ÉCOLE SUPÉRIEURE D'ÉLECTRICITÉ

Rapport de projet de
Fin d'études

Conception d'une architecture Spark Streaming

Elèves :
FRIANT Hugo
GICQUEL Adrien

Chargé de projet : M. Vialle

26 Mars 2020

Table des matières

Chapitre 1

Contexte et objectifs de l'étude

Le traitement de grande quantité de données est une problématique qui se pose aussi bien pour la recherche universitaire que pour de nombreuses entreprises, en devenant presque indispensable pour continuer à évoluer dans le monde d'aujourd'hui. La collecte de données a explosé mais ces données restent trop souvent inexploité et ne sont pas valorisées. Dans ce contexte il est indispensable de développer des solutions qui permettent d'extraire de l'information efficacement, en passant à l'échelle facilement mais surtout ces solutions doivent être suffisamment génériques.

Le calcul distribué, sous condition de disposer d'installation suffisamment puissante, apparaît comme le seul moyen de contrebalancer la quantité immense de données. On notera que les outils qui permettent de distribuer les calculs, en particulier pour des opérations de type map/reduce existent et ne cessent de progresser mais encore faut-il les intégrer dans une architecture matérielle et logicielle qui permettent d'atteindre les performances recherchées. Parmi ces outils on citera Spark de l'éditeur de logiciel libre Apache sur lequel il a été choisit de se concentrer.

Au delà de l'aspect quantitatif, il ressort dans certaines situations, la nécessité d'effectuer le traitement des données en temps réel ou dans un temps très limité sous peine de rater une information essentielle. On pourra donner l'exemple classique de l'analyse de logs qui peut apporter des informations critiques sur des infrastructures. Il y a bien sûr aussi les milliards de messages générés sur les réseaux sociaux ou bien l'instantanéité du système bancaire et boursier.

Face à ce besoin de traiter des flux de données volumineux, ce projet a pour objectifs d'exploiter les outils proposés par Apache au travers de Spark et son écosystème en les déployant sur un cluster de PCs. Dans un premier temps il s'agira d'identifier l'architecture matérielle et logicielle à la fois disponible et nécessaire, puis d'interfacer les outils de calculs distribué en temps réels avec un système de fichier HDFS déjà en place. Deux exemples concrets d'utilisation viendront appuyer la faisabilité et l'intérêt d'un tel système.

Chapitre 2

Plateforme logicielle et matérielle utilisée

2.1 Hadoop File System (HDFS)

Un système de fichiers distribué est un système de fichiers qui gère le stockage de données sur plusieurs machines connectées sur un réseau. Par nature, un système de fichiers distribué est nécessairement plus complexe qu'un système de fichiers fonctionnant sur une seule machine. L'une des parties les plus difficiles est de s'assurer que les données ne sont pas perdues lorsqu'une machine tombe en panne.

Le Hadoop Distributed File System (HDFS) est l'implémentation d'un système de fichiers distribués utilisé par Hadoop qui présente les caractéristiques suivantes :

— Il gère des fichiers très volumineux, dont la taille peut être de l'ordre de plusieurs téraoctets.

— Le HDFS fonctionne avec du matériel de base, ce qui signifie qu'il n'a pas besoin de machines à l'architecture spécifique ou très puissantes.

Comme tout autre système de fichiers, le HDFS divise un fichier en une collection de blocs, c'est-à-dire des unités de données uniques. Cela signifie que chaque opération de lecture charge des blocs entiers de données, il n'est pas possible de lire un demi-bloc par exemple. Dans un système de fichiers classique fonctionnant sur une seule machine, la taille des blocs est de quelques centaines d'octets (par exemple 512), alors que dans le HDFS, la taille des blocs est considérablement plus importante (64 Mo par défaut), un choix qui vise à minimiser le coût des recherches. Plus précisément, si un bloc est suffisamment grand, le temps de recherche de ce bloc est considérablement plus court que le temps de lecture. En d'autres termes, prendre trop de temps pour rechercher un petit bloc à lire est une perte de temps. Bien sûr, la taille du bloc ne peut pas être trop grande, sinon chaque lecture prendrait un temps excessif, tout en chargeant trop de données qui pourraient même être inutiles.

Il y a trois grands avantages à stocker les fichiers sous forme de blocs :

— Un fichier peut être plus grand que n'importe quel disque sur toutes les machines du cluster. En effet, les blocs composant un fichier ne sont pas nécessairement stockés dans la même machine. En poussant ces considérations à l'extrême, il serait possible de remplir le système de fichiers du cluster avec un seul fichier.

— Les blocs facilitent la réplication. Pour garantir la tolérance aux pannes, le HDFS réplique chaque bloc d'un fichier sur trois machines différentes. Si une machine tombe en panne, le bloc reste accessible sur les deux autres.

Un cluster HDFS se compose de deux types de noeuds, un noeud "namenode" (le "master") et les noeuds de données (les "workers"). Le noeud "master" est une composante essentielle du HDFS car il conserve l'arborescence de tous les fichiers du système de fichiers

et la position de chaque bloc dans le cluster. Les blocs sont stockés par les noeuds de données. Une application cliente obtient les fichiers du système de fichiers en demandant au noeuds "master" de les récupérer. Le noeud "master" ordonne aux noeuds de données qui contiennent les fichiers demandés de récupérer les blocs qui composent les fichiers. Si la machine correspondant au "namenode" tombe en panne, l'ensemble du système de fichiers ne fonctionne plus et les données sont perdues car il n'y a aucun moyen de savoir où se trouvent les données. Afin d'éviter cette situation, Hadoop fournit depuis la 2ème version de HDFS, la possibilité d'avoir un "namenode" actif et un namenode de secours. Lorsque le "namenode" actif est inaccessible, le "namenode" secours est activé. [Que18]

le fonctionnement est résumé dans le figure ci dessous.

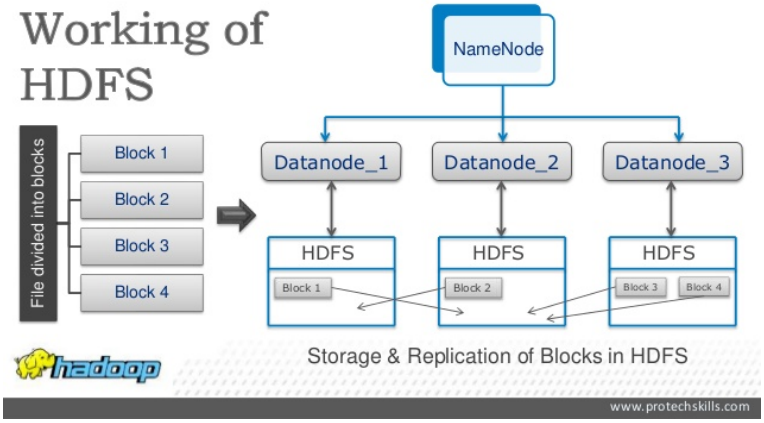

FIGURE 2.1 – Représentation du Hadoop File System

2.2 Apache Spark

Apache Spark (ou Spark) est un framework open source de calcul parallèle pour le traitement massif de données. Le projet Spark est né en 2009 dans un laboratoire de l'université de Berkley avec des chercheurs qui avaient notamment travaillé sur Hadoop. Leur constat est qu'Hadoop est très peu efficace lorsqu'il s'agit de faire des calculs itératifs.

En effet, les calculs itératifs nécessitent souvent de réutiliser les résultats des calculs précédents, une chose qu'il n'est possible de faire sur Hadoop qu'en écrivant les résultats intermédiaires sur un système de fichier Hadoop (HDFS). Cependant, écrire des fichier sur un système HDFS est une réelle pénalité de temps car les opération de lecture et d'écriture sont longues et le fichier doit être répliqué sur plusieurs disques. Par conséquent, le temps nécessaire à l'écriture des résultats intermédiaires peut dépasser le temps de calcul réel. Certains frameworks ont été proposés pour traiter cette question. Par exemple, Pregel est un système qui supporte des calculs itératifs sur des graphiques, tandis que HaLoop fournit une implémentation d'une version itérative de MapReduce. Une autre limite de Hadoop est qu'il ne peut effectuer que des traitements de données par batchs et doit s'appuyer sur des outils externes pour le traitement des données en continu et pour le machine learning.

