La plateforme Hadoop : architecture et principe de fonctionnement

14 novembre 2019

Note : *La présentation qui suit se limite aux points essentiels de l'architecture et du fonctionnement de Hadoop, en mettant l'accent sur les éléments directement utiles à la réalisation de votre projet. De nombreux aspects relatifs à l'optimisation, à l'initialisation, au placement, à l'administration, ou à l'interface programmatique ou utilisateur sont omis. Si vous êtes curieux, ou désireux d'approfondir certains aspects (par exemple en vue de réaliser la seconde étape du projet), les références données sur la page Moodle de l'UE (rubrique « Projet ») constituent un bon point de départ.*

La philosophie générale de la plateforme Hadoop est de permettre de résoudre des problèmes par la force brute : on se propose de permettre au programmeur de considérer qu'il dispose de ressources (stockage, calcul) en quantité illimitée, et d'un coût négligeable. Sous ces hypothèses, on peut chercher à minimiser les temps de traitement, en négligeant l'efficacité des traitements.

La mise en œuvre de ce paradigme de programmation implique de résoudre les difficultés résultant du recours à une grande quantité de ressources (processeur, stockage, réseau...) :

- dès lors que des milliers de machines et périphériques concourent à un traitement, la défaillance d'un ou plusieurs composants en cours d'exécution devient la règle. Il est donc indispensable de mettre en place des mécanismes de *tolérance aux pannes*, permettant de masquer (gérer automatiquement) les défaillances. Ces mécanismes s'appuient notamment sur la redondance et sur la capacité à ajuster dynamiquement les ressources allouées à un traitement.
- Une gestion efficace des ressources physiques est cruciale au regard des volumes de calcul et de données en jeu. Les besoins d'un traitement excèdent en effet de plusieurs ordres de grandeur les capacités d'un composant pris isolément. Pour tirer profit du grand nombre de ressources, il faut donc éviter que certains composants se trouvent surchargés alors que d'autres restent inemployés. C'est l'objet des stratégies d'*équilibrage de charge*. Par ailleurs, les calculs intermédiaires peuvent s'avérer particulièrement gourmands en termes d'espace mémoire/de stockage ; il faut donc veiller à ce que chaque machine dispose de suffisamment de ressources libres pour mener à bien les tâches qui lui sont échues.
- Enfin, la gestion de la synchronisation entre tâches d'un même traitement impacte fortement les temps d'exécution : il s'agit de limiter le plus possible les points de synchronisation globaux.

1 Système de fichiers

Le système de fichiers HDFS (Hadoop Distributed File System) est conçu pour faciliter un accès massivement concurrent à de larges volumes de données [1]. Pour cela, les fichiers utilisateur sont fragmentés en blocs (*chunks*) de taille identique (typiquement 64 MB), qui sont dupliqués à des fins de tolérance aux pannes et d'efficacité. Un pool (*cluster*) de serveurs HDFS est coordonné par un serveur maître, le *NameNode*, qui gère le catalogue du système de fichiers (métadonnées et localisation des différentes copies des chunks de chaque fichier). Les serveurs du pool (*DataNodes*) gèrent les accès aux chunks conservés sur la machine à laquelle ils sont affectés. Habituellement chaque machine de la grappe HDFS accueille un *DataNode*.

