Systèmes Distribués pour le Traitement de Données

Projet traitement de données

Maxime Chaloupe, Gabriele Degola,
Concetto Antonino Privitera, Quentin Stentzel

Janvier 2021

Summary

1 Introduction **2**

2 Software stack **2**
2.1 Overview . . . 2
2.2 Details . . . 3
2.2.1 Spark . . . 3
2.2.2 MongoDB . . . 4
2.2.3 Kafka . . . 4
2.2.4 NodeJS . . . 4
2.2.5 NGINX . . . 5
2.2.6 Prometheus . . . 5
2.2.7 Grafana . . . 5
2.3 Web Interface . . . 5

3 Architecture **7**

4 Development process **7**

5 Application life-cycle management **7**
5.1 Amazon Web Services . . . 7
5.2 Terraform . . . 7
5.3 Kubernetes . . . 7
5.3.1 MongoDB . . . 8
5.3.2 Spark . . . 8

6 Continuous integration with GitLab **8**
6.1 Container registry . . . 8
6.2 Continuous integration pipelines . . . 8

7 Non-functional properties **9**
7.1 Fault-Tolerance & availability . . . 9
7.1.1 Destroying pods . . . 9
7.1.2 Nodes draining . . . 10
7.2 Performances . . . 12
7.3 Costs . . . 15

8 Conclusion **16**

References **17**

1 Introduction

Our application is a web application that processes text reviews and displays the results on a web interface. In particular, our application performs:

- word count of reviews, counting the total number of occurrences for each word across all reviews;
- sentiment analysis, with prediction of the review's nature, either "positive" or "negative".

Our objective is to build a distributed system, which is able to manage and analyse a huge amount of data. To achieve that, we worked on Amazon Web Services with our student accounts deploying the components of our system on a Kubernetes cluster.

The complete project, together with instructions on how to deploy it, is available at https://github.com/gabridego/book-reviews-processing.

2 Software stack

2.1 Overview

- **Data processing:** Spark (Scala)
- **Database:** MongoDB with replication
- **Messaging:** Kafka
- **Web server:** NGINX (+ React)
- **API Server:** NodeJS (+ Express)
- **API Gateway:** NGINX
- **Monitoring:** Prometheus
- **Visualization:** Grafana

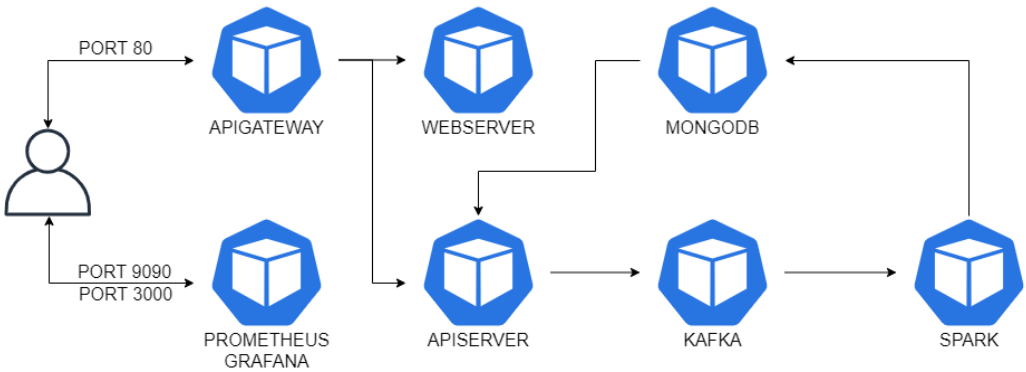


Figure 1: Architecture of the application, with user access

2.2 Details

2.2.1 Spark

For our application, we used Spark version 3.0.1, because it is a well known data processing framework that is widely used in the industry. Although it supports several programming languages, we chose to use Scala (version 2.12) to write our Spark code to be sure there would be the right connectors available. To build the application, we used *sbt* with the *sbt-assembly*[1] plugin when fat JARs are needed.

As we need to analyse streaming data, we exploited Spark Structured Streaming, a stream processing engine built on the Spark SQL engine with exactly-once guarantees. Structured Streaming queries are processed in almost-real-time using a micro-batch processing engine, which processes data streams as a series of small batch jobs. Different queries can be triggered on the same micro-batch, which have different duration depending on the incoming data[2].

To process data, our Spark job subscribes to a Kafka topic, through the official Spark interface, to get a data input on which to performs its' tasks. The data is received in JSON format and contains two fields, the text of the review and a numerical value representing the overall score. The results are stored in a MongoDB database, to which Spark connects through the MongoDB Spark Connector[3] and writes data via a Foreach sink.

As said before, the two tasks performed are word count and a sentiment analysis, computed by means of three separate queries.

Word count To compute the number of occurrences of each word, the reviews' text is split, punctuation characters are removed, every character is converted to lower case and the words contained in a list of common stop words are removed, as they don't bring useful information. Finally, the number of occurrences is counted for each remaining word. An entry is then added to the database, possibly replacing the previous existing one for that word.

