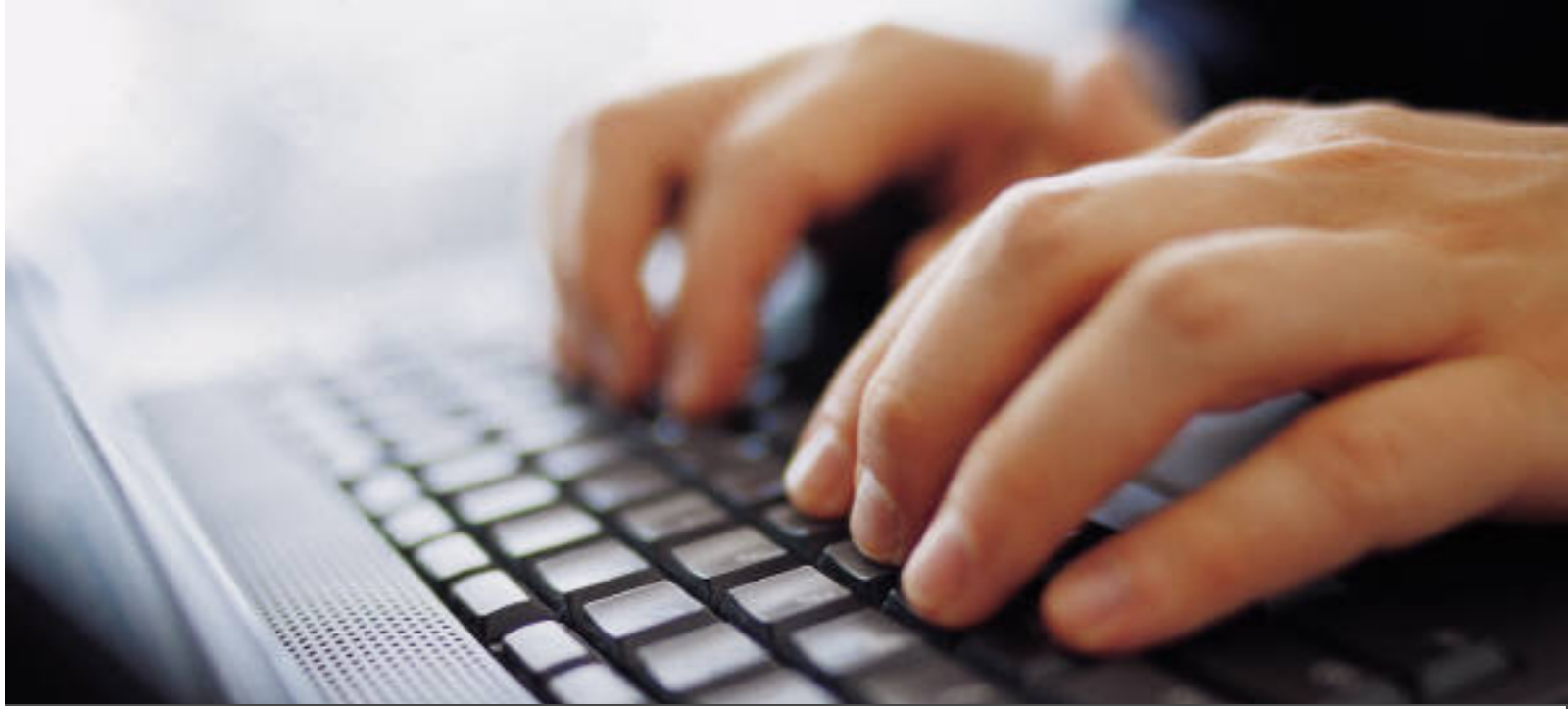
Hadoop / Big Data

Benjamin Renaut <renaut.benjamin@tokidev.fr>

MBDS

2019 - 2020



1

Introduction

Programme – Planning – Objectifs – TP/Évaluations

Introduction

Benjamin Renaut

Tokidev SAS

- Bureau d'étude
- Développement
- Consulting

http://www.tokidev.fr/

Avant de commencer

- **Posez des questions.**
  Si un point n'est pas clair, n'hésitez pas à poser des questions à tout moment.

- **Point de contact: ben@tokidev.fr**
  Vous pouvez m'envoyer toute question par e-mail concernant le cours, les TDs ou le partiel.

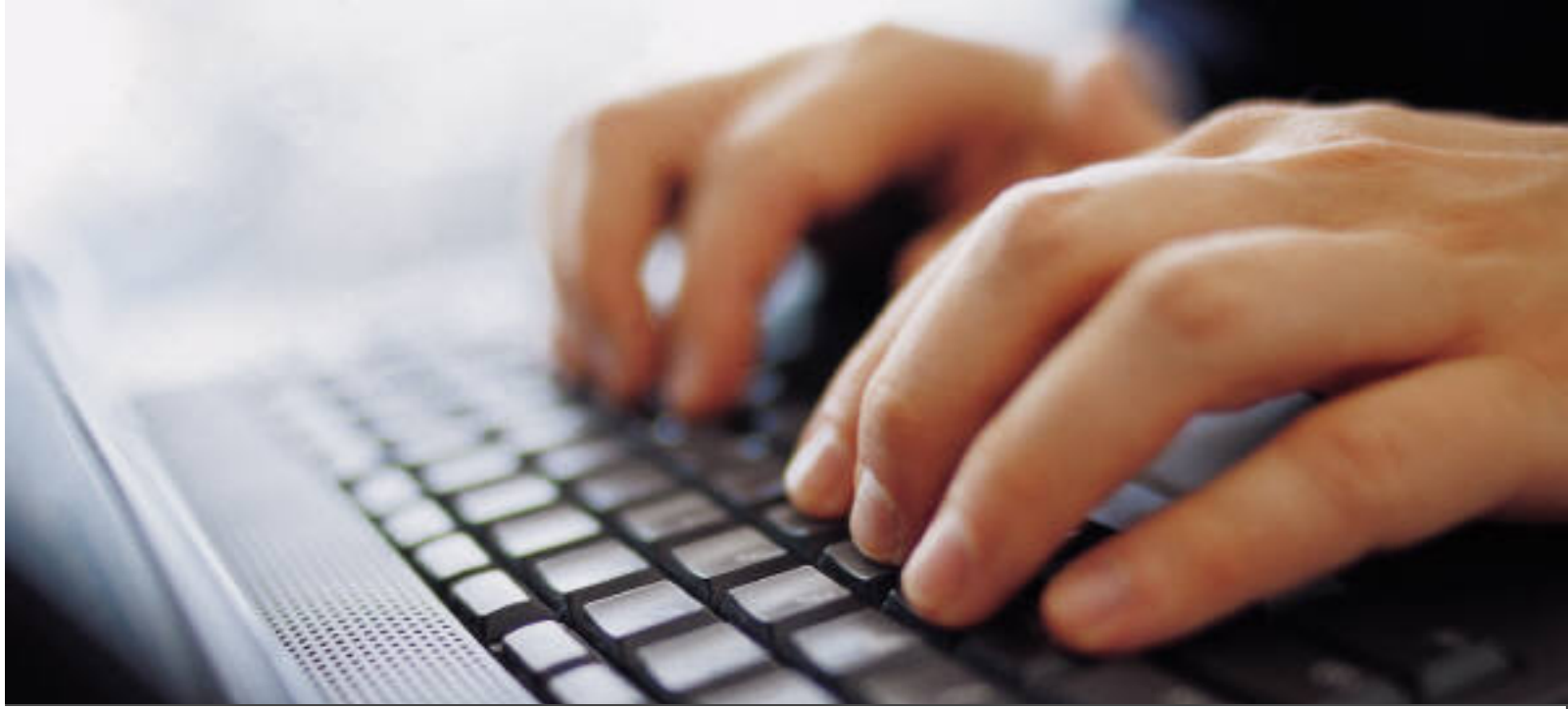
- **Google est votre ami !**
  Si vous rencontrez un problème, prenez le réflexe d'effectuer une recherche – dans une majorité de cas, quelqu'un d'autre aura eu le même problème que vous :-)

- **Cours en ligne: http://cours.tokidev.fr/bigdata/**
  Disponible à cette adresse: la machine virtuelle utilisée pour les TPs, les slides des cours.

Objectifs

- **Découvrir la méthodologie map/reduce.**
- **Apprendre à installer et utiliser Hadoop.**
- **Apprendre à rédiger et exécuter des programmes pour Hadoop.**
- **Découvrir diverses solutions complémentaires liées à Hadoop et aux problématiques « Big Data » en général (Pig, MongoDB, etc.).**
- **Apprendre à utiliser Apache Spark.**

► **Apprentissage basé sur la pratique.**



2

Le calcul distribué / Historique Hadoop

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.**

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.

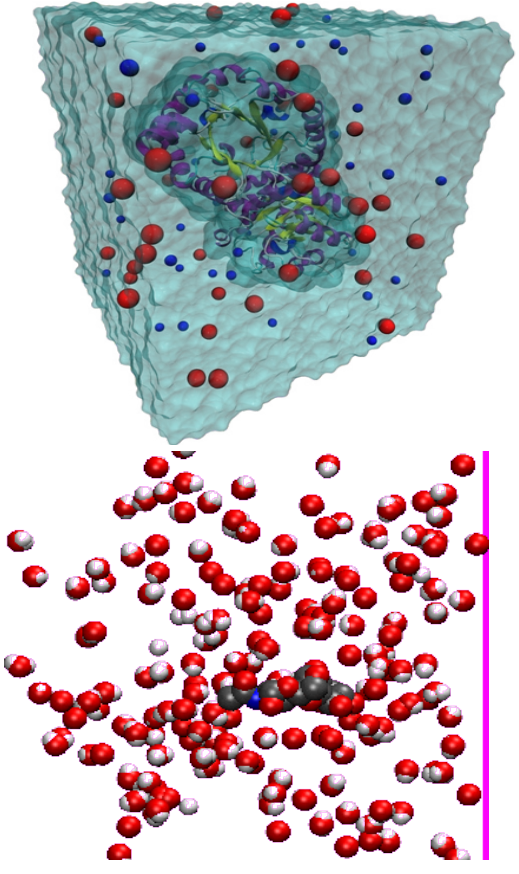
Utilisé pour des simulations médicales, l'étude de signaux radio astronomiques, etc.

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.



Exemple: clusters 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.

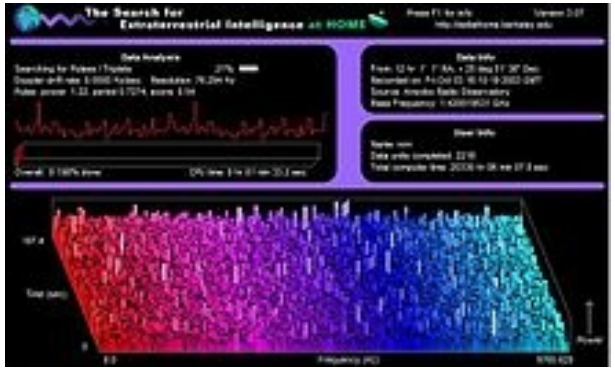
Généralement concrétisé via plusieurs machines sous GNU/Linux utilisant des interfaces standardisées. L'ensemble du *cluster* ainsi formé apparaît au niveau du serveur maître comme une seule et même machine.

Permet à tout un chacun d'obtenir un système de calcul distribué à hautes performances à partir d'un matériel peu onéreux.

La NASA utilise ainsi un cluster Beowulf.

Exemple: SETI@Home/BOINC

A l'origine, un des premiers logiciels à permettre le calcul distribué *via* l'utilisation des ordinateurs de volontaires tout autour du monde.



Objectif : détecter d'éventuels signes de civilisations extra-terrestres en analysant des signaux radios reçus par l'observatoire Arecibo au Porto Rico.

Architecture et protocoles de communication ensuite standardisés et mis à disposition sous la forme d'un framework, BOINC, donnant lieu à de nombreux autres projets (Einstein@Home, Folding@Home, etc.).
Remarque: GPUGRID utilise désormais lui aussi le framework BOINC.

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.**
- **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.

La solution: Apache Hadoop

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, 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.**

Historique (1/2)

- 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.

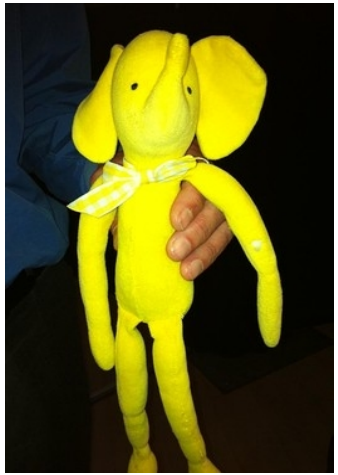
- 2003/2004: le département de recherche de Google publie deux whitepapers, le premier sur GFS (un système de fichier distribué) et le second sur le paradigme Map/Reduce pour le calcul distribué.

- 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...

Historique (2/2)

… 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.

Qui utilise Hadoop

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

