Big Data and Technologies of Storage and Processing of Massive Data: Understand the basics of the HADOOP ecosystem (HDFS, MAPREDUCE, YARN, HIVE, HBASE, KAFKA and SPARK)

Keita, Moussa

October 2021

Big Data et Technologies de Stockage et de Traitement des Données Massives :

Comprendre les bases de l'écosystème HADOOP

(HDFS, MAPREDUCE, YARN, HIVE, HBASE, KAFKA et SPARK)

Data center Google, Saint-Ghislain, Belgique (Image Google)

Document version 1.0

(Octobre 2021)

Par

Moussa KEITA*

Expert/Formateur
Big Data–Data Science

Consultant Data à EDF, Société Générale, Caisse des Dépôts

Paris

* Contact Email : keitam09@ymail.com

SOMMAIRE

1 INTRODUCTION: LES ENJEUX LIES AU BIG DATA ET LA SOLUTION HADOOP

L'ère du digital se caractérise par une explosion de données provenant de différentes sources: historiques de navigation sur les pages web, données de e-commerce, données de jeux vidéo, données collectées par les applications de santé (fréquences cardiaques, pressions artérielles, poids etc…), logs générés par les applications, données de capteurs automatiques, traces de GPS, etc. Sur la dernière décennie, le monde a été témoin d'un véritable déluge de la data. Par exemple un article récent de IBM indique qu'en 2010, environ 2,5 trillions d'octets de données étaient générées chaque heure. Une décennie plus tard (en 2020), plus de 40 Zettaoctets étaient générées par heure. Un tel niveau de déferlement de la data a très rapidement soulevé des problématiques majeures quant aux moyens de stockage et d'analyse de ces données. Les technologies traditionnelles de stockage notamment les SGBD traditionnels ont très vite montré leurs limites face à la volumétrie grandissante des données. De nouvelles solutions technologiques ont dues être imaginées par les ingénieurs IT pour faire face aux défis posés par le Big Data, caractérisé, rappelons-le, par les 3V (volume, vélocité, variété). Deux principaux concepts sont au cœur des solutions proposées : ***stockage en architecture distribuée*** et ***traitement parallélisé***. Dans la nouvelle approche, il ne s'agit plus de centraliser le stockage et le traitement des données sur un même serveur, principe de fonctionnement des SGBD traditionnels. Désormais, il s'agit de distribuer le stockage des données sur un ensemble de machines organisés en cluster et de paralléliser leurs traitements sur plusieurs machines appelées nœuds. En effet, dans l'approche traditionnelle, la gestion et le traitement des données étaient jusque-là basés sur une architecture client/serveur central. Dès lors, tout accroissement important de la volumétrie de données nécessitait une augmentation de la capacité du serveur central : soit par une augmentation de la capacité de stockage du disque dur, soit par une augmentation de la mémoire de traitement ou par l'augmentation de la fréquence du processeur, etc… Malheureusement, face à l'ingestion d'un flux continu de données, une telle approche montre rapidement ses limites. D'abord, le fait de centraliser le stockage et le traitement sur un serveur central entraine une augmentation de la latence des requêtes (augmentation du temps de réponse des requêtes). Ce qui peut s'avérer problématique dans de nombreux secteurs d'activité. Ensuite l'upsizing c'est-à-dire l'augmentation continue de la capacité du serveur central, n'est pas toujours une solution efficace et optimisée. En effet, à partir d'un certain niveau de volumétrie de données, l'upsizing ne se traduit plus par une augmentation de la performance du système.

La plupart des géants de la data comme Yahoo, Google ou Facebook ont été très tôt confrontés aux limites de l'architecture centralisée. Google est l'une des premières entreprises à se tourner vers la solution d'architecture distribuée. Les dirigeants et les ingénieurs de Google étaient alors persuadés qu'une solution efficace pour répondre aux enjeux de stockage et de traitement du Big Data reposait sur le montage d'un Data Center constitué par l'assemblage de plusieurs machines commodes, bons marchés.

Cette approche était justifiée au vu de la baisse continue des coûts des disques durs et des processeurs sur le marché tels que prédits par la loi de Moore. Ainsi était né le paradigme des clusters.

L'approche de stockage *distribué* et de traitement *parallélisé* au moyen d'une architecture en cluster apparaissait, de loin, la solution la plus adaptée pour faire face à la problématique de volumétrie et de performance de traitement du Big Data. Elle apparaissait également comme la solution la plus économique, car avec la baisse continue des coûts du matériel informatique, les coûts d'acquisition, de maintenance et d'évolution d'un cluster reviendraient moins chers, à terme, que ceux d'un serveur central.

Cependant la principale difficulté de l'approche par le cluster ne réside pas nécessairement dans le montage de l'infrastructure du cluster en elle-même. La difficulté réside plutôt dans la conception, l'implémentation et le déploiement d'une solution technologique permettant d'exploiter cette infrastructure. En effet, l'adoption d'une architecture distribuée implique d'abord de répondre à un certain nombre de questions importantes. Quel système de fichier (FileSystem) faut-il adopter ? Quel algorithme de traitement faut-il adopter pour implémenter le moteur de traitement des données stockées sur le FileSystem ? etc. La technologie HADOOP fait partie des premières solutions qui ont été proposées pour tenter de répondre à ces différentes questions. En effet, HADOOP est un framework qui propose à la fois un système de stockage distribué scalable grâce au système de fichier HDFS et un concept de programmation permettant d'implémenter le traitement parallélisé des données grâce à l'algorithme MAPREDUCE.

Le but de ce document est proposer un aperçu général sur le framework HADOOP, ses principales fonctionnalités et ainsi que quelques couches technologiques qui forment son écosystème. Dans un premier temps, nous présentons les composantes de base de la technologie HADOOP à savoir : HDFS, MAPREDUCE et YARN. Et dans un second temps, nous présentons quelques outils permettant d'exploiter les données stockées sur HADOOP. Il s'agit notamment du moteur de requêtage HIVE, de la base de données HBASE, de l'outil d'ingestion et d'intégration de flux continus de données KAFKA et du moteur de traitement SPARK.

2 LE FRAMEWORK HADOOP ET SES PRINCIPAUX COMPOSANTS

Le framework HADOOP a été proposé à ses débuts par les ingénieurs de Yahoo. Il s'inspire des concepts de Google File System (GFS) et MapReduce élaborés par les ingénieurs de Google[1]. Aujourd'hui le framework HADOOP est un projet open-source géré par Apache Software Foundation.

Trois composantes majeures forment l'architecture du framework HADOOP :

- **HDFS** : *Hadoop Distributed File System* (HDFS) est un système de fichier distribué qui gère le stockage des données et garantit la tolérance aux pannes lors de l'exploitation d'un cluster HADOOP. HDFS est implémenté sur les mêmes principes que le GFS (Google File System)
- **MAPREDUCE** : est un moteur de traitement et d'analyse de données permettant d'implémenter les calculs parallèles. Il reprend les mêmes concepts que l'algorithme MapReduce proposé par Google.
- **YARN** : *Yet Another Resource Negotiator* (YARN) est le gestionnaire de ressources et l'orchestrateur de traitements sur le cluster HADOOP. Il permet de faire tourner en simultanée plusieurs traitements sur le cluster en attribuant à chacun les ressources (mémoire, CPU, etc) nécessaires à sa bonne exécution. YARN été intégré au framework HADOOP à partir de la version 2 de HADOOP. L'ajout de YARN au framework constitue une évolution majeure par rapport à l'architecture de base de la version 1 qui ne contenait que HDFS et MAPREDUCE.

Bien entendu, le cluster HADOOP est matérialisé par l'installation et la configuration de ces trois composantes sur une couche infrastructure (matériel informatique) : serveurs lames, serveurs virtuels VMware, etc.

Dans les sections qui suivent, nous procédons à une présentation détaillée de chacune des trois composantes de base : HDFS, MAPREDUCE et YARN.

[1] Voir Ghemawat et al., 2003, The Google file system, Proceedings of the nineteenth ACM Symposium on Operating Systems Principles
Dean, J., Ghemawat S., 2004, MapReduce: Simplified Data Processing on Large Clusters: 137–150.

2.1 HDFS: Le FileSystem de HADOOP

2.1.1 Présentation de HDFS

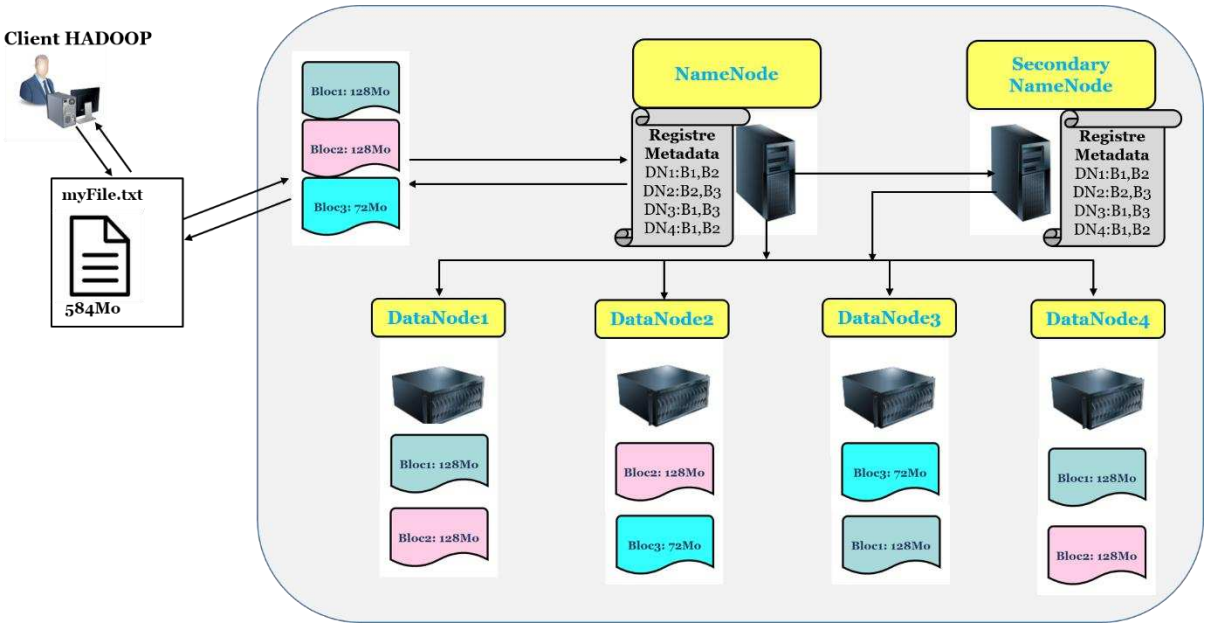
Hadoop Distributed File System (HDFS) inspiré de *Google File System* (GFS) est le système de fichier conçu pour servir de couche de stockage et d'accès aux données sur le cluster HADOOP. Le FileSystem HDFS reprend de nombreux concepts des FileSystems classiques comme EXT3 (pour Linux) ou FAT (pour Windows). On retrouve par exemple la notion de *blocs de données* : la plus petite unité de stockage. On retrouve également la notion de *métadonnées* permettant de retrouver les blocs à partir d'un nom de fichier. On retrouve aussi des notions comme droits d'accès ou arborescence. Cependant, HDFS se distingue des FileSystems classiques sur de nombreux points. D'abord, HDFS est indépendant du noyau du système d'exploitation. Il est portable et peut être déployé sur différents systèmes d'exploitation. Sur HDFS, les fichiers sont découpés en blocs de données de grande taille pour optimiser les temps de transfert et d'accès aux données. HDFS utilise des tailles de blocs largement supérieures à ceux des systèmes classiques. La taille d'un bloc de données HDFS est, par défaut, fixée à 128 Mo et reste configurable à souhait. Alors que sur les FileSystems classiques, la taille est généralement de 4 Ko. De plus, HDFS fournit un système de réplication des blocs de données. Le nombre de réplications (encore appelé facteur de réplication) est fixé par défaut à 3. Cela signifie que chaque bloc de donnée est répliqué sur trois nœuds différents du cluster. Bien sûr le facteur de réplication reste aussi configurable et modifiable selon les souhaits. Un des points fort du système HDFS est sa tolérance aux pannes. La tolérance aux pannes est rendue possible grâce notamment à la fonctionnalité de réplication. Notons tout de même que le FileSystem HDFS ne s'inscrit pas dans le standard POSIX. Par conséquent toutes les commandes exécutables sur un FileSystem traditionnel ne sont pas disponibles sur HDFS.

2.1.2 Architecture de HDFS

HDFS est un FileSystem fonctionnant sur le principe d'une architecture maitre-esclaves constituée d'un nœud maître (appelé NameNode) et d'un ensemble de nœuds esclaves appelés DataNodes. Dans le cluster HDFS, les DataNodes sont en communication permanente avec le NameNode afin d'assurer d'une part la distribution et la réplication des blocs de données écrits sur le cluster et d'autre part de faciliter la location des blocs lors des opérations de lecture. Dans une architecture HA (High Availability), le NameNode peut être secondé par un autre nœud appelé Secondary NameNode. Le secondary NameNode sert de backup au NameNode et prend le relais en cas de panne de celui-ci afin d'assurer la disponibilité du service.

La figure 1 présente l'architecture simplifiée d'un cluster HDFS.

Figure 1 : Architecture d'un cluster HDFS



2.1.2.1 Le NameNode

Le NameNode est un serveur central qui assure la gestion et la maintenance des métadonnées de l'ensemble des fichiers de données stockés sur HDFS. Le NameNode se présente comme un catalogue de l'ensemble des fichiers de données disponibles sur le cluster (voir figure 1). Le NameNode dispose des informations sur chacun des DataNodes du cluster : adresse, blocs de données et réplications hébergées, etc. Cela permet de retrouver l'ensemble des blocs d'un fichier et de reconstituer le fichier en entier lors de la lecture par un client. Par exemple, quand un client HADOOP souhaite récupérer un fichier stocké sur HDFS, la requête est d'abord adressée au NameNode. Celui-ci indique au client les DataNodes qu'il doit contacter afin de récupérer les blocs de données et ainsi reconstituer le fichier demandé.

Notons que toutes les métadonnées disponibles sont stockées physiquement sur le disque système du NameNode dans deux fichiers spécifiques *edits_xxx* et *fsimage_xxx*. Ces métadonnées sont, en outre, chargées en permanence sur la mémoire du Namenode. D'ailleurs, cela peut s'avérer problématique dans certaines situations notamment lorsqu'il s'agit de multitudes de petits fichiers stockés sur HDFS. C'est pourquoi, il est fortement recommandé d'une part de choisir une taille optimale pour les blocs de données et d'autre part de concaténer les petits fichiers de données avant de les pousser sur HDFS. En effet, pour chaque fichier stocké sur HDFS (quelle que soit sa taille) les métadonnées associées à ce fichier sont représentées dans un objet spécifique. Et chaque objet spécifique occupe 150 octets sur la mémoire du NameNode. Pour des soucis de performance du NameNode, il devient primordial de limiter le stockage de petits fichiers sur HDFS. De plus, les traitements comme MapReduce s'avèrent plus performants sur un nombre limité de gros fichiers que sur une multitude de petits fichiers.

2.1.2.2 Le Secondary NameNode

Dans l'architecture HDFS, le NameNode est un service critique car il représente le ***Single Point of Failure*** (point unique de défaillance). Cela signifie que lorsque le NameNode tombe en panne, c'est l'ensemble du système HDFS qui devient défaillant. Les métadonnées n'étant pas accessibles, il n'est plus possible de localiser les blocs de données et leurs réplications. Par conséquent, il ne sera pas possible ni d'écrire de nouveaux fichiers sur HDFS, encore mois de lire les fichiers qui y étaient déjà stockés. Pour gérer une telle éventualité, les architectes HADOOP ont choisi d'ajouter un second service de catalogue nommé *Secondary NameNode* pour servir de backup au NameNode. Le Secondary NameNode sert alors de secours au NameNode en cas de défaillance de celui-ci.

Le montage d'un Secondary NameNode sur le cluster HADOOP permet d'assurer la continuité du service lorsque le NameNode est indisponible. Il vérifie régulièrement l'état du NameNode et copie les métadonnées qui y sont stockées via les fichiers *edits_xxx* et *fsimage_xxx*. Pour rappel, sur le NameNode, toutes les métadonnées sont stockées sur le disque dans deux fichiers spécifiques *edits_xxx* et *fsimage_xxx*. Ces deux fichiers représentent l'état du cluster à l'instant t. L'information sur la position des blocs dans les DataNodes est reconstruite à chaque redémarrage du Namenode. Ce redémarrage se fait dans un mode appelé safemode. Lorsque le safemode est activé, toute nouvelle écriture sur HDFS est momentanément suspendue. Durant le safemode, le Namenode recharge les fichiers edits_xxx et fsimage_xxx et attend le retour des DataNodes sur la position des blocs. L'accès en écriture sera de nouveau possible dès que toutes les opérations de re-synchronisation des métadonnées auront été réalisées et que le safemode ait été désactivé.

L'architecture HDFS en mode High Availability (HA)

Dans le mode HA, la gestion des métadonnées du système HDFS n'est plus assurée par un seul NameNode mais plutôt par deux NameNodes qui se relaient à tour de rôle en cas de panne constatée sur l'un ou l'autre. En effet, plutôt que de s'appuyer sur le Secondary NameNode, l'architecture HDFS en mode *High Availability* configure deux NameNodes distincts capables de fonctionner en mode actif/passif. Dans le mode HA, le NameNode passif est un clone du NameNode actif. Il est mis à jour en permanence à l'aide de services appelés *JournalNodes*. En cas de panne du NameNode actif, le deuxième NameNode qui était jusque-là en Stand-by prend le relais et devient le NameNode actif et assure le fonctionnement du cluster afin de garantir la continuité du service et la disponibilité des données.

2.1.2.3 Les DataNodes

Contrairement au(x) NameNode(s) qui assure(nt) le stockage et la gestion des métadonnées, les DataNodes, eux, représentent les nœuds de stockage des blocs de données sur HDFS. Les DataNodes ont pour rôle d'exécuter les instructions envoyées par le NameNode. En particulier, les DataNodes sont sollicités par le NameNode lors

des opérations de lecture et d'écriture à la demande d'un client HADOOP. Lors d'une opération de lecture, les DateNodes transmettent au client les blocs correspondant au fichier à lire. De même lors d'une opération d'écriture, les DataNodes indiquent au client l'emplacement des blocs créés.

Les DataNodes sont également sollicités par le NameNode pour tenir à jour le catalogue des métadonnées. Par exemple, chaque DataNode envoie régulièrement des signaux au NameNode pour l'informer de son état de santé (envoi des heartbeats). Par défaut, le DataNode envoi un heartbeat toutes les 3 secondes. Et si au bout de 10 minutes, le NameNode ne reçoit aucun heartbeat de la part d'un DataNode, il considère que celui-ci est perdu (mort). Dès lors, le NameNode demande aux autres DataNodes de faire des réplications des blocs qui étaient disponibles sur le DataNode perdu.

La scalabilité du cluster HADOOP (c'est-à-dire l'augmentation de la capacité de stockage et de la performance de traitement) s'obtient simplement en ajoutant de nouveaux DataNodes au cluster. D'ailleurs c'est cette propriété qui fait la force d'un cluster HADOOP par rapport à une architecture traditionnelle centralisée. La scalabilité s'obtient avec un coût moindre en ajoutant des nœuds supplémentaires avec des matériels informatiques commodes.

2.1.3 Ecriture et lecture sur HDFS

2.1.3.1 Ecriture

L'écriture d'un fichier de données sur HDFS fonctionne sur le principe suivant :

1. Avec une commande PUT, le client indique au NameNode les informations sur le fichier à écrire : nom du fichier, arborescence d'écriture, etc. Le NameNode récupère en plus d'autres métadonnées du fichier: nom, taille, etc. Par la suite, le NameNode s'assure que le fichier a été splitté en blocs selon la taille maximale préconfigurée (ex : 128 Mo maximum par bloc)
2. Pour chaque bloc de données, le NameNode indique au client l'adresse du DataNode à contacter et qui prendra en charge le stockage du bloc.
3. Le client contacte le DataNode indiqué par le NameNode et lui envoie le bloc de données correspondant. Cette action est réalisée pour chaque bloc de données.
4. Enfin les DataNodes répliquent entre eux les blocs qu'ils ont reçus suivant le facteur de réplication préconfiguré (ex :3X). Les informations de réplication sont ensuite remontées au NameNode afin de mettre à jour le catalogue de métadonnées.

2.1.3.2 Lecture

La lecture d'un fichier de données sur HDFS fonctionne sur le principe suivant :

1. Avec une commande HDFS de type GET, CAT, etc…, le client indique au NameNode les informations sur le fichier à lire : nom, arborescence de lecture, etc.
2. Le NameNode disposant déjà de toutes les métadonnées sur le fichier à lire (taille, réplications, localisation des blocs, etc..) indique au client les DataNodes à contacter afin de récupérer les blocs de données.
3. Le client contacte chacun des DataNodes hébergeant un bloc de fichier et récupère ainsi l'ensemble des blocs permettant de reconstituer le fichier en entier. Lorsqu'un DataNode apparaît indisponible, le client en contacte un autre DataNode hébergeant une réplication du bloc de données.

2.1.4 Formats de stockage des données sur HDFS

En plus de son architecture distribuée et décentralisée permettant une grande capacité de stockage et une meilleure tolérance aux pannes, le FileSystem HDFS offre aussi une variété de format de stockage des fichiers de données. Il supporte non seulement des formats de fichiers « standard » comme des fichiers textes, CSV, XML, JSON mais aussi des formats de fichiers binaires comme des images. Toutefois, plusieurs formats de fichiers ont été spécifiquement conçus pour HADOOP afin d'optimiser le stockage sur HDFS et améliorer la performance lors des traitements. On dénote par exemple les formats comme le SequenceFile, le Avro ou le parquet. Nous passons ici en revue quelques formats de stockage des données supportés par HDFS.

2.1.4.1 TEXTFILE :TXT/CSV/JSON

Le format Text (fichier brut de texte : TXT, CSV, etc.) est le format de base de stockage sur HDFS. Le format TextFile est généralement utilisé dans les cas où les données doivent être facilement accessibles au format texte comme par exemple les e-mails, les numéros de téléphone, toute information à accès direct.

Lors du stockage sur HDFS, chaque TextFile est d'abord splitté en lignes. La clé de la ligne est la position dans le fichier et la valeur correspond à la ligne de texte.

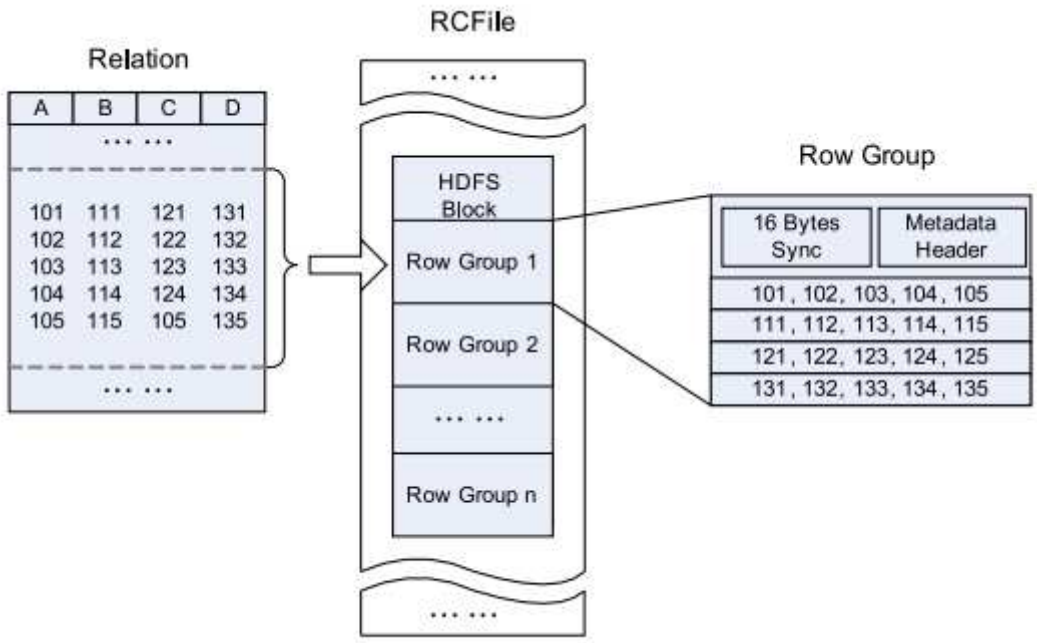
Les TextFiles peuvent occupér d'importants espace-disques, car ils ne prennent pas en charge la compression par bloc. Ce qui rend moins efficaces les opérations de lecture par rapport à d'autres formats de stockage.

2.1.4.2 RCFILE

RCFile (Record Columnar File) est un format initialement adopté par les projets HIVE et PIG pour servir de format de stockage sur HDFS. Plusieurs arguments étaient avancés en faveur de l'utilisation du format RCFILE. Par exemples, il permet un chargement rapide des données, un traitement rapide des requêtes, une utilisation efficace de l'espace de stockage et une forte adaptabilité aux modèles de charge de travail dynamiques.

