DATUM ACADEMY

Hadoop, Spark & Map/Reduce

Benjamin Renaut
Mis à jour par Sergio Simonian

Plan

Module 1 :

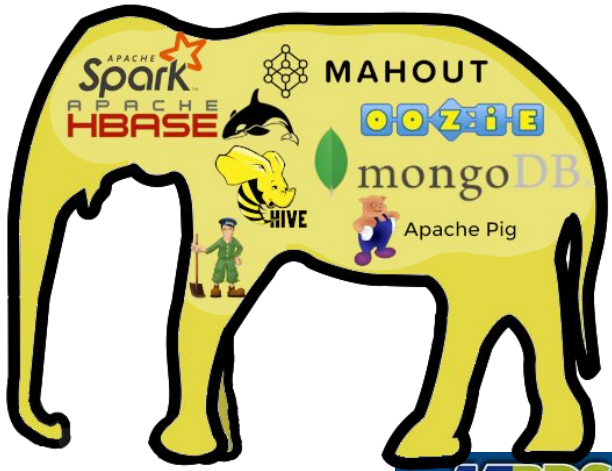
- Big Data et le calcul distribué
- Le paradigme Map/Reduce
- Introduction à Hadoop

Module 2 :

- Programmation Hadoop

Module 3 :

- Spark
- Écosystème autour d'Hadoop



Prérequis

- Connaissances générales en informatique. (Module 1)
- Des notions en Java. (Module 2)
- Des notions en Python. (Module 3)

DATUM ACADEMY

Big Data et le calcul distribué

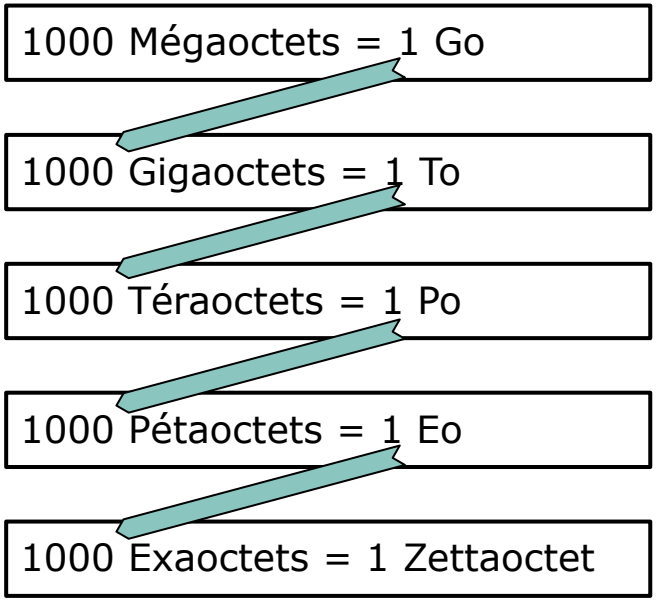
Introduction - Problématiques - Cas d'usages - Histoire d'Hadoop

Introduction

Le monde est de plus en plus connecté :

- → En 2016 - **16.1 zettaoctets** de données ont été créées ou répliquées
- → En 2020 - **64.2 zettaoctets**
  (source: Dave Reinsel, vice-président senior, Global DataSphere d'IDC)

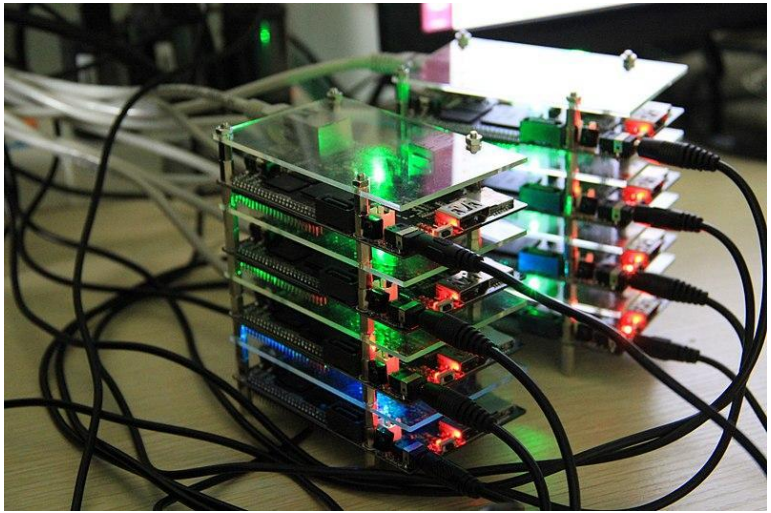
=> Nécessité des traitements distribués



Calcul distribué

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

Exemple d’un cluster Cubieboard

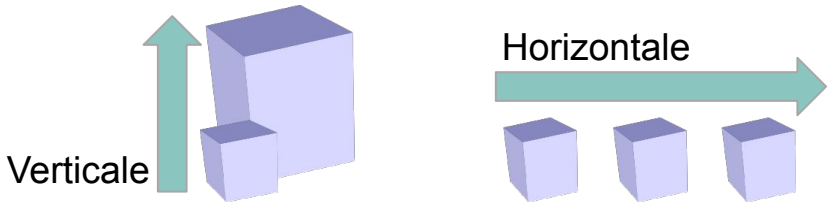


Problématiques du calcul distribué

Extensibilité

On doit pouvoir gérer la mise à l'échelle verticale et horizontale.

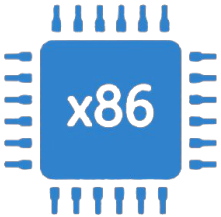
- ➔ Verticale = ajout de ressources supplémentaires aux machines courantes.
- ➔ Horizontale = ajout de nouvelles machines.

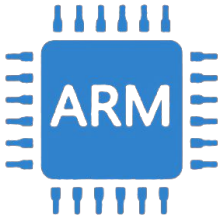


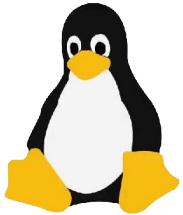
Problématiques du calcul distribué

Hétérogénéité

Les machines doivent pouvoir avoir différentes architectures, utilisant différents systèmes d'exploitation et langages de programmation.



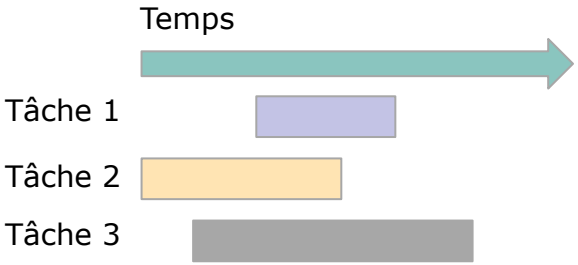


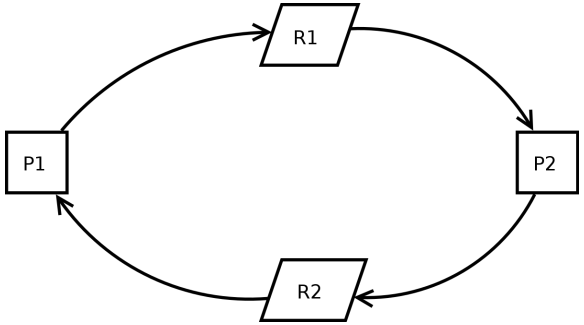


Problématiques du calcul distribué

Accès et partage des ressources pour toutes les machines

Problème de l'accès concurrentiel aux données (consistance) et ressources (équité / optimisation)

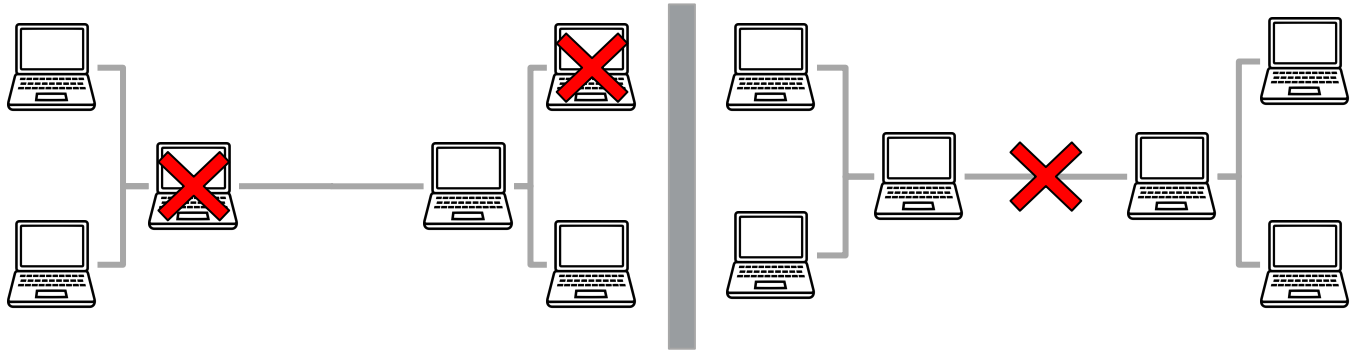




Problématiques du calcul distribué

Tolérance aux pannes

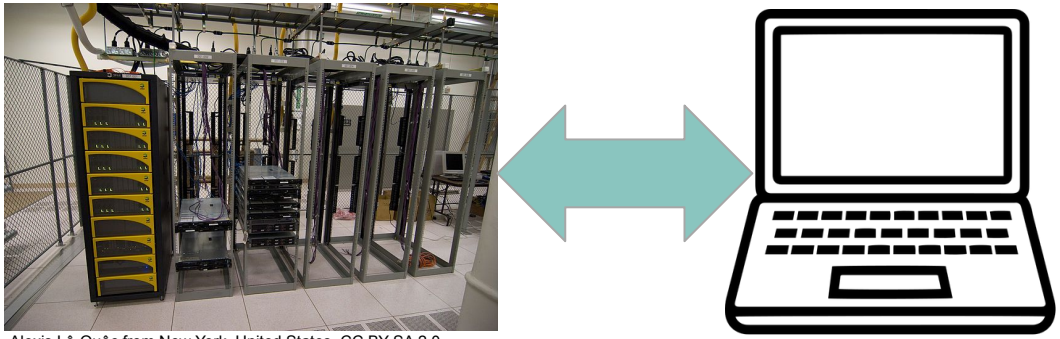
Une machine en panne faisant partie du cluster ne doit pas produire d'erreur pour le calcul dans son ensemble.



Problématiques du calcul distribué

Transparence

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



Alexis Lê-Quôc from New York, United States, CC BY-SA 2.0 <https://creativecommons.org/licenses/by-sa/2.0>, via Wikimedia Commons

Cas d'usage du calcul distribué

Exemple : Blue Gene (1999) supercalculateur

- ➔ Connecte **131072 CPUs** et **32 tera-octets de RAM**, le tout sous un contrôle centralisé pour assurer l'exécution de tâches distribuées.
- ➔ Premier supercalculateur à être commercialisé et produit (par IBM) en plusieurs exemplaires.
- ➔ Utilisé pour des simulations médicales, l'étude de signaux radio astronomiques, etc.

Argonne National Laboratory
USA

Cas d'usage du calcul distribué

Exemple : Folding@home (cluster distribué via Internet)

- ➔ Équivalent à un supercalculateur exascale
- ➔ Pour but de simuler le repliement des protéines afin de permettre de développer de nouveaux médicaments, notamment contre la maladie d'Alzheimer, certains types de cancers, la COVID etc.

Cas d'usage du calcul distribué

Exemple : Cluster Beowulf (1998)

- Architecture logique définissant un moyen de connecter plusieurs ordinateurs personnels entre eux pour l'exécution de tâches distribuées sur un réseau local.
- Une machine maître distribue les tâches à une série de machines esclaves.
- Permet d'obtenir un système de calcul distribué à hautes performances à partir d'un matériel peu onéreux.

Le problème

Des universités et entreprises ont des besoins d'exécution locale de tâches parallélisables sur des données massives. Les solutions qui étaient disponibles avant:

- → Des **supercalculateurs** comme Blue Gene: très onéreux, souvent trop puissants.
- → Des **solutions développées en interne** : investissement initial très conséquent, nécessite des compétences en systèmes distribués.
- → **Architecture Beowulf** : complexe à mettre en œuvre.

Le besoin

Avoir un outil :

- ➔ Disponible
- ➔ Facile à déployer, simple à supporter
- ➔ Permettant l'exécution de tâches parallélisables (avec le support et le suivi de ces tâches)
- ➔ Permettant la création de clusters de taille variables extensibles à tout moment
- ➔ S'occupant de toutes les problématiques liées au calcul distribué

DATUM ACADEMY

Le paradigme Map/Reduce

Présentation - Exemples - Schéma général

Présentation

- Stratégie algorithmique : **diviser pour régner** (*divide and conquer)*
- Approche formelle pour les algorithmes parallèles
- Inspiré par les fonctions Map/Reduce existant dans les langages fonctionnels (par exemple : Lisp)
- Article de recherche de Google publié en 2004 (« MapReduce : Simplified Data Processing on Large Clusters »).