La nouveauté d'Apache Spark est qu'il permet de réaliser des calculs itératifs efficacement et de traiter des flux en temps réel, de manière interactive, sous forme de graphes, en mémoire ou par batchs. Le tout, dans un seul et même framework. De plus, les applications Spark peuvent être écrites avec différents langages de programmation, tels que Python, Java, Scala et SQL et peuvent être utilisées avec d'autres outils Big Data, dont Hadoop. Il est important d'observer que, contrairement à Hadoop, Spark ne fournit pas de système de fichiers distribué ; par conséquent, Spark s'appuie sur les systèmes de stockage existants, y compris les HDFS et les bases de données.

Spark est composé de multiples éléments présenté sur la figure ci dessous.

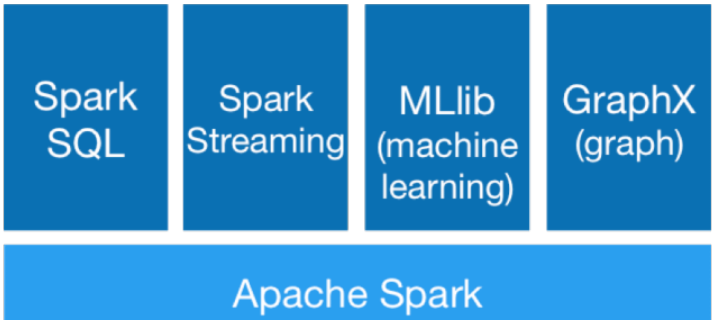

FIGURE 2.2 – Stack Apache Spark

— **Spark Core** inclue les fonctionnalités de base de Spark, telles que la planification des tâches, la gestion de la mémoire, la récupération suite à une erreur et les interfaces avec les systèmes de stockage sous-jacents.
— **Spark SQL** fournit une interface SQL pour le traitement des données structurées dans Spark. Spark ne prend pas seulement en charge les bases de données relationnelles mais aussi d'autres données structurées telles que JSON. Une caractéristique intéressante de Spark SQL est la possibilité d'intégrer des requêtes SQL avec du code écrit en Java, Python et Scala dans un seul programme.
— **Spark Streaming** offre des possibilités de traiter des données en temps réel. Nous verrons par la suite que Spark SQL le permet aussi.
— **MLib** est une bibliothèque d'apprentissage automatique qui propose la mise en œuvre de plusieurs algorithmes de machine learning. Les implémentations sont spécifiquement conçues pour s'étendre sur les noeuds d'un cluster.

Le choix d'intégrer étroitement plusieurs éléments dans un seul framework présente plusieurs avantages. Premièrement, étant donné que les composants de niveau supérieur (tels que Spark SQL et MLib) dépendent fortement du noyau Spark, toute amélioration de ce dernier a un effet positif immédiat sur le niveau supérieur. Deuxièmement, l'utilisation d'un seul cadre d'application au lieu d'une douzaine simplifie considérablement la vie d'une organisation. Enfin, Spark permet le développement d'un programme qui combine de manière transparente différents processus, comme par exemple une application qui applique des algorithmes d'apprentissage automatique à des données en temps réel et qui analyse les résultats à l'aide de SQL. On peut donc avoir une chaîne entière de traitement de données avec des composantes temps réel et d'autres non.

2.3 Spark Streaming et Structured Streaming

Depuis sa création, Spark permet de traiter des données en temps réel grâce à Spark Streaming qui fonctionne sur les RDDs. Cependant depuis la version 2.0 de Spark, une nouvelle bibliothèque a fait son apparition : Structured Streaming qui elle fonctionne sur les *DataFrames*. Le *DataFrame* est un concept Spark équivalent d'une table dans une base de données relationnelle, mais apportant un plus haut niveau d'optimisation.

Lors de notre projet, nous avons principalement travaillés avec Spark Structured Streaming, c'est pourquoi nous développerons principalement ce point.

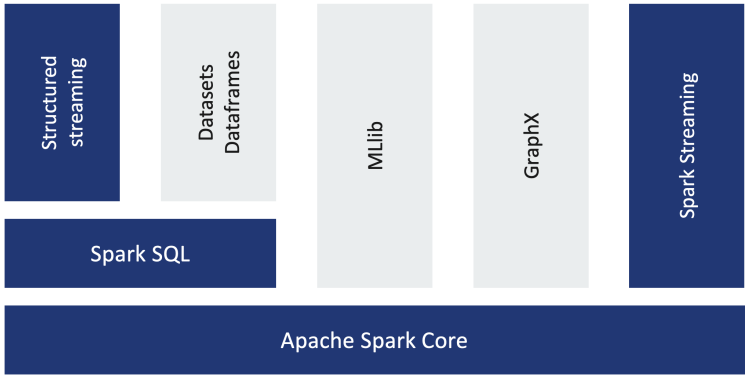

FIGURE 2.3 – Bibliothèques de streaming dans Apache Spark

2.3.1 Compatibilité avec Spark SQL

Le principal avantage de Spark Structured Streaming est qu'il est construit sur Spark SQL. Ainsi, la majorité des fonction de Spark SQL sont disponibles avec l'outil de streaming. Cela évite d'avoir à ré-écrire des fonctions qui ne seraient pas optimisées avec des RDDs comme c'était le cas dans Spark Streaming.

Vous pouvez donc appliquer toutes sortes d'opérations sur les DataFrames/Datasets de streaming. Allant des opérations non typées, de type SQL (par exemple select, where, groupBy), aux opérations typées de type RDD (par exemple map, filter, flatMap). [Apa]

2.3.2 Le modèle de programmation

Spark Structured Streaming traite toutes les données arrivant en les considérant comme les lignes d'un tableau sans limite. Chaque nouvel élément dans le flux est comme une ligne du tableau d'entrée. Cependant, Spark ne conserve pas toutes les données d'entrée, mais les résultats seront équivalents à un travail par batchs.

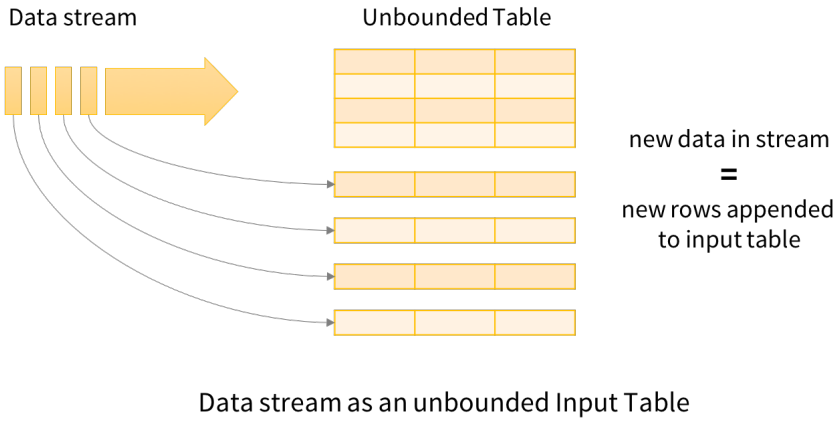

FIGURE 2.4 – L'entrée de Spark Structured Streaming : un tableau sans limites

Le développeur définit ensuite une requête sur ce tableau d'entrée, comme s'il s'agissait d'un tableau statique, pour calculer un tableau de résultat final qui sera écrit dans un système de sortie compatible. Spark convertit automatiquement cette requête de type batch en un plan d'exécution en continu. C'est ce qu'on appelle l'incrémentation : Spark

détermine quel état doit être maintenu pour mettre à jour le résultat à chaque fois qu'un enregistrement arrive. Enfin, les développeurs spécifient des déclencheurs pour contrôler le moment de la mise à jour des résultats. Chaque fois qu'un déclencheur s'active, Spark vérifie s'il y a de nouvelles données (nouvelle ligne dans le tableau), et met à jour le résultat de manière incrémentale.

