HADOOP

Les mégadonnées ou Big Data sont des collections d'informations qui auraient été considérées comme gigantesques, impossible à stocker et à traiter, il y a une dizaine d'années.

• **Internet** : Google en 2015 : 10 Eo (10 milliards de Go), Facebook en 2014 : 300 Po de données

(300 millions de Go), 4 Po de nouvelles données par jour, Amazon : 1 Eo.

• **BigScience** : télescopes (1 Po/jour), CERN (500 To/jour, 140 Po de stockage), génome, environnement. . .

Les informations sont très difficiles à trouver.

La raison est que tout est enregistré sans discernement, dans l'idée que ça pourra être exploité. Certains

prêchent pour que les données collectées soient pertinentes (smart data) plutôt que volumineuses.

Le calcul distribué

Désigne l'exécution d'un traitement informatique sur une multitude de machines différentes (un cluster de machines) de manière transparente.

Problématiques:

Accès et partage des ressources pour toutes les machines.

Extensibilité: on doit pouvoir ajouter de nouvelles machines pour le calcul si nécessaire.

Hétérogénéité: les machines doivent pouvoir avoir différentes architectures, l'implémentation différents langages.

Tolérance aux pannes: une machine en panne faisant partie du cluster ne doit pas produire d'erreur pour le calcul dans son ensemble.

Transparence: le cluster dans son ensemble doit être utilisable comme une seule et même machine « traditionnelle ».

Le calcul distribué

Ces problématiques sont complexes et ont donné lieu à des années de recherche et d'expérimentation.

On distingue historiquement deux approches/cas d'usage:

- Effectuer des calculs intensifs **localement** (recherche scientifique,rendu 3D, etc.) - on souhaite avoir un cluster de machines local pour accélérer le traitement. Solution qui était jusqu'ici coûteuse et complexe à mettre en oeuvre.
- **Exploiter la démocratisation** de l'informatique moderne et la bonne volonté des utilisateurs du réseau pour créer un cluster distribué via Internet à moindre coût. Solution qui suppose qu'on trouve des volontaires susceptibles de partager leur puissance de calcul.

Le calcul distribué

Ces problématiques sont complexes et ont donné lieu à des années de recherche et d'expérimentation.

On distingue historiquement deux approches/cas d'usage:

- Effectuer des calculs intensifs **localement** (recherche scientifique,rendu 3D, etc.) - on souhaite avoir un cluster de machines local pour accélérer le traitement. Solution qui était jusqu'ici coûteuse et complexe à mettre en oeuvre.
- **Exploiter la démocratisation** de l'informatique moderne et la bonne volonté des utilisateurs du réseau pour créer un cluster distribué via Internet à moindre coût. Solution qui suppose qu'on trouve des volontaires susceptibles de partager leur puissance de calcul.

Exemple : Blue Gene (1999)

Supercalculateur « classique ».

Connecte 131072 CPUs et 32 tera-octets de RAM, le tout sous un contrôle centralisé pour assurer l'exécution de tâches distribuées.

L'architecture est ici spécifiquement construite pour le calcul distribué à grande échelle. Il s'agit d'un cluster « local » (ne passant pas par Internet).

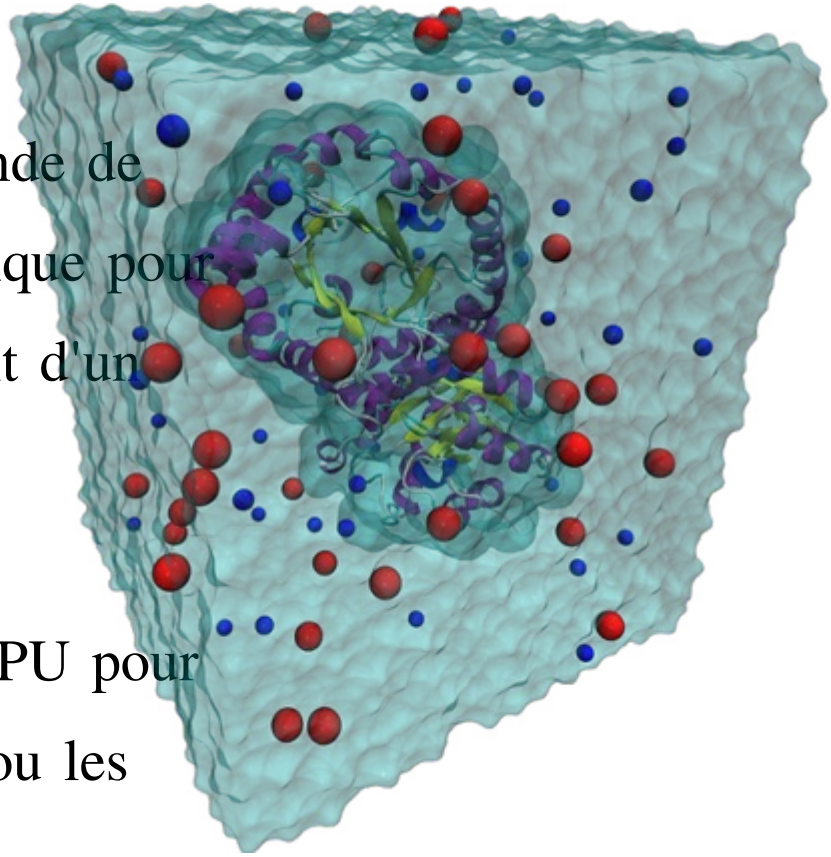
Premier supercalculateur à être commercialisé et produit (par IBM) en plusieurs exemplaires.

Utilisé pour des simulations médicales, l'étude de signaux radio astronomiques, etc.

Exemple : GPUGRID.net (2007)

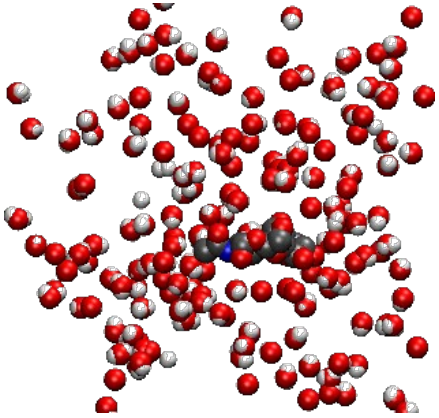
Projet de l'université Pompeu Fabra (Espagne)

permettant à des volontaires partout dans le monde de mettre à disposition le GPU de leur carte graphique pour le calcul distribué (via NVIDIA CUDA). Il s'agit d'un cluster « distant » : distribué via Internet.



