ENSIMAG

RAPPORT

SDTD

Réalisé par:

BRITEL Malak
GOUZERH Yohan
JACECZKO Benoît
LARAQUI Omar
MRINI Ayoub

Encadré par:

Ropars Thomas, Professeur,
UGA

2018-2019

Contents

1 Introduction

Notre projet consiste à visualiser sur une carte l'humeur moyenne de chaque pays, à l'aide de l'analyse des tweets émanant de ces différents pays. Pour nous affranchir de la barrière des langues, nous avons décidé d'analyser uniquement les emojis émis par les utilisateurs. Le concept du projet est de traiter ces données tout en faisant appel à des systèmes distribués. Nous essayerons tout au long de ce rapport d'expliciter les moyens déployés ainsi que les outils utilisés pour la réalisation de ce projet.

2 Description des composants logiciels utilisés

Le pipeline par lequel transitent les données avant d'être consommées par l'application web comporte un ensemble de briques logiciels, chacune en charge de la réalisation d'un processus particulier.
La figure 1 décrit l'architecture globale.

FIGURE 1: Architecture

2.1 Couche d'ingestion : Kafka

Au départ, les tweets sont extraits à l'aide de l'API tweeter (utilisation de la librairie python tweepy), et passent par le message broker de Kafka. Le rôle principal de Kafka est de fournir un système unifié, en temps réel à latence faible pour la manipulation de flux de données.
Kafka tient sa puissance du fait qu'il peut organiser les flux de données par groupes

dont chacun peut contenir un ensemble de topics. Le traitement de chacun des messages donne la possibilité d'associer une clé par message au sein d'un topic particulier, et donc offre de grandes possibilités pour le traitement des flux de messages tout en restant scalable, tolérant aux pannes et rapide.
En sortie de cette étape, Kafka renverra une série de messages nettoyés[1]

2.2 Couche d'agrégation : Spark

Les messages provenant de Kafka passent ensuite par cette couche qui porte sur l'agrégation des données, et comme nos données sont reçues en flux, nous utilisons Spark en sa version Streaming, de telle sorte à lancer des opérations de Map-Reduce à intervalles de temps réguliers. C'est dans ce contexte qu'une nouvelle abstraction appelée Resilient Distributed Datasets (RDDs) permet de réutiliser efficacement les données dans une large famille d'applications. Les RDDs sont tolérants à la panne et proposent des structures de données parallèles qui laissent les utilisateurs:

1. Persister explicitement les données intermédiaires en mémoire,
2. Contrôler leur partitionnement afin d'optimiser l'emplacement des données,
3. Manipuler les données en utilisant un ensemble important d'opérateurs.

Chaque RDD généré comporte des données qui seront stockées sur Cassandra.

2.3 Couche de stockage : la base de données distribuée, Cassandra

Après les traitements effectués sur Spark, les emojis extraits des tweets sont stockés dans Cassandra.
Pour rappel, Cassandra est une base de données distribuée en NoSQL, qui suit une structure en paires clé-valeur, et dont les avantages sont la haute disponibilité et la haute performance.
Étant donné que Spark envoie un "paquet" d'emojis extraits à intervalles réguliers, nous avons opté pour un schéma de table permettant le stockage des emojis extraits durant cet intervalle. Par exemple, la première entrée de la table représentée dans la Figure 2 indique que pendant un intervalle, 12 smilies souriants ont été extraits, et la date de stockage dans la base est conservée. Celle-ci servira en particulier dans la partie application pour récupérer les dernières entrées.

[1] On ne renverra que les émois et enlèverons le reste du texte afin d'alléger la charge de la partie agrégation

FIGURE 2: Table EmojiPackage

id	pays	id_emoji	nb_occurence	package_date
1	France		12	13:26:52
2	Allemagne		7	13:27:04

3 Automatisation de notre solution

3.1 Introduction : Utilisation de boto3

Pour automatiser la création des machines EC2, nous avons décidé de partir de l'API boto3 en python. Cette api fournit quasiment les mêmes fonctionnalités de AWS Cli, mais permet une meilleure organisation du code qu'un simple script bash.

Nous pensions au départ que les étapes à réaliser seraient simples, que nous avions juste à lancer quelques commandes, cependant l'automatisation se révéla bien plus complexe que nous l'avions prévu.

3.2 Architecture du projet

Virtualenv & requirements.txt

Afin d'éviter de polluer les machines des utilisateurs, nous avons utilisé le principe des virtualenv. De plus, l'utilisation d'un fichier requirements.txt, permettant de lister les dépendances, nous permet de récupérer assez rapidement les dépendances utiles.