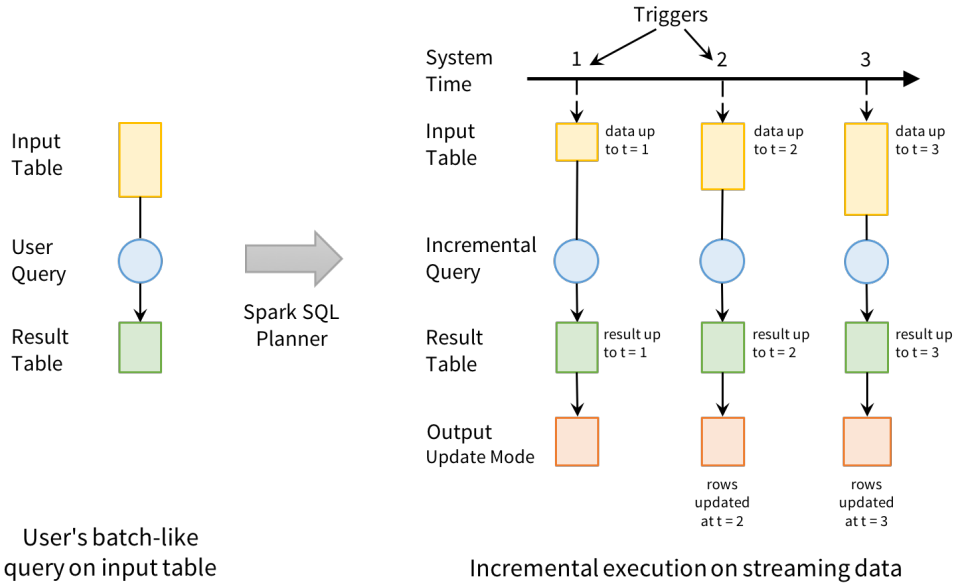

FIGURE 2.5 – Fonctionnement de Spark Structured Streaming

L'"output" est définie comme ce qui est écrit sur le stockage externe. La sortie peut être définie dans trois modes différents :

- **Complete Mode** - A chaque déclenchement on ré-écrit l'ensemble du tableau des résultats sur le stockage externe. Il appartient au logiciel qui gère le stockage de décider comment gérer l'écriture de l'ensemble de la table.
- **Append Mode** - Seules les nouvelles lignes ajoutées dans le tableau des résultats depuis le dernier déclenchement seront écrites dans le stockage externe. Ceci n'est applicable qu'aux requêtes pour lesquelles les lignes existantes dans le tableau des résultats ne sont pas censées changer.
- **Update Mode** - Seules les lignes qui ont été mises à jour dans le tableau des résultats depuis le dernier déclenchement seront écrites dans le stockage externe. Notez que ce mode est différent du mode "complete" dans la mesure où il ne produit que les lignes qui ont été modifiées depuis le dernier déclencheur. Si la requête ne contient pas d'agrégations, elle sera équivalente au mode "append".

2.3.3 Traitement des données en retard et des données temporelles

Le timecode est le temps intégré dans les données elles-mêmes. Pour de nombreuses applications, vous pouvez utiliser ce timecode. Par exemple, si vous voulez connaître le nombre d'événements générés par les appareils connectés chaque minute, vous voudrez probablement utiliser l'heure à laquelle les données ont été générées (c'est-à-dire l'heure de l'événement dans les données), plutôt que l'heure à laquelle Spark les reçoit. Le timecode permet aux opérations sur une fenêtre de temps donnée (par exemple, le nombre d'évé-

nements par minute) de n'être qu'un type spécial de regroupement et d'agrégation sur la colonne de l'heure de l'événement. Par conséquent, de telles requêtes d'agrégation basées sur des fenêtres temporelles d'événements peuvent être définies de manière cohérente à la fois sur un ensemble de données statiques (par exemple, à partir de journaux d'événements de dispositifs collectés) ainsi que sur un flux de données, ce qui facilite grandement la vie de l'utilisateur.

De plus, ce modèle traite naturellement les données qui sont arrivées plus tard que prévu en fonction de leur heure d'apparition. Étant donné que Spark met à jour le tableau des résultats, il a le contrôle total de la mise à jour des anciens agrégats lorsqu'il y a des données en retard, ainsi que du nettoyage des anciens agrégats pour limiter la taille des données sur les états intermédiaires.

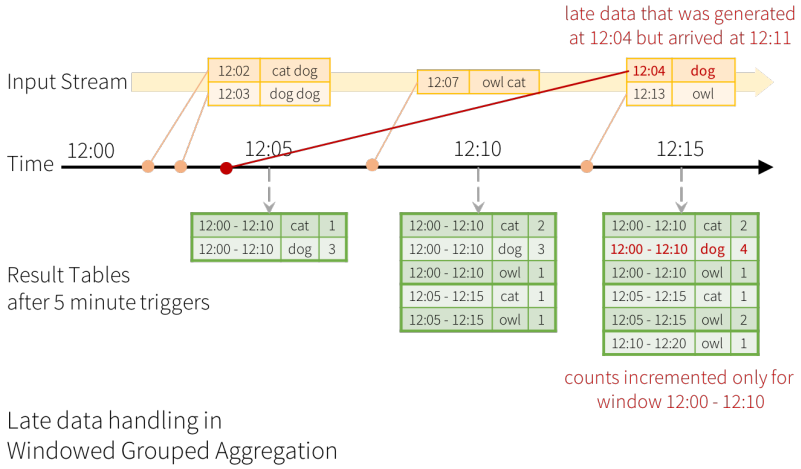

FIGURE 2.6 – Traitement des données en retard

2.4 Présentation du cluster

Le cluster sur lequel nous travaillons est composé de 16 machines disposant chacune de deux processeurs de 4 coeurs physiques (8 coeurs logiques) et de 32 GB de RAM. Le processeur sur lequel nous avons travaillé est un E5-2637 v3 « haswell » de 2014. Nous avons utilisé Spark sur l'ensemble des machines dans une configuration avec 1 "master" et 15 workers. Par défaut, lors de la soumission d'une tâche Spark au clutser, chaque worker dispose d'un coeur physique et de 1 GB de mémoire vive. Les applications Spark s'exécutent comme des ensembles de processus indépendants sur un cluster, coordonnés par un unique SparkContext de notre programme principal (appelé programme pilote) [24]. L'architecture est décrite sur le schéma ci-dessous, le master étant le "Driver Program" :

Il y a plusieurs choses utiles à savoir sur cette architecture :

1. Chaque application a ses propres "spark executors", qui restent en place pendant toute la durée de l'application et exécutent des tâches dans plusieurs threads. Cela présente l'avantage d'isoler les applications les unes des autres, tant du côté de la planification (chaque pilote planifie ses propres tâches) que du côté de l'exécution (les tâches de différentes applications s'exécutent dans des machines virtuelles différentes). Toutefois, cela signifie également que les données ne peuvent pas être partagées entre différentes applications Spark (instances de SparkContext) sans être écrites dans un système de stockage externe.

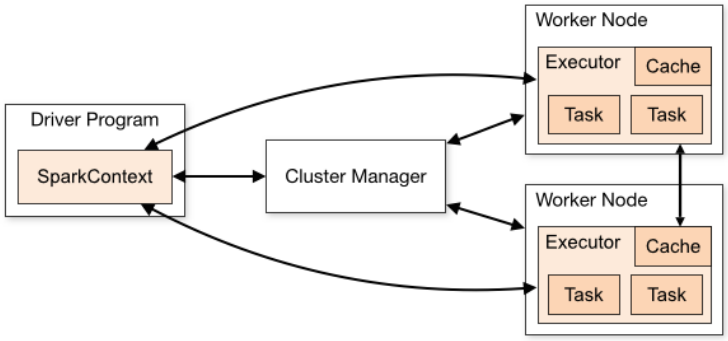

FIGURE 2.7 – Fonctionnement de Spark sur un cluster

2. Spark est agnostique à l'égard du responsable du cluster sous-jacent. Tant qu'il peut acquérir des processus d'exécution, et que ceux-ci communiquent entre eux, il est relativement facile de l'exécuter même sur un gestionnaire de cluster qui prend également en charge d'autres applications (par exemple Mesos/YARN).
3. Comme le Spark master planifie des tâches sur un cluster, toutes les machines doivent être sur le même réseau local afin que le réseau et la connexion internet ne ralentisse pas l'exécution du programme.

2.5 Test de performances sur le cluster Sarah

Spark Structured Streaming fournit les outils nécessaires pour effectuer des mesures de performances basiques de l'installation. D'une part on utilise la source de type *Rate* pour notre application. Cette source génère une suite de nombre avec un débit donné. D'autre part nous nous connectons à l'interface *Spark UI* qui permet de visualiser les tâches effectuées par Spark. Les deux figures suivantes représentent les jobs exécutés par Spark Structured Streaming dans le temps pour deux débits de messages différents. Chaque rectangle représente un job qui correspond au traitement des messages accumulés pendant l'exécution du job précédent.