D'un point de vue architecture, le RCFile divise les données horizontalement en groupes de lignes (row groups). Les données du row group sont dans un format colonnaire. Ainsi, au lieu de stocker les lignes les unes à la suite des autres comme dans un fichier standard, le RCFILE stocke toutes les valeurs correspondant à la ligne dans une colonne dédiée en les regroupant à d'autres lignes. Autrement dit, le RCFILE transpose les colonnes en lignes tout en procédant à un regroupement de lignes dénommé Row Groups. Voir figure 2 ci-dessous.

Figure 2 : Structure des données dans un RCFILE



Source : http://www.semantikoz.com/blog/wp-content/uploads/2014/02/hadoop-rcfile-format.png

Dans cet exemple de RCFILE, les lignes 1 à 100 sont stockées dans un groupe et les lignes 101 à 200 dans un autre groupe et ainsi de suite. Un ou plusieurs row groups sont stockés dans un même block de réplication HDFS.

La recherche de données dans les tables stockées au format RCFile permettait à HIVE d'exclure une grande partie des données et d'obtenir les résultats plus rapidement. Le stockage des tables intermédiaires en tant que RCFile réduit de manière significative les charges IO et les exigences de stockage par rapport aux fichiers de type Textfile.

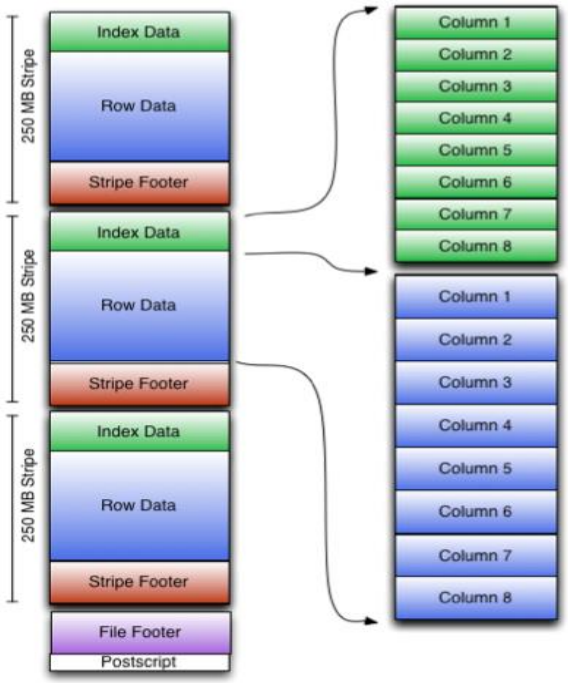
2.1.4.3 ORC

Proposé en remplacement du RCFile dans le projet Apache Hive, le format ORC (Optimized Row Columnar) est un format de données créé dans le but d'optimiser la taille et l'espace de stockage des fichiers HDFS afin d'accélérer les requêtes Hive.

D'un point de vue architecture, les fichiers ORC sont divisés en bandes (stripes) représentant environ 250 Mo par défaut. Les stripes sont indépendantes les unes des autres et forment l'unité de traitement. Chaque fichier avec la disposition en colonne est optimisé pour la compression et le saut de données/colonnes pour réduire la charge de lecture et de décompression.

La figure 3 ci-dessous illustre l'architecture d'un bloc de données stocké sous format ORC.

Figure 3 : Structure d'un fichier ORC



Source : http://www.semantikoz.com/blog/wp-content/uploads/2014/02/hadoop-rcfile-format.png

Comme illustré sur le schéma, chaque bande dans un fichier ORC contient des données d'index (index data), des Row Data et footer de bande.

Les Index Data contiennent les valeurs min et max pour chaque colonne et les positions de rangée dans chaque colonne. Les Row Data sont utilisées pour analyser le tableau de données. Le footer de bande contient un répertoire d'emplacements des Row data.

Le footer page du fichier contient la liste de stripes dans le fichier, le nombre de lignes par stripe et le type de donnée de chaque colonne. Il contient également un nombre d'agrégats au niveau des colonnes, ex : min, max et somme.

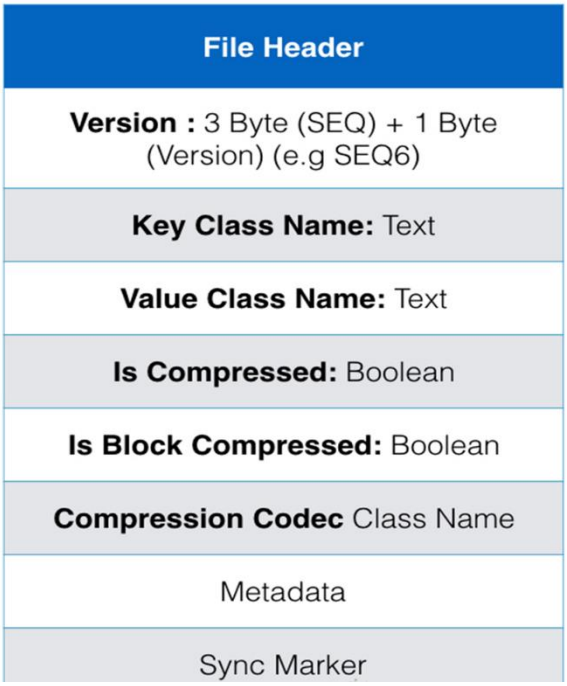
Le postscript contient les paramètres de compression et la taille du pied de page compressé.

2.1.4.4 SEQUENCEFILE

Le SequenceFile est un type de fichier plat composé de paires de clé-valeur binaires. Il est plus utilisé dans les algorithmes MapReduce en tant que formats d'input/output. Le sequencefile est plus efficient qu'un TextFile standard car les fichiers binaires sont plus compacts.

Le sequencefile se compose d'un en-tête fournissant toutes les métadonnées utilisées par le lecteur de fichiers pour déterminer le format du fichier mais aussi indiquer si le fichier est compressé ou non. La figure 4 ci-dessous illustre les informations contenues dans l'entête d'un sequencefile.

Figure 4 : En-tête d'un sequencefile



Source : https://examples.javacodegeeks.com/enterprise-java/apache-hadoop/hadoop-sequence-file-example

2.1.4.5 AVRO

Le format AVRO peut être présenté comme un cas particulier de SEQUENCEFILE mais sérialisé. Cependant, en règle générale, AVRO est une couche de sérialisation des données. La sérialisation consiste en un découpage de l'information en mémoire en petits morceaux d'octets ou des bits (utilisé pour la sauvegarde, la persistance ou le transport sur le réseau (proxy, RPC ...).

Un schéma Avro est indépendant du langage et se décrit lui-même. Le schéma est écrit dans l'en-tête de chaque fichier. Les fichiers de données AVRO sont compressibles et divisibles.

Avro spécifie deux encodages de sérialisation : binaires et JSON. La plupart des applications utilisent le codage binaire, car il est plus léger et plus rapide.

Le format AVRO s'impose dans de nombreuses applications comme un format de stockage privilégié sur HDFS.

2.1.4.6 PARQUET

Le ParquetFile est un format de fichier binaire orienté-colonne. Il a été conçu pour apporter plus d'efficacité lors de requêtes à grande échelle. Dans le parquet, chaque fichier de données contient les valeurs pour un ensemble de lignes ("Row groups"). Dans un fichier de données PARQUET, les valeurs de chaque colonne sont organisées de manière à ce qu'elles soient toutes adjacentes, ce qui permet une bonne compression du fichier.

Le ParquetFile est très comparable aux RCFile et ORCFile. Tous les trois formats font partie de la catégorie de stockage de données en colonnes dans l'écosystème HADOOP. Ils proposent une meilleure compression et un meilleur encodage des données. Ils affichent de meilleurs performances de lecture mais au prix d'écritures plus lentes.

2.1.5 HADOOP et les systèmes de fichier tiers

Le framework HADOOP peut tout être déployé sur des systèmes de fichier tiers en dehors de HDFS qui est le FileSystem par défaut. Mais ce déploiement hors HDFS se fait généralement à travers l'ajout de couches supplémentaires pour assurer la compatibilité. Plusieurs constructeurs de baies de stockage comme EMC, HP ou IBM ont ainsi proposé des couches de compatibilité HDFS au-dessus de leurs baies afin de permettre de stocker les données d'un cluster HADOOP. Par exemple MapR, l'un des pionniers dans la distribution de la solution HADOOP, a développé son propre système de gestion de fichiers pour faire face au problème de fragilité lié aux NameNodes. La solution proposée par MapR consiste à distribuer les métadonnées sur les nœuds du cluster et à ajouter des fonctionnalités avancées comme les snapshots, la réplication ou le clonage.

2.2 MAPREDUCE: le moteur de traitement natif de HADOOP

2.2.1 Principe de fonctionnement

La couche MAPREDUCE est la deuxième composante majeure de l'architecture HADOOP. MAPREDUCE est un moteur de traitement spécifiquement conçu pour exploiter les données stockées sur HDFS. Il est implémenté sous forme d'un algorithme de parallélisation des calculs dit MapReduce.

Comme nous l'avons déjà indiqué, le Big Data soulève deux problématiques majeures. D'un côté la problématique de stockage et de gestion d'un flux important et continu des données. De l'autre, la problématique de traitement et d'exploitation des données collectées. Le FileSystem HDFS s'est rapidement imposé comme une solution viable face à la première problématique. Et pour trouver une réponse adéquate à la deuxième problématique, le framework HADOOP a proposé le modèle MAPREDUCE.

Le modèle MAPREDUCE a été proposé en premier par les ingénieurs de Google[2]. Il est basé sur deux principales fonctions : la fonction Map et la fonction Reduce. Les termes Map et Reduce ne sont pas nouveaux aux traitement du Big Data. Ce sont des concepts empruntés aux langages de programmation fonctionnelle. Ils sont appliqués aux problématiques de traitement du Big Data à cause de la facilité qu'ils offrent pour implémenter la parallélisation des opérations de base comme les tris, les filtrages, les agrégations ou les regroupements.

L'une des particularités de l'algorithme MapReduce est sa capacité à tirer parti de la localité entre les données et les traitements sur le même nœud du cluster. La localité permet de minimiser les charges de transferts de données entre les nœuds. Ce qui se traduit par une meilleure performance des traitements.

Les points ci-dessous décrivent les différentes étapes d'exécution d'un programme MapReduce :

1. *Etape de prétraitement des données en entrée*

   C'est une étape dans laquelle plusieurs actions préparatoires sont réalisées : décompression des fichiers, split des fichiers en blocs qui seront traités séparément, mise des entrées sous forme de paires (clé, valeur). Une clé peut être de n'importe quel type de données : entier, texte. . . Et une valeur peut aussi être de n'importe quel type de données. Par exemple, un fichier texte peut être représenté sous la forme de clé-valeur où la clé est le numéro de ligne et la valeur la chaîne de texte formant la ligne.

2. *Etape de Map*

   Dans cette étape, le NameNode envoie une instance de la fonction Map à chaque DataNode qui héberge une partie des blocs de données à traiter. Cela

[2] Dean, J., Ghemawat S., 2004, MapReduce: Simplified Data Processing on Large Clusters: 137–150.

permet de traiter les données là où elles sont localisées (data-locality), limitant ainsi le transfert des données entre les nœuds pour avoir plus de performance. Notons toutefois que dans certains cas, les DataNodes sont amenés à distribuer en cascade les tâches entre eux. La fonction Map reçoit toujours une paire (clé-valeur) en entrée et mais peut renvoyer un nombre quelconque de paires clé-valeur en sortie.

La fonction Map est par nature parallélisable, car les calculs sont indépendants. Par exemple, pour quatre éléments de blocs de données à traiter, nous avons le schéma suivant :

- v1 = Map(e1)
- v2 = Map(e2)
- v3 = Map(e3)
- v4 = Map(e4)

Ces quatre instances Map peuvent s'exécuter en simultané et en parallèle sur 4 nœuds différents. Toutefois, pour chaque instance, la fonction *mappée* doit dépendre uniquement de ses propres paramètres. Par exemple, elle ne devrait pas modifier une variable globale au risque d'entrainer des effets de bord.

3. *Etape de Shuffle & Sort*

   C'est l'étape où les paires clé-valeur produites par les instances Map sur les nœuds sont redistribuées entre les nœuds de sorte que les valeurs ayant les mêmes clés soient regroupées sur les mêmes nœuds. Cela permet de produire de nouvelles paires clé-valeur homogènes et proches qui serviront d'entrée à l'étape Reduce. Durant l'étape *Shuffle & Sort*, le transfert de données entre les DataNodes s'avère parfois nécessaire et inévitable.

4. *Etape de Reduce*

   C'est l'étape d'agrégation des paires clé-valeur ayant la même clé en vue de construire le résultat final.

   La fonction Reduce peut renvoyer un nombre quelconque de paires en sortie, mais il arrive qu'elle renvoie une valeur unique. Tout dépend de la spécification du problème initial. Par exemple, dans un problème qui consiste à compter le nombre d'occurrences de chaque mot dans un fichier de texte, l'action Reduce renvoie un ensemble de paires clé-valeur où la clé est représentée par chaque mot et la valeur le nombre de fois où ce mot apparait dans le fichier. En revanche, dans un problème qui consiste à calculer le montant maximum des salaires renseignés dans un fichier, l'action Reduce renvoie nécessairement une valeur unique.

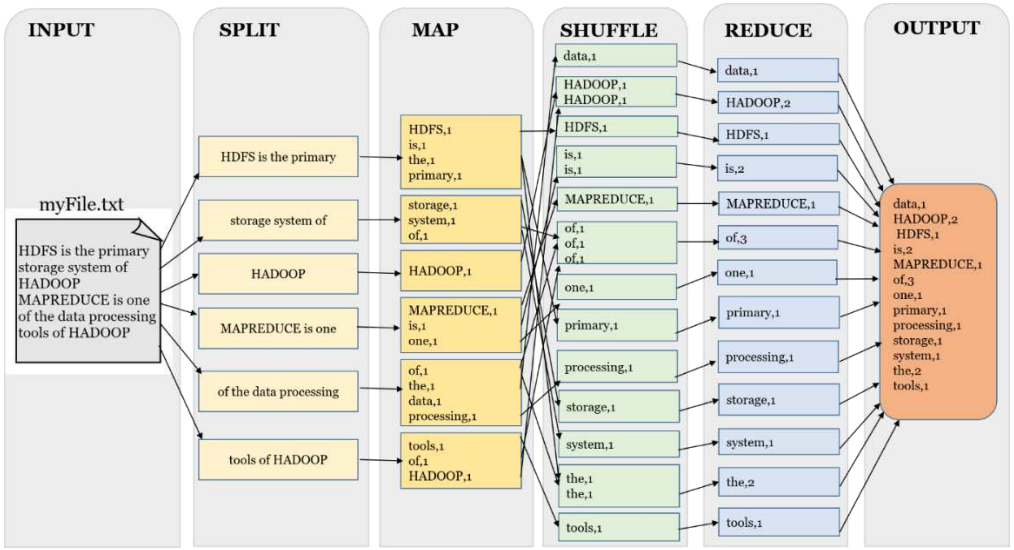
   Pour ce qu'il s'agit de la parallélisation de l'étape Reduce, il faut noter que contrairement à l'étape Map (qui est full-parallélisable), l'étape Reduce n'est que partiellement parallélisable. En effet, la parallélisation se fait de manière séquentielle en regroupant d'abord les résultats intermédiaires des Reduce précédentes et en appliquant de nouvelles Reduce. Par exemple, en prenant les

quatre paires issues de l'étape Map, l'étape Reduce peut s'appliquer comme suit :

- v1,2 = Reduce(v1 ; v2)
- v3,4 = Reduce (v3 ; v4)
- output = Reduce(v1,2 ; v3,4)

On constate que seules les deux premières instances de Reduce peuvent être lancées et exécutées en parallèle. La troisième instance doit attendre la fin des deux premières afin de récupérer les résultats intermédiaires. La figure 5 ci-dessous illustre les séquences d'exécution d'un job MapReduce : le wordcount

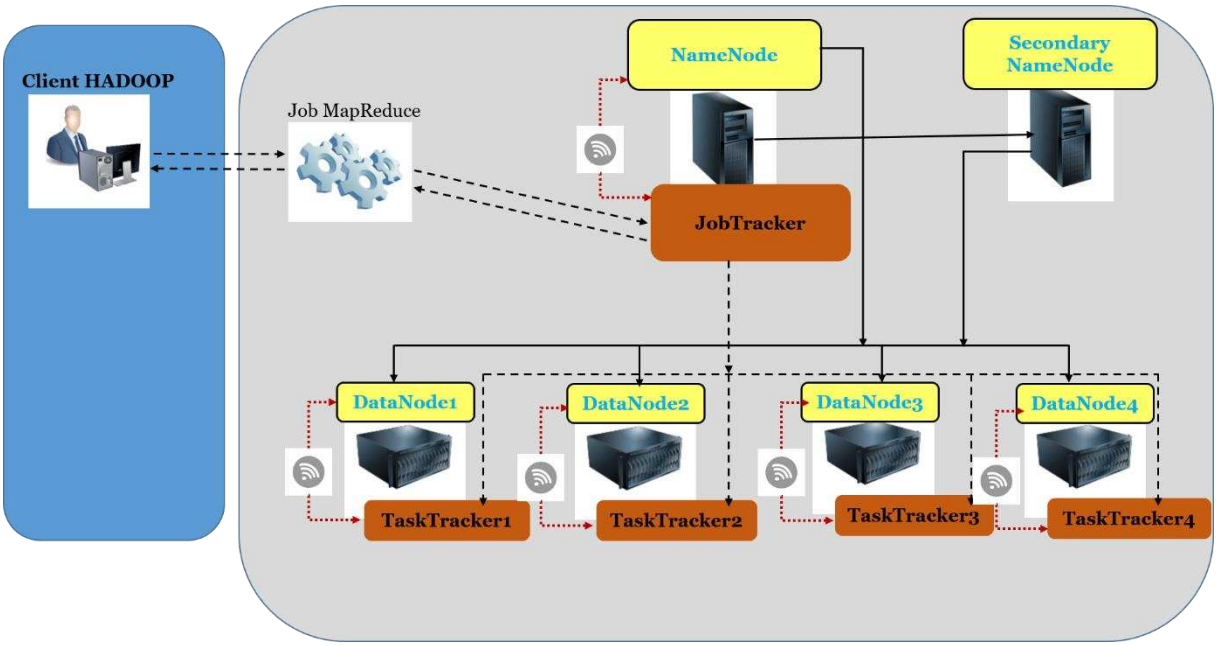
Figure 5 : Illustration des étapes d'un job MapReduce



2.2.2 Architecture de la couche MAPREDUCE

MAPREDUCE est une couche implémentée au-dessus de HDFS. Elle fonctionne de manière décentralisée selon une architecture maître-esclave. Deux services assurent l'exécution et le monitoring des jobs ainsi que la coordination des étapes. Il s'agit notamment du JobTracker (service central assurant le rôle de Maître) et les TaskTrackers (services décentralisés assurant le rôle d'esclaves). A noter que sur le cluster HADOOP, le JobTracker est généralement hébergé sur le même serveur que le NameNode de HDFS alors que les TaskTrackers se situent sur les DataNodes. Mais cette co-localisation des services de HDFS et de MAPREDUCE n'est pas obligatoire. Il suffit simplement de configurer le JobTracker de sort à ce qu'il puisse communiquer avec le NameNode afin de pouvoir accéder aux données stockées sur la couche HDFS. La figure 6 ci-dessous illustre l'implémentation de la couche MAPREDUCE au-dessus de la couche HDFS sur un cluster HADOOP.

Figure 6 : Architecture du cluster MAPREDUCE/ HDFS



2.2.2.1 Le JobTracker

C'est un service central matérialisé par un processus JVM (Java Virtual Machine) qui assure l'ordonnancement des tâches MapReduce et la gestion des ressources nécessaires à l'exécution des tâches. Lorsqu'un client HADOOP soumet un job MapReduce, c'est le JobTracker qui se charge de la réception et du dispatch des fichiers ressources nécessaires à la bonne exécution du job. C'est lui qui assure la planification et leur assignation au TaskTrackers qui se chargent ensuite de leur exécution. Par ailleurs, le JobTracker est toujours en communication avec le NameNode HDFS qui lui fournit toutes les métadonnées nécessaires au traitement des fichiers. Le JobTracker assure la co-localisation des traitements et des données pour une meilleure performance.

2.2.2.2 Le TaskTracker

Le TaskTracker est une unité de calcul dans l'architecture MAPREDUCE de même façon que le DataNode représente une unité de stockage dans l'architecture HDFS. En règle générale, le TaskTracker est matérialisé par une machine virtuelle java (JVM) qui est instanciée lors de la soumission d'un job MapReduce par le client HADOOP. Le TaskTracker se charge de l'exécution et du suivi des tâches Map et Reduce qui lui sont envoyées par le JobTracker. Durant toute la phase d'exécution du job MapReduce, le TaskTracker est en communication constante avec le JobTracker (à travers des heartbeat calls) pour l'informer de l'état d'avancement des tâches. Ainsi, en cas de défaillance ou d'échec d'exécution d'une tâche par un TaskTracker, le JobTracker doit pouvoir ordonner l'instanciation d'une nouvelle JVM sur le DataNode en vue de reexécuter la tâche perdue.

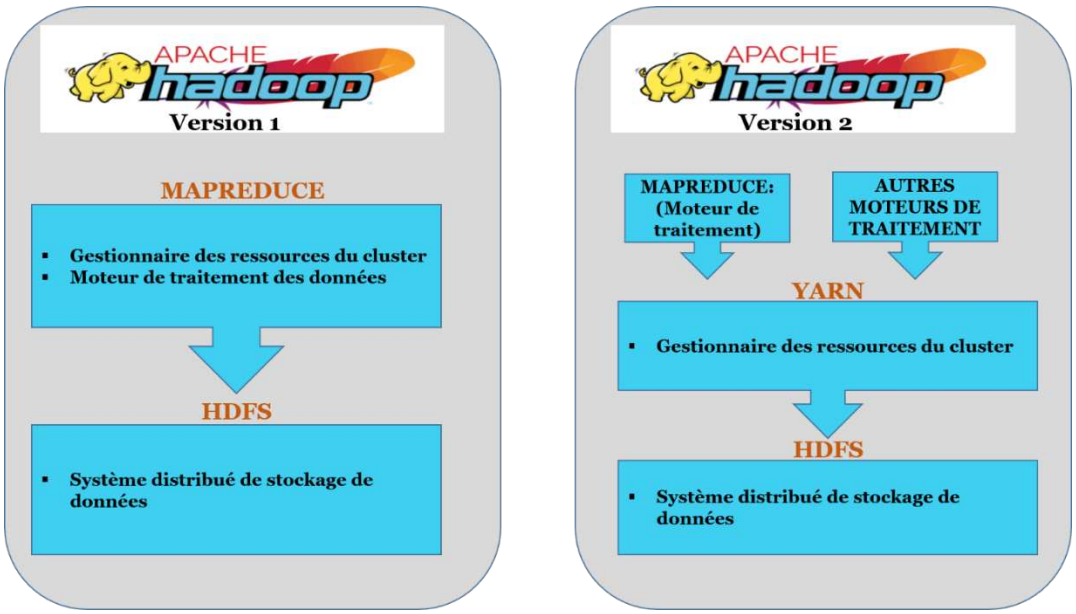
2.3 YARN: l'orchestrateur du cluster HADOOP

2.3.1 Présentation YARN

L'architecture de la version 1 de HADOOP se limitait à deux composantes : HDFS et MAPREDUCE. La couche MAPREDUCE assurait deux principaux rôles : moteur de traitement pour les données stockées sur HDFS et gestionnaire de ressources du système lors de l'exécution des jobs. C'est à partir de la version 2 de HADOOP qu'une troisième composante a été intégrée à l'architecture. Il s'agit de YARN (*Yet Another Resource Negociator*). YARN est une solution conçue pour répondre aux limites de MAPREDUCE dans les tâches de gestion de ressources et d'orchestration des jobs. Avec l'intégration de YARN dans l'architecture de HADOOP, le rôle de gestionnaire de ressources et de planificateur de tâches qui était jusque-là dévolu à MAPREDUCE est transféré à YARN. Désormais, la composante MAPREDUCE servira uniquement de moteur de traitement des données. YARN propose, en effet, de séparer la gestion des ressources du cluster et la gestion des jobs MapReduce. Cette distinction permet ainsi de généraliser la gestion des ressources à d'autres applications HADOOP.

La figure suivante illustre l'évolution de l'architecture du cluster HADOOP entre la version 1 et 2.

