Ecole Supérieure de Technologie de Meknès

Département Génie Informatique

Filière Génie Informatique

Stage De Fin d'Etudes

Mémoire Intitulé :

Recherche Théorique sur Hadoop et Spark

Présenté par :

EL BOUAYADI Aiman

EL KHABBAZ Mohamed

Soutenu : le ……

Encadré par :

Encadrant :	Pr. My LAHCEN HASNAOUI
Examinateur :	Pr. Abdallah RHATTOY

2019/2020

DEDICASE

***O**n dédie ce modeste travail à :*

***A** nos parents qui nous ont soutenu et encouragé durant ces années d'études.*

***A** nos sœurs et frères, qui ont partagé avec nous tous les moments d'émotion lors de la réalisation de ce travail. Ils nous ont chaleureusement supporté et encouragé tout au long de notre parcours.*

***A** nos familles, nos proches et à ceux qui nous ont donnée de l'amour et de la vivacité.*

***A** tous nos professeurs tout au long de notre parcours scolaire.*

***A** tous nos amis qui nous ont toujours encouragé.*

***A** tous ceux que nous aimons.*

Merci !

EL BOUAYADI Aiman

EL KHABBAZ Mohamed

REMERCIEMENT

***A**vant tout, nous remercions Dieu le Tout Puissant de nous avoir donné le courage, pour réaliser ce travail avec succès.*

***A**u terme de ce travail, nous tenons à exprimer nos profondes gratitudes à notre cher professeur et encadrant Mr. My Lahcen HASNAOUI, enseignant au sein de l'Ecole Supérieure de Technologie (EST-Meknès) pour nous avoir permis de réaliser ce travail riche en nouveauté. Nous le remercions très sincèrement pour son encadrement, son soutien, son aide précieuse et sa confiance sans lesquels ce travail n'aurait pas été ce qu'il est.*

***N**ous adressons aussi nos vifs remerciements aux membres des jurys pour avoir bien voulu examiner et juger ce travail.*

***N**ous ne nous laisserons pas cette occasion passer, sans remercier tous les enseignants et le personnel de l'Ecole Supérieure de Technologie de Meknès (EST-Meknès), et particulièrement ceux du département génie informatique pour leur aide et leurs précieux conseils et pour l'intérêt qu'ils portent à notre formation.*

***E**nfin, nos remerciements à tous ceux qui ont contribué de près ou de loin au bon déroulement de ce projet.*

TABLE DE MATIERES

Listes des figures

Introduction

Depuis des vingtaines d'années, les données n'était pas trop lourd pour les traites qu'avec les SGBD traditionnelles, Mais avec l'avancement de la technologie et la grande utilisation des sites web et des applications Mobiles et les différents logiciels qui stock des données, on a trouvé que nous sommes contre une masse des données importante estimée à près de 3 trillions d'octets (3.10^18) des données.

On estime ainsi qu'en 2016, 90% des données dans le monde ont été créés au cours des deux années précédentes (2014 et 2015). Et selon le rapport IDC (international national Data corporation) la masse totale des données crée et copie par le monde pour 2011 était de 1.8 zettaoctets (10^21) octets.

Donc chaque 5 ans le monde connu un croissement d'un facteur de 9. Cet accroissement des données touche tous les secteurs, tant scientifique et économiques et les réseaux sociaux...

Par conséquence des nouvelles technologies de gestion des données ont été apparu pour résoudre le problème de la masse des données.

Dans ce rapport on va parler du Big Data et ses caractéristiques ainsi que leur domaine d'application, ensuite on va faire une étude pour montrer la différence entre Hadoop et Spark.

1. *Big data*

a) Définition

Mégadonnées, grosses données ou encore données massives ; Ils désignent un ensemble très volumineux des données qu'aucun outil classique de gestion de base des données ou de gestion de l'information ne peut vraiment travailler.

En effet, nous procréons environ 2,5 trillions d'octets des données tous les jours. Ce sont les informations provenant de partout : messages que nous nous envoyons, vidéos que nous publions, informations climatiques, signaux GPS, enregistrements transactionnels d'achats en ligne et bien d'autres encore. Ces données sont nommées Big Data ou volumes massifs des données. Les géants du Web, au premier rang desquels Yahoo (mais aussi Facebook et Google), ont été les tous premiers à déployer ce type de technologie.

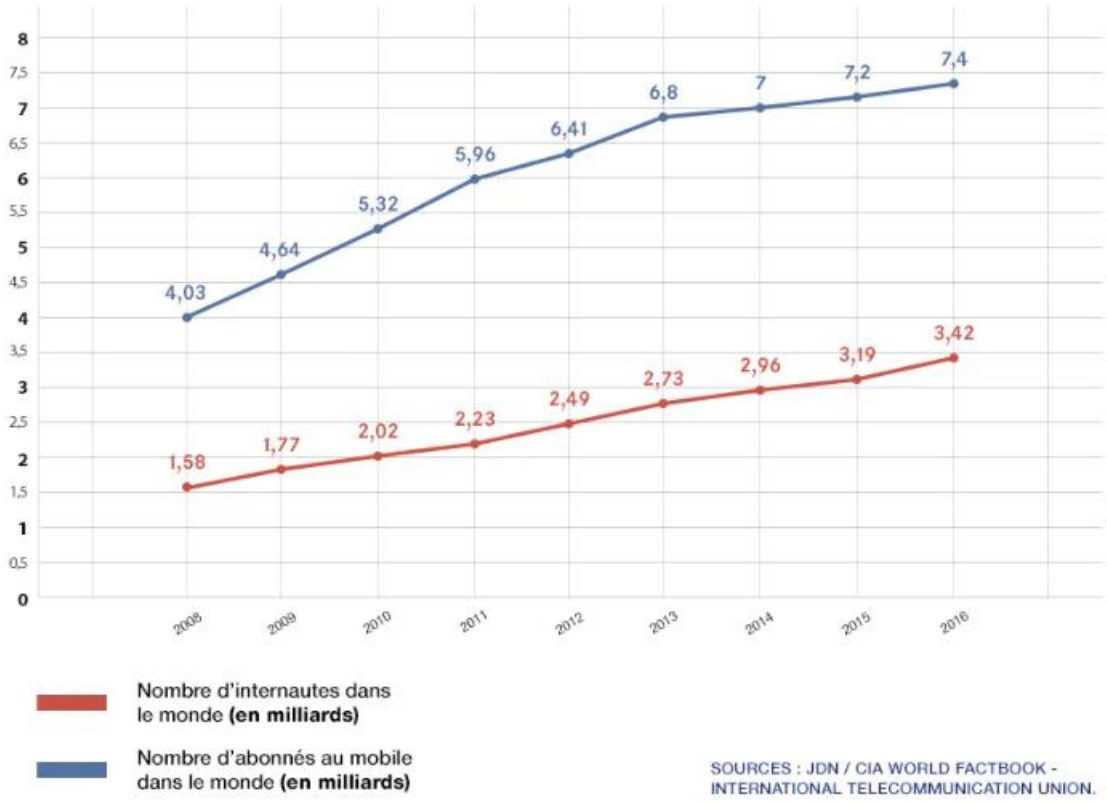

Figure 1 : Utilisation des téléphones Mobile entre 2008 et 2016

b) Objectif

Le Big Data se présente comme une solution dessinée pour permettre à tout le monde d'accéder en temps réel à des bases des données géantes. Il vise à proposer un choix aux solutions classiques de bases des données et d'analyse (plate-forme de Business Intelligence en serveur SQL…).

c) Marché du Big data

Les dépenses mondiales dans le marché de l'analyse de grands volumes des données (BDA) vont connaître **une croissance annuelle de 11,7% jusqu'en 2020**, selon le cabinet de recherches IDC.

Dans son rapport, le monde du Big Data attend des investissements à hauteur de **130 milliards de dollars en 2016**, soit une hausse de 11,3% par rapport à 2015. Et en 2020, les analystes d'IDC prévoient des valeurs supérieures à 203 milliards de dollars.

Ces investissements seront surtout conduits dans le secteur bancaire, la fabrication et la transformation manufacturière, les gouvernements centraux et fédéraux ainsi que les prestataires de services professionnels.

d) Caractéristiques

Figure 2 : Les 5 V du Big Data

- Volume :

  Le volume décrit la quantité des données générées par des entreprises ou des personnes. Le Big Data est généralement associé à cette caractéristique. Les entreprises, tous secteurs d'activité confondus, devront trouver des moyens pour gérer le volume des données en constante augmentation qui est créé quotidiennement.

❖ Vitesse :

La vitesse est la rapidité à laquelle les données affluent. C'est-à-dire la fréquence à laquelle elles sont générées, capturées et partagées.

Avec les nouvelles technologies, les données sont générées toujours plus rapidement et dans des temps beaucoup plus courts. Les entreprises sont obligées de les collecter et de les partager en temps réel mais le cycle de génération des nouvelles données se renouvelle très vite, rendant rapidement les informations obsolètes.

❖ Variété :

Les types des données et leurs sources sont de plus en plus diversifiés supprimant ainsi les structures nettes et faciles à consommer des données classiques. Ces nouveaux types des données incluent un grand nombre de contenus très diversifié : géolocalisation, connexion, mesures, processus, flux, réseaux sociaux, texte, web, images, vidéos, mails, livres, tweets, enregistrements audio…

De par cette diversité qui supprime la structure, l'intégration des données à des feuilles de calcul ou application de base des données est de plus en plus complexe voire impossible.

❖ La véracité

La véracité concerne la fiabilité et la crédibilité des informations collectées. Comme le Big Data permet de collecter un nombre indéfini et plusieurs formes des données, il est difficile de justifier l'authenticité des contenus, si l'on considère les post Twitter avec les abréviations, le langage familier, les hashtags, les coquilles etc. Toutefois, les génies de l'informatique sont en train de développer de nouvelles techniques qui devront permettre de faciliter la gestion de ce type des données notamment par le W3C (World Wide Web Consortium).

❖ La valeur

La notion de valeur correspond au profit qu'on puisse tirer de l'usage du Big Data. Ce sont généralement les entreprises qui commencent à obtenir des avantages incroyables de leurs Big Data. Selon les gestionnaires et les économistes, les entreprises qui ne s'intéressent pas sérieusement au Big Data risquent d'être pénalisées et écartées. Puisque l'outil existe, ne pas s'en servir conduirait à perdre un privilège concurrentiel.

e) Domaine d'application

Figure 3 : Top Domaine d'application du Big Data

❖ **Internet des Object**

L'Internet que nous connaissons actuellement est 'Internet of People'. C'est là que les gens interagissent les uns avec les autres, avec des machines qui facilitent cette communication. Nous visualisons les sites que les gens conçoivent. Nous lisons des mots que les gens ont tapés.

L'Internet of Things est celui où les appareils communiquent directement les uns avec les autres sans intervention humaine. Un seul appareil surveille le temps qu'il fait. Un thermostat intelligent accède à cette information et ajuste la température de notre maison.

Le Big Data et l'Internet of Things sont interdépendants. Ces appareils sont capables d'agir par eux-mêmes grâce à toutes les données dont ils disposent. Plus il y a d'appareils qui fonctionnent de cette façon, plus il y a des données qui sont générées.

❖ **Machine Learning**

Le Machine Learning fait référence à la capacité d'un ordinateur d'apprendre à partir des données. C'est ainsi que les stations de radio Pandora (Music streaming) s'adaptent à notre style particulier. L'apprentissage machine est également à la base des recommandations de contenu sur YouTube et Netflix. Ces prédictions sont grâces à des algorithmes, comme l'algorithme de recherche de Google et l'algorithme qui détermine ce que nous voyons dans le fil de nouvelles de Facebook, tout cela, c'est de l'apprentissage machine au travail.

❖ **Intelligence artificielle**

L'intelligence artificielle est l'étape suivante après l'apprentissage machine. Ici, non seulement un ordinateur apprend à partir des données, mais il utilise cette information pour prendre ses propres décisions et façonner son propre comportement.

❖ **Transports**

Dans le domaine des transports, l'analyse des données du Big Data (données provenant des *passes* de transport en commun, géolocalisation des personnes et des voitures, etc.) permet de modéliser les déplacements des populations afin d'adapter les infrastructures et les services (horaires et fréquence des trains, par exemple).

❖ **Gestion énergétique**

Dans le domaine de la gestion énergétique, l'analyse des données issues du Big Data intervient dans la gestion de réseaux énergétiques complexes via les réseaux électriques intelligents (*smartgrids*) qui utilisent des technologies informatiques pour optimiser la production, la distribution et la consommation de l'électricité.

❖ **Domaine airien**

De la même manière, l'analyse des données provenant de capteurs sur les avions (données de vol) associées à des données météo permet de modifier les couloirs aériens afin de réaliser des économies de carburant et d'améliorer la conception et la maintenance des avions.

❖ **Recherche scientifique**

Le big data en est issu et il alimente une partie de la recherche. Ainsi THE Large Hadron Collider du CERN (la plus grande machine au monde et le plus grand collisionneur de l'énergie) utilise environ 150 millions de capteurs délivrant des données 40 millions de fois par seconde ; Pour 600 millions de collisions par seconde, il reste après filtrage 100 collisions d'intérêt par seconde, soit 25 Po des données à stocker par an, et 200 Po après réplication. Les outils d'analyse du big data pourraient affiner l'exploitation de ces données.

❖ **Politique**

L'analyse du *big data* a joué un rôle important dans la campagne de resélections de Barack Obama, notamment pour analyser les opinions politiques de la population.

Depuis 2012, le département de la Défense américain investit annuellement sur les projets *big data* plus de 250 millions de dollars. Le gouvernement américain possède six des dix plus puissants supercalculateurs de la planète. La National Security Agency (NSA) est actuellement en train de construire le Utah Data Center qui stockera jusqu'à un yottaoctet d'informations collectées par la NSA sur internet.

❖ **L'industrie**

L'industrie est un des tous premiers secteurs qui se voit transformé par l'émergence du Big Data. La maintenance prédictive, permise par le Big Data permet ainsi d'anticiper les risques de pannes et de défections de certaines pièces ou composants. On peut ainsi positionner des capteurs sur les machines pour avoir des informations en temps réel sur leur usure et éviter les pannes.

La Data permet également d'optimiser les dépenses : on peut optimiser la consommation d'énergie d'un point de vente, optimiser les coûts de transport, de stocks notamment en croisant plusieurs données entre elles.

f) Comment gérer les big data ?

Les créations technologiques qui ont facilité la venue et la croissance du Big Data peuvent globalement être catégorisées en deux familles : d'une part, les technologies de stockage, portées particulièrement par le déploiement du Cloud Computing. D'autre part, l'arrivée de technologies de traitement ajustées, spécialement le développement de nouvelles bases des données adaptées aux données non-structurées (Hadoop) et la mise au point de modes de calcul à haute performance (MapReduce).

Il existe plusieurs solutions qui peuvent entrer en jeu pour optimiser les temps de traitement sur des bases des données géantes à savoir les bases des données NoSQL (MongoDB, Cassandra ou Redis), les infrastructures du serveur pour la distribution des traitements sur les nœuds et le stockage des données en mémoire :

- ✓ La première solution permet d'implémenter les systèmes de stockage considérés comme plus performants que le traditionnel SQL pour l'analyse des données en masse (orienté clé/valeur, document, colonne ou graphe).
- ✓ La deuxième est aussi appelée le traitement massivement parallèle. Le Framework Hadoop en est un exemple. Celui-ci combine le système de fichiers distribué HDFS, la base NoSQL HBase et l'algorithme MapReduce.

Chaque technologie, appartenant au système mégadonnée, a son utilité, ses atouts et ses inconvénients. Etant un milieu en perpétuelle évolution, le Big Data cherche toujours à optimiser les performances des outils. Ainsi, son paysage technologique bouge très vite, et de nouvelles solutions naissent très fréquemment, pour but d'optimiser encore plus les technologies existantes. Pour illustrer cette évolution, Hadoop et Spark représentent des exemples très concrets.

2. *Hadoop*

a) Introduction

Hadoop est un projet open source d'Apache qui a commencé par Yahoo en 2006, il est un software librairie est un Framework qui permet de traiter les grands ensembles des données (big data).

Il est composé des modules qui travaillent ensemble pour créer le Framework Hadoop, les principaux modules sont :

- ✓ Hadoop Common
- ✓ Hadoop Distributed File System (HDFS)
- ✓ Hadoop YARN
- ✓ Hadoop MapReduce

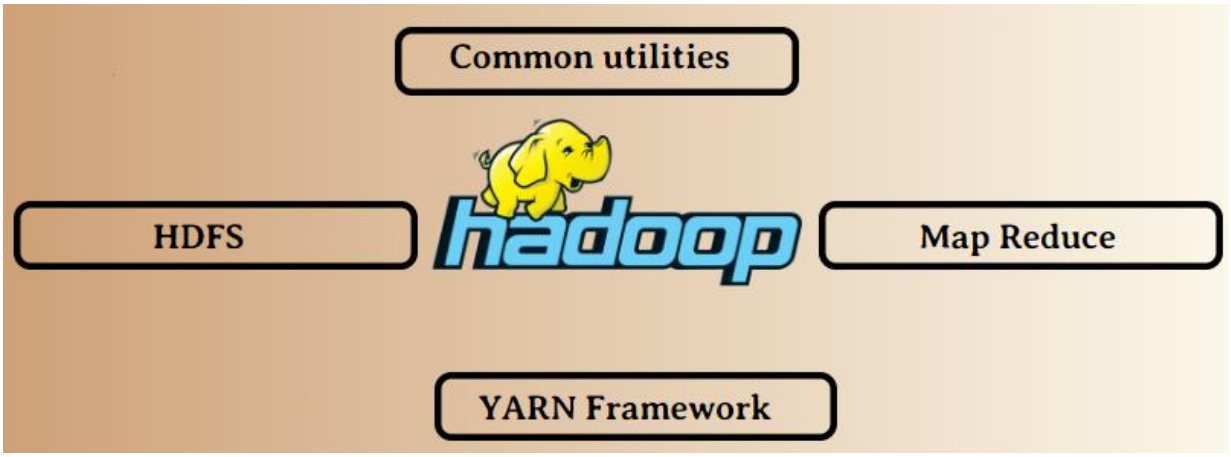

Figure 4: Les principaux Modules de Hadoop

Il y a d'autre module aussi : Ambari, Avro, Cassandra, Hive, Pig, Oozie, Flume, and Sqoop .