FIGURE 2.8 – Timeline d'exécution des jobs Spark avec un débit de 100 000 msg/s

FIGURE 2.9 – Timeline d'exécution des jobs Spark avec un débit de 100 000 000 msg/s

Dans le premier cas, on observe que l'installation arrive à suivre sans problème et à traiter les messages "en temps réels" ou plutôt par batch mais sans prendre de retard, c'est-à-dire que la durée entre le moment où le message arrive et celui où Spark donne le résultat reste à peu près stable.

Dans le second cas, au contraire le débit de message devient trop important, Spark ne peut plus suivre et prend du retard. Chaque job est de plus en plus long car plus un job met du temps à s'exécuter, plus le nombre de messages reçus pendant cette période est conséquent et donc plus le job suivant sera long. Cet effet cascade entraîne l'accumulation du retard et c'est ce qu'on observe sur le graphe d'exécution.

Spark Structured Streaming peut fonctionner selon deux modes différents : le premier, utilisé ci-dessus, déclenche un nouveau job dès que le précédent est terminé, le second permet de déclencher les jobs à intervalle régulier. Bien que le premier soit très adapté au traitement de données en temps réel, c'est le second que nous allons utiliser pour effectuer des mesures de performances. En effet, les temps étant réguliers tout comme la quantité de données traitées, les mesures sont plus simples et plus précises. Nous avons effectué deux séries de mesures sur le cluster Sarah présenté ici. La première répartissait le travail sur les 15 machines du cluster en utilisant un seul coeur par machine et la seconde, toujours sur les 15 machines, utilisait quatre coeurs par machine. Nous mesurons le temps de traitement d'un batch de messages correspondant à l'accumulation des messages reçus pendant 30 secondes. Les résultats sont présentés en échelle logarithmique.

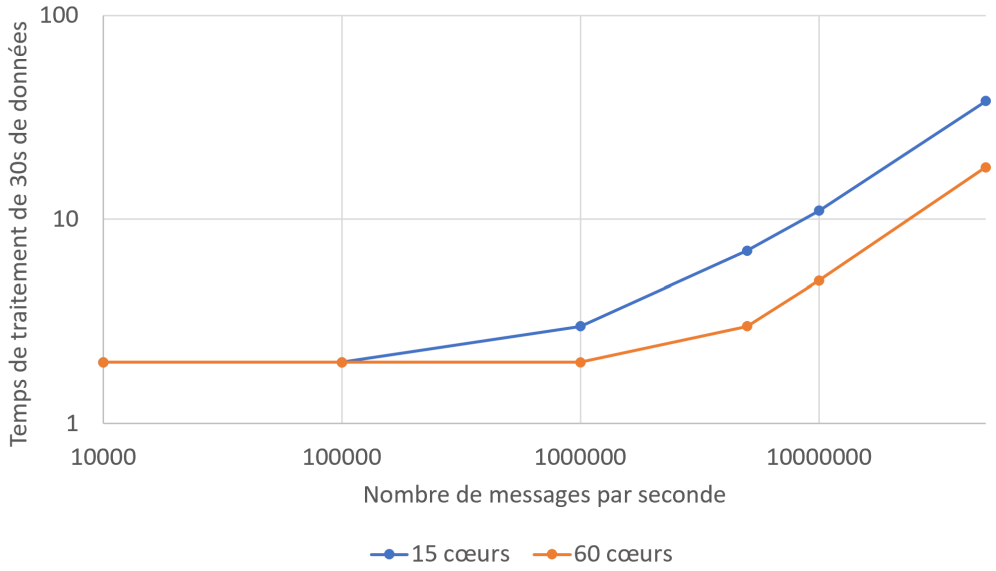

FIGURE 2.10 – Temps de traitement d'un batch de 30s en fonction du débit de message

On observe qu'il existe un temps d'exécution minimum quelque soit la quantité de données traitée. Celui-ci est d'environ 2s et reste le même lorsqu'on fait varier le nombre de coeurs utilisés. Une fois que le débit de messages devient plus conséquent le temps d'exécution, après la transition, semble augmenter de manière linéaire avec le débit de messages. Cela est cohérent si l'on considère le moment où les machines tournent à plein régime, chaque message augmente le temps d'exécution et l'on s'attend bien à observer une augmentation linéaire avec le nombre de message. Considérons maintenant la variation du nombre de coeurs utilisés. On remarque que le palier minimum reste le même et que lorsque le débit de message augmente les courbes sont similaires mais les performances sont logiquement meilleures pour quatre coeurs par machine. Néanmoins le temps d'exécution n'est que deux fois plus faible alors qu'on pourrait s'attendre à ce qu'il soit divisé par quatre.

Chapitre 3

Spark Streaming et Spark Structured Streaming sur un flux de données Twitter

3.1 Objectifs de l'expérimentation

Pour mieux appréhender la construction d'une application de traitement de données en streaming grâce à Spark, il est important de le faire sur un cas pratique. Nous avons choisi dans un premier temps un exemple certes classique mais qui donne justement une bonne approche du problème. Il s'agit de se connecter à l'API twitter pour récupérer des tweets en temps réels et distribuer leur traitement avec Spark. Pour la partie traitement, on pourrait imaginer toute sorte d'analyse, nous nous sommes concentrés sur l'extraction de mots-dièses. C'est-à-dire que le programme doit se charger d'analyser le texte et d'y récupérer les mots-dièses dans le but d'établir une liste de tendance de tweets. L'application prends ainsi un flux de tweets en entrée et donne un compte des sujets les plus abordés en sortie.

3.2 Architecture logicielle

Le développement de cet exemple c'est fait en deux temps car nous avons commencé en utilisant Spark Streaming, c'est-à-dire la version utilisant les RDDS. C'est au cours de ce développement que nous nous sommes rendus compte que Spark Structured Streaming était plus adapté. Nous avons donc décidé de basculer sur cette version plus récente et plus performante utilisant les datasets. L'architecture se découpe en deux grandes parties :

— Un serveur python qui se charge de se connecter à l'API Twitter, récupérer les tweets, puis les transmettre à travers une socket vers l'application Spark.
— Une application Spark qui récupère les tweets par socket et effectue le traitement de chacun des tweets, ie l'extraction de hashtags.

FIGURE 3.1 – Architecture logicielle du traitement d'un flux de tweets

3.3 Programmation

Comme mentionné plus haut, nous avons écrit deux versions de l'application, l'une en Spark Streaming et l'autre avec Spark Structured Streaming. Tous nos codes sont écrits en python qui est parfaitement pris en charge par Spark. Dans les deux cas, les résultats sont au format Spark SQL, c'est donc aussi un avantage de Spark Structured Streaming qui travaille directement avec Spark SQL. On peut aussi relever que la syntaxe est plus claire et plus concise pour le second ce qui est particulièrement important pour un outil comme Spark qui se veut de haut niveau.

Pour aller plus loin sur l'avantage de Spark Structured Streaming et le choix de l'utiliser, on peut citer le fait que Structured Streaming est plus récent et donc, au-delà des gains de performance apportés par l'optimisation des dataframes, il sera maintenu plus longtemps. Un autre point important est que Structure Streaming est plus adapté aux traitements en temps réel. En effet, Spark Streaming utilise un système de batchs à intervalles réguliers qui a été en partie repensé pour Structured Streaming : les données sont accumulées par batchs mais sur des durées variables ce qui permet de lancer un job dès que le précédent est terminé. Il y a deux intérêts à cela : les machines sont utilisées au maximum de leurs capacités et les données sont traitées le plus tôt possible. Structured Streaming bénéficie aussi de la capacité à traiter des données qui arrivent en retard et offre une meilleure tolérance aux fautes.

Les codes qui suivent permettent notamment de se rendre compte de la différence de syntaxe et de remarquer l'avantage de travailler directement avec les fonctions Spark SQL dans le cas de Structured Streaming plutôt que d'utiliser des RDDs puis de passer à Spark SQL dans le cas de Spark Streaming.

