university of groningen / faculty of science and engineering / bernoulli institute

Visualizing demography vulnerability to contagious viruses

Scalable Computing course project

March 2020

Students: J. G. S. Overschie, J. de Jong, P. Dekker

Contents

1 Introduction

We are in the midst of a global pandemic. At the time this project started, the Corona virus was still just a headline for most of us here, but in the meantime it reached and hit us completely. Fighting such a pandemic happens in many ways on multiple scales. We are interested in how this can be done on the societal level. Data that can be used to do so is mostly demographic in nature and obtained on a global scale. Therefore, we build a data processing pipeline that can process streams of data and can analyze both historical and streaming data. The pipeline will have a distributed architecture of components that inter operate but are not dependent of each other for their own life cycle. The components are deployed, monitored and maintained using Kubernetes and run on a Google Cloud cluster.

A brief description of our data processing pipeline would mention the three main components: a Kafka cluster for message brokering, a MongoDB cluster for data persistence and a Spark cluster for scalable data analytics. These components exist across multiple containers in a Kubernetes cluster running on Google Cloud virtual machines.

The rest of this document will explain the architecture of our pipeline, the technologies used and how they are used (Section 2), how we create and manage the pipeline and its infrastructure (Section 3), some results (Section 4) and a conclusion (Section 5).

2 Architecture

In designing our architecture, several decisions had to be made. We had to decide on systems to use for several different functionalities; a message queue, both a batch- and a streaming data processor and a database. For these things we chose Kafka [7], Spark [1], Faust [9] and MongoDB [3], respectively. Kafka and Spark are Apache products and MongoDB is a widely known NoSQL database, with good support for distributed processing. Faust, however, might be less widely known. Faust is a stream processor written in Python - it will be further explained in Section 2.4.2.

Since we run our entire application on *Google Kubernetes Engine* (GKE), we also had to come up with a systematic way to run and test our code. For this we chose to install Jupyter Notebook servers onto the cluster, which would then communicate with the different components in the application.

2.1 General architecture

The architecture can be split in two parts: the batch processing part and the stream processing part. Since both use different data sources, they are different enough to consider explaining them in different sections. Both use a combination of proprietary software components and our own written software. Our custom software mainly exist to pass data through the layers, but also to execute the main data processing algorithms performed on the data.

First, let us take a look at the **Batch processing pipeline**. See Figure 1. In the pipeline there exist some steps to (1) pre-process the data, and some to then (2) process the data. The pre-processing steps are downloading the dataset, reading it and converting it to GeoJSON [2], and finally storing it in the database. On the other hand, the processing step includes actually infering more meaning from the data stored in the database, by using Spark. Spark reads data from MongoDB, processes it, and writes back to MongoDB.

Preparing the data at a given sampling rate is a one-time operation. First, we use the `worldpop-downloader.ipynb` script to download the desired datasets to disk, coming from *Worldpop* [13]. The `.tif` files are stored on disk in the Jupyter Notebook, which has a Kubernetes `PersistentVolume` attached to it with 18Gi of storage capacity. Next, the tif data is read in small partitions at a time and sent to Kafka in GeoJSON format. This stream is consumed by another pod and fed into a MongoDB collection.

After having prepared the data, we are able to make analyses using Spark. Processing is started by running our algorithm in Apache Zeppelin, which utilises the running Spark master- and worker instances to distribute workload. The Zeppelin script then reports results back to the database in a separate collection. Finally, we run a separate server to visualize the GeoJSON collection results.

The visualization server watches the results collection for changes and updates the chart accordingly, by supplying it with a new `.json` file written to disk.

Secondly, we take a look at the **Stream processing pipeline**. See Figure 2. For our stream processing pipeline, we use twitter status updates containing the word 'corona' as an input. Since this is currently a hot topic, a gigantic amount of tweets are coming in every minute. For our use case, however, only a limited amount of messages are usable; we filter to retain only geo-tagged tweets. Those tweets are sent to Kafka, which is then consumed by Faust. In Faust, this stream is consumed and aggregated in some meaningful way. Finally, another pod then consumes the output Faust generates and reports it back to the results collection. The visualization pod then takes care of the rest, visualizing whatever is in the results collection.

Finally, we can depict the entire application architecture in one picture, containing both batch- and stream processing. See Figure 3. In this illustration it can be observed that in our pipeline the batch and stream processing pipelines are completely separate until they are merged into the *results* collection in the MongoDB database. Even though in this application architecture both data consumption types do not share one single datasource, the way our architecture merges in a later stage much resembles how this happens in a λ-architecture (lambda architecture). Both methods write to one collection, and eventually only one component is used to visualize both, which is the *visualization* component.