Elle repose principalement sur HDFS (Hadoop File System), un système de fichier (file system), comme ceux qui gèrent les disques durs, capable de reconnaître et d'indexer les sources et formats des données SQL et NoSQL, donc d'offrir une vision unique d'un ensemble des données. En plus de HDFS pour le stockage des fichiers, Hadoop peut être aussi configurer pour utiliser S3 buckets ou Azure blobs comme des entrés.

Hadoop est écrit en Java avec des parties natives avec C, et les commandes de ligne sont écrites comme un Shell script, mais accessible par plusieurs langages de programmation pour écrire les codes de MapReduce comme Phyton.

Hadoop contient aussi HIVE qui est une interface pour écrire des requêtes SQL, et Mahout pour le MACHING LEARNING.

b) Domaine d'application

Figure 5: Domaine d'application de Hadoop

❖ **Suivi des sites web (Web site Tracking)**

Hadoop va capturer une énorme quantité des données à propos des visiteurs de notre site web. Il fournira des informations sur l'emplacement du visiteur, la page visitée en premier et le plus, combien de temps passé sur le site Web et sur quelle page, combien de fois un visiteur a visité la page, quel visiteur aime le plus. Cela fournira une analyse prédictive de l'intérêt des visiteurs, les performances du site Web prédiront ce qui serait l'intérêt des utilisateurs. Hadoop accepte les données dans plusieurs formats provenant de plusieurs sources. Apache HIVE sera utilisé pour traiter des millions des données.

❖ **Données géographiques**

Lorsque nous achetons des produits sur un site Web de commerce électronique. Le site Web suivra l'emplacement de l'utilisateur, prédira les achats des clients à l'aide de smartphones, de tablettes et de PC. Le cluster Hadoop aidera à comprendre les activités de géolocalisation. Cela aidera les industries à afficher le graphique des entreprises dans chaque ville /région (positif ou négatif).

❖ **Commerce de détail**

Les détaillants utiliseront les données des clients qui sont présentes dans le format structuré et non structuré, pour comprendre et analyser les données. Cela aidera un utilisateur

à comprendre les besoins des clients et à leur offrir de meilleurs avantages et des services améliorés.

- **Secteur financier**

L'industrie financière et les sociétés financières évalueront le risque financier, la valeur marchande et construiront le modèle qui donnera aux clients et à l'industrie de meilleurs résultats en termes d'investissement comme le marché boursier, FD, etc.

- **L'industrie de la santé**

Hadoop peut stocker de grandes quantités des données. Les données médicales sont présentes dans un format non structuré. Cela aidera le médecin pour un meilleur diagnostic. Hadoop conservera des antécédents médicaux du patient de plus d'un an, analysera les symptômes de la maladie.

- **Le marketing numérique**

Nous sommes à l'ère des années 20, chaque personne est connectée numériquement. Les informations sont communiquées à l'utilisateur par les téléphones portables ou par les ordinateurs et les gens sont informés de chaque détail concernant les actualités, les produits, etc. Hadoop stockera massivement les données générées en ligne, stockera, analysera et fournira le résultat aux sociétés de marketing numérique.

c) Avantage de Hadoop

- **Open Source**

Hadoop est de nature open source, c'est-à-dire que son code source est disponible gratuitement. Nous pouvons modifier le code source selon nos exigences commerciales. Même des versions propriétaires de Hadoop comme Cloudera et Horton sont également disponibles.

- **Évolutif**

Hadoop travaille sur le cluster de machines. Il est hautement évolutif. Nous pouvons augmenter la taille de notre cluster en ajoutant de nouveaux nœuds selon les besoins sans aucun temps d'arrêt. Cette façon d'ajouter de nouvelles machines au cluster est connue sous le nom de

mise à l'échelle horizontale, tandis que l'augmentation de composants comme le doublement du disque dur et de la RAM est appelée mise à l'échelle verticale.

- **Tolérance de panne**

La tolérance aux pannes est la caractéristique saillante de Hadoop. Par défaut, chaque bloc de HDFS a un facteur de réplication de 3. Pour chaque bloc des données, HDFS crée deux copies supplémentaires et les stocke à un emplacement différent dans le cluster. Si un bloc est manquant en raison d'une défaillance de la machine, nous avons encore deux copies du même bloc et celles-ci sont utilisées. De cette façon, la tolérance aux pannes est obtenue dans Hadoop.

- **Indépendant du schéma**

Hadoop peut fonctionner sur différents types des données. Il est suffisamment flexible pour stocker différents formats des données et peut fonctionner à la fois avec des données de schéma (structurées) et sans schéma (non structurées).

- **Haut débit et faible latence**

Le débit signifie une quantité de travail effectuée par unité de temps et une faible latence signifie de traiter les données sans délai ou moins. Hadoop étant guidé par le principe du stockage distribué et du traitement parallèle, le traitement est effectué simultanément sur chaque bloc des données et indépendamment les uns des autres. De plus, au lieu de déplacer des données, le code est déplacé vers les données du cluster. Ces deux éléments contribuent au débit élevé et à la faible latence.

- **Localité des données**

Hadoop fonctionne sur le principe de « déplacer le code, pas les données ». Dans Hadoop, les données restent stationnaires et pour le traitement des données, le code est déplacé vers les données sous forme de tâches, c'est ce que l'on appelle la localité des données. Comme nous traitons des données dans la plage de pétaoctets, il devient à la fois difficile et coûteux de déplacer les données à travers le réseau, la localité des données garantit que le mouvement des données dans le cluster est minimum.

❖ **Performance**

Dans les systèmes hérités comme le SGBDR, les données sont traitées séquentiellement mais dans le traitement Hadoop démarre sur tous les blocs à la fois, ce qui permet un traitement parallèle. En raison des techniques de traitement parallèle, les performances de Hadoop sont beaucoup plus élevées que les systèmes hérités comme le SGBDR. En 2008, Hadoop a même battu le supercalculateur le plus rapide présent à l'époque.

❖ **Architecture de partage rien**

Chaque nœud du cluster Hadoop est indépendant les uns des autres. Ils ne partagent ni ressources ni stockage, cette architecture est connue sous le nom de Share Nothing Architecture (SN). Si un nœud du cluster tombe en panne, il ne fera pas tomber l'ensemble du cluster car chaque nœud agit indépendamment, éliminant ainsi un point de défaillance unique.

❖ **Prise en charge de plusieurs langues**

Bien que Hadoop ait été principalement développé en Java, il étend la prise en charge d'autres langages comme Python, Ruby, Perl et Groovy.

❖ **Rentable**

Hadoop est de nature très économique. Nous pouvons créer un cluster Hadoop en utilisant du matériel standard, réduisant ainsi les coûts matériels. Selon l'ère du cloud, les coûts de gestion des données de Hadoop, à savoir le matériel et les logiciels et les autres dépenses, sont très minimes par rapport aux systèmes ETL traditionnels.

❖ **Abstraction**

Hadoop fournit l'abstraction à différents niveaux. Cela rend le travail plus facile pour les développeurs. Un gros fichier est divisé en blocs de même taille et stocké à différents emplacements du cluster. Lors de la création de la tâche de réduction de carte, nous devons nous soucier de l'emplacement des blocs. Nous donnons un fichier complet en entrée et le Framework Hadoop s'occupe du traitement de différents blocs des données qui se trouvent à différents endroits. Hive fait partie de l'écosystème Hadoop et c'est une abstraction au-dessus de Hadoop. Comme les tâches MapReduce sont écrites en Java, les développeurs SQL du monde entier n'ont pas pu profiter de MapReduce. Ainsi, Hive est introduit pour résoudre ce problème. Nous pouvons écrire des requêtes de type SQL sur Hive, ce qui déclenche à son tour des travaux de

réduction de carte. Ainsi, grâce à Hive, la communauté SQL est également en mesure de travailler sur les tâches de MapReduce.

❖ **Compatibilité**

Dans Hadoop, HDFS est la couche de stockage et MapReduce est le moteur de traitement. Mais il n'y a pas de règle rigide selon laquelle MapReduce devrait être le moteur de traitement par défaut. De nouveaux cadres de traitement comme Apache Spark et Apache Flink utilisent HDFS comme système de stockage. Même dans Hive, nous pouvons également changer notre moteur d'exécution en Apache Tez ou Apache Spark selon nos exigences. Apache HBase, qui est la base des données en colonnes NoSQL, utilise HDFS pour la couche de stockage.

❖ **Prise en charge de divers systèmes de fichiers**

Hadoop est de nature très flexible. Il peut ingérer différents formats des données comme des images, des vidéos, des fichiers, etc. Il peut également traiter des données structurées et non structurées. Hadoop prend en charge divers systèmes de fichiers comme JSON, XML, Avro, Parquet, etc.

d) Architecture Hadoop/Composant majeurs

Hadoop est composée de 5 composent majeurs :

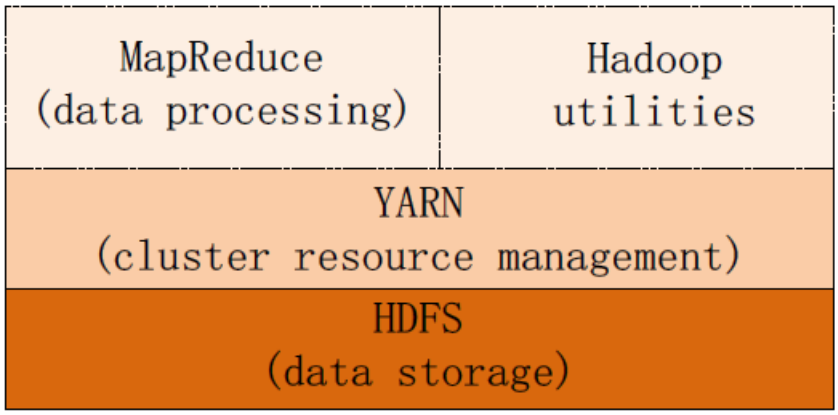

Figure 6: Composant majeurs de Hadoop

(1) Common utilities

Aussi appelé le Hadoop commun et Hadoop Core. Ce ne sont que les bibliothèques, fichiers, scripts et utilitaires JAVA réellement requis par les autres composants Hadoop pour fonctionner, Il s'agit d'une partie ou d'un module essentiel du Framework Apache Hadoop, avec le système de fichiers

distribués Hadoop (HDFS), Hadoop YARN et Hadoop MapReduce. Comme tous les autres modules, Hadoop Common suppose que les pannes matérielles sont courantes et que celles-ci doivent être gérées automatiquement dans le logiciel par Hadoop Framework.

(2) HDFS

HDFS est la couche de stockage pour le Big Data, c'est un cluster de nombreuses machines, les données stockées peuvent être utilisées pour le traitement à l'aide de Hadoop. Une fois les données sont transmises à HDFS, nous pouvons les traiter à tout moment, jusqu'au moment où nous traitons les données, elles résideront dans HDFS jusqu'à ce que nous supprimions les fichiers manuellement.

HDFS stocke les données sous forme de bloc, la taille minimale du bloc est de 128 Mo dans Hadoop 2.x et pour 1.x, il était de 64 Mo. HDFS réplique les blocs pour les données disponibles si les données sont stockées sur une machine et si cette dernière tombe en panne, les données ne sont pas perdues, mais pour les éviter, les données sont répliquées sur différentes machines. Le facteur de réplication par défaut est 3 et nous pouvons changer dans HDFS-site.xml ou en utilisant la commande Hadoop fs -strep -w 3 / dir en répliquant nous avons les blocs sur différentes machines pour une haute disponibilité.

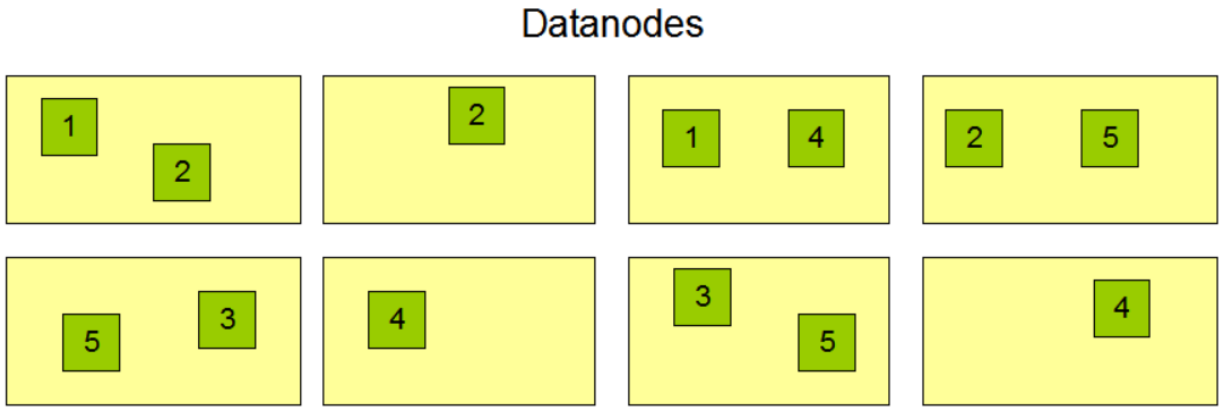

Figure 7: Réplication des Données

Puisque HDFS suit une architecture maître-esclave il a deux principaux composants de HDFS sont NAME NODE et DATA NODE.

- *NAME NODE* : est le maître, nous pouvons également avoir NAME NODE secondaire au cas où le principal cesserait de fonctionner, NAME NODE secondaire agirait comme une sauvegarde. Le NAME NODE gère essentiellement les nœuds des données en stockant des métadonnées.
- *DATA NODE* : est l'esclave, qui est essentiellement le matériel de base à faible coût. Nous pouvons avoir plusieurs DATA NODES. Il stocke les données réelles. Ce DATA NODE prend en charge le facteur de réplication, supposons que si un DATA NODE tombe en panne, les données peuvent être accessibles par l'autre DATA NODE répliqué. Par conséquent, l'accessibilité des données est améliorée et la perte des données est évitée.

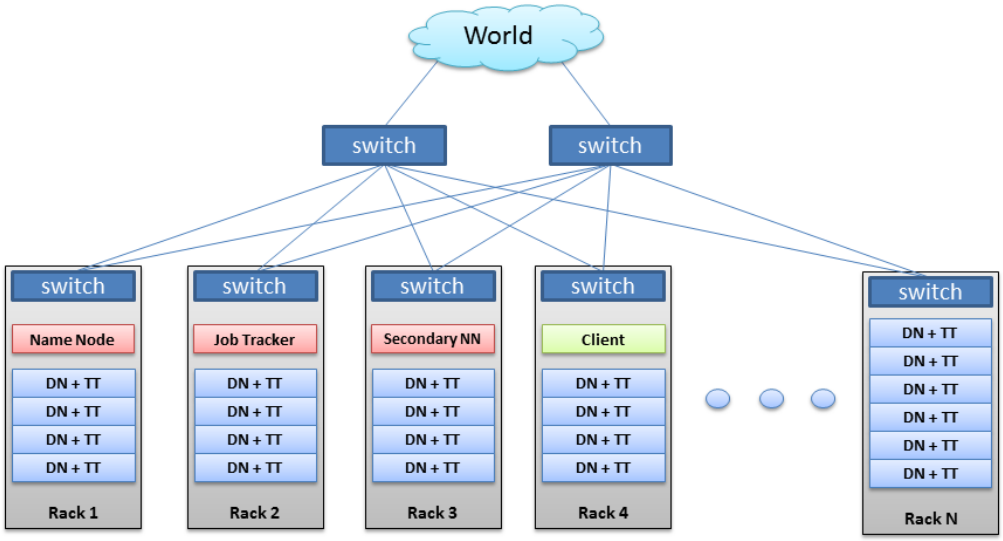

Figure 8: Architecture Maitre-Esclave de HDFS

❖ **<u>Opération d'écriture HDFS</u>**

Hadoop suit les étapes ci-dessous pour écrire n'importe quel gros fichier :

- ✓ Créez un fichier et mettez à jour l'image FS après avoir reçu une demande d'écriture de fichier de n'importe quel client HDFS.
- ✓ Obtenez des informations sur l'emplacement du bloc ou les détails du nœud des données à partir du nœud de nom.
- ✓ Écrivez le paquet d'une manière parallèle de nœuds des données individuels.
- ✓ Reconnaissez l'achèvement ou l'acceptation de l'écriture de paquets et renvoyez les informations au client Hadoop.