Présentation

On distingue **4 étapes distinctes** dans un traitement Map/Reduce :

1. **Découper (Split)** les données d'entrée en plusieurs fragments.
2. **Mapper (MAP)** chacun de ces fragments pour obtenir des couples (clef ; valeur).
3. **Grouper (Shuffle)** ces couples (clef ; valeur) par clef.
4. **Réduire (REDUCE)** les groupes indexés par clef en une forme finale.

Présentation

Pour résoudre un problème via la méthodologie Map/Reduce avec Hadoop :

- → 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**.
- → Écrire le programme pour l'opération **REDUCE**.

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

Exemple 1 : Comptage des mots

- **Objectif :** compter les occurrences des mots dans un texte
- **Problème :** Imaginez une quantité importante de texte, plus que ce qu'une seule machine peut stocker.
- Exemple Hadoop très courant

Exemple 1 : Comptage des mots

Nos données d'entrée (le texte) :

Et mes voeux et mes promesses
Ne sont que feintes caresses,
Et mes voeux et mes promesses
Ne sont jamais que du vent.

(Pierre Corneille, Chanson, 1626, fragment)

Pour simplifier l'exemple, nous allons :

- ➔ Supprimer toute ponctuation
- ➔ Passer le texte en minuscules

Exemple 1 : Comptage des mots

Découpage du texte **par ligne :**

et mes voeux et mes promesses

ne sont que feintes caresses

et mes voeux et mes promesses

ne sont jamais que du vent

→ On obtient **4 fragments** depuis nos données d'entrée.

→ Chaque fragment sera la donnée d'entrée de notre opération MAP.

Exemple 1 : Comptage des mots

- L'opération Map produit un **ensemble des couples** (clef; valeur) qui seront par la suite regroupées par clef.
- C'est à nous d'implémenter l'opération MAP et de déterminer la clef à utiliser.
- Pour regrouper par mot il suffit de produire des couples **(MOT; 1)** pour chaque mot présent dans le fragment.

Exemple 1 : Comptage des mots

→ Le code de notre opération MAP (en pseudo code) :

```
POUR MOT dans LIGNE, FAIRE:
    GENERER COUPLE (MOT; 1)
```

→ Pour chacun de nos fragments, les couples (clef; valeur) générés seront donc :

et mes voeux et mes promesses	(et;1) (mes;1) (voeux;1) (et;1) (mes;1) (promesses;1)
ne sont que feintes caresses	(ne;1) (sont;1) (que;1) (feintes;1) (caresses;1)
et mes voeux et mes promesses	(et;1) (mes;1) (voeux;1) (et;1) (mes;1) (promesses;1)
ne sont jamais que du vent	(ne;1) (sont;1) (jamais;1) (que;1) (du;1) (vent;1)

Exemple 1 : Comptage des mots

- **Shuffle :** Groupement par **clef** distincte.
- Est effectuée **automatiquement** par Hadoop (de manière distribuée).
- On obtiendra les **12 groupes** suivants :

(et;1) (et;1) (et;1) (et;1)	(ne;1) (ne;1)	(feintes;1)
(mes;1) (mes;1) (mes;1) (mes;1)	(sont;1) (sont;1)	(caresses;1)
(promesses;1) (promesses;1)	(que;1) (que;1)	(vent;1)
(voeux;1) (voeux;1)	(jamais;1)	(du;1)

Exemple 1 : Comptage des mots

- L'opération **REDUCE** reçoit un seul groupe à la fois.
- Dans notre cas, elle va **additionner** toutes les **valeurs** liées à la clé spécifiée et renvoyer un seul couple clef/valeur (MOT, TOTAL).

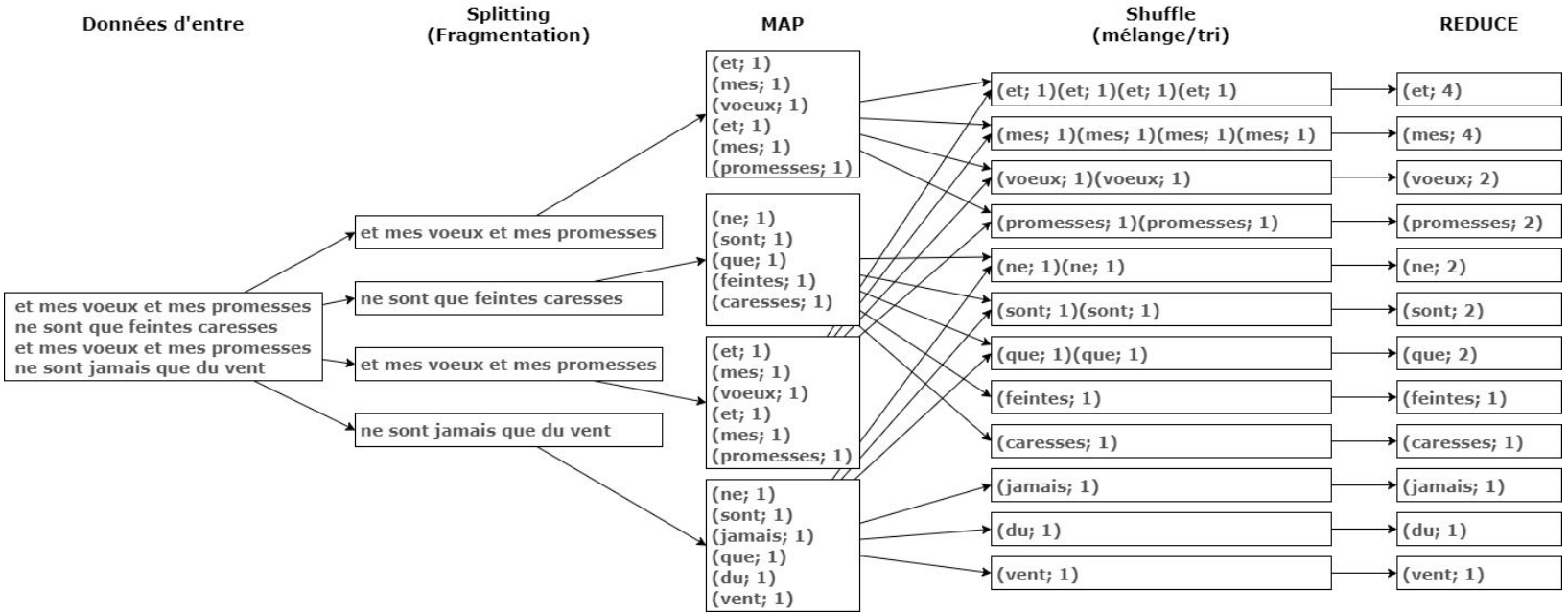
```
TOTAL = 0
POUR VALEUR dans VALEURS, FAIRE:
      TOTAL += VALEUR
RENVOYER COUPLE (CLEF; TOTAL)
```

Exemple 1 : Comptage des mots

- **Résultat final :** un couple (clef; valeur) par mot distinct.
- La clef est le mot, la valeur est le nombre d'occurrences.

et: 4
mes: 4
ne: 2
promesses : 2
que: 2
sont: 2
voeux: 2
du: 1
vent: 1
[…]

Schéma général



Exemple 2 : Statistiques web

- **Exemple :** compter le nombre de visites par page d'un site Internet.
- **Données d'entrée :** (fichiers de log d'un serveur web)

```
/index.html [19/Oct/2013:18:45:03 +0200]
/contact.html [19/Oct/2013:18:46:15 +0200]
/news.php?id=5 [24/Oct/2013:18:13:02 +0200]
/news.php?id=18 [24/Oct/2013:18:14:31 +0200]
...etc...
```

Exemple 2 : Statistiques web

- **Split :** par ligne.
- **MAP :** supprime les paramètres GET et la date. Génère des couples **(SITE; 1).**
- **REDUCE :** est identique à l'exemple « comptage des mots ».
- On pourrait aussi utiliser MAP pour filtrer par date, IP etc.

Exemple 3 : Amis en commun

- Un réseau social comportant des millions d'utilisateurs.
- Pour chaque utilisateur, on a la liste des ses amis.
- **Objectif :** Trouver, pour tout couple d'utilisateurs, leurs amis communs.

Exemple 3 : Amis en commun

➔ **Données d'entrée** : (sous la forme Utilisateur => Amis)

A => B, C, D

B => A, C, D, E

C => A, B, D, E

D => A, B, C, E

E => B, C, D

Exemple 3 : Amis en commun

- **Split :** par ligne.
- **MAP :**
  - Séparer l'utilisateur et ses amis.
  - Pour chaque ami, générer un couple (clef; valeur).
  - La clef est la combinaison « Utilisateur - Ami » **triée par ordre alphabétique**. (clef « B-A » devient « A-B »).
  - La valeur est la liste d'amis.

Exemple 3 : Amis en commun

- Le pseudo code de notre opération MAP :

```
UTILISATEUR = [PREMIERE PARTIE DE LA LIGNE]
POUR AMI dans [RESTE DE LA LIGNE], FAIRE:
  SI UTILISATEUR < AMI:
        CLEF = UTILISATEUR + "-" + AMI
  SINON:
        CLEF = AMI + "-" + UTILISATEUR
  GENERER COUPLE (CLEF; [RESTE DE LA LIGNE])
```

- Pour la première ligne :

A => B, C, D  ("A-B"; "B, C, D") ("A-C"; "B, C, D") ("A-D"; "B, C, D")

Exemple 3 : Amis en commun

→ Pour la seconde ligne:

B => A, C, D, E

("A-B"; "A, C, D, E")

("B-D"; "A, C, D, E")

("B-C"; "A, C, D, E")

("B-E"; "A, C, D, E")

→ Pour la troisième ligne :

C => A, B, D, E

("A-C"; "A, B, D, E")

("C-D"; "A, B, D, E")

("B-C"; "A, B, D, E")

("C-E"; "A, B, D, E")

Exemple 3 : Amis en commun

→ Pour la quatrième ligne:

D => A, B, C, E

("A-D"; "A, B, C, E")

("C-D"; "A, B, C, E")

("B-D"; "A, B, C, E")

("D-E"; "A, B, C, E")

→ Pour la cinquième ligne :

E => B, C, D

("B-E"; "B, C, D")

("D-E"; "B, C, D")

("C-E"; "B, C, D")

Exemple 3 : Amis en commun

→ Après l'opération **Shuffle**, nous avons **neuf** groupes :

("A-B"; "B, C, D")	("B-C"; "A, C, D, E")	("C-E"; "B, C, D")
("A-B"; "A, C, D, E")	("B-C"; "A, B, D, E")	("C-E"; "A, B, D, E")
("A-C"; "A, B, D, E")	("B-D"; "A, B, C, E")	("C-D"; "A, B, D, E")
("A-C"; "B, C, D")	("B-D"; "A, C, D, E")	("C-D"; "A, B, C, E")
("A-D"; "A, B, C, E")	("B-E"; "B, C, D")	("D-E"; "B, C, D")
("A-D"; "B, C, D")	("B-E"; "A, C, D, E")	("D-E"; "A, B, C, E")

Exemple 3 : Amis en commun

→ Données d'entrée de l'opération **REDUCE :**

Pour la clef "A-B": valeurs "A C D E" et "B C D"
Pour la clef "A-C": valeurs "A B D E" et "B C D"
Pour la clef "A-D": valeurs "A B C E" et "B C D"
Pour la clef "B-C": valeurs "A B D E" et "A C D E"
Pour la clef "B-D": valeurs "A B C E" et "A C D E"
Pour la clef "B-E": valeurs "A C D E" et "B C D"
Pour la clef "C-D": valeurs "A B C E" et "A B D E"
Pour la clef "C-E": valeurs "A B D E" et "B C D"
Pour la clef "D-E": valeurs "A B C E" et "B C D"

→ Pour chaque clef «UTILISATEUR1-UTILISATEUR2», on obtient deux listes d'amis.