Figure 7 : Evolution de l'architecture de HADOOP et l'intégration de YARN

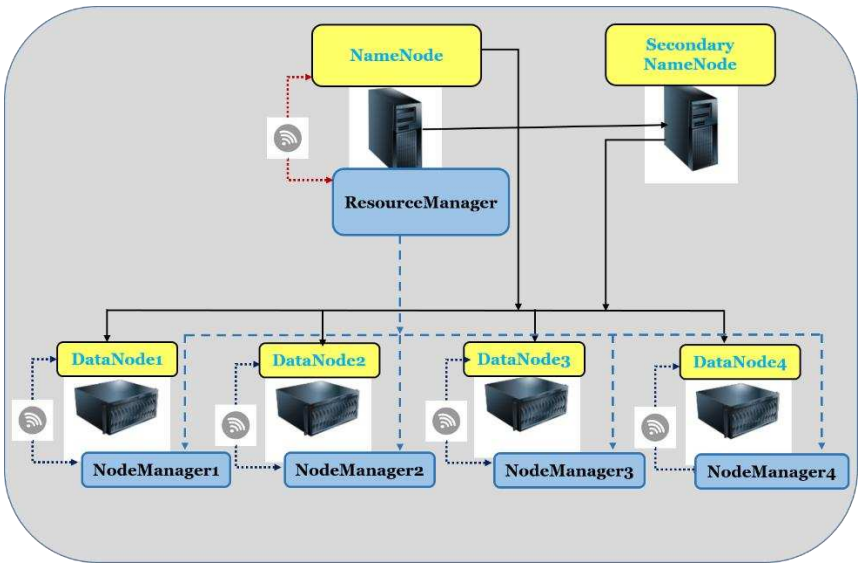


Avant l'intégration de YARN à l'architecture, on ne pouvait exécuter que des applications MapReduce sur le cluster HADOOP. En récupérant le rôle de gestionnaire de ressource et de planificateur de tâches, YARN a permis de décupler la puissance de HADOOP. Les jobs MapReduce ne sont plus les seules solutions pour tirer parti des données stockées sur le cluster HDFS. Contrairement à MAPREDUCE qui ne permet d'exécuter que des jobs MapReduce, YARN permet à plusieurs autres types

d'applications de tirer parti des capacités d'HDFS. Avec la coordination de YARN, plusieurs applications (y compris les jobs MapReduce) peuvent s'exécuter de façon simultanée et en parallèle.

Le figure ci-dessous illustre l'architecture du cluster YARN disposé en parallèle du cluster HDFS.

Figure 8 : Architecture du cluster YARN/ HDFS



Tout comme le cluster HDFS, le cluster YARN fonctionne sur une architecture maître-esclave où le nœud maître est représenté par le ResourceManager et les nœuds esclaves représentés par les NodeManagers. Le ResourceManager et les NodeManagers sont des processus JVM qui tournent en permanence sur le cluster (daemons). En règle générale, le ResourceManager tourne sur le même serveur que le NameNode du cluster HDFS tandis que les NodeManagers tournent côte à côte avec les DataNodes HDFS. Le rôle du ResourceManager est la planification et l'ordonnancement des ressources nécessaires à l'exécution des applications. Il s'appuie sur les NodeManagers dont le rôle est de fournir des containers (environnements cloisonnés d'exécution des jobs). Voir ci-dessous pour un détail sur chacune des composantes de l'architecture YARN.

2.3.2 Les composantes de l'architecture de YARN

Les principaux composants de l'architecture YARN sont : le RessourceManager, les NodeManagers, l'ApplicationMaster et les Containers. Les sections ci-dessous détaillent les caractéristiques et les rôles de chacun des composants.

2.3.2.1 Le ResourceManager (RM)

Le RM est l'orchestrateur des ressources du cluster HADOOP. C'est lui qui assure la répartition des ressources de calcul sur le cluster. Il a pour rôle de réceptionner les tâches soumises par les utilisateurs, d'ordonnancer ces tâches et d'allouer les

ressources nécessaires à leur exécution. En règle générale, le RM est un daemon (un processus JVM à longue vie) configuré sur le même serveur que le NameNode du cluster HDFS.

Le ResourceManager a deux principales composantes : le *Scheduler* (ordonnanceur) et l'*ApplicationManager*. Ces deux composantes jouent les rôles suivants. Dès qu'une tâche est reçue par le ResourceManager, c'est le Scheduler qui se charge de planifier la tâche. Pour cela, il tient compte de l'état et de la disponibilité des ressources sur le cluster. Dès lors que la tâche est planifiée, c'est l'ApplicationManager qui prend la suite pour s'assurer que la tâche s'exécute jusqu'au bout. Pour cela, il crée une instance de l'*ApplicationMaster* sur un des DataNodes du cluster. L'ApplicationMaster est un service temporaire qui se charge de réunir toutes les ressources nécessaires pour exécuter le job soumis. Il assure le monitoring de la tâche et rend compte à l'ApplicationManager du statut d'exécution de la tâche (voir ci-dessous pour plus de détails sur le rôle de l'ApplicationMaster).

2.3.2.2 Les NodeManagers (NM)

Les NodeManagers ont pour rôle de lancer et de contrôler les containers dans lesquels sont exécutés les jobs. Le NodeManager est un service (un daemon) configuré sur chaque DataNode du cluster HDFS. Alors que le ResourceManager gère les ressources de l'ensemble du cluster, le NodeManager lui s'occupe de gérer les ressources du DataNode et rend compte au ResourceManager avec lequel il est en communication permanente via des heartbeat calls.

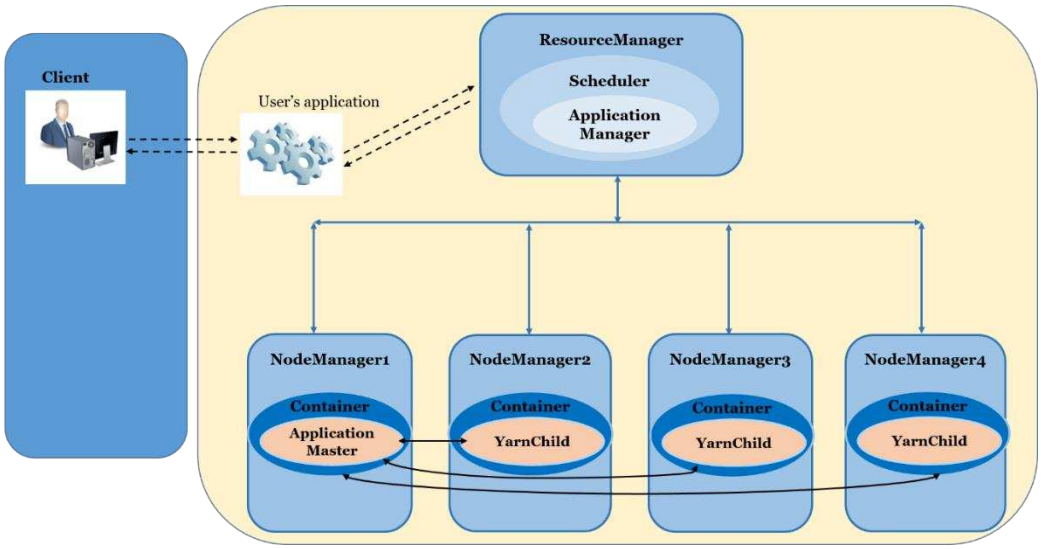
2.3.2.3 L'ApplicationMaster (AM)

L'AM est un framework disposant d'une librairie spécifique qui est instanciée par l'ApplicationManager sur un DataNode lors du lancement d'une tâche par YARN. Attention à ne pas confondre l'ApplicationMaster (AM) avec l'ApplicationManager. L'ApplicationManager est une composante du ResourceManager qui se charge d'instancier l'ApplicationMaster sur un DataNode en utilisant un container propre. Le rôle de l'AM est de négocier les ressources avec le ResourceManager et de travailler conjointement avec le NodeManager pour exécuter les tâches et de les surveiller.

2.3.2.4 Les containers

Un container est un assemblage de ressources nécessaires à l'exécution d'une application : mémoire RAM, CPU, Cores, etc. On distingue deux types de containers. Le container qui héberge l'ApplicationMaster (container AM) et les containers qui servent à l'exécution des tâches (container YarnChild). Le container AM est instancié par le RM sur un des NodeManagers alors que les containers YarnChild dans lesquels sont exécutées les tâches sont créés par le NodeManager lui-même à la demande de l'ApplicationMaster. La figure ci-dessous illustre les différentes composantes du service YARN.

Figure 9 : Les composants du service YARN



Les étapes ci-dessous décrivent les étapes d'exécution d'une application sur une architecture YARN :

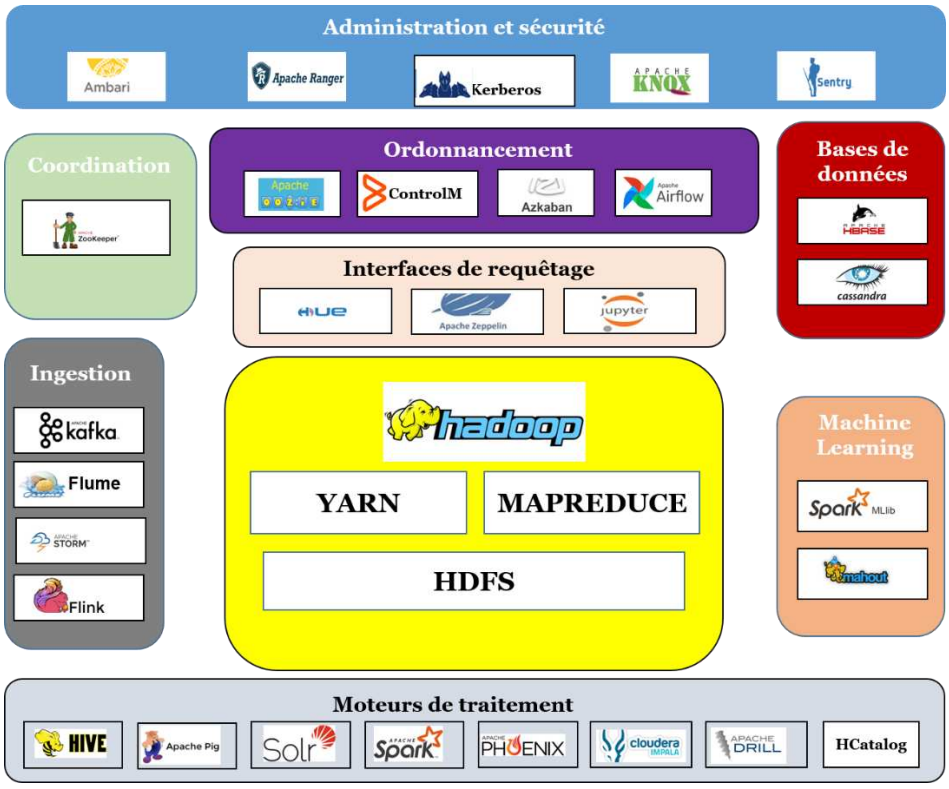
1. Le client soumet une application au ResourceManager.
2. Le ResourceManager (à travers l'ApplicationManager) alloue un container sur un NodeManager et crée une instance de l'ApplicationMaster dans ce container.
3. Une fois instanciée, l'ApplicationMaster demande un ou plusieurs containers en au ResourceManager vue de lancer les tâches
4. Le ResourceManager crée sur les NodeManager un ou plusieurs containers demandés par l'ApplicationMaster.
5. L'ApplicationMaster démarre alors une instance de la tâche dans un des containers qui lui ont été attribués. Il communique alors avec le NodeManager qui héberge ce container afin de pouvoir utiliser les ressources octroyées par le ResourceManager. L'Application communique en permanence avec le ResourceManager (ApplicationManager) sur le statut d'exécution des tâches.

3 L'ECOSYSTEME HADOOP ET LES PLATEFORMES DE DISTRIBUTION

3.1 L'écosystème HADOOP

Les fonctionnalités offertes par les couches de base de HADOOP à savoir HDFS, MAPREDUCE et YARN ont favorisé le développement de nombreuses briques technologiques afin de tirer parti au mieux de la puissance de HADOOP. Voici ci-dessous une liste (non exhaustive) des principales technologies qui forment l'écosystème HADOOP.

- **Bases de données** : HBase, Cassandra.
- **Moteurs de traitement** : Hive, Pig, HCatalog, Spark, Tez, Solr, Phoenix, Impala, Drill.
- **Outils d'ingestion et d'intégration de données** : Sqoop, Kafka, Flume, Flink.
- **Outils d'apprentissage automatique** : Mahout, Spark MLlib.
- **Synchronisation et coordination de ressources** : ZooKeeper.
- **Interfaces de requêtage** : Hue, Zeppelin, Jupyter.
- **Ordonnancement de traitements** : Oozie, Azkaban, Control M for Hadoop, Airflow.
- **Administration de plateformes et gestion de sécurité** : Ambari, Ranger, Sentry, Kerberos, Knox.



3.2 Les plateformes de distribution HADOOP

Le framework HADOOP, tout comme la plupart des composantes de son écosystème, est un projet « Open Source » maintenu par Apache Software Foundation. Par conséquent, les entreprises ont la possibilité de récupérer les binaires des codes sources, les modifier, les packager, les installer sur leurs environnements et les utiliser librement pour adresser les problématiques Big Data auxquelles elles font face. Cependant, le packaging et le déploiement d'une infrastructure HADOOP sont des opérations complexes qui nécessitent des expertises qui ne sont pas à la portée de toutes les entreprises. C'est pourquoi, certains acteurs spécialisés se proposent de fournir des solutions prépackagées appelées « ***distributions HADOOP*** ».

Une distribution HADOOP, à la différence de la version native Open Source, est une solution pré-packagée « production-ready » prête à être déployée sur une architecture en cluster. Une distribution HADOOP intègre plusieurs composantes de l'écosystème HADOOP. L'utilisation d'une distribution permet d'éliminer d'éventuels problèmes de compatibilité de versions entre les composants packagés. De nouveaux releases de la distribution sont aussi régulièrement livrés afin, non seulement, de corriger des bugs constatés sur les versions précédentes mais d'intégrer des fonctionnalités nouvelles. En plus du package logiciel, les distributions HADOOP offrent également des solutions de déploiement, d'administration et de monitoring du cluster.

Tout comme la version native, la plupart des distributions sont des produits Open Sources non commerciales. Généralement, les fournisseurs de distribution facturent uniquement les services de support de déploiement, de maintenance, de patchs spécifiques mais aussi des formations.

A l'heure actuelle, trois grands fournisseurs dominent le marché de la distribution HADOOP. Il s'agit notamment de Cloudera, Hortonworks et MapR. A noter que depuis 2019, Cloudera et Hortonworks se sont engagés dans une opération de fusion qui vise à proposer une distribution unifiée proposée sous le nom CDP (Cloudera Data Plateform). Les deux distributions continuent d'exister en parallèle. Mais à long terme, la distribution Hortonworks devrait disparaître pour laisser place à la solution unifiée estampillée CDP.

Voici les spécificités de chacune des trois grandes distributions actuellement disponibles sur le marché :

Cloudera Distribution Hadoop CDH

CDH est la première distribution de HADOOP qui intègre, en plus de composantes natives de HADOOP, quelques outils propriétaires comme Impala.

Hortonworks Data Platform HDP

HDP dispose, presque, des mêmes fonctionnalités que CDH. Mais contrairement à CDH, HDP reste une distribution entièrement Open Source. Elle se base uniquement

sur des composants sous licence Apache. Depuis sa fusion avec Cloudera en 2019, la maintenance de HDP est assurée par Cloudera en parallèle avec CDH.

MapR

La distribution MapR est une version reconditionnée et enrichie du package de base de HADOOP. En effet, à la différence de HDP et de CDH, MapR utilise de nombreux composants non natives de HADOOP. Par exemple, à la place de HDFS, MapR utilise son propre système de fichier : MapR-FS et MapR XD Distributed File and Object Store. Pour adresser la question de la High Availability, MapR utilise des solutions de snapshots ou de basculement avec état ("stateful failover"). Ce qui permet la reprise et la continuité de services même après une panne ou un sinistre.

Rappelons, tout de même, que le marché de la distribution HADOOP est un marché très instable dans le sens où de nombreuses entreprises qui étaient pourtant pionnières en la matière ont cessé de fournir les distributions pour se consacrer à des solutions spécialisées. C'est le cas par exemple de **Pivotal** qui a cessé de distribuer HADOOP. Cependant, en 2015, elle a noué un partenariat avec Hortonworks en vue de promouvoir HAWQ, son moteur de requêtage SQL-on-Hadoop, et Gemfire, en frontal. C'est le cas aussi de **IBM** qui a décidé de mettre un terme à la distribution de sa solution **IBM BigInsights** et de se consacrer à développer un partenariat avec Hortonworks en vue de proposer des solutions Hortonworks à ses propres clients.

A l'image de **Pivotal** et de **IBM**, d'autres acteurs du Big Data ont préféré adopter des stratégies intermédiaires consistant à intégrer à leur offre Big Data un package HADOOP construit autour d'un nombre limité de composants. C'est sur ce principe que se sont développées les offres de services comme : Azure HDInsight, Amazon Elastic MapReduce (EMR), Alibaba Cloud E-MapReduce (EMR), Google Cloud Dataproc, Qubole Data Service (QDS), etc.

4 HIVE

4.1 Fonctionnalités

Initialement développé par les ingénieurs de Facebook en 2008, Apache HIVE est un moteur de traitement permettant de requêter les données stockées sur HDFS dans un langage proche du SQL. Notons d'entrée que Hive n'est pas une base de données relationnelle. C'est une brique technologique qui offre des outils pour accéder aux données stockées sur HDFS présentant une structure et un schéma spécifiés sous forme de table.

La création d'une table HIVE se fait d'une manière quasi-similaire à la création d'une table dans un RDBMS classique. Les structures des tables (métadonnées) sont stockées de manière persistante dans une base de données relationnelle appelée metastore. Par défaut c'est Derby qui sert de metastore. Mais d'autres bases de données comme MySQL ou Postgres peuvent également servir de metastore. Les tables HIVE prennent en charge plusieurs formats de fichiers de stockage sur HDFS (RCFile, TEXTFILE, ORC, PARQUET, AVROFILE, …) mais aussi plusieurs formats de stockage hors HDFS tels que Amazone S3 et Azure Blob Storage.

Le langage de requêtage de HIVE est HiveQL (ou HQL). La maturité actuelle de ce langage fait de HIVE le moteur de requêtage SQL-like les plus utilisé sur HADOOP.

Bien que dans sa version d'origine HIVE ait été conçu pour s'exécuter uniquement sous forme de tâches MapReduce, les versions ultérieures ont apporté plus de flexibilité. Les requêtes HIVE peuvent désormais s'exécuter aussi bien avec le MAPREDUCE mais aussi avec d'autres moteurs d'exécution comme TEZ, SPARK ou LLAP.

Notons, tout de même, que dans certaines distributions HADOOP comme HDP 3.0, le moteur d'exécution MAPREDUCE a été abandonné au profit de TEZ qui devient l'engin d'exécution par défaut de HIVE. TEZ permet d'améliorer les performances des requêtes en utilisant les expressions de Directed Acyclic Graphs (DAG). On peut également utiliser le moteur d'exécution SPARK qui, comme TEZ, utilise les expressions DAG mais avec plus de performance. Avec le moteur SPARK, toute l'exécution du plan se fait sur la mémoire du nœud de traitement évitant ainsi les multiples Read/Write sur le disque. LLAP peut aussi servir de moteur d'exécution aux requêtes Hive.

4.2 Architecture de Hive

L'architecture de HIVE est constituée de trois principales couches :

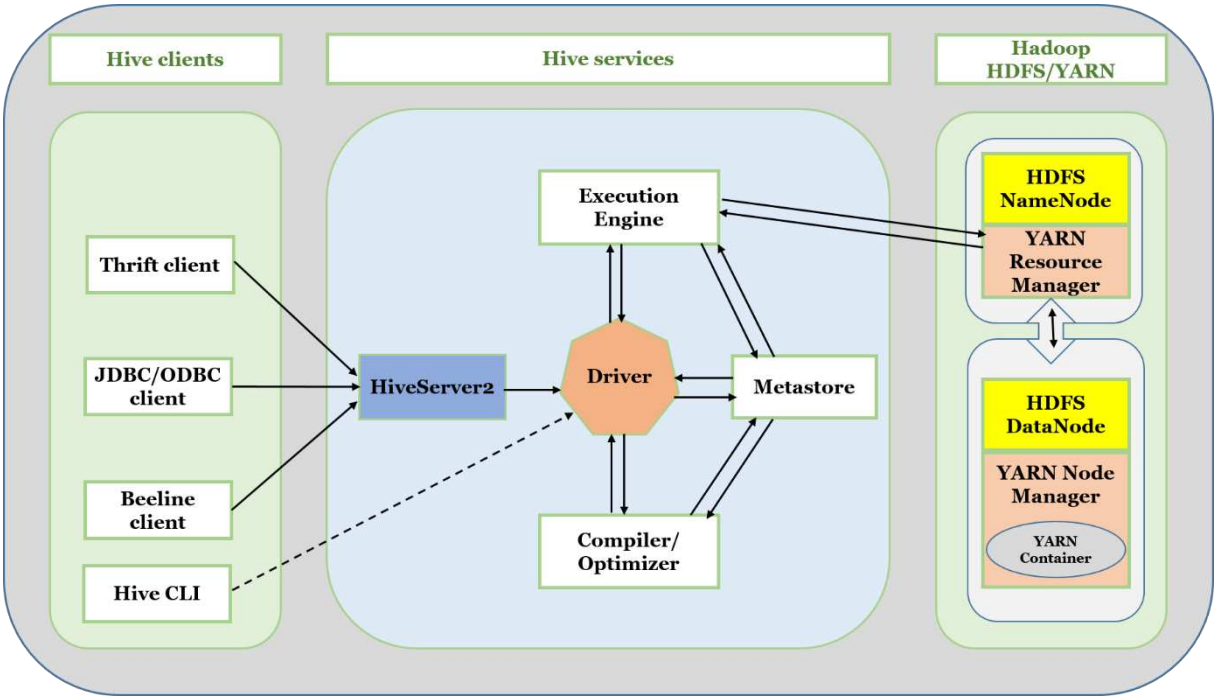
1) - la couche de requêtage (Hive Clients),

2) - la couche de traitements des requêtes (Hive Services)

3)- la couche d'exécution et de Stockage (HADOOP HDFS/YARN).

La figure ci-dessous illustre les composition des trois couches.

Figure 10 : Les composantes de l'architecture Hive



4.2.1 Hive Clients

On distingue deux catégories de clients Hive : les clients distants (externes) et les clients intégrés.

4.2.1.1 Les clients externes

Parmi les clients distants, on distingue :

- *Les clients Thrift* : Un client Thrift est toute application externe permettant d'interroger HIVE via un protocole Thrift. Les protocoles Thrift sont implémentés dans plusieurs langages tels que Java, Python, Ruby, C ++. Cela permet aux utilisateurs d'interroger la même source (serveur) selon le langage qu'ils souhaitent.
- *Les clients externes JDBC/ODBC* : ici, il s'agit de toute application externe permettant d'interroger Hive par l'intermédiaire d'un connecteur (pilote) spécifique. Parmi ces types de client, on peut citer par exemples : SQuirreL SQL Client, SQL Workbench, DbVisualizer, Dbeaver, Hue, Ms Excel.

4.2.1.2 Les clients intégrés

D'autres clients sont directement intégrés à Hive. Il s'agit notamment de Hive CLI (Hive Command Line Interface) ou de Beeline.

- Hive CLI: permet d'interroger Hive en passant directement par le metastore et HDFS.
- Beeline : est un client JDBC léger qui passe par HiveServer2 pour soumettre les requêtes Hive.

4.2.2 Hive Services

Les principales composantes des services Hive sont : le HiveServer2, le metastore, le Driver, le compiler/Optimizer et le moteur d'exécution (Execution Engine).

4.2.2.1 HiveServer2

HiveServer2 est le service à travers lequel les clients Hive soumettent leurs requêtes à Hive. HiveServer2 est bâti sur du Thrift et est parfois appelé *Thrift Server*. Le Thrift Server est une interface qui agit comme une passerelle permettant aux clients de soumettre des requêtes à Hive dans divers langages de programmation. Il prend en charge les connexions JDBC et ODBC locales et distantes.

En plus de recevoir les requêtes de clients, HiveServer2 offre deux nouvelles fonctionnalités : la gestion de l'authentification du client et la gestion des requêtes concurrentes. En effet, pour chaque connexion client, HiveServer2 crée un nouveau contexte d'exécution (connexion + session). L'interface RPC (Remote procedure call) de HiveServer2 permet au serveur d'associer le contexte d'exécution Hive avec le thread qui sert la requête client. Cette interface implémente un service Thrift pour communiquer avec les clients et exécuter leurs requêtes.

4.2.2.2 Hive Metastore

C'est un repository central qui sert de namespace des tables Hive. Le metastore est généralement une base de données relationnelle standard qui fournit une interface Thrift pour interroger et manipuler les métadonnées. Comme son nom l'indique, le metastore stocke les métadonnées, c'est-à-dire les informations sur la structure des tables Hive : type des colonnes, partitions, localisations HDFS, etc. Le metastore stocke également les informations sur les *sérialiseurs* et des *désérialiseurs*, objets souvent nécessaires lors des opérations de lecture/écriture. En résumé, le metastore sert de catalogue de toutes les données HDFS pouvant être accéder via Hive.