Le GPU est nettement plus performant que le CPU pour certaines tâches comme le traitement du signal ou les calculs distribués sur les nombres flottants.

Utilisé pour la simulation de protein folding (maladies à prion), la simulation moléculaire, et d'une manière générale des applications médicales.



Conclusions

Pour l'exécution de tâches distribuées distantes via la mise à disposition de machines tout autour du monde par des volontaires, la création du framework BOINC a apporté une réponse efficace – de nombreux projets universitaires et scientifiques exploitant aujourd'hui la technologie.

Cependant, de nombreuses universités et entreprises ont des besoins d'exécution

locale de tâches parallélisables sur des données massives. Les solutions qui étaient disponibles jusqu'ici:

Des super calculateurs « classiques » comme Blue Gene: très onéreux, souvent trop puissants par rapport aux besoins requis, réservés aux grands groupes industriels.

Conclusions

Des solutions développées en interne: investissement initial très conséquent, nécessite des compétences et une rigueur coûteuses.

Architecture Beowulf: un début de réponse, mais complexe à mettre en œuvre pour beaucoup d'entreprises ou petites universités, et nécessitant là aussi un investissement initial assez conséquent.

Le problème

Le problème qui se posait jusqu'ici pour ce cas d'usage:

Avoir un framework déjà disponible, facile à déployer, et qui permette l'exécution de tâches parallélisables – et le support et le suivi de ces tâches – de manière rapide et simple à mettre en œuvre.

L'idée étant d'avoir un outil « off the shelf » qui puisse être installé et configuré rapidement au sein d'une entreprise/d'une université et qui permettre à des développeurs d'exécuter des tâches distribuées avec un minimum de formation requise.

L'outil en question devant être facile à déployer, simple à supporter, et pouvant permettre la création de clusters de taille variables **extensibles** à tout moment.

Avantages:

Projet de la fondation Apache – Open Source, composants complètement ouverts, tout le monde peut participer.

Modèle simple pour les développeurs: il suffit de développer des tâches map-reduce, depuis des interfaces simples accessibles via des librairies dans des langages multiples (Java, Python, C/C++...).

Déployable très facilement (paquets Linux pré-configurés), configuration très simple elle aussi.

S'occupe de toutes les problématiques liées au calcul distribué, comme l'accès et le partage des données,

Avantages:

la tolérance aux pannes, ou encore la répartition des tâches
aux machines membres du cluster : le programmeur a simplement à s'occuper du développement logiciel pour l'exécution de la tâche.

La solution: Apache Hadoop

- 2002: Doug Cutting (directeur archive.org) et Mike Cafarella (étudiant) développent
Nutch, un moteur de recherche Open Source exploitant le calcul distribué.
L'implémentation peut tourner seulement sur quelques machines et a de multiples
problèmes, notamment en ce qui concerne l'accés et le partage de fichiers.
- 2003/2004: le département de recherche de Google publie deux whitepapers, le
premier sur GFS (un système de fichier distribué) et le second sur le paradigme Map/Reduce pour le calcul distribué.

La solution: Apache Hadoop

- 2004: Doug Cutting et Mike Cafarella développent un framework (encore assez primitif) inspiré des papers de Google et portent leur projet Nutch sur ce framework.
- 2006: Doug Cutting (désormais chez Yahoo) est en charge d'améliorer l'indexation du moteur de recherche de Yahoo. Il exploite le framework réalisé précédemment...

La solution: Apache Hadoop

... et créé une nouvelle version améliorée du framework en tant que projet Open Source de la fondation Apache, qu'il nomme Hadoop (le nom d'un éléphant en peluche de son fils).

A l'époque, Hadoop est encore largement en développement – un cluster pouvait alors comporter au maximum 5 à 20 machines, etc.

- 2008: le développement est maintenant très abouti, et Hadoop est exploité par le moteur de recherche de Yahoo Ainsi que par de nombreuses autres divisions de l'entreprise.

- 2011: Hadoop est désormais utilisé par de nombreuses autres entreprises et des universités, et le cluster Yahoo comporte 42000 machines et des centaines de peta- octets d'espace de stockage.

La solution: Apache Hadoop

... et des centaines d'entreprises et universités à travers le monde.

Une technologie en plein essort

De plus en plus de données produites par des systèmes d'information de plus en plus nombreux. Ces données doivent toutes être analysées, corrélées, etc. et Hadoop offre une solution idéale et facile à implémenter au problème.

Pour le public, l'informatisation au sein des villes (« smart cities ») et des administrations se développe de plus en plus et va produire des quantités massives de données.

... le domaine de recherche/industriel autour de la gestion et de l'analyse de ces données – et de Hadoop et les technologies associées – est communément désigné

sous l'expression « Big Data ».

Estimations IDC: croissance de 60% par an de l'industrie « Big Data », pour un marché de 813 millions de dollars en 2016 uniquement pour la vente de logiciels autour de Hadoop.3

Présentation

Pour exécuter un problème large de manière distribué, il faut pouvoir découper le problème en plusieurs problèmes de taille réduite à exécuter sur chaque machine du cluster (stratégie algorithmique dite du divide and conquer / diviser pour régner).

De multiples approches existent et ont existé pour cette division d'un problème en plusieurs « sous-tâches ».

MapReduce est un paradigme (un modèle) visant à généraliser les approches existantes pour produire une approche unique applicable à tous les problèmes.

MapReduce existait déjà depuis longtemps, notamment dans les langages fonctionnels (Lisp, Scheme), mais la présentation du paradigme sous une forme « rigoureuse », généralisable à tous les problèmes et orientée calcul distribué est attribuable à un whitepaper issu du département de recherche de Google publié en 2004 (« MapReduce: Simplified Data Processing on Large Clusters »).