Sentiment analysis To perform a sentiment analysis task, we locally trained a random forest model on an Amazon book reviews dataset[4], using Spark MLlib, and applied it online to streaming data.

In order to train the model, 10000 reviews are randomly extracted from the original dataset (which contains almost nine million documents) and the overall numeric score is mapped to a string, considering each review as positive if the score is above 3, negative otherwise. The review text is then tokenized, stop words are removed and weights are assigned to each word according to a TF-IDF schema[5]. These operations are performed using MLlib functions. The extracted features are then used to train a random forest with cross validation on five folds, in order to select the most suitable value for the number of trees (between 10 and 50) and for the maximum tree depth (between 1, 5 and 10). Finally, the resulting model is exported and stored in memory.

When the streaming application is executed, the model is loaded from memory, streaming reviews are processed accordingly and a label is predicted, either positive or negative. Two different queries are then executed. The first simply stores an entry for each review, containing the text, the ground truth and the predicted label, in a MongoDB collection. The second writes an entry at each micro-batch in a different collection, containing the total number of predictions and the model classification accuracy on the corresponding reviews since the beginning of the execution ($\frac{number\ of\ correct\ predictions}{total\ number\ of\ predictions}$).

Note that the *wordcount* and *accuracy* queries are possible thanks to the Structured Streaming model, which allows for simple stateful streaming processing.

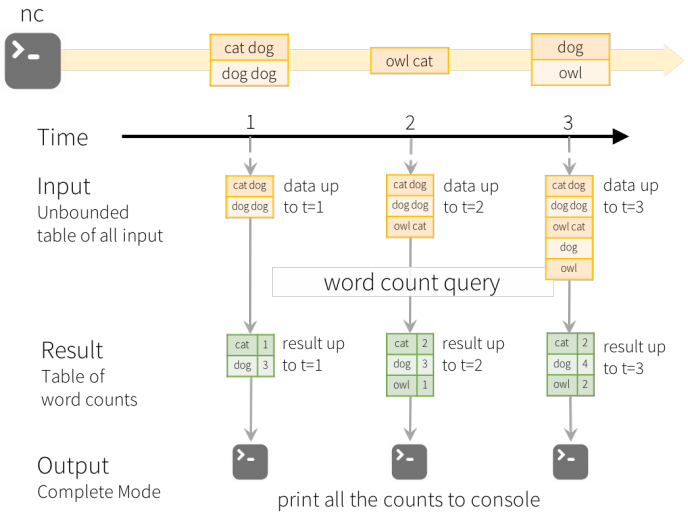


Figure 2: An example of stateful Structured Streaming processing[2]

2.2.2 MongoDB

MongoDB is a well established database based on documents. It is meant to be scalable which is a very important property of our application. Moreover, our team had some previous experience with it, and it is also a part of the Databases systems course. As said, the Spark connector for MongoDB is well supported, although it is not so straightforward with Structured Streaming. Given our application purpose to process reviews, a document based database is a great fit.

2.2.3 Kafka

Kafka is a distributed event streaming platform. It works with a producer-consumer scheme, with topics to write the events on. A producer writes on a topic and a consumer subscribes to that topic to receive the events from the producer. We chose to use Kafka for three main reasons:

- it is widely used in big data system, scalable, fault-tolerant, etc.;
- Structured Streaming connectors with Spark, which eases the integration;
- a bit of prior experience.

2.2.4 NodeJS

NodeJS is an environment capable of running JS code in form of an HTTP server. Thanks to the express framework, we have been able to build an application designed for scalable solutions. In fact, in this way we have all features of nodeJS combined with the features available over express layer. Our goal has been to build an extremely fast API server. In order to do that, we needed to:

- set the NODE_ENV environment variable in production in order to enable caches and reduce verbose messages;
- use mainly asynchronous instructions by exploiting JS promises;
- ensure automatically restart in case of errors.

2.2.5 NGINX

NGINX allows us to build an api-gateway which is perfect for load balancing. Therefore, we are able to redirect requests to the correct services based on the URL while keeping a high response time. Moreover, the web application, based on REACT framework, has been moved from nodeJS to NGINX for production.

2.2.6 Prometheus

Prometheus is a time series database that stores the metrics of a deployed system. Our choice of Prometheus is a direct consequence of having our application deployed to a Kubernetes cluster. Indeed, Prometheus is very popular choice for monitoring Kubernetes clusters, as it stores the metrics exposed by the Kubernetes API. Moreover, it integrates well with the visualization tool Grafana, and can be extended with an AlertManager that can be used to send notifications to the system administrators (emails, slack etc.) if certain conditions are met, like a pod failure or high CPU usage.

2.2.7 Grafana

Grafana is a visualization dashboard, it displays Prometheus' metrics using charts and graphics. It allows to check the Kubernetes cluster's status efficiently.

2.3 Web Interface

As said before, the user interface runs over NGINX and is built with the REACT framework. It is structured in order to allow a page to visualize data and another one to post reviews. Therefore, the dashboard in figure 3 shows data computed by Spark in different ways:

- a word count table, to show the 100 most frequent words and their count;
- a word cloud, which represents the same information as the table but with a more user friendly and nice representation;
- an accuracy chart, which shows how the test accuracy of the model varies increasing the number of streaming reviews;
- a "sentiment" table, containing the predicted labels and the ground truth of the last 100 reviews.