Exemple 3 : Amis en commun

- Notre operation **REDUCE** va déterminer quels sont les amis qui apparaissent dans les deux listes.
- Renvoie un couple clef/valeur : (“UTILISATEUR-AMI”, AMIS_EN_COMMUN)
- Pseudo-code:

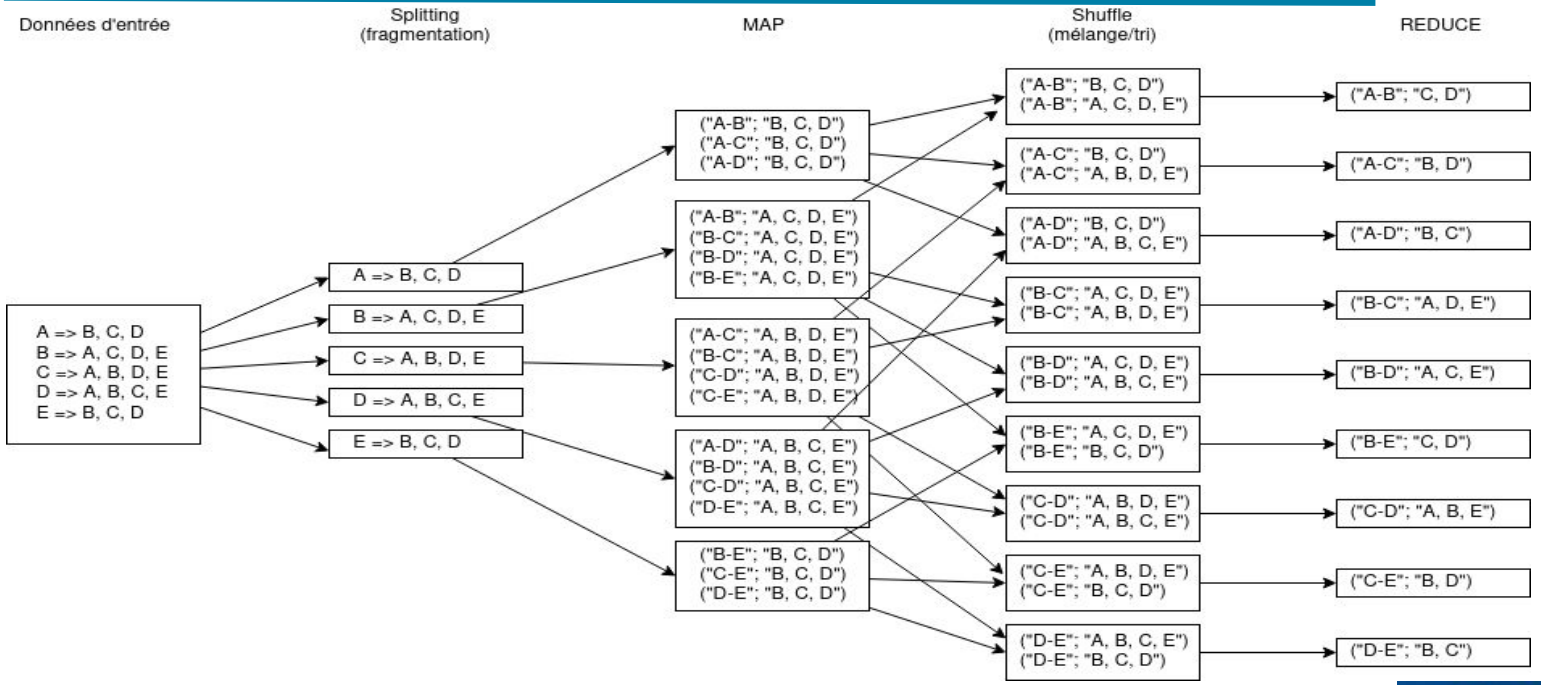
```
LISTE_AMIS_COMMUNS = []  // Liste vide au départ.
SI LONGUEUR(VALEURS) != 2, ALORS:  // Ne devrait pas se produire.
        RENVOYER ERREUR
SINON:
        POUR AMI DANS VALEURS[0], FAIRE:
                SI AMI ÉGALEMENT PRÉSENT DANS VALEURS[1], ALORS:
                // Présent dans les deux listes d'amis, on l'ajoute.
                LISTE_AMIS_COMMUNS += AMI
RENVOYER COUPLE (CLEF; LISTE_AMIS_COMMUNS)
```

Exemple 3 : Amis en commun

➔ Le résultat final :

```
"A-B": "C, D"
"A-C": "B, D"
"A-D": "B, C"
"B-C": "A, D, E"
"B-D": "A, C, E"
"B-E": "C, D"
"C-D": "A, B, E"
"C-E": "B, D"
"D-E": "B, C"
```

Schéma général



Exemple 4 : Parcours de graphe

- On cherche à **déterminer la profondeur maximale** de tous les nœuds d'un graphe à partir d'un nœud de départ.
- Algorithme "**breadth-first search**", parcours en largeur.
- Cet algorithme est assez problématique car les étapes sont **fortement interconnectées** - mais est solvable en faisant plusieurs itérations du même programme Map/Reduce.

Exemple 4 : Parcours de graphe

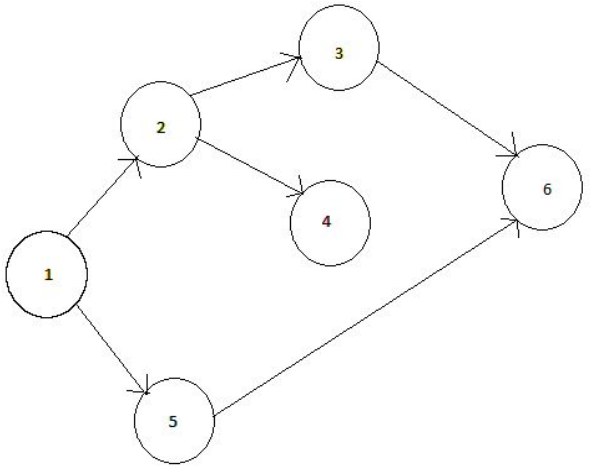
- Le graphe sous forme textuelle :
- Format :

(ID_DU_NEUD; "LIENS|ÉTAT|PROFONDEUR")

- Chaque couple clef-valeur représente un nœud avec ses liens, son état courant (en code couleur) et sa profondeur (distance de la racine).

```
(1; "2,5|GRIS|0")
(2; "3,4|BLANC|-1")
(3; "6|BLANC|-1")
(4; "|BLANC|-1")
(5; "6|BLANC|-1")
(6; "|BLANC|-1")
```

=



Exemple 4 : Parcours de graphe

- 3 États (en couleurs) :
  - **BLANC** = pas encore traité.
  - **GRIS** = à traiter dans cette exécution.
  - **NOIR** = déjà traité (profondeur trouvée).

- Une profondeur de "**-1**" indique qu'elle n'est pas encore calculée.

Exemple 4 : Parcours de graphe

→ Pseudo-code de la fonction **MAP** :

```
SI NODE.COULEUR == "GRIS":
      POUR CHAQUE FILS DANS NODE.CHILDREN:
            FILS.COULEUR = "GRIS"
            FILS.PROFONDEUR = NODE.PROFONDEUR + 1
            RENVOYER (FILS.ID; FILS)
      NODE.COULEUR = "NOIR"
RENVOYER (NODE.ID; NODE)
```

Exemple 4 : Parcours de graphe

→ Pseudocode de la fonction **REDUCE** :

```
MAX_CHILDREN = ""; MAX_PROFONDEUR = -1; MAX_COULEUR = "BLANC";
POUR CHAQUE VALEUR:
        SI VALEUR.CHILDREN.LENGTH() > MAX_CHILDREN.LENGTH():
                MAX_CHILDREN = VALEUR.CHILDREN
        SI VALEUR.PROFONDEUR > MAX_PROFONDEUR:
                MAX_PROFONDEUR = VALEUR.PROFONDEUR
        SI VALEUR.COULEUR > MAX_COULEUR:
                MAX_COULEUR = VALEUR.COULEUR
NODE = NOUVEAU NOEUD
NODE.COULEUR = MAX_COULEUR
NODE.CHILDREN= MAX_CHILDREN
NODE.PROFONDEUR = MAX_PROFONDEUR
RENVOYER COUPLE (CLEF; NODE)
```

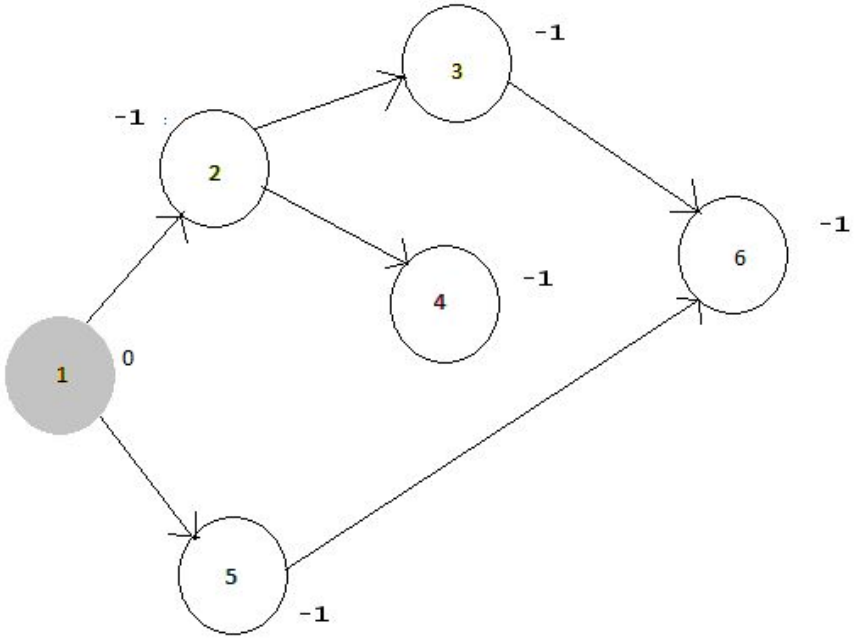
Exemple 4 : Parcours de graphe

- On (ré)exécute le programme Map/Reduce **indéfiniment** jusqu'à ce qu'une **condition d'arrêt** soit atteinte.
- Notre **condition d'arrêt** : l'état couleur de tous les noeuds est **NOIRE** ⇔ tous les nœuds ont été traité.
- À chaque exécution, les données de la sortie de l'exécution précédente deviennent les données d'entrée.

Exemple 4 : Parcours de graphe

➔ Première itération :

(1; "2,5|GRIS|0")
(2; "3,4|BLANC|-1")
(3; "6|BLANC|-1")
(4; "|BLANC|-1")
(5; "6|BLANC|-1")
(6; "|BLANC|-1")



Exemple 4 : Parcours de graphe

Sortie de MAP :

(1; "2,5|NOIR|0")
(2; "|GRIS|1")
(5; "|GRIS|1")
(2; "3,4|BLANC|-1")
(3; "6|BLANC|-1")
(4; "|BLANC|-1")
(5; "6|BLANC|-1")
(6; "|BLANC|-1")

Sortie de Shuffle :

1: ("2,5|NOIR|0")
2: ("3,4|BLANC|-1"), ("|GRIS|1")
3: ("6|BLANC|-1")
4: ("|BLANC|-1")
5: ("6|BLANC|-1"), ("|GRIS|1")
6: ("|BLANC|-1")

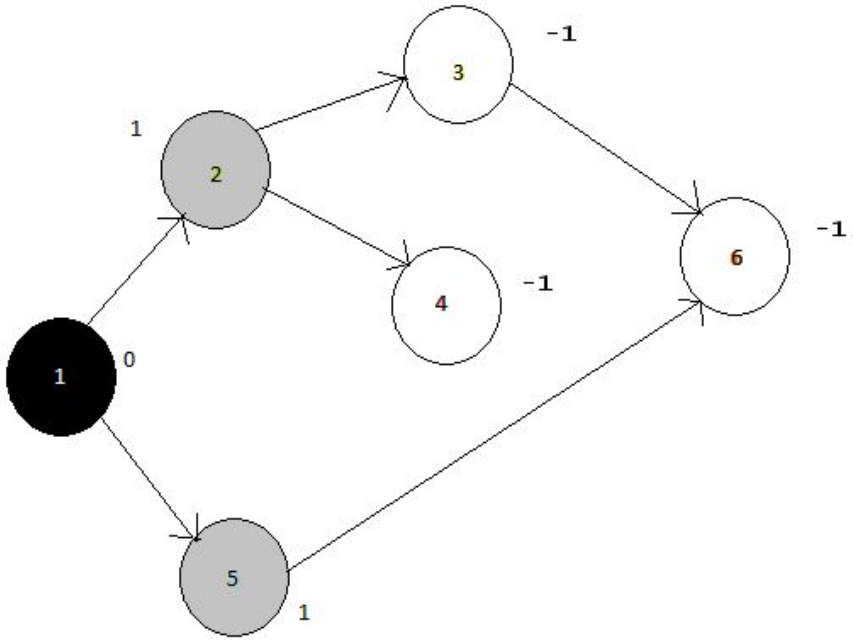
Sortie de REDUCE :

(1; "2,5|NOIR|0")
(2; "3,4|GRIS|1")
(3; "6|BLANC|-1")
(4; "|BLANC|-1")
(5; "6|GRIS|1")
(6; "|BLANC|-1")

Exemple 4 : Parcours de graphe

→ Deuxième itération :

(1; "2,5|NOIR|0")
(2; "3,4|GRIS|1")
(3; "6|BLANC|-1")
(4; "|BLANC|-1")
(5; "6|GRIS|1")
(6; "|BLANC|-1")



Exemple 4 : Parcours de graphe

Sortie de MAP :

(1; "2,5|NOIR|0")
(2; "3,4|NOIR|1")
(3; "|GRIS|2")
(4; "|GRIS|2")
(3; "6|BLANC|-1")
(4; "|BLANC|-1")
(5; "6|NOIR|1")
(6; "|GRIS|2")
(6; "|BLANC|-1")

Sortie de Shuffle :

