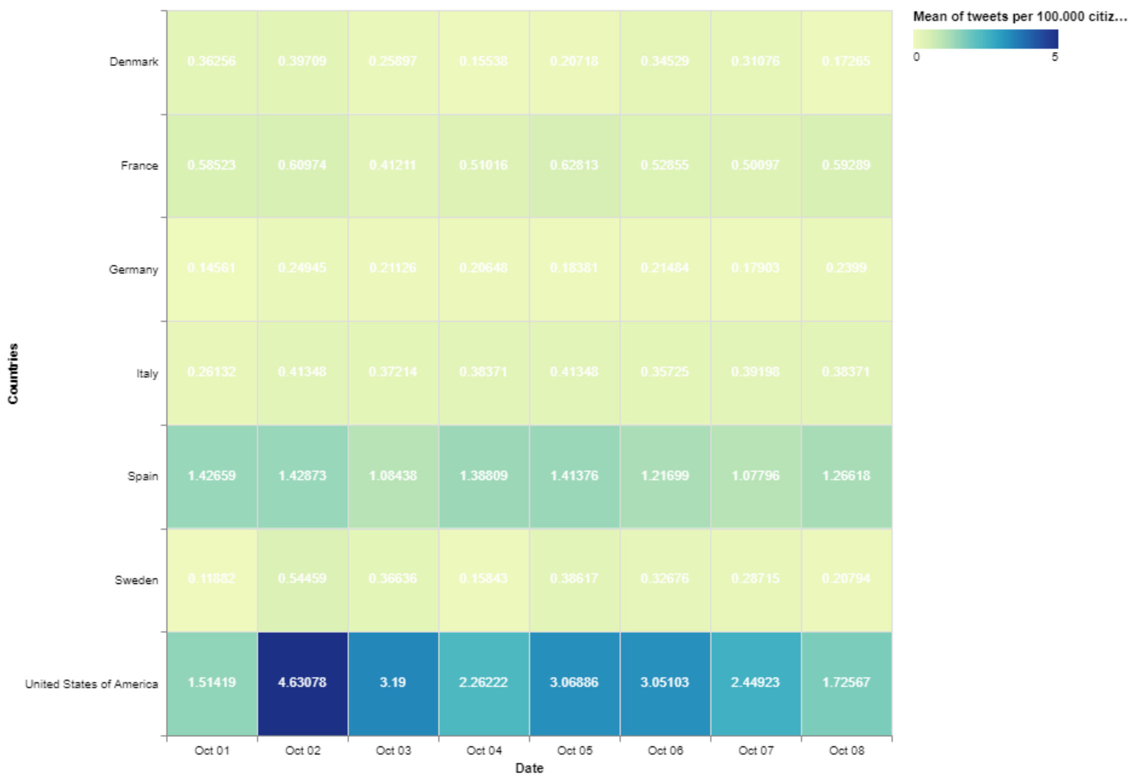


Figure 8: *Heat map of tweets (full scale)*

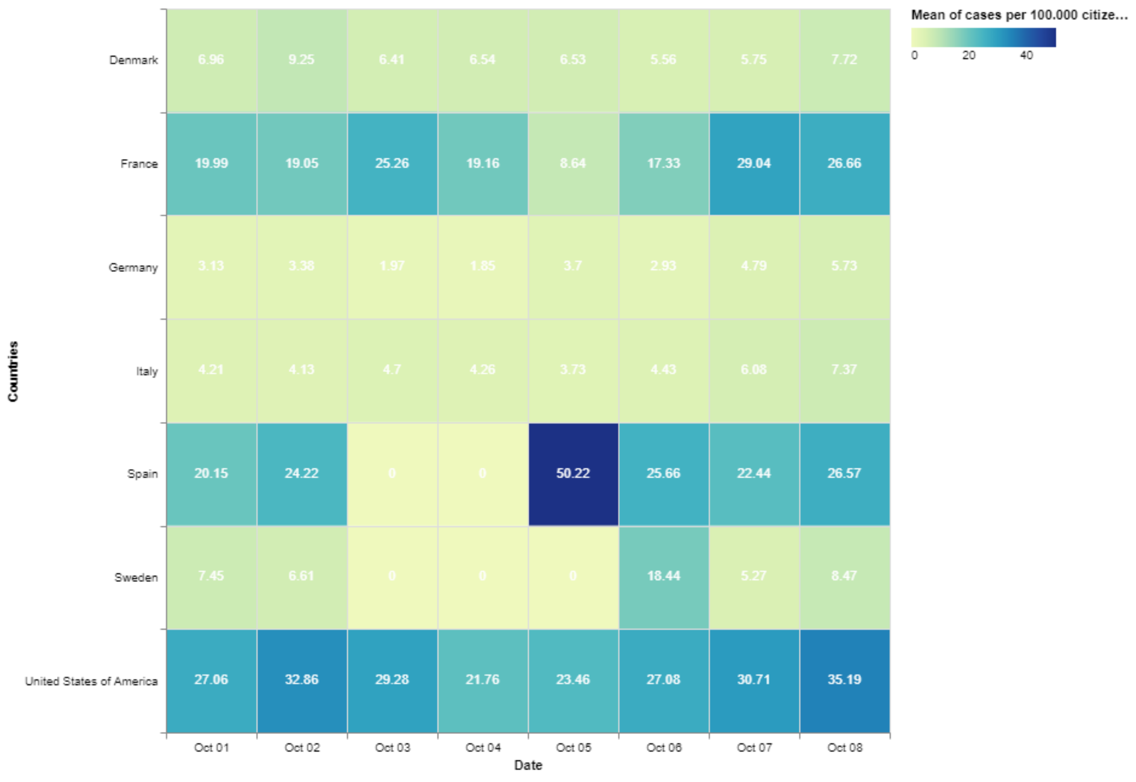


Figure 9: *Heat map of cases (full scale)*