Stateless

L'application est sans état, afin de permettre à un utilisateur externe d'effectuer des commandes directement sans avoir à relancer l'application. Nous avons pour cela utilisé les modules de tagging, de récupération des données, et de fichiers de configuration que nous avons créés (Voir les prochaines parties).

Défi : gestion des importations

Au niveau architecture du projet, nous avions décidé d'organiser le code en différents répertoires et sous-répertoires. Nous n'avions pas pris en compte le mécanisme d'import assez particulier en python[2], mécanisme assez complexe à prendre en main pour les gros projets, que nous n'avions pas encore rencontrés dans nos précédents projets.

Nous avons passé un certain temps avant de comprendre qu'il fallait toujours exécuter un script python depuis la racine du projet, et comment réaliser de bonnes importations. Nous avons donc créé à la racine un script manage.py qui nous permet d'utiliser notre système grâce à une api simple.

Pour créer ce script, nous avons utilisé le module Fire[3] de Google, permettant de créer des commandes très rapidement, sans avoir besoin de parser. Ce module nous a ainsi grandement aidé dans le développement de notre solution, et permis de regagner un peu du temps perdu avec la question des imports.

3.3 Développement des fonctionnalités

Contrôle du cycle de vie d'une ec2

Nous sommes partis au départ du management des EC2 déjà créées, c'est à dire d'activer les fonctions start / stop / terminate. Ces fonctions vérifient de plus les droits sur les EC2, et si il est possible d'effectuer ces actions dessus. Les fonctions start et stop ne nous ont pas servies par la suite, cependant nous avons utilisé la fonction terminate afin de supprimer automatiquement toutes les machines EC2 utilisées.

Tag des ressources

Ensuite, pour éviter de travailler sur des machines EC2 utilisées pour d'autres projets, nous avons décidé de tagger toutes nos machines avec le nom du projet.

[2]Voir le principe du python path : http://sametmax.com/les-imports-en-python/
[3]Fire : https://github.com/google/python-fire

Nous avons ainsi réalisé un "Tagguer", module qui nous permet de tagguer rapidement une ressource.

Récupérer des informations sur notre infrastructure

Nous avons développé un module permettant d'obtenir de multiples informations, comme la récupération de l'instance sur laquelle nous avons mis le master kubernetes, son addresse ip publique, privée, les adresses publiques et privées des workers,...

Cela nous est très pratique notamment pour la partie de déploiement de la partie applicative de notre solution.

Gestionnaire de configuration

Afin de pouvoir contrôler les paramètres que nous utilisons, nous avons utilisé un système de fichiers de configuration. Plusieurs choix s'offraient à nous : yml, json, ini,... Nous sommes partis sur des .ini, étant donné que le système est plus simple à utiliser pour un utilisateur lambda que json, qu'il n'y a pas de soucis d'indentation comme en yml, et qu'il y a un parser directement dans la librairie standard.

Le gestionnaire de configuration que nous avons créé fut justement ce qui nous a permis de découvrir le fameux problème des imports, étant donné qu'il doit être accessible de n'importe où dans le projet.

La décision d'avoir utilisé ce gestionnaire fut bénéfique pour la suite, il nous a permis dans le futur de développer plus rapidement de nouvelles fonctionnalités.

Création d'un vpc

Après cela, nous nous sommes occupés de la création d'un VPC (Virtual Private Cloud), pour qu'une personne qui n'en a pas créé un, puisse utiliser notre script directement.

C'est ici où nous avons commencé à nous rendre compte que les actions réalisées grâce à l'interface web, cachaient certaines actions qui sont effectuées en arrière plan, et qu'il est nécessaire d'effectuer manuellement en utilisant l'API. Par exemple, il est nécessaire de créer une internet gateway, créer une route, activer le DNS,...

Gestion de la sécurité

Nous nous sommes surtout rendus compte qu'il fallait redéclarer beaucoup de choses manuellement lors de la mise en place de la sécurité, qui fut le plus gros bottleneck ici.

Par exemple, pour associer un rôle à une EC2, il faut tout d'abord définir une policy, puis ensuite associer cette policy à un rôle. Ensuite, il faut créer un profil d'instance, associer le rôle avec ce profile d'instance, puis ensuite associer ce profile d'instance à l'EC2. Pour la suppression, il faut ensuite annuler récursivement toutes ces étapes.