Le metastore peut être configuré dans deux modes : le mode distant (Remote) ou le mode intégré (Embedded). Dans le mode distant : le métastore est un service Thrift et est surtout utile pour les applications non-Java. Car Thrift est un système de type

Language-agnostic. Dans le mode intégré, le Driver, et le metastore s'exécutent sur la même Machine Virtuelle Java (JVM). Cependant le mode intégré ne peut supporter qu'une session Hive à la fois. On ne peut pas ouvrir plusieurs sessions Hive en parallèle. Ce qui fait qu'il n'est pas adapté pour les usages en production. Il existe deux autres modes à savoir - *local* et *distant* – qui, eux, peuvent supporter plusieurs sessions Hive.

Le metastore peut associer jusqu'à 51 tables de métadonnées à une seule table Hive. Mais seulement 3 de ces tables fournissent la majorité des informations sur la structure de la table Hive. Il s'agit en particulier des tables : TBLS, DBS et COLUMNS_V2.

- La table TBLS stocke les métadonnées de la table Hive telles que : nom de la table, propriétaire, type de table (interne ou externe).
- La table DBS fournit les informations sur la base de données hébergeant la table HIVE : nom de la base de données, propriétaire, localisation HDFS de la base de données)
- La table COLUMNS_V2 fournit les détails sur les colonnes de la table Hive : nom de colonne, type de données, etc.

Les couches d'extensions du metastore : Hcatalog et WebHcat

L'écosystème HADOOP offre de nombreux outils pour accéder aux informations stockées dans le metastore Hive afin de pouvoir les utiliser dans d'autres applications. Il s'agit notamment de Hcatalog et WebHcat.

- HCatalog est un outil de gestion de tables et de stockage pour HADOOP. Il permet aux utilisateurs disposant de différents outils de traitement de données tels que Pig, Spark SQL ou MapReduce, de lire et d'écrire les données sur HDFS en utilisant les informations stockées dans le metastore Hive.
- WebHCat est l'API REST pour HCatalog. Il offre une interface HTTP pour effectuer des opérations sur les métadonnées Hive. Il fournit également un service pour exécuter différents jobs : MapReduce, Pig, Hive, SparkSQL.

4.2.2.3 Le Driver

Le driver est la composante centrale à l'architecture de Hive. Il a pour rôle de coordonner les différentes étapes de l'exécution des requêtes. Il reçoit les instructions HiveQL soumises par l'utilisateur via les Hive clients. Il crée les sessions Hive en vue de traiter la requête. Il envoie la requête au Compiler/Optimizer pour apporter des optimisations. Il transmet le plan d'exécution à l'Execution Engine pour la réalisation des opérations. Un ensemble de fichiers jar embarqués avec le package Hive aident à convertir les requêtes HiveQL en tâches exécutables suivant le moteur d'exécution choisi.

4.2.2.4 Le Compiler/Optimizer

Le compiler est le service qui réceptionne les requêtes envoyées par le Driver. Une fois que la requête est reçue, le Compiler effectue une analyse sémantique de la requête, vérifie et valide la syntaxe des requêtes. Ensuite, il se base sur les métadonnées stockées dans le metastore pour générer un plan d'exécution de la requête. D'une manière générale, le plan généré est un DAG (Directed Acyclic Graph) où chaque étape est un job de Map/Reduce, une opération sur HDFS et une opération sur les métadonnées.

Le plan d'exécution généré par le compiler est généralement soumis à l'Optimizer en vue d'éventuel optimisation. L'Optimizer est un service complémentaire qui effectue des opérations de transformation sur le plan d'exécution et subdivise les tâches pour plus d'efficacité et de performance lors de l'exécution des tâches.

4.2.2.5 Le moteur d'Exécution : Execution Engine

Le moteur d'exécution est l'interface entre Hive et la couche HADOOP YARN/HDFS. Suite à la compilation et l'optimisation des tâches par le Compiler/Optimizer, le moteur d'exécution reçoit le plan d'exécution et déroule les actions selon l'ordre des dépendances défini dans le plan. L'exécution du plan prend place dans des containers mis à disposition par HADOOP YARN. Plusieurs moteurs d'exécution peuvent être utilisés : MAPREDUCE, TEZ, SPARK ou LLAP.

4.2.3 La couche d'exécution et de stockage : HADOOP YARN/HDFS

4.2.3.1 YARN

Une fois que la requête Hive a été convertie en jobs MR, TEZ ou SPARK selon le moteur exécution choisi, la planification des tâches est assurée par HADOOP YARN. En effet, YARN se charge de la gestion des ressources et de la coordination des différentes étapes de l'exécution des jobs. Pour cela, il crée les containers et réunit les ressources nécessaires à l'exécution des tâches (voir la section sur les étapes d'exécution d'une application sur YARN).

4.2.3.2 HDFS

Hive étant une surcouche de HADOOP, il utilise naturellement le FileSystem HDFS pour la lecture et le stockage des données. Bien entendu, l'exécution des requêtes Hive peuvent avoir lieu sur d'autres espaces de stockage comme par exemple AWS S3, etc…

4.2.4 Les étapes d'exécution d'une requête Hive

Les étapes ci-dessous résument les rôles de chaque composante de l'architecture Hive dans l'exécution d'une requête.

1. Lorsqu'un client Hive soumet une requête, celle-ci est d'abord envoyée à HiveServer2 qui se charge ensuite de la transmettre au Driver
2. Le Driver initialise la session Hive et transmet la requête au Compiler/Optimizer. Ces deux services vont se charger, à leur tour, de générer le plan d'exécution et d'y apporter les optimisations nécessaires.
3. Lorsque le compiler a généré le plan d'exécution, il communique avec le Hive Metastore pour récupérer toutes les métadonnées nécessaires.
4. Le metastore renvoie alors les informations demandées par le Compiler.
5. Le Compiler transmet au Driver le plan d'exécution généré en vue de son exécution
6. Le Driver transfère le plan d'exécution à l'Execution Engine choisi : MAPREDUCE, TEZ ou SPARK.
7. L'Execution Engine (EE) envoie les tâches au gestionnaire d'exécution des applications dans HADOOP (ex : YARN), qui se alors charge alors de planifier l'exécution des tâches sur la couche HDFS.
8. A la fin de l'exécution des tâches, les résultats sont récupérés dans l'Execution Engine en vue d'être transmis au Driver.
9. Le Driver récupère les résultats à partir de l'Execution Engine et les transmet au client.

4.3 Evolution de l'architecture de Hive : la couche LLAP

L'architecture Hive est en constante évolution. Depuis la version 2 de Hive plusieurs fonctionnalités ont été ajoutées à l'architecture en vue d'obtenir une meilleure performance dans l'utilisation de l'outil. Deux évolutions majeures récentes ont retenu notre attention, il s'agit de l'ajout des propriétés ACID aux Tables et l'ajout d'une couche de requêtage LLAP. Ci-dessous les détails sur chacune des évolutions.

4.3.1 Les propriétés ACID

Pendant longtemps de nombreuses opérations de mise à jour de table n'étaient pas possible dans Hive. Par exemple, pour mettre à jour ou supprimer une ligne de données, il fallait d'abord supprimer l'ensemble de la partition ou vider entièrement la table avant de réinsérer les lignes avec les nouvelles valeurs. Désormais Hive supporte les opérations transactionnelles ACID. Les propriétés ACID (Atomicité, Cohérence, Isolation, Durabilité) sont des propriétés fondamentales que les transactions doivent respecter au sein d'une base de données en vue de garantir leur validité même en cas d'erreur ou de pannes.

En prenant en compte les propriétés ACID, Hive permet, par exemple, à un utilisateur d'insérer des lignes dans une partition tandis qu'au même moment un autre utilisateur lit les données déjà disponibles dans cette même partition. Les opérations se passent sans aucune interférence aux niveaux des lignes. Avec les propriétés ACID, Hive

supporte désormais la plupart des opérations CRUD : Create, Read, Update, Delete, Merge.

4.3.2 La fonctionnalité LLAP

LLAP (Low Latency Analytical Processing) est une couche supplémentaire dans l'architecture Hive qui permet la mise en cache des tables pour une meilleure performance des requêtes. Les services LLAP sont gérés et exécutés en tant que service YARN de longue durée. Avec une configuration LLAP, toutes les données d'une table sont chargées et disponibles en permanence en mémoire. Ce qui permet d'améliorer considérablement la performance des requêtes. Le temps de réponse d'une requête envoyée via LLAP est généralement de quelques secondes, alors que cette même requête prendrait plusieurs minutes lorsqu'envoyées suivant le mode normal.

5 HBASE

5.1 Fonctionnalités

Apache Hbase est un système de base de données distribuée orientée-colonne. Il appartient à la catégorie des SGBD dit NoSQL « Not Only SQL ». Les SGBD NoSQL ont été conçues pour répondre aux limitations des SGBD traditionnels en particulier le problème de latence des requêtes sur de grosses volumétries de données. HBase est une solution qui permet un accès aléatoire en temps réel à des tables pouvant contenir des millions de colonnes et des milliards de lignes. L'accès aléatoire (aussi appelé accès direct) est une propriété qui permet de lire ou d'écrire une donnée en se rendant directement à l'endroit où la donnée doit être écrite ou lue sans passer par les données précédentes tel dans l'accès séquentiel. Cette propriété permet ainsi d'améliorer la performance des requêtes.

HBase est un data store qui s'exécute au-dessus de HDFS. Il reprend les concepts et les fonctionnalités de BigTable de Google qui s'appuie sur le GFS (Google File System).

Bien que Hbase ne soit pas un outil de requêtage SQL standard, il est possible d'y intégrer une couche externe pour pouvoir exécuter des requêtes SQL. Par exemple, les outils comme Apache Phoenix, Hive et Pig s'intègrent facilement à HBase pour un accès SQL.

5.2 Structure d'une table Hbase

La structure d'une table HBase est définie autour de quatre niveaux d'information :

- la ***Rowkey*** : qui représente la clé primaire de la table
- la ***Column Family*** : qui représente le regroupement d'un ensemble de colonnes.
- la ***Column qualifier***: qui représente l'attribut (le nom) de la colonne. Ex : age, sexe, etc… Un ensemble de column Qualifier sont regroupées pour former une column Family. Voir plus bas, pour plus de détails.
- la ***Qualifier Value*** (ou Cell) : c'est l'unité permettant de stocker la valeur d'une Qualifier. Ex : la value 38 ans pour la Qualifier Age. Ou encore la Value Masculin pour la Qualifier Sexe, etc…
- Le ***Timestamp*** : est un horodateur qui permet d'indiquer l'instant auquel la qualifier value a été insérée dans la table.

Le tableau ci-dessous illustre l'exemple de structure d'une table Hbase conçue pour stocker les infos sur les patients suivis dans des protocoles cliniques.

Tableau 1 :Structure d'une table Hbase

Rowkey	**Column Family :** *biometrique_infos*				**Column Family :** *contact_infos*		
protoc1_patient_001	Column Qualifier : *Age* Value : *30* TimeStamp : *1622982369000*	Column Qualifier : *Poids* Value : *75* TimeStamp : *1622982383000*	Column Qualifier : *Taille* Value : *170* TimeStamp : *1622982383000*	Column Qualifier : *Freq_cardiaque* Value : *80* TimeStamp : *1622982408*	Column Qualifier : *adresse* Value : *2 boulevard du marché, 12345 Grand Ville*	Column Qualifier : *num_tel* Value : *+001 213 145 644* TimeStamp : *1622982429*	Column Qualifier : *email* Value : *testEmail1@test.com* TimeStamp : *1622982429000*
protoc1_patient_002	Column Qualifier : *Age* Value : *40* TimeStamp : *1622982369000*	Column Qualifier : *Poids* Value : *80* TimeStamp : *1622982383000*	Column Qualifier : *Taille* Value : *160* TimeStamp : *1622982383000*	Column Qualifier : *Freq_cardiaque* Value : *60* TimeStamp : *1622982408*	Column Qualifier : *adresse* Value : *4 boulevard du marché, 12345 Grand Ville*	Column Qualifier : *num_tel* Value : *+001 648 79 0 123* TimeStamp : *1622982429*	
protoc2_patient_001	Column Qualifier : *Age* Value : *20* TimeStamp : *1622983027000*	Column Qualifier : *Poids* Value : *72* TimeStamp : *1622983027000*		Column Qualifier : *Freq_cardiaque* Value : *70* TimeStamp : *1622983027*	Column Qualifier : *adresse* Value : *9 boulevard du marché, 12345 Grand Ville*		Column Qualifier : *email* Value : *testEmail3@test.com* TimeStamp : *1622983027000*

Aide à la lecture :

La Rowkey : sert de clé primaire et permet d'identifier toute les informations rattachées à un patient donné. Pour lire ou écrire les informations relatives à un patient, il faut obligatoirement passer par la rowkey.

La Column Family : Deux Columns Families sont renseignées dans la table : *biometrique_infos* et *contact_infos*. D'une manière générale, les column families servent à regrouper les attributs qui se rattachant à un aspect particulier du sujet. Ici, par exemple, la column family *biometrique_infos* stocke les attributs relatifs aux données biométriques du patient. Alors que la column Family *contact_infos* stocke les attributs relatifs au contact du patient. Notons que rattacher tel attribut à telle ou telle Column familly relève d'un choix arbitraire. Néanmoins, il peut être pertinent de regrouper dans une même Column family les attributs liés un aspect particulier du sujet étudié. Par exemple dans un contexte commercial, on peut choisir par exemple de stocker les informations métier dans une column Family dédiée et les informations techniques dans une autre column Family.

La Column qualifier : La Qualifier représente l'attribut c'est-à-dire le nom de la colonne comme dans un SGBD classique. Cependant, contrairement à un SGBD classique, la Colunm qualifier n'est pas obligatoire au niveau d'un Rowkey dans la structure d'une table HBase. En effet, dans une table SGBD traditionnelle, les noms

des colonnes doivent être déclarées lors de la création de la table ou lors de la modification de sa structure. Mais Hbase n'impose pas ce type de contrainte. On remarque par exemple, dans le tableau ci-dessus que la Column qualifier *email* n'est pas renseignée pour la Rowkey *protoc1_patient_002*. Et que les Column qualifiers *taille* et *num_tel* sont manquantes pour la Rowkey *protoc2_patient_001*. Comme l'insertion des données se fait au niveau de la Rowkey, alors la Column qualifier peut être renseignée pour certaines Rowkeys et être manquante pour d'autres.

Contrairement à une base de donnée traditionnelle, la Column qualifier n'est pas renseignée lors de la création de la table Hbase, elle est juste renseignée lors de l'insertion des données. De ce point de vue, Hbase offre une grande flexibilité d'usage.

Notons que lorsqu'une Column qualifier est manquante pour une Rowkey, cela signifie que la valeur de cette Column qualifier est égale à nulle pour cette rowkey. Et quand une Column qualifier a une valeur nulle pour une Rowkey en Hbase, il n'est pas nécessaire de définir cette Column qualifier pour cette Rowkey.

La Qualifier Value (Cell) : C'est la valeur associée d'une Column qualifier. Contrairement à de nombreux système de bases de données où l'on a la possibilité de distinguer les valeurs des colonnes selon leur type (entier, chaîne de caractère, décimal, etc…), Hbase stocke les valeurs uniquement sous forme de Byte[]. Ainsi, tous les attributs quel que soit leur type initial sont stockés dans la table Hbase sous le format Byte[].

Le TimeStamp : c'est une information qui indique, pour une Column Qualifier donnée, l'instant auquel la Qualifier Value a été insérée dans la table. En effet, dans Hbase, l'insertion d'une nouvelle value d'une Column Qualifier n'écrase pas les anciennes valeurs, elle ajoute une nouvelle valeur à une liste contenant les anciennes valeurs. Grâce à l'horodatage par le timestamp, on peut récupérer à tout moment une valeur insérée dans la table dans le passé. Pour une Column qualifier, chaque valeur historisée représente une *version*. L'une des particularités de Hbase est sa capacité à empiler différentes versions d'une même qualifier value. Le versionning des valeurs se fait en associant un Timestamp à chaque valeur insérée. Toutefois, par défaut, Hbase ne garde qu'une seule version. Ce qui revient, de facto, à écraser chaque fois l'ancienne valeur lorsqu'une nouvelle est insérée. Mais, lors de la création de la table, Hbase laisse la possibilité de définir le nombre maximum de versions à garder. On peut alors fixer à souhait le nombre maximum de versions à conserver.

5.3 Requêtage d'une table Hbase

Le requêtage d'une table Hbase se fait à travers les quatre opérations CRUD suivantes : : PUT, GET, SCAN et DELETE.

- le PUT permet l'insertion des données sur une nouvelle Rowkey ou une mise à jour des données d'une Rowkey existante.
- le GET permet de lire les données d'une Rowkey.

- le SCAN permet de lire les données de toute la table Hbase. Mais on peut aussi effectuer un scan limité à un ensemble de Rowkey. Pour cela, il faut définir un critère de filtrage et l'associer à la requête de SCAN. Le critère de filtrage pourrait être défini par exemple comme : « Rowkeys contenant un substring donné », « Rowkeys insérééés ou mise à jour entre deux valeurs de timestamp », « Column qualifier ayant une valeur spécifiée », etc.... Notons que la fonction SCAN est l'équivalent de la clause SELECT… WHERE…dans le langage SQL.

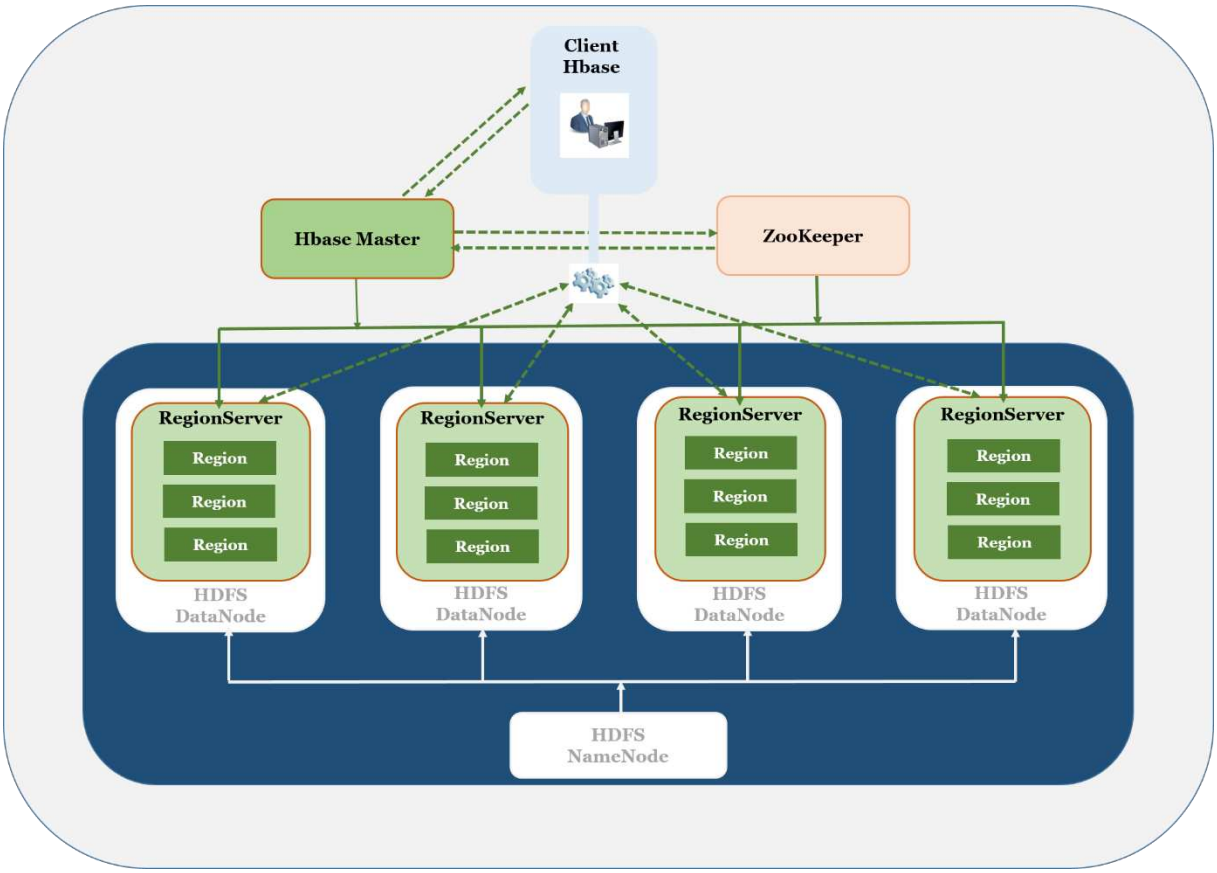
  Par ailleurs notons que le GET peut être considéré comme un cas particulier du SCAN. La seule différence est que le GET permet de cibler directement le RegionServer qui contient la Rowkey alors que le SCAN nécessite parfois d'interroger tous les RegionServers pour retrouver l'ensemble des Rowkey répondant aux critères définis dans le filtre.
- le DELETE est une action qui permet de supprimer une Rowkey et l'ensemble des données qui lui sont rattachées. Notons toutefois que les données supprimées par l'action DELETE ne sont effectivement supprimées (sur HDFS) qu'après une opération de Major Compaction sur la table Hbase (voir plus bas pour plus de détails sur la Major Compaction).

  L'une des particularités de Hbase est la possibilité qu'elle offre pour supprimer automatiquement les données au bout d'un certain temps appelé TTL (Time-To-Live). Le TTL est un paramètre qui s'applique à une Column Family lors de la création de la table qui permet d'indiquer au bout de combien de temps les données sur cette Column Family seront automatiquement détruites. Cependant tout comme pour l'action DELETE, la destruction effective des données sur HDFS par l'expiration TTL ne prend effet que lorsqu'une Major Compaction est réalisée sur la table. D'où l'importance de programmer cette action à une fréquence régulière.

5.4 Architecture d'un cluster Hbase

Un cluster HBase fonctionne en architecture maître-esclaves. Le nœud maître est le HMaster, et les nœuds esclaves sont les RegionsServers. Cependant l'architecture du cluster Hbase ne fonctionne pas en toute autonomie. D'une part, elle se base sur le système de fichier HDFS du cluster HADOOP pour le stockage des données. D'autre part, Hbase s'appuie sur le cluster Zookeeper pour synchroniser l'utilisation des ressources, sauvegarder les métadonnées du cluster et assurer la disponibilité du cluster. La figure ci-dessous illustre l'architecture générale simplifiée d'un cluster HBase.

Figure 11 : Architecture du cluster Hbase



5.4.1 Le client Hbase

Pour soumettre des requêtes, le client contacte dans un premier le HMaster. Celui-ci lui fournit les adresses des RegionsServers qui pourront répondre à la demande du client. Le client contacte alors ces RegionsServers qui se chargent de traiter la requête. La communication entre le client et le HMaster ou les RegionsServers se fait suivant la procédure RPC (Remote procedure call).

5.4.2 Le Hbase Master (HMaster)

Le HMaster assure la surveillance et la coordination de toutes les instances de RegionServers dans le cluster. Il sert également d'interface pour l'admin du cluster Hbase, la gestion et la maintenance des métadonnées. S'agissant du rôle de coordination, le HMaster gère l'affectation des Regions au RegionsServers. Il assure les fonctions critiques telles que l'équilibrage de charge des Regions entre les RegionServers, le basculement des Regions en cas de panne d'un RegionServer mais aussi le split des Regions au sein d'un RegionServer.

Pour ce qui concerne le rôle d'admin, le HMaster agit comme une interface pour créer, mettre à jour ou supprimer des tables dans HBase. HMaster assure aussi la gestion et la mise à jour des métadonnées sur les tables et sur l'état du cluster en rapport direct avec Zookeeper. Au niveau de la table, on retrouve des opérations comme : createTable,

deleteTable, enable, disable. Au niveau de la Column Family, il y a des opérations comme : add Column, alter Column. Et au niveau des Regions, on a des opérations comme : move, assign. Et au niveau du cluster, le HMaster communique en permanence avec ZooKeeper pour mettre à jour les métadonnées sur les RegionsServers et les Regions que ceux-ci hébergent.

5.4.3 Le Hbase RegionServer(HRegionServer)

Les RegionServers assurent le stockage, la distribution et la réplication des données des tables Hbase. Ils opèrent sous la supervision du HMaster qui coordonne l'exécution des tâches et gère les métadonnées.