3.3.1 Spark Streaming

```
1  import os
2  import sys
3
4  from pyspark import SparkConf, SparkContext
5  from pyspark.sql import Row, SQLContext
6  from pyspark.streaming import StreamingContext
7
8  conf = SparkConf()
9  conf.setAppName("TwitterStreamApp")
10 sc = SparkContext(conf=conf)
11 sc.setLogLevel("ERROR")
12 ssc = StreamingContext(sc, 2)
13 ssc.checkpoint(os.getcwd())
14
15 dataStream = ssc.socketTextStream("sparkdesk.ic.metz.supelec.fr", 9009)
16
17 def aggregate_tags_count(new_values, total_sum):
18     return sum(new_values) + (total_sum or 0)
19
20 # Retourne le contexte SQL en s'assurant qu'un seul contexte existe
21 def get_sql_context_instance(spark_context):
22     # Si le contexte SQL n'existe pas, il est cree
23     if ('sqlContextSingletonInstance' not in globals()):
24         globals()['sqlContextSingletonInstance'] = SQLContext(spark_context
       )
25     return globals()['sqlContextSingletonInstance']
26
27 def process_rdd(time, rdd):
28     print("----------- %s -----------" % str(time))
29     try:
30         # Recupere le singleton spark sql depuis le contexte actuel
```

```
        sql_context = get_sql_context_instance(rdd.context)
        # Convertit le RDD vers un Row RDD
        row_rdd = rdd.map(lambda w: Row(hashtag=w[0], hashtag_count=w[1]))
        # Cree un Dataframe depuis le Row RDD
        hashtags_df = sql_context.createDataFrame(row_rdd)
        # Souscrit le dataframe comme une table
        hashtags_df.registerTempTable("hashtags")
        # Recupere et affiche le top 10 des hashtags depuis la table en utilisant SQL
        hashtag_counts_df = sql_context.sql(
            "select hashtag, hashtag_count from hashtags order by hashtag_count desc limit 10")
        hashtag_counts_df.show()
        # Appelle cette methode pour preparer Dataframe du top 10 et l'envoyer
    except:
        e = sys.exc_info()[0]
        print("Error: %s" % e)

# Separe chaque tweet en mots
words = dataStream.flatMap(lambda line: line.split(" "))
# Filtre les mots pour recuperer seulement les hashtags, puis transforme chaque hashtag en une paire (hashtag, 1)
hashtags = words.filter(lambda w: '#' in w).map(lambda x: (x, 1))
# Ajoute le compte de chaque hashtag a son precedent
tags_totals = hashtags.updateStateByKey(aggregate_tags_count)
# Effectue le traitement pour chaque RDD dans chaque intervalle
tags_totals.foreachRDD(process_rdd)
# Demarre le stream
ssc.start()
# Attend la fin du stream
ssc.awaitTermination()
```

Listing 3.1 – Code python de l'application Spark (version Spark Streaming)

3.3.2 Spark Structured Streaming

```
from pyspark.sql import SparkSession
from pyspark.sql.functions import explode, split, col

# Creation de l'environnement Spark
spark = SparkSession \
    .builder \
    .appName("TopHashatgs") \
    .getOrCreate()

spark.sparkContext.setLogLevel("ERROR")

# Connexion par socket au serveur python
lines = spark \
    .readStream \
    .format("socket") \
    .option("host", "sparkdesk.ic.metz.supelec.fr") \
    .option("port", 9009) \
    .load()

# Traitement des tweets : extraction des hashtags
# La methode explode est equivalente a un flatMap :
# on applique a chaque ligne la methode split qui separe les mots
# et on retourne un unique tableau de mots (au lieu d'un tableau de tableaux)
hastags = lines.select(
    explode(
```

```
        split(lines.value, " ")
    ).alias("hashtag") # la methode alias permet de nommer le resultat pour
     le reutiliser ensuite
).where(col("hashtag").startswith("#"))

hashtagCounts = hastags.groupBy("hashtag").count()

# Execution de la requete
query = hashtagCounts \
    .writeStream \ # Indique que l'on est en mode streaming
    .outputMode("complete") \ # cf 2.3.2
    .format("console") \ # Affiche le resultat dans la console
    .start()

query.awaitTermination()
```

Listing 3.2 – Code python de l'application Spark (version Structured Streaming)

3.4 Mesures de performances

Bien qu'un grand nombre de tweets soient générés en continu, l'utilisation gratuite de l'API ne permet que d'en recevoir un nombre limité. Dans tous les cas de figure l'application Spark a réussi a suivre le rythme sans aucun problème et le nombre de tweets reçus est trop faible pour pouvoir faire de réelles mesures de performances, d'autant qu'il est impossible de choisir le débit. Nous avons donc décidé que des mesures de performance n'étaient pas perspicaces sur cette application et elles seront bien plus intéressantes sur la partie suivante. Nous avons bien écrit un faux serveur de tweets pour générer des tweets à un rythme choisit mais le système est très proche de la source Rate que nous avons utilisée dans la partie précédente et donc l'analyse reste la même.

Chapitre 4

Spark Structured Streaming et Graphframe sur l'exemple d'un graphe client-produit Amazon

4.1 Objectifs de l'expérimentation

Comme nous disposions d'un cluster de machines performant, nous souhaitions tester ses capacités et ses limites sur un cas pratique. Cependant, il est très difficile d'obtenir des flux de données important et en temps réel sur internet. Nous avons donc décidé d'utiliser une base de donnée existante et de simuler l'envoi de ces données en temps réel. La base de données que nous avons utilisé est fournie par Amazon. Elle contient des avis de consommateurs sur des produits Amazon publiés sur le site d'Amazon entre 1995 et 2015 et en plusieurs langues. Le dataset contient plus de 130 millions de commentaires. Nous utiliserons cette base de données afin de venir alimenter en temps réel un graphe client-produit. Le fichiers sont présentés au format TSV (Tabulation-separated values, comme un CSV mais avec des tabulations comme séparateur) avec les colonnes suivantes [Datb] [Data] :

marketplace	2 letter country code of the marketplace where the review was written.
customer_id	Random identifier that can be used to aggregate reviews written by a single author.
review_id	The unique ID of the review.
product_id	The unique Product ID the review pertains to. In the multilingual dataset the reviews. For the same product in different countries can be grouped by the same product_id.
product_parent	Random identifier that can be used to aggregate reviews for the same product.
product_title	Title of the product.
product_category	Broad product category that can be used to group reviews (also used to group the dataset into coherent parts).
star_rating	The 1-5 star rating of the review.
helpful_votes	Number of helpful votes.
total_votes	Number of total votes the review received.
vine	Review was written as part of the Vine program.
verified_purchase	The review is on a verified purchase.
review_headline	The title of the review.
review_body	The review text.
review_date	The date the review was written.

La base de données est aussi disponible au format `parquet` qui est un format développé par Apache optimisé pour le Big Data.

L'objectif de ce cas pratique sera d'alimenter en temps réel un graphe client-produit pour ensuite effectuer des opérations sur ce graphe comme le calcul des composants connectés à l'aide de la bibliothèque GraphFrames qui est une sur-couche à la bibliothèque de graphes de Spark et qui contient de nombreux algorithmes d'analyse de graphe avec des DataFrames [070]. Les algorithmes disponibles dans GraphFrames ne sont pas conçus pour le temps réel, il en est de même pour GraphX. En effet ces algorithmes sont calculés sur un graphe entier et ne peuvent pas tirer avantage de petites mises à jour.

Dans la littérature, les chercheurs se concentrent principalement sur l'optimisation des algorithmes qui sont utilisés de façons ponctuelles plutôt que sur la possibilité de faire des mises à jours partielles et d'intégrer de nouvelles données. Ces calculs ne sont donc pas réalisables en temps réels mais on peut penser à une entreprise qui utilise ces algorithmes lorsque son réseau est moins chargé (la nuit par exemple) [AZ17] [Sha18].

4.2 Architecture logicielle

4.2.1 Présentation de GraphFrames

GraphFrames est un package pour Apache Spark qui aide au traitement de graphes basés sur des DataFrames. Il fournit des API de haut niveau en Scala, Java et Python. Il vise à fournir à la fois les fonctionnalités de GraphX et des fonctionnalités étendues en tirant parti des Spark DataFrames. Cette fonctionnalité étendue comprend la recherche de motifs, la sérialisation basée sur les DataFrames et des requêtes intuitives sur les graphes .

Si vous connaisez GraphX, GraphFrames est très simple à utiliser. En effet, il suffit de fournir à GraphFrames deux DataFrames : l'un contenant les noeuds du graphe et l'autre contenant les arêtes du graphe. Les algorithmes disponibles sont les suivants :

— Algorithme de parcours en largeur (BFS)

— Algorithme d'étiquetage des composantes connexes

— Algorithme sur la propagation des labels (LPA)
— PageRank
— Plus court chemin
— Comptage des triangles

4.2.2 Description de l'architecture

Notre application se découpe en trois parties :

— Un premier programme va simuler le streaming et va venir alimenter un dossier sur HDFS avec des lignes de la base de donnée. Ainsi on peut simuler des clients qui achètent des produits.
— Un second qui est le programme Spark Structured Streaming. Ce programme va être déclenché à chaque nouvel ajout de fichier dans le HDFS par le premier et va construire au fur et à mesure deux DataFrames : l'un contenant les noeuds de notre graphe et le second contenant les arêtes. Ces DataFrames sont ensuite stockés dans le HDFS par le même programme sous formes de deux fichiers CSV.
— Le troisième programme va calculer l'algorithme des composantes connexes et sera déclenché par l'utilisateur. Le résultat est ensuite stocké dans le HDFS.