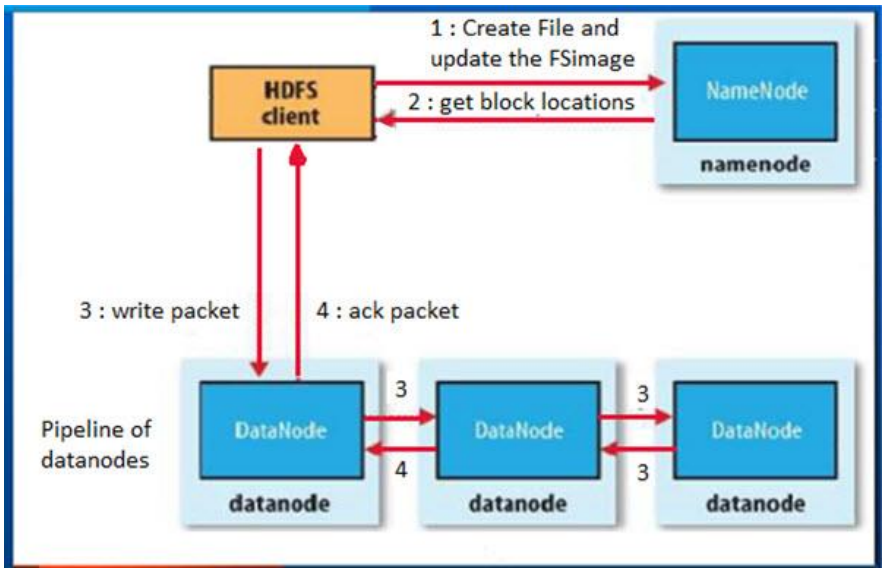

Figure 9: Opération d'architecture sur HDFS

❖ **<u>Pourquoi Hadoop a choisi d'intégrer un système de fichiers distribué ?</u>**

Comprenons cela avec un exemple :

Nous devons lire 1 To des données et nous avons une machine avec 4 canaux d'E / S, chaque canal ayant 100 Mo / s, il a fallu 45 minutes pour lire la totalité des données. Maintenant, la même quantité des données est lue par 10 machines avec chacune 4 canaux d'E / S, chaque canal ayant 100 Mo / s. Le temps qu'il a fallu pour lire les données et envirent 4 et 3 minutes. HDFS résout le problème du stockage des mégadonnées.

(3) MapReduce

L'écosystème Hadoop est une façon rentable, évolutive et flexible de travailler avec des ensembles des données aussi volumineux. Hadoop est un cadre qui utilise un modèle de programmation particulier, appelé MapReduce, pour diviser les tâches de calcul en blocs qui peuvent être distribués autour d'un cluster de machines de base à l'aide de Hadoop Distributed Filesystem (HDFS).

MapReduce est Framework de programmation qui nous permet d'effectuer un traitement distribué et parallèle sur de grands ensembles des données dans un environnement distribué. Il se compose de deux tâches distinctes : Map et Reduce.

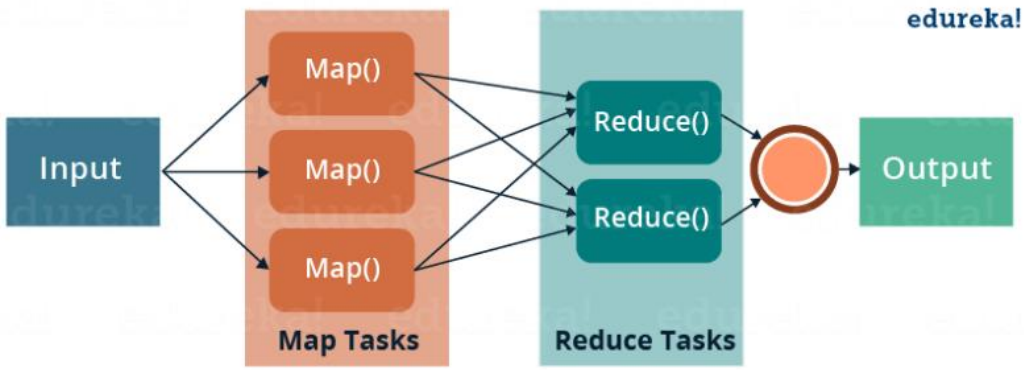

Figure 10: les class de MapReduce

Comme le nom MapReduce le suggère, la phase de réduction a lieu une fois la phase de mappage terminée. Le premier est le travail de mappage, où un bloc des données est lu et traité pour produire des paires clé-valeur en tant que sorties intermédiaires, la sortie d'un travail de mappage ou de mappage (paires clé-valeur) est entrée dans le réducteur, le réducteur reçoit la paire clé-valeur de plusieurs travaux de mappage. Ensuite, le réducteur agrège ces tuples des données intermédiaires (paire clé-valeur intermédiaire) en un plus petit ensemble de tuples ou paires clé-valeur qui est la sortie finale.

Comprenons mieux MapReduce et ses composants. MapReduce a principalement les trois classes suivantes :

- **Classe Mapper**

  La première étape du traitement des données à l'aide de MapReduce est la classe Mapper. Ici, RecordReader traite chaque enregistrement d'entrée et génère la paire clé-valeur respective. Le Mapper de Hadoop enregistre ces données intermédiaires sur le disque local.

  - ✓ Split d'entrée : C'est la représentation logique des données. Il représente un bloc de travail qui contient une seule tâche de carte dans le programme MapReduce.
  - ✓ RecordReader : Il interagit avec le fractionnement d'entrée et convertit les données obtenues sous la forme de paires valeur-clé.

- **Classe de Reducer**

  Reduce est la classe qui accepte les clés et les valeurs de la sortie de la phase des mappeurs. Les clés et les valeurs générées par le mappeur sont acceptées en entrée dans le réducteur pour un traitement ultérieur. Le réducteur accepte les données de plusieurs mappeurs. Le réducteur agrège ces données intermédiaires en un nombre réduit de clés et de valeurs qui est la sortie finale.

❖ Classe de Driver

Outre la classe mapper et Reducer, nous avons besoin d'une classe supplémentaire qui est la classe Driver. Ce code est nécessaire pour MapReduce car il est le pont entre le Framework et la logique implémentée. Il spécifie la configuration, le chemin des données d'entrée, le chemin de stockage de sortie et, plus important encore, les classes de mappage et de réduction qui doivent être implémentées et de nombreuses autres configurations doivent être définies dans cette classe.

❖ Un exemple de comptage de mots de MapReduce

Comprenons comment fonctionne MapReduce en prenant un exemple où on a un fichier texte appelé example.txt dont le contenu est le suivant :

Dear, Bear, River, Car, Car, River, Deer, Car and Bear

Supposons maintenant que nous devons effectuer un comptage de mots sur sample.txt à l'aide de MapReduce. Ainsi, nous trouverons les mots uniques et le nombre d'occurrences de ces mots uniques.

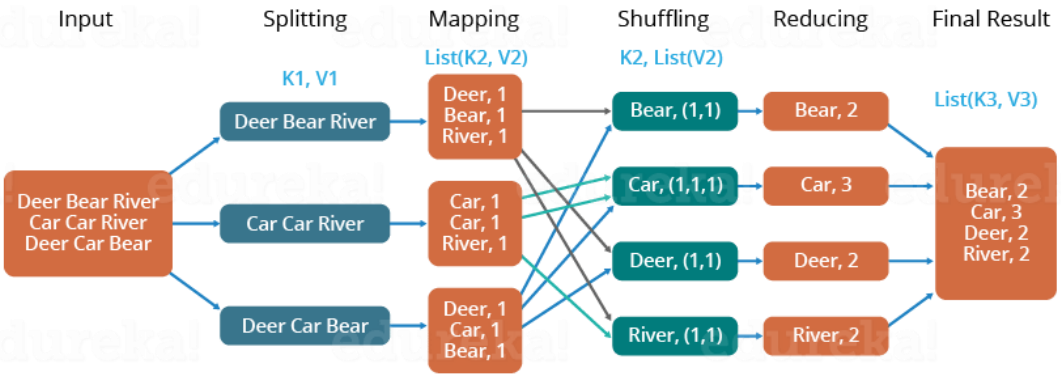

Figure 11: Exemple de comptage de mots

Tout d'abord, nous divisons l'entrée en trois divisions, comme le montre la figure. Cela répartira le travail entre tous les nœuds de la carte.

Ensuite, nous jetons les mots dans chacun des mappeurs et donnons une valeur codée en dur (1) à chacun des jetons ou mots. La raison pour laquelle une valeur codée en dur égale à 1 est que chaque mot, en soi, apparaîtra une fois.

Maintenant, une liste de paires clé-valeur sera créée où la clé n'est rien mais les mots individuels et la valeur en sont un. Ainsi, pour la première ligne (Dear Bear River), nous avons 3 paires clé-valeur :

Dear, 1 ; Ours, 1 ; River, 1.

Le processus de cartographie reste le même sur tous les nœuds. Après la phase de mappage, un processus de partition a lieu où le tri et le brassage se produisent de sorte que tous les tuples avec la même clé soient envoyés au réducteur correspondant.

Ainsi, après la phase de tri et de brassage, chaque réducteur aura une clé unique et une liste de valeurs correspondant à cette même clé. Par exemple, Bear, [1,1] ; cars, [1,1,1] ..., etc.

Maintenant, chaque réducteur compte les valeurs présentes dans cette liste de valeurs. Comme le montre la figure, le réducteur obtient une liste de valeurs qui est [1,1] pour la clé Bear. Ensuite, il compte le nombre de ceux de la liste et donne la sortie finale comme - Bear, 2.

Enfin, toutes les paires clé / valeur de sortie sont ensuite collectées et écrites dans le fichier de sortie.

- **<u>Explication du programme MapReduce</u>**

L'ensemble du programme MapReduce peut être fondamentalement divisé en trois parties :

- Code de phase du mappeur
- Code de phase du réducteur
- Code du conducteur

Nous comprendrons le code de chacune de ces trois parties de manière séquentielle.

- ✓ **Mapper code :**

```
//mapper
public static class Map extends Mapper<LongWritable,Text,Text,IntWritable> {
    public void map(LongWritable key, Text value, Context context) throws IOException,InterruptedException {
        String line = value.toString();
        StringTokenizer tokenizer = new StringTokenizer(line);
        while (tokenizer.hasMoreTokens()) {
            value.set(tokenizer.nextToken());
            context.write(value, new IntWritable(1));
        }
```

Figure 12: Class Map

Nous avons créé une classe Map qui étend le class Mapper qui est déjà défini dans le cadre MapReduce.

Nous définissons les types des données de la paire clé / valeur d'entrée et de sortie après la déclaration de classe à l'aide de crochets angulaires, l'entrée et la sortie du mappeur sont une paire clé / valeur :

- *Entrée (Input)* :

  La clé n'est rien d'autre que le décalage de chaque ligne du fichier texte : LongWritable

  La valeur est chaque ligne individuelle (comme indiqué dans la figure de droite) : Texte

- *Sortie (Output)* :

  La clé est les mots symbolisés : Texte

Nous avons la valeur codée en dur dans notre cas qui est 1 : IntWritable

Exemple - Dear 1, Bear 1, etc.

Nous avons écrit un code java dans lequel nous avons symbolisé chaque mot et leur avons attribué une valeur codée en dur égale à 1.

✓ Reducer Code :

```
11  //reduce
12  public static class Reduce extends Reducer<Text,IntWritable,Text,IntWritable> {
13      public void reduce(Text key, Iterable<IntWritable> values,Context context)
14              throws IOException,InterruptedException {
15                  int sum=0;
16                  for(IntWritable x: values)
17                  {
18                          sum+=x.get();
19                  }
20                  context.write(key, new IntWritable(sum));
21              }
22      }
```

Figure 13: Class Reduce

Nous avons créé une classe Reduce qui étend la classe Reducer comme celle de Mapper. Nous définissons les types des données de la paire clé / valeur d'entrée et de sortie après la déclaration de classe en utilisant des parenthèses angulaires comme pour Mapper. L'entrée et la sortie du réducteur sont une paire clé-valeur :

- Contribution :

  La clé rien que ces mots uniques qui ont été générés après la phase de tri et de mélange : Texte

  La valeur est une liste d'entiers correspondant à chaque clé : IntWritable

  Exemple - Ours, [1, 1], etc.

- Production :

  La clé est tous les mots uniques présents dans le fichier texte d'entrée : Texte

  La valeur est le nombre d'occurrences de chacun des mots uniques : IntWritable

  Exemple - Ours, 2 ; Voiture, 3, etc.

Nous avons agrégé les valeurs présentes dans chacune des listes correspondant à chaque clé et produit la réponse finale. En général, un seul réducteur est créé pour chacun des mots uniques, mais vous pouvez spécifier le nombre de réducteurs dans mapred-site.xml.

✓ **Driver Code :**

```
24    //driver
25    Configuration conf= new Configuration();
26   Job job = new Job(conf,"My Word Count Program");
27   job.setJarByClass(WordCount.class);
28   job.setMapperClass(Map.class);
29   job.setReducerClass(Reduce.class);
30   job.setOutputKeyClass(Text.class);
31   job.setOutputValueClass(IntWritable.class);
32   job.setInputFormatClass(TextInputFormat.class);
33   job.setOutputFormatClass(TextOutputFormat.class);
34   Path outputPath = new Path(args[1]);
35   //Configuring the input/output path from the filesystem into the job
36   FileInputFormat.addInputPath(job, new Path(args[0]));
37   FileOutputFormat.setOutputPath(job, new Path(args[1]));
```

Figure 14: Driver Code

Dans la classe de pilote, nous avons défini la configuration de notre travail MapReduce pour qu'il s'exécute dans Hadoop. Nous spécifions le nom du travail, le type des données d'entrée/sortie du mappeur et du réducteur. Nous spécifions également les noms des classes de mappage et de réduction.

Le chemin du dossier d'entrée et de sortie est également spécifié. La méthode setInputFormatClass() est utilisée pour spécifier comment un mappeur lira les données d'entrée ou quelle sera l'unité de travail. Ici, nous avons choisi TextInputFormat pour qu'une seule ligne soit lue par le mappeur à la fois à partir du fichier texte d'entrée. La méthode main() est le point d'entrée du pilote. Dans cette méthode, nous instancions un nouvel objet Configuration pour le travail.

❖ **Source code :**

```
40  package co.edureka.mapreduce;
41  import java.io.IOException;
42  import java.util.StringTokenizer;
43  import org.apache.hadoop.io.IntWritable;
44  import org.apache.hadoop.io.LongWritable;
45  import org.apache.hadoop.io.Text;
46  import org.apache.hadoop.mapreduce.Mapper;
47  import org.apache.hadoop.mapreduce.Reducer;
48  import org.apache.hadoop.conf.Configuration;
49  import org.apache.hadoop.mapreduce.Job;
50  import org.apache.hadoop.mapreduce.lib.input.TextInputFormat;
51  import org.apache.hadoop.mapreduce.lib.output.TextOutputFormat;
52  import org.apache.hadoop.mapreduce.lib.input.FileInputFormat;
53  import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;
54  import org.apache.hadoop.fs.Path;
55
56  public class WordCount{
57      public static class Map extends Mapper&lt;LongWritable,Text,Text,IntWritable&gt; {
58          public void map(LongWritable key, Text value,Context context) throws IOException,InterruptedException{
59              String line = value.toString();
60              StringTokenizer tokenizer = new StringTokenizer(line);
61              while (tokenizer.hasMoreTokens()) {
62                  value.set(tokenizer.nextToken());
63                  context.write(value, new IntWritable(1));
64              }
65          }
66      }
```

Figure 15: Première partie du Main

```
67      public static class Reduce extends Reducer&lt;Text,IntWritable,Text,IntWritable&gt; {
68          public void reduce(Text key, Iterable&lt;IntWritable&gt; values,Context context) throws IOException,InterruptedException {
69             int sum=0;
70             for(IntWritable x: values)
71             {
72                 sum+=x.get();
73             }
74             context.write(key, new IntWritable(sum));
75          }
76      }
77      public static void main(String[] args) throws Exception {
78          Configuration conf= new Configuration();
79          Job job = new Job(conf,"My Word Count Program");
80          job.setJarByClass(WordCount.class);
81          job.setMapperClass(Map.class);
82          job.setReducerClass(Reduce.class);
83          job.setOutputKeyClass(Text.class);
84          job.setOutputValueClass(IntWritable.class);
85          job.setInputFormatClass(TextInputFormat.class);
86          job.setOutputFormatClass(TextOutputFormat.class);
87          Path outputPath = new Path(args[1]);
88          //Configuring the input/output path from the filesystem into the job
89          FileInputFormat.addInputPath(job, new Path(args[0]));
90          FileOutputFormat.setOutputPath(job, new Path(args[1]));
91          //deleting the output path automatically from hdfs so that we don't have to delete it explicitly
92          outputPath.getFileSystem(conf).delete(outputPath);
93          //exiting the job only if the flag value becomes false
94          System.exit(job.waitForCompletion(true) ? 0 : 1);
95      }
96  }
```

Figure 16: Deuxième partie du Main

(4) YARN

YARN a été introduit dans Hadoop 2.x, avant que Hadoop ne dispose d'un JobTracker pour la gestion des ressources. Job Tracker était le maître et il avait un TaskTracker comme esclave. Job Tracker était celui qui se chargeait de planifier les tâches et d'allouer les ressources.

Le traqueur de tâches était utilisé pour gérer les tâches de mappage et de réduction et le statut était mis à jour périodiquement pour le traqueur de travaux. Avec un type de gestionnaire de ressources, il avait une limite d'évolutivité et l'exécution simultanée des tâches était également limitée. Ces problèmes ont été traités dans YARN et il s'est occupé de l'allocation des ressources et de la planification des travaux sur un cluster. L'exécution d'un travail MapReduce nécessite des ressources dans un cluster, pour obtenir les ressources allouées pour le travail que YARN aide.

YARN détermine quel travail est effectué et quelle machine il est effectué. Il a toutes les informations des cœurs et de la mémoire disponibles dans le cluster, il suit la consommation de mémoire dans le cluster. Il interagit avec le NameNode sur les données où il réside pour prendre la décision sur l'allocation des ressources, donc on peut conclure qu'il y a deux rôles clés à jouer :

- Planification des tâches : lorsqu'une grande quantité des données est destinée au traitement, elles doivent être réparties et réparties en différentes tâches / tâches. Maintenant, il décide quel travail doit recevoir la priorité absolue, l'intervalle de temps entre deux travaux, la dépendance entre les travaux, vérifie qu'il n'y a pas de chevauchement entre les travaux en cours d'exécution.
- Gestion des ressources : pour traiter les données et stocker les données, nous avons besoin de ressources.

Ainsi, le gestionnaire de ressources fournit, gère et maintient les ressources pour stocker et traiter les données.