Présentation

MapReduce définit deux opérations distinctes à effectuer sur les données d'entrée:

- **La première, MAP**, va transformer les données d'entrée en une série de couples clef/valeur. Elle va regrouper les données en les associant à des clefs, choisies de telle sorte que les couples clef/valeur aient un sens par rapport au problème à résoudre. Par ailleurs, cette opération doit être parallélisable: on doit pouvoir découper les données d'entrée en plusieurs fragments, et faire exécuter l'opération MAP à chaque machine du cluster sur un fragment distinct.

Présentation

- **La seconde, REDUCE**, va appliquer un traitement à toutes les valeurs de chacune

des clefs distinctes produite par l'opération MAP. Au terme de l'opération REDUCE,

on aura un résultat pour chacune des clefs distinctes.

Ici, on attribuera à chacune des machines du cluster une des clefs uniques produites par MAP, en lui donnant la liste des valeurs associées à la clef. Chacune des machines effectuera alors l'opération REDUCE pour cette clef.

Présentation

On distingue donc 4 étapes distinctes dans un traitement MapReduce:

- **Découper (split)** les données d'entrée en plusieurs fragments.
- **Mapper chacun** de ces fragments pour obtenir des couples (clef ; valeur).
- **Grouper (shuffle)** ces couples (clef ; valeur) par clef.
- **Réduire (reduce)** les groupes indexés par clef en une forme finale, avec une valeur pour chacune des clefs distinctes.

En modélisant le problème à résoudre de la sorte, on le rend parallélisable – chacune de ces tâches à l'exception de la première seront effectuées de manière distribuée.

Présentation

Pour résoudre un problème via la méthodologie MapReduce avec Hadoop, on devra donc:

- Choisir une manière de découper les données d'entrée de telle sorte que l'opération MAP soit parallélisable.
- Définir quelle CLEF utiliser pour notre problème.
- Écrire le programme pour l'opération MAP.
- Ecrire le programme pour l'opération REDUCE.

... et Hadoop se chargera du reste (problématiques calcul distribué, groupement par clef distincte entre MAP et REDUCE, etc.).

Des racines dans la programmation fonctionnelle

- Map
  - Applique une fonction sur chaque élément d'une liste
  - Retourne une liste de résultats
    - Map(f(x),X[1:n]) → [f(X[1]),...,f(X[n])]
  - Exemple :
    - Map(x^2,[0,1,2,3,4,5])=[0,1,4,9,16,25]
- Reduce/fold
  - Fait l'itération d'une fonction sur une liste d'éléments
  - Applique la fonction sur les résultats précédents et l'élément courant
  - Retourne un seul résultat
  - Exemple :
    - Reduce(x+y,[0,1,2,3,4,5])=(((((0+1)+2)+3)+4)+5)=15

- Un algorithme **Map-Reduce** = job
- Opère avec des pairs **clé-valeur** : (k,V)

  -Des types primitifs, Strings ou des structures de données complexes

- Les entrées et sorties d'un job Map-Reduce ont la forme de pairs {(k,V)}
- Un job MR est défini par deux fonctions

  -map: (k1;v1)→{(k2;v2)}

  - reduce: (k2;{v2})→{(k3;v3)}

Presque tous les tutoriaux Hadoop utilisent l’exemple du **WordCount**

- Simple pour comprendre
- Pas vraiment intéressant (et très lent)
- Entrée : un grand fichier texte
- Sortie : le nombre d’occurences de chaque mot dans le fichier

Exemple :

Pour le texte "Robur the Coqueror" de Jules Verne, nous avons

11 fois le mot "corpuscles", une fois le mot "susceptible", 5 fois "clear", etc.

WordCount – comment penser en MR

Dans un code séquentiel, WordCount peut être implémenté avec un parseur String et un HashList <k, V> où :

- La clé est le "mot"
- La valeur est un entier incrémenté à chaque fois que ce mo apparaît

Même dans un environnement distribué, la procédure est presque la même :

- **Etape 1** : chaque machine parse une partie du fichier et enregistre le résultat partiel
- **Etape 2 :** à la fin, nous faisons la somme des valeurs associées à chaque clé

La partie Map

- L'entrée de Map est un ensemble de mots {w} d'une partition du texte
  - Clé=w Valeur=null
- La fonction Map calcule
  - Le nombre de fois qu'une clé w apparaît dans la partition
- La sortie de la fonction map (pour une machine) est une liste sous la forme
  - <w, nombre de mots>
- Si nous avons plusieurs machines calculant Map, la sortie "finale" est plutôt une liste d'entrées
  - {<w, {valeur1, valeur2, ...}>}

La partie Reduce

- L'entrée de reduce correspond à la sortie de la fonction map
  - {<clé, {valeur}>}, où
    - cle= “mot”
    - Chaque valeur est un entier
- La fonction Reduce calcule
  - Le nombre total d'occurrences d'un mot k :
    - La somme de toutes les valeurs avec la clé k
- La sortie de la fonction Reduce
  - <clé, N>