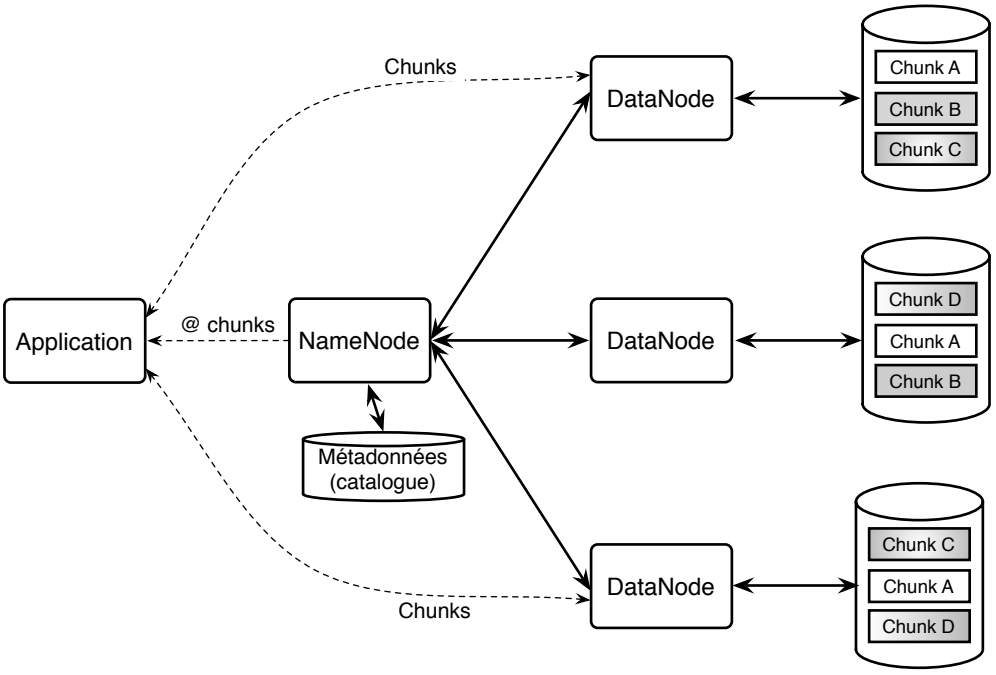

FIGURE 1 – architecture de HDFS

1.1 Applications cibles

HDFS est conçu pour faciliter l'accès aux données d'applications suivant le modèle MapReduce. Ces applications présentent des caractéristiques très particulières :

— le mode d'accès principal est le parcours séquentiel en lecture de fichiers de données. Ce type d'accès est propice à la parallélisation, puisqu'il permet de lire le fichier en parallèle, par fragments.

— les fichiers de données sont écrits une seule fois, puis lus (utilisés) plusieurs fois. Dans ces conditions, les difficultés liées aux conflits entre écritures ou entre lectures et écritures sont réduits au minimum.

Sur cette base, HDFS est nettement moins efficace en ce qui concerne

— les accès aléatoires à un fichier

— les écritures sur un fichier existant ; il existe cependant un mode d'écriture qui reste raisonnablement efficace compte tenu du profil des applications MapReduce, à savoir l'ajout de données en fin de fichier.

1. Un traitement Hadoop typique met en jeu des milliers de processeurs travaillant sur des dizaines de téraoctets de données.

1.2 Architecture et structures de données

Le *NameNode* gère les métadonnées du système de fichiers : pour chaque fichier, une structure analogue à un i-nœud regroupe les informations utiles à la gestion du fichier. Cette structure comporte en particulier :

- la taille du fichier
- la taille des blocs (*chunks*)
- la liste des identifiants de blocs (*chunk handles*) constituant le fichier
- le facteur de duplication des blocs du fichier (habituellement 3) : chacun des blocs est en effet dupliqué et ses copies sont distribuées sur différents *DataNodes*.

La taille importante des chunks permet à cet ensemble de métadonnées (*image*) de rester suffisamment compact pour être conservé en mémoire vive, ce qui assure l'efficacité des accès aux données du *NameNode*.

Outre l'*image*, le *NameNode* gère l'arborescence des noms de fichiers, ainsi qu'une table d'implantation des copies de chaque *chunk* sur les différents *DataNodes*.

La fiabilité du *NameNode* et de ses métadonnées est essentielle. Aussi,

- les métadonnées sont elles régulièrement sauvegardées en mémoire stable ;
- un (ou des) serveurs de secours réalisent des copies de ces métadonnées (périodiquement ou en continu), afin de prendre le relai en cas de défaillance du *NameNode*.

Les *DataNodes* conservent les copies des blocs (*chunks*). Ces blocs sont généralement conservés dans un répertoire spécifique du nœud hôte, chaque bloc apparaissant comme un fichier dont le nom est l'identifiant de bloc (*chunk handle*). Afin de permettre un contrôle d'intégrité des données du bloc, un fichier est associé à chaque bloc, contenant un checksum du contenu du bloc.