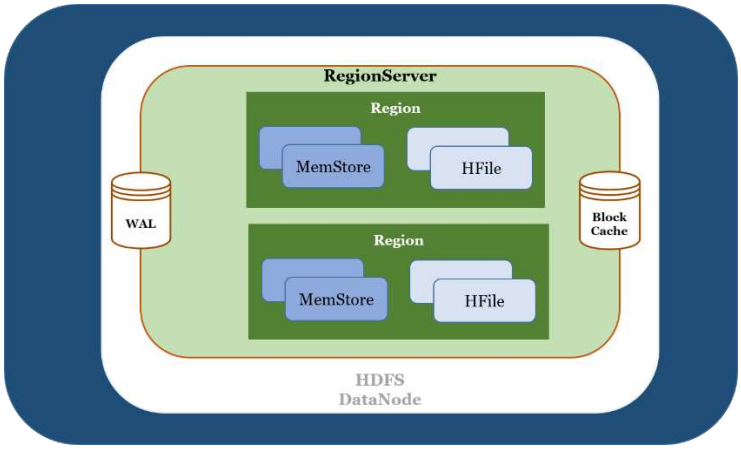
Chaque nœud du cluster Hbase possède un RegionServer qui stocke les données d'une ou de plusieurs régions.

Le contenu des RegionServers est indexé dans HMaster. Ce qui permet à celui d'aiguiller les clients vers les nœuds adéquats lors des opérations de lecture et d'écriture. Le transfert de données entre les clients et les RegionServers se fait via une connexion directe sans l'intermédiation du HMaster.

Les RegionServers tournent au-dessus des DataNodes de HADOOP et toutes les données sont stockées dans des fichiers HDFS. Ce qui permet la localité des données. La localité fait référence au fait de placer les données à l'endroit où elles seront traitées et utilisées. La co-location du RegionServer avec le DataNode HADOOP permet au NameNode HDFS de conserver les informations de métadonnées sur les blocs de fichiers et leurs réplications.

La figure ci-dessous illustre l'architecture fonctionnelle d'un RegionServer Hbase.

Figure 12 : Architecture d'un RegionServer Hbase



Les différents éléments qui forment cette architecture sont :

5.4.3.1 Le WAL (Write Ahead Log)

Le WAL est un cache rattaché à chaque RegionServer. C'est un fichier tampon permettant de stocker les données nouvellement insérées dans la table Hbase mais qui n'ont pas encore été persistées sur le disque HDFS. Par exemple, lorsqu'un client envoie une requête PUT à un RegionServer, les données sont d'abord écrites dans le WAL qui est un journal d'écriture anticipée. Ensuite, elles sont transférées dans le MemStore. Le WAL sert surtout de point de restauration des données en cas de panne d'un serveur avant de persister les données en mémoire sur le disque.

5.4.3.2 Le MemStore

Le MemStore est un cache d'écriture dont le rôle est de stocker les données qui ont été insérées dans la table Hbase mais n'ont pas encore été persistées sur le disque (c'est-à-dire dans les Hfiles). Le MemStore reçoit les données à partir du WAL et les retient jusqu'à ce qu'une taille maximale soit atteinte. Il les persiste ensuite dans les Hfiles.

Le MemStore trie les Rowkeys. Ce qui permet de bien disposer les données sur le disque une fois persistées dans le Hfile. Il y a un MemStore par Column Family, autrement dit, toutes les données d'une Column Family transitent par un même MemStrore avant d'être envoyées vers des Hfiles qui permettront par la suite d'alimenter la (les) Region(s).

5.4.3.3 Le *Hfile*

Le Hfile est un fichier qui stocke les données des Column Familiy triées suivant les Rowkeys. Les Hfiles sont créées ou alimentées dès que les MemStores sont remplis.

Un Hfile ne contient que les données d'une même Column Family. En d'autres termes, les données de deux Column family ne peuvent pas se retrouver dans un même Hfile. En revanche les données d'une même Column family peuvent être réparties sur plusieurs Hfiles. En effet, lors d'une opération d'écriture, les données sont d'abord écrites dans un WAL. Ensuite elles sont déplacées dans la Memstore. Lorsque la taille du Memstore dépasse la valeur maximale fixée, tout son contenu est écrit dans un HFile stocké sur le disque HDFS. Chaque fois que le contenu d'un Memstore est écrit sur le disque, de nouveaux HFiles sont créés. Le nombre de HFiles augmente donc au fur et à mesure que de nouvelles données sont insérées dans la table Hbase.

Cependant pour limiter le nombre de fichiers et optimiser l'usage de l'espace disque HDFS, les Hfiles doivent être continuellement mergés. Ces opérations de merge sont appelées « Compactions ». On distingue deux types d'opération de compaction : la « Minor Compaction » et la « Major Compaction ». Dans la Minor Compaction, HBase sélectionne automatiquement les plus petits HFiles et les merge dans les gros HFiles tout en triant les Rowkeys. Par contre, dans la Major Compaction, tous les petits Hfiles sont automatiquement mergées pour créer un nouveau Hfile. A rappeler que c'est lors de la Major Compaction que les Rowkeys supprimées (avec la commande DELETE) ou expirées (avec l'option TTL) sont effectivement supprimées sur HDFS.

5.4.3.4 Le BlockCache

Le BlockCache est un espace mémoire-tampon dont le rôle est de stocker les données fréquemment lues dans la table Hbase. Contrairement au WAL qui est un cache d'écriture, le BlockCache est un cache de lecture. C'est une fonctionnalité qui permet d'améliorer significativement la performance des requêtes GET. Lorsque le BlockCache est plein, les données les moins réutilisées récemment sont automatiquement supprimées du cache.

5.4.4 La Hbase Region (HRegion)

Une Region est un regroupement des données issues d'une Column family. Chaque Region contient les données d'une plage contigue de Rowkeys stockés.

Les données situées dans des Regions qui correspondent à un set de Hfiles proviennant d'une même table. Les Regions sont gérées par les RegionServers. Chaque RegionServer peut posséder une ou plusieurs Regions.

La taille par défaut d'une Region est de 256 Mo. Mais cette valeur peut être modifiée selon les besoins. Notons que lorsqu'une Region se remplit jusqu'à atteindre la taille maximale définie, elle se scinde automatiquement en deux pour donner naissance à deux nouvelles Regions de tailles identiques. Cette scission est ensuite directement notifiée au HMaster qui mettra alors à jour les métadonnées en notifiant à son tour le Zookeeper. Ce mode de scission des Region est qualifiée de Region auto-Split. Il existe plusieurs autres modes de split de Regions. On distingue le *Pre-Split* qui consiste à fixer un nombre prédéfini de Regions à la création de la table ; le *Force-Split* qui consiste à spliter les Regions déjà existantes suivant un critère bien défini.

5.4.5 Le rôle de Zookeeper

De base, Apache Zookeeper est un outil conçu pour la gestion de configuration pour des systèmes fonctionnant en architecture distribuée. Le cluster Zookeeper est un cluster totalement distinct du cluster Hbase. Cependant, il joue un rôle central dans le fonctionnement de l'architecture Hbase. Zookeeper est responsable de la coordination des ressources du cluster Hbase et le maintien en un état de disponibilité et de fonctionnement correct.

ZooKeeper sert de registre de métadonnées pour les RegionServers. Il gère les informations de disponibilité des RegionServers actifs et envoie une notification au HMaster lorsqu'il y a défaillance d'un server. Par exemple, chaque fois qu'un RegionServer tombe en panne, ZooKeeper informe automatiquement le HMaster. Dès lors, HMaster contacte d'autres RegionServers et leur distribue les Regions et les WALs qui étaient localisées dans le RegionServer défaillant. La récupération des WALs permet ensuite de retrouver les données du MemStore du RegionServer défaillant. Chaque RegionServer sollicité réexécute le WAL afin de reconstruire le MemStore de la région défaillante.

Et en cas de défaillance du HMaster lui-même, le nœud qui assure la relève (pour garantir la continuité du service) est élu par consensus au sein du cluster Zookeeper. En effet, grâce à Zookeeper, il n'y a pas de « *single point of failure* » dans l'architecture Hbase. Il est possible de démarrer plusieurs HMasters en parallèle. Un HMaster fonctionnera en mode actif tandis que les autres seront en mode passif. Le HMaster actif envoie des heartbeats réguliers à Zookeeper. Les HMasters inactifs écoutent les notifications et se tiennent prêts en cas de défaillance du HMaster actif. Et en cas de défaillance du HMaster actif, le mécanisme interne d'élection par consensus se met en place dans Zookeeper. Un nouveau HMaster sera alors activé et prend le relais. Ce mécanisme de HMaster actif/passif permet de garantir la disponibilité du cluster Hbase.

5.4.6 La table « hbase :meta »

hbase :meta est une table de catalogue contenant la liste des RegionServers, les Regions ainsi que les tranches de Rowkey de chaque région. Cette table permet à ZooKeeper de retrouver dans un RegionServer, la Region qui contient une Rowkey donnée. Par exemple, lorsqu'un client envoie une requête sur une Rowkey donnée, ZooKeeper interroge la table "hbase:meta" pour récupérer les informations de la Region contenant la Rowkey. Il renvoie ensuite cette information au client, qui va alors directement s'adresser au RegionServer contenant la Region. Finalement, la RegionServer traite la requête et renvoyer au client les données de la Rowkey.

6 KAFKA

6.1 Fonctionnalités

Apache Kafka est une technologie de gestion de flux de données en temps réel. A l'origine développé par LinkedIn et utilisé comme outil de gestion de queues de messages, Kafka s'est très vite imposé comme une alternative aux systèmes classiques de messagerie d'entreprise.

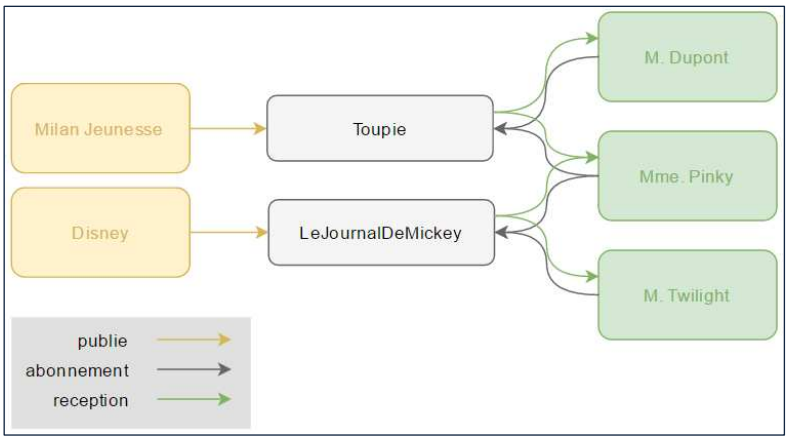
D'abord, en tant que système de messagerie, Kafka permet la publication des messages dans une sorte de file d'attente. Une file d'attente de messages Kafka permet de répondre à la problématique liée au fait qu'il est impossible de stocker en mémoire-tampon des données ou des messages, dans le cas où le destinataire n'est pas disponible (par exemple en cas de problèmes avec le réseau). Une file d'attente de messages Kafka permet aussi à l'expéditeur de ne pas surcharger d'un coup le destinataire. Cela peut être le cas, par exemple, lorsque, dans une connexion directe, les informations sont envoyées à un rythme plus rapide qu'elles ne sont réceptionnées et lues. Kafka peut aussi servir d'espace de stockage de données à court et moyen terme.

La technologie Kafka fonctionne sur le principe de streaming évènementiel. Le streaming événementiel est un processus qui consiste à traiter des flux d'événements au fur et à mesure qu'ils sont créés. Le streaming d'événements concerne par exemples l'analyse en continu des fichiers journaux générés par des applications Web, les logs de navigation des utilisateurs sur des sites Web d'e-commerce, les logs des interactions sur les réseaux sociaux, ou encore la collecte et la gestion des données de télémétrie provenant des appareils IoT (Internet des objets). Dans la plupart des dispositifs de gestion de données, le traitement des données est réalisé par lots périodiques (en mode batch). C'est-à-dire que les données brutes sont d'abord stockées, puis traitées à des intervalles de temps définis. Par exemple, une grande chaîne hôtelière pourrait attendre la fin de la journée, de la semaine ou du mois pour analyser l'ensemble des évènements : appels, réservations, annulations, paiements, etc. Or, les entreprises veulent de plus en plus analyser les données en temps réel afin de pouvoir prendre des décisions commerciales adaptées. Grâce à sa fonctionnalité de streaming, Kafka permet aux entreprises de tirer profit de la valeur temporelle des données.

6.2 Principe de fonctionnement

Apache Kafka est un système d'intermédiation entre des applications productrices de données et des applications consommatrices de données. C'est d'ailleurs pourquoi il est qualifié de brokers de messages. Kafka est un MOM (*Message Oriented Middleware*) qui se distingue des MOM standards par son architecture. C'est un système de messagerie distribué fonctionnant sur le principe de Publish-Subscribe

Pour comprendre le principe de *Publish-Subscribe* (encore appelé *Pub-Sub*), partons d'un exemple concret faisant l'analogie avec le service d'un postier[3].



Milan Jeunesse et Disney sont deux éditeurs de magazines pour jeunes enfants. Milan Jeunesse édite le magazine « Toupie » et Disney édite le magazine « Le Journal de Mickey ». En étant éditeurs Milan Jeunesse et Disney font le *Publish*

M. Dupont, M. Twilight et Mme. Pinky sont abonnés à ces magasines pour leurs enfants. M. Twilight et Mme. Pinky sont abonnés au Journal de Mickey et peuvent régulièrement trouver le journal dans leur boîte aux lettres ; M. Dupont et Mme. Pinky abonnés à Toupie peuvent également trouver leur magazine dans leur boîte aux lettres. Les trois parents représente l'aspect *Subscribe*.

Néanmoins, ce schéma peut évoluer. Voici, par exemples quelques scénarios :

- Si Disney passe un accord de partenariat avec Milan Jeunesse, cette dernière pourra aussi éditer le Journal de Mickey. Il y aura alors deux éditeurs pour le même magazine ;
- Lorsque la fille de M. Dupont aura grandi, ce dernier pourra se désabonner de Toupie et s'abonner au Journal de Mickey ;
- Un nouvel éditeur, PlayBack Press, peut aussi se lancer dans l'activité d'édition et proposer un nouveau journal, par exemple : « Mon Quotidien ». D'autres personnes pourront s'y abonner.

Ce système *Publish-Subscribe* est un cadre modulable dans lequel chaque acteur a la possibilité d'accéder à l'information qui l'intéresse sans contrainte particulière. Il est même possible d'imaginer un autre schéma dans lequel le rôle de *Publish* est assuré par les parents alors que le rôle de *Subscribe* est assuré par les éditeurs de magazine.

[3] L'exemple est tiré de l'article de Valentin Michalak publié en 2020 disponible à https://soat.developpez.com/tutoriels/bigdata/apprendre-kafka-concepts-fonctionnements/

Transposons ce cadre schématique au cas d'une architecture logicielle. Le modèle Publish-Subscribe est un modèle de système de messagerie qui met en relation un ensemble Publishers et un ensemble de Subscribers. Les Publishers ont pour rôle d'envoyer des messages (numéro du magazine) dans des topics (magazines). Les subscribers (les abonnés) peuvent donc régulièrement regarder le contenu des topics auxquels ils sont abonnés et vérifier si de nouveaux numéros de magazines sont arrivés. La mise en place d'un tel système permet à deux abonnés de profiter des mêmes informations, ou permet à deux Publishers de remplir le même topic de manière transparente.

La compréhension du modèle Publish-Subscribe est nécessaire pour comprendre l'architecture et le fonctionnement de Kafka. En effet, dans le langage Kafka, on dénomme par *Producer* : tout système qui envoie des données vers un ou plusieurs topics Kafka. Le Producer correspond au Publisher dans le système *Pub-Sub*. Et on dénomme par *Consumer* : tout système qui lit des données à partir d'un ou de plusieurs topics Kafka (Subscriber dans pub-sub.).

Le système de messagerie Publish-Subscribe sur lequel fonctionne Kafka offre plusieurs avantages par rapport au système de messagerie traditionnel. En effet, dans un système de file d'attente classique, les consommateurs d'un même groupe lisent les messages à partir d'un même serveur. Chaque enregistrement de la file est redirigé vers un consommateur donné. Les autres consommateurs n'ont pas la possibilité de lire le message déjà consommé par un autre. A l'inverse, dans le *Publish-Subscribe* l'enregistrement dans la file est mis à la disposition de tous les consommateurs. Chaque consommateur a donc la possibilité de lire tous les messages présents dans la file. Bien entendu, chacun de ces deux systèmes de messagerie a sa force et sa faiblesse. La force de la file d'attente classique est qu'elle permet de répartir la consommation des données entre plusieurs instances de consommateurs. Ce qui permet d'échelonner leur traitement. Mais l'inconvénient d'une file d'attente classique c'est qu'une fois qu'un consommateur lit un message, ce message n'est plus disponible pour les autres consommateurs. De plus, dans une file d'attente, les enregistrements sont distribués entre les consommateurs de façon asynchrone. C'est-à-dire que les messages arrivent en ordre dispersé sur les consommateurs et sont consommés de manière parallèle. L'ordre des enregistrements consommés est perdu en sortie. C'est d'ailleurs pourquoi, les systèmes de messagerie traditionnels mettent souvent en place un « consommateur exclusif » qui permet à un seul processus de consommer la file. Mais un tel mode de consommation ne permet pas de paralléliser le traitement. Le Publish-Subscribe, lui, permet de diffuser les messages vers plusieurs processus.

6.3 L'écosystème Kafka

La technologie Kafka est en évolution constante. L'outil se présente aujourd'hui comme un véritable écosystème. Les principales composantes de l'écosystème sont : Kafka Core, Kafka Streams et Kafka Connect.

- *Kafka Core* est la librairie de base constituant le noyau de l'architecture Kafka. Il est construit sur la base des notions de brokers, topics, Producers, Consumers, messages, Logs et réplications. Nous présentons les détails sur chacun de ces éléments plus bas.
- *Kafka Streams* est une librairie permettant de développer des outils capables de consommer des flux continus de données à partir d'un ou plusieurs topics d'entrée, les traiter, les transformer et produire des flux continus de données vers un ou plusieurs topics de sortie. Le dépilement des messages, leur consommation et la gestion de la concurrence sont automatiquement assurées par la librairie. Le client s'occupe uniquement du traitement et de la transformation des messages consommés. Kafka Streams facilite le développement d'applications de traitement de flux de données, similaire à d'autres solutions telles que Spark ou Flink. Elle accélère la création d'architectures micro-services se basant sur Kafka.
- *Kafka Connect* est une bibliothèque d'intégration de données permettant d'implémenter des connecteurs pour ingérer des données dans Kafka à partir des applications sources externes ou de pousser des données vers des applications externes à partir de Kafka.

La distribution Kafka proposée par la société Confluent inclut plusieurs couches dont notamment :

- *KSQL* : un moteur de requêtage SQL-like sur les données sur Kafka.
- *Schema Registry* : la couche de gestion des schémas des messages
- *Kafka REST Proxy* : permet de créer des clients Producer et Consumer en mode REST (http).
- *Replicator* : une solution de réplication fiable des données
- *Automatic Rebalancer*: une solution de répartition et de rééquilibrage de charge sur un cluster

6.4 Architecture de Kafka

L'architecture de Kafka est définie autour d'un certain de nombre de notions telles que : cluster, brokers, topic, partition, Producer, Consumer, message. Les sections suivantes donnent des précisions sur chaque notion.

6.4.1 Le cluster Kafka

Un cluster Kafka est une architecture distribuée, tolérante aux pannes et de faible latence. Il est construit autour de plusieurs nœuds (appelés brokers) dont le nombre peut être étendu horizontalement grâce à l'ajout de nœuds supplémentaires.

6.4.2 Les brokers

Les brokers sont des composants logiciels exécutés sur chaque nœud du cluster Kafka. Les données sont distribuées entre plusieurs brokers.

Dans un cluster de brokers Kafka, il existe toujours un broker « contrôleur » qui assure la gestion des partitions, le suivi des replicas et le monitoring de l'état des autres brokers. Le contrôleur assure également le rééquilibrage des partitions et l'attribution de nouveaux leaders de partitions.

Le broker contrôleur est élu par consensus parmi les autres brokers du cluster. Ce mécanisme d'élection prend place au sein du cluster Zookeeper géré entièrement en dehors du cluster Kafka. Voir plus bas les détails sur le rôle de Zookeeper dans l'architecture Kafka.

6.4.3 Les Producers

Les Producers sont des instances applicatives permettant de générer des données et le déposer dans une file Kafka (Topic). Il existe différents types de producteurs Kafka : les serveurs web, les composants d'applications, les appareils IoT, etc. Par exemple, un thermomètre connecté produira chaque heure des « événements » contenant des informations sur la température, l'humidité ou la vitesse du vent.

6.4.4 Les Consumers

Les Consumers sont des instances applicatives qui lisent les données écrites dans une file Kafka (topic) à par les Producers. En guise d'exemples de Consumers, on peut citer les bases de données, les Data Lakes ou encore les applications analytiques. Une entité peut être à la fois Producer et Consumer, à l'instar des applications ou des composants d'applications.

6.4.5 Le Message

Un message Kafka est une unité d'enregistrement de donnée se présentant sous forme d'un objet constitué d'une clé (optionnelle), d'une valeur et d'un timestamp. Au plus bas niveau, voici ci-dessous comment se présente la structure d'un message Kafka.

Tableau 2 : Structure d'un message Kafka

Objet MessageAndOffset

Champ	Taille	Description
MessageSize	32 bits	Taille du Message.
Offset	64 bits	Index unique de chaque message.
Message	bytes	Voir détail Objet Message.

Objet Message

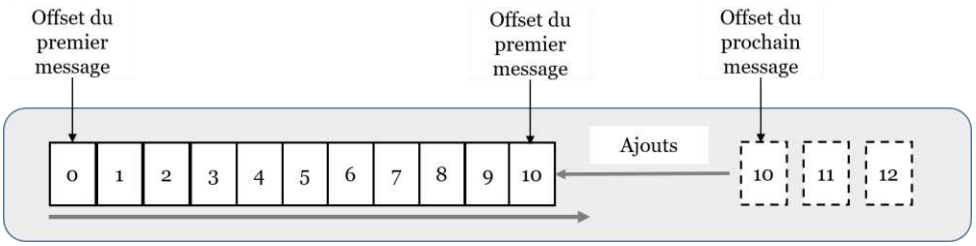
Champ	Taille	Description
CRC	32 bits	Contrôle de Redondance Cyclique : permet de vérifier l'intégrité d'un message au niveau du broker et du consumer.
MagicByte	8 bits	Identifiant de la version de Kafka utilisé pour la rétrocompatibilité.
Attributes	8 bits	Cet octet contient les métadonnées du message (codec de compression…).
Timestamp	64 bits	Timestamp du message.
KeySize	32 bits	Taille de la clé.
Key	bytes	La clé est optionnelle (peut être nulle) et est utilisée pour l'assignation à une partition.
ValueSize	32 bits	Taille du contenu.
Value	bytes	Contenu du message dans un byte-array « opaque ».

Chaque message dispose d'un CRC (Contrôle de Redondance Cyclique) permettant de vérifier son intégrité. Ce contrôle d'intégrité est totalement géré par Kafka. Cependant les attributs les plus couramment utilisés restent l'offset, le timestamps, la clé et la value. Chaque message est identifié par un Offset qui sert d'identifiant unique du message. L'offset est un identificateur séquentiel permettant d'identifier le message dans une partition donnée. Le timstamp représente l'instant auquel l'objet message a été déposé dans le topic par le Producer. La clé est une valeur optionnelle dont le rôle est de définir la partition de stockage. En effet, tous les objets messages ayant la même clé seront stockés dans la même partition (voir détails plus bas). Quant à la value, c'est le message proprement-dit. Elle contient l'information sur l'évènement publié dans le topic.

Notons que Kafka ne définit pas la façon dont sont formatées les valeurs de messages. Le choix du format des messages reste sous l'entière responsabilité de Producer. Il peut s'agir du JSON, du XML, du Avro, du Turtle, du Protobuff ou même du binaire, etc.

Par ailleurs, chaque message envoyé par un Producer et reçu par un Consumer est encapsulé dans un Log. Attention, le terme Log n'est pas employé ici au sens classique de log comme journal d'événements. Le Log ici fait référence à un modèle de données ressemblant à une file d'attente (ou queue). À la manière d'une file d'attente, le Log est un tableau dans lequel sont stockés les messages et ordonnés par rapport à leur ordre de réception. Les numéros d'identification correspondent aux Offsets.

Figure 13 : Disposition des messages dans une partition



Les messages sont persistés sur le cluster à mesure qu'ils sont publiés. Kafka utilise un délai de rétention configurable permettant de déterminer la durée de persistance d'un enregistrement donné, indépendamment de sa consommation. Tant que le délai de rétention maximal n'est pas dépassé, le message reste disponible pour la consommation. Mais dès que ce délai est dépassé, le message est automatiquement supprimé et l'espace-disque est libéré.