Le flux des données

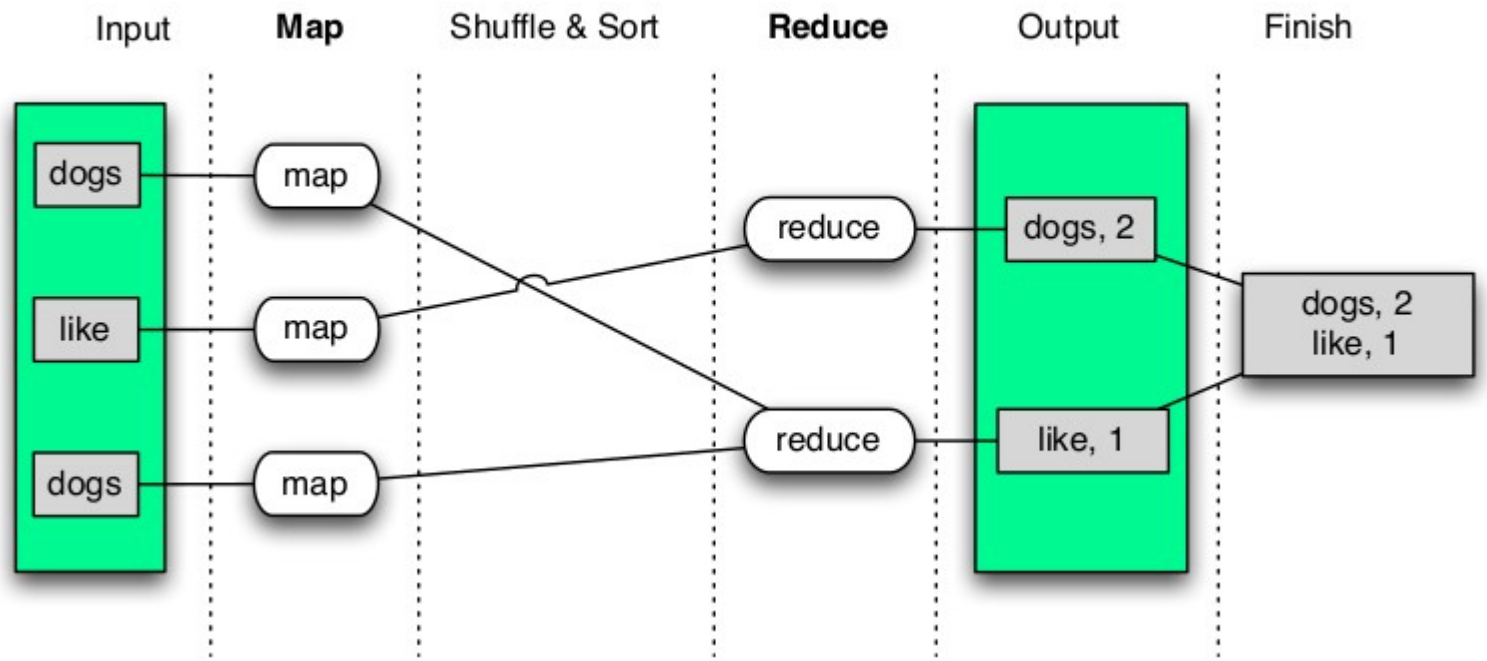
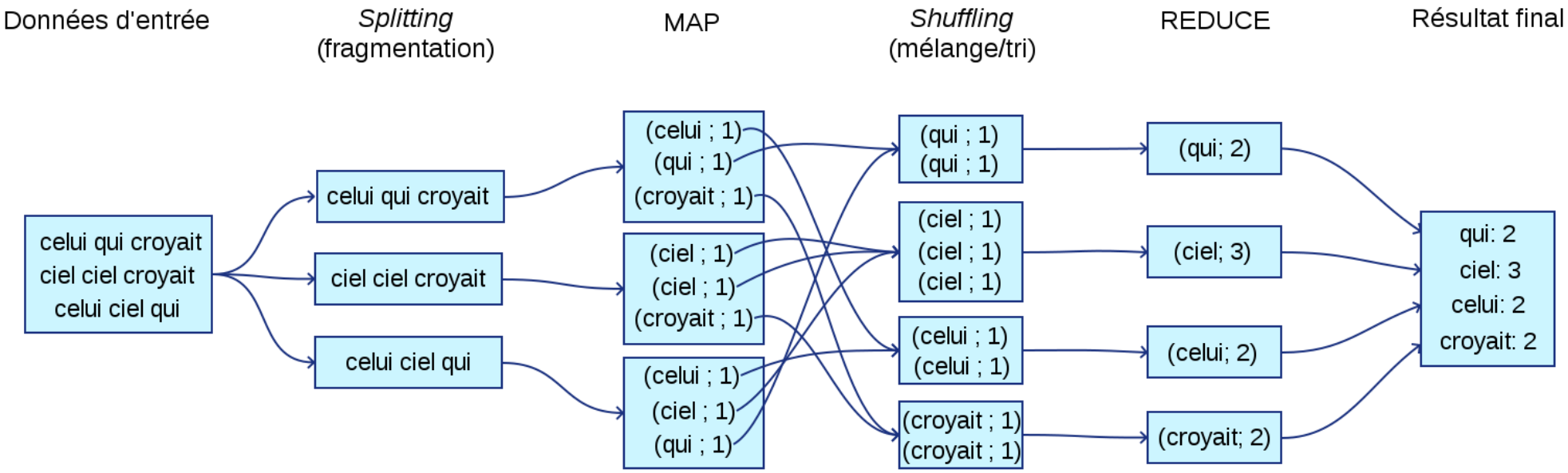
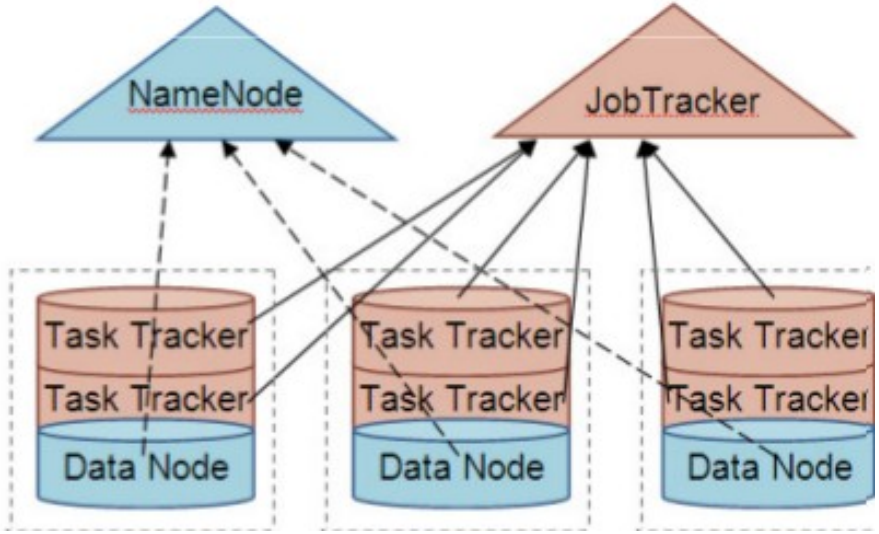


Schéma général



HDFS

- Le framework Hadoop travaille sur deux clusters virtuels :
  - Un cluster pour les données (HDFS) et un autre pour le calcul (MapReduce)
  - Ces deux clusters ont été conçus pour être couplés (travailler ensemble)
- Le Hadoop Distributed File System
  - Système de fichiers distribué (comme NFS, Lustre, etc)
  - Organise les données comme des fichiers et répertoires
- MapReduce
  - Ordonnancement et exécution des jobs



HDFS

Pour stocker les données en entrée de nos tâches Hadoop, ainsi que les résultats de nos traitements, on va utiliser **HDFS**:

Hadoop Distributed FileSystem

Il s'agit du système de fichier standard de Hadoop - au même sens que les systèmes de fichiers FAT32, NTFS ou encore Ext3FS, à la différence qu'il est évidemment distribué.