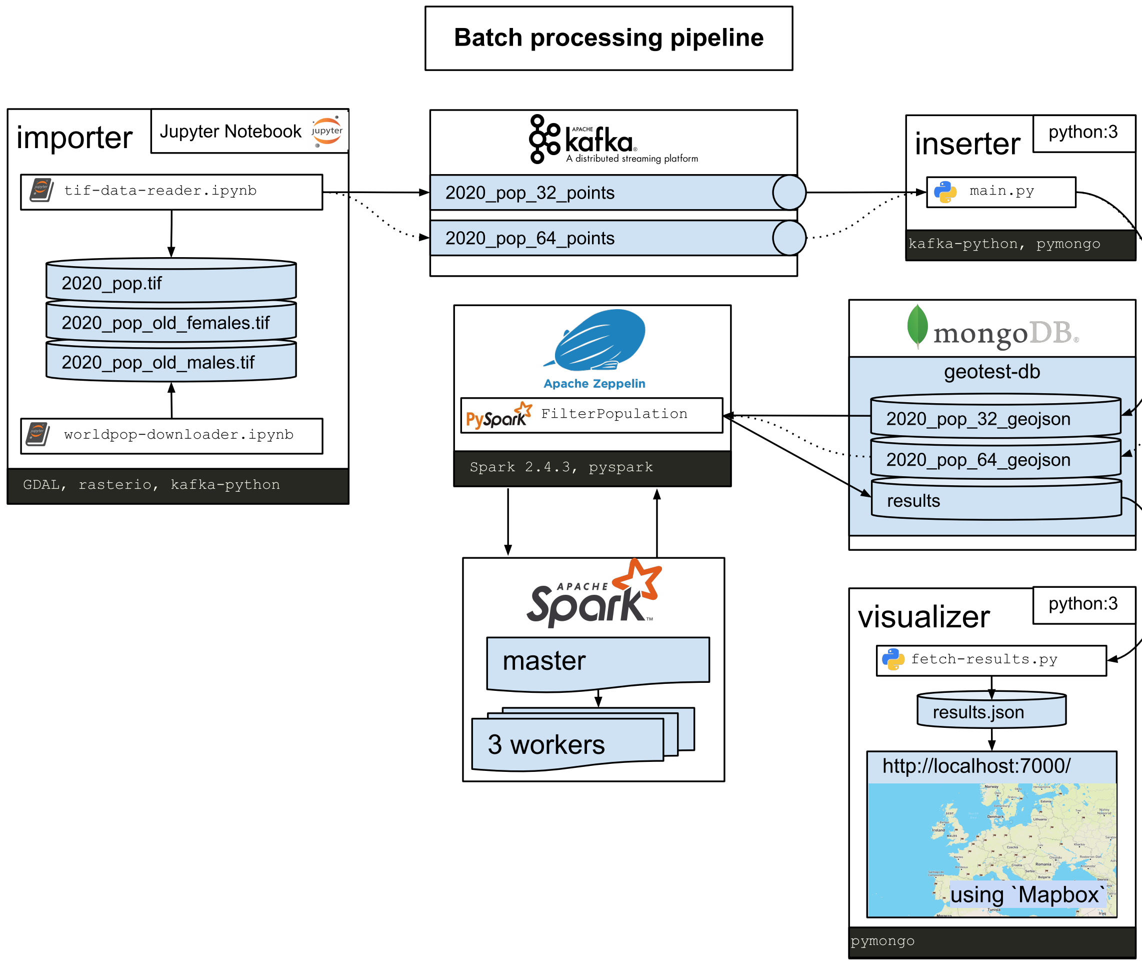


Figure 1: Architecture overview of **Batch** processing pipeline. Arrows between the components indicate (directional) data flow.