Le *NameNode* supervise la disponibilité des *DataNodes* au moyen d'un protocole systolique : les *DataNodes* contactent le *NameNode* à intervalles réguliers. Un *DataNode* dont la requête ne parvient pas dans les délais est considéré comme défaillant, et les copies des blocs qu'il gère sont considérées comme indisponibles. Il faut noter que ce protocole permet un fonctionnement asynchrone, donc plus efficace du *NameNode*.

1.3 Mise en œuvre de la duplication : opérations sur les fichiers

Lecture d'un fichier

1. Contacter le *NameNode*.
2. Obtenir les métadonnées : identifiants de blocs (*chunk handles*) formant le fichier.
3. Pour chaque *chunk handle*, obtenir la liste des *DataNodes*.
4. Contacter l'un quelconque des *DataNodes* pour obtenir les données (sans passer par le *NameNode*).

Ecriture d'un fichier

1. Une requête d'écriture auprès du *NameNode* fournit un *DataNode* responsable (primaire) et un nouveau numéro de version.
2. Envoi des données
   - le client envoie ses données à écrire au *DataNode* primaire ;
   - le primaire propage à l'une des copies (secondaire), qui propage à la suivante, etc... Les données sont conservées en mémoire, non écrites.
3. Écriture des données
   - le client attend l'acquittement de toutes les copies (ou de la dernière) ;
   - il envoie un message de validation au primaire ;
   - le primaire ordonne les écritures (numéro de version), effectue son écriture et informe les copies secondaires ;
   - les secondaires effectuent l'écriture et l'acquittent au primaire ;
   - le numéro de version garantit que l'ordre des opérations est identique sur tous les nœuds.
4. Le primaire informe le client.

1.4 Interface utilisateur

Le système de fichiers HDFS est implanté au dessus du système de fichiers hôte. Il offre une interface programmatique, ainsi qu'un shell proposant des opérations basiques sur les répertoires (mkdir, rmdir, ls) et les fichiers (cp, mv, rm, cat...).

2 Plateforme d'exécution

La plateforme d'exécution ($Yarn$ [2]) est calquée sur l'architecture de HDFS. Elle est constituée

- d'un *RessourceManager*, qui arbitre et supervise globalement l'allocation de ressources aux différentes applications ; ce composant critique est exécuté par un nœud dédié.
- de *NodeManagers* exécutés sur les différentes machines de la grappe Hadoop. Chaque *NodeManagers* fournit l'ensemble de ressources locales [3] (mémoire, processeur...) nécessaires à l'exécution des différentes tâches qui lui sont soumises. Il assure la réservation et la supervision de ces $containers/slots$. De manière analogue à HDFS, les *NodeManagers* permettent au *RessourceManager* de suivre la disponibilité des ressources au moyen d'un protocole systolique.
- d'*ApplicationMasters* (un pour chaque application lancée), contrôlant l'ordonnancement et supervisant l'exécution des tâches de chacune des applications. L'*ApplicationMaster* obtient du *RessourceManager* la liste des *NodeManagers* qui accueilleront les tâches à exécuter, et leur soumet celles-ci. La progression des *ApplicationMasters* est suivie par le *RessourceManager* au moyen d'un protocole systolique.

Note : $Yarn$ est conçu comme une plateforme d'exécution générique, pouvant supporter différents modèles et moteurs d'exécution, mais pour la suite nous nous limiterons au modèle MapReduce, qui est le modèle initial et central de la plateforme $Yarn$.

2.1 Le schéma MapReduce

Conçu pour le traitement concurrent de masses de données, le schéma MapReduce est le modèle de programmation privilégié de la plateforme Hadoop. Un programme MapReduce reçoit un ensemble de données au format ⟨Clé,Valeur⟩ et produit un ensemble de résultats au format ⟨Clé,Valeur⟩. Le programmeur spécifie des fonctions *Map*, qui vont produire des données intermédiaires à partir des données d'entrée, et des fonctions *Reduce*, qui vont agréger les résultats des *Map* pour produire les résultats finaux.