Gestion de la cohérence

Nous nous sommes rendus compte de la difficulté dans un système distribué comme AWS d'obtenir un état de cohérence dans tout le système. Par exemple, une des grosses difficulté que nous avons eu et qui nous a pris beaucoup de temps à déboguer, fut la découverte de ressources fantômes. La modification d'une ressource AWS prend du temps à être diffusée à tout le réseau.

Par exemple, lorsque nous supprimions un rôle pour en recréer un autre[4], la machine EC2 utilisait alors l'ancien rôle, car pour elle, elle ne savait pas encore que ce rôle avait été supprimé et remplacé. Il n'y avait pas d'erreur détectée ensuite par le système, car le nouveau rôle portait le même nom, et alors que même que nous avions une référence nulle. La seule solution que nous avons trouvée fut de mettre un temps d'attente arbitraire entre la suppression d'un rôle et la création d'un autre.

Pour la création d'EC2, c'est un peu la même problématique mais en plus simple. Le temps de création est aléatoire. Pour vérifier si une instance EC2 a bien été créée, nous allons ainsi récupérer à intervalle régulier la liste des EC2 en mode running, et vérifier si notre EC2 est parmi cette liste.

[4] fonctionnalité créé au cas où entre deux déploiement ce rôle avait changé, ce qui arrivait dans nos phases de développement de développement

3.4 Automatisation des différents processus

3.5 Automatisation de la création de l'infrastructure

Pour automatiser la création de l'infrastructure, nous regardons tout d'abord si un VPC du projet smack-it est déjà en place. Si oui, nous supprimons les instances ec2, sinon nous recréons un vpc complet. Ensuite, nous créons les instances dans ce vpc. Le tout en utilisant les paramètres de notre fichier de configuration.

3.6 Automatisation du déploiement de la partie applicative

Pour automatiser la création du cluster kubernetes, nous allons procéder en deux étapes.

Tout d'abord, nous allons créer l'infrastructure grâce au système précédent, et générer une clé privée nous permettant de nous connecter aux instances ec2.

Ensuite, nous allons récupérer les adresses ip du master et des workers grâce là aussi à notre précédent système.

Nous allons ensuite procéder à l'installation par envoi de fichiers de configurations et de scripts aux différentes instances par scp, puis nous allons nous connecter en ssh sur chaque machines afin de lançer les scripts.

Une amélioration possible serait d'effectuer ces commandes en parallèle afin de diminuer le temps de création pour un grand nombre de machines.

3.7 Conlusion sur la partie automatisation

Cette partie fut très gourmande en terme de temps. Nous avons ici au final recréé notre propre Ansible, fonctionnant aussi par fichier de configuration, et permettant de créer des instances et déployer notre partie applicative sur celles-ci.

Ce fut enrichissant d'un point de vue compréhension, mais le travail se réduisait surtout à de longues heures de débogage (notre infrastructure met aux alentour de 5 minutes à se lancer à chaque fois pour 3 machines).

Nous proposons aux prochaines équipes de ne pas utiliser boto3 et de partir directement sur des solutions robustes comme Ansible ou bien Terraform. Au lieu de

passer leur temps sur du debogage, elles pourront se focaliser sur des parties plus intéressantes comme la scalabilité par exemple, partie que nous n'avons pas eu le temps de faire.

4 Orchestration par Kubernetes

Le pipeline présenté dans la section 2 a été entièrement déployé dans un cluster Kubernetes, qui assure l'opérabilité de tous les composants, la communication entre eux ainsi que d'autres fonctionnalités liées au pipeline en entier, à savoir la résilience, la scalabilité et la haute disponibilité.

4.1 L'environnement Kubernetes

Le cluster Kubernetes utilisé est créé à l'aide de l'outil *kubeadm*, qui permet d'initialiser un master avec tous les composants clés Kubernetes, notamment Le scheduler qui répartit les workloads entre les noeuds et le controller manager qui assure que l'état du système correspond à l'état désiré dans les paramètres de configuration Etcd[5]. Il permet aussi de joindre les workers qui vont héberger à leur tour les différents clusters propres aux composants du pipeline. Le schéma indiqué dans la figure 3 représente avec plus de détails l'architecture de l'application.