L'élément central est donc ici le système de fichier HDFS qui va nous permettre de stocker de gros fichiers avec une réplication entre différents espaces de stockage afin d'éviter la perte de données et d'accélérer l'accès à ces dernières. La structure de l'application est présentée ci-après.

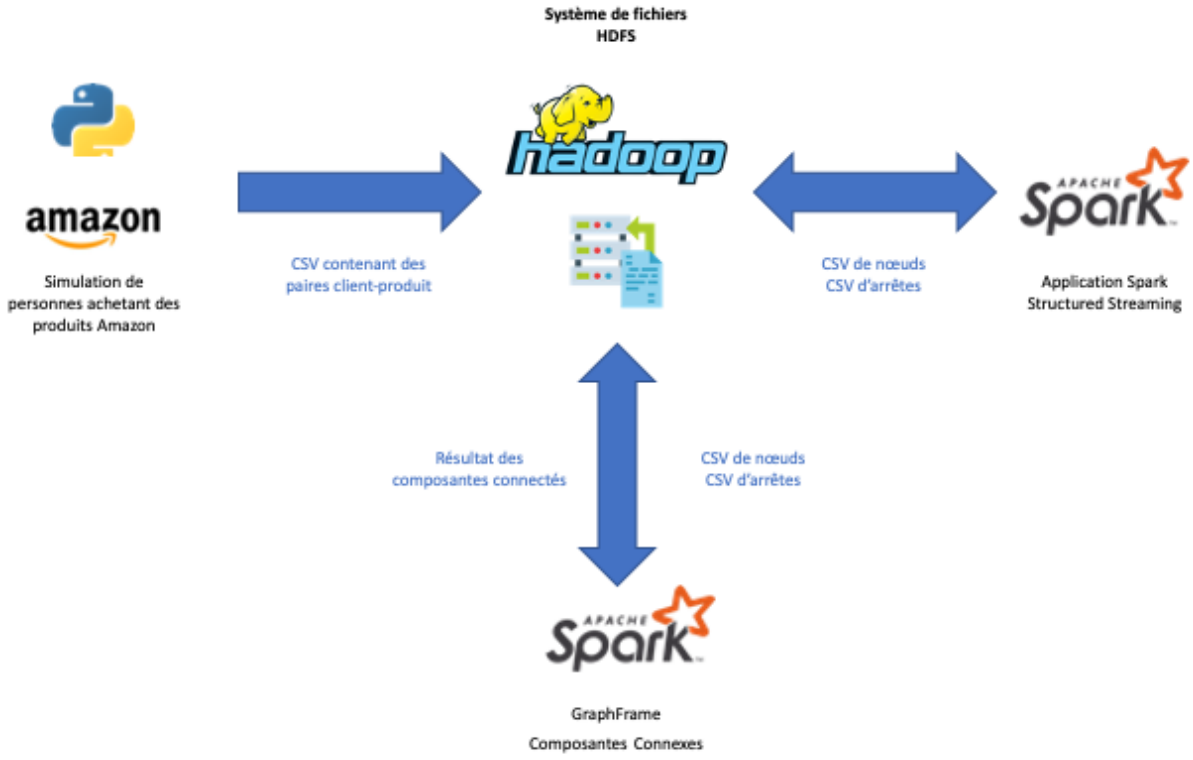

FIGURE 4.1 – Architecture logicielle du traitement d'un graphe client-produit Amazon

4.3 Programmation

Comme indiqué précédemment, l'application se présente sous forme de trois programmes.

4.3.1 Simulation du streaming

Dans ce premier programme nous simulons l'envoie de données sur notre système de stockage HDFS. Nous avons préalablement téléchargé des fichiers TSV contenant des commentaires de clients sur des produits Amazon

```
import time
import os

# Chemins d'acces aux dossiers sur le HDFS
filepath = '../amazon_reviews_multilingual_FR_v1_00.tsv'
hdfspath = 'hdfs://sar01:9000/spark_project_2020'
outpath = hdfspath + '/review_stream/'

# On nettoie d'abord le HDFS
os.system('hdfs dfs -rm {}/*'.format(outpath))
os.system('hdfs dfs -rmr {}/streaming_output/*'.format(hdfspath))
os.system('hdfs dfs -rmr {}/checkpoint/vertices/*'.format(hdfspath))
os.system('hdfs dfs -rmr {}/checkpoint/edges/*'.format(hdfspath))

with open(filepath) as f:
    headers = f.readline()
    k = 0
    # On genere 1000 commentaires a partir du dataset existant
    while True:
        lines = f.readlines(1000)
        lines.insert(0, headers)
        content = ('\n'.join(lines)).replace('"', '\"')
        outfilepath = outpath + 'reviews_{}.tsv'.format(k)
        os.system('echo "{content}" | hdfs dfs -put - {path}' \
            .format(content='\n'.join(lines), path=outfilepath))
        k += 1
        print('New file: ' + outfilepath)
        time.sleep(10) # Fichiers generes toutes les 10 secondes
```

Listing 4.1 – Code Python simulant le streaming de commentaires produits Amazon

4.3.2 Traitement des données en streaming

Le second programme va traiter les données en temps réel. Il construira deux DataFrames : un de noeuds de graphe et un d'arêtes. Ces DataFrames seront ensuite stockés dans le HDFS. Ces DataFrames pourront ensuite être lu par la dernière application qui applique un algorithme sur le graphe. Ce qui est important au début d'une application Spark Structured Streaming est de bien spécifier la structure des données qu'il reçoit. L'application pourra ainsi émettre une alerte si les données arrivent dans un format différent.
Notons qu'il est nécessaire d'indiquer un dossier pour le checkpointing afin que Spark puisse stocker les états intermédiares et y revenir en cas d'erreur dans le système..

```
from pyspark.sql import *
from pyspark.sql.functions import lit
from pyspark.sql.types import StructType

# Chemin d'acces au HDFS
PATH = "hdfs://sar01:9000/spark_project_2020/review_stream/"

# Structure du document en entree
schema = StructType() \
    .add('marketplace', 'string') \
    .add('customer_id', 'integer') \
```

```
    .add('review_id', 'string') \
    .add('product_id', 'string') \
    .add('product_parent', 'integer') \
    .add('product_title', 'string') \
    .add('product_catgory', 'string') \
    .add('star_rating', 'integer') \
    .add('helpful_votes', 'integer') \
    .add('total_votes', 'integer') \
    .add('vine', 'string') \
    .add('verified_purchase', 'string') \
    .add('review_headline', 'string') \
    .add('review_body', 'string') \
    .add('review_date', 'timestamp')

# Creation de la Spark Session
spark = SparkSession \
    .builder \
    .master("spark://sar01:7077") \
    .appName("AmazonReviewStreaming") \
    .getOrCreate()

spark.sparkContext.setLogLevel("ERROR")

# Checkpointing permettant la recuperation en cas d'erreur
spark.sparkContext.setCheckpointDir(
    'hdfs://sar01:9000/spark_project_2020/checkpoint')

# Entree du streaming
df = spark.readStream \
    .csv(path=PATH, schema=schema, sep='\t', header=True)

# Filtrage des produits et clients
products = df \
    .select("product_id") \
    .distinct()
products = products.withColumn("type", lit("product"))

customers = df \
    .select("customer_id") \
    .distinct()
customers = customers.withColumn("type", lit("customer"))

# Construction des DataFrames vertices et edges suivant le format voulu par
    GraphFrames
vertices = products.union(customers).withColumnRenamed('product_id', 'id')
edges = df.select('customer_id', 'product_id')
edges = edges.withColumnRenamed('customer_id', 'src').withColumnRenamed(
    'product_id', 'dst')
edges = edges.withColumn("relation", lit("bought"))

# Ecriture des DataFrames dans le HDFS avec checkpointing
query = vertices \
    .writeStream \
    .outputMode("append") \
    .format("csv") \
    .option("path",
            "hdfs://sar01:9000/spark_project_2020/streaming_output/vertices
    .csv") \
    .option("checkpointLocation",
            "hdfs://sar01:9000/spark_project_2020/checkpoint/vertices") \
    .start()

query_2 = edges \
```

```
    .writeStream \
    .outputMode("append") \
    .format("csv") \
    .option("path",
            "hdfs://sar01:9000/spark_project_2020/streaming_output/edges.
    csv") \
    .option("checkpointLocation",
            "hdfs://sar01:9000/spark_project_2020/checkpoint/edges") \
    .start()

# Attente de la fin de la requete de streaming
query.awaitTermination()
```