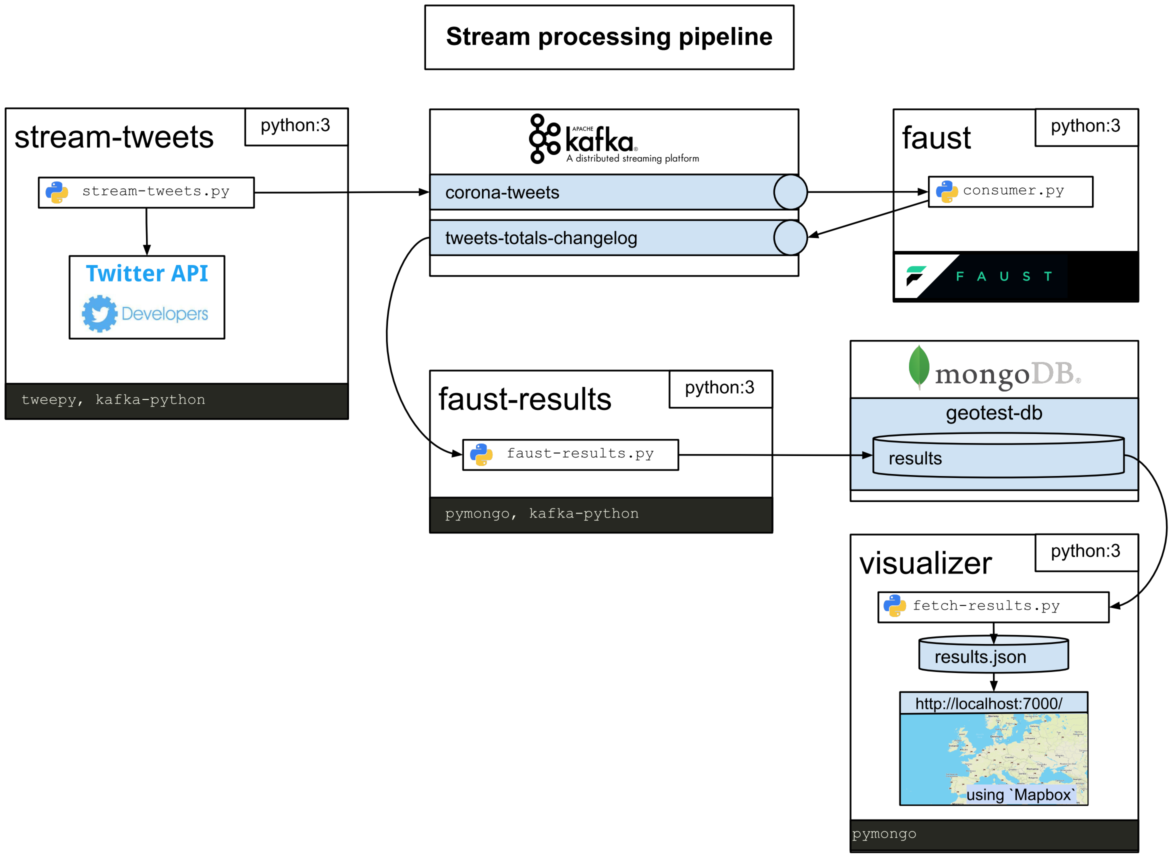


Figure 2: Architecture overview of **Streaming** processing pipeline. Arrows between the components indicate (directional) data flow.

2.2 Proprietary components

Now that we have a grasp of the general idea of our pipeline, let us learn about how we installed the proprietary components. To install the different third-party components on GKE, we used *Helm*. Helm is a package manager for Kubernetes, which allows one to install various software packages using Helm '*charts*'. A chart contains Kubernetes `.yaml` configuration files, setting up a template for any deployments or services that are to be made to configure the application. A suitable Docker image along with the image version is already defined in the chart, but could be manually modified as well. In installing all proprietary components on our GKE cluster, the following notes are to be made:

1. **Kafka**. Using the *Bitnami* chart, we installed Kafka version 5.0.1. In order to utilise the benefit of using multiple replicas, we set various parameters as such:

   ```
   --set replicaCount=3,defaultReplicationFactor=3,offsetsTopicReplicationFactor=3,
   transactionStateLogReplicationFactor=3,transactionStateLogMinIsr=3
   ```

   This installed Kafka and Zookeeper pods using a replication factor of 3.

2. **Kafdrop**. To visualize Kafka's internal memory, we decided to install a third-party tool to do this. Kafdrop provides a user-interface for Kafka. We installed it using a Helm chart on Github [8]. The interface allows us to see what topics were created and what messages exist in each. See Figure 7.

3. **MongoDB**: We use MongoDB for persistent storage, installed in our cluster using the *Bitnami* chart at version 7.8.10 which installs MongoDB version 4.2.4. MongoDB allows us to easily set