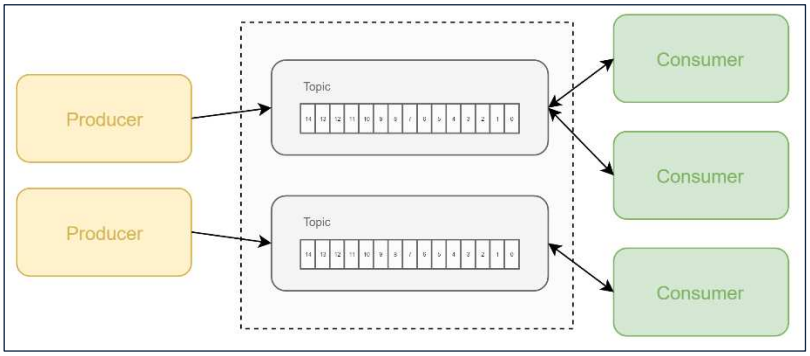
6.4.6 Les topics

Le topic est une séquence ordonnée et nommée de messages envoyés par les Producers et pouvant être lus par les Consumers.

Le topic est un concept universel dans le système de messagerie de type Publish-Subscribe. C'est une couche d'abstraction utilisée par l'application pour montrer son intérêt pour un flux de données fournie (série d'enregistrements/messages). Un topic peut être publié et faire l'objet d'un abonnement. Plus concrètement, un topic permet de regrouper les données relatives à un sujet particulier. En reprenant l'exemple précédemment présenté, les magazines « Le journal de Mickey » et « Toupie » représentent les topics. Disney et Milan Jeunesse représentent les Producers. M. Dupont, M. Twilight et Mme. Pinky (par l'intermédiaire de leurs enfants) sont les Consumers des messages déposés dans les topics (magazines).

Le schéma ci-dessous illustre le mode d'alimentation et de consommation d'un topic Kafka.

Figure 14 : Producers, Consumers, topics et messages



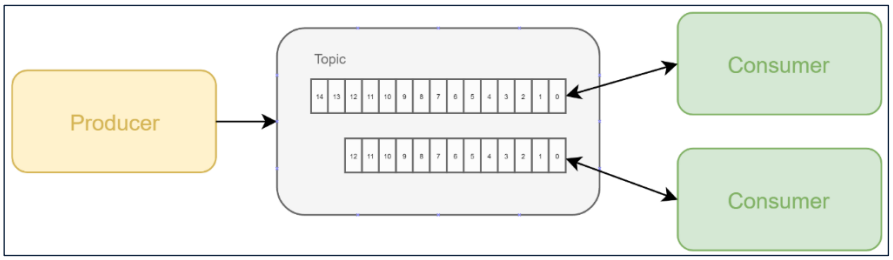
Notons que le topic n'est pas modifiable par le Producer à l'exception de l'ajout de messages à la fin de queue. Quant au Consumer, il ne peut que lire le contenu des topics. Notons aussi que Kafka ne maintient pas une liste des Consumers et de leur avancement dans la lecture des messages. Chaque consumer est donc responsable de son avancement dans la lecture des messages.

6.4.7 Les partitions

Les partitions sont des subdivisions des topics en une série de files d'attente persistées. Les partitions sont continuellement ajoutées pour former un Log des commits séquentiels. En général, chaque topic Kafka est divisé en plusieurs partitions, chaque partition pouvant être placée sur un broker séparé.

Les partitioning d'un topic Kafka vise plusieurs objectifs : d'abord distribuer les données entre les brokers pour mieux équilibrer la charge de stockage et d'assurer la réplication des données ; ensuite répartir la charge de lecture entre les Consumers appartenant au même groupe (appelé consumer Group).

Pour mieux détailler les propos, prenons l'exemple d'un topic découpé en deux partitions stockées sur le même broker.



Ici, le broker crée un Log pour chaque partition. Lorsqu'un Producer envoie un message dans le topic, ce message sera stocké dans l'une des deux partitions. L'assignation du message à l'une ou l'autre partition se fait sur la base des clés des messages. En principe, les messages ayant les mêmes clés sont stockés dans la même partition. Le choix de la partition peut être fait de manière aléatoire lorsque la clé de l'objet message est nulle (voir la structure de l'objet message plus haut).

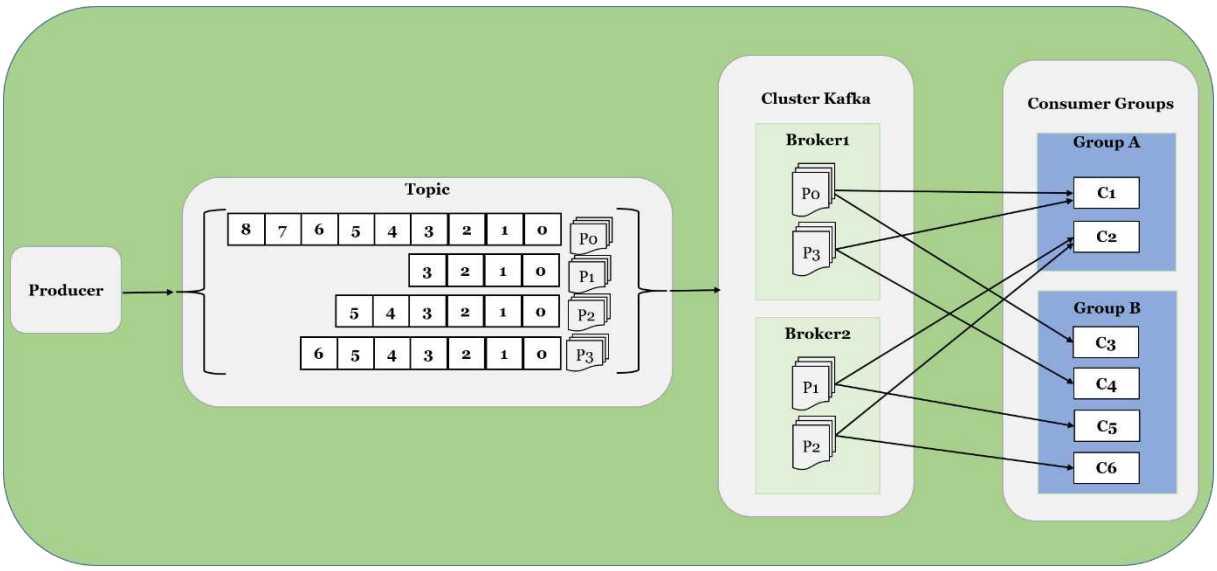
Dans le cas où le cluster Kafka est monté sur plusieurs brokers, le mécanisme de gestion des partitions et des réplications se déroule différemment. Par exemple, les partitions d'un topic ne sont plus stockées sur un même broker. Désormais, pour chaque partition, il existe un broker « leader » et des brokers de réplication. D'une manière générale chaque broker est non seulement responsable de la partition qui lui a été assignée et mais aussi sert de broker de réplication des partitions des autres brokers. Ainsi, lorsqu'un Producer dépose un message dans le topic, celui-ci est d'abord adressé au broker qui est leader de la partition. Ce broker enregistre le message dans son Log. Ensuite, selon le facteur de réplication défini, d'autres brokers répliquent le

contenu du Log principal dans leur propre Log. Dès lors, en cas de panne ou d'indisponibilité du broker leader de la partition, un autre broker prend le relais et devient le broker leader pour la partition. A noter qu'un même broker peut être leader d'une ou de plusieurs partitions.

Si les Producers écrivent dans les partitions en fonction de la valeur des clés des messages, le Consumers eux ne lisent que les partitions qui leur ont été explicitement assignées à travers un Consumer Group. En effet, dans un consumer Group, chaque partition est assignée à un consumer spécifique. Les messages déposés dans cette partition ne seront donc lus que par ce Consumer. En principe, pour répartir la charge de lecture entre les Consumers dans un Consumer group, il doit y avoir autant de Consumers que de partitions. Lorsqu'il y a plus de Consumers que de partitions, certains Consumers resteront inactifs car n'ayant pas de partitions assignées. Par contre, lorsqu'il y a moins de Consumers que de partitions dans un Group, certains Consumers seront plus chargés que d'autres car, ils devront lire le contenu d'au moins deux partitions. Notons par ailleurs que lorsqu'il n'y a qu'un seul consumer dans un Group, naturellement ce consumer aura la charge de lire le contenu de la totalité des partitions du topic.

Le schéma ci-dessous illustre la notion de Consumer Group et la répartition des charges de lecture entre les Consumers selon le nombre de partitions et le nombre de Consumers dans le groupe.

Figure 15 : Topic et consumer Groups



Dans ce schéma, nous avons un topic réparti en quatre partitions P0, P1, P2 et P3. Les partitions sont distribuées sur deux brokers Server1 et Server2.

Les messages de ce topic sont lus par deux Consumer Groups. Le Consumer Group A formé par deux Consumers C1 et C2 et le Consumer Group B formé par 4 Consumers C3, C4, C5 et C6.

Comme on peut le constater, dans le Consumer Group B, il y a autant de Consumers que de partitions. Alors, chaque Consumer sera chargé de lire les messages d'une seule partition. En revanche, dans le Consumer Group A, chaque Consumer sera chargé de lire deux partitions chacun pour assurer le load-balancing.

Pour ce qui concerne le mode de lecture des messages disponibles dans une partition, il existe deux possibilités pour un Consumer. Si le Consumer dispose d'un offset et réclame la lecture de la partition à partir de cet offset, alors il récupèrera tous les messages qui sont arrivés dans le Log après le message correspondant à l'offset ; Par contre, si le Consumer ne fournit pas d'offset à lecture, il récupère alors l'intégralité des messages disponibles dans la partition.

6.4.8 Le rôle de Zookeeper

Le cluster Zookeeper joue un rôle central dans le fonctionnement du cluster Kafka. En effet, Kafka utilise Zookeeper pour stocker les métadonnées sur les brokers, les topics, les partitions, les offsets, les réplicas, etc. Ces métadonnées permettent ainsi d'assurer la disponibilité du cluster Kafka même en cas de pannes d'un broker. Par exemple, en cas de défaillance d'un broker hébergeant les données d'une partition principale, le mécanisme d'élection interne de Zookeeper permet de choisir un autre broker hébergeant les réplications des partitions du server défaillant.

La communication entre Zookeeper et les brokers Kafka se fait à travers des « Zookeeper watch ». Les Zookeeper watch sont des évènements à travers lesquels le broker contrôleur est notifié par Zookeeper de tous les changements survenus dans les données. Une fois l'information reçue, le contrôleur se charge alors d'agir en conséquence en donnant des instructions aux brokers concernés : création ou mise à jour de partitions, lecture de partitions, etc…

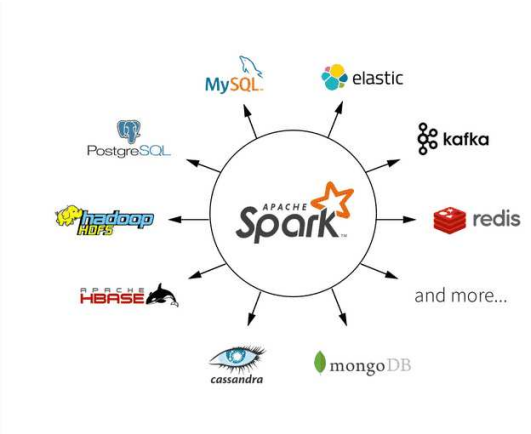
Le fonctionnement de Kafka reste aujourd'hui très fortement dépendant de Zookeeper. Toutefois, il n'est pas exclu qu'à l'avenir cette dépendance soit progressivement réduite. C'est la volonté affichée par de nombreux promoteurs de Kafka.

7 SPARK

7.1 Fonctionnalités

Spark est un framework open-source de *cluster computing* conçu pour le traitement *in-memory* de larges volumes de données. Initialement, la technologie Spark visait à répondre aux nombreuses limitations de HADOOP MAPREDUCE. En effet, l'une des causes de la lenteur des traitements de type MAPREDUCE est qu'à chaque étape d'un traitement, les résultats intermédiaires sont d'abord écrits sur disque pour ensuite être rechargés pour continuer à l'étape suivante. Ces écritures multiples augmentent considérablement le temps de traitement et entrainent une baisse de la performance. En introduisant deux concepts majeurs à savoir le RDD (*Resilient Distributed Dataset*) et le DAG (*Direct Acyclic Graph*)[4], Spark a quasi-révolutionné l'approche de traitement des données stockées sur HADOOP HDFS. D'abord, grâce au RDD (ensemble de blocs de données distribuées en mémoire sur le cluster), Spark évite autant que possible l'écriture sur disque. Car le RDD fonctionne sur le principe de traitement *in-memory,* un principe grâce auquel Spark conserve sur la mémoire les résultats intermédiaires de calcul. Le traitement *in-memory* est le premier facteur de performance de Spark par rapport à MAPREDUCE. Ensuite, grâce au concept de DAG, Spark exprime la séquence des opérations de traitement sous forme d'un graphe orienté permettant ainsi d'optimiser l'exécution des tâches.

Développé et maintenu, au départ, par les ingénieurs de AMPLab à l'Université de Californie à Berkeley en 2009, Spark est aujourd'hui un projet phare de la fondation Apache Software. Son périmètre d'utilisation ne se limite plus seulement à HADOOP. Grâce à ses nombreux connecteurs et fonctionnalités, Spark s'intègre aujourd'hui avec la plupart des bases de données et solutions de stockage du Big Data présentes sur le marché.



Source : https://databricks.com/fr/spark/about

[4] Nous allons étudier en détail ces deux concepts plus tard.

Il faut gardez à l'esprit que Spark est un moteur de traitement et non une base de données, ni un espace de stockage persistant. Il n'a pas de système de gestion de fichiers propre à lui. En fonction du cas d'usage, il utilise des systèmes de stockage externes comme HADOOP HDFS, Kafka, AWS S3, Cassandra, MongoDB, Elastic Search, etc.

7.2 Les composants de l'écosystème Spark

A l'heure actuelle, le framework Spark est un véritable écosystème construit autour de plusieurs modules qui permettent de répondre à de nombreux cas d'usage de traitement de données. Ci-dessous les principaux composants de l'écosystème Spark :

7.2.1 Spark Core

C'est le socle du framework Spark. Il implémente le RDD ainsi que toutes les fonctionnalités de parallélisation et de distribution de calculs sur le cluster Spark. Spark Core est également responsable de la gestion de la mémoire, des pannes, de la planification et de la surveillance des jobs sur le cluster. Il assure, en outre, les interactions avec les systèmes de stockage externes (ex : HDFS, AWS S3, etc.).

Le framework Spark est développé en langage Scala, qui reste le langage natif et recommandé pour développer des programmes de traitement. Cependant, le framework offre de nombreux API permettant aux développeurs d'implémenter leurs programmes dans d'autres langages comme Java, Python, SQL, R, JavaScript, Clojure.

7.2.2 Spark SQL

Spark SQL est le module Spark permettant d'utiliser un langage SQL pour le traitement des données structurées et semi-structurées. Spark SQL s'intègre aussi bien avec Apache Hive (en utilisant le langage HQL) qu'avec des bases de données traditionnelles via des connexions JDBC.

Le module Spark SQL propose plusieurs fonctionnalités dont notamment :

- l'API DataFrame : qui est une couche d'abstraction du RDD permettant d'exécuter des requêtes SQL sur des données dont la structure et le schéma sont connus.
- l'API DataSources : qui permet d'exécuter des requêtes SQL sur des données stockées sous différents formats : CSV, JSON, Parquet, Avro, etc.
- le connecteur JDBC : qui est un outil permettant de se connecter à des bases de données relationnelles et de réaliser des analyses Big Data à partir des outils de BI traditionnels.

7.2.3 Spark Streaming

C'est un module Spark conçu pour le traitement en temps réel des flux continus de données provenant de diverses sources externes : Kafka, Flume, Kinesis, etc. Basé sur le Spark Core, le module Spark Streaming traite les données sous forme de micro-batches espacés d'un instant t (paramétrable). Il utilise des objets DStreams (discretized streams) qui sont, en fait, une série de RDDs groupés. Le mode de traitement des Dstreams par le module Spark Streaming est quasi-identique au mode de traitement des RDDs standards par le Spark Core.

7.2.4 MLlib

MLlib est un module de Machine-Learning conçu dans la but de pouvoir appliquer des algorithmes d'apprentissage à grande échelle sur des données stockées sur un cluster. MLlib fournit la plupart des algorithmes d'apprentissage classiques : réduction de dimensions, classification, régression, etc.

Notons tout de même qu'en dehors du module MLlib, il existe aussi une librairie de Machine Learning disponible sous Spark Core, Spark ML, qui elle est basée sur l'API DataFrame.

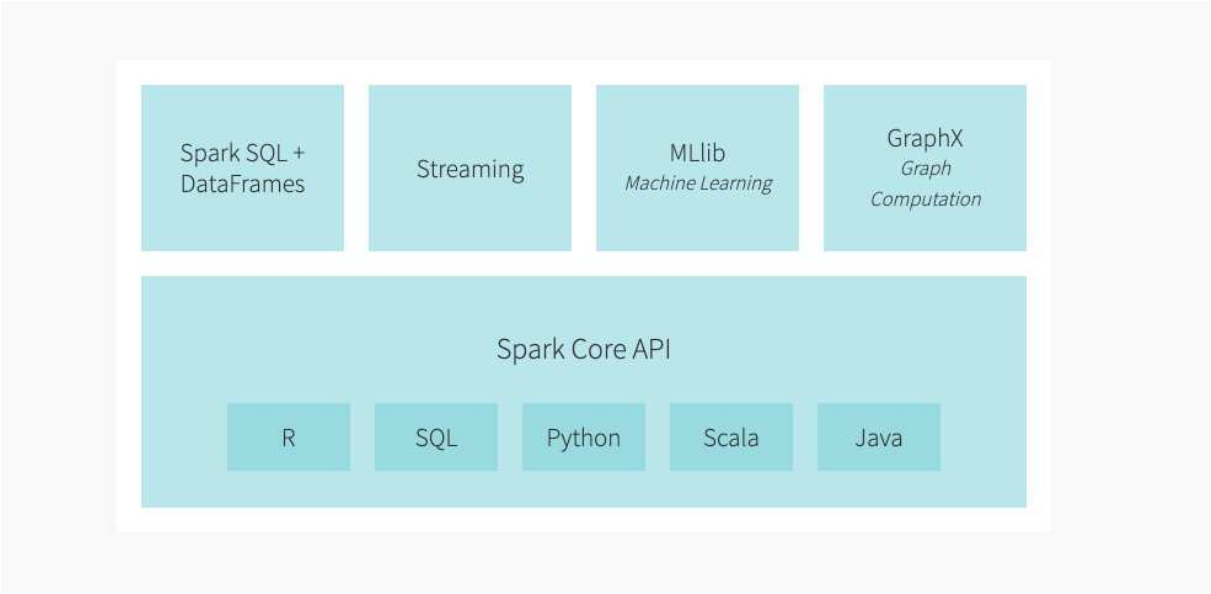
7.2.5 GraphX

GraphX est un framework permettant le traitement des données dont les relations sont modélisées sous forme de graphes distribués. GraphX est généralement utilisé pour l'analyse des données de types : réseaux sociaux, des pages web (PageRank), etc.

GraphX étend les fonctionnalités du RDD de Spark Core en introduisant le concept de *Resilient Distributed Graph* (RDG). Le RDG est un modèle de graphe orienté dans lequel les lignes de données (représentant les sommets du graphe) sont reliées entre elles par des arêtes qui traduisent les influences réciproques.

L'un des avantages de GraphX par rapport aux systèmes traditionnels de traitement de graphes est sa capacité à considérer les structures de données à la fois comme un graphe (en appliquant ainsi les méthodes de traitement appropriées) et mais également comme des collections séparées de sommets et d'arêtes. En effet, un objet GraphX renvoie deux RDDs, un pour les nœuds (sommets) et un pour les arcs (arêtes). Avec cette distinction, il devient possible d'appliquer des opérations de traitements classiques sur les RDDs : map, join, etc.

Figure 16 : Ecosystème Spark



Source : https://databricks.com/spark/about

7.3 RDD et DAG : les fondements du concept Spark

La performance de Spark repose sur deux concepts majeurs : le *Resilient Distributed Dataset* (RDD) et le *Directed Acyclic Graph* (DAG). Les sections ci-après présentent en détails chacun des deux concepts.

7.3.1 Le RDD

7.3.1.1 Définition et caractéristiques

Le *Resilient Distributed Dataset* (RDD) est un ensemble logique de blocs de données distribués et pouvant être conservés sur la mémoire RAM en vue d'accélérer leurs traitements. Le RDD est l'objet de base du traitement des données par Spark. Le terme RDD renvoie les significations suivantes :

- ***Resilient :*** il est possible de reconstituer les blocs de données (partitions) même en cas de panne lors de l'exécution du programme de traitement. C'est cette tolérance aux pannes qui justifie le qualificatif de Résiliant.
- ***Distributed :*** les blocs de données sont distribués sur différents nœuds du cluster Spark. Cela permet non seulement de paralléliser les opérations de traitement mais aussi faciliter la reconstitution des blocs en cas de panne.
- ***Dataset :*** le RDD est une collection d'enregistrements de données partitionnées.

Un RDD se construit généralement en lisant les données stockées sur des systèmes de fichiers externes : HDFS, Cassandra, Hbase, Kafka, etc. La construction du RDD se fait alors via l'API RDD de Spark, qui est disponible dans plusieurs langages : Scala, Java, Python, R, etc. Le RDD peut également être créé en appliquant une transformation sur un RDD existant ; le RDD original reste alors inchangé. En effet, le RDD est un objet immuable et n'est accessible qu'en lecture seule.

Un RDD présente plusieurs caractéristiques importantes.

- *Le RDD est un objet partitionné*

Le RDD est une collection de données réparties en plusieurs blocs appelées partitions. La partition est l'unité de base du parallélisme du traitement Spark. Les partitions sont construites lors de la création du RDD, soit par la lecture des données à partir d'une source de données externe, soit par application d'une transformation sur un RDD existant ou bien par une opération de repartitionnement d'un RDD existant.

- *Le RDD est tolérant aux pannes*

L'un des grands avantages du RDD se trouve dans sa capacité à conserver l'historique des informations sur la manière dont une partition a été créée. Grâce à cette historique, il devient facile de reconstituer, à tout moment, la partition, même en cas de perte accidentelle. En effet, une panne affectant une partition individuelle d'un RDD peut être réparée (par reconstitution de l'historique de création) indépendamment des autres partitions, évitant ainsi d'avoir à recalculer tout le RDD.

- *Le RDD permet une évaluation paresseuse des opérations de traitement (lazy-evaluation)*

Toutes les opérations de transformation sur un RDD sont dites « lazy » (paresseuses). Quand on applique une « *transformation* » sur un RDD, cette transformation ne sera effectivement exécutée que lorsqu'une « *action* » est invoquée. Comme nous allons le voir plus tard, deux types d'opérations sont applicables sur le RDD : les *transformations* et les *actions*. Une transformation est une séquence d'opérations de traitement appliquée à un RDD et qui aboutit à la création d'un nouveau RDD. Les actions, quant à elles, sont des opérations qui visent à écrire (ou à afficher) les résultats d'une opération de traitement appliquée à un RDD. Les transformations sont de nature *lazy*, car elles ne s'exécutent que lors de l'appel d'une *action* qui est généralement définie en aval. Ainsi, l'appel d'une *action* déclenche toutes les *transformations* qui ont été spécifiées en amont.

- *Le RDD permet le traitement des données en mémoire*

Grâce au RDD, Spark peut stocker les résultats intermédiaires de traitement sur la mémoire (RAM). Cette fonctionnalité permet, comme nous l'avons déjà évoqué, de gagner en performance.

- *Mise en cache et persistance*

Spark offre plus de flexibilité en permettant de persister le RDD aussi bien sur la mémoire que sur le disque. Persister un RDD consiste à le mettre en cache sur la mémoire (si la taille le permet) ou alors le stocker sur disque.

La persistance est une propriété qui permet d'éviter d'appliquer plusieurs fois les mêmes transformations. Comme indiqué précédemment, l'appel d'une *action* entraîne le déclenchement de toutes les *transformations* spécifiées en amont. Si, dans un job Spark, plusieurs *actions* font appel à un même RDD, il devient alors pertinent de persister ce RDD afin d'améliorer la performance du traitement.

Le RDD est immuable

Une caractéristique fondamentale du RDD réside dans son immuabilité. En effet, un RDD n'est ni modifiable, ni altérable. Toute transformation appliquée sur un RDD aboutit nécessairement à la création d'un nouveau RDD, laissant ainsi inchangé le RDD d'origine. La propriété d'immuabilité permet de garantir l'intégrité des données.

7.3.1.2 Opérations sur un RDD : transformations et actions

Les transformations

Au sens Spark, une transformation est une opération de traitement appliquée sur un RDD dans le but de l'enrichir ou de le restructurer. Une transformation est généralement une fonction qui prend en entrée un RDD et qui retourne en sortie un nouveau RDD. A noter que le RDD pris en entrée reste toujours inchangé, à cause de la propriété d'immuabilité.

Parmi les fonctions de transformation, on peut citer par exemples : map, filter, flatMap, groupByKey, reduceByKey, join, union, sort….

On distingue deux types de transformations : les transformations étroites (*Narrow transformations*) et les transformations étendues (*Wide transformations*).

Les *Narrow transformations* sont des transformations qui sont entièrement exécutées en local sur les nœuds de traitement Spark sans qu'il y ait besoin de transférer les données entre les nœuds. A chaque partition du RDD d'entrée correspond zéro ou une partition du RDD de sortie. Exemples de narrow transformations : filter, map, etc.