This page performs a pool of data every two seconds and updates the page in case new data is got. Improvements are possible, such as using a web socket to update data and reduce bandwidth required, but that was not our goal.

In the management page, as shown in figure 4, we are able to post a custom review which is composed by a text and a score from 1 to 5. Then it is sent as JSON to Spark, which processes it and stores results on MongoDB.

Figure 3: Dashboard UI

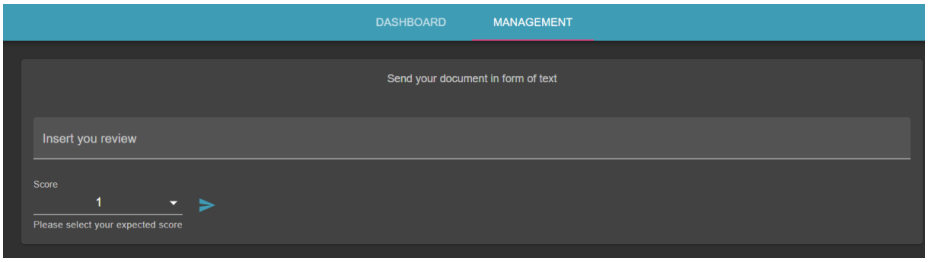


Figure 4: Management UI

3 Architecture

The user, whether it is an actual user or a system admin, has to use the application's features through our cluster exposed endpoints: API gateway, Prometheus, and Grafana. The rest of our resources are not accessible from outside the cluster by anyone but the administrators. As shown in figure 1, the user interface is accessible through a load balancer, on port 80. To reach Prometheus the port is 9090 and for Grafana the port is 3000.

One can observe the data processing pipeline in the same figure. The user can get a web interface or send text through the API gateway to the webserver or to the API server respectively. The latter is a producer for Kafka, it streams the user's text on a single topic. Our Spark cluster subscribes to that topic and receives the streams of data, then computes the word count and the sentiment analysis as explained in 2.2.1. It stores the results in our MongoDB database. API server queries data from MongoDB and updates the user web page periodically to display a dashboard with the processed data. Figure 1 does not represent the replications of our services, which will be discussed later.

4 Development process

In order to get our final system running, we started to build our single services by using docker images and running them as containers. We starting to build services as api-server, webserver and MongoDB. Instead, Kafka and Spark has been installed and tested locally for simplicity. Both servers were connected in proxy mode but we changed it later on. After that, we decided to move to AWS and use directly Kubernetes keeping everything in development mode and checking that everything works. Finally, when our application reached a good state of working, we moved webserver and api-server in production mode, removing the proxy between them and introducing an api-gateway load balancer based on NGINX.

Regarding Spark, our job can independently run locally or on Kubernetes, it just depends on the `spark-submit` command and on the services deployed locally.

5 Application life-cycle management

5.1 Amazon Web Services

Our AWS students accounts have some restrictions, listed here. The application only uses EC2 instances and S3 buckets. The machines are of type `t3.medium`, and operate on Ubuntu. To operate our code we need at least 9 EC2 instances.

5.2 Terraform

Terraform manages cloud services with infrastructure as code. One can describe a resource with code, and terraform will provide it when applied. We use Terraform to provision a S3 bucket that is necessary for kOps to create a Kubernetes cluster.

5.3 Kubernetes

Kubernetes, or K8s for short, is a container orchestrator. It manages our application life-cycle. To create and delete our Kubernetes cluster, we use the CLI tool kOps. kOps reserves resources on AWS cloud without needing to provision them beforehand.

All the software we use is running on containers and is managed by Kubernetes. To deploy our application, we use `.yaml` files to describe our pods, deployments, services and statefulsets. For automatic deployment, we wrote a few shell scripts that call Terraform, initialize the cluster and start the required services through `kubectl`. Kubernetes ensures high availability and easy monitoring, as we will discuss in the following sections.

Let's take a closer look at peculiar resources deployments.

5.3.1 MongoDB

The MongoDB Community Kubernetes Operator[6] deploys MongoDB into Kubernetes clusters. It supports scaling replica sets up and down, and creating users in the database. We used it to create a replica set with 3 replicas. We also created a regular user that has readWrite access to our *sdtddb* database and an admin user that has clusterAdmin role, with basically all privileges.

5.3.2 Spark

To run Spark on Kubernetes we followed the official documentation[7] and used `spark-submit` to submit the spark job to our Kubernetes cluster. In order to run on Kubernetes, Spark must be anyway present on the administrator's machine and the `spark-submit` command requires the user to provide a custom Spark docker image, containing the application JAR file. To obtain it, we built an image for the basic Spark installation, with Spark's `docker-image-tool` and the provided Dockerfile, and incrementally worked on it adding our application JAR and the machine learning model. Indeed, Spark does not allow to retrieve local data when executing in cluster mode. Although the official Spark documentation[8] suggests to list Spark's dependencies as `provided` when building "uber" JARs as they should be provided by the cluster manager at runtime, we had to include them when building with *sbt-assembly*[1] in order to correctly execute our application and load all dependencies, which results in a huge docker image (more than 500 MB) that has to be downloaded each time we restart the cluster.