Listing 4.2 – Application Spark Structured Streaming traitant les commentaires sur les produits Amazon

4.3.3 Traitement du graphe avec GraphFrames

Le dernier programme est à exécuter quand on le désire. Dans notre exemple nous allons appliquer l'algorithme des composantes connexes sur notre graphe. Comme nous pouvons le voir ci-dessous, le programme prend en entrée un DataFrame comprenant les noeuds et un autre comprenant les arêtes du graphe. Le programme va nous donner en résultat chaque noeud et la communauté à laquelle il appartient. Notons ici qu'il est obligatoire d'indiquer un dossier pour le checkpointing car la bibliothèque GraphFrames le demande (le traitement nécessitant beaucoup de données intermédiaires).

```

import time

from graphframes import *
from pyspark.sql import *
from pyspark.sql.functions import lit

# Chemin d'acces au HDFS
PATH = 'hdfs://sar01:9000/spark_project_2020/amazon_review_dataset/'

# Creation de la Spark Session
spark = SparkSession.builder \
    .master("spark://sar01:7077") \
    .appName("ConnectedComponnents") \
    .getOrCreate()

# Mise en place du checkpointing, obligatoire pour les algorithmes de
    graphe
spark.sparkContext.setCheckpointDir(
    'hdfs://sar01:9000/spark_project_2020/checkpoint')

start = time.time()


df = spark.read.load(PATH, format="csv", sep="\t", inferSchema="true",
                     header="true")

products = df \
    .select("product_id") \
    .distinct()
products = products.withColumn("type", lit("product"))

customers = df.select("customer_id") \
    .distinct()
customers = customers.withColumn("type", lit("customer"))

```

```
vertices = products.union(customers).withColumnRenamed('product_id', 'id')
edges = df.select('customer_id', 'product_id')
edges = edges.withColumnRenamed('customer_id', 'src').withColumnRenamed(
    'product_id', 'dst')
edges = edges.withColumn("relation", lit("bought"))

# Creation de l'objet graphe
g = GraphFrame(vertices, edges)

# Algorithme des composantes connectees
result = g.connectedComponents()
result.select("id", "component").orderBy("component").show()

print(time.time() - start)
```

Listing 4.3 – Programme GraphFrame sur le calcul de communautés

4.4 Mesures de performances

Nous avons effectué différentes séries de mesures sur l'exécution de la totalité de l'application, c'est-à-dire de la construction du graphe à partir des données brutes jusqu'à l'extraction des composants connectés. Toujours en utilisant les 15 machines du cluster Sarah, nous avons fait varier le nombre de coeurs utilisés sur chaque machine ainsi que la taille du dataset traité. On précisera que nous étions seuls à utiliser le cluster lorsque les mesures ont été effectuées.

Les trois courbes, correspondant aux trois tailles de dataset, sont relativement similaire. Dans un premier temps on peut observer que le temps d'exécution décroît linéairement en échelle logarithmique de un à quatre coeurs utilisés mais, comme on l'avait remarqué dans le chapitre 1, le gain de performance n'est que double quand il devrait être quadruple. Ensuite, lorsque l'on passe à huit coeurs par machine, on observe encore un gain de performance mais avec une pente encore plus faible. En passant finalement à 16 coeurs, on parle cette fois de coeurs logiques puisque les machines ne possède que huit coeurs physiques, on ne note aucun gain de performance et même une légère augmentation du temps d'exécution pour le dataset le plus petit. On se souviendra donc que l'application est optimisée pour s'exécuter sur tous les coeurs *physiques* des machines.

Si l'on analyse maintenant plus en détail l'influence de la taille du dataset traité, on pourra remarquer que les pentes sont d'autant plus importantes que le dataset est conséquent. Cela signifie que répartir la charge de travail permet des gains de performances de plus en plus intéressant quand la quantité de données augmente.

La seconde signification est que le temps d'exécution semble tendre vers un minimum autour de la minute quelque soit la taille du dataset. Nous n'avons pas été en mesure de confirmer ce dernier point car lors des mesures sur un dataset de taille plus petite, nous avons fait face à un fonctionnement particulier de Spark (il y a peut-être un paramètre pour contrôler cela) : l'exécution ne se faisait plus que sur une seul machine et donc le temps d'éxécution explosait. Dans le cas de ce dataset, nous avons observé ce phénomène à chaque fois et nous avons également eu ce problème avec les datasets plus gros mais avec des occurrences beaucoup plus rare.

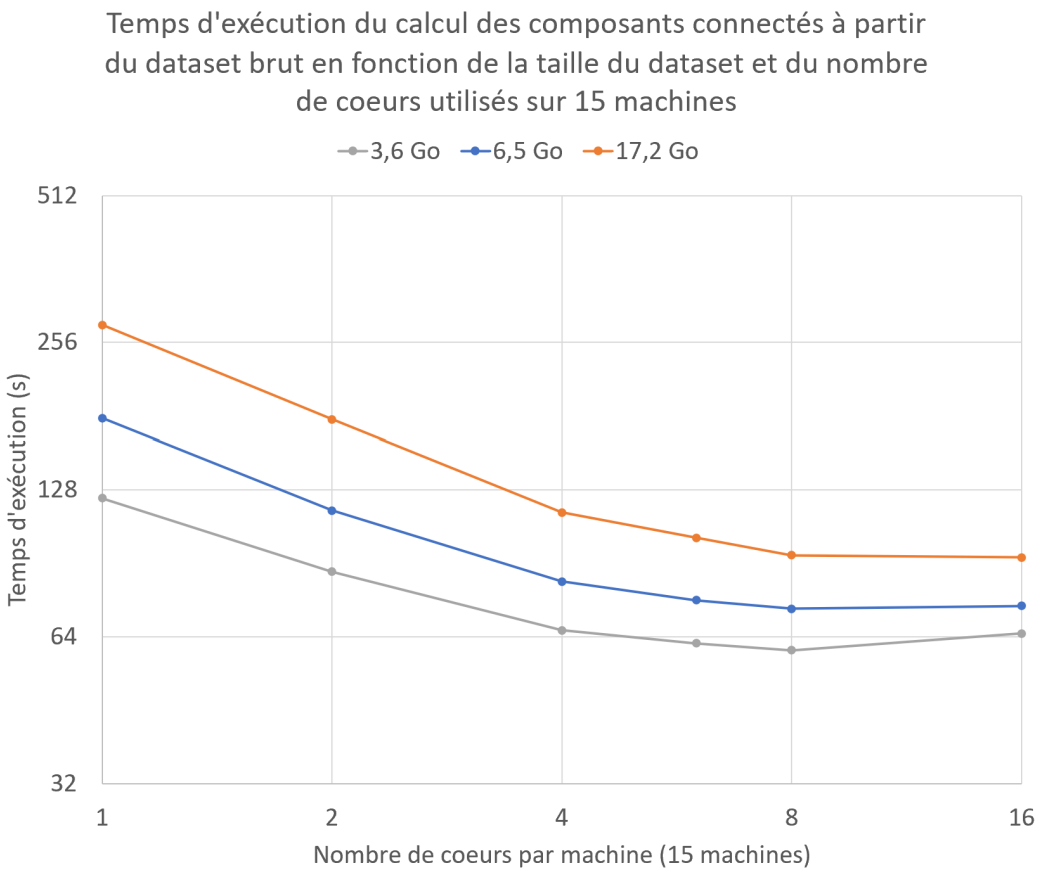

FIGURE 4.2 – Temps d'exécution du calcul des composants connectés à partir des données brutes en fonction de la taille du dataset et du nombre de coeurs utilisés sur chacune des 15 machines

Chapitre 5

Mode d'emploi et principaux problèmes rencontrés

5.1 Installation et configuration

5.1.1 Requirements

Nous donnons les versions utilisés à titre indicatifs, il n'est pas exclu que tout fonctionne avec d'autres versions mais aucun test n'a été effectué.