2.2 Exécution d'un programme MapReduce

L'exécution d'un programme MapReduce comporte ainsi plusieurs phases :

- le *RessourceManager* crée un *ApplicationMaster*, chargé de contrôler et superviser l'exécution de l'application ;
- l'*ApplicationMaster* détermine les *chunks* utiles au calcul, les localise auprès du $NameNode$, définit un ensemble de tâches Map (une tâche par *chunk* de données d'entrée [4]) et un ensemble de tâches Reduce, et planifie chaque tâche pour un *slot* d'exécution auprès d'un *NodeManager* ;
- les tâches Map sont exécutées en parallèle. Chaque tâche Map produit (en mémoire vive) des paires ⟨Clé,Valeur⟩.
- Ces données intermédiaires peuvent (éventuellement) être prétraitées avant d'être transmises aux tâches Reduce. Ce prétraitrement ($Shuffle$) consiste à regrouper et trier par clés les paires ⟨Clé,Valeur⟩, ce qui aboutit à associer à chaque clé une liste de valeurs. Cet ensemble ⟨Clé, Liste de valeurs⟩ est alors partitionné par clés, selon les clés associées aux tâches Reduce.

2. Yet Another Resource Negociator
3. appelé *container* (ou *slot* dans la première version d'Hadoop)
4. Avec un ajustement nécessaire, dans le cas où un enregistrement se trouve à cheval sur 2 blocs.

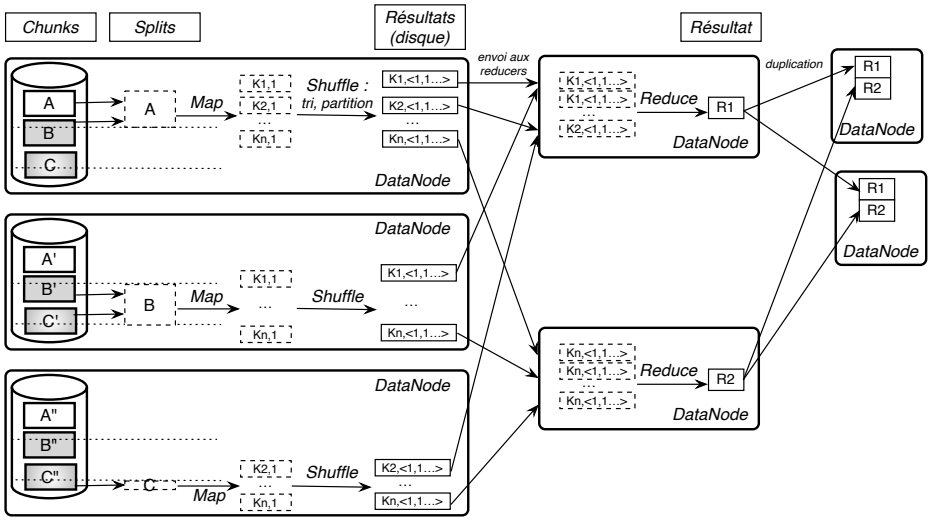

FIGURE 2 – Flot d'exécution MR

— Le résultat du prétraitement (ou sinon l'ensemble des paires ⟨Clé,Valeur⟩) est alors enregistré sur disque (un fichier par partie), puis transmis aux nœuds correspondant aux différentes tâches Reduce. [5]
— Une fois qu'elles disposent de l'ensemble des données des tâches Map, les tâches Reduce traitent les données intermédiaires et produisent leur résultat sur HDFS.

Exemple Fréquence d'occurrence des mots d'un texte Les tâches Map comptent le nombre d'occurrences des mots de leur fragment, et produisent des listes triées mot (clé),nb occurrence(valeur)), qui sont ensuite agrégées. L'annexe C est un exemple de mise en œuvre de cette application.