6 Continuous integration with GitLab

We used the Ensimag GitLab to version our code. Anyway, GitLab is not just a version control platform. It offers other tools to help ease out application deployments.

6.1 Container registry

GitLab provides a container registry[9] which we used to store our private Docker images. As Kubernetes works with Docker, pods can just pull images from the registry.

6.2 Continuous integration pipelines

GitLab CI/CD[10] is a tool built into GitLab that helps developers follow continuous methodologies: continuous integration, continuous delivery, continuous deployment. We took advantage of that to build a continuous integration chain.

It works by adding a file called `.gitlab-ci.yaml` at the root of the repository. This file creates a pipeline, which are a list of steps to run, each step containing one or more jobs. Those pipelines are run when some changes are pushed to the GitLab repository and certain precised conditions are met.

The jobs are executed on a GitLab Runner[11]. The Ensimag GitLab possesses such a runner[12] that is shared between all users.

Our project only has one pipeline and one job. It builds the Docker images of API server and web server on the shared runner and pushes it to our GitLab's Docker repository. Combined with an image pull policy set to "Always" and an image tag set to "latest" for the concerned pods, one only need to push code and restart the pod to update the service on the cluster.

7 Non-functional properties

For what concerns the other aspects of our application, and more specifically non-functional properties, we performed different tests that are related to:

- fault-tolerance & availability;
- performances;
- costs.

7.1 Fault-Tolerance & availability

In this section, we analyze how strong our system is in case some pods or nodes fall down. Different techniques are possible to simulate these effects.

7.1.1 Destroying pods

A first possible technique is to destroy a pod and check if it is redeployed again by Kubernetes.

Kubernetes restart policy Kubernetes management of containerized applications allows failed pods to be restarted. Most pods of our applications are managed with "Deployments" object by K8s, so they are restarted whenever they fall into a failed state. Upon restart, all pods are able to reconnect to the others without any issue. The restart process takes a few seconds so it ensures availability together with replication. The exception is for Spark, which is explained in the next section.

It has to be noted that, for exposed pods, namely `api-gateway`, `grafana`, and `prometheus`, their public url changes when they are restarted, which can cause a denial of service. In a true production setting, we could buy a domain name and force the public urls to static values on the purchased domain name. The change of url is a process that can take up to 10 minutes in the case of our load balancer, `api-gateway`.

From our tests we also noticed that, if a MongoDB replica pod is deleted, we get errors when retrieving data, but all store operations are in a sort of queue and computed when all replicas are ready again. This means that we do not lose data in this interval of time, but we can't retrieve them.

Spark job The processing part of our application is a Spark job submitted using the `spark-submit` which spawns a `spark-driver` pod. This pod in turn creates Spark executors pods that are restarted if they are deleted. However, the `spark-driver` pod itself does not restart upon failure, as Spark overrides the pod restart policy to "never". Moreover, this pod is affected by the failure of other pods, such as `kafka`. This part of our application is a **single point of failure**.

Kafka Given our use of Kafka, with only one topic, here is our current configuration:

- Zookeeper: 1 replica
- Kafka broker: 1 replica
- Topics: 1 topic
- Topic partition: 1 partition
- Replication factor: 1

As advised by the official documentation[13], to really achieve fault-tolerance as well as high availability we should have at least 3 zookeepers, 3 brokers, a replication factor of 3 and 3 partitions. Indeed with our current configuration there would be some data loss if a broker died. To fix that issue we would simply need to change a few values in our Kafka yaml deployment file, but we wanted to keep our application demo simple.

Kubernetes master Currently, we have only one Kubernetes master, so, if it dies, the whole cluster dies too. This is identified as a **single point of failure** for our application. As advised by kOps documentation[14], we could tell kOps that we want 3 masters to ensure fault-tolerance as well as high availability, but we chose to keep only one master to keep our demonstration app simple.

MongoDB Our MongoDB cluster is a good example of a resource that takes advantage of both Kubernetes restart policy and replication. Our 3 nodes cluster ensures high availability.

7.1.2 Nodes draining

Another fundamental test is node draining. It consists in choosing a node, evicting all pods living there and powering down the node[15].

As we can see in figure 5, running a client brings to an high usage of CPU, especially from Spark service.

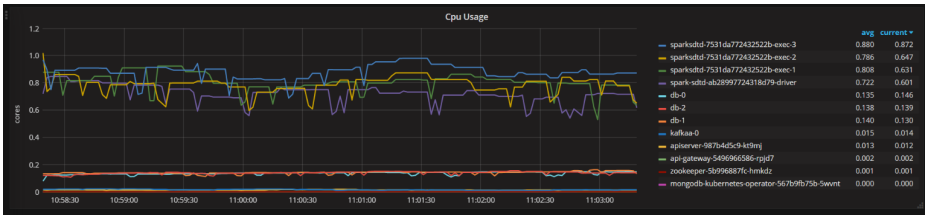


Figure 5: Resource usage, with a client which sends reviews automatically

In order to perform this kind of operation we can show all nodes with the command `kubectl get nodes`. After choosing a node, we can power it down and evict all its pods with the command `kubectl drain <node name>`. See figure 6.