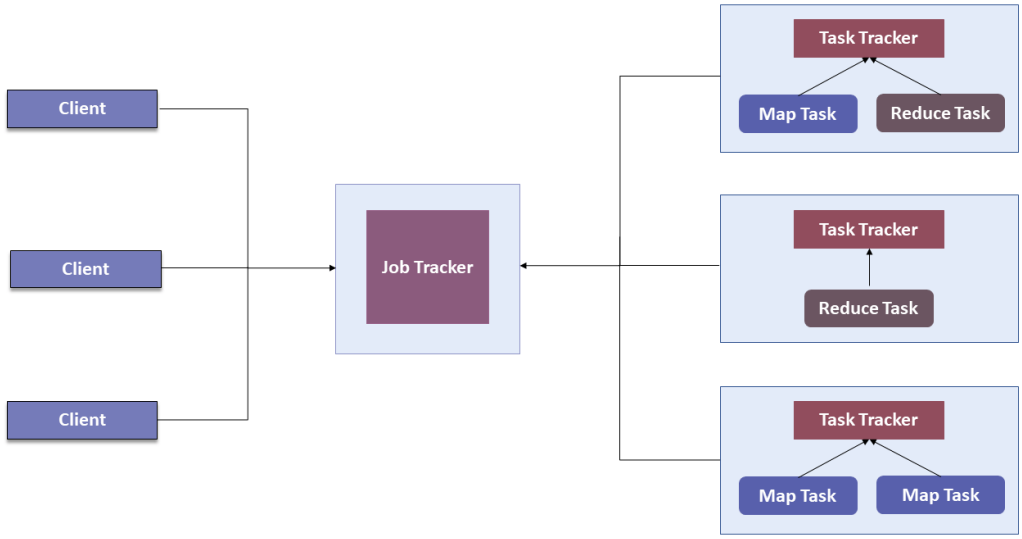

Figure 17: Architecture de YARN

3. *Spark*

a) Introduction :

Spark est un projet plus récent, initialement développé en 2012, à l'AMPLab à UC Berkeley. Il s'agit également d'un projet Apache de haut niveau axé sur le traitement des données en parallèle sur un cluster, mais la plus grande différence est qu'il fonctionne en mémoire.

Spark est structuré autour de Spark Core, le moteur qui pilote la planification, les optimisations et l'abstraction RDD (Resilient Distributed Dataset), ainsi que connecte Spark au système de fichiers approprié (HDFS, S3, RDBM ou Elasticsearch). Il existe plusieurs bibliothèques qui fonctionnent au-dessus de Spark Core, y compris Spark SQL, qui vous permet d'exécuter des commandes de type SQL sur des ensembles des données distribués, MLLib pour l'apprentissage automatique, GraphX pour les problèmes de graphique et le streaming qui permet l'entrée de streaming en continu consigner les données.

b) Domaine d'application

- **Industrie financière**

  La plupart du temps, les banques utilisent l'alternative Hadoop - Spark. Il permet d'accéder et d'analyser de nombreux paramètres du secteur bancaire. Par exemple, les profils de médias sociaux, les e-mails, le forum, les enregistrements d'appels et bien d'autres. De plus, il acquiert des connaissances qui aident à prendre les bonnes décisions pour plusieurs zones. Comme l'évaluation du risque de crédit, la publicité ciblée et la segmentation de la clientèle.

- **Industrie du commerce électronique**

  Dans le commerce électronique, il aide à fournir des informations sur une transaction en temps réel. Celles-ci sont transmises aux algorithmes de clustering en streaming. Tels que l'alternance des moindres carrés ou l'algorithme de clustering K-means. Il contribue également à améliorer les recommandations aux clients en fonction des nouvelles tendances. Quelques exemples en temps réel comme Alibaba, eBay utilisant Spark dans le commerce électronique.

- **Industrie des médias et du divertissement**

  Dans le jeu, nous utilisons Spark pour identifier les modèles des événements en temps réel dans le jeu. Il aide à réagir afin de récolter des opportunités commerciales lucratives. Par exemple, publicité ciblée, ajustement automatique de la complexité du niveau de jeu, rétention des joueurs, etc.

  En outre, certains sites Web de partage de vidéos utilisent Spark avec MongoDB. Il aide à montrer des publicités pertinentes à ses utilisateurs en fonction des vidéos qu'ils consultent, partagent et

parcourent. Certaines entreprises en temps réel qui utilisent Spark sont Yahoo, Netflix, Pinterest, Conviva etc.

❖ **L'industrie du voyage**

Les industries du voyage utilisent rapidement Apache Spark. Il aide les utilisateurs à planifier un voyage parfait en accélérant les recommandations personnalisées. Ils l'utilisent également pour conseiller les voyageurs en comparant de nombreux sites Web pour trouver les meilleurs prix d'hôtel. De plus, le processus d'examen des hôtels dans un format lisible se fait en utilisant Spark.

De plus, certaines applications utilisent Spark pour nous fournir une plateforme de réservation en ligne (en temps réel). Ils utilisent Spark pour gérer de nombreux restaurants et réservations de dîner en même temps. La vitesse atteinte par eux n'est possible qu'en utilisant Apache Spark. La réduction du temps d'exécution de l'apprentissage automatique de quelques semaines à quelques heures a permis d'améliorer le travail d'équipe. Certains des meilleurs exemples de cette section sont TripAdvisor et OpenTable.

c) Avantages de Spark

Apache Spark possède les fonctionnalités suivantes :

❖ **Vitesse**

Spark permet d'exécuter une application dans le cluster Hadoop, jusqu'à 100 fois plus rapide en mémoire et 10 fois plus rapide lors de l'exécution sur disque. Ceci est possible en réduisant le nombre d'opérations de lecture / écriture sur le disque. Il stocke les données de traitement intermédiaires en mémoire.

❖ **Prend en charge plusieurs langues**

Spark fournit des API intégrées en Java, Scala ou Python. Par conséquent, vous pouvez écrire des applications dans différentes langues. Spark propose 80 opérateurs de haut niveau pour les requêtes interactives.

❖ **Advanced Analytics**

Spark ne prend pas seulement en charge « Map » et « Reduce ». Il prend également en charge les requêtes SQL, les données en streaming, l'apprentissage automatique (ML) et les algorithmes de graphique.

❖ **Traitement rapide**

En utilisant Apache Spark, nous atteignons une vitesse de traitement des données élevée environ 100 fois plus rapide en mémoire et 10 fois plus rapide sur le disque. Ceci est rendu possible en réduisant le nombre de lecture-écriture sur le disque.

❖ **Dynamique dans la nature**

Nous pouvons facilement développer une application parallèle, car Spark fournit 80 opérateurs de haut niveau.

❖ **Calcul en mémoire dans Spark**

Avec le traitement en mémoire, nous pouvons augmenter la vitesse de traitement. Ici, les données sont mises en cache, nous n'avons donc pas besoin de récupérer les données du disque à chaque fois, ce qui permet de gagner du temps. Spark possède un moteur d'exécution DAG qui facilite le calcul en mémoire et le flux des données acyclique, ce qui se traduit par une vitesse élevée.

❖ **Réutilisabilité**

Nous pouvons réutiliser le code Spark pour le traitement par lots, joindre le flux à des données historiques ou exécuter des requêtes ad hoc sur l'état du flux.

❖ **Tolérance aux pannes dans Spark**

Apache Spark offre une tolérance aux pannes via Spark abstraction-RDD. Les RDD Spark sont conçus pour gérer la défaillance de tout nœud de travail dans le cluster. Ainsi, il garantit que la perte des données est réduite à zéro. Découvrez différentes façons de créer un RDD dans Apache Spark.

❖ **Traitement de flux en temps réel**

Spark a une disposition pour le traitement de flux en temps réel. Auparavant, le problème avec Hadoop MapReduce était qu'il peut gérer et traiter les données qui sont déjà présentes, mais pas les données en temps réel. Mais avec Spark Streaming, nous pouvons résoudre ce problème.

- **Évaluation paresseuse dans Apache Spark**

  Toutes les transformations que nous faisons dans Spark RDD sont de nature paresseuse, c'est-à-dire qu'elles ne donnent pas le résultat tout de suite, mais un nouveau RDD est formé à partir de l'existant. Ainsi, cela augmente l'efficacité du système. Suivez ce guide pour en savoir plus sur Spark Lazy Evaluation en détail.

- **Intégré à Hadoop**

  Spark peut s'exécuter indépendamment et également sur Hadoop YARN Cluster Manager et peut ainsi lire les données Hadoop existantes. Ainsi, Spark est flexible.

- **Spark GraphX**

  Spark a GraphX, qui est un composant pour le calcul graphique et parallèle graphique. Il simplifie les tâches d'analyse graphique par la collecte d'algorithmes de graphiques et de constructeurs.

- **Rentable**

  Apache Spark est une solution rentable pour les problèmes de Big Data, car dans Hadoop, une grande quantité de stockage et le grand centre des données sont nécessaires lors de la réplication.

d) Comment Apache Spark facilite-t-il le travail ?

Spark est un puissant moteur de traitement des données open source. Il est conçu pour faciliter et accélérer le traitement des mégadonnées. Il prend en charge Java, Python, Scala et SQL, ce qui donne au programmeur la liberté de choisir le langage avec lequel il est à l'aise et de démarrer rapidement le développement. Spark est basé sur MapReduce mais contrairement à MapReduce, il ne mélange pas les données d'un cluster à un autre, Spark a un traitement en mémoire qui le rend plus rapide que MapReduce mais toujours évolutif. Il peut être utilisé pour créer des bibliothèques d'applications ou effectuer des analyses sur les mégadonnées, les applications par lots, les algorithmes itératifs, les requêtes interactives et le streaming.

Spark prend en charge l'évaluation paresseuse. Cela signifie qu'il attendra d'abord l'ensemble complet des instructions, puis le traitera. Supposons donc que l'utilisateur souhaite filtrer les enregistrements par date, mais qu'il ne souhaite que les 10 premiers enregistrements. Spark ne récupérera que 10 enregistrements du filtre donné plutôt que de récupérer tous les enregistrements du filtre, puis d'afficher 10 comme réponse. Cela permettra d'économiser du temps ainsi que des ressources.

Alors que Hadoop lit et écrit des fichiers sur HDFS, Spark traite les données dans la RAM en utilisant un concept appelé RDD (Resilient Distributed Dataset). Spark peut fonctionner soit en mode autonome, avec un cluster Hadoop servant de source des données, soit en conjonction avec Mesos. Dans ce dernier scénario, le maître Mesos remplace le maître Spark ou YARN à des fins de planification

e) Ecosystème de Spark

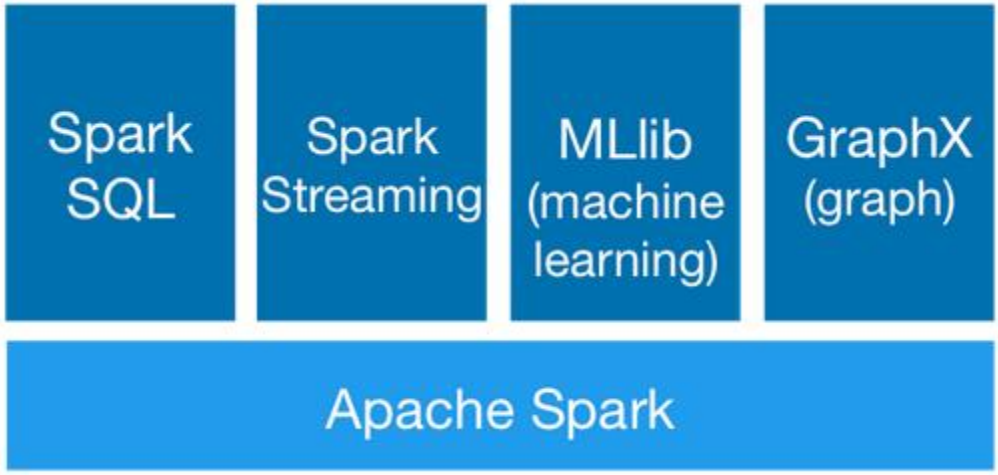

Figure 18: Ecosystème de Spark

- **Apache Spark Core**

  Spark Core est le moteur d'exécution général sous-jacent de la plateforme Spark sur lequel toutes les autres fonctionnalités sont basées. Il fournit des jeux des données de calcul et de référencement en mémoire dans des systèmes de stockage externes.

- **Spark SQL**

  Spark SQL est un composant au-dessus de Spark Core qui introduit une nouvelle abstraction des données appelée Schéma RDD, qui prend en charge les données structurées et semi-structurées.

- **Spark Streaming**

  Spark Streaming exploite la capacité de planification rapide de Spark Core pour effectuer des analyses de streaming. Il ingère les données en mini-lots et effectue des transformations RDD (Resilient Distributed Datasets) sur ces mini-lots des données.

- **MLlib (bibliothèque d'apprentissage automatique)**

  MLlib est un cadre d'apprentissage machine distribué au-dessus de Spark en raison de l'architecture Spark basée sur la mémoire distribuée. Il est, selon les repères, fait par les développeurs MLlib contre les implémentations ALS (Alternating Least Squares). Spark MLlib est

neuf fois plus rapide que la version sur disque Hadoop d'Apache Mahout (avant que Mahout n'ait acquis une interface Spark).

- **GraphX**

GraphX est un Framework de traitement graphique distribué au-dessus de Spark. Il fournit une API pour exprimer le calcul de graphes qui peut modéliser les graphes définis par l'utilisateur à l'aide de l'API d'abstraction Pregel. Il fournit également un runtime optimisé pour cette abstraction. En plus de ces librairies, on peut citer BlinkDB et Tachyon :

- ✓ *BlinkDB :* est un moteur de requêtes approximatif qui peut être utilisé pour exécuter des requêtes SQL interactives sur des volumes des données importants. Il permet à l'utilisateur de troquer la précision contre le temps de réponse. Il fonctionne en exécutant les requêtes sur des extraits des données et présente ses résultats accompagnés d'annotations avec les indicateurs d'erreurs significatifs.
- ✓ *Tachyon :* est un système de fichiers distribué qui permet de partager des fichiers de façon fiable à la vitesse des accès en mémoire à travers des Framework de clusters comme Spark et MapReduce. Il évite les accès disques et le chargement des fichiers fréquemment utilisés en les cachant en mémoire. Ceci permet aux divers Framework, tâches et requêtes d'accéder aux fichiers en cache rapidement.

f) Architecture de Spark

L'architecture de Spark comprend les trois composants principaux suivants :

- **Le stockage des données**

Spark utilise le système de fichiers HDFS pour le stockage des données. Il peut fonctionner avec n'importe quelle source des données compatible avec Hadoop, dont HDFS, HBase, Cassandra, etc.

- **L'API**

L'API permet aux développeurs de créer des applications Spark en utilisant une API standard. L'API existe en Scala, Java et Python.

- **Gestion des ressources**

Spark peut être déployé comme un serveur autonome ou sur un Framework de traitements distribués comme Mesos ou YARN. La figure 2 illustre les composants du modèle d'architecture de Spark.

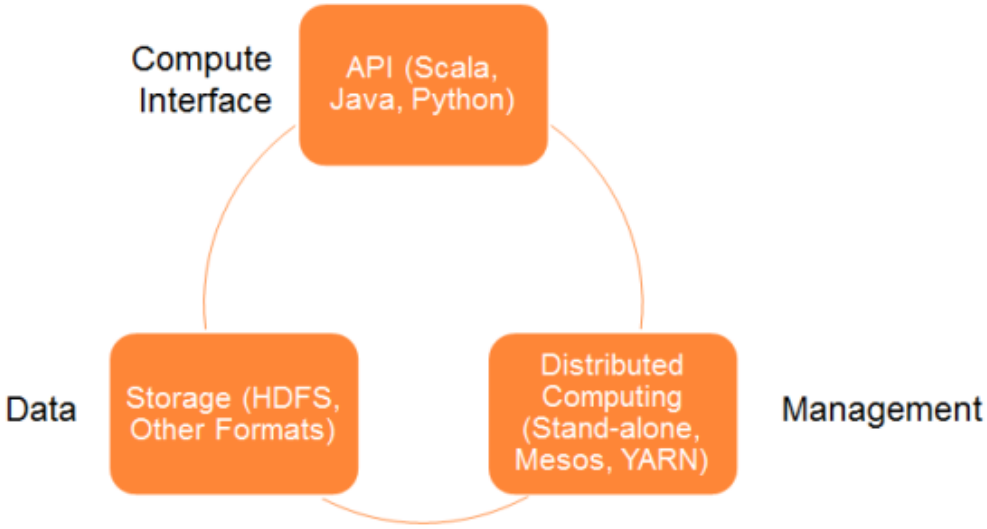

Figure 19: Architecture de Spark

- ***Les "Resilient Distributed Datasets"***

Les Resilient Distributed Datasets (basés sur la publication de recherche de Matei), ou RDD, sont un concept au cœur du Framework Spark. Vous pouvez voir un RDD comme une table dans une base des données. Celui-ci peut porter tout type des données et la stocké par Spark sur différentes partitions. Les RDD permettent de réarranger les calculs et d'optimiser le traitement. Ils sont aussi tolérants aux pannes car un RDD sait comment recréer et recalculer son ensemble des données. Les RDD sont immutables. Pour obtenir une modification d'un RDD, il faut y appliquer une transformation, qui retournera un nouveau RDD, l'original restera inchangé. Les RDD supportent deux types d'opérations :

- ✓ *Les transformations* **:** les transformations ne retournent pas de valeur seule, elles retournent un nouveau RDD. Rien n'est évalué lorsque l'on fait appel à une fonction de transformation, cette fonction prend juste un RDD et retourne un nouveau RDD. Les fonctions de transformation sont par exemple map, filter, flatMap, groupByKey, reduceByKey, aggregateByKey, pipe et coalesce.
- ✓ *Les actions* **:** les actions évaluent et retournent une nouvelle valeur. Au moment où une fonction d'action est appelée sur un objet RDD, toutes les requêtes de traitement des données sont calculées et le résultat est retourné. Les actions sont par exemple reduce, collect, count, first, take, countByKey et foreach.