1: ("2,5|NOIR|0")
2: ("3,4|NOIR|1")
3: ("6|BLANC|-1"), ("|GRIS|2")
4: ("|BLANC|-1"), ("|GRIS|2")
5: ("6|NOIR|1")
6: ("|BLANC|-1"), ("|GRIS|2")

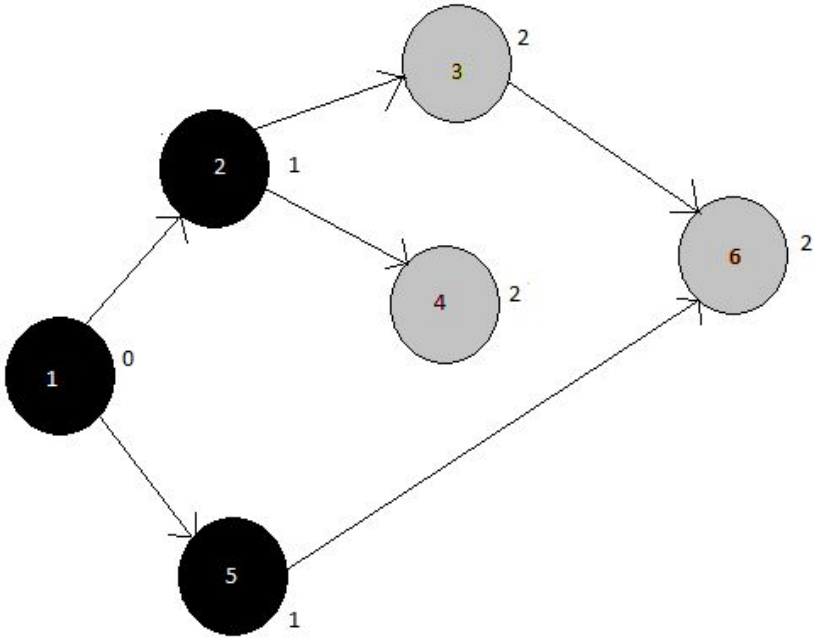
Sortie de REDUCE :

(1; "2,5|NOIR|0")
(2; "3,4|NOIR|1")
(3; "6|GRIS|2")
(4; "|GRIS|2")
(5; "6|NOIR|1")
(6; "|GRIS|2")

Exemple 4 : Parcours de graphe

➔ Troisième itération :

(1; "2,5|NOIR|0")
(2; "3,4|NOIR|1")
(3; "6|GRIS|2")
(4; "|GRIS|2")
(5; "6|NOIR|1")
(6; "|GRIS|2")



Exemple 4 : Parcours de graphe

Sortie de MAP :

(1; "2,5|NOIR|0")
(2; "3,4|NOIR|1")
(3; "6|NOIR|2")
(6; "|GRIS|3")
(4; "|NOIR|2")
(5; "6|NOIR|1")
(6; "|NOIR|2")

Sortie de Shuffle :

1: ("2,5|NOIR|0")
2: ("3,4|NOIR|1")
3: ("6|NOIR|2")
4: ("|NOIR|2")
5: ("6|NOIR|1")
6: ("|NOIR|2"), ("|GRIS|3")

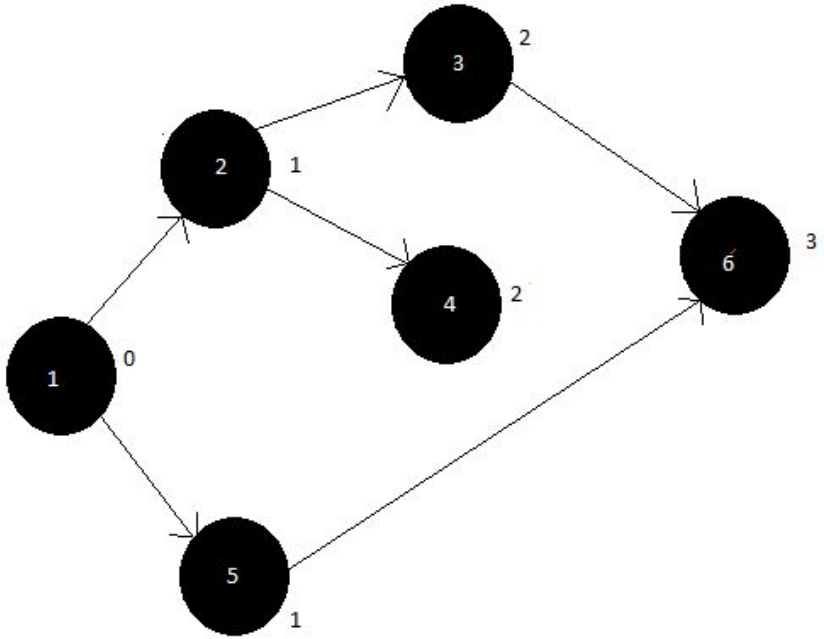
Sortie de REDUCE :

(1; "2,5|NOIR|0")
(2; "3,4|NOIR|1")
(3; "6|NOIR|2")
(4; "|NOIR|2")
(5; "6|NOIR|1")
(6; "|NOIR|3")

Exemple 4 : Parcours de graphe

→ Troisième itération :
(Condition d'arrêt atteinte)

(1; "2,5|NOIR|0")
(2; "3,4|NOIR|1")
(3; "6|NOIR|2")
(4; "|NOIR|2")
(5; "6|NOIR|1")
(6; "|NOIR|3")



Conclusion

- En utilisant le modèle MapReduce, on a ainsi pu créer quatre programmes de quelques lignes de code seulement, qui permettent d'effectuer un traitement assez complexe.
- Il suffit de diviser les données d'entrée et d'implémenter les opérations MAP et REDUCE.
- Mieux encore, notre traitement est **parallélisable** : même avec des centaines de gigaoctets de données, du moment qu'on a assez de machines au sein du cluster Hadoop, le traitement sera effectué rapidement. Pour aller plus vite, il nous suffit de rajouter plus de machines.

Conclusion

- → Le **choix de la clef est critique** afin d'exploiter au maximum le shuffle.
  Un mauvais choix de la clé ou de l'algorithme cause un faible parallélisme.
  Une clef qui produit de **nombreux groupes de taille similaire est préférable** à une clef qui ne crée que quelques groupes de taille disproportionnée.
- → Map/Reduce est **généralement adapté** aux problèmes pour lesquels les approches de type "**diviser pour régner**" ont un sens.
- → Map/Reduce est **moins adapté aux problèmes hautement connectés**.
- → Certains problèmes sont moins faciles à mettre en œuvre (parcours de graphes, etc.)

Conclusion

Applications courantes de Map/Reduce et Hadoop :

- Analyse de logs et données en général
- Traitements sur des volumes de données massifs.
- Exécution de tâches intensives en CPU de manière distribuée (simulations scientifiques, encodage vidéo, entraînement de modèles de machine learning etc.).

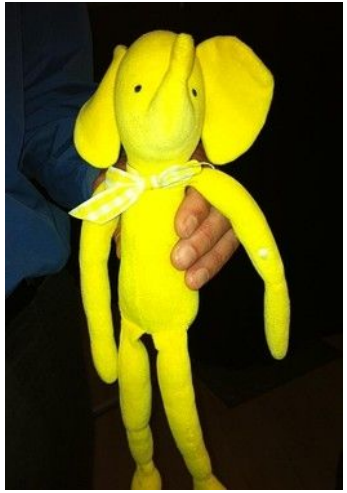
DATUM ACADEMY

HADOOP

Présentation

Apache Hadoop

- Premier **framework Map/Reduce** largement utilisé.
- Origine : Yahoo / Doug Cutting et Mike Cafarella
  2006 (indexation du moteur de recherche)
- En 2011 : entièrement fonctionnel, le cluster Yahoo compte 42 000 nœuds
- “Hadoop” n'est pas un acronyme ; nom de l'éléphant en peluche de l'enfant de Cutting. Cependant, il est parfois appelé : High Availability Distributed Object Oriented Platform



Apache Hadoop

- Projet de la fondation Apache – Open Source.
- **Implémente le modèle Map/Reduce de manière très stricte**.
- Développé en **Java**; l'API est également en Java.
- Très facile à déployer (paquets Linux préconfigurés), configuration très simple également.
- S'occupe de toutes les problématiques liées au calcul distribué.
- Deux composants principaux : **HDFS** (système de fichiers) et **YARN** (moteur d'exécution).

Hadoop

- Moyen principal d'interagir avec le cluster : Outil CLI (interface en ligne de commande) - **hadoop**.
- Il permet de lire/écrire des fichiers, de soumettre des programmes à exécuter, etc.
- Des interfaces graphiques existent également.
- L'utilisation de la commande **hadoop** sera détaillée dans les prochaines sections.

DATUM ACADEMY

HADOOP: HDFS

Présentation - Architecture - Lecture/Écriture - Commandes

Présentation

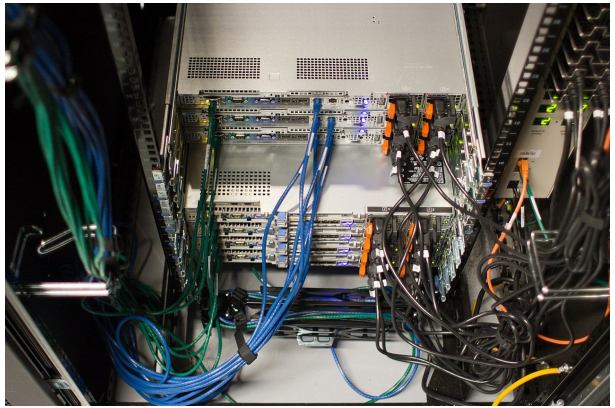
HDFS: Hadoop Distributed FileSystem – le système de fichier distribué de Hadoop.

- ➔ Inspiré par le papier de recherche : "The Google File System", Google, 2003.
- ➔ Requis pour **l'accès concurrent** aux données par les nœuds du cluster.
- ➔ **Remarque:** Hadoop peut également communiquer directement avec une base de données. Ce mode d'intégration passe par le biais de "ponts d'interconnexion" (bridges).

Présentation

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** : les données sont répliquées sur des racks différents - si un rack de serveurs entier tombe en panne, aucune donnée n'est perdue. HDFS peut aussi **optimiser les transferts** de données pour limiter la « distance » à parcourir pour la réplication.



Présentation

Sur **HDFS** les données sont :

- ➔ Structurées comme dans un système de fichiers Unix traditionnel (/ comme racine).
- ➔ Découpées en « blocks » de **128 MB** (ou 64 MB sur anciennes versions) **par default** (configurable).

Architecture

Deux principaux serveurs (des daemons) :

- → Le **NameNode** : **stocke les informations relatives aux fichiers** (comme: les noms des fichiers, leur localisation sur HDFS, etc.). Il y a **un seul NameNode active dans tout le cluster** Hadoop.
- → Le **DataNode** : **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…

Architecture

- L'état du **NameNode** est persisté sur disque avec 2 principaux types de fichier :
- **fsimage :** Est une sauvegarde de l'état du Système de Fichier à un moment donné.
- **edit-logs :** Contient tous les changements (création, modification, suppression) effectués sur les fichiers après la création du dernier fichier fsimage.

Architecture

Quand le **NameNode** se lance :

- ➔ Il charge le dernier **fsimage en mémoire**.
- ➔ Applique les changements trouvées dans les edit-logs sur son **fsimage**.
- ➔ Les prochaines changements fait sur HDFS seront **persistés** dans les edit-logs et appliqués sur le **fsimage** en mémoire.

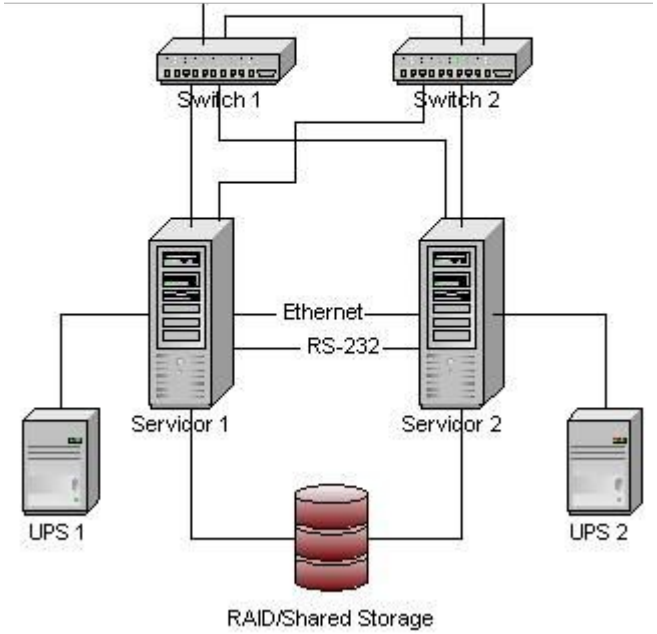
Architecture

Il y a également 2 daemons supplémentaires :

- Le **SecondaryNameNode**: Aide le NameNode à produire des "checkpoints". Il applique les changements enregistrés dans les edit-logs sur le dernier fsimage et persiste le nouveau fsimage mis à jour sur disque. Mais il n'est pas un NameNode de sauvegarde.
- Le **JournalNode**: Peut être utilisé lors d'une configuration de haute disponibilite pour partager efficacement les edit-logs et les métadonnées entre deux NameNodes.