Remarque: Hadoop peut – et c'est le cas le plus fréquent – également communiquer directement avec une base de données (qu'elle soit « classique » comme MySQL ou PostGreSQL ou plus exotique comme MongoDB ou VoltDB). Ce mode d'intégration passe par le biais de ponts d'interconnexion, qui seront abordés plus loin dans le cours.

La solution: Apache Hadoop

HDFS: Hadoop Distributed File System.

Système de fichiers distribué associé à Hadoop. C'est là qu'on stocke données d'entrée, de sortie, etc.

Caractéristiques:

Distribué

- Redondé
- Conscient des caractéristiques physiques de l'emplacement des serveurs (racks) pour l'optimisation.

Repose sur deux serveurs:

Le NameNode, unique sur le cluster. Stocke les informations relative aux noms de fichiers et à leurs caractéristiques de manière centralisée (Contient des métadonnées , Permet de retrouver les nœuds qui exécutent les blocs d'un fichier)

, qui stocke les informations relatives aux noms de fichiers.

C'est ce serveur qui, par exemple, va savoir que le fichier « livre_5321 » dans le répertoire « data_input » comporte 58 blocs de données, et qui sait où ils se trouvent. Il y a un seul NameNode dans tout le cluster Hadoop.

Le DataNode, qui stocke les blocs de données eux-mêmes. Il y a un DataNode pour chaque machine au sein du cluster, et ils sont en communication constante avec le NameNode pour recevoir de nouveaux blocs, indiquer quels
blocs sont contenus sur le DataNode, signaler des erreurs, etc...

Le DataNode, plusieurs par cluster. Stocke le contenu des fichiers eux-même, fragmentés en blocs (64KB par défaut). Inspiré de GFS, lui-même issu de recherches de Google. « The Google File System », 2003.

HDFS- Réplication

Les caractéristiques de HDFS:

Il est distribué: les données sont réparties sur tout le cluster de machines.

Il est répliqué: si une des machines du cluster tombe en panne, aucune donnée n'est perdue.

Il est conscient du positionnement des serveurs sur les racks. HDFS va répliquer les données sur des racks différents, pour être certain qu'une panne affectant un rack de serveurs entier (par exemple un incident d'alimentation) ne provoque pas non plus de perte de données, même temporaire. Par ailleurs, HDFS peut aussi optimiser les transferts de données pour limiter la « distance » à parcourir pour la réplication (et donc les temps de transfert).

HDFS

Par ailleurs, le système de gestion des tâches de Hadoop, qui distribue les fragments de données d'entrée au cluster pour l'opération MAP ou encore les couples (clef;valeur) pour l'opération REDUCE, est en communication constante avec HDFS.

Il peut donc optimiser le positionnement des données à traiter de telle sorte qu'une machine puisse accéder aux données relatives à la tâche qu'elle doit effectuer localement, sans avoir besoin de les demander à une autre machine.

Ainsi, si on a par exemple 6 fragments en entrée et 2 machines sur le cluster, Hadoop estimera que chacune des 2 machines traitera probablement 3 fragments chacune, et positionnera les fragments/distribuera les tâches de telle sortes que les machines y aient accès directement, sans avoir à effectuer d'accès sur le réseau.

Écriture d'un fichier

Pour écrire un fichier:

- Le client contacte le NameNode du cluster, indiquant la taille du fichier et son nom.
- Le NameNode confirme la demande et indique au client de fragmenter le fichier en blocs, et d'envoyer tel ou tel bloc à tel ou tel DataNode.
- Le client envoie les fragments aux DataNode.
- Les DataNodes assurent ensuite la réplication des blocs.

Écriture d'un fichier

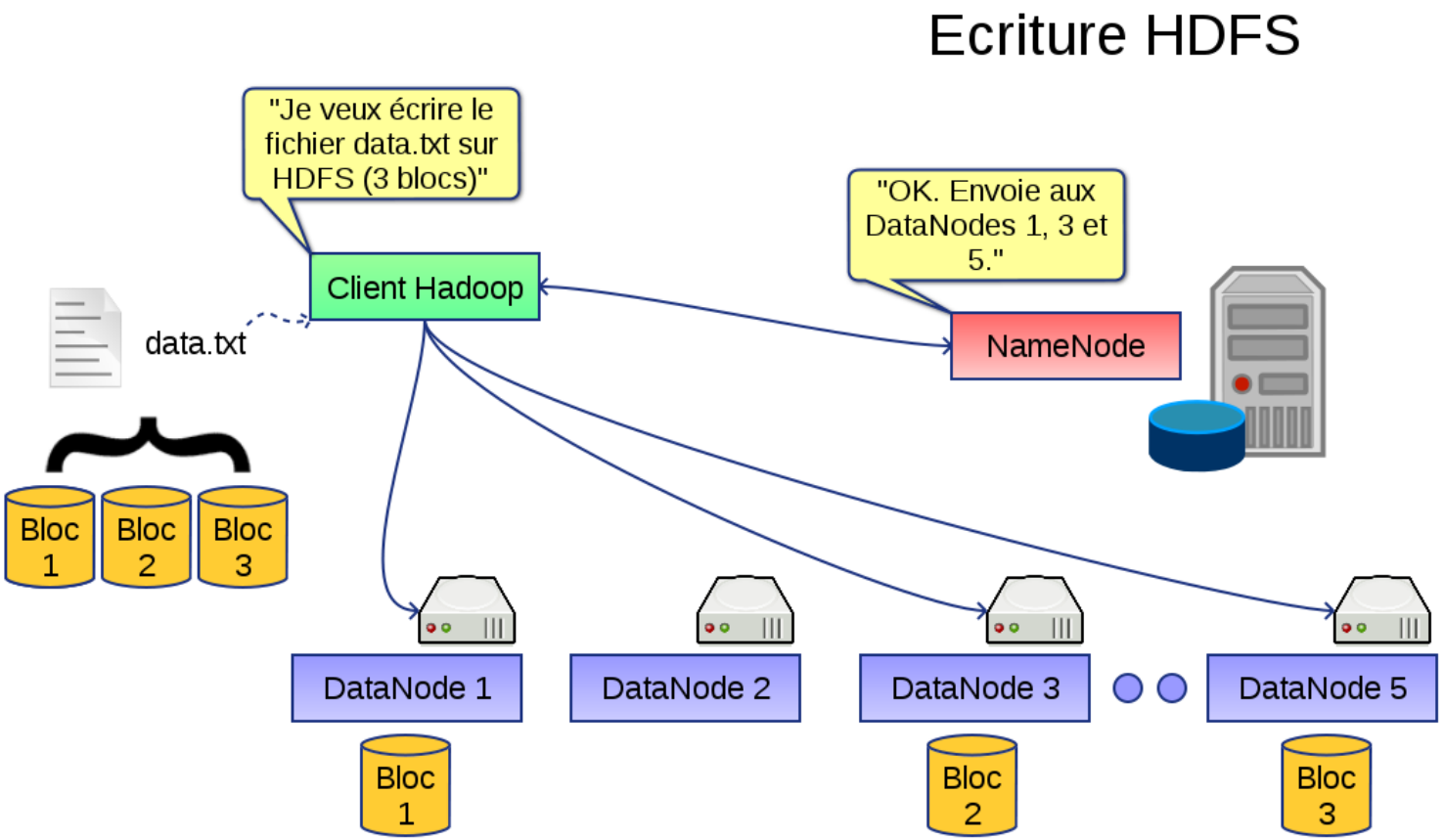
Si on souhaite écrire un fichier au sein de HDFS, on va utiliser la commande