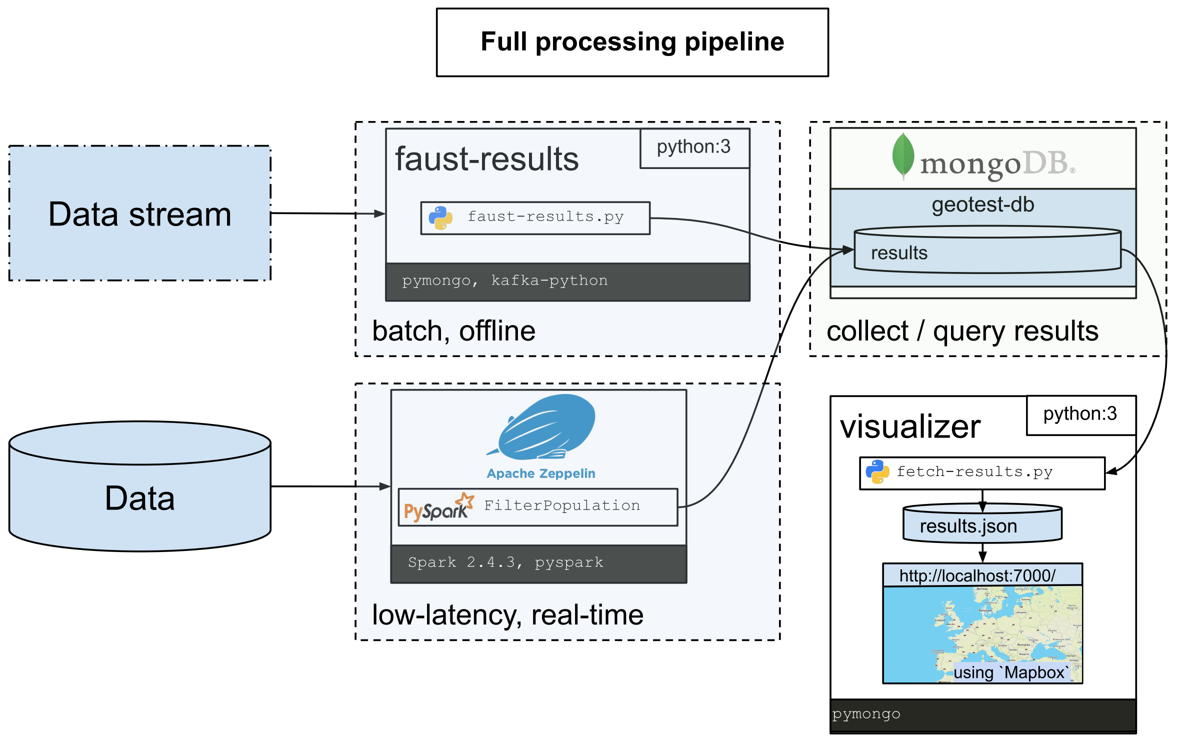


Figure 3: Architectural overview of the entire application, with both batch- and stream processing. Arrows between the components indicate (directional) data flow.

up a very scalable database cluster. It also allows for the creation of geospatial indexes on the data it stores which is essential to our application.

4. **Mongo-Express**: To aid development, we also installed Mongo-express in our cluster. This gives us a graphical interface to quickly peek into MongoDB collections, check for existing indexes etc. The chart creates an arbiter node, a primary node and a secondary node. Next to this, the chart creates a Kubernetes secret store to store the root password. This is a small volume that contains a key-value store. Using this secret store, we can configure other components of the system to have this password available through e.g. an environment variable, so we access to our MongoDB cluster can be controlled. To see an example of our mongo-express interface, see Figure 8.

5. **Jupyter Notebook**: We used Jupyter Notebook servers in our cluster during development. This allowed us to develop python scripts that interacted with components in the cluster without us having to continually rebuild and re-upload images or exposing all our components openly to the Internet.

6. **Spark**: We use the *Bitnami* Spark chart at version 1.0.0. to install Spark at version 2.4.3 in our cluster.

   (a) Zeppelin: The chart comes with Zeppelin, a service very similar to Jupyter notebook, but specifically for Spark.

   (b) Livy: The chart also provided an option to install Apache Livy, which is a service that provides a REST-interface to submit jobs to Spark. We, however, manually disabled the Livy pods because we decided to use Zeppelin directly.

   (c) Master/Worker: Chart installs a specified amount of master and worker nodes. Our setup runs 1 master node and 3 workers. Workers have 2Gi of memory.

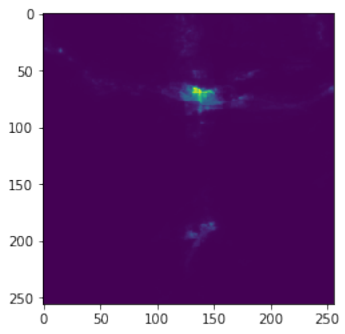


Figure 4: An example of a plotted window from our dataset. This specific window is of 256 × 256 dimensions, which is the standard size of windows in the dataset, following the way the dataset is internally segmented in `.tif` specification.

We can show the entire listing of our Helm installed components by running `helm ls`:

Listing 1: Display of all installed Helm components

```
jeroenoverschie@cloudshell:~ (sixth-utility-268609)$ helm ls
NAME                    REVISION    UPDATED                         STATUS
CHART                   APP VERSION NAMESPACE
kafdrop                 1           Tue Mar  3 09:37:46 2020        DEPLOYED
kafdrop-0.1.0           3.x         default
my-kafka                1           Mon Mar 23 22:29:34 2020        DEPLOYED
kafka-0.20.8            5.0.1       default
my-mongo-express        1           Tue Mar 24 12:55:02 2020        DEPLOYED
mongo-express-1.1.0     0.54.0      default
my-mongodb              1           Tue Mar 24 12:31:43 2020        DEPLOYED
mongodb-7.8.10          4.2.4       default
my-spark                4           Wed Mar 25 14:06:07 2020        DEPLOYED
spark-1.0.0             2.4.0       default
```

2.3 Batch processing

In our batch processing pipeline, several steps are taken. In our code, a dataset is downloaded from the internet and can travel all the way through the pipeline. Several components of the batch processing pipeline can be outlined as follows:

- Importer: download dataset from internet, read and sample data, send to Kafka.
- Inserter: consume Kafka topic and insert data into a database collection accordingly.
- Spark: data analytics. Perform on-demand data analysis in Zeppelin and store results back to MongoDB.

Each component is outlined below.

2.3.1 Importer

Our main data source is population data from Worldpop in GeoTIFF files. Worldpop makes estimations on the population for any given square kilometer, by analyzing satellite images. We read the files, divide them into cells and send these cells as messages to Kafka in GeoJSON format. Our dataset contains over 800 million datapoints, which to our estimates would require hundreds of gigabytes of storage for both Kafka and MongoDB. Since storage is not free on Google Cloud, we decided to sample the data, taking a number of cells together into a window and broadcasting the total population count for these cells with their center coordinate. Figure 4 shows an example window, with the example output shown below:

```
center point: (62.13208309980001, 33.86625006591001)
total population: 2188118.5
total old female population: 2228.070556640625
total old male population: 2870.031982421875
```

2.3.2 Inserter

This component reads data from a Kafka topic and inserts each message in MongoDB, for persistence. Each message sent to Kafka on this particular top contains a GeoJSON document as value, which is stored directly in MongoDB. This collection is indexed on the coordinates that are contained in the GeoJSON documents for quicker data retrieval. This index is used immediately, since we do an upsert rather than insert, meaning we update the population data for a certain coordinate if that coordinate or one very close to it is already present in the collection.