Architecture

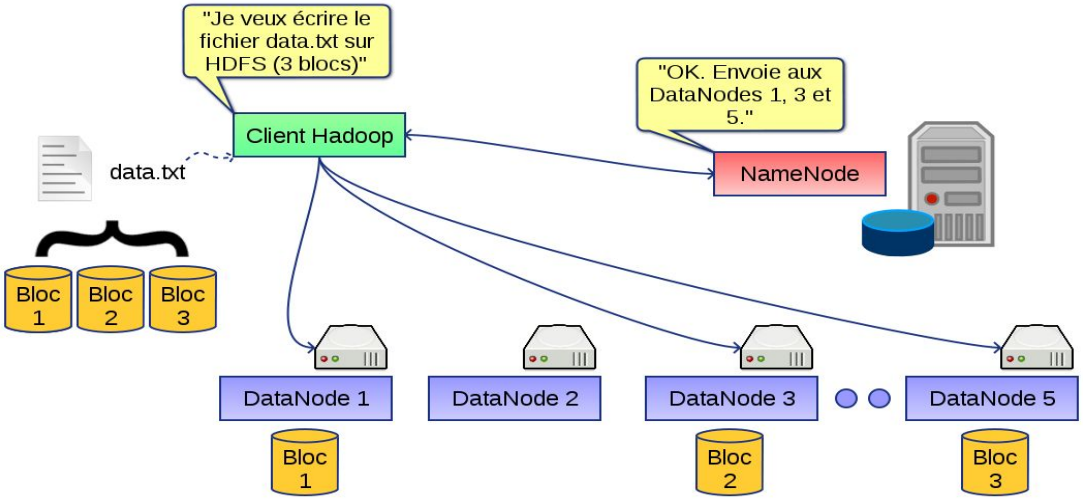
- Architecture haute disponibilité



Nuno
Tavares

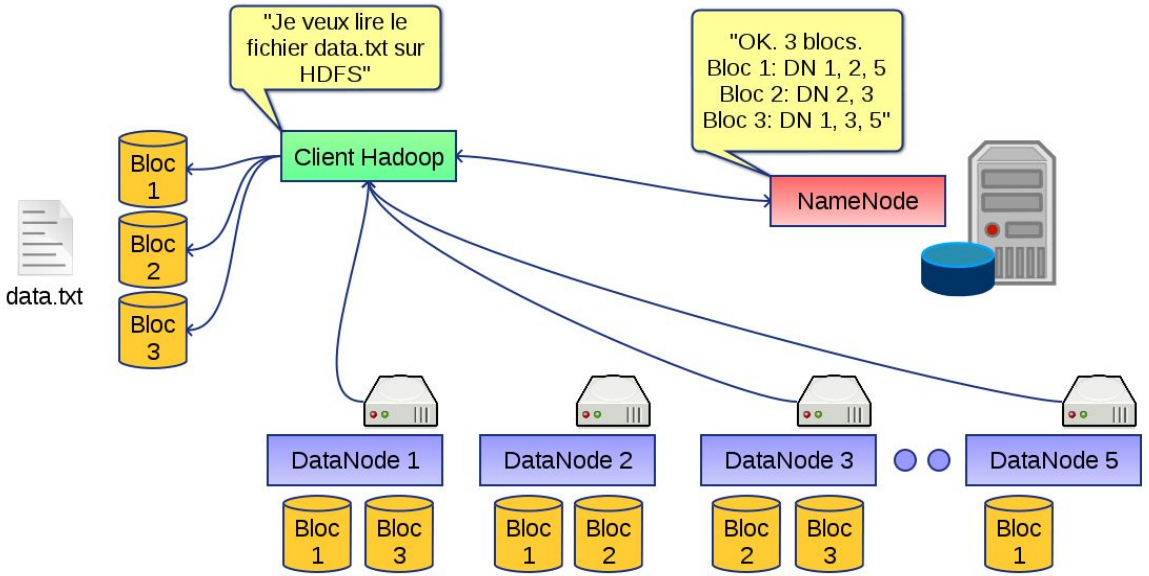
ÉCRITURE HDFS

- 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 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

- La commande "**hadoop**" avec l'option "**fs**" permet de stocker ou extraire des fichiers de HDFS.
- Usage :

hadoop fs -COMMANDE_UNIX ARGUMENTS

- La plupart des **commandes du système de fichiers unix** sont supportées : -mkdir, -ls, etc.
- Peut être exécuté sur n'importe quel nœud du cluster.

- Il existe également une commande alternative (mois générique que "hadoop fs"):

hdfs dfs -COMMANDE_UNIX ARGUMENTS

La commande hadoop fs

➔ Exemples :

```
hadoop fs -put livre.txt /user/john
```

➔ Pour stocker le fichier livre.txt sur HDFS dans le répertoire /user/john.

```
hadoop fs -ls /user
```

➔ Pour afficher le contenu du répertoire HDFS /user

```
hadoop fs -get /results/part-r-0001
```

➔ Pour récupérer le fichier /results/part-r-0001 de HDFS.

```
hadoop fs -rm -r /results
```

➔ Pour supprimer le dossier /results:

La commande hadoop fs

```
hdoop@hp:~$
hdoop@hp:~$ hadoop fs -ls /user/wolf/
hdoop@hp:~$ echo "Hello World" > test.txt
hdoop@hp:~$ cat test.txt
Hello World
hdoop@hp:~$ hadoop fs -put test.txt /user/wolf/
hdoop@hp:~$ hadoop fs -ls /user/wolf/
Found 1 items
-rw-r--r--   3 hdoop supergroup         12 2021-08-30 10:31 /user/wolf/test.txt
hdoop@hp:~$ hadoop fs -cat /user/wolf/test.txt
Hello World
hdoop@hp:~$ hadoop fs -mv /user/wolf/test.txt /user/wolf/renamed.txt
hdoop@hp:~$ hadoop fs -ls /user/wolf/
Found 1 items
-rw-r--r--   3 hdoop supergroup         12 2021-08-30 10:31 /user/wolf/renamed.txt
hdoop@hp:~$ hadoop fs -rm /user/wolf/renamed.txt
Deleted /user/wolf/renamed.txt
hdoop@hp:~$ hadoop fs -ls /user/wolf/
hdoop@hp:~$
```

Remarques

- HDFS peut être utilisé comme un système de stockage scalable indépendant de Hadoop.
- Hadoop peut également être utilisé sans HDFS.
- HDFS est optimisé pour des **lectures concurrentes**: écrire de manière concurrente est nettement moins performant.
- Il existe également des alternatives à HDFS : bases de données, protocoles de réseau, alternatives propriétaires (par exemple: MongoDB, Amazon S3, FTP, NFS, MapR-FS)

DATUM ACADEMY

HADOOP: YARN

Présentation - Architecture de Yarn - Lecture/Écriture - Commandes

Présentation

- **YARN : Yet Another Resource Negotiator** (Aussi appelé "MRv2")
- Est le **système de gestion des ressources** et le **moteur d'exécution** de Hadoop.
- Intègre à partir de la version 2.x de Hadoop

Présentation

Deux serveurs principaux (daemons) :

- ➔ **ResourceManager :** un par cluster.
- ➔ **NodeManager :** un par nœud.

Yarn – ResourceManager

- Le démon principal est le **ResourceManager**.
- Il est **unique** sur le cluster.
- Il gère le concept de **« ressources » disponibles** du cluster: machines, slots d'exécution, mémoire/CPU etc.
- Il reçoit les soumissions des programmes (applications) et commence leur exécution.

Yarn – ResourceManager

Le **ResourceManager** est divisé de deux **principaux composants** :

- Le **Scheduler** :
  Planifie et distribue des tâches aux différents slots d'exécution du cluster en fonction des ressources disponibles.
- **L'ApplicationsManager** :
  Reçoit les programmes à exécuter sur le cluster venant des clients.
  Quand un programme est reçu, il lance une première tache sur le cluster où la classe driver (le main du programme) est exécutée.
  Cette première tâche est spéciale et s'appelle **ApplicationMaster**.
  Elle devient le « chef d'orchestre » de l'exécution du programme et soumettra par elle-même de nouvelles tâches à effectuer au **Scheduler**.

Yarn – NodeManager

- Le second démon, **NodeManager**, tourne sur chaque machine du cluster.
- Il maintient **les slots d'exécution locaux** (conteneurs d'exécution).
- Il reçoit et exécute des tâches dans ces conteneurs attribués aux applications par le **Scheduler**.
- Ces tâches à exécuter peuvent être de simples opérations Map ou Reduce, mais aussi le cœur de l'application lui-même - **l'ApplicationMaster.**
- Le **NodeManager** envoie à intervalles réguliers des **heartbeats** au **ResourceManager**; ils contiennent des métriques de ressources disponibles, des informations sur les tâches en cours et permettent de détecter les pannes.

Yarn – Soumission d'un programme

Les étapes d'exécution d'un programme via **YARN** :

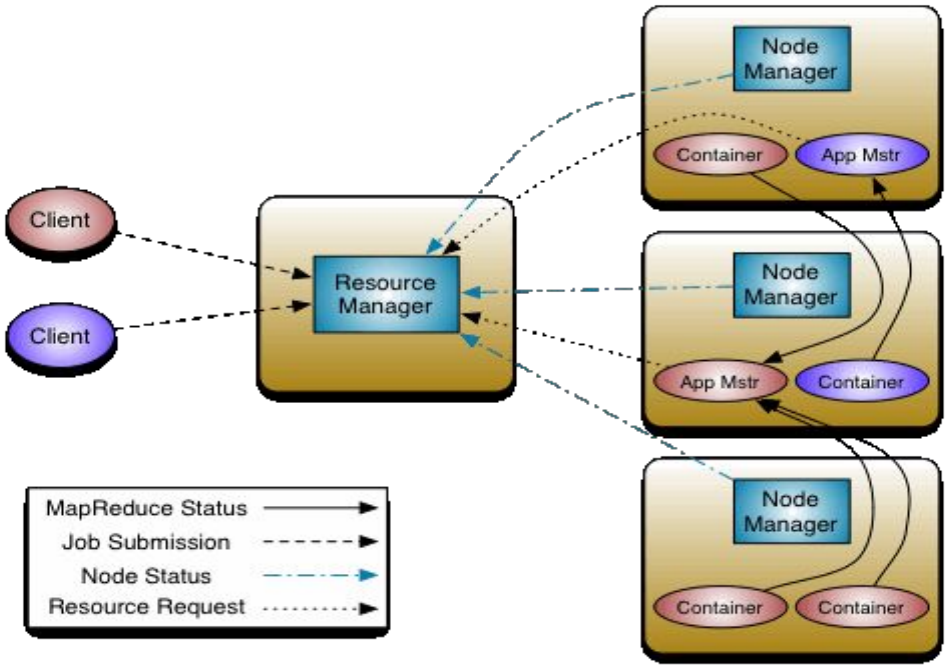
- ➔ **Un client** se connecte au **ResourceManager**, et annonce l'exécution du programme / fournit son code (jar).
- ➔ **L'ApplicationsManager** du **ResourceManager** prépare un container libre pour y exécuter **l'ApplicationMaster** du programme (son main).
- ➔ **L'ApplicationMaster** du programme est lancée; il s'enregistre immédiatement auprès du ResourceManager et effectue ses **demandes de ressources** (containers pour exécuter tâches Map et Reduce, par exemple).

  Il contacte alors directement les **NodeManager** correspondants aux containers qui lui ont été attribués pour leur soumettre les tâches.