principale de gestion de Hadoop: hadoop, avec l'option fs. Mettons qu'on souhaite stocker le fichier page_livre.txt sur HDFS.

Le programme va diviser le fichier en blocs de 64KB (ou autre, selon la configuration) – supposons qu'on ait ici 2 blocs. Il va ensuite annoncer au NameNode: « Je souhaite stocker ce fichier au sein de HDFS, sous le nom page_livre.txt ».

Écriture d'un fichier

Le NameNode va alors indiquer au programme qu'il doit stocker le bloc 1 sur le
DataNode numéro 3, et le bloc 2 sur le DataNode numéro 1.
Le client hadoop va alors contacter directement les DataNodes concernés et leur demander de stocker les deux blocs en question.
Par ailleurs, les DataNodes s'occuperont – en informant le NameNode – de répliquer les données entre eux pour éviter toute perte de données.



- Le client indique au NameNode qu'il souhaite écrire un bloc.
- Celui-ci lui indique le DataNode à contacter.
- Le client envoie le bloc au Datanode.
- Les DataNodes répliquent le bloc entre eux.
- Le cycle se répète pour le bloc suivant.

Lecture d'un fichier

Pour lire un fichier:

- Le client contacte le NameNode du cluster, indiquant le fichier qu'il souhaite obtenir.
- Le NameNode lui indique la taille, en blocs, du fichier, et pour chaque bloc une liste de DataNodes susceptibles de lui fournir.

Le client contacte les DataNodes en question pour obtenir les blocs, qu'il reconstitue sous la forme du fichier.

En cas de DataNode inaccessible/autre erreur pour un bloc, le client contacte un DataNode alternatif de la liste pour l'obtenir.

Lecture d'un fichier

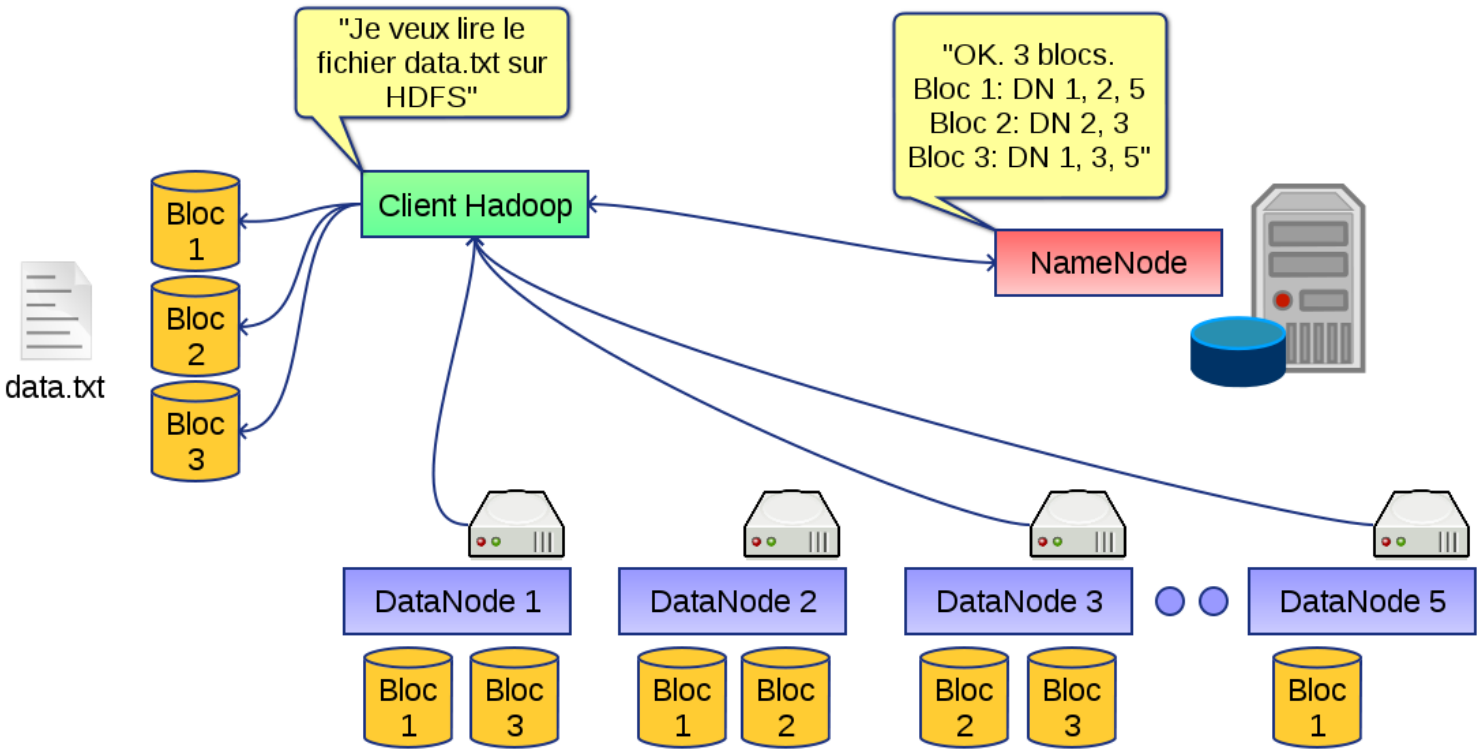
Si on souhaite lire un fichier au sein de HDFS, on utilise là aussi le client Hadoop. Mettons qu'on souhaite lire le fichier page_livre.txt.