Quant aux *Wide transformations*, elles se caractérisent par le fait qu'il est nécessaire de transférer les données entre les nœuds de traitement afin d'obtenir le RDD de sortie. Ce transfert de données entre les nœuds est appelé *shuffle*. Le *shuffle* est une opération de transfert de données à travers le réseau en vue de constituer les partitions du RDD de sortie. Un exemple de *wide transformation* est la fonction join.

Les actions

Une action est une séquence d'opérations de traitement appliquée sur un RDD mais dont le résultat n'est pas un RDD. En effet, contrairement à une transformation qui renvoie toujours un RDD, une action renvoie un résultat agrégé qui peut être soit un objet, soit une valeur ou un ensemble de valeurs, un fichier de données, etc… Parmi les actions, on peut citer par exemples les fonctions : collecte, count, countByKey, saveAsTextFile, etc…

Dans un programme de traitement Spark, l'appel d'une action sur un RDD déclenche toutes les transformations qui ont été préalablement définies. De ce point de vue, on peut dire que ce sont les actions qui « *réveillent* » les transformations.

7.3.1.3 Evaluation paresseuse des transformations et persistance du RDD

Comme nous l'avons déjà indiqué, les opérations de transformation sont dites « *paresseuses* ». C'est à dire que la transformation n'est effectivement exécutée que lorsqu'une action est appelée. En effet, on peut considérer qu'une transformation, lorsqu'elle est définie, reste en « *sommeil* » tant qu'une action n'est pas appelée. Les transformations se comportent alors comme des pointeurs qui ne seront compilés et exécutés que lors de l'appel d'une action. L'avantage de l'évaluation paresseuse c'est qu'elle permet d'optimiser les coûts de traitement : faible utilisation des ressources, etc. Ce qui constitue un critère de performance d'un traitement.

Cependant, même si l'évaluation paresseuse est une fonctionnalité importante de Spark, elle montre ses limites dans certaines situations. Par exemple, dans un programme de traitement, il arrive parfois qu'un même RDD soit appelé à plusieurs endroits du programme. Les actions qui sont déclenchées à la suite de chaque appel du RDD déclenchent systématiquement toutes les transformations définies depuis le début du programme. En clair, les mêmes transformations sont ré-exécutée à chaque fois qu'une action est appliquée sur le même RDD. Cette situation peut générer une contre-performance lors de l'exécution du programme. Mais heureusement, grâce à la fonctionnalité de persistance du RDD, il est possible d'éviter de ré-exécuter chaque fois les mêmes transformations pour reconstituer le RDD. Pour cela, il suffit d'appeler la fonction cache() ou la fonction persist() sur le RDD juste avant le premier appel. Les transformations qui permettent de calculer le RDD ne seront alors exécutées qu'une seule fois. Car les fonctions cache()/persist() stockent le RDD sur la mémoire et/ou sur le disque en vue d'une utilisation future. Dès lors, tous les appels utilisent directement ce RDD, sans avoir à ré-exécuter les transformations qui ont permis de le calculer. La fonctionnalité de mise en cache (ou de persistance) permet de limiter les effets de bord de l'évaluation paresseuse des transformations.

Notons toutefois qu'il n'est pas toujours nécessaire de mettre en cache ou de persister un RDD, même si plusieurs actions sont appliquées dessus. En effet, la persistance (surtout lorsqu'elle se fait sur disque) exige parfois des opérations de sérialisation et de désérialisation qui prennent généralement beaucoup de temps. Ainsi, dès lors que

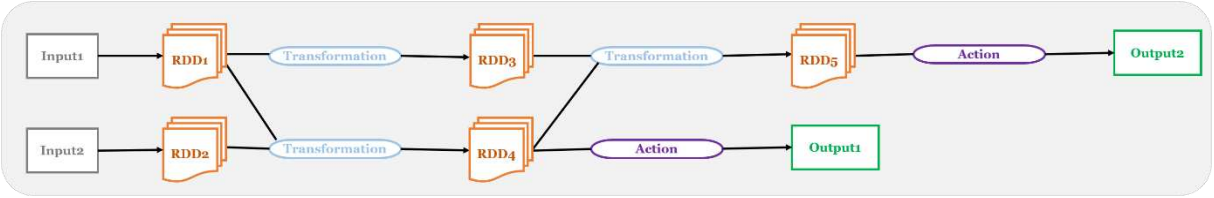
le temps nécessaire pour recalculer un RDD (en ré-exécutant les transformations) est inférieur au temps nécessaire pour sérialiser : désérialiser le RDD, la persistance n'est plus pertinente.

7.3.2 Le DAG

7.3.2.1 Le concept de DAG

Le **Directed Acyclic Graph** (DAG) est un formalisme graphique qui permet au moteur d'exécution de Spark de traduire l'ensemble des opérations de traitement en une séquence d'étapes en vue de générer un plan d'exécution optimisé. Le DAG est généré à partir du graph des transformations et actions sur les RDD (voir figure ci-dessous).

Figure 17 : DAG : Transformations et actions sur les RDD



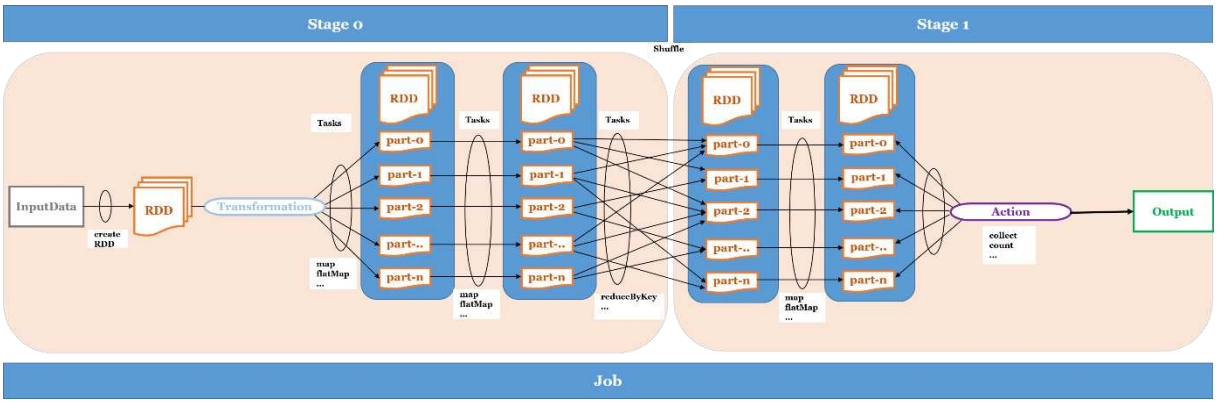
Le graph des transformations et actions est la combinaison d'un ensemble de sommets et d'arêtes. Les sommets représentent les RDD (issus des transformations) ou les résultats (outputs) des actions. Les arêtes sont les transformations ou les actions qui ont permis d'obtenir les RDD et les résultats. Le graphe est dit acyclique car les sommets sont reliés les uns aux autres dans un sens unique de telle sorte qu'il est impossible d'avoir des références circulaires.

Faisons juste remarquer que le DAG n'est pas un concept propre à Spark. Des outils comme Tez, Drill et Presto se basent également sur le DAG pour générer les plans d'exécution de leurs jobs.

7.3.2.2 Le DAG et le découpage du plan d'exécution en Job, Tasks et Stages

Le DAG permet de structurer le plan d'exécution en plusieurs niveaux à savoir : la Task (tâche), la Stage (étape) et le Job. Le graph ci-dessous illustre la structure d'un plan d'exécution d'un traitement Spark.

Figure 18 : DAG : Découpage du plan d'exécution du traitement



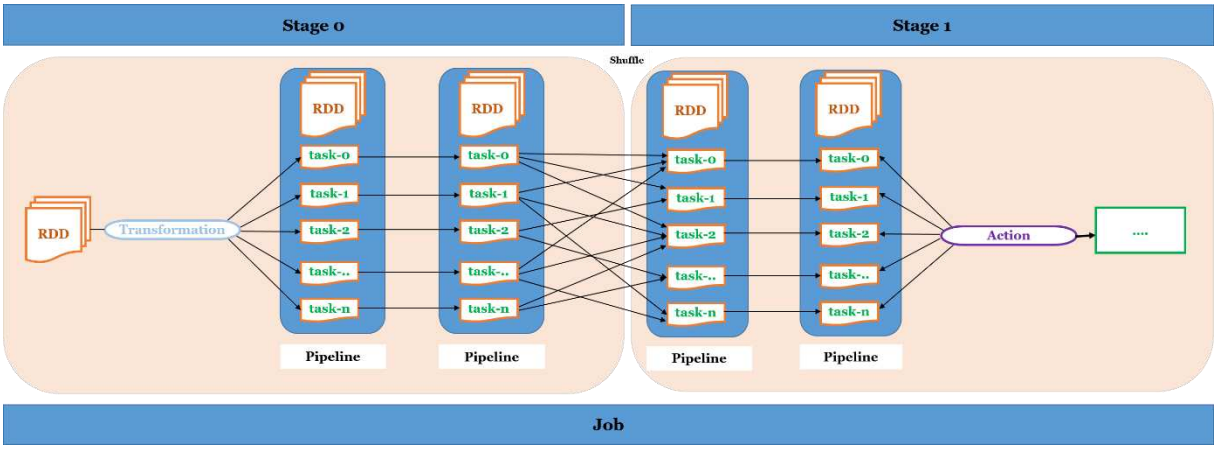
Task (Tâche)

Une Task est une unité de traitement exécutée sur chaque partition du RDD. La partition étant l'unité de parallélisme du traitement Spark, il y aura autant de Tasks que de partitions lors d'une opération de transformation d'un RDD.

Stage (Etape)

Une Stage est une succession de pipelines de Tasks. Un pipeline de Tasks est un ensemble de Tasks qui s'exécutent en parallèle sur les partitions d'un même RDD.

Figure 19 : DAG : pipelines de tasks



Une Stage regroupe l'ensemble des opérations de transformation et d'action qui ne nécessitent pas de Shuffle. Pour rappel, le Shuffle est une opération au cours de laquelle le RDD est repartitionné et redistribué sur les nœuds de traitement Spark. Il nécessite le transfert des données via le réseau. Le Shuffle a lieu entre deux Stages. Les résultats d'une première Stage sont Shufflés dans le but de calculer de nouvelles partitions qui serviront d'inputs à la Stage suivante. Une nouvelle étape est donc créée dans le plan d'exécution après chaque opération de Shuffle.

Job

Un job représente l'ensemble des Stages définies dans le plan d'exécution. C'est la matérialisation du plan d'exécution dans son ensemble.

7.3.2.3 Le DAG, lineage et mécanisme de reprise des tâches

Le DAG propose des mécanismes de reprise de tâches en cas de pannes. Par exemple, lors de l'exécution d'un pipeline, lorsqu'une tâche est perdue à cause d'une panne quelconque, celle-ci sera automatiquement relancée. La relance peut se faire sur le même nœud de traitement ou sur un autre nœud si le premier nœud est indisponible.

En cas de panne sévère, le DAG permet aussi de reconstituer tout un RDD en se basant sur le Lineage. Le lineage est la trace de toutes les dépendances existantes entre les RDD. En effet, lorsqu'un nouveau RDD est dérivé d'un RDD existant à l'aide d'une transformation, ce nouveau RDD contient un pointeur vers le RDD parent. Le lineage est la représentation de toutes les dépendances qui relie un RDD avec ses RDD parents et ses RDD enfants. Dès lors, en cas de perte d'un RDD suite à une panne, il suffit de réappliquer la chaîne traitement en remontant jusqu'aux RDD qui précèdent dans le graphe. Si l'un des RDD parents est persisté ou mis en cache, Spark relance les calculs à partir de ce RDD persisté et ainsi déroule les transformations successives pour reconstituer le RDD perdu.

7.4 Le DataFrame et le DataSet : surcouches optimisées du RDD

Les API DataFrame et DataSet sont des abstractions de haut niveau qui ajoutent des optimisations supplémentaires à l'API RDD afin de corriger les limitations de celle-ci. En effet, l'API RDD montre un certain nombre de limitations. Par exemple, le RDD n'offre aucune possibilité d'exploiter les métadonnées associées à la structure des données : schéma des données, nom et types des colonnes. Aussi, le RDD ne comporte aucun moteur d'optimisation intégré qui permettrait d'apporter des optimisations supplémentaires lors de la génération du plan d'exécution. Lorsque des optimisations sont nécessaires lors de l'exécution d'un traitement, il revient à l'utilisateur de les spécifier et de les implémenter au niveau des fonctions du programme du traitement. L'API RDD se charge simplement d'appliquer sur les données les fonctions de traitement spécifiées par l'utilisateur. Elle n'apporte aucune optimisation supplémentaire au plan d'exécution. Les API DataFrame et DataSet visent à corriger ces lacunes. Grâce au DataFrame et au DataSet, Spark peut non seulement exploiter les schémas associés aux données mais aussi ajouter des optimisations supplémentaires au plan d'exécution des traitements. Les API DataFrame et DataSet sont des surcouches optimisées du RDD qui permettent d'améliorer significativement la performance des programmes de traitement Spark. Les lignes ci-dessous fournissent les détails sur les deux API.

7.4.1 L'API DataFrame

Le DataFrame est une couche d'abstraction du RDD qui présente les données sous une forme tabulaire (un ensemble lignes dont le schéma est contenu) en vue de pouvoir y appliquer des techniques de traitement issues du langage SQL. Tout comme le RDD,

le DataFrame peut être construit de différentes manières : à partir de sources de données externes telles que les fichiers de données structurées, des tables Hive, des bases de données relationnelles externes ou à partir de RDDs existants.

L'API DataFrame est intégré à Spark depuis la version 1.3.0. Elle est accessible dans plusieurs langues : Scala, Java, Python et R.

L'exploitation des données sous forme de DataFrame présente plusieurs avantages par rapport à l'exploitation des données sous forme de RDD. D'une part, le DataFrame permet une gestion optimisée de la mémoire lors de l'exécution de la requête. D'autre part, le DataFrame permet d'ajouter des optimisations supplémentaires au plan d'exécution physique des tâches.

Gestion optimisée de la mémoire :

L'utilisation du DataFrame à la place du RDD permet de réduire considérablement l'utilisation de la mémoire lors de l'exécution des tâches. Contrairement au RDD, le DataFrame stocke les données dans la mémoire off-heap au format binaire. Ce qui permet d'éviter les opérations de sérialisation inhérentes au RDD, parfois très couteuses. En effet, dans le cas d'un RDD, l'application de certaines transformations ou actions amène Spark à redistribuer les données au sein du cluster ou à les écrire sur disque. Ces mouvements de données à travers le réseau nécessitent une sérialisation des objets. Mais grâce à l'encodage des données au format binaire en mémoire (off-heap), on peut directement effectuer les transformations d'autant que le schéma des données est connu à l'avance.

Par ailleurs, l'utilisation du DataFrame à la place du RDD permet d'éviter la garbage collection. Un RDD étant généralement une collection d'objets Java, sa manipulation nécessite l'instanciation d'un objet individuel pour chaque ligne de données auquel on applique les traitements. Ces objets individuels doivent, par la suite, être ramassés et détruits par le garbage collector afin de libérer la mémoire. Mais avec le DataFrame, ce genre d'opérations de nettoyage est évité. Le schéma des données étant connu, il n'y a pas besoin d'instancier des objets pour représenter les lignes de données.

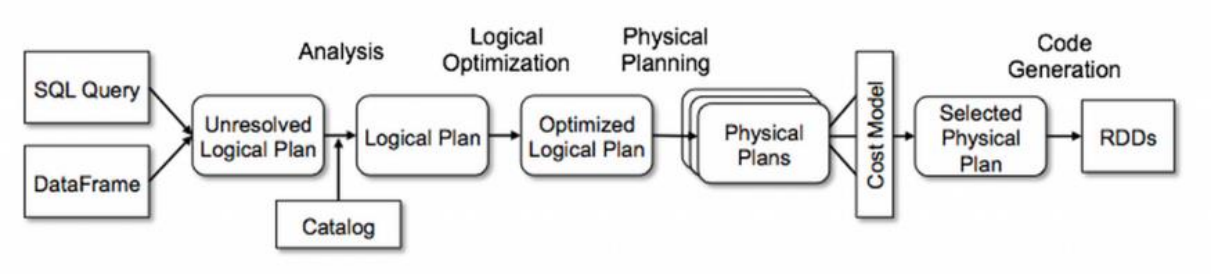
Remarquons en dernier que la gestion optimisée de la mémoire à travers un DataFrame est rendu possible grâce au projet Tungsten.

Plan d'exécution optimisé :

Lorsqu'on exécute une opération sur un RDD, cette opération est généralement spécifiée sous forme d'une fonction qui est directement appliquée aux données sans aucune optimisation supplémentaire. Mais avec un DataFrame, toute fonction de transformation et d'action est d'abord optimisée via ***Catalyst Optimizer*** qui est un framework d'optimisation de requête. Le *Catalyst Optimizer* permet de trouver le plan d'exécution le plus efficace pour appliquer les fonctions de traitement aux données.

La figure ci-dessous illustre les différentes étapes de l'exécution d'un traitement sur un DataFrame ainsi que le rôle du Catalyst Optimizer.

Figure 20 : Plan d'exécution optimisée



source : https://databricks.com/glossary/catalyst-optimizer

1. Lorsqu'une requête SQL est lancée sur les données sources ou sur un DataFrame existant, le moteur d'exécution Spark SQL crée d'abord un plan logique non résolu et vérifie la validité de la syntaxe et du schéma des données : noms des tables, nom de colonnes, etc.
2. Après la validation de la syntaxe et des métadonnées, le moteur génère un plan logique résolu. À cette étape, les commandes peuvent être réorganisées afin de les optimiser et ainsi générer un plan logique optimisé.
3. Dans un troisième temps, le *Catalyst Optimizer* génère un ou plusieurs plan physiques candidats capables d'exécuter la requête soumise.
4. Dans le cas où plusieurs plans physiques sont candidats, le coût de chaque plan est évalué à l'aide d'un modèle de coût. Ainsi, le plan qui présente le plus faible coût est sélectionné pour être exécuté.
5. Le plan choisi est d'abord compilé pour générer le bytecode Java et appliqué aux données. Notons cependant, qu'une fois que le plan optimisé est créé et compilé en bytecode Java, l'exécution finale des tâches se fait sur le RDD. De ce fait, le RDD reste la couche d'abstraction de base de Spark.

7.4.2 L'API DataSet

Introduit dans la version 1.6 de Spark, le DataSet est une extension du DataFrame qui associe un schéma relationnel fortement typé aux données et tire profit de la performance optimisée fournit par le Catalyst Otimizer. L'utilisation du DataSet permet de surmonter plusieurs limitations du DataFrame. Par exemple dans un DataFrame, les données ne peuvent être modifiées sans connaître à l'avance leur structure et leur schéma. Une telle contrainte n'existe pas pour le DataSet, car la vérification et l'analyse de la syntaxe de la requête sont effectuées au moment de la compilation en bytecode. En outre, après avoir transformé un RDD en DataFrame, on ne peut plus régénérer ce même RDD en partant du DataFrame. Par exemple, si on génère un DataFrame nommé myDataDF à partir d'un RDD nommé myDataRDD, on ne pourra pas récupérer le RDD d'origine myDataRDD. Mais avec un DataSet, il est possible de régénérer le RDD original à partir duquel le DataSet a été construit.

Le DataSet reste, tout de même, intimement lié au DataFrame. Dans le langage de programmation, un DataFrame n'est rien d'autre qu'un DataSet d'objets de type Row

(DataFrame = Dataset[Row]). La distinction du typage des structures entre DataSet et DataFrame est surtout lié au langage de programmation utilisé. Par exemple, les objets en Python et R étant dynamiquement typés, l'accès aux données à travers Spark SQL se fera à travers des DataFrames. En revanche Scala et Java qui sont des langages où les objets sont fortement typés, on utilise des DataSets.

Pour illustrer les particularités entre un RDD, un DataFrame et un DataSet, utilisons l'exemple suivant. Nous disposons d'une collection de données sur les albums de musique. Cette collection est représentée par un objet distribué sur le cluster et noté `musics`. Nous souhaitons appliquer un filtre sur cet objet pour ne retenir que les albums de musique dont le genre musical est « `Jazz` ». Supposons que ce filtre peut être spécifié comme suit :

```
musics.filter(music.getGenre() == 'Jazz');
```

Lorsque l'objet `musics` est un RDD d'objets `music`, Spark doit d'abord instancier un objet Java pour chaque `music`, ensuite appeler la méthode `getGenre()` pour chaque objet instancié. Et enfin il doit comparer la valeur renvoyée au critère `Jazz`. Cette action de test de valeurs peut s'avérer couteuse surtout lorsque la taille de la collection est importante. Mais en utilisant un DataFrame ou un DataSet, le schéma de l'objet `musics` pourra être défini et connu d'avance. Ainsi, Spark utilisera son propre système d'encodage/décodage à la place de la sérialisation Java nécessaire pour un RDD. De plus, dans le cas des DataSets, la valeur du champ `genre` peut être testée directement sans même effectuer de décodage depuis la représentation binaire. En résumé, il est préférable d'utiliser les DataSets dès que l'on a affaire à des données structurées avec des objets fortement typés.

Mais qu'il s'agisse du DataFrame ou du DataSet, la seule connaissance du schéma des données permet à Spark d'utiliser le langage SQL pour traiter les données. C'est le fondement du module Spark SQL. La connaissance du schéma permet surtout d'éviter la sérialisation/désérialisation des objets Java, opérations toujours effectuées dans le cas des RDD et qui sont parfois très couteuses.

Rappelons tout de même que les DataFrames et Datasets reposent toujours *in fine* sur les RDDs. Toute transformation ou action appliquée sur un DataFrame/DataSet est d'abord convertie en un plan d'exécution optimisé, ensuite compilée en bytecode et enfin exécutée sur les données stockées sous forme de RDD.

7.5 Les variables partagées : Broadcast et Accumulator

Les variables partagées en Spark sont des variables qui sont définies au niveau du Driver Spark et qui sont accessibles en même temps par tous les exécuteurs sans conflit de partage. On distingue deux types de variables partagées : les variables Broadcasts et les variables Accumulators. Les variables Broadcasts servent à déposer une copie d'un set de données sur tous les exécuteurs en co-location avec les partitions afin de pouvoir les utiliser directement dans les tâches. Les variables Accumulators servent à agréger

au sein du Driver en recueillant des informations à partir de tous les exécuteurs. Les lignes ci-dessous détaillent les caractéristiques des variables Broadcasts et Accumulators.

7.5.1 Les variables Broadcasts

Les variables Broadcasts sont des variables en *lecture seule* que Spark met en cache sur tous les exécuteurs afin d'éviter de les expédier à travers le réseau chaque fois qu'elles sont appelées dans le programme de traitement. Les variables Broadcasts sont souvent utilisées pour distribuer les données de lookup (données d'enrichissement). Car étant donné qu'une copie du set de données est disponible en local sur chaque exécuteur, la variable Broadcast ne génère aucune surcharge I/O lié au transfert de données via le réseau. Pour illustrer l'importance d'une variable Broadcast, prenons l'exemple d'un traitement Spark Streaming dans lequel chaque événement en entrée doit être enrichi avec un set de données de lookup. Dans une approche standard, pour traiter chaque bloc de DStream, le set de données d'enrichissement doit être envoyé sur les exécuteurs via un transfert un réseau pour pouvoir le combiner aux partitions disponibles sur les exécuteurs. Cette opération de transfert continu génère une importante surcharge I/O a niveau du réseau et peut se traduire ainsi par une baisse de performance du traitement. Pour éviter cette situation, il suffit de spécifier le set de données d'enrichissement en tant que variable Broadcast. Ce qui permet de déposer une copie sur tous les exécuteurs, évitant ainsi les transferts multiples au sein du réseau. La mise en Broadcast d'une variable permet une amélioration significative de la performance du traitement.

7.5.2 Les variables Accumulators

Les variables Accumulators sont des variables partagées qui servent à recueillir des informations à partir des exécuteurs et de les agréger au niveau du Driver. Prenons un exemple concret pour illustrer l'usage d'une variable Accumulator. Nous disposons d'un fichier de données textuelles de taille relativement importante chargé et distribué sur le cluster Spark sous forme de RDD. Nous souhaitons connaître le nombre de lignes contenant le caractère "$". Le fichier de données étant distribué et reparti sur tous les nœuds de traitement Spark, nous pouvons utiliser une variable Accumulator pour récupérer cette l'information.