This inserter is implemented as a python script with using libraries to communicate with Kafka and MongoDB. This script is contained in a Docker image which is used to create a continuously running process in our pipeline, always taking any new messages in Kafka that are sent to the appropriate topic and storing or updating them in MongoDB.

2.3.3 Spark using Zeppelin

Our batch processing algorithm uses the MongoDB connector for Spark. This extension to spark allows us to create a Spark job that can fetch its data directly from spark, preventing a data aggregation bottleneck from occurring and minimizing the amount of code needed to program the job.

The goal of the algorithm we run on spark is to find clusters of data points where the population density is high. Those clusters are areas where covid-19 spreads easy and since the elderly are a vulnerable group, the data of elderly men and women are compared to the general population data. Once the clusters are found, the cluster central points are stored in a separate MongoDB collection, again as GeoJSON.

Now we discuss the algorithm in a bit more detail. After the data is collected, the data is preprocessed. The data is imported as GeoJSON, meaning that we are given a value for a longitude and latitude. In Spark we can work with three main data objects, namely a Resilient Distributed Dataset (RDD), a DataFrame, or a DataSet. Though, the DataSet API is not available for python and it seems that working with DataFrames provides better performance, so we will mostly work with DataFrames [11, 5, 4]. After a quick inspection of the data it is evident that the data seems to be logarithmically scaled. Therefore, we take the logarithm with base 10 of each element and then we normalize the data to a $0 - 1$ scale.

To find clusters in data there are many algorithms available. Since we expect densely populated areas such, as cities, to form a cluster, we cannot properly estimate the number of clusters. This characteristic eliminates a lot algorithms already. One widely-used algorithm for finding clusters in data where the number of clusters is given after the algorithm is done, is the Mean Shift algorithm. The Mean Shift algorithm is as follows:

1. Initialize N cluster points and a corresponding window W, which contains all the data points within a given radius. We let the initial cluster points be evenly distributed on a grid
2. Compute the mean of W and shift the cluster points to the mean of W
3. Repeat step 2 until convergence. Note that we can also set a number of iteration, after which we think the algorithm has converged.

In our case, the result is a list of cluster points, where each cluster point denotes a longitude and latitude around which the population density is relatively large.

It is too extensive to discuss every bit of code, but a few things about the code are worth mentioning:

- Spark-native functions are always better than User-Defined Function's (udf), which is why we try to avoid using udf's and try to use Spark-native function as much as possible [6].
- It is important to know how the data is partitioned, which we can modify if necessary by setting the number of partitions. Spark recommends the number of partitions to be $2-3$ times the number of cores in the cluster [12].
- It is important to think about improving efficiency of the algorithm before optimizing spark performance.

The code is written with the above remarks taken into account. Finally, the results are stored as a GeoJSON in MongoDB. The entire algorithm execution happens within Zeppelin, which provides a nice user interface to see exactly what parts of the entire spark job are either; cancelled, failed, or were successfully executed. See Figure 5.

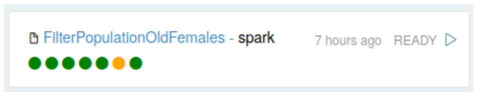


Figure 5: Our algorithm written in a Notebook, having ran some parts of a Spark job. User interface identifies exactly what parts of your job were executed and which failed.

2.4 Stream processing

Our stream-processing pipeline consists of several light-weight python modules packed into separate pods in GKE. The main communication channel used is Kafka, with the final results being stored to same MongoDB results collection like in the Batch processing pipeline.

Our main goal in the stream processing pipeline is to visualize which countries currently tweet about corona a lot, to possibly be able to make interesting conclusions based on this data. This poses us with a good opportunity to process real-time data and apply a 'group-by' aggregation operation; grouping tweets by the respective country in which the tweet was sent.

2.4.1 Tweet streamer

First, we setup a connection with the Twitter API to listen for tweets. To connect to the Twitter API, we are using a python library called *tweepy* [10]. Using this API, we can subscribe to something specific to 'track', for example twitter status updates containing some specific word. This is indeed what we do: we set our listener to receive twitter status updates containing the word 'corona'. Quickly, a massive amount of tweets come in. In our use case, however, we can filter out any tweets that do not contain a geo-tag.

Next, we process the tweet data to be compatible for formatting in the next step. We strip off any values we do not need, and re-compute a central coordinate from some several coordinates representing a polygon as was supplied by the Twitter API. Finally, we send the tweet into the Kafka topic '`corona-tweets`'.

2.4.2 Faust

Next, to perform the actual stream processing, we use Faust [9]. Faust is a stream processing library written completely in Python. It allows you to read streams, process them in distributed manner, and output the results into some store. The library is similar in idea to *Spark Streams* and *Kafka Streams*. The main difference is its implementation in Python instead of Java or Scala.