In this case, we are draining a node where few Kubernetes pods and a MongoDB replica live. From what we can see in the figure 7, the MongoDB replica *db-02* reaches a zero usage CPU during the eviction process and stabilizes again after few seconds.

Then, we tried to drain a node where api-gateway, mongo operator and a Spark executor live, as shown in figure 8.

```
antop@ubuntu:~/Documents/projet-sdtd$ kubectl drain ip-172-20-33-191.ec2.internal --ignore-daemonsets --delet
e-local-data
node/ip-172-20-33-191.ec2.internal already cordoned
WARNING: ignoring DaemonSet-managed Pods: monitoring/node-directory-size-metrics-zjzbn, monitoring/prometheus
-node-exporter-hqjpt
evicting pod default/db-2
evicting pod kube-system/kube-dns-6c699b5445-8vr92
evicting pod kube-system/kube-dns-autoscaler-cd7778b7b-nnvl8
pod/db-2 evicted
pod/kube-dns-autoscaler-cd7778b7b-nnvl8 evicted
pod/kube-dns-6c699b5445-8vr92 evicted
node/ip-172-20-33-191.ec2.internal evicted
```

Figure 6: Draining a node

Figure 7: Metrics with one node less

```
antop@ubuntu:~/Documents/projet-sdtd$ kubectl drain ip-172-20-55-141.ec2.internal --delete-local-data --ignor
e-daemonsets
node/ip-172-20-55-141.ec2.internal already cordoned
WARNING: ignoring DaemonSet-managed Pods: monitoring/node-directory-size-metrics-4jml8, monitoring/prometheus
-node-exporter-8qjqq
evicting pod default/api-gateway-5496966586-rpjd7
evicting pod default/mongodb-kubernetes-operator-567b9fb75b-5wvnt
evicting pod default/sparksdtd-7531da772432522b-exec-1
pod/mongodb-kubernetes-operator-567b9fb75b-5wvnt evicted
pod/api-gateway-5496966586-rpjd7 evicted
pod/sparksdtd-7531da772432522b-exec-1 evicted
node/ip-172-20-55-141.ec2.internal evicted
```

Figure 8: Draining another node

In this case, everything is restored again after eviction, but Spark crashes as usual due to cache and checkpointing problems (figure 9).

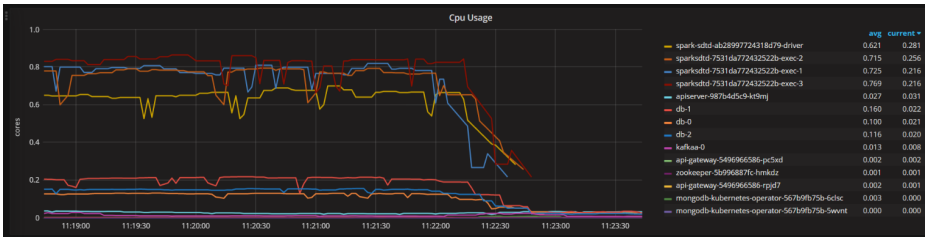


Figure 9: Metrics with 2 nodes less, after draining

Thanks to this test, we can confirm that the main problem is Spark and its crashes.

7.2 Performances

Regarding the performances, we have collected different metrics. Some of those are:

- initialization cluster time: $400s \sim 600s$;
- destroy cluster time: $200s \sim 300s$.

They are quite useful, especially to know how long is a denial of service in case a cluster update is needed.

Response time is computed taking into account the average number of requests and the average response time for each request, distinguishing GET and POST. In case a user accesses the dashboard, two GET requests per seconds are sent, so the average response time is about 0.0422s if the response is empty, otherwise about 0.050s if the response has 100 elements. In case of POST requests, we run 1, 2 and 3 clients. Some results are displayed in figure 10.

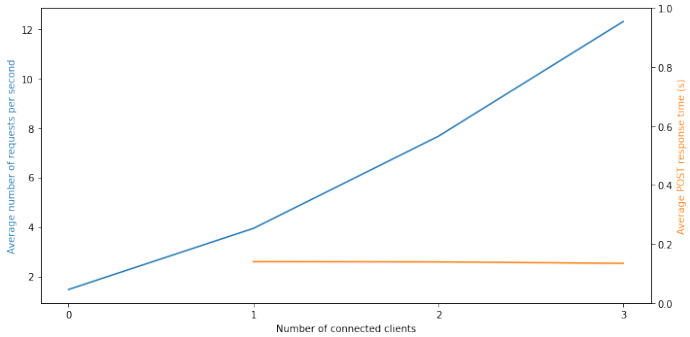


Figure 10: POST Response time

The POST response time remains almost equal to 0.134s which is $\frac{0.134}{0.0422} \cong 3.17$ times of a GET response time. As POSTs are sent asynchronously from the api-server without waiting for Kafka, we obtain a low response time and that confirms the reactivity of our application. The best approach would be to increase the number of requests and check when these values change, but it is an impossible operation considering the computation power of AWS. Therefore, it could be tested in a real traffic scenario.