Vous devez disposer des programmes suivants sur votre machine ou cluster :

— python (version 2.7.12)
— java (version 9-internal)
— hadoop (version 2.7.7)
— La bibliothèque python `pyspark` (version 2.4.0)
— La bibliothèque python `tweepy`
— La bibliothèque Spark `graphframes`
— L'interface `aws` en ligne de commandes

Si vous avez accès au cluster de CentraleSupélec Metz, connectez vous en ssh sur la machine *sar01* du cluster Sarah. Sinon remplacez l'adresse par celle de votre machine dans les commandes suivantes et utilisez les ports correspondants.

5.1.2 Configuration ssh

La configuration ci-dessous fonctionne sur un terminal Unix.

Compte tenu de la configuration du réseau à CentraleSupelec, il convient d'utiliser un fichier de configuration ssh situé ici : `/.ssh/config`. Ce fichier nécessite préalablement d'avoir des clés de chiffrement public et privé .

Le fichier de configuration est le suivant :

```
Host *
        AddKeysToAgent yes
Host supelec
        User <dupont_nic>
        HostName ghome.metz.supelec.frnom
Host sparkdesk
        User cpupro_spark
        HostName sparkdesk.ic.metz.supelec.fr
        ProxyJump supelec
Host sar01
        User cpupro_spark
        HostName sar01.ic.metz.supelec.fr
        ProxyJump supelec
        LocalForward 4040 localhost:4040
```

Il faut enfin se connecter par ssh une première fois à l'aide du terminal à l'aide de la commande :

```
ssh-copy-id sar01 #ou sparkdesk suivant la machine
```

La connexion à chacune des machine se fera ensuite de manière totalement transparente avec le terminal grâce à la commande `ssh sar01` ou `ssh sparkdesk`

5.1.3 Récupérer les fichiers

Tous les programmes décrits dans ce rapport sont disponibles sur le le git suivant : `https://github.com/hugofriant/spark_project_2020`

A l'aide d'un terminal, clonez le git dans le répertoire de votre choix pour récupérer les fichiers :

```
$ git clone https://github.com/hugofriant/spark_project_2020.git
```

5.2 Exécuter les programmes

5.2.1 Test de performance

Ce programme permet de tester la capacité du cluster à traiter un flux d'informations à haute fréquence. La commande suivante permet de lancer le programme `fake_source_processor.py` suer le cluster Sarah

```
$ spark-submit --master spark://sar01:7077 fake_source_processor.py
```

Vous pouvez ajuster le débit de message et la pente au démarrage aux lignes 14 et 15 du fichier python :

```
14      .option("rowsPerSecond", 1000) \
15      .option("rampUpTime", 5) \
```

De plus l'interface *Spark UI* est accessible sur le port 4040.

5.2.2 Flux de tweets

Les programes sont placés dans le dossier `twitter_streaming`. Ils permettent de traiter un flux de tweets en temps réel comme expliqué au chapitre 3.

1. Lancer le serveur `receive_tweets.py` après avoir remplacé les valeurs de `consumer_key`, `consume_secret`, `access_token` et `access_secret` par vos credentials Twitter.

   ```
   $ python receive_tweets.py
   ```

2. Lancer l'application spark `process_tweets_spark_sql.py` après avoir remplacé l'adresse ligne 14 du programme par celle de la machine sur laquelle est lancé le serveur de l'étape 1, si la machine est la même remplacez simplement par `localhost`

   ```
   14      .option("host", "sparkdesk.ic.metz.supelec.fr") \
   ```

   ```
   $ spark-submit --master spark://sar01:7077 process_tweets_spark_sql.
       py
   ```

L'interface *Spark UI* est accessible sur le port 4040 et permet de suivre le traitement des données.

5.2.3 Graphe de reviews Amazon

L'ensemble des datasets de reviews est disponible à l'adresse suivante : `https://s3.amazonaws.com/amazon-reviews-pds/readme.html`

Les commandes suivantes permettent de peupler le HDFS avec des fichiers de reviews :

```
$ aws s3 cp s3://amazon-reviews-pds/tsv/amazon_reviews_us_Camera_v1_00.tsv.
    gz .
$ gzip -d amazon_reviews_us_Camera_v1_00.tsv.gz
$ hdfs dfs -put amazon_reviews_us_Camera_v1_00.tsv
    hdfs://sar01:9000/spark_project2020/amazon_review_dataset/
```

L'ensemble des programmes suivants sont disponibles dans le dossier `amazon_review_streaming`

1. Le premier programme à lancer simule l'envoi de commentaires Amazon en temps réel. Il faut définir le chemin vers le fichier TSV à traiter et le dossier HDFS de sortie dans le fichier `review_streamer.py`.

```
4 filepath = '../../amazon_reviews_multilingual_FR_v1_00.tsv'
5 hdfspath = 'hdfs://sar01:9000/spark_project_2020'
```

```
$ python review_streamer.py
```

2. Il faut ensuite lancer le programme Spark qui va construire le graphe client-produit en temps réel. Lancer l'application Spark `test_graphframe.py`

```
$ spark-submit --master spark://sar01:7077 streaming_amazon_review.py
```

Le dernier programme permet d'appliquer l'algorithme des composantes connexes sur le graphe construit précédemment.

```
$ spark-submit --master spark://sar01:7077 --packages
    graphframes:graphframes:0.7.0-spark2.4-s_2.11
    spark_connected_components.py
```

Chapitre 6

Bilan et perspectives

Durant ce projet nous avons pu apprendre à utiliser les différents modules d'Apache Spark et notamment les modules de traitement en temps réel de données. Nous avons effectué tous ces développements sur un cluster de 16 ordinateurs disponibles à Supélec sur le campus de Metz. Nous avons notamment réussi à interfacer de manière efficace un système de fichier HDFS, Spark Structured Streaming (le module temps réel de Spark SQL) et une bibliothèque de graphe liée à Spark (GraphFrames). Nous avons donc vu que, sous réserve d'avoir une installation adéquate, il est possible de traiter avec Apache Spark des données afin d'analyser celles-ci en temps réel sur un cluster.

Nous avons travaillé sur deux projets. Le premier consistait à analyser des tweets en temps réel. Malheuresement, la version gratuite de l'API Twitter pour développeurs n'offre qu'un échantillon de tweets. Nous avons donc rapidement remarqué que notre installation était sur-dimensionnée pour cette application et que nous ne pourrions pas tester les limites du cluster.

La deuxième application consistait à construire et analyser un graphe en temps réel. Pour ce faire, nous avons simulé l'envoi en temps réel d'avis sur des produits Amazon sur notre système de fichiers HDFS. Cela nous a permis de construire un graphe entre des clients et des produits disponibles sur le site Amazon. Pour terminer, nous avons analysé les communautés de ce graphe à l'aide d'un algorithme classique des composantes connexes.

Cependant nous avons remarqué que des analyses de graphe ne peuvent pas encore se faire en temps réel avec Spark. En effet, les bibliothèques telles que GraphX ou GraphFrames ne sont pas taillés pour ce type de problématique même si elles restent très performantes pour un traitement unique. Une perspective d'étude peut donc être le traitement de graphes en temps réel.

Table des figures

Listings

Bibliographie

[AZ17] Tariq Abughofa et Farhana Zulkernine. "Towards online graph processing with spark streaming". In : *2017 IEEE International Conference on Big Data (Big Data)* (2017). doi : 10.1109/bigdata.2017.8258245.

[Que18] Gianluca Quercini. *A Big Data Primer*. CentraleSupelec, 2018.

[Sha18] Sonam Sharma. "Building Real-time knowledge in Social Media on Focus Point : An Apache Spark Streaming Implementation". In : *2018 IEEE Punecon* (2018). doi : 10.1109/punecon.2018.8745394.

[070] GraphFrames v 0.7.0. *GraphFrames Overview*. url : https://graphframes.github.io/graphframes/docs/_site/index.html.

[24] Apache Spark v 2.4. *Cluster Mode Overview*. url : https://spark.apache.org/docs/latest/cluster-overview.html.

[Apa] Apache. *Structured Streaming Programming Guide*. url : https://spark.apache.org/docs/latest/structured-streaming-programming-guide.html#handling-event-time-and-late-data.

[Data] AWS Public Datasets. *Amazon Customer Reviews Dataset*. url : https://s3.amazonaws.com/amazon-reviews-pds/readme.html.

[Datb] AWS Public Datasets. *Index of the Amazon Customer Review Dataset*. url : https://s3.amazonaws.com/amazon-reviews-pds/tsv/index.txt.