Faust can be connected to a Kafka topic. Faust will then internally transform the stream into smaller batches, to be further processed. Once Faust is done processing, it can output its aggregation results back into Kafka. In this way Kafka acts as a storage provider, with Faust in between.

In this case, we consume the Kafka topic '`corona-tweets`'. Upon every new write, we instantiate a new processing step. In Faust, one can use some data structure called 'tables', which is some distributed way of storing its aggregations. Because we are only interested in what country sends how many tweets in total, we can perform a group-by operation here. Faust is smart, and can distribute its workload automatically. Faust actually creates 2 topics to write its results; one to continuously re-partition the data, and one to actually store the results in a huge changelog. Faust will find what keys are being used to perform the group-by operations, and will partition the Kafka topic accordingly. In this way, the workload of the entire operation is load-balanced. In our case, for example, 8 partitions are made with different country labels - each partition handling aggregations for some specific set of countries. In a scenario where the group-by operation would be applied to a larger set of keys, possibly more paritions would have been created.

2.4.3 Faust result passthrough

Finally, because Faust stores its data back into Kafka, we require a small component in between Kafka and MongoDB to pass through the results. This component will setup a KafkaConsumer which will listen for any changes Faust will produce. It will then read the data and transform it to fit the GeoJSON format, after which it is written to the 'results' MongoDB collection.

2.5 Visualization

Finally, our last component is the *Visualization* component. In this component two services are ran: (1) a service watching the 'results' collection in mongo and (2) a webserver serving a static html file. The execution of the MongoDB collection watcher happens in the background using the `nohup` command, whilst the webserver will be bound to a local port such that it can be made visible externally.

To use the MongoDB collection watching functionality, we had to make sure our Mongo database was properly replicated. Since this is a performance-heavy operation, Mongo does not allow one to watch a collection when only one instance of the mongo database is running. Because we are using *Helm*, we were easily able to configure our mongo database to use some specified amount of replication. In our implementation, our replication is configured such, that we obtained 'arbiter', 'primary' and 'secondary' deployments to run MongoDB. We were then able to watch our Mongodb 'results' collection for any new changes coming in. On every new change, the script writes the entire collection - containing GeoJSON 'Feature' objects - to a `results.json` local file.

This local file is then used by the web service. The web service loads in packages from the MapBox library. MapBox is a toolkit to create custom made maps in a web environment. As an input argument to the data source used in the plot, we can specify a (local) `.json` file. Because in the previous step, on every change to the results database collection, the collection was written to disk, now all we have to do is specify the path to the json file in the frontend code, to then have MapBox do the plotting work. On the client-side, in our JavaScript code, we continuously read in the .json file, such that, when new changes do occur, they are immediately reflected on the screen. Built-in to the MapBox tool is functionality to render new changes to the screen quickly. For the output resulting from the visualization, see the Results section (Section 4).

3 Infrastructure

We use Kubernetes to deploy, monitor and maintain our entire system. Kubernetes allows us to define resources of many kinds, some for performing computational tasks, some for resources, some for networking, and manage these using an API. This automation is wonderful, but the true power lies in Kubernetes' health monitoring and scaling capabilities. With Kubernetes we could have a high availability cluster of, for example, MongoDB instances running and with a few keystrokes add any number of extra MongoDB instances to that cluster.

Kubernetes requires just one resource to run: nodes. Nodes are (virtual) machines that run a Kubernetes process which maintains contact with the Kubernetes master. The virtual machines we use are provided by Google Cloud. We use a Google Cloud cluster consisting of six moderate machines. The machines have just enough resources to deploy our pipeline for demonstrative purposes, but are too limited in resources for very large scale deployment. However, due to the nature of the deployment of our pipeline, it is fairly easy to achieve such large scale deployment after procuring machines with sufficient resources, and then using Kubernetes to make the necessary changes to the deployment to make use of these resources.

In our implementation, we configured our cluster to have 6 total nodes using *n1-standard-1* machine types, allocating a CPU capacity of about 1Ghz per node. Total RAM memory capacity for all nodes is 22.50 GB. Persistent volumes vary in size. The entire cluster is hosted in the *europe-north* region.

4 Results

In Figure 6 we can see the results of our pipeline visualized on a map. The pipeline marks Brussels, London and Paris as highly vulnerable areas in case of a pandemic.

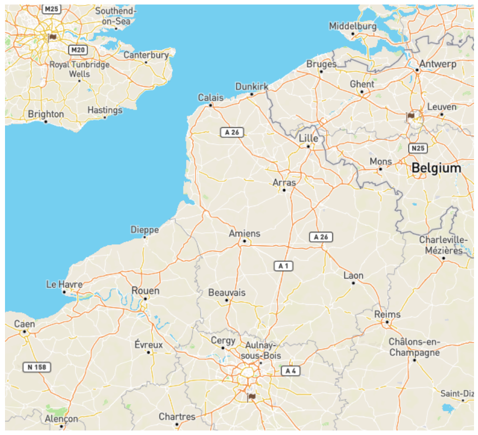


Figure 6: An excerpt from our visualized results, generated with the MapBox Javascript library. Visible on the chart are various hotspot points, indicated by brown flags.