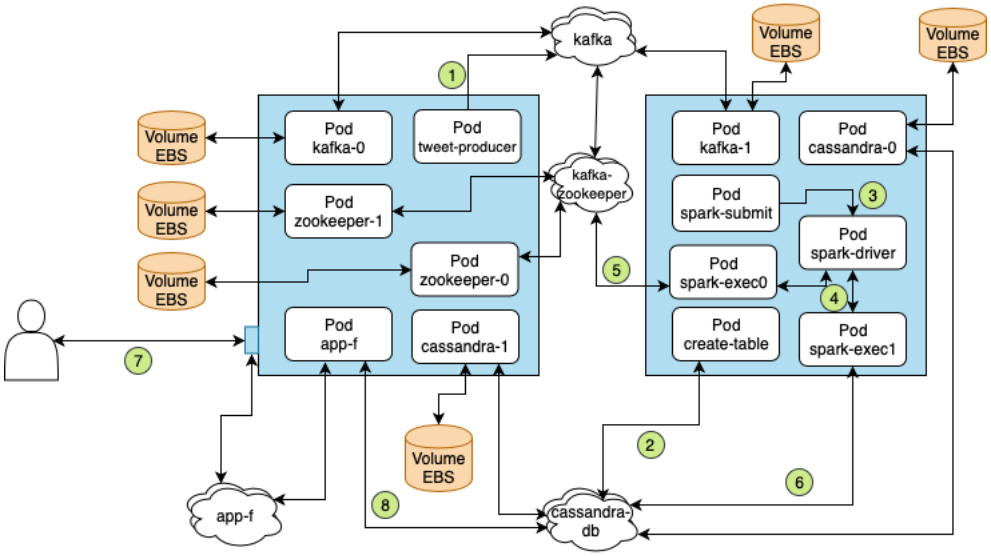

FIGURE 3: Architecture détaillée du pipeline.

Les numéros sur le schéma 3 montrent le parcours des tweets, au début le générateur aléatoire tweet-producer envoie des tweets aux brokers Kafka via le service kafka, le pod spark-submit lance le cluster Spark qui traitera les messages récupérés

[5]Base de données clé-valeur distribuée

via le service kafka-zookeeper et stocke les résultats dans le cluster Cassandra via le service cassandra-db, l'utilisateur peut alors se connecter à l'application qui tourne sur le pod app-f et qui affiche les résultats tirés de Cassandra.

Kubernetes est destiné en premier lieu à la gestion des applications dites stateless, celles qui ne sont pas censées stocker de la donnée et qui n'ont pas besoin d'une découverte et listages des peers. Le déploiement d'une base de donnée multi-noeuds comme Cassandra ou d'un système de messages distribué comme Kafka qui ont comme objectif de manipuler des données persistantes et qui demandent une cohérence et une communication stable entre leurs noeuds, entraîne l'utilisation de ressources Kubernetes spéciales et nécessite beaucoup plus de configuration. Pour avoir accès à des volumes persistants, Un rôle IAM[6] permettant de créer et de gérer des volumes EBS[7] d'AWS a été attribué à tous les workers Kubernetes, ce qui autorise leurs Pods à se procurer d'une façon dynamique des volumes avec une taille donnée.

5 Propriétés non fonctionnelles

1. Débit : En simulant un flux de tweets, notre système arrive à traiter 150000 tweets par seconde.

2. Temps de réponse : Étant donné que Spark envoie à Cassandra les emojis extraits toutes les secondes, le temps de trajet d'un emoji de l'API Tweeter jusqu'à son impact sur l'application web est d'environ une seconde.

3. Temps de déploiement : Pour déployer un système avec un master et 8 workers, le temps nécessaire est de plus d'une dizaine de minutes.

4. Tolérances aux pannes : A l'heure actuelle, notre système ne gère pas les pannes.

6 Application web

Concernant notre application web, nous avons opté pour une carte du monde interactive et dynamique. En cliquant sur un pays, on affiche son nom et un emoji représentant la tendance de l'humeur actuelle du pays d'après les emojis extraits plus tôt.
Pour ce faire, nous avons utilisé Leaflet, qui est une librairie open source de JavaScript

[6] Sert à fournir des autorisations à utiliser les ressources AWS.
[7] espace de stockage persistant sur AWS.

pour les cartes interactives. Pour accéder à cette carte, nous avons utilisé Flask qui est un framework Open-source de développement web en Python.

7 Problèmes rencontrés

7.1 création des EBS, tag, IAM, Setting up the Kubernetes AWS Cloud Provider

7.2 Limitation de l'API tweeter

A cause de la limite imposée par l'API tweeter, le nombre de tweets reçus en streaming est assez limité, ce qui nous empêchait d'obtenir des résultats intéressants. Pour pallier à ce problème, nous avons pensé à deux solutions : premièrement, nous avons stocké dans un fichier un grand nombre de tweets, pour une utilisation ultérieure. Deuxièmement, nous avons créé un script qui crée des exemples de tweets, pour simuler le flux de tweets.