Yarn – Soumission d'un programme

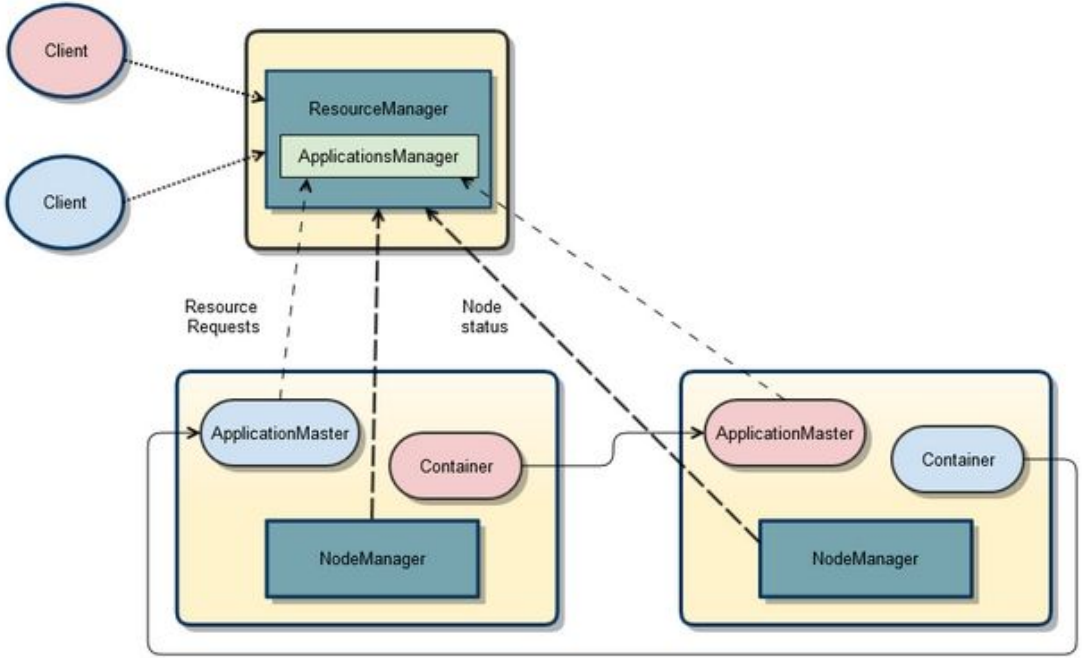
- Pendant l'exécution des différentes tâches, les containers (par le biais des démons NodeManager) mettent à **jour l'ApplicationMaster** sur leur statut continuellement. En cas de problème, celui-ci peut demander de nouveaux containers au ResourceManager pour ré-exécuter une tâche.
- L'utilisateur peut par ailleurs obtenir une mise à jour sur le statut de l'exécution soit en contactant le **ResourceManager**, soit en contactant **l'ApplicationMaster** directement.
- Une fois toutes les tâches exécutées, **l'ApplicationMaster** signale le **ResourceManager** et s'arrête.
- Le **ResourceManager** libère alors le container occupé restant, qui exécutait le code de **l'ApplicationMaster**.

Architecture de Yarn



Documentation Hadoop

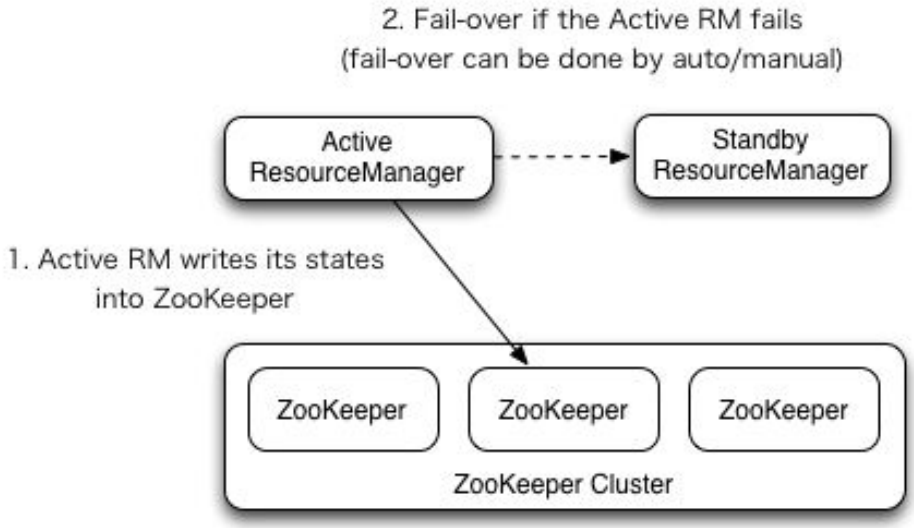
Yarn – Soumission d'un programme



Documentation Hadoop

Yarn – Remarques

Le **ResourceManager** est un « **point unique de défaillance** » : il est nécessaire d'assurer sa **haute disponibilité** via des mécanismes classiques de failover. (Active/Standby, souvent réalisé avec **Zookeeper**)



Documentation Hadoop

Yarn – Remarques

- Quand **YARN** est utilisé avec **HDFS**, le **ResourceManager** essaie de lancer les tâches sur les nœuds qui possèdent les données requises. Cela accélère l'exécution en minimisant le transfert des données entre les nœuds du cluster.
- La communication directe entre l'**ApplicationMaster** et **les NodeManagers** est un point de différence notable avec la version 1 d'Hadoop.

Interfaces utilisateur

Hadoop est essentiellement piloté par des outils consoles (CLI), mais quelques interfaces utilisateurs sont mises à disposition de l'utilisateur :

- ➔ **NameNode** expose une interface web (via un serveur web intégré) qui permet de parcourir les fichiers et répertoires stockés sur HDFS.
- ➔ **RessourcesManager** expose une interface web similaire, qui permet de voir les ressources disponibles du cluster et les applications en cours d'exécution.

Par ailleurs, il y a aussi des interfaces utilisateur tierces qui ont été développées pour simplifier l'utilisation de Hadoop, par exemple le projet **Hue** (Open Source)

DATUM ACADEMY

Programmation Hadoop

API Hadoop: classes Driver, Mapper, Reducer

Plan

- API Java de Hadoop MapReduce
- API Java de HDFS

- Concepts Hadoop avancés (Propriétés de configuration / Compteurs)
- L'environnement de développement

Programmation Hadoop

Comme indiqué précédemment, Hadoop est développé en **Java**.

- ➔ Les tâches **MapReduce** sont donc implémentables par le biais d'interfaces Java.
- ➔ Il existe cependant un outil très simple permettant d'implémenter ses tâches dans n'importe quel langage (**Hadoop Streaming**).

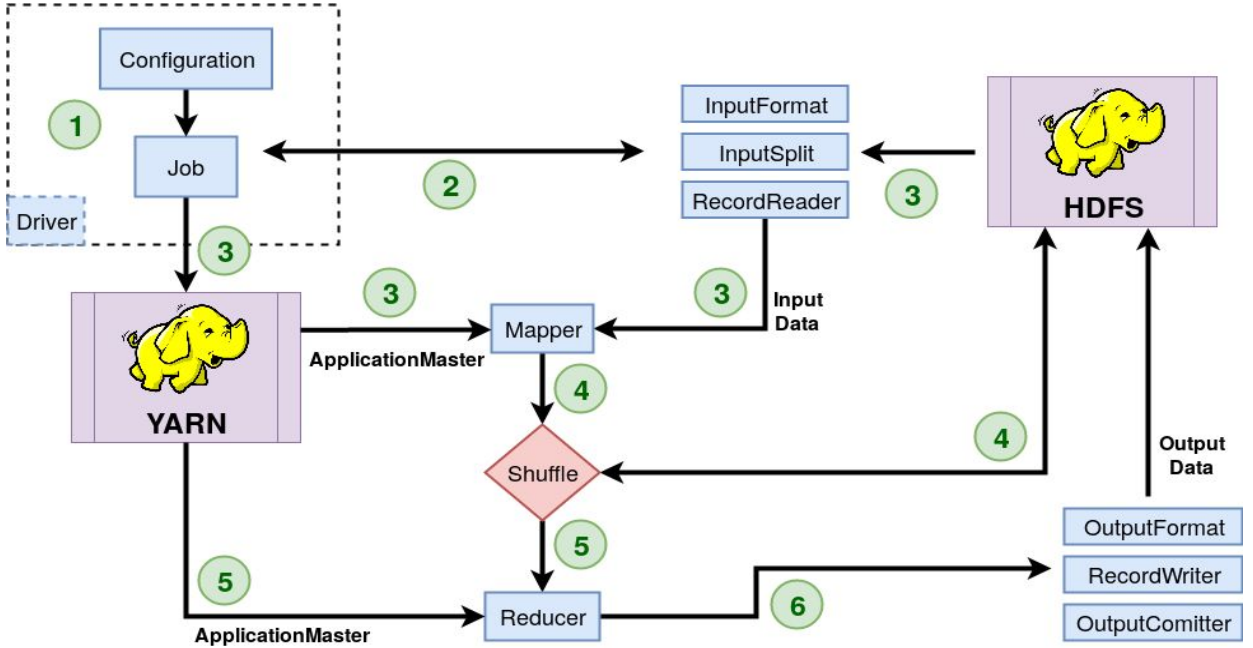
Programmation Hadoop

Un programme Hadoop se compile au sein d'un .jar.

Pour développer un programme Hadoop, on va créer **trois classes** distinctes:

- → Une classe dite **Driver** qui **contient la fonction main** du programme. Cette classe se chargera d'informer Hadoop des **types** de données **clé/valeur** utilisées, des classes se chargeant des opérations MAP et REDUCE, et des fichiers HDFS à utiliser pour les entrées/sorties.
- → Une classe **MAPPER** (qui effectuera l'opération MAP).
- → Une classe **REDUCER** (qui effectuera l'opération REDUCE).

- Vue d'ensemble:



Programmation Hadoop - Environnement de développement

- Pour le développement, un **IDE Java** est fortement recommandé.
- Par exemple: IntelliJ, Eclipse, Netbeans, emacs… etc.
- Pour la gestion des dépendances, on peut utiliser **Gradle, Maven** ou le faire manuellement.

Programmation Hadoop - Environnement de développement

Versions Java supportées :

- ➔ Hadoop 2.X – Java 7 et 8
- ➔ Hadoop 3.X – Java 8 et 11

Programmation Hadoop - Environnement de développement

Il y a 2 versions de l'API Hadoop :

- ➔ **La nouvelle API :** import org.apache.hadoop.**mapreduce**.Mapper
- ➔ **L'ancienne API :** import org.apache.hadoop.**mapred**.Mapper

L'ancienne API est beaucoup moins lisible et produit des messages d'alerte.

Programmation Hadoop – Classe Driver

La classe **Driver** contient le **main (point d'entrée)** de notre programme.

- Elle gère la configuration, met en place une ou plusieurs tâches MapReduce (**Job's**) et les lance sur le cluster.
- Elle peut exécuter le programme Hadoop en arrière-plan, ou de façon bloquante pour le client.

Programmation Hadoop – Classe Driver

La fonction **main** devrait au moins effectuer les opérations suivantes :

- ➔ Créer un objet **Configuration** de Hadoop.
- ➔ Créer un objet **Job** de Hadoop, qui représente une tâche MapReduce.
- ➔ Informer Hadoop sur les classes **Driver**, **Mapper** et **Reducer**.
- ➔ Informer Hadoop sur les **types de données** utilisés pour les couples (clef; valeur) entre MAP et REDUCE et à l'issue de REDUCE.
- ➔ Préciser les données d'entrée et la destination pour les résultats.
- ➔ Lancer la tâche.

Programmation Hadoop – Classe Driver

Déclaration de la méthode **main** dans la classe Driver :

```
public static void main(String[] args) throws Exception
```

- → On se sert de **args** pour récupérer les arguments de la ligne de commande.
- → Il est recommandé de laisser Hadoop récupérer ses paramètres depuis args.

Programmation Hadoop – Classe Driver

Création d'un objet de configuration :

- **import org.apache.hadoop.conf.Configuration;**

```
Configuration conf = new Configuration();
```

- Le constructeur par défaut trouvera automatiquement le cluster Hadoop local et obtiendra sa configuration.
- Cet objet peut également être ajusté pour se connecter à un cluster différent.

Programmation Hadoop – Classe Driver

Passage des arguments de la ligne de commande à Hadoop :

➔ **import org.apache.hadoop.util.GenericOptionsParser;**

```
String[] ourArgs = new GenericOptionsParser(conf, args).getRemainingArgs();
```

➔ **GenericOptionsParser** permet à Hadoop d'obtenir ses propres arguments, comme :

- -D key=value
- -conf <CONF_FILE>
- -fs <local|NAMENODE_LOCATION>
- etc....

➔ **getRemainingArgs()** permet de récupérer les arguments non exploités par Hadoop.

Programmation Hadoop – Classe Driver

Création d'un objet Hadoop Job :

➔ **import org.apache.hadoop.mapreduce.Job;**

```
Job job = Job.getInstance(conf, "Compteur de mots v1.0");
```

➔ Applique le patron de conception **fabrique** - utilise **getInstance** à la place d'un constructeur.

➔ Accepte deux arguments – l'objet de **Configuration** et une **description textuelle**.

➔ Permet de **configurer notre tâche** MapReduce et lancer son exécution.

Programmation Hadoop – Classe Driver

→ Configuration des classes **Driver**, **Mapper** et **Reducer** avec l'objet **Job** :

```
job.setJarByClass(Driver.class);
job.setMapperClass(Map.class);
job.setReducerClass(Reduce.class);
```

Programmation Hadoop – Classe Driver

Configuration des **types** de données utilisés pour les **couples (clef; valeur)** :

- **import org.apache.hadoop.io.IntWritable;**
- **import org.apache.hadoop.io.Text;**

```
job.setOutputKeyClass(Text.class);
job.setOutputValueClass(IntWritable.class);
```

- Règle les types de clés et valeurs utilisés **entre Map et Reduce et en sortie de Reduce**.
- On ne doit pas utiliser les classiques Java comme Integer et String.
- Nécessite de types **Writable** - types **sérialisables** spéciales d'Hadoop.