Although the visualization is best interactively explored, some observations can be made about its functioning during the processing process. When the data is processed in Zeppelin, data passes through several windows of the entire dataset in some (random) order. Since in the processing step, results are written to database immediately when they are obtained without waiting for other pieces to finish their work, the map is automatically updated with the latest processed information. In this way, our experiment to plot real, live data resulted in an actually-deployed service live on the Google Cloud Platform.

5 Conclusion

With our pipeline we are able to find areas that are extra vulnerable during a pandemic. The results are useful and nicely presented, and can easily be updated by presenting new data.

We used modern, distributed, scalable software, deployed on a Google Cloud cluster using Kubernetes to create data processing pipeline that processes both historical and streaming data. Through the process, we gained experience in working with separate services in a distributed setting, all with dynamically provisioned resources from a cloud provider. This experience is very applicable in many modern day data analytics usecases.

References

[1] M. Armbrust, R. S. Xin, C. Lian, Y. Huai, D. Liu, J. K. Bradley, X. Meng, T. Kaftan, M. J. Franklin, A. Ghodsi, et al. Spark sql: Relational data processing in spark. In *Proceedings of the 2015 ACM SIGMOD international conference on management of data*, pages 1383–1394, 2015.

[2] H. Butler, M. Daly, A. Doyle, S. Gillies, S. Hagen, T. Schaub, et al. The geojson format. *Internet Engineering Task Force (IETF)*, 2016.

[3] K. Chodorow. *MongoDB: the definitive guide: powerful and scalable data storage.* " O'Reilly Media, Inc.", 2013.

[4] J. Damji. A tale of three apache spark apis: Rdds vs dataframes and datasets. when to use them and why, 2016.

[5] Dataflair. Apache spark rdd vs dataframe vs dataset, 2019.

[6] fqaiser. Udfs vs map vs custom spark-native functions, 2018.

[7] N. Garg. *Apache Kafka*. Packt Publishing Ltd, 2013.

[8] E. Koutanov. Kafdrop - kafka web ui, 2016.

[9] Robinhood. Faust stream processing, 2016.

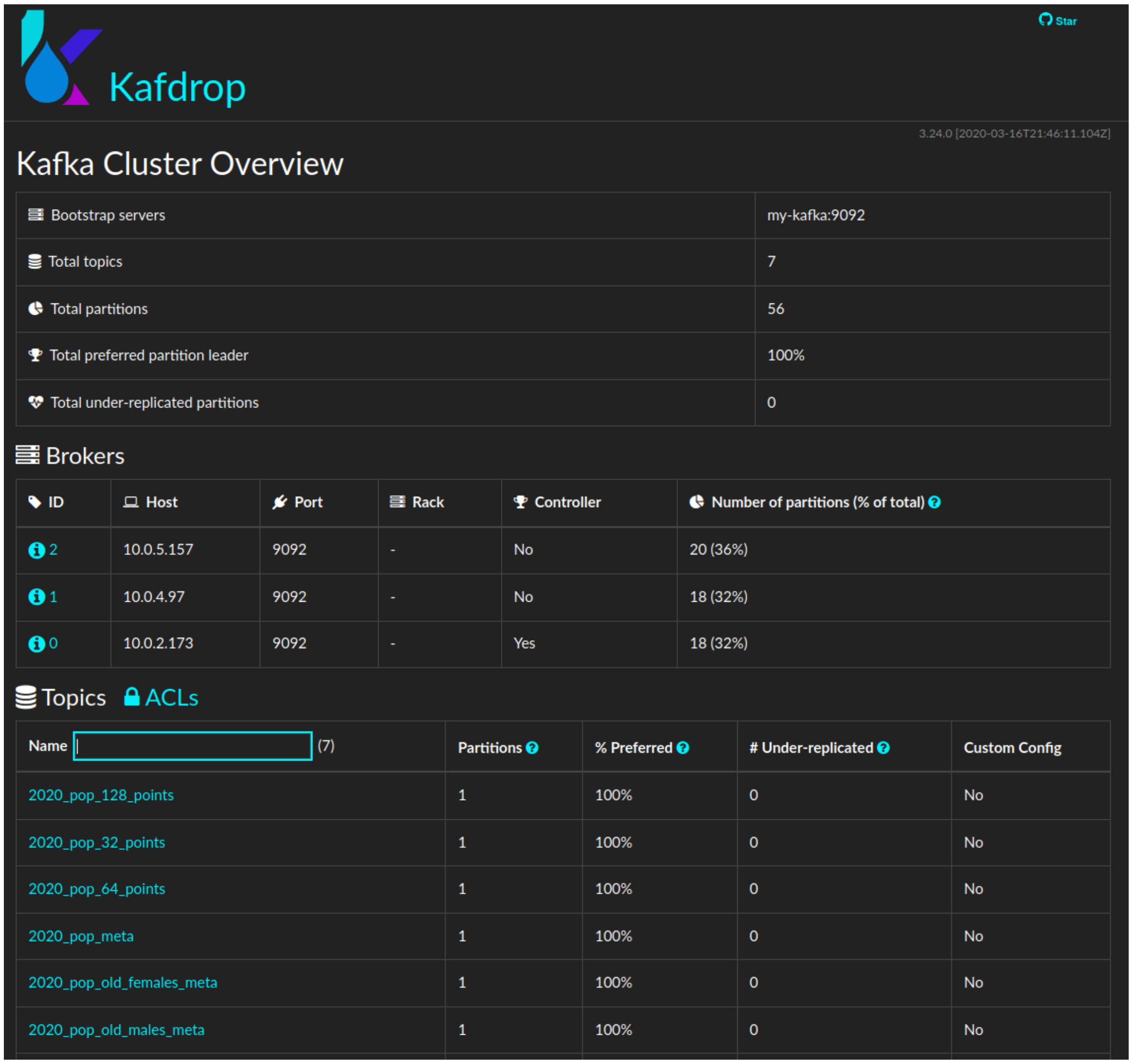
[10] J. Roesslein. tweepy documentation. *Online] http://tweepy. readthedocs. io/en/v3*, 5, 2009.