D'un point de vue programme, la solution du problème peut être spécifiée comme suit :

```
val inputRdd = sc.textFile("myTextFile.txt")
val accum = sc.longAccumulator("Dollar sign Counter")
inputRdd.foreach { line =>
 if (line.contains("$"))
   accum.add(1)
}
println(accum.value);
```

La variable *inputRdd* représente le RDD construit à partir du fichier de données texte. La variable *accum* représente la variable Accumulator. Elle est d'abord initialisée dans le Driver avant d'être alimentée à partir des informations disponibles sur les exécuteurs. Ici, nous utiliserons une variable Accumulator de type Long. La fonction lambda *foreach* appelée sur inputRdd permet d'appliquer le critère de calcul sur chaque ligne de texte d'une partition du RDD. Chaque fois qu'une ligne texte contient le caractère $, la variable *accum* est incrémentée de 1. A l'issue du traitement, la valeur finale renvoyée par la variable *accum* correspond au nombre total de lignes du fichier contenant au moins le caractère $. Ceci permet d'illustrer le rôle d'une variable Accumulator.

Remarquons que contrairement à une *action* Spark, une variable Accumulator permet d'obtenir une valeur agrégée à partir d'un RDD sans avoir besoin de déclencher une opération couteuse comme le count(), etc… L'utilisation de variables Accumulators constitue ainsi une façon optimisée de récupérer des informations.

7.6 Architecture du cluster Spark

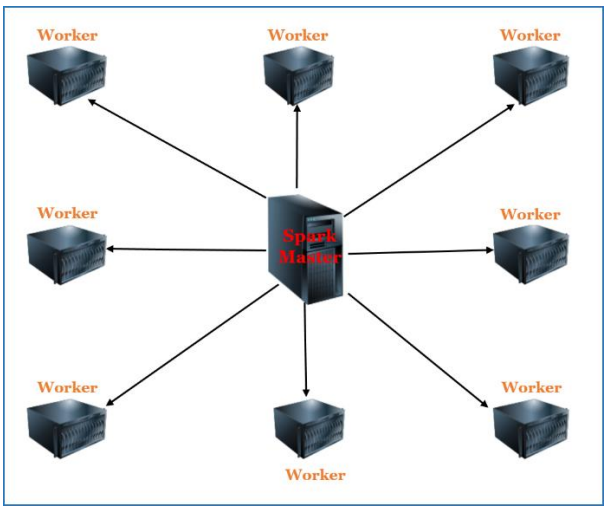
Le cluster Spark est construit sur une architecture maître-esclaves constituée d'un nœud central et de plusieurs nœuds de travail. Remarquons d'emblée que le cluster Spark superpose deux types d'architecture : une architecture "physique" et une architecture "applicative". La nature et les caractéristiques du nœud central et des nœuds de travail sont définies suivant le type d'architecture considérée. Dans l'architecture physique (cluster physique Spark), le rôle de nœud maître est assuré par le *SparkMaster* et les nœuds de travail sont appelés *Workers*. En revanche, dans l'architecture applicative (qui se matérialise lors de l'exécution d'une application) le nœud maître est appelé *Driver* tandis que les nœuds de travail sont appelés *Executors*.

Cette section a pour but de distinguer les différentes composantes de l'architecture d'un cluster Spark. Faisons tout de même remarquer que la distinction entre l'architecture *physique* et l'architecture *applicative* n'est pas nécessairement pertinente dans toutes les situations. Car la plupart des composantes de l'architecture applicative s'appuie sur les composantes de l'architecture physique. Par exemple, les *Executors* (qui sont des entités de l'architecture applicative) sont instanciés à l'intérieur des *Workers* (entités de l'architecture applicative). Dès lors, l'utilisation des termes architecture *physique* et architecture *applicative* n'a pas d'intérêt à proprement parler. Néanmoins nous continuons d'en garder l'usage ici pour des raisons pédagogiques.

7.6.1 Architecture physique

La figure ci-dessous illustre l'architecture physique du cluster Spark.

Figure 21 : Architecture physique du cluster Spark



Le *SparkMaster* représente le nœud central du cluster physique. Il assure la maintenance des métadonnées du cluster, la gestion des ressources (CPU, Mémoire, Disque, etc.). Il assure également la coordination des *Workers* ainsi que la répartition des charges entre ceux-ci. Les *Workers*, quant à eux, représentent les nœuds esclaves du cluster. Ils assurent le stockage et la réplication des données sur le cluster. Ils mettent à la disposition du SparkMaster les ressources nécessaires à l'exécution des traitements Spark. Dans l'architecture physique, le *SparkMaster* et les *Workers* sont des daemons, c'est-à-dire des processus JVM[5] à longue vie (*Long running processes*) installés sur des machines physiques interconnectés via le réseau.

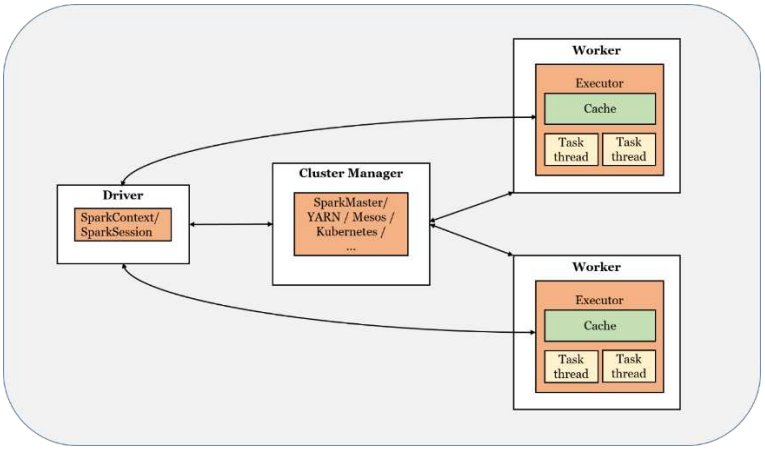
7.6.2 Architecture applicative

Contrairement à l'architecture physique où les daemons (*SparkMaster* et *Workers*) tournent en permanence sur les nœuds cluster, dans l'architecture applicative la durée de vie des daemons se limite au temps d'exécution d'une application. Ils sont matérialisés au début de l'exécution du programme de traitement et disparaissent systématiquement à la fin l'exécution de l'application. Ce sont donc des daemons à courte durée de vie.

[5] Rappelons qu'une JVM (Java Virtual Machine) est un moteur d'exécution multiplateforme capable d'exécuter toute(s) instruction(s) compilée(s) en bytecode Java.

Trois composantes majeures forment une architecture applicative Spark : le *Driver*, les *Executors* et le *Cluster Manager*. Le schéma ci-dessous illustre ces différentes composantes.

Figure 22 : Architecture applicative du cluster Spark



Le *Driver* joue le rôle de nœud central. Il assure entre autres : l'exécution de la fonction main() de l'application Spark, l'ordonnancement des tâches et leur distribution entre les Executors, le suivi et le monitoring de l'exécution de l'application. Les Executors sont les nœuds de traitement du cluster. C'est sur ces nœuds que sont distribuées les partitions des RDD et les fonctions de traitement des données. Les Executors sont instanciés à l'intérieur des Workers (composants de l'architecture physique). Le Cluster Manager est la composante qui assure la planification des ressources et l'ordonnancement des tâches sur le cluster. Nativement, le rôle de cluster Manager est assuré par le *SparkMaster*. Mais Spark laisse aussi la possibilité de choisir un cluster Manager externe en lieu et place du *SparkMaster*. On peut choisir, par exemples YARN, Mesos ou Kubernetes.

7.7 Etudes des composantes de l'architecture applicative

Cette section vise à donner des détails sur chacune des composantes de l'architecture applicative du cluster Spark.

7.7.1 Le Driver

Le Driver est un processus JVM qui joue le rôle de nœud maître dans l'architecture applicative. Il gère toutes les phases d'exécution d'une application Spark. En particulier, il :

- crée le *SparkContext* (ou la *SparkSession*) qui sert de point d'entrée à toutes les fonctionnalités de l'application Spark.
- crée les RDDs, les découpe en partitions et les distribue sur les différents nœuds de traitement du cluster (les Executors).

- stocke et maintien les métadonnées sur les RDDs et leurs partitions.
- convertit les instructions du programme de traitement en plusieurs unités de tâches qui sont envoyées aux Executors.
- exécute la fonction main() du programme de traitement
- négocie, avec le Cluster Manager, les ressources nécessaires à l'exécution des tâches.
- génère le DAG pour une exécution optimisée des tâches.

7.7.1.1 Les composants du Driver

Le Driver gère l'application Spark à travers le *SparkContext* (ou *SparkSession* intégrée à Spark depuis la version 2). En plus du SparkContext/SparkSession, le Driver est construit sur divers autres composants que sont notamment : le DAGScheduler, le TaskScheduler, le BackendScheduler et le BlockManager. Les lignes ci-dessous détaillent chacun des composants.

SparkContext

Le *SparkContext* est une passerelle qui sert de point d'entrée à toutes les fonctionnalités de l'application Spark en cours d'exécution. C'est grâce au SaprkContext que le Driver contrôle toute l'application Spark : répartition des ressources, coordination des Executors, suivi et monitoring des tâches, gestion des métadonnées. Les principaux rôles du SparkContext sont :

- la création de RDD
- l'exécution et suivi des tâches
- la gestion des variables partagées, etc.
- le monitoring du statut de l'application Spark

Le SparkContext permet aussi d'instancier des contextes spécifiques tels que le SQLContext, le HiveContext et le StreamingContext. Le SQLContext et le HiveContext sont utilisés pour gérer les applications de type SparkSQL tandis que le StreamingContext est utilisé gérer pour les applications Spark Streaming.

Par défaut, il n'y a qu'un *SparkContext* par processus Driver. Néanmoins, il est possible de modifier ce comportement en utilisant le paramètre de configuration *spark.driver.allowMultipleContexts* qui permet d'instancier plusieurs *SparkContext* dans un même processus Driver. La bonne pratique reste, toutefois, d'avoir un seul *SparkContext* par processus Driver.

SparkSession

Avant la version 2.0 de Spark, le SparkContext était l'unique point d'entrée aux applications Spark. Mais à partir de la version 2.0, Spark a introduit l'objet *SparkSession* qui a permet de combiner en un seul objet le SparkContext et ses objets dérivés notamment SQLContext et HiveContext. Désormais l'objet *SparkSession* est

l'objet qui fournit toutes les fonctionnalités pour interagir avec l'application Spark. *SparkSession* prend en charge non seulement les opérations sur les RDD mais aussi les opérations de type SQL/Hive. Rappelons, toutefois que même si SparkSession représente l'unique point d'entrée aux fonctionnalités d'une application Spark, les objets *SparkContext* et *sqlContext/HiveContext* ne sont pas encore été dépréciés. Ils restent toujours utilisables comme objets distincts pour gérer les applications Spark.

L'objet *SparkSession* (*SparkContext*) est instancié par le Driver au début de l'exécution de l'application Spark. Le Driver expose les informations d'exécution via une interface utilisateur accessible par défaut à l'adresse http://*${master_host_name}:*4040

Le DAGScheduler :

Le DAGScheduler est l'un des principaux objets qui permet au Driver de gérer l'ordonnancement des tâches de l'application Spark. Le DAGScheduler est instancié lors de l'initialisation de SparkContext. Il joue trois principaux rôles :

- **Générer le DAG d'exécution des étapes** : le DAGScheduler transforme le plan d'exécution logique (lineage de RDDs issus de transformations) en un plan d'exécution physique (matérialisé par des Stages). Lorsqu'une action est appliquée sur un RDD, le *SparkContext* transmet un plan logique au DAGScheduler qu'il traduit à son tour en un ensemble d'étapes qui sont soumises en tant que TaskSets au TaskScheduler pour exécution.
- **Déterminer les emplacements pour exécuter les tâches** : le DAGScheduler détermine les emplacements sur lesquels les tâches individuelles doivent être exécutées en tenant compte de l'état actuel du cache.
- **Gérer les échecs d'exécution :** le DAGScheduler gère d'éventuels échecs d'exécution dus à la perte de fichiers lors des shuffle en resoumettant les anciennes Stages pour reconstituer les fichiers perdus.

Le TaskScheduler :

Le *TaskScheduler* est le planificateur de tâches à l'intérieur des Stages. Son rôle est notamment de :

- réceptionner les TaskSets qui lui sont envoyés par le DAGScheduler et les transférer au Cluster Manager en vue de leur exécution.
- assurer le suivi et le monitoring de l'exécution des tâches
- gérer d'éventuelles erreurs d'exécution des tâches : en cas d'échec d'une Task à l'intérieur d'une Stage, le TaskScheduler tente de resoumettre la tâche, à plusieurs reprises si nécessaire. Et face à une impossibilité d'exécution en succès, il envoie un avertissement au DAGScheduler qui fait échouer toute la Stage et éventuellement toute l'application.

Le SchedulerBackend :

Le *SchedulerBackend* est un objet qui implémente le mécanisme par lequel le *TaskScheduler* soumet les tâches au cluster Manager. En effet, les jobs Spark peuvent être gérés par plusieurs types de cluster Managers (ex : SparkMaster en Standalone, YARN, Mesos, Kubernetes, Spark local). Mais ces cluster Managers peuvent différer non seulement par leur mécanisme d'attribution de ressources mais également par leur mode de planification de tâches. Le rôle du *SchedulerBackend* est d'agir comme un contrat d'interface qui permet au Driver d'utiliser le protocole de communication approprié pour soumettre les tâches au cluster Manager.

Le BlockManager :

Le BlockManager est un objet permettant au Driver de gérer les blocs de données, leur stockage (en cache ou sur disque) et leur réplication sur les différents nœuds de traitement du cluster Spark. Le BlockManager fournit une interface pour charger ou récupérer des blocs de données à la fois localement et à distance en utilisant espace de stockage : mémoire, disk, etc. Les objets gérés par le BlockManager sont notamment les partitions des RDDs, les sorties shuffle, les variables Broadcast, les données stream, etc.

Un BlockManager est instancié sur tous les nœuds de l'application Spark (Driver et Executors). On distingue le BlockManager maître (qui réside sur le Driver) et les BlockManagers esclaves qui résident sur les Executors. Durant toute la phase d'exécution de l'application Spark, les BlockManagers esclaves rendent compte en permanence au BlockManager maître sur l'état des blocs : localisation, taille, etc.

7.7.1.2 Localisation du Driver : Exécution en mode « client » ou en mode « cluster »

Dans l'architecture applicative Spark, il n'est pas nécessaire que le Driver réside sur un nœud du cluster Spark. Il peut être instancié sur un client distant (hors du cluster Spark). Le choix de la localisation du Driver se fait au moment de la soumission à exécution du programme de traitement. La location dépend du mode d'exécution choisi. On distingue deux modes d'exécution : le mode client et le mode cluster. Lorsque l'application est lancée sur le cluster en mode client, le Driver est instancié sur la machine cliente qui lance l'application (située le plus souvent hors du cluster Spark). Par contre, lorsque l'application est lancé en mode cluster, quel que soit l'endroit où ce lancement est effectué, le Driver sera instancié sur l'un des nœuds du cluster.

Le mode d'exécution d'une application Spark est spécifié dans les paramètres de lancement avec l'argument *--deploy-mode* qui prend deux valeurs possibles : *client* ou *cluster*.

7.7.2 Les Executors

Les Executors sont les nœuds de traitement du cluster Spark. Ils assurent l'exécution des tâches assignées par le Driver. Les Executors hébergent les blocs de données des RDD et fournissent l'ensemble des ressources nécessaires à l'exécution de ces tâches (CPU, Mémoire, disque, etc..). Un Executor est un processus JVM distinct. Ce processus est instancié à l'intérieur d'un Worker. Un Worker peut héberger un ou plusieurs Executors selon la disponibilité de ressources. Mais dans une configuration standard, le cluster Manager instancie un Executor par Worker.

Les Executors sont supervisés par le Driver. Durant l'exécution de l'application Spark, chaque Executor établit une communication avec le Driver pour lui rendre compte de l'état d'avancement des tâches dont il a la charge. La communication se fait via l'envoi des heartbeats réguliers et également par l'envoi de metrics sur les tâches actives.

Parallélisme et volumétrie : critères de performance du traitement Spark

D'une manière générale, pour augmenter la performance d'exécution d'une application, il suffit dans un premier temps d'augmenter le nombre d'Executors. Le nombre d'Executors est le premier niveau de parallélisme d'une application Spark. Le nombre d'Executors est géré à travers le paramètre de lancement *num-executors*.

En plus d'augmenter le nombre d'Executors au niveau de l'application, on peut augmenter le nombre de cores au sein de chaque Executors. Le nombre de cores constitue le deuxième niveau de parallélisme. Chaque core correspond à un thread distinct capable d'exécuter une tâche sur des blocs de données en parallèle des autres threads. Ainsi, en augmentant le nombre de cores au niveau d'un Executor, on augmente le degré de parallélisme des tâches au niveau des Executors. Pour augmenter le nombre de cores, on joue sur le paramètre de lancement : *executor-cores*.

Rappelons simplement qu'en plus du nombre d'Executors et du nombre de cores par Executor, il existe un troisième niveau de parallélisme du traitement Spark. Il s'agit du nombre de partitions du RDD/DataFrame. Le nombre de partitions correspond au nombre de blocs de données distincts. La partition est l'unité de base du parallélisme Spark, car toutes les tâches de transformation sont appliquées au niveau de la partition. Lors de l'exécution d'une application, il y a autant de Tasks que de partitions. Ainsi, pour apporter encore plus de parallélisme dans le traitement, il suffit d'augmenter le nombre de partitions. A noter que le nombre de partitions du RDD/DataFrame est défini par défaut par Spark. Mais on peut modifier cette valeur par défaut en appliquant sur le RDD/DataFrame la fonction *repartition(n)* où *n* est le nombre de partitions souhaité.

Au-delà du critère de performance lié à la parallélisation des tâches, il existe un autre critère de performance majeur de Spark. C'est la capacité à traiter de très grosses volumétries de données. Grâce à son architecture distribuée, Spark répartit les données sur plusieurs Executors de façon à équilibrer les charges de traitement. Ainsi, pour pouvoir ingérer et traiter une grosse volumétrie de données, il suffit d'augmenter la taille de mémoire de chaque Executor. La taille de mémoire des Executors est gérée à

travers le paramètre de lancement *executor-memory*. Il est également possible d'ajuster la taille de mémoire du Driver lorsque celui-ci doit réaliser des opérations de grande envergure telles que, par exemple, les actions de type *.collect()*. Le paramètre à ajuster dans ce cas est *driver-memory*.

7.7.3 Le cluster manager

Un cluster Manager est une application de gestion de ressources, de planification et d'ordonnancement des tâches sur le cluster Spark. Nativement, c'est le *SparkMaster* qui sert de cluster Manager. Mais des frameworks externes tels que YARN, Mesos, ou Kubernetes peuvent aussi être utilisés pour assurer la gestion de ressources et l'ordonnancement des tâches sur le cluster Spark.

Le choix du cluster Manager se fait lors du lancement de l'application Spark. La nature du cluster Manager dépend du "type" de lancement choisi. On distingue trois types de lancement pour une application Spark : le lancement *standalone*, le lancement avec *gestion externe* et le lancement *local*. Nous présentons ci-dessous un aperçu rapide de chacun des trois types de lancement.

7.7.3.1 Le lancement standalone

Dans le lancement *standalone*, le rôle de cluster Manager est assuré par le *SparkMaster*. C'est lui qui attribue les ressources nécessaires à l'exécution des tâches. Il est important de ne pas confondre le *SparkMaster* et le *Driver*. Le SparkMaster met à disposition du Driver les ressources nécessaires à l'exécution de l'application. Il n'est pas impliqué dans l'exécution des tâches. C'est le rôle du Driver qui, lui, est un processus (instancié soit en local sur la machine cliente, soit sur un nœud du cluster) et qui assure la coordination des Executors dans l'exécution des tâches.

En lancement standalone, le cluster Manager est défini en spécifiant l'adresse physique de *SparkMaster* dans les paramètres de lancement : *--master ${master_host_name} :${port}*.

7.7.3.2 Le lancement avec gestion externe : cas de YARN

Dans ce lancement, la planification des ressources est assurée par un cluster Manager externe : YARN, Mesos ou Kubernetes. Pour choisir YARN comme cluster Manager lors de l'exécution d'une application, il suffit de spécifier les paramètres de lancement avec l'argument *--master yarn*. Lorsqu'une application Spark est lancé avec l'argument *--master yarn*, l'attribution des ressources se déroule comme suit :

- Le Driver contacte le *ResourceManager* YARN pour demander les ressources nécessaires à l'exécution de l'application.
- Le *ResourceManager* prend contact un *NodeManager* et lui demande d'instancier l'*ApplicationMaster* qui prendra en charge le suivi de l'exécution de l'application Spark. L'*ApplicationMaster*, une fois instanciée s'enregistre auprès du *ResourceManager*. Lorsque l'application Spark est lancé avec *--deploy-*

mode cluster, le Driver est instancié sur le même container que l'ApplicationMaster. Par contre lorsque l'application est lancée avec *--deploy-mode client*, le Driver est instancié sur la machine cliente.

- L'*ApplicationMaster* négocie les ressources auprès du *ResourceManager* suivant les demandes exprimées par le Driver.
- Suivant les ressources demandées, YARN contacte les NodeManagers et leur demande d'instancier les containers dans lesquels sont instanciés les Executors.
- Les exécuteurs lancés s'enregistrent auprès du Driver afin de permettre à celui-ci de coordonner l'exécution des étapes et des tâches définies dans le DAG.
- Dès que l'exécution de l'application est terminée, le Driver s'arrête, l'*ApplicationMaster* se déconnecte du ResourceManager et s'arrête à son tour. Ainsi YARN libère tous les containers utilisés.

Pour configurer YARN comme un cluster Manager externe de Spark, il suffit de définir une variable d'environnement Spark qui pointe vers les fichiers de configuration HADOOP et YARN du cluster HADOOP. Ainsi, en soumettant une application via la commande *spark-submit* et en spécifiant *--master yarn*, le Driver Spark contacte automatiquement le RessourceManager YARN dont l'adresse est préalablement définie dans les fichiers de configuration HADOOP et YARN (ex : yarn-site.xml).

7.7.3.3 Le lancement local

Le lancement local est la troisième possibilité pour lancer une application Spark. Dans le lancement *local*, le traitement Spark est lancé et exécuté en local sur la machine cliente sans aucun cluster Manager. Pour lancer une application Spark en *local*, il suffit de spécifier le paramètre *--master local[*]*.

Dans un lancement local, il n'y a que deux processus : un Driver et un Executor. Tous les deux processus sont instanciés en local sur la machine cliente.

Rappelons que dans lancement *local*, il n'y a aucune possibilité d'augmenter le nombre d'Executors pour avoir plus de parallélisme. Car un seul Executor thread est instancié pour gérer l'exécution de l'application. Par contre, il est tout à fait possible d'augmenter le degré de parallélisme des tâches en augmentant le nombre de cores de l'Executor. Le nombre de cores est défini dans les paramètres de lancement avec *--master local[n]* où *n* représente le nombre de cores choisi.

Rappelons tout de même que le lancement *local* reste purement pédagogique. Il est utile seulement pour des tests. Il n'est nullement adapté à une utilisation en condition de production. Le lancement en standalone et surtout le lancement avec *gestion externe* restent les plus fréquemment utilisés.

8 RESSOURCES DOCUMENTAIRES

Tout d'abord, je tiens ici à remercier tous les auteurs de documents techniques, d'articles, de blogs et de ressources numériques qui ont été d'un apport considérable à la rédaction de ce document mais qui ne figurent pas dans le liste des ressources documentaires.

Bibliographie

Chambers, Bill, (2017), Spark: The Definitive Guide. O'Reilly Media.

Chang, et al., (2006), Bigtable: A Distributed Storage System for Structured Data

Dean, J., Ghemawat S., (2004), MapReduce: Simplified Data Processing on Large Clusters: 137–150.

Dimiduk, Nick; Khurana, Amandeep, (2012), HBase in Action (1st ed.). Manning Publications. p. 350. ISBN 978-1617290527.

George, Lars, (2011), HBase: The Definitive Guide (1st ed.). O'Reilly Media. p. 556. ISBN 978-1449396107.

Ghemawat et al., (2003), The Google file system, Proceedings of the nineteenth ACM Symposium on Operating Systems Principles

Lam, Chuck, (2010), Hadoop in Action (1st ed.). Manning Publications. p. 325. ISBN 978-1-935-18219-1.

Lejeune, J, (2015), Hadoop:une plate-forme d'exécution de programme Map-Reduce, École des Mines de Nantes, 83p.

M. Grover, (2017), Zookeeper fundamentals, deployment, and applications.

Renaut, B.,(2014), Hadoop/Big Data, Université de Nice Sophia-Antipolis, 114p

Vohra, Deepak, (2016), Practical Hadoop Ecosystem: A Definitive Guide to Hadoop-Related Frameworks and Tools (1st ed.). Apress. p. 429. ISBN 978-1-4842-2199-0.

White, T., (2015), Hadoop: the definitive guide, Beijing, China: Tsinghua University Press.

Zaharia M. et al, (2010), Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing. USENIX Symp. Networked Systems Design and Implementation

Urls utiles

https://hadoop.apache.org/docs/current/
https://hbase.apache.org/book.html
https://spark.apache.org/documentation.html
https://kafka.apache.org/documentation/
https://cwiki.apache.org/confluence/display/Hive/Home