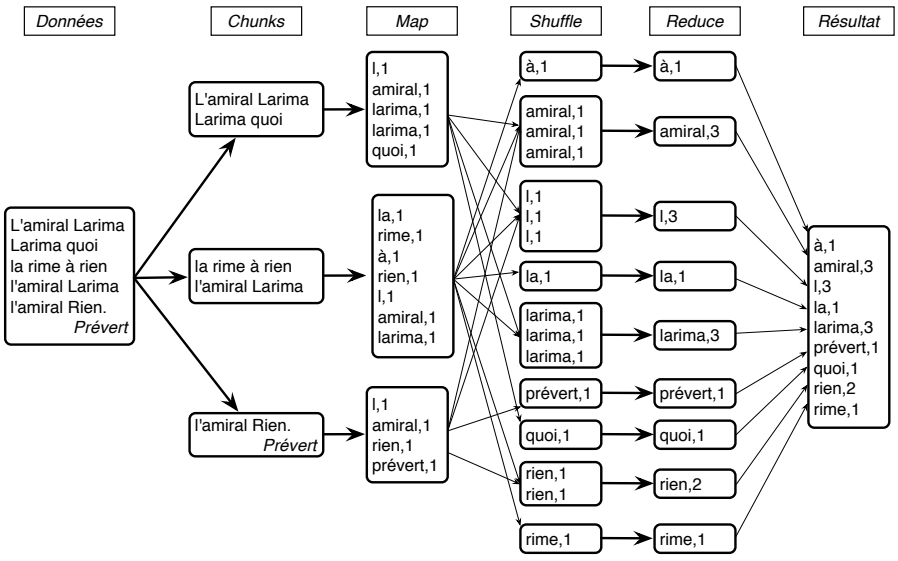

FIGURE 3 – WordCount

5. Une étape (optionelle) de prétraitement supplémentaire (*Combiner*) assimilable à un Reduce peut être effectuée avant l'envoi aux nœuds des tâches Reduce, afin d'alléger le traitement des Reduce.

2.3 Ordonnancement des tâches

Le *ResourceManager* choisit la liste de machines (*NodeManager*) qui devront exécuter les Map et les Reduce. D'une manière générale, les stratégies mises en œuvre par les ordonnanceurs diffèrent selon les critères visés, mais reposent sur quelques principes récurrents :

- une partie est constituée d'heuristiques classiques en ordonnancement, mises en œuvre dans le cadre de la gestion des ressources : mixer les profils, afin d'obtenir un usage globalement équilibré des ressources, recourir à un ordonnancement hiérarchique/multiniveaux afin de composer des politiques...
- une partie est spécifique au schéma d'exécution MapReduce, structuré par le traitement d'une masse de données en phases successives :
  - la stratégie essentielle est de colocaliser autant que possible les tâches et les données qu'elles ont à traiter (chunks, données intermédiaires), ceci afin de réduire les temps d'accès aux données à traiter, ainsi que le trafic réseau ;
  - dans cette optique, certaines politiques reportent l'exécution de tâches en attendant que leurs données deviennent localement disponibles ;
  - d'autres politiques synchronisent l'exécution des tâches, à partir des dépendances (établies ou estimées) dans le flux de données traité ;
  - une stratégie adaptative originale consiste à superviser les tâches, et à lancer des tâches en renfort si une tâche est estimée en retard ou défaillante ;
  - enfin, certaines politiques allouent les nœuds les plus fiables aux tâches coûteuses, afin de réduire la déperdition de ressources induite par la nécessité de relancer une tâche en cas de panne.

Quelques politiques classiques

- La stratégie *Full-Map* consiste à lancer un processus Map sur chaque machine hébergeant un bloc de données à traiter, via le *NodeManager* correspondant. Les processus Reduce sont affectés en fonction du nombre de slots restant disponibles. L'avantage de cette stratégie est que chaque processus Map réalisera un traitement local. Son principal inconvénient est la contention possible au niveau des slots disponibles, lors de l'exécution d'un grand nombre d'applications MapReduce.
- Dans la *stratégie sans duplication*, le *ResourceManager* utilise uniquement la copie originale des blocs, et non leurs copies dupliquées. Dans cette stratégie, le nombre des Map et de Reduce est déterminé par l'utilisateur lors de la soumission de l'application. Le *ResourceManager* utilise la même politique que précédemment pour placer les tâches Reduce.
- Dans la *stratégie avec duplication*, le *ResourceManager* peut utiliser les copies dupliquées d'un bloc. L'avantage de cette politique est que le *ResourceManager* va éviter de chercher d'autres blocs sur d'autres machines, si cette machine en possède des copies dupliquées. Pour cette stratégie également, le nombre des Map et de Reduce est fixé par l'utilisateur lors de la soumission de l'application, et la même politique est suivie pour les tâches Reduce.
- L'objectif de *la stratégie spéculative* est de réduire le temps d'exécution en lançant des tâches de renfort en appui des tâches Map dont la progression paraît ralentie.