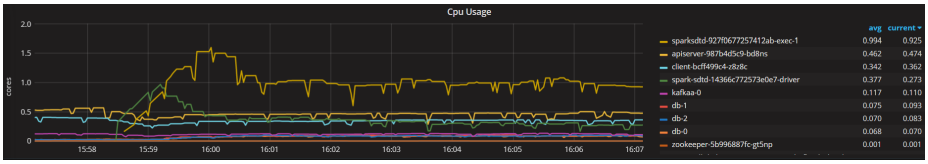
Another metric we got is the bandwidth required for the dashboard. In fact, in the worst case the dashboard gets the following responses every 2 seconds:

- a vector of 100 elements for the accuracy ($\sim$ 3.88KB);
- two vectors of 100 elements for word count and word cloud ($\sim 2 * 6.36$KB);
- a vector of 100 elements for sentiment analysis ($\sim$ 44.2KB).

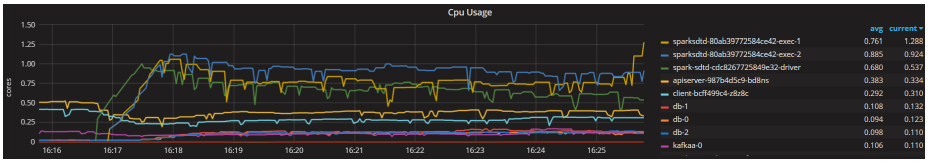
In total, a page requests 60.8 KB every 2 seconds, so $\sim$ 30.4KB/s or 0.0002432 Gbit/s. Given that we are using `t3.medium` machines, then we have available 0.25 Gbit/s as baseline[16]. Under the hypothesis that all the bandwidth is available at download side, we can compute the maximum number of active dashboards our system can support, which is $\frac{0.25}{0.0002432} \cong 1028$. If we overcome this value, then we would not be able to update the page every two seconds, with a lower reactivity of page.

Spark Different tests has been performed in order to know how much resources Spark uses. In particular, we run our Spark job varying the number of executors, from 1 to 4, which is the maximum number of supported executors with 8 `t3.medium` nodes. Each Spark job is submitted when a client is already running inside the cluster, after emptying the MongoDB collections and is run for five minutes, in order to have a coherent testing environment. We report here the results collected from Grafana and from the Spark UI.

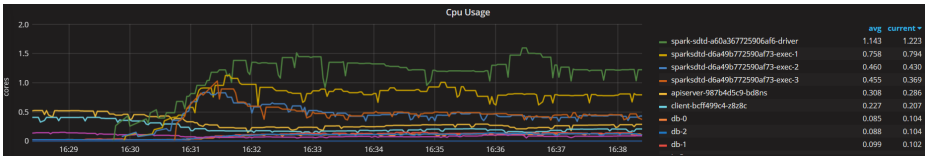
From the Grafana dashboard, some interesting results are CPU and network usage. From figure 11, it can be noticed that, with one executor, the executor has a high CPU usage while the driver's consumption is much lower. Instead, with four executors it is the opposite, with a high CPU usage for the driver and a lower consumption for the executors. With more than one executor, one of them has a much higher CPU usage than the others.



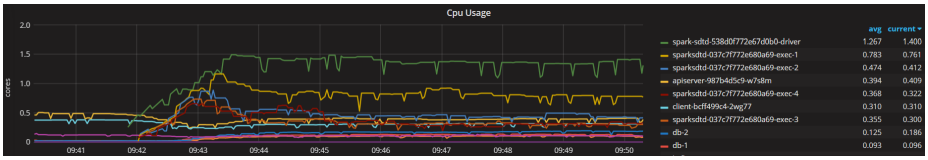
(a) 1 executor



(b) 2 executors



(c) 3 executors



(d) 4 executors

Figure 11: CPU usage

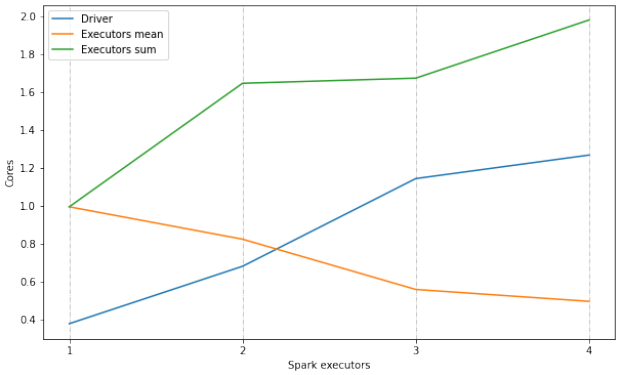
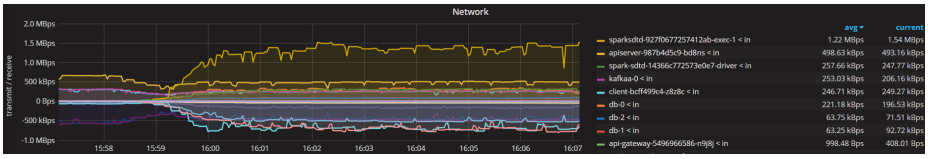
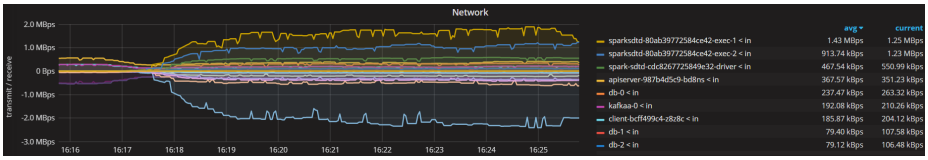