4. Etude

a) Introduction

Afin de montrer la différence entre Hadoop et Spark, un travail est fait dans un cluster avec huit machines virtuelles où Hadoop et Spark sont installés et déployés. Trois études de cas basées sur le même algorithme et langage de programmation sont utilisés pour fonctionner sur ce cluster. Les temps de course de chaque étude de cas sur le système Hadoop et le système Spark sont présentés pour montrer la performance différente.

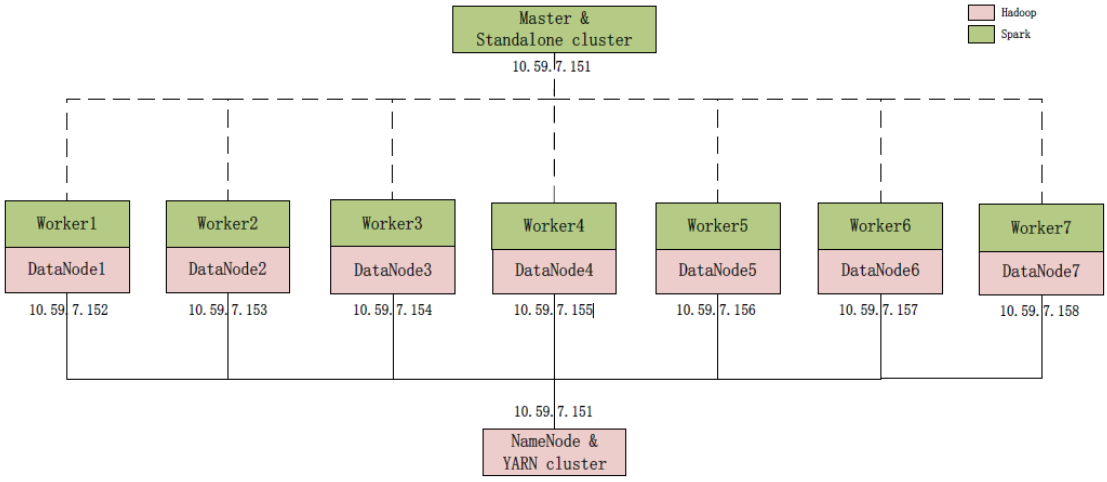

Figure 20: Schéma du réseau

Dans ce travail, Hadoop et Spark sont déployés sur 8 machines virtuelles. Le maître l'IP du nœud est 10.59.7.151, et les autres nœuds esclaves ou travailleurs sont de 10.59.7.152 à 10.59.7.158.

Pour le système Hadoop, le processus de Namenode et le gestionnaire de cluster YARN sont lancés sur le nœud maître, et chaque nœud esclave est responsable du lancement de son propre nœud des données processus. Pour le système Spark, le processus maître et le cluster autonome intégré sont démarré sur le nœud maître, chaque travailleur est responsable du lancement du processus exécute.

Software	Version
OS	Ubuntu 12.04, 32-bit
Apache Hadoop	2.7.1
Apache Spark	1.4
JRE	Java (TM) SE Runtime Environment (build 1.8.0_66- b17)
SSH	Openssh_5.9p1
Platform de virtualisation	VMware vSphere 5.5
Storage size dans chaque machine : 100G	
Memory size dans chaque Machine : 6G	

Tableau 1: Performances des machines

❖ *Étude de cas*

Quelques cas sont utilisés pour l'évaluation dans ce travail. Chaque cas utilise différents nombres d'itérations, mais les exemples de chaque cas exécuté sur Hadoop et Spark sont basés sur le même algorithme. Dans ce cas, avec les mêmes configurations de matériel une comparaison des performances devrait être réalisable et raisonnable.

b) Word Count-Trie par clés

L'exemple classique du nombre de mots est fourni pour Hadoop et Spark, ce qui fait utiliser MapReduce pour faire le travail. La source des données est générée par un programme qui choisit au hasard des mots dans un fichier de dictionnaire qui comprend 5000 mots anglais, et place un mot sur chaque ligne du fichier de sortie. Le tableau 2 suivant montre les données taille utilisée dans cette étude de cas et la prochaine étude de cas Nombre de mots – triés par valeurs.

Nom	Taille
Wc_100m.txt	99,96MB
Wc_500m.txt	499,72MB
Wc_1g.txt	999,44MB
Wc_2g.txt	1,95GB

Tableau 2: Taille des fichiers

❖ **Data simple**

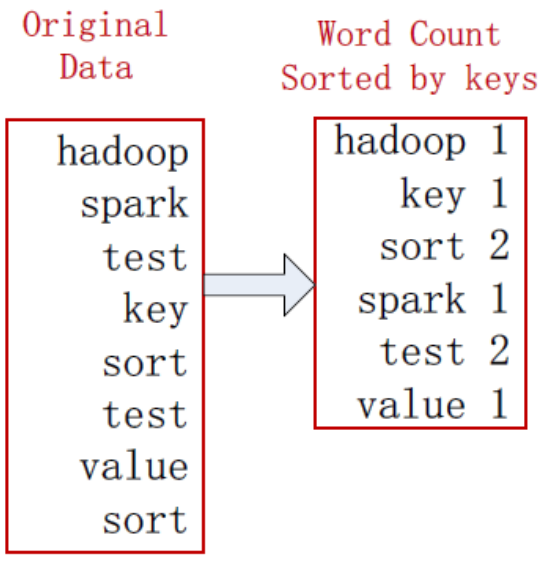

Figure 21: Exemple de Word Count triés par valeurs

❖ **Description de l'algorithme** :

L'algorithme 1 de la figure 22 montre comment l'étude de cas Word Count est implémentée dans Hadoop. Au début, la fonction de Mapper divise une ligne à la fois en mots à l'aide d'un jeton, puis génère des paires valeur / clé au format <<Mot>, 1> comme données d'entrée de la fonction de réduction. Avant d'appeler la fonction Reducer, les clés sont triées avec l'ordre du dictionnaire par défaut. Ensuite, la fonction Reducer résume le nombre de chaque mot unique. Enfin, la réduction La fonction produit le résultat sur HDFS.

L'algorithme 2 de la figure 22 montre comment l'étude de cas Word Count est implémentée dans Spark. Dans un premier temps, un RDD est créé en chargeant des données à partir de HDFS en utilisant la fonction textFile(). Ensuite, quelques fonctions de transformation, telles que flatMap(), map() et ReduceByKey (), sont invoquées pour enregistrer les métadonnées sur la façon de traiter les données réelles. Enfin, toutes les transformations sont appelées pour calculer les données immédiates réelles une fois qu'une action comme la fonction saveAsText() est appelée.

Algorithm 1: Word Count in Hadoop

```
1: class Mapper<K1,V1,K2,V2>
2:     function map(K1,V1)
3:         List words = V1.splitBy(token);
4:         foreach K2 in words
5:             write(K2,1);
6:         end for
7:     end function
8: end class
```

```
1: class Reducer<K2,V2,K3,V3>
2:     function reduce(K2,Iterable<V2> itor)
3:         sum = 0;
4:         foreach count in itor
5:             sum += count;
6:         end for
7:         write(k3,sum);
8:     end function
9: end class
```

Algorithm 2: Word Count in Spark

```
1: class WordCount
2:     function main(String[] args)
3:         file = sparkContext.textFile(filePath);
4:         JavaRDD<String> words = flatMap <- file;
5:         JavaPairRDD<String,Integer> pairs = map <- words;
6:         JavaPairRDD<String,Integer> counts = reduceByKey <- pairs
7:         result = sortByKey <- counts;
8:     end function
9: end class
```

Figure 22: Algorithme de Word Count

c) Word Count-Trie par valeurs

Par défaut, la sortie de l'exemple de Word Count est triée par clé avec ordre. Dans l'exemple suivant de la figure 23, la fonction de tri par défaut sera remplacée par une fonction de tri entier pour trier les valeurs. Comme le montre la figure 23, il y a trois Job qui complètent l'ensemble du programme. Le premier Job produit les mêmes données immédiates comme ce que fait la première étude de cas. Le deuxième Job échange la valeur-clé paires, puis utilise la fonction de tri des nombres entiers pour trier les fréquences. Enfin, le troisième Job consiste à regrouper les données immédiates par fréquence, puis à les trier par mots.

❖ **Data simple**

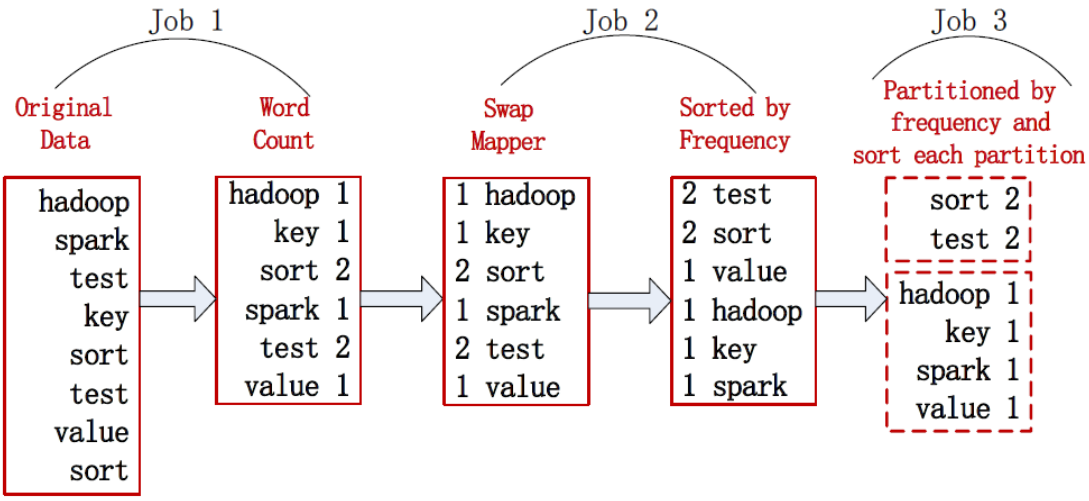

Figure 23: Word Count-Trie par valeurs

❖ **Description de l'algorithme**

- ✓ Sur la base de l'étude de cas du nombre de mots, échangez la clé et la valeur de Mapper
- ✓ Remplacez le comparateur de Mapper et triez la fréquence avec le type entier
- ✓ Partitionnez le résultat de course par fréquence
- ✓ Échangez la clé et la valeur de chaque partition
- ✓ Trier chaque partition par mots
- ✓ Fusionnez la partition
- ✓ Enfin, sortez le résultat

d) Algorithme itératif

Ici, PageRank est choisi pour montrer la différence de performances entre Hadoop et Spark pour les raisons suivantes :

- ✓ La mise en œuvre de l'algorithme PageRank est impliquée dans plusieurs itérations de Calcul.
- ✓ Dans Hadoop, pour chaque calcul d'itération, MapReduce écrit toujours immédiatement des données vers HDFS (système de fichiers distribué Hadoop) après une carte ou une action de réduction 3 Cependant, Spark traite les données immédiates dans un cache en mémoire.

❖ **Description de l'algorithme**

PageRank apparaît avec le développement de moteurs de recherche Web. C'est considéré comme la probabilité qu'un utilisateur, qui reçoit une page au hasard et clique sur des liens au hasard tout le temps, finit par s'ennuyer et passe à une autre page au hasard. Comme résultat, il est utilisé pour calculer un classement de qualité pour chaque page dans la

structure de liens du Web, et améliore ainsi la précision des résultats de recherche. C'est ainsi que Google Le moteur de recherche évalue la qualité des pages Web.

Fondamentalement, l'idée centrale du PageRank est la suivante :

- ✓ Si une page est liée par un grand nombre d'autres pages, il est beaucoup plus important qu'une page liée par quelques autres, et cette page possède également une valeur de rang plus élevée
- ✓ Si une page est liée par une autre page avec une valeur de classement plus élevée, sa valeur de classement est améliorée
- ✓ Le but final de cet algorithme est de trouver des rangs stables pour tous les liens après plusieurs itérations

Dans cette étude de cas, une valeur de PageRank sera calculée par la formule suivante, qui a été proposée par les fondateurs de Google Brin et Page en 1998 :

$$R_i = d * \sum_{j \in S} (R_j / N_j) + (1 - d)$$

Figure 24: Formulaire de PageRank

R_i : La valeur PageRank du lien i
R_j : La valeur PageRank du lien j
N_j : Le nombre de liens sortants du lien j pointant vers ses liens voisins
S : L'ensemble des liens qui pointent vers le lien i
d : Le facteur d'amortissement- The damping factor - (généralement, d = 0,85).

e) Exemple de préparation des données et d'exécution des résultats

Les exemples de jeux des données du tableau 3 sont utilisés pour évaluer les performances de l'Application PageRank fonctionnant respectivement dans Hadoop et Spark. Toutes les données la source provient de http://snap.stanford.edu/data/

Name	Taille	Nœuds	Bords
web-NotreDame.txt	20.56MB	325,729	1,497,134
web-Google.txt	73.6MB	875,713	5,105,039
as-Skitter.txt	142.2MB	1,696,415	11,095,298

Tableau 3: Taille des fichiers

❖ **Data simple**

L'exemple des données illustré à la figure 24 provient de web-Google.txt. Chaque ligne représente un bord dirigé. La colonne de gauche représente les points de liaison de départ et la colonne de droite est les points de liaison de fin.

From LinkId	To LinkId
0	11342
0	824020
0	867932
11342	0
11342	27469
11342	23689
11342	867932
824020	0
824020	91807

Figure 25: Exemple des données illustré

Prenons quelques exemples des données pour montrer comment les valeurs du PageRank sont calculé. Initialement, la valeur du PageRank de chaque lien est égale à 1.0, puis leur valeur finale les valeurs sont calculées comme suit :

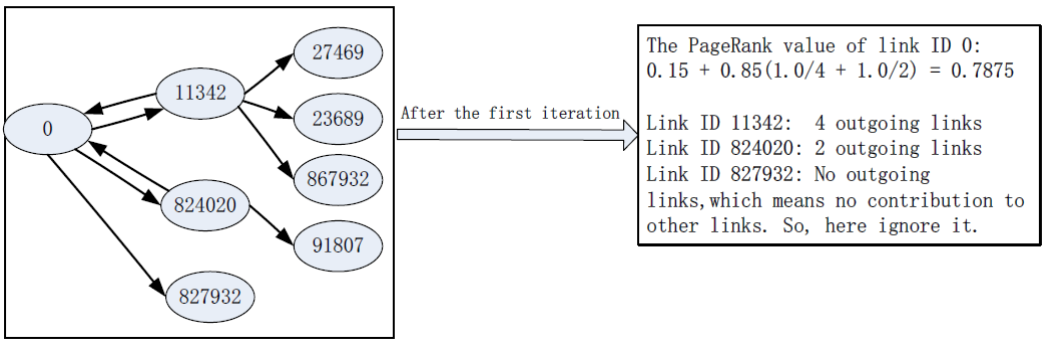

Figure 26: Calcule du PageRank

f) L'évaluation des résultats de course

❖ **Mesure du rendement et mesures**

Pour les trois études de cas ci-dessus, leurs performances dans Hadoop et dans Spark sont comparées par le temps de fonctionnement. Pour garder une comparaison équitable, nous garantissons les mesures suivantes qui sont appliquées à Hadoop et Spark :

- ✓ Les plates-formes Hadoop et Spark s'exécutent sur les mêmes machines de cluster

- ✓ Hadoop et Spark utilisent HDFS comme système de stockage de fichiers
- ✓ Les études de cas implémentées dans Hadoop et Spark sont basées sur la même programmation langage et algorithme
- ✓ Enfin, nous profitons de l'interface utilisateur Web de l'application Hadoop et de l'application Spark Interface Web où l'heure de début et l'heure de fin sont répertoriées pour calculer le temps écoulé moment d'une application Hadoop ou Spark, comme le montrent les figures 27 et 28 :

Running Applications

Application ID	Name	Cores	Memory per Node	Submitted Time	User	State	Duration

Completed Applications

Application ID	Name	Cores	Memory per Node	Submitted Time	User	State	Duration
app-20160124174056-0002	PageRank	14	3.0 GB	2016/01/24 17:40:56	student	FINISHED	1.1 min
app-20160124173957-0001	WordCount Secondary Sort	14	3.0 GB	2016/01/24 17:39:57	student	FINISHED	23 s
app-20160124173913-0000	WordCount	14	3.0 GB	2016/01/24 17:39:13	student	FINISHED	21 s

Figure 27: Durée du Test

Show 20 entries

ID	User	Name	Application Type	Queue	StartTime	FinishTime	State
application_1453677815715_0002	student	WCSortValues for wc_500M.txt	MAPREDUCE	default	Sun Jan 24 17:27:23 -0600 2016	Sun Jan 24 17:28:41 -0600 2016	FINISHED
application_1453677815715_0001	student	WordCount for wc_500M.txt	MAPREDUCE	default	Sun Jan 24 17:25:02 -0600 2016	Sun Jan 24 17:26:26 -0600 2016	FINISHED

Figure 28: Temps de début et de Fin

❖ La comparaison des résultats de course

Dans cette recherche, chaque étude de cas a été répétée plus de 10 fois les tests pour obtenir les résultats de fonctionnement moyens. Parfois, en raison du trafic réseau instable, il y a quelques secondes de bande d'erreur pour un petit travail, ou des dizaines de secondes de bande d'erreur pour un gros travail. Les tableaux 4, 5 et 6 présentent la comparaison du temps de fonctionnement moyen basée sur différentes tailles des données pour chaque étude de cas dans Hadoop et dans Spark. Les observations sont les suivantes :

- ✓ Dans la première étude de cas Word Count, pour une petite taille des données inférieure à la taille de bloc par défaut de 128 Mo, il existe un rapport de performances stable entre Spark et Hadoop car les données sont traitées dans le nœud local, que ce soit dans Hadoop ou dans Spark. Cependant, à mesure que la taille des données augmente, c'est-à-dire que davantage de blocs divisés sont générés, il y a un rapport de performance croissant entre Spark et Hadoop. Ici, le rapport de performance (PR) est défini comme :

$$PR = \frac{Le\ temps\ d'exécution\ d'une\ taille\ de\ données\ donnée\ dans\ Spark}{Le\ temps\ d'exécution\ de\ la\ même\ taille\ de\ données\ dans\ Hadoop}$$

Data size / System	100 MB	500 MB	1 GB	2 GB
Hadoop	58 secs	1 min 12 secs	1 min 48 secs	2 mins 25 secs
Spark	16 secs	23 secs	25 secs	30 secs
	PR=3.63	PR=3.13	PR=4.32	PR=4.83

Tableau 4: Résultat de temps de travail pour Word Count

- ✓ Dans la deuxième étude de cas, Word Count – triée par valeur, la valeur du rapport de performance est supérieure à celle de la première étude de cas car il existe plusieurs itérations.