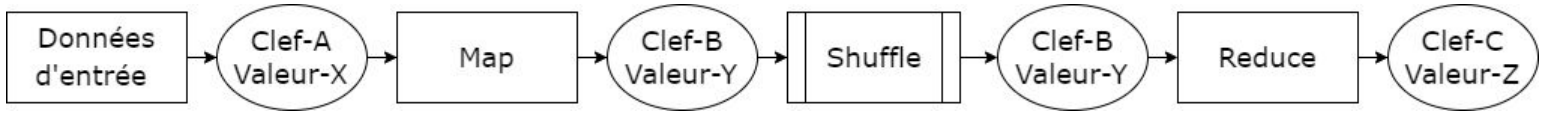
Programmation Hadoop – Classe Driver

Types standard **Writable** d'Hadoop: (Il en existe beaucoup d'autres dans **org.apache.hadoop.io.***).

- **Text** (à la place de String)
- **IntWritable** (à la place de Integer)
- **LongWritable** (à la place de Long)
- **FloatWritable** (à la place de Float)
- **DoubleWritable** (à la place de Double)

Programmation Hadoop – Classe Driver

→ Il y a six **types** de données utilisés pour les **couples (clef; valeur)** :



Programmation Hadoop – Classe Driver

Si les **types de clefs et valeurs** en sortie de **Map** sont **égaux à** ceux utilisés en sortie de **Reduce** :

→ Exemple : Map (Text; IntWritable) et Reduce (Text; IntWritable)

```
job.setOutputKeyClass(Text.class);
job.setOutputValueClass(IntWritable.class);
```

Si les **types de clefs et valeurs** en sortie de **Map** sont **différent** de ceux utilisés en sortie de **Reduce** :

→ Exemple : Map (Text; IntWritable) et Reduce (Text; Text)

```
job.setMapOutputKeyClass(Text.class);
job.setMapOutputValueClass(IntWritable.class);
job.setOutputKeyClass(Text.class);
job.setOutputValueClass(Text.class);
```

Programmation Hadoop – Classe Driver

Indication des données d'entrée et de sortie sur HDFS :

- **import org.apache.hadoop.mapreduce.lib.input.FileInputFormat**
- **import org.apache.hadoop.mapreduce.lib.input.FileOutputFormat**
- **import org.apache.hadoop.fs.Path**

```
FileInputFormat.addInputPath(job, new Path("/users/john/poeme.txt"));
FileOutputFormat.setOutputPath(job, new Path("/users/john/output"));
```

- Ces méthodes mettent à jour l'objet Job.

Programmation Hadoop – Classe Driver

- **Path** précise le chemin vers les fichiers ou les répertoires.
  (Exemple : “hdfs://namenode:9000/user/john/livres/*”, “/user/john/livres/*”, “file:///home/tom/*”).
- Si le **Schema** (comme : “hdfs://”, “file:///”) n’est pas fourni, Hadoop utilise **“hdfs://”** par défaut.
- Nous pouvons spécifier autant d'entrées **addInputPath** que nous le souhaitons.
- Il est possible d’utiliser la méthode **setInputPaths** pour définir toutes les entrées en une fois.
- La sortie doit être unique - un (nouveau) répertoire.

Programmation Hadoop – Classe Driver

→ Exemples:

```
FileInputFormat.setInputPaths(
   job,
   new Path("hdfs:///poeme.txt"),
   new Path("/poeme2.txt")
);
FileInputFormat.addInputPath(job, new Path("file:///opt/bigdata/ref.txt"));
FileInputFormat.addInputPath(job, new Path("/user/john/input_poeme"));
FileOutputFormat.setOutputPath(job, new Path("/users/john/output"));
```

Programmation Hadoop – Classe Driver

- Exécution de la tâche (Approche 1/2) :

```
job.waitForCompletion(true)
```

- Mode **synchrone** : attend la fin de l'exécution.
- L'argument contrôle la verbosité. S'il est **true**, Hadoop affichera plus de logs sur la sortie standard.
- Renvoie **true** en cas de succès et **false** en cas d'échec.

Programmation Hadoop – Classe Driver

- Exécution de la tâche (Approche 2/2) :

```
job.submit();
```

- Mode **asynchrone :** soumet la tâche au cluster, puis retourne immédiatement.
- Il y a de nombreuses méthodes permettent d'obtenir des informations sur l'exécution ou d'influencer l'exécution :
  - getJobState()
  - getStatus()
  - killJob()
  - getTaskCompletionEvents()
  - … et d'autres ...
- Les deux approches soulèvent des exceptions en cas de problème.

Programmation Hadoop – Classe Driver

→ Le code complet de notre classe Driver (1/3):

```
package org.mbds.hadoop.wordcount;

import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.io.IntWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Job;
import org.apache.hadoop.mapreduce.lib.input.FileInputFormat;
import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;
import org.apache.hadoop.util.GenericOptionsParser;
```

Programmation Hadoop – Classe Driver

➔ Le code complet de notre classe Driver (2/3):

```
public class WCount {
   public static void main(String[] args) throws Exception {
      // Crée un objet de configuration Hadoop.
      Configuration conf = new Configuration();
      // Récupère les arguments restants dans ourArgs.
      String[] ourArgs = new GenericOptionsParser(conf, args).getRemainingArgs();
      // Crée un objet Job. On fourni la configuration et une description de la tâche.
      Job job = Job.getInstance(conf, "Compteur de mots v1.0");
      // Définit les classes Driver, Mappper et Reducer.
      job.setJarByClass(WCount.class);
      job.setMapperClass(WCountMap.class);
      job.setReducerClass(WCountReduce.class);
```

Programmation Hadoop – Classe Driver

→ Le code complet de notre classe Driver (3/3):

```
        // Définit les types des clefs/valeurs de notre programme Hadoop.
        job.setOutputKeyClass(Text.class);
        job.setOutputValueClass(IntWritable.class);
        // Définit les fichiers d'entrée du programme et le répertoire des résultats.
        FileInputFormat.addInputPath(job, new Path(ourArgs[0]));
        FileOutputFormat.setOutputPath(job, new Path(ourArgs[1]));
        // Lance la tâche Hadoop.
        System.exit(job.waitForCompletion(true) ? 0 : 1);
    }
}
```

Programmation Hadoop – Classe Mapper

La classe **Mapper** est chargée de l'opération Map. Elle doit :

- ➔ Étendre la classe Hadoop **org.apache.hadoop.mapreduce.Mapper**.
- ➔ Redéfinir la méthode **map** qui effectue la tâche Map.

Rappel - Classes génériques Java

- Souvent utilisé dans des types tels que ArrayList :

```
ArrayList<Integer> list = new ArrayList<>();
```

- Exemple :

```
public class Variable<T> {
        private T value;

        public void setValue(T t) {
                this.value = t;
        }

        public T getValue() {
                return(this.value);
        }
}
```

```
public static void main(String[] args) {

        Variable<Integer> varInt = new Variable<Integer>();
        Variable<String> varStr = new Variable<String>();

        varInt.setValue(42);
        varStr.setValue("Hello");

        System.out.println("Int: " + varInt.getValue());
        System.out.println("String: " + varStr.getValue());
}
```

Programmation Hadoop – Classe Mapper

La classe Mapper est une classe générique qui se paramétrise avec quatre types :

- ➔ Le type pour la **clef d'entrée**
- ➔ Le type pour la **valeur d'entrée**
- ➔ Le type pour la **clef de sortie**
- ➔ Le type pour la **valeur de sortie**

Programmation Hadoop – Classe Mapper

- Les couples clef/valeur d'entrée pour Map sont construits à partir de **fragments** de données d'entrée.
- Par défaut, pour les fichiers texte sur HDFS :
  - L'opération Map découpe le fragment **par ligne**.
  - Le **numéro** de la ligne est passé comme la **clef** d'entrée (type : **LongWritable**).
  - Le **contenu** de la ligne est passé comme la **valeur** d'entrée (type : **Text**)
- Si le numéro de ligne ne vous intéresse pas, vous pouvez spécifier Object comme type de clé d'entrée.
- Le comportement par défaut peut être ajusté par les classes **InputFormat** / **RecordReader**

Programmation Hadoop – Classe Mapper

- Exemple de déclaration d'une classe Mapper :

```
public class Map extends Mapper<Object, Text, Text, IntWritable>
```

- Utilisant les types :
  - **Object** pour la **clef d'entrée**
  - **Text** pour la **valeur d'entrée**
  - **Text** pour la **clef de sortie**
  - **IntWritable** pour la **valeur de sortie**

Programmation Hadoop – Classe Mapper

La **méthode map** prends **trois arguments** :

- ➔ La clef d'entrée.
- ➔ La valeur d'entrée.
- ➔ Un object Context qui est utilisé pour interagir avec Hadoop et retourner les couples (clef; valeur) résultant de l'opération Map.

Exemple de déclaration de la méthode **map** :

```
protected void map(Object key, Text value, Context context)
    throws IOException, InterruptedException
```

Programmation Hadoop – Classe Mapper

- La méthode **map** implémente la tâche Map.
- La méthode **write** de l'objet **Context** permet de renvoyer un couple (clef; valeur).
- **write** peut être appelée plusieurs fois pour renvoyer plusieurs couples clef/valeur.

```
context.write(new Text("ciel"), new IntWritable(1));
```

Programmation Hadoop – Classe Mapper

Important:

- Il faut que les types génériques de la ***clé*** et ***valeur*** **d'entrée** de notre classe Mapper correspondent avec les arguments de la méthode map :

```
public class Map extends Mapper<LongWritable, Text, Text, IntWritable> {
    protected void map(LongWritable lineno, Text line, Context context)
    throws IOException, InterruptedException {
```

- Il faut également que les types de clef et la valeur renvoyées par la méthode map correspondent aux types génériques de la ***clé*** et ***valeur*** **de sortie** de notre classe Mapper :

```
public class Map extends Mapper<LongWritable, Text, Text, IntWritable> {
    protected void map(LongWritable lineno, Text line, Context context)
    throws IOException, InterruptedException {
        context.write(new Text("ciel"), new IntWritable(5))
```

Programmation Hadoop – Classe Mapper

Important:

→ Il faut aussi que les types de ***clé*** et ***valeur*** **de sortie** de notre classe Mapper correspondent avec les types indiqués dans notre classe Driver :

```
job.setMapOutputKeyClass(Text.class);
job.setMapOutputValueClass(IntWritable.class);
```

=> dans le main du Driver

```
public class Map extends Mapper<LongWritable, Text, Text, IntWritable> {
    protected void map(LongWritable lineno, Text line, Context context)
    throws IOException, InterruptedException {
        context.write(new Text("ciel"), new IntWritable(5))
```

Programmation Hadoop – Classe Mapper

➔ Le code complet de notre classe Mapper (1/2):

```
package org.mbds.hadoop.wordcount;

import org.apache.hadoop.mapreduce.Job;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.io.IntWritable;
import java.util.StringTokenizer;
import org.apache.hadoop.mapreduce.Mapper;
import java.io.IOException;
```

Programmation Hadoop – Classe Mapper

→ Le code complet de notre classe Mapper (2/2):

```
public class WCountMap extends Mapper<Object, Text, Text, IntWritable>
{
   protected void map(Object offset, Text value, Context context)
   throws IOException, InterruptedException {
      // Parcourt chacun des mots de la ligne.
      StringTokenizer tok = new StringTokenizer(value.toString(), " ");
      while(tok.hasMoreTokens() {
         Text word = new Text(tok.nextToken());
         // Renvoie notre couple (clef; valeur)
         context.write(word, new IntWritable(1));
      }
   }
}
```

Programmation Hadoop – Classe Reducer

La classe **Reducer** est chargée de l'opération Reduce. Elle doit :

- ➔ Étendre la classe Hadoop **org.apache.hadoop.mapreduce.Reducer**.
- ➔ Redéfinir la méthode **reduce** qui effectue la tâche Reduce.

Programmation Hadoop – Classe Reducer

La clase Reducer est aussi une classe générique qui se paramétrise avec quatre types :

- ➔ Le type pour la **clef d'entrée**
- ➔ Le type pour la **valeur d'entrée**
- ➔ Le type pour la **clef de sortie**
- ➔ Le type pour la **valeur de sortie**

Programmation Hadoop – Classe Reducer

- Exemple de déclaration d'une classe Reducer :

```
public class Reduce extends Reducer<Text, IntWritable, Text, Text>
```

- Utilisant les types :
  - **Text** pour la **clef d'entrée**
  - **IntWritable** pour la **valeur d'entrée**
  - **Text** pour la **clef de sortie**
  - **Text** pour la **valeur de sortie**

Programmation Hadoop – Classe Reducer

La méthode **reduce** prends aussi **trois** arguments :

- ➔ La clef d'entrée.
- ➔ Un objet itérable contenant les valeurs associées à la clef en entrée.
- ➔ Un object Context qui est utilisé pour interagir avec Hadoop et retourner les couples (clef; valeur) résultant de l'opération Reduce.

Exemple de déclaration de la méthode **reduce** :

```
protected void reduce(Text key, Iterable<IntWritable> values, Context context)
    throws IOException, InterruptedException
```

Programmation Hadoop – Classe Reducer

- Les couples clef/valeur d'entrée pour Reduce sont construits à partir de **groupes** des couples clef/valeur issus de l'opération **Shuffle**.
- Pour renvoyer un couple (clef; valeur) en résultat Reduce utilise le même principe que dans Map :

```
context.write(new Text("ciel"), new Text("5 occurrences"));
```

Programmation Hadoop – Classe Reducer

Par défaut, la **sortie** de Reduce est persisté dans un **dossier sur HDFS** :

- ➔ Un fichier texte par exécution de Reduce (par groupe de clef commune issue de Shuffle).
- ➔ Un couple clef/valeur par ligne, avec le caractère de tabulation pour séparer la clé de la valeur.
- ➔ Le comportement par défaut d'écriture final peut être ajusté par les classes OutputFormat / RecordWriter.

Exemple d'un fichier de résultat sur HDFS nommé '/resulat/part-r-00000' :

```
Clef__1  Valeur__1
Clef__2  Valeur__2
Clef__3  Valeur__3
Clef__4  Valeur__4
```

Programmation Hadoop – Classe Reducer

Important:

→ Il faut que les types génériques de la ***clé*** et ***valeur*** **d'entrée** de notre classe Reducer correspondent avec les arguments de la méthode reduce :

```
public class Reduce extends Reducer<Text, IntWritable, Text, Text> {
        protected void reduce(Text key, Iterable<IntWritable> values, Context context)
        throws IOException, InterruptedException {
```

→ Il faut également que les types de clef et la valeur renvoyées par la méthode reduce correspondent avec types génériques de la ***clé*** et ***valeur*** **de sortie** de notre classe Reducer :

```
public class Reduce extends Reducer<Text, IntWritable, Text, Text> {
        protected void reduce(Text key, Iterable<IntWritable> values, Context context)
        throws IOException, InterruptedException {
                context.write(new Text("ciel"), new Text("5 occurrences"));
```

Programmation Hadoop – Classe Reducer

Important:

→ Il faut aussi que les types de ***clef*** et ***valeur*** **de sortie** de notre classe Reducer correspondent avec les types indiqués dans notre classe Driver :

```
job.setOutputKeyClass(Text.class);
job.setOutputValueClass(Text.class);
```

=> dans le main du Driver

```
public class Reduce extends Reducer<Text, IntWritable, Text, Text> {
        protected void reduce(Text key, Iterable<IntWritable> values, Context context)
        throws IOException, InterruptedException {
                context.write(new Text("ciel"), new Text("5 occurences"));
```

Programmation Hadoop – Classe Reducer

Important:

→ Finalement, il faut aussi que les types génériques de la ***clé*** et ***valeur*** **de sortie** de la classe **Mapper correspondent** avec les types génériques de la ***clé*** et ***valeur*** **d'entrée** de notre classe **Reducer** :

```
public class Map extends Mapper<LongWritable, Text, Text, IntWritable> {
        protected void map(LongWritable lineno, Text line, Context context)
        throws IOException, InterruptedException {
                context.write(new Text("ciel"), new IntWritable(5))
```

```
public class Reduce extends Reducer<Text, IntWritable, Text, Text> {
        protected void reduce(Text key, Iterable<IntWritable> values, Context context)
        throws IOException, InterruptedException {
```

Programmation Hadoop – Classe Reducer

→ Le code complet de notre classe Reducer (1/2):

```
package org.mbds.hadoop.wordcount;

import org.apache.hadoop.io.Text;
import org.apache.hadoop.io.IntWritable;
import org.apache.hadoop.mapreduce.Reducer;
import java.util.Iterator;
import java.io.IOException;
```

Programmation Hadoop – Classe Reducer

→ Le code complet de notre classe Reducer (2/2):

```
public class WCountReduce extends Reducer<Text, IntWritable, Text, Text> {
   public void reduce(Text key, Iterable<IntWritable> values, Context context)
   throws IOException, InterruptedException {
      // Parcourt toutes les valeurs associées à la clef fournie.
      Iterator<IntWritable> i = values.iterator();
      int count = 0;
      while(i.hasNext()) {
         count += i.next().get();
      }
      // Renvoie notre couple (clef; valeur)
      context.write(key, new Text(count + " occurrences."));
   }
}
```

DATUM ACADEMY

Exécution

Compilation, Exécution, Hadoop Streaming

Programmation Hadoop – Compilation

- Pour compiler un programme Hadoop, il suffit d'utiliser le compilateur **javac**.
- Il est nécessaire d'inclure certaines **librairies d'Hadoop** :
  - hadoop-common.jar
  - hadoop-mapreduce-client-core.jar
  - hadoop-client.jar

Programmation Hadoop – Exécution

- La commande "**hadoop**" avec l'option "**jar**" permet d'exécuter un programme Hadoop.
- Usage :

```
hadoop jar <JAR_FILE> <DRIVER_CLASSPATH> [ARG1] [ARG2] [ARG3] …
```

- Exemple :

```
hadoop jar wordcount.jar org.mbds.hadoop.wordcount.WCount /poeme.txt  /results
```

- Nous pouvons omettre **<DRIVER_CLASSPATH>** si un fichier **Manifest** dans le .jar spécifie la classe principale.

Programmation Hadoop – Exécution

Arguments **spécifiques** à Hadoop (récupérés via **GenericOptionsParser**) :

hadoop jar <JAR_FILE> <DRIVER_CLASSPATH> [ARG1] [ARG2] [ARG3] …

- -libjars : pour charger des bibliothèques jar supplémentaires.
- -fs : pour spécifier l'adresse du NameNode.
- -jt : pour spécifier l'adresse de YARN.
- -D : pour définir manuellement les variables de configuration.
- -conf : pour récupérer des variables de configuration depuis un fichier de configuration.
- … et d'autres …

Programmation Hadoop – Exécution

Une fois la commande “hadoop jar” est lancé, elle peut :

- ➔ Soit attendre que toutes les tâches Hadoop soient terminées. (job.waitForCompletion())
- ➔ Soit sortir immédiatement. (job.submit())
- ➔ Soit jamais se terminer. (boucle infinie dans Mapper/Reducer, etc.)

Programmation Hadoop – Exécution - Résultats

Les **résultats** de l'exécution sont écrit dans un répertoire **sur HDFS** (par défaut) :

- ➔ Un fichier par partition - par exécution de Reduce.
  Sauf en mode nœud unique : produira un seul fichier part-r-00000
- ➔ Les fichiers sont nommés « **part-r-XXXXX** », où XXXXX est un compteur incrémental.
- ➔ Un seul fichier vide est également créé en cas de succès : « **_SUCCESS** »
  (utile pour l'automatisation)

Programmation Hadoop – Exécution - Résultats

Pour **afficher les résultats**, par exemple :

```
hadoop fs -cat /results/*
```

- Vous pouvez également les récupérer les résultats sur le disque local avec « hadoop fs –get ».
- Tenter d'exécuter un programme avec **un répertoire de sortie existant est une erreur fatale**.

Programmation Hadoop - Autres langages – Streaming

- Hadoop Streaming permet d'exécuter un programme Hadoop écrit dans d'autres langages que Java.
- Il est disponible dans le répertoire d'installation Hadoop. (hadoop-streaming-VERSION.jar, où VERSION est la version de Hadoop concernée).
- Il prend en argument deux scripts/programmes, un pour l'opération Map et un pour Reduce.
- Il est nécessaire que les dépendances des scripts/programmes soient disponibles/installées sur tous les nœuds de travail du cluster.

Streaming - MAP

Dans Streaming, le programme **Map** :

- ➔ Reçoit sur **l'entrée standard (stdin)** les données d'entrée, sous forme d'une série de lignes.
- ➔ Envoie sur **la sortie standard (stdout)** les données de sortie, sous forme d'une série de lignes au format: "**CLEF**[**TABULATION**]**VALEUR**"

Streaming - REDUCE

Dans Streaming, le programme **Reduce** :

- ➔ Reçoit sur **l'entrée standard (stdin)** une série de lignes: des couples (clef;valeur) au format : **CLEF[TABULATION]VALEUR**
  - Les couples sont triés par clef distincte, et la clef est répétée à chaque fois.
  - Par ailleurs, on est susceptible d'avoir des clefs différentes au sein d'une seule et même exécution du programme Reduce.
- ➔ Envoie sur **la sortie standard (stdout)** les données de sortie, sous forme d'une ou plusieurs lignes au même format: "**CLEF[TABULATION]VALEUR**"

Streaming - Exemple (Python)

Occurrences de mots version Python – opération MAP :

```
import sys

for line in sys.stdin:
    words = line.strip().split()
    for word in words:
        print(f"{word}\t1")
```

Streaming - Exemple (Python)

Occurrences de mots version Python – opération REDUCE :

```
import sys

total = 0
lastword = None
for line in sys.stdin:
    word, count = line.strip().split('\t', 1)
    count = int(count)
    if word != lastword and lastword:
        print(f"{lastword}\t{total}")
        total = 0
    lastword = word
    total = total + count
print(f"{lastword}\t{total}")
```

Streaming - Exécution

- Streaming est un programme Hadoop standard.
- Pour l'exécuter :

```
hadoop jar hadoop-streaming-X.Y.Z.jar \
    -input [HDFS INPUT FILES] \
    -output [HDFS OUTPUT FILES] \
    -mapper [MAP PROGRAM]  \
    -reducer [REDUCE PROGRAM]
```

- Par exemple :

```
hadoop jar hadoop-streaming-X.Y.Z.jar -input /poeme.txt \
    -output /results -mapper ./map.py -reducer ./reduce.py
```

DATUM ACADEMY

HDFS API

Interagir avec HDFS depuis un programme Java

HDFS API

- Permet de lire, et d'écrire, les fichiers et répertoires sur HDFS.
- Peut être utilisé à l'intérieur d'un programme Hadoop, ou dans n'importe quel programme Java.
- Classe principale de l'API : **org.apache.hadoop.fs.FileSystem**
- Utilisé le patron de conception "**Factory**" : on obtient une instance à partir de l'objet **Configuration**.

HDFS API

→ Création d'un nouvel objet **FileSystem** (quand le nœud local est un nœud du cluster) :

```
FileSystem fs = FileSystem.get(conf);
```

→ Création d'un nouvel objet **FileSystem** (spécifiant le **NameNode** à contacter) :

```
Configuration conf = new Configuration();
String hdfs_url =
"hdfs://10.0.0.1:8020/users/john/poem.txt";
conf.set("fs.defaultFS", hdfs_url);
System.setProperty("HADOOP_USER_NAME", "hdfs");
System.setProperty("hadoop.home.dir", "/");
FileSystem fs = FileSystem.get(URI.create(hdfs_url), conf);
```

HDFS API

- Ouverture d'un fichier pour la lecture :

```
fs.open(new Path("/users/john/poem.txt"));
```

- Retourne un InputStream Java.
- Pour lire et afficher toutes les lignes :

```
InputStreamReader r = new InputStreamReader(fs.open(new Path("/poem.txt")));
BufferedReader br = new BufferedReader(r);
String line = null;
while((line = br.readLine()) != null) {
        System.out.println(line);
}
br.close();
```

HDFS API

- Création d'un fichier pour l'écriture :

```
fs.create(new Path("/users/john/poem.txt"));
```

- Retourne un OutputStream Java.
- Pour écrire une ligne :

```
OutputStreamWriter o = new OutputStreamWriter(fs.create(new Path("/poem.txt")));
BufferedWriter bw = new BufferedWriter(o);
bw.write("Ceci est une nouvelle ligne dans le nouveau fichier\n");
bw.close();
```

HDFS API

- Ouverture d'un fichier existant pour écriture (append) :

```
fs.append(new Path("/users/john/poem.txt"));
```

- Retourne un OutputStream Java.
- Pour ajouter une ligne au fichier :

```
OutputStreamWriter o = new OutputStreamWriter(fs.append(new Path("/poem.txt")));
BufferedWriter bw = new BufferedWriter(o);
bw.write("Ceci est une nouvelle ligne à la fin du fichier existant\n");
bw.close();
```

HDFS API

- Utilisation du métacaractère (“*”) avec **globStatus()**:

```
FileStatus[] list = fs.globStatus(new Path("/results/part-r-*"));
for(int i = 0; i < list.length; ++i) {
        InputStreamReader r = new InputStreamReader(fs.open(list[i].getPath()));
        BufferedReader br = new BufferedReader(r);
…
```

- Particulièrement utile pour les répertoires de résultats.

HDFS API

Autres fonctions :

- copyFromLocalFile()
- copyToLocalFile()
- getDefaultBlockSize()
- mkdirs()
- delete()
- … et beaucoup plus encore ...