Figure 12: Spark CPU usage varying the number of executors

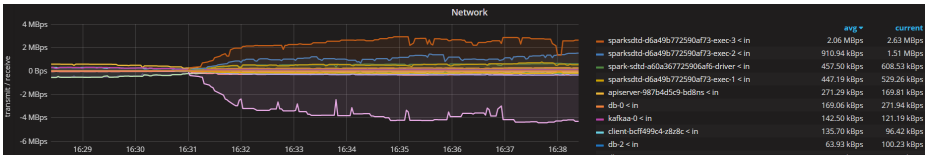
This behavior is confirmed by the network usage in figure 13, with a high incoming network traffic for a single executor. The curve with the most significant outgoing network traffic corresponds to Spark driver. Its outgoing traffic increases with the number of executors, as it must forward data to all of them.



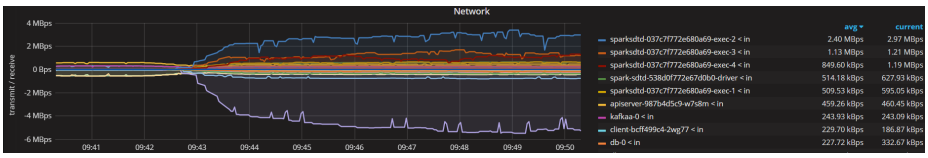
(a) 1 executor



(b) 2 executors



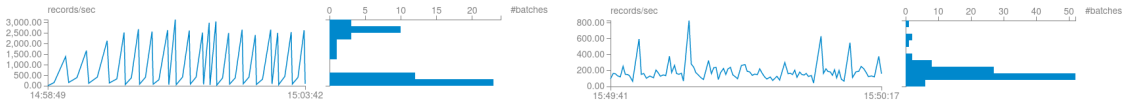
(c) 3 executors



(d) 4 executors

Figure 13: Network usage

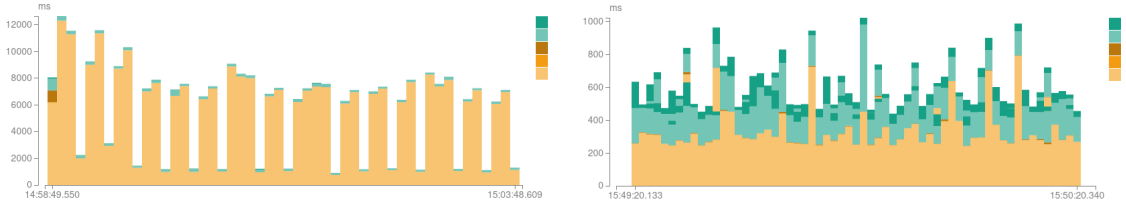
From Spark 3.0, a Structured Streaming UI is available inside Spark Web UI, through which streaming queries jobs can be monitored with information and statistics. For our three queries, we can visualize input rate, process rate, input rows, batch duration and operation duration. At first glance, a few interesting information emerges. Generally, varying the number of executors the input rate is comparable, while the process rate has high peaks with one executor and tends to stabilize with more executors (figure 14). The duration of the micro-batches also tends to decrease.



(a) 1 executor (b) 4 executors

Figure 14: Process rate

Finally, the operation duration chart is also interesting. It shows the amount of time taken to perform various operations in milliseconds. As reported in figure 15, with one executor operations take a long time and are mainly devoted to the *addBatch* operation (in light orange), the time taken to read, process and write the output of the micro-batch. With four executors, the operation time is much shorter and the *addBatch* time is comparable to the sum of *walCommit* and *queryPlanning* (in green), in which metadata and execution plan are generated[17]. With many executors the driver's workload is higher, as it has to manage and distribute queries among them.



(a) 1 executor (b) 4 executors

Figure 15: Operation duration

7.3 Costs

Regarding the costs, we analysed the AWS Cost Explorer dashboard. It allows us to get an estimation of the money spent per day and also forecasts for the end of the month (figure 16).

We would have liked to get these metrics per hours, but unfortunately we are not able to due to the limitations of the educate accounts. For this reason, we opted to run the cluster for one hour with three Spark executors in two different scenarios, running in idle or receiving reviews from active automatic external clients. In this way, we could get an approximation of the cost per request. However, this cost is affected by fixed costs to run a cluster and by other hidden costs.