Data size / System	100 MB	500 MB	1 GB	2GB
Hadoop	1 min 43 secs	1 min 56 secs	2 mins 27 secs	3 mins 2 secs
Spark	12 secs	22 secs	23 secs	30 secs
	PR=8.58	PR=5.27	PR=6.39	PR=6.07

Tableau 5: Résultat de temps de travail pour Word Count pour la deuxième trie

- ✓ Lorsque plusieurs itérations, telles que 15 itérations, s'appliquent respectivement à Hadoop et Spark, Spark a une amélioration des performances convaincante par rapport à Hadoop.

Data size / System	20.56 MB NotreDame	67.02 MB Google	145.62 MB as-skitter
Hadoop	7 mins 22 secs	15 mins 3 secs	38 mins 51 secs
Spark	37 secs	1 min 18 secs	2 mins 48 secs
	PR = 11.95	PR = 11.58	PR=13.86

Tableau 6: Résultat de temps de travail pour PageRank

Comme mentionné précédemment, Spark utilise un stockage basé sur la mémoire pour les RDD, mais MapReduce dans Hadoop traite les opérations sur disque, il va donc de soi que les performances de Spark surpassent celles d'Hadoop. Cependant, Spark permet de limiter l'utilisation de la mémoire de chaque exécuteur en affectant spark.executor.memory à une valeur appropriée, par exemple 2 Go. Par conséquent, comme la limite d'utilisation de la mémoire varie entre 1 et 3 Go sur chaque exécuteur, des résultats d'exécution comparables sont répertoriés dans les tableaux 7, 8 et 9 (unité de temps : secondes), et nous avons les observations suivantes :

- ✓ Pour une petite taille des données ou moins d'itérations, l'augmentation de la mémoire ne contribue pas à l'amélioration des performances, comme le montrent les tableaux 7 et 8.
- ✓ À mesure que la croissance de la taille des données et plusieurs itérations sont exécutées, il y a une amélioration significative des performances avec l'augmentation de l'utilisation de la mémoire en définissant la valeur de spark.executor.memory, comme indiqué dans le tableau 8.

Data size / Memory usage (GB)	100 MB	500MB	1GB	2GB
3	16 secs	24 secs	25 secs	29 secs
2	16 secs	24 secs	24 secs	30 secs
1	16 secs	23 secs	25 secs	31 secs

Tableau 7: Temps d'exécution pour Word Count - triés par clés sur Spark

Data size / Memory usage (GB)	100 MB	500 MB	1 GB	2 GB
3	16 secs	24 secs	25 secs	31 secs
2	17 secs	23 secs	27 secs	29 secs
1	15 secs	23 secs	24 secs	31 secs

Tableau 8: Temps d'exécution pour Word Count - triés par valeurs sur Spark

Data size / Memory usage (GB)	20.56 MB NotreDame	67.02 MB Google	145.62 MB as-skitter
3	36 secs	78 secs	162 secs
2	37 secs	78 secs	180 secs
1	33 secs	114 secs	780 secs

Tableau 9: Temps de fonctionnement de PageRank sur Spark

Basé sur la même utilisation de la mémoire, Spark fonctionne toujours mieux que Hadoop (la mémoire par défaut pour une tâche de carte est de 1 Go). Les raisons résultent principalement des facteurs suivants :

- ✓ Les charges de travail Spark ont un nombre d'accès au disque par seconde plus élevé que celui de Hadoop
- ✓ Spark a une meilleure utilisation de la bande passante mémoire que Hadoop
- ✓ Spark atteint des IPC plus élevés que Hadoop

De plus, dans Spark, la planification des tâches est basée sur un mode piloté par les événements, mais Hadoop utilise des pulsations pour suivre les tâches, ce qui entraîne périodiquement des retards de quelques secondes.

De plus, dans Hadoop, il y a des frais généraux pour répondre aux exigences minimales de configuration des tâches, de démarrage des tâches et de nettoyage en raison de la charge minimale de la pile logicielle Hadoop.

Pour certaines applications impliquées dans l'algorithme itératif, Hadoop est totalement submergé par Spark car plusieurs travaux dans Hadoop ne peuvent pas partager des données et doivent accéder fréquemment à HDFS.

g) Conclusion

D'après ce travail, nous constatons que Spark surpasse totalement Hadoop sur les performances dans toutes les études de cas, en particulier celles impliquées dans l'algorithme itératif. Nous concluons que plusieurs facteurs peuvent entraîner une différence de performance significative.

5. *Différence entre Spark et Hadoop*

a) Hadoop et Spark font des choses différentes

Tous deux sont des Framework big data, mais ils n'ont pas vraiment le même usage. Hadoop est essentiellement une infrastructure des données distribuées : ce Framework Java libre distribue les grandes quantités des données collectées à travers plusieurs nœuds (un cluster de serveurs x86), et il n'est donc pas nécessaire d'acquérir et de maintenir un hardware spécifique et coûteux. Hadoop est également capable d'indexer et de suivre ces données big data, ce qui facilite grandement leur traitement et leur analyse par rapport à ce qui était possible auparavant. Comparativement, Spark sait travailler avec des données distribuées. Mais il ne sait pas faire du stockage distribué. Il a donc besoin de s'appuyer sur un système de stockage distribué.

b) Il est possible d'utiliser Hadoop indépendamment de Spark et réciproquement

Hadoop a un composant de stockage HDFS (Hadoop Distributed File System), et l'outil de traitement MapReduce. De fait, il n'est pas nécessaire de faire appel à Spark pour traiter ses données Hadoop. Et inversement, il est possible d'utiliser Spark sans faire intervenir Hadoop. Spark n'a pas de système de gestion de fichiers propre, ce qui veut dire qu'il faut lui associer un système de fichiers - soit HDFS, soit celui d'une autre plate-forme des données dans le cloud. Néanmoins, Spark a été conçu pour Hadoop, et la plupart des gens s'accordent pour dire qu'ils fonctionnent mieux ensemble.

c) Vitesse

Spark utilise de la mémoire et peut utiliser le disque pour le traitement des données qui ne tiennent pas toutes en mémoire, alors que MapReduce est strictement basé sur le disque. La principale différence entre MapReduce et Spark est que MapReduce utilise un stockage persistant et Spark utilise des ensembles des données distribués résilients (RDD), qui sont traités plus en détail dans la section Tolérance aux pannes.

Le mode de fonctionnement de MapReduce peut être suffisant si les besoins opérationnels et les besoins de rapports sont essentiellement statiques et s'il est possible d'attendre la fin du traitement des lots. Mais si l'on a besoin d'analyser des données en streaming, comme c'est le cas pour traiter des données remontées par capteurs dans une usine, ou si les applications nécessitent une succession d'opérations, il faudra probablement faire appel à Spark. C'est le cas de la plupart des algorithmes d'apprentissage machine qui ont besoin d'effectuer des opérations

multiples. Spark est tout à fait adapté pour les campagnes de marketing en temps réel, les recommandations de produits en ligne, la cybersécurité et la surveillance des logs machine.

d) Tolérance aux pannes

Pour la tolérance aux pannes, MapReduce et Spark résolvent le problème dans deux directions différentes. MapReduce utilise des TaskTrackers qui fournissent des pulsations au JobTracker. Si une pulsation est manquée, le JobTracker replanifie toutes les opérations en attente et en cours vers un autre TaskTracker. Cette méthode est efficace pour fournir une tolérance aux pannes, mais elle peut augmenter considérablement les délais de réalisation des opérations qui ont même une seule défaillance.

Spark utilise des ensembles des données distribués résilients (RDD), qui sont des ensembles d'éléments tolérants aux pannes pouvant être exploités en parallèle. Les RDD peuvent référencer un ensemble des données dans un système de stockage externe, tel qu'un système de fichiers partagé, HDFS, HBase ou toute source des données offrant un Hadoop InputFormat. Spark peut créer des RDD à partir de n'importe quelle source de stockage prise en charge par Hadoop, y compris les systèmes de fichiers locaux ou l'un de ceux répertoriés précédemment.

e) Frais

MapReduce et Spark sont des projets Apache, ce qui signifie qu'il s'agit de logiciels libres et open source. Bien qu'il n'y ait aucun coût pour le logiciel, il existe des coûts associés à l'exécution de l'une ou l'autre plate-forme en personnel et en matériel. Les deux produits sont conçus pour fonctionner sur du matériel de base, tel que des systèmes de serveurs à bas prix, appelés boîtes blanches.

MapReduce utilise des quantités standard de mémoire car son traitement est basé sur le disque, donc une entreprise devra acheter des disques plus rapides et beaucoup d'espace disque pour exécuter MapReduce. MapReduce nécessite également davantage de systèmes pour distribuer les E / S disque sur plusieurs systèmes.

Il est vrai, cependant, que les systèmes Spark coûtent plus cher en raison des grandes quantités de RAM requises pour tout exécuter en mémoire. Mais ce qui est également vrai, c'est que la technologie Spark réduit le nombre de systèmes requis. Ainsi, vous disposez de beaucoup moins de systèmes qui coûtent plus cher. Il y a probablement un moment où Spark réduit réellement les coûts par unité de calcul, même avec l'exigence supplémentaire de RAM.

f) Compatibilité

MapReduce et Spark sont compatibles l'un avec l'autre et Spark partage toutes les compatibilités de MapReduce pour les sources des données, les formats de fichiers et les outils de Business Intelligence via JDBC et ODBC.

g) Traitement DATA

MapReduce est un moteur de traitement par lots. MapReduce fonctionne par étapes séquentielles en lisant les données du cluster, en effectuant son opération sur les données, en réécrivant les résultats dans le cluster, en lisant les données mises à jour du cluster, en effectuant l'opération des données suivante, en réécrivant ces résultats dans le cluster, etc…

Spark effectue des opérations similaires, mais il le fait en une seule étape et en mémoire. Il lit les données du cluster, effectue son opération sur les données, puis les réécrit dans le cluster.

Spark comprend également sa propre bibliothèque de calcul de graphes, GraphX. GraphX permet aux utilisateurs de visualiser les mêmes données que des graphiques et des collections. Les utilisateurs peuvent également transformer et joindre des graphiques avec des ensembles des données distribués résilients (RDD), abordés dans la section Tolérance aux pannes.

h) Evolutivité

Par définition, MapReduce et Spark sont évolutifs à l'aide de HDFS. Yahoo aurait un cluster Hadoop à 42 000 nœuds, alors le ciel est peut-être vraiment la limite. Le plus grand cluster Spark connu est de 8 000 nœuds, mais à mesure que le Big Data se développe, la taille du cluster devrait augmenter pour maintenir les attentes de débit.

i) Sécurité

Hadoop prend en charge l'authentification Kerberos, ce qui est quelque peu pénible à gérer. Cependant, des fournisseurs tiers ont permis aux organisations de tirer parti d'Active Directory Kerberos et LDAP pour l'authentification. Ces mêmes fournisseurs tiers proposent également le chiffrement des données en vol et des données au repos.

Le système de fichiers distribués de Hadoop prend en charge les listes de contrôle d'accès (ACL) et un modèle traditionnel d'autorisations de fichiers. Pour le contrôle des utilisateurs dans la soumission des travaux, Hadoop fournit une autorisation de niveau de service, qui garantit que les clients disposent des autorisations appropriées.

La sécurité de Spark est un peu rare en ne prenant actuellement en charge que l'authentification via un secret partagé (authentification par mot de passe). Le bonus de sécurité dont Spark peut bénéficier est que si vous exécutez Spark sur HDFS, il peut utiliser des ACL HDFS et des autorisations au niveau des fichiers. De plus, Spark peut fonctionner sur YARN, ce qui lui permet d'utiliser l'authentification Kerberos.

j) Machine Learning

Hadoop utilise Mahout pour le traitement des données. Mahout inclut le clustering, la classification et le filtrage collaboratif basé sur des lots, qui s'exécutent tous sur MapReduce. Cela est progressivement abandonné en faveur de Samsara, un langage DSL soutenu par Scala qui permet des opérations en mémoire et algébriques, et permet aux utilisateurs d'écrire leurs propres algorithmes.

Spark dispose d'une bibliothèque d'apprentissage automatique, MLLib, utilisée pour les applications itératives d'apprentissage automatique en mémoire. Il est disponible en Java, Scala, Python ou R, et comprend la classification et la régression, ainsi que la possibilité de créer des pipelines d'apprentissage automatique avec un réglage hyperparamétrique.

k) Résumer

La comparaison principale entre Hadoop et Spark est discutée ci-dessous

Base de comparaison	Hadoop	Spark
Catégorie	Moteur de traitement des données de base	Moteur d'analyse des données
Usage	Traitement par lots avec un énorme volume des données	Traitez les données en temps réel, à partir d'événements en temps réel comme Twitter, Facebook
Latence	Informatique à latence élevée	Informatique à latence faible
Data	Traiter les données en mode batch	Peut traiter de manière interactive
Facilité d'utilisation	Le modèle MapReduce de Hadoop est complexe, il faut gérer les API de bas niveau	Plus facile à utiliser, l'abstraction permet à un utilisateur de traiter des données à l'aide d'opérateurs de haut niveau
Planificateur	Planificateur de travaux externe requis	Calcul en mémoire, aucun planificateur externe requis

Sécurité	Hautement sécurisé	Moins sûr que Hadoop
Coût	Moins coûteux car le modèle MapReduce offre une stratégie moins chère	Plus coûteux que Hadoop car il dispose d'une solution en mémoire

Tableau 10: Hadoop VS Spark

Conclusion

À première vue, il semble que l'utilisation de Spark soit le choix par défaut pour toute application de Big Data. Mais ce n'est pas le cas. MapReduce a fait son entrée sur le marché des mégadonnées pour les entreprises qui ont besoin d'énormes ensembles des données contrôlés par les systèmes de produits. La vitesse, l'agilité et la relative facilité d'utilisation de Spark complètent parfaitement le faible coût d'exploitation de MapReduce.

La vérité est que Spark et MapReduce ont une relation symbiotique l'un avec l'autre. Hadoop fournit des fonctionnalités que Spark ne possède pas, comme un système de fichiers distribué et Spark fournit un traitement en temps réel en mémoire pour les ensembles des données qui en ont besoin. Le scénario parfait du Big Data est exactement comme les concepteurs l'ont voulu : Hadoop et Spark travaillent ensemble au sein de la même équipe.