[11] A. Spark. Spark documentation, 2020.

[12] A. Spark. Tuning spark, 2020.

[13] A. J. Tatem. Worldpop, open data for spatial demography. *Scientific data*, 4(1):1–4, 2017.

A Appendix



Star

Kafdrop

3.24.0 [2020-03-16T21:46:11.104Z]

Kafka Cluster Overview

Bootstrap servers	my-kafka:9092
Total topics	7
Total partitions	56
Total preferred partition leader	100%
Total under-replicated partitions	0

Brokers

ID	Host	Port	Rack	Controller	Number of partitions (% of total)
2	10.0.5.157	9092	-	No	20 (36%)
1	10.0.4.97	9092	-	No	18 (32%)
0	10.0.2.173	9092	-	Yes	18 (32%)

Topics ACLs

Name (7)	Partitions	% Preferred	# Under-replicated	Custom Config
2020_pop_128_points	1	100%	0	No
2020_pop_32_points	1	100%	0	No
2020_pop_64_points	1	100%	0	No
2020_pop_meta	1	100%	0	No
2020_pop_old_females_meta	1	100%	0	No
2020_pop_old_males_meta	1	100%	0	No

Figure 7: Kafdrop user interface.

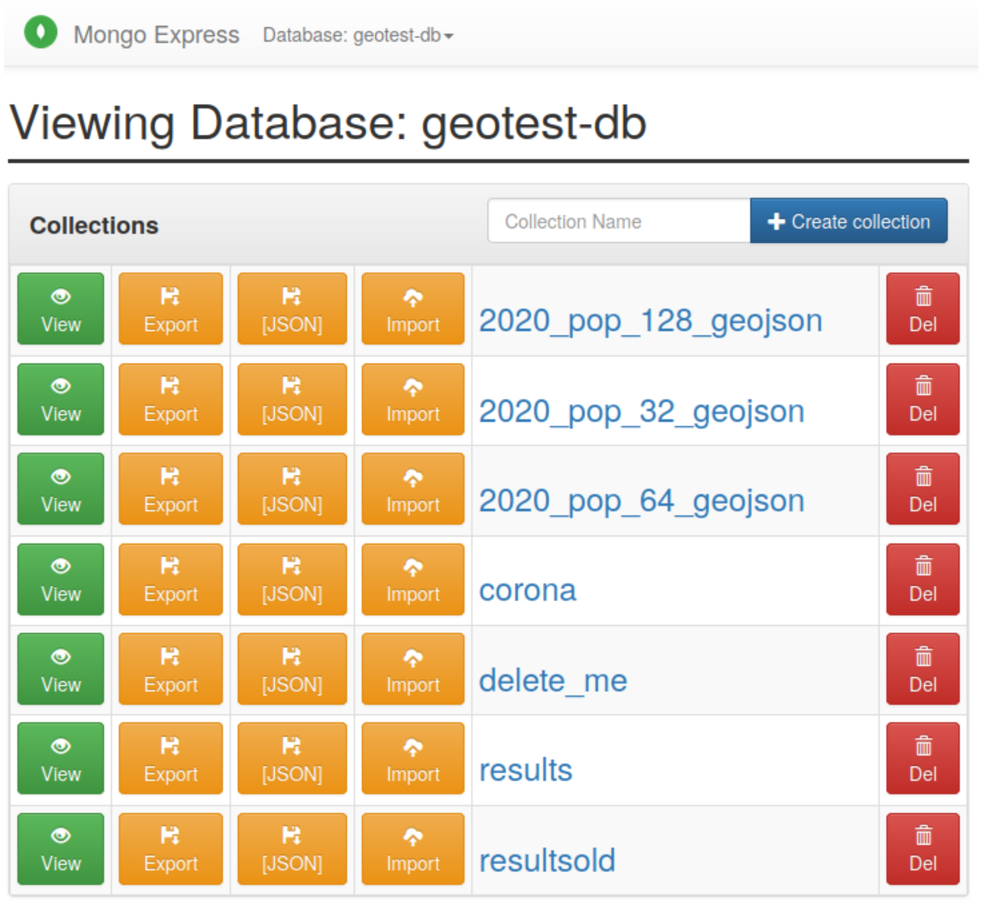


Figure 8: Mongo-express user interface.