The cost in idle is mainly due to creation and destruction of cluster, but also by a small CPU usage of Spark which is about 0.1 core, as shown in figure 17.

Although we got some costs, they are not so reliable because of just one hour of test and few requests. Therefore, we should for instance test for more hours with huge traffic in order to reduce the estimation error.

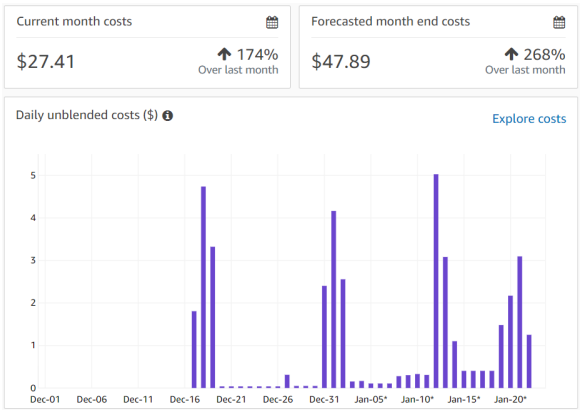


Figure 16: AWS cost explorer example

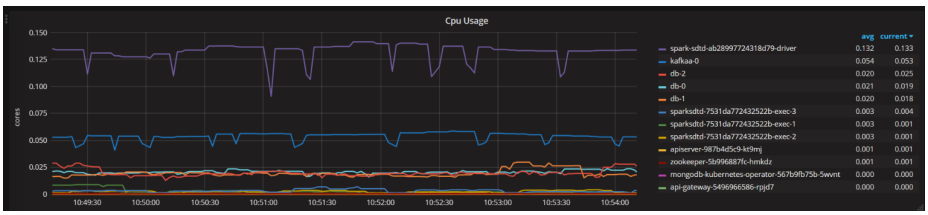


Figure 17: Cluster usage in idle

8 Conclusion

In general, considering all statistics we collected during our tests, we can say that our system is able to run quite well, although we had some problems with Spark. Even though some pods are running in production mode, our cluster is not, we are still far from a production grade application, so there are different improvements that we could apply, mainly:

- add replicas to Kafka in case of high load;
- use autoscaler for api-gateway, api-server and webserver;
- implement Spark checkpointing, in order to exploit at the maximum the Structured Streaming system and be able to recover from failures;
- enable restart policy to "Always" for the Spark driver;
- use web sockets instead of pooling on dashboard to reduce bandwidth.

To all readers, we thank you for reading this report.

References

[1] *GitHub - sbt/sbt-assembly: Deploy fat JARs. Restart processes. (port of codahale/assembly-sbt)*. URL: https://github.com/sbt/sbt-assembly.

[2] *Structured Streaming Programming Guide - Spark 3.0.1 Documentation*. URL: http://spark.apache.org/docs/latest/structured-streaming-programming-guide.htmls.

[3] *MongoDB Connector for Spark — MongoDB Spark Connector*. URL: https://docs.mongodb.com/spark-connector/current.

[4] *Amazon review data*. URL: http://jmcauley.ucsd.edu/data/amazon/.

[5] *tf-idf - Wikipedia*. URL: https://en.wikipedia.org/wiki/Tf-idf.

[6] *MongoDB Community Kubernetes Operator*. URL: https://github.com/mongodb/mongodb-kubernetes-operator.

[7] *Running spark on Kubernetes - Spark 3.0.1 Documentation*. URL: https://spark.apache.org/docs/latest/running-on-kubernetes.html#submitting-applications-to-kubernetes.

[8] *Submitting Applications - Spark 3.0.1 Documentation*. URL: http://spark.apache.org/docs/latest/submitting-applications.html.

[9] *GitLab container registry*. URL: https://gitlab.ensimag.fr/sdtd1/projet-sdtd/container_registry.

[10] *GitLab CI/CD Docker integration*. URL: https://docs.gitlab.com/ee/ci/docker/using_kaniko.html.

[11] *GitLab runner*. URL: https://docs.gitlab.com/runner/.

[12] *Ensimag GitLab shared runner*. URL: https://gitlab.ensimag.fr/sdtd1/projet-sdtd/-/settings/ci_cd#js-runners-settings.

[13] *Running Kafka in production*. URL: https://docs.confluent.io/platform/current/kafka/deployment.html.

[14] *Kops - production settings*. URL: https://kops.sigs.k8s.io/getting_started/production/.

[15] *Safely Drain a Node — Kubernetes*. URL: https://kubernetes.io/docs/tasks/administer-cluster/safely-drain-node/.

[16] *EC2 network performance of t3*. URL: https://cloudonaut.io/ec2-network-performance-of-t3/.

[17] *How to Better Monitor Streaming Queries with Spark 3.0 Structured Streaming - The Databricks Blog*. URL: https://databricks.com/blog/2020/07/29/a-look-at-the-new-structured-streaming-ui-in-apache-spark-3-0.html.