Références

https://en.wikipedia.org/wiki/Big_data

https://www.axess.fr/definition-big-data-3v/

https://www.lebigdata.fr/definition-big-data

https://www.lcl.fr/mag/tendances/big-data-definition-enjeux-et-applications

https://www.upwarddata.fr/nos-conseils/article/grands-domaines-de-la-data/

https://www.educba.com/hadoop-vs-spark/

https://itsocial.fr/articles-decideurs/article/quest-big-data-7-meilleurs-articles-big-data/

https://www.datamation.com/data-center/hadoop-vs.-spark-the-new-age-of-big-data.html

https://logz.io/blog/hadoop-vs-spark/

https://hadoop.apache.org

https://www.guru99.com/learn-hadoop-in-10-minutes.html

https://www.tutorialspoint.com/apache_spark/apache_spark_introduction.htm

https://data-flair.training/blogs/spark-tutorial/

https://www.infoq.com/fr/articles/apache-spark-introduction/

https://comarketing-news.fr/big-data-un-marche-a-plus-de-200-milliards-des-2020/

Code Source d'Etude

❖ Source Code : WordCountHadoop.java

```
1.  /**
2.    * Copyright [2015] [Shengti Pan]
3.    *
4.    * Licensed under the Apache License, Version 2.0 (the "License");
5.    * you may not use this file except in compliance with the License.
6.    * You may obtain a copy of the License at
7.    *
8.    *   http://www.apache.org/licenses/LICENSE-2.0
9.    *
10.   * Unless required by applicable law or agreed to in writing, software
11.   * distributed under the License is distributed on an "AS IS" BASIS,
12.   * WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
13.   * See the License for the specific language governing permissions and
14.   * limitations under the License.
15.   */
16.
17.
18. import java.io.IOException;
19. import java.util.StringTokenizer;
20. import org.apache.hadoop.conf.Configuration;
21. import org.apache.hadoop.fs.Path;
22. import org.apache.hadoop.io.IntWritable;
23. import org.apache.hadoop.io.Text;
24. import org.apache.hadoop.mapreduce.Job;
25. import org.apache.hadoop.mapreduce.Mapper;
26. import org.apache.hadoop.mapreduce.Reducer;
27. import org.apache.hadoop.mapreduce.lib.input.FileInputFormat;
28. import org.apache.hadoop.mapreduce.lib.output.*;
29. import org.apache.hadoop.io.compress.*;
30.
31. /**
32. * This Hadoop program is to implement counting the frequency
33. * of words in a text file which is stored in HDFS.
34.  */
35.
36. public class WordCountHadoop {
37.
38.     private final static String rootPath = "/user/hadoop/";
39.
```

```
40.  //map each word to a value one
41.  public static class TokenizerMapper extends
42.        Mapper<Object, Text, Text, IntWritable> {
43.     private final static IntWritable one = new IntWritable(1);
44.     private Text word = new Text();
45.     public void map(Object key, Text value, Context context)
46.           throws IOException, InterruptedException {
47.        StringTokenizer itr = new StringTokenizer(value.toString());
48.        while (itr.hasMoreTokens()) {
49.           word.set(itr.nextToken());
50.           context.write(word, one);
51.        }
52.     }
53.  }
54.
55.  //reduce values by a unique word
56.  public static class IntSumReducer extends
57.        Reducer<Text, IntWritable, Text, IntWritable> {
58.     private IntWritable result = new IntWritable();
59.     public void reduce(Text key, Iterable<IntWritable> values,
60.           Context context) throws IOException, InterruptedException {
61.        int sum = 0;
62.        for (IntWritable val : values) {
63.           sum += val.get();
64.        }
65.        result.set(sum);
66.        context.write(key, result);
67.     }
68.  }
69.
70.  public static void main(String[] args) throws Exception {
71.     //this program accepts two parameters by default;
72.     //if there is a third paramter, it is treated as the number of the reducers
73.     if(args.length < 2 || args.length > 3){
74.        System.out.println("Usage: wc.jar <input file> <output file> or");
75.        System.out.println("wc.jar <input file> <output file> <reduce number>");
76.        System.exit(1);
77.     }
78.
79.     //set up Hadoop configuration
80.     Configuration conf = new Configuration();
81.
82.     //set the compression format for map output
83.     conf.setBoolean(Job.MAP_OUTPUT_COMPRESS,true);
```

```
84.         conf.setClass(Job.MAP_OUTPUT_COMPRESS_CODEC,GzipCodec.class,
85.                                         CompressionCodec.class);
86.
87.         //create a Hadoop job
88.         Job job = Job.getInstance(conf, "WordCount for " + args[0]);
89.         job.setJarByClass(WordCountHadoop.class);
90.         job.setMapperClass(TokenizerMapper.class);
91.         job.setCombinerClass(IntSumReducer.class);
92.         job.setReducerClass(IntSumReducer.class);
93.         job.setOutputKeyClass(Text.class);
94.         job.setOutputValueClass(IntWritable.class);
95.
96.         //set the number of reducers. By default, No. of reducers = 1
97.         if (args.length == 3) {
98.            job.setNumReduceTasks(Integer.parseInt(args[2]));
99.         }
100.          FileInputFormat.addInputPath(job, new Path(rootPath + "input/"
101.                  + args[0]));
102.          FileOutputFormat
103.                  .setOutputPath(job, new Path(rootPath + args[1]));
104.          System.exit(job.waitForCompletion(true) ? 0 : 1);
105.      }
106.  }
```

❖ Source Code : WordCountSpark.java

```
1.  /**
2.    * Copyright [2015] [Shengti Pan]
3.    *
4.    * Licensed under the Apache License, Version 2.0 (the "License");
5.    * you may not use this file except in compliance with the License.
6.    * You may obtain a copy of the License at
7.    *
8.    *   http://www.apache.org/licenses/LICENSE-2.0
9.    *
10.   * Unless required by applicable law or agreed to in writing, software
11.   * distributed under the License is distributed on an "AS IS" BASIS,
12.   * WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
13.   * See the License for the specific language governing permissions and
14.   * limitations under the License.
```

```
15.  */
16.
17. import scala.Tuple2;
18. import org.apache.spark.SparkConf;
19. import org.apache.spark.api.java.JavaPairRDD;
20. import org.apache.spark.api.java.JavaRDD;
21. import org.apache.spark.api.java.JavaSparkContext;
22. import org.apache.spark.api.java.function.FlatMapFunction;
23. import org.apache.spark.api.java.function.Function2;
24. import org.apache.spark.api.java.function.PairFunction;
25. import org.apache.spark.serializer.KryoRegistrator;
26. import java.util.Arrays;
27. import java.util.List;
28. import java.util.regex.Pattern;
29.
30.
31. /**
32. * This spark program is to implement counting the frequency
33. * of words in a text file which is stored in HDFS.
34.  */
35.
36. public class WordCountSpark {
37.     private static int num = 0;//the partition number
38.     //the path to access HDFS
39.     private final static String rootPath = "hdfs://10.59.7.151:9000/user/hadoop/";
40.
41.     public static void main(String[] args) throws Exception {
42.         //this program accepts two parameters by default;
43.         //if there is a third paramter, it is treated as the
44.         parameter of a partition number
45.         if(args.length < 2 || args.length > 3){
46.             System.out.println("Usage: wc.jar <input file> <output file> or");
47.             System.out.println("wc.jar <input file> <output file> <partition number>");
48.             System.exit(1);
49.         }
50.         if(args.length == 3)
51.             num = Integer.parseInt(args[2]);
52.
53.         //set up the configuration and context of this spark application
54.         SparkConf sparkConf = new SparkConf().setAppName("WordCountSpark");
55.         JavaSparkContext spark = new JavaSparkContext(sparkConf);
56.
57.         //words are split by space
58.         JavaRDD<String> textFile = spark
```

```
            .textFile(rootPath + "input/"
                  + args[0]);
      JavaRDD<String> words = textFile
            .flatMap(new FlatMapFunction<String, String>() {
               public Iterable<String> call(String s) {
                  return Arrays.asList(s.split(" "));
               }
            });

      //map each word to the value 1
      JavaPairRDD<String, Integer> wordsMap = words
            .mapToPair(new PairFunction<String, String, Integer>() {
               public Tuple2<String, Integer> call(String s) {
                  return new Tuple2<String, Integer>(s, 1);
               }
            });

      //reduce the value of a unique word
      JavaPairRDD<String, Integer> freqPair = wordsMap
            .reduceByKey(new Function2<Integer, Integer, Integer>() {
               public Integer call(Integer a, Integer b) {
                  return a + b;
               }
            });

      //if num == 0, using the default partition rule
      if(num == 0)
         freqPair.sortByKey().map(x -> x._1 + "\t" + x._2).
            saveAsTextFile(rootPath + args[1]);
      else
         //else manually assign the partition number
         freqPair.repartition(num).
            sortByKey().map(x -> x._1 + "\t" + x._2).
            saveAsTextFile(rootPath + args[1]);

      //terminate the spark application
      spark.stop();
   }
}
```

❖ Source Code : WCHadoopSortValues.java

```
1. /**
2.  * This Hadoop program is to implement a secondary sort
```

```
3.    * of the WordCount example, which means that the final
4.     * result is not only sorted by frequency, but also sorted
5.     * by words.
6.     */
7.
8. import java.util.*;
9. import java.io.*;
10. import org.apache.hadoop.fs.FileSystem;
11. import org.apache.hadoop.io.*;
12. import org.apache.hadoop.conf.Configuration;
13. import org.apache.hadoop.fs.Path;
14. import org.apache.hadoop.mapreduce.lib.input.*;
15. import org.apache.hadoop.mapreduce.lib.output.*;
16. import org.apache.hadoop.mapreduce.*;
17. import org.apache.hadoop.mapreduce.Partitioner;
18. import org.apache.hadoop.mapreduce.lib.partition.HashPartitioner;
19. import org.apache.hadoop.mapreduce.lib.map.InverseMapper;
20.
21. public class WCHadoopSortValues {
22.
23.     public static String rootPath = "/user/hadoop/";
24.
25.     public static class TokenizerMapper extends
26.             Mapper<Object, Text, Text, IntWritable> {
27.
28.         private final static IntWritable one = new IntWritable(1);
29.         private Text word = new Text();
30.
31.         // map each word to a value one
32.         public void map(Object key, Text value, Context context)
33.                 throws IOException, InterruptedException {
34.             StringTokenizer itr = new StringTokenizer(value.toString());
35.             while (itr.hasMoreTokens()) {
36.                 word.set(itr.nextToken());
37.                 context.write(word, one);
38.             }
39.         }
40.     }
41.
42.     // calculate the frequency of a unique word via reduce
43.     public static class IntSumReducer extends
44.             Reducer<Text, IntWritable, Text, IntWritable> {
45.         private IntWritable result = new IntWritable();
46.
```

```
47.     public void reduce(Text key, Iterable<IntWritable> values,
48.             Context context) throws IOException, InterruptedException {
49.         int sum = 0;
50.         for (IntWritable val : values) {
51.             sum += val.get();
52.         }
53.         result.set(sum);
54.         context.write(key, result);
55.     }
56. }
57.
58. // construct a map with a composite key, such as ((hadoop,1),null);
59. public static class SecondaryMapper extends
60.         Mapper<IntWritable, Text, CompositeKey, Text> {
61.     private Text word = new Text();
62.
63.     public void map(IntWritable value, Text key, Context context)
64.             throws IOException, InterruptedException {
65.         context.write(new CompositeKey((Text) key, value), word);
66.     }
67. }
68.
69. // implement a comparator for the comparison between two integers
70. private static class IntWritableComparator extends IntWritable.Comparator {
71.     public int compare(WritableComparable a, WritableComparable b) {
72.         return -super.compare(a, b);
73.     }
74.
75.     public int compare(byte[] b1, int s1, int l1, byte[] b2, int s2, int l2) {
76.         return -super.compare(b1, s1, l1, b2, s2, l2);
77.     }
78. }
79.
80. public static void main(String[] args) throws Exception {
81.     Configuration conf = new Configuration();
82.     Job job = Job.getInstance(conf, "WordCount for " + args[0]);
83.
84.     // save the immediate result into a temp file
85.     Path tempDir = new Path("temp_wc_" + System.currentTimeMillis());
86.     job.setJarByClass(WCHadoopSortValues.class);
87.     job.setMapperClass(TokenizerMapper.class);
88.     job.setCombinerClass(IntSumReducer.class);
89.     job.setReducerClass(IntSumReducer.class);
90.     job.setOutputKeyClass(Text.class);
```

```
91.         job.setOutputValueClass(IntWritable.class);
92.         FileInputFormat.addInputPath(job, new Path(rootPath + "input/"
93.                 + args[0]));
94.         FileOutputFormat.setOutputPath(job, tempDir);
95.         job.setOutputFormatClass(SequenceFileOutputFormat.class);
96.
97.         // order by frequency
98.         if (job.waitForCompletion(true)) {
99.             Job job2 = new Job(conf, "sorted by frequency");
100.            job2.setJarByClass(WCHadoopSortValues.class);
101.
102.            FileInputFormat.addInputPath(job2, tempDir);
103.            job2.setInputFormatClass(SequenceFileInputFormat.class);
104.
105.            job2.setMapperClass(InverseMapper.class);
106.            FileOutputFormat.setOutputPath(job2, new Path(args[1]));
107.
108.            job2.setOutputKeyClass(IntWritable.class);
109.            job2.setOutputValueClass(Text.class);
110.            job2.setSortComparatorClass(IntWritableComparator.class);
111.            FileSystem.get(conf).deleteOnExit(tempDir);
112.            tempDir = new Path("temp_wc_" + System.currentTimeMillis());
113.            FileOutputFormat.setOutputPath(job2, tempDir);
114.            job2.setOutputFormatClass(SequenceFileOutputFormat.class);
115.
116.            // order by word
117.            if (job2.waitForCompletion(true)) {
118.                Job job3 = new Job(conf, "sorted by word");
119.                job3.setJarByClass(WCHadoopSortValues.class);
120.
121.                FileInputFormat.addInputPath(job3, tempDir);
122.                job3.setInputFormatClass(SequenceFileInputFormat.class);
123.                job3.setMapperClass(SecondaryMapper.class);
124.
125.                // set parameters for the job3, such as paritioner and
126.                // comparator
127.                job3.setMapOutputKeyClass(CompositeKey.class);
128.                job3.setPartitionerClass(KeyPartitioner.class);
129.                job3.setSortComparatorClass(CompositeKeyComparator.class);
130.                job3.setGroupingComparatorClass(KeyGroupingComparator.class);
131.
132.                FileOutputFormat.setOutputPath(job3, new Path(rootPath
133.                        + args[1]));
134.                job3.setOutputKeyClass(IntWritable.class);
```

```
135.              job3.setOutputValueClass(Text.class);
136.
137.              System.exit(job3.waitForCompletion(true) ? 0 : 1);
138.           }
139.        }
140.        FileSystem.get(conf).deleteOnExit(tempDir);
141.     }
142.     }
143.
144.  // partitioned by frequency
145.  class KeyPartitioner extends Partitioner<CompositeKey, Text> {
146.     HashPartitioner<IntWritable, Text> hashPartitioner =
147.              new HashPartitioner<IntWritable, Text>();
148.     IntWritable newKey = new IntWritable();
149.
150.     @Override
151.     public int getPartition(CompositeKey key, Text value, int numReduceTasks) {
152.
          try {
153.        return hashPartitioner.getPartition(key.getFrequency(), value,
154.              numReduceTasks);
155.        } catch (Exception e) {
156.        e.printStackTrace();
157.        return (int) (Math.random() * numReduceTasks);
159.        }
160.     }
161.     }
162.
163.  // group words together by frequency order
164.  class KeyGroupingComparator extends WritableComparator {
165.     protected  KeyGroupingComparator()  {
166.
167.        super(CompositeKey.class, true);
168.     }
169.
170.     @SuppressWarnings("rawtypes")
171.     @Override
172.     public int compare(WritableComparable w1, WritableComparable w2) {
173.        CompositeKey key1 = (CompositeKey) w1;
174.        CompositeKey key2 = (CompositeKey) w2;
175.        return key2.getFrequency().compareTo(key1.getFrequency());
176.     }
177.     }
178.
```

```
179. // comparison between composite keys
180. class CompositeKeyComparator extends WritableComparator {
181.     protected CompositeKeyComparator() {
182.         super(CompositeKey.class, true);
183.     }
184.
185.     @SuppressWarnings("rawtypes")
186.     @Override
187.     public int compare(WritableComparable w1, WritableComparable w2) {
188.
189.         CompositeKey key1 = (CompositeKey) w1;
190.         CompositeKey key2 = (CompositeKey) w2;
191.         int cmp = key2.getFrequency().compareTo(key1.getFrequency());
192.         if (cmp != 0)
193.             return cmp;
194.         return key1.getWord().compareTo(key2.getWord());
195.     }
196. }
197.
198. // construct a composite key class
199. class CompositeKey implements WritableComparable<CompositeKey> {
200.     private Text word;
201.     private IntWritable frequency;
202.
203.     public CompositeKey() {
204.         set(new Text(), new IntWritable());
205.     }
206.
207.     public CompositeKey(String word, int frequency) {
208.
209.         set(new Text(word), new IntWritable(frequency));
210.     }
211.
212.     public CompositeKey(Text w, IntWritable f) {
213.         set(w, f);
214.     }
215.
216.     public void set(Text t, IntWritable n) {
217.         this.word = t;
218.         this.frequency = n;
219.     }
220.
221.     @Override
222.     public String toString() {
```

```
223.         return (new StringBuilder()).append(frequency).append(' ').append(word)
224.                 .toString();
225.     }
226. 
227.     @Override
228.     public boolean equals(Object o) {
229.         if (o instanceof CompositeKey) {
230.             CompositeKey comp = (CompositeKey) o;
231.             return word.equals(comp.word) && frequency.equals(comp.frequency);
232.         }
233.         return false;
234.     }
235. 
236.     @Override
237.     public void readFields(DataInput in) throws IOException {
238.         word.readFields(in);
239.         frequency.readFields(in);
240.     }
241. 
242.     @Override
243.     public void write(DataOutput out) throws IOException {
244.         word.write(out);
245.         frequency.write(out);
246.     }
247. 
248.     @Override
249.     public int compareTo(CompositeKey o) {
250.         int result = word.compareTo(o.word);
251.         if (result != 0) {
252.             return result;
253.         }
254.         return result = frequency.compareTo(o.frequency);
255.     }
256. 
257.     public Text getWord() {
258.         return word;
259.     }
260. 
261.     public IntWritable getFrequency() {
262.         return frequency;
263.     }
264. }
```