Le client va contacter le NameNode, et lui indiquer « Je souhaite lire le fichier

page_livre.txt ». Le NameNode lui répondra par exemple « Il est composé de deux blocs. Le premier est disponible sur le DataNode 3 et 2, le second sur le DataNode 1 et 3 ».

Là aussi, le programme contactera les DataNodes directement et leur demandera de lui transmettre les blocs concernés. En cas d'erreur/non réponse d'un des DataNode, il passe au suivant dans la liste fournie par le NameNode.

Lecture HDFS



- Le client indique au NameNode qu'il souhaite lire un fichier.
- Celui-ci lui indique sa taille et les différents DataNode contenant les N blocs.
- Le client récupère chacun des blocs à un des DataNodes.
- Si un DataNode est indisponible le client le demande à un autre.

La commande hadoop fs

Comme indiqué plus haut, la commande permettant de stocker ou extraire des fichiers de HDFS est l'utilitaire console hadoop, avec l'option fs.

Il réplique globalement les commandes systèmes standard Linux, et est très simple à utiliser:

hadoop fs -put livre.txt /data_input/livre.txt

Pour stocker le fichier livre.txt sur HDFS dans le répertoire /data_input.

La commande hadoop fs

hadoop fs -get /data_input/livre.txt livre.txt

Pour obtenir le fichier /data_input/livre.txt de HDFS et le stocker dans le fichier

local livre.txt.

hadoop fs -mkdir /data_input

Pour créer le répertoire /data_input

hadoop fs -rm /data_input/livre.txt

Pour supprimer le fichier /data_input/livre.txt

d'autres commandes usuelles: -ls, -cp, -rmr, -du, etc...

HDFS + MapReduce

- L'entrée de MapReduce doit être fait à partir de fichiers dans HDFS
  - Chaque bloc contient une liste de pairs clé-valeur
- Les tâches **Map** sont préférentiellement assignées aux nœuds contenant un block
  - L'entrée des tâches Map sont des fichiers locaux, tout comme la sortie
  - Les résultats des maps seront groupés : un groupe par reducer
  - Chaque groupe est ordonnée (sorted)
- Les tâches **Reduce** sont assignées à un ou plusieurs nœuds
  - Les tâches Reduce récupèrent ses entrées à partir de tous les nœuds Map associées à son groupe
  - Le résultat est calculé et stocké dans un fichier HDFS
- La sortie est un ensemble de fichiers dans HDFS (un fichier par reducer)

Hadoop 1 vs Hadoop 2

- Jusqu'à 2012, Hadoop reposait entièrement sur le couplage MapReduce-HDFS
  - Ce sont les versions 0.2x et 1.x
- EN 2012 on a vu l'arrivée de Hadoop 2.x, qui déconnecte HDFS de MapReduce
  - Possibilité d'exécuter des applications autres que MapReduce
  - Possible grâce au nouveau mécanisme d'ordonnancement YARN
- Ce qui change
  - Une redistribution des rôles (pas très compliqué)
  - Une gestion des ressources plus structurée
  - Une API légèrement plus souple pour la programmation MapReduce
    - On garde l'ancienne API quand même
  - La réplication des services clés (Haute Disponibilité)

Hadoop 1 vs Hadoop 2

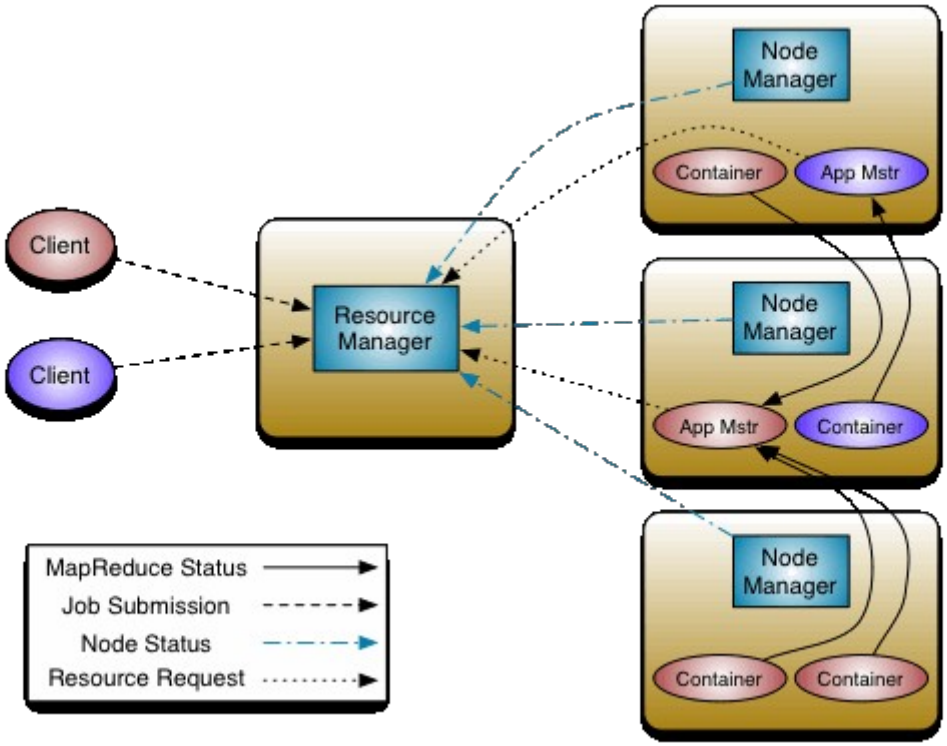
Le JobTracker a disparu de l'architecture

Ou plus précisément, ses rôles ont été répartis différemment.

L'architecture est maintenant organisée autour d'un ResourceManager dont le périmètre d'action est global au cluster et à des ApplicationMaster locaux dont le périmètre est celui d'un job ou d'un groupe de jobs.

En terme de responsabilités, on peut donc dire que : JobTracker = ResourceManager + ApplicationMaster. La différence, de part le découplage, se trouve dans la multiplicité. En effet, Un ResourceManager gère n ApplicationMaster, lesquels gèrent chacun n jobs.

Hadoop 1 vs Hadoop 2



Le ResourceManager

Le ResourceManager est le remplaçant du JobTracker du point de vue du client qui soumet des jobs (ou plutôt des applications en Hadoop 2) à un cluster Hadoop.

Il n'a maintenant plus que deux tâches bien distinctes à accomplir :

- Scheduler
- ApplicationsManager

Le Scheduler

Le Scheduler est responsable de l'allocation des ressources des applications tournant sur le cluster.

Il s'agit uniquement d'ordonnancement et d'allocation de ressources.

Les resources allouées aux applications par le Scheduler pour leur permettre de s'exécuter sont appelées des Containers.

Un Container désigne un regroupement de mémoire, de cpu, d'espace disque, de bande passante réseau, …

Ces informations sont remontées du cluster par les NodeManager, qui sont des agents tournant sur chaque noeud et tenant le Scheduler au fait de l'évolution des resources disponibles. Ce dernier peut ainsi prendre ses décisions d'allocation des Containers en prenant en compte des demandes de ressources cpu, disque, réseau, mémoire

L'ApplicationsManager

Il accepte les soumissions d'applications.

Une application n'étant pas gérée par le ResourceManager, la partie ApplicationsManager ne s'occupe que de négocier le premier Container que le Scheduler allouera sur un noeud du cluster. La particularité de ce premier Container est qu'il contient l'ApplicationMaster.

L'ApplicationMaster est le composant spécifique à chaque application qui est en charge des jobs qui y sont associés.

Lancer et au besoin relancer des jobs

Négocier les Containers nécessaires auprès du Scheduler

Superviser l'état et la progression des jobs.

Un ApplicationMaster gère donc un ou plusieurs jobs tournant sur un framework donné. Dans le cas de base, c'est donc un ApplicationMaster MRv2 qui lancer un job MapReduce. De ce point de vue, il rempli un rôle de TaskTracker.

L'ApplicationsManager est l'autorité qui gère les ApplicationMaster du cluster. A ce titre, c'est donc via l'ApplicationsManager que l'on peut

Superviser l'état des ApplicationMaster

Relancer des ApplicationMaster

De gros changements dans les API

Les évolutions architecturales présentés ci dessus ont induit des modifications dans les APIs d'Hadoop.

Cependant, une couche de rétrocompatibilité avec Hadoop 0.20.2 est maintenue afin de permettre une migration en douceur.

La seule contrainte étant de recompiler ses jobs avec la nouvelle version d'Hadoop.

Hadoop 2

Dans Hadoop 2, on peut mettre deux namenodes en mode actif/attente.

Cela signifie que le namenode actif est celui qui est utilisé par le cluster tandis que celui en attente se contente d'écouter ce qui se passe. Cela est différent d'un maître/esclave dans la mesure où chaque datanode envoi maintenant ses informations aux deux namenodes. L'actif et celui en attente. La bascule peut donc se faire à chaud et donc plus rapidement lorsque le noeud actif tombe. La conséquence de ce mode de fonctionnement est que le SLA est meilleur puisque le temps moyen avant redémarrage est diminué.

Un point d'attention majeur toutefois est qu'il faut que les deux namenodes partagent un répertoire. Cela peut être via NFS ou d'autres méthodes mais c'est nécessaire pour faire fonctionner le mode actif/attente. Il faut donc être vigilant afin de ne pas remplacer un SPOF par un autre.

La fédération HDFS

Dans un cluster HDFS, un namenode correspond à un espace de nommage (namespace). Jusqu'à présent, on ne pouvait utiliser qu'un namenode par cluster. La fédération HDFS permet de supporter plusieurs namenodes et donc plusieurs namespace sur un même cluster.

Un cas d'usage de la fédération HDFS serait d'isoler les directions métier sur un socle commun de stockage. On évite ainsi qu'un job lancé par une direction métier n touche, ne déplace voire supprime (collision de noms, faute de frappe dans le job, … un fichier, un répertoire important pour une autre direction.

En terme d'infrastructure et de dev, la fédération HDFS est intéressante car elle permet, sur un même cluster HDFS déjà en place avec ses procédures d'exploitation, de maintenance, … de fournir un espace de production, un espace de recette et un espace de développement parfaitement isolés. Il devient donc possible par exemple de développer des algorithmes de d'analyse nécessitant de gros volumes de données en entrée sur un seul et même cluster HDFS commun.

Conclusion

Les changements architecturaux d'Hadoop 2.0.0 sont, à minima, intéressants à deux titres :

Du point de vue ops, haute disponibilité et la capacité de mutualiser l'infrastructure HDFS;

Du point de vue dev et métier, le découplage par rapport à Map Reduce va permettre de faire émerger d'autres framework. Hadoop n'est donc plus un moteur MapReduce, mais bien un moteur générique pour du calcul distribué.

Écosystème Hadoop : une foire aux sous-projets

Comme expliqué précédemment, Hadoop est un système distribué orienté batch, taillé pour le traitement de jeux de données volumineux. Les utilisateurs d'Hadoop se retrouvent alors à manipuler le système de fichiers HDFS ou à développer des programmes MapReduce bas niveau en partant souvent de rien. Des sous-projets à Hadoop sont nés de ce constat et offrent des mécanismes et fonctionnalités qui simplifient la manipulation et le traitement des jeux de données volumineux.