❖ Source Code : WCSparkSortValues.java

```
1. /**
2.  * This Spark program is to implement a secondary sort
3.  * of the WordCount example, which means that the final
4.  * result is not only sorted by frequency, but also sorted
5.  * by words.
6.  */
7.
8. import scala.Tuple2;
9. import org.apache.spark.SparkConf;
10. import java.util.*;
11. import org.apache.spark.api.java.*;
12. import org.apache.spark.Partitioner;
13. import org.apache.spark.HashPartitioner;
14. import org.apache.hadoop.conf.Configuration;
15. import org.apache.spark.api.java.function.FlatMapFunction;
16. import org.apache.spark.api.java.function.Function2;
17. import org.apache.spark.api.java.function.PairFunction;
18. import org.apache.hadoop.fs.*;
19. import java.io.Serializable;
20. import java.util.regex.Pattern;
21.
22.
23. public final class WCSparkSortValues {
24.     private static final Pattern SPACE = Pattern.compile(" ");
25.     private static String rootPath =
"hdfs://10.59.7.151:9000/user/hadoop/"; 26.
27.     public static void main(String[] args) throws Exception {
28.         SparkConf sparkConf = new SparkConf().setAppName("WordCount in Spark");
29.         JavaSparkContext spark = new JavaSparkContext(sparkConf);
30.
31.         FileSystem fs = FileSystem.get(spark.hadoopConfiguration());
32.         JavaRDD<String> textFile = spark.textFile(rootPath + "/input/" + args[0]);
33.         //load data for HDFS and split each line by space
34.         JavaRDD<String> words = textFile
35.                 .flatMap(new FlatMapFunction<String, String>()         {
36.                     public Iterable<String> call(String s) {
37.                 return  Arrays.asList(s.split("  "));}  38.
                    });
39.
40.         //map each word to the value one
41.         JavaPairRDD<String, Integer> pairs = words
42.                 .mapToPair(new PairFunction<String, String, Integer>() {
```

```
43.                 public Tuple2<String, Integer> call(String s) {
44.                     return new Tuple2<String, Integer>(s, 1); }
45.                 });
46.
47.         //reduce by key, namely, the words.
48.         JavaPairRDD<String, Integer> counts = pairs
49.                 .reduceByKey(new Function2<Integer, Integer, Integer>()
50.                 {
51.                   public Integer call(Integer a, Integer b) { return a + b; }
52.                 });
53.
54.         //sort by key
55.         JavaPairRDD<String, Integer> sortedByKeyList = counts.sortByKey(true);
56.
57.         //reverse key-to-value to value-to-key
58.         JavaPairRDD<Tuple2<Integer, String>, Integer> countInKey = sortedByKeyList
59.                 .mapToPair(a -> new Tuple2(
60.                         new Tuple2<Integer, String>(a._2, a._1), null));
61.
62.         //construct a composite RDD pair and also group by frequency
63.         JavaPairRDD<Tuple2<Integer, String>, Integer> groupAndOrderByvalues
64.         = countInKey.repartitionAndSortWithinPartitions(new MyPartitioner(1),
65.                                             new TupleComparator());
66.
67.         //extract the key of the composite RDD pair
68.         JavaRDD<Tuple2<Integer,String>> data = groupAndOrderByvalues.keys();
69.
70.         //convert JavaPairRDD to JavaRDD
71.         JavaPairRDD<Integer,String> results = JavaPairRDD.fromJavaRDD(data);
72.
73.         //make sure only one output and also format it
74.         results.repartition(1).map(s -> s._1 + "\t" + s._2)
75.                 .saveAsTextFile(rootPath + args[1]);
76.
77.         //stop the spark application
78.         spark.stop();
79.     }
80. }
81.
82. class TupleStringComparator implements
83.         Comparator<Tuple2<Integer, String>>, Serializable {
84.     @Override
85.     public int compare(Tuple2<Integer, String> tuple1,
86.             Tuple2<Integer, String> tuple2) {
```

```
87.         return tuple1._2.compareTo(tuple2._2);
88.     }
89. }
90.
91. //construct a practitioner by frequency
92. class MyPartitioner extends Partitioner {
93.     private int partitions;
94.     public MyPartitioner(int partitions) {
95.         this.partitions = partitions;
96.     }
97.
98.     @Override
99.     public int getPartition(Object o) {
100.         Tuple2 newKey = (Tuple2) o;
101.         return (int) newKey._1 % partitions;
102.     }
103.
104.     @Override
105.     public int numPartitions() {
106.         return partitions;
107.     }
108. }
109.
110. //construct a key comparator in a composite RDD
111. class TupleComparator implements Comparator<Tuple2<Integer, String>>,
112.         Serializable {
113.     @Override
114.     public int compare(Tuple2<Integer, String> tuple1,
115.             Tuple2<Integer, String> tuple2) {
116.         return tuple2._1 - tuple1._1;
117.     }
118. }
```

❖ Source Code : PageRankHadoop.java

```
1.  /**
2.   * This Hadoop program is to implement a PageRank algorithm,
4.   * The datasouce is from SNAP (https://snap.stanford.edu/).
6.
8. import java.text.*;
```

```
10. import org.apache.hadoop.io.*;
11. import org.apache.hadoop.mapreduce.Mapper;
12. import org.apache.hadoop.mapreduce.Reducer;
13. import org.apache.hadoop.conf.Configuration;
14. import org.apache.hadoop.fs.FileSystem;
15. import org.apache.hadoop.fs.Path;
16. import org.apache.hadoop.mapreduce.lib.input.*;
17. import org.apache.hadoop.mapreduce.lib.output.*;
18. import org.apache.hadoop.mapreduce.Job;
19.
20. public class PageRankHadoop {
21.     // utility attributes
22.     public static NumberFormat NF = new DecimalFormat("00");
23.     public static String LINKS_SEPARATOR = "|";
24.
25.     // configuration values
26.     public static Double DAMPING = 0.85;
27.     public static int ITERATIONS = 1;
28.     public static String INPUT_PATH = "/user/hadoop/input/";
29.     public static String OUTPUT_PATH = "/user/hadoop/";
30.
31.     // A map task of the first job: transfer each line to a map pair
32.     public static class FetchNeighborsMapper extends
33.             Mapper<LongWritable, Text, Text, Text> {
34.         public void map(LongWritable key, Text value, Context context)
35.                 throws IOException, InterruptedException {
36.             //skip the comment line with #
37.             if (value.charAt(0) != '#') {
38.                 int tabIndex = value.find("\t");
39.                 String nodeA = Text.decode(value.getBytes(), 0, tabIndex);
40.                 String nodeB = Text.decode(value.getBytes(), tabIndex + 1,
41.                         value.getLength() - (tabIndex + 1));
42.                 context.write(new Text(nodeA), new Text(nodeB));
43.             }
44.         }
45.     }
46.
47.     // A reduce task of the first job: fetch the neighbor's links
48.     // and set the initial value of fromLinkId 1.0
49.     public static class FetchNeighborsReducer extends
50.             Reducer<Text, Text, Text, Text> {
51.         public void reduce(Text key, Iterable<Text> values, Context context)
52.                 throws IOException, InterruptedException {
53.             boolean first = true;
```

```
54.          String links = "1.0\t";
55.          int count = 0;
56.          for (Text value : values) {
57.             if (!first)
58.                links += ",";
59.             links += value.toString();
60.             first = false;
61.             count++;
62.          }
63.          context.write(key, new Text(links));
64.       }
65.
66.    }
67.
68.    // A map task of the second job:
69.    public static class CalculateRankMapper extends
70.          Mapper<LongWritable, Text, Text, Text> {
71.       public void map(LongWritable key, Text value, Context context)
72.             throws IOException, InterruptedException {
73.
74.          int tIdx1 = value.find("\t");
75.          int tIdx2 = value.find("\t", tIdx1 + 1);
76.
77.          // extract tokens from the current line
78.          String page = Text.decode(value.getBytes(), 0, tIdx1);
79.          String pageRank = Text.decode(value.getBytes(), tIdx1 + 1, tIdx2
80.                - (tIdx1 + 1));
81.
82.          // Skip pages with no links.
83.          if (tIdx2 == -1)
84.             return;
85.
86.          String links = Text.decode(value.getBytes(), tIdx2 + 1,
87.                value.getLength() - (tIdx2 + 1));
88.          String[] allOtherPages = links.split(",");
89.          for (String otherPage : allOtherPages) {
90.             Text pageRankWithTotalLinks = new Text(pageRank + "\t"
91.                   + allOtherPages.length);
92.             context.write(new Text(otherPage), pageRankWithTotalLinks);
93.          }
94.
95.          // put the original links so the reducer is able to produce the
96.          // correct output
```

```
97.             context.write(new Text(page), new Text(PageRankHadoop.LINKS_SEPARAT
    OR
98.                     + links));
99.         }
100.    }
101.
102.    public static class CalculateRankReducer extends
103.            Reducer<Text, Text, Text, Text> {
104.        public void reduce(Text key, Iterable<Text> values, Context context)
105.                throws IOException, InterruptedException {
106.            String links = "";
107.            double sumShareOtherPageRanks = 0.0;
108.
109.            for (Text value : values) {
110.
111.                String content = value.toString();
112.
113.                //check if a linke has an appending 'links' string
114.                if (content.startsWith(PageRankHadoop.LINKS_SEPARATOR)) {
115.                    links += content.substring(PageRankHadoop.LINKS_SEPARATOR
116.                            .length());
117.                } else {
118.                    String[] split = content.split("\\t");
119.
120.                    // extract tokens
121.                    double pageRank = Double.parseDouble(split[0]);
122.                    if (split[1] != null && !split[1].equals("null")) {
123.                        int totalLinks = Integer.parseInt(split[1]);
124.
125.                        // calculate the contribution of each outgoing link
126.                        // of the current link
127.                        sumShareOtherPageRanks += (pageRank / totalLinks);
128.                    }
129.                }
130.
131.            }
132.            //get the final page rank of the current link
133.            double newRank = PageRankHadoop.DAMPING * sumShareOtherPageRanks

134.                    + (1 - PageRankHadoop.DAMPING);
135.            //ignore the link which has no outgoing links
136.            if (newRank > 0.15000000000000002
137.                    && !key.toString().trim().equals(""))
138.                context.write(key, new Text(newRank + "\t" + links));
```

```
139.         }
140.     }
141.
142.     // A map task of the third job for sorting
143.     public static class SortRankMapper extends
144.             Mapper<LongWritable, Text, Text, DoubleWritable> {
145.         public void map(LongWritable key, Text value, Context context)
146.                 throws IOException, InterruptedException {
147.
148.             int tIdx1 = value.find("\t");
149.             int tIdx2 = value.find("\t", tIdx1 + 1);
150.
151.             // extract tokens from the current line
152.             String page = Text.decode(value.getBytes(), 0, tIdx1);
153.             double pageRank = Double.parseDouble(Text.decode(value.getBytes(),
154.                     tIdx1 + 1, tIdx2 - (tIdx1 + 1)));
155.             context.write(new Text(page), new DoubleWritable(pageRank));
156.         }
157.
158.     }
159.
160.     public static void main(String[] args) throws Exception {
161.
162.         if(args.length == 6){
163.             //set the iteration numbers
164.             if(args[0].equals("-c"))
165.             PageRankHadoop.ITERATIONS = Math.max(Integer.parseInt(args[1]), 1);
166.             else printHelp();
167.             //set input path
168.             if(args[2].equals("-i"))
169.             PageRankHadoop.INPUT_PATH = PageRankHadoop.INPUT_PATH + args[3
    ];
170.             else printHelp();
171.             //set output path
172.             if(args[4].equals("-o"))
173.             PageRankHadoop.OUTPUT_PATH = PageRankHadoop.OUTPUT_PATH + a
    rgs[5];
174.             else printHelp();
175.         }else{
176.              printHelp();
177.         }
178.
179.         String inPath = null;
180.         String lastOutPath = null;
```

```
181.        PageRankHadoop pagerank = new PageRankHadoop();
182.
183.        System.out.println("Start to fetch neighbor links ...");
184.        boolean isCompleted = pagerank.job("fetchNeighborLinks",
185.                FetchNeighborsMapper.class, FetchNeighborsReducer.class,
186.                INPUT_PATH, OUTPUT_PATH + "/iter00");
187.        if (!isCompleted) {
188.            System.exit(1);
189.        }
190.
191.        for (int runs = 0; runs < ITERATIONS; runs++) {
192.            inPath = OUTPUT_PATH + "/iter" + NF.format(runs);
193.            lastOutPath = OUTPUT_PATH + "/iter" + NF.format(runs + 1);
194.            System.out.println("Start to calculate rank [" + (runs + 1) + "/"
195.                    + PageRankHadoop.ITERATIONS + "] ...");
196.            isCompleted = pagerank.job("jobOfCalculatingRanks",
197.                    CalculateRankMapper.class, CalculateRankReducer.class,
198.                    inPath, lastOutPath);
199.            if (!isCompleted) {
200.                System.exit(1);
201.            }
202.        }
203.
204.        System.out.println("Start to sort ranks ...");
205.        isCompleted = pagerank.job("jobOfSortingRanks", SortRankMapper.class,
206.                SortRankMapper.class, lastOutPath, OUTPUT_PATH + "/result");
207.        if (!isCompleted) {
208.            System.exit(1);
209.        }
210.
211.        System.out.println("All jobs done!");
212.        System.exit(0);
213.    }
214.
215.    public boolean job(String jobName, Class m, Class r, String in, String out)
216.            throws IOException, ClassNotFoundException, InterruptedException {
217.        Configuration conf = new Configuration();
218.        Job job = Job.getInstance(conf, jobName);
219.        job.setJarByClass(PageRankHadoop.class);
220.
221.        // input / mapper
222.        FileInputFormat.addInputPath(job, new Path(in));
223.        job.setInputFormatClass(TextInputFormat.class);
224.        if (jobName.equals("jobOfSortingRanks")) {
```

```
225.            job.setOutputKeyClass(Text.class);
226.            job.setMapOutputValueClass(DoubleWritable.class);
227.        } else {
228.            job.setMapOutputKeyClass(Text.class);
229.            job.setMapOutputValueClass(Text.class);
230.        }
231.
232.        job.setMapperClass(m);
233.
234.        // output / reducer
235.        FileOutputFormat.setOutputPath(job, new Path(out));
236.        job.setOutputFormatClass(TextOutputFormat.class);
237.
238.        if (jobName.equals("jobOfSortingRanks")) {
239.            job.setOutputKeyClass(Text.class);
240.            job.setMapOutputValueClass(DoubleWritable.class);
241.        } else {
242.            job.setOutputKeyClass(Text.class);
243.            job.setOutputValueClass(Text.class);
244.            job.setReducerClass(r);
245.        }
246.
247.        return  job.waitForCompletion(true);
248.
249.    }
250.
251.    //Print help message if the user does know how to run the program
252.    public static void printHelp() {
253.        System.out.println("Usage: PageRank.jar -c <iterations>
254.                                    -i <input file>
255.                                    -o <output file> \n");
256.    }
257. }
```

❖ Source Code : PageRankSpark.java

```
1.  /**
2.    * Copyright [2015] [Shengti Pan]
3.    *
4.    * Licensed under the Apache License, Version 2.0 (the "License");
5.    * you may not use this file except in compliance with the License.
6.    * You may obtain a copy of the License at
7.    *
8.    *  http://www.apache.org/licenses/LICENSE-2.0
```

```
 *
 * Unless required by applicable law or agreed to in writing, software
 * distributed under the License is distributed on an "AS IS" BASIS,
 * WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
 * See the License for the specific language governing permissions and
 * limitations under the License.
 */

import scala.Tuple2;
import com.google.common.collect.Iterables;
import org.apache.spark.SparkConf;
import org.apache.spark.api.java.JavaPairRDD;
import org.apache.spark.HashPartitioner;
import org.apache.spark.storage.StorageLevel;
import org.apache.spark.api.java.JavaRDD;
import org.apache.spark.api.java.JavaSparkContext;
import org.apache.spark.api.java.function.*;
import java.util.*;
import java.util.regex.Pattern;

/**
 * This Spark program is to implement a PageRank algorithm,
 * which was proposed by the Google founders Brin and Page.
 *        The        datasouce        is        from        SNAP
(https://snap.stanford.edu/).  */

public final class PageRankSpark {
  private static final Pattern SPACES = Pattern.compile("\\t+");
  private static final String ROOT_PATH = "hdfs://10.59.7.151:9000/user/hadoop/";
  private static final double DAMPING_FACTOR = 0.85d;
  public static void main(String[] args) throws Exception {
    if (args.length < 3) {
      System.err.println("Usage: PageRankSpark <file> <iteration number> <output>");
      System.exit(1);
    }

    SparkConf sparkConf = new SparkConf().setAppName("PageRankSpark");
    JavaSparkContext ctx = new JavaSparkContext(sparkConf);
    JavaRDD<String> lines = ctx.textFile(ROOT_PATH + "/input/" + args[0], 1);
    //filter the data and ignor the comment line
    final JavaRDD<String> data = lines.filter(new Function<String,Boolean>(){
```

```
52.           public Boolean call(String s) {
53.             return !s.startsWith("#");
54.           }
55.         });
56.
57.     //load all links and fetch their neighbors.
58.     JavaPairRDD<String, Iterable<String>> links = data.mapToPair(
59.                                   new PairFunction<String, String, String>() {
60.       @Override
61.       public Tuple2<String, String> call(String s) {
62.         String[] parts = SPACES.split(s);
63.         return new Tuple2<String, String>(parts[0], parts[1]);
64.       }
65.     }).groupByKey().persist(StorageLevel.MEMORY_ONLY());
66.
67.     //initialize the rank value of each link to 1.0
68.     JavaPairRDD<String, Double> ranks = links.mapValues(new Function<Iterable<Stri
     ng>,
69.                                                         Double>() {
70.       @Override
71.       public Double call(Iterable<String> rs) {
72.         return 1.0;
73.       }
74.     });
75.
76.     //calculate and update the ranks in multiple iterations
77.     for (int current = 0; current < Integer.parseInt(args[1]); current++) {
78.       //calculate the contributions to its outgoing links of the current link.
79.       JavaPairRDD<String, Double> contribs = links.join(ranks).values()
80.         .flatMapToPair(new PairFlatMapFunction<Tuple2<Iterable<String>, Double>,
81.                                       String, Double>() {
82.           @Override
83.           public Iterable<Tuple2<String, Double>>
84.               call(Tuple2<Iterable<String>, Double> s) {
85.             int urlCount = Iterables.size(s._1);
86.             List<Tuple2<String, Double>> results = new
87.                                         ArrayList<Tuple2<String, Double>>();
88.             for (String n : s._1) {
89.               results.add(new Tuple2<String, Double>(n, s._2() / urlCount));
90.             }
91.             return results;
92.           }
93.       });
94.
```

```
95.       //get the final rank of the current link
96.       ranks = contribs.reduceByKey(new Sum()).mapValues(new Function<Double, Dou
      ble>()
97.       {
98.         @Override
99.         public Double call(Double sum) {
100.             return (1.0 - DAMPING_FACTOR) + sum * DAMPING_FACTOR;
101.           }
102.         });
103.     }
104.
105.     //sort and format the result
106.     ranks.sortByKey().repartition(1).map(x -> x._1 + "\t" + x._2)
107.                     .saveAsTextFile(ROOT_PATH + args[2]);
108.     ctx.stop();
109.   }
110.
111.   private static class Sum implements Function2<Double, Double, Double> {
112.     @Override
113.     public Double call(Double a, Double b) { return a + b; }
114.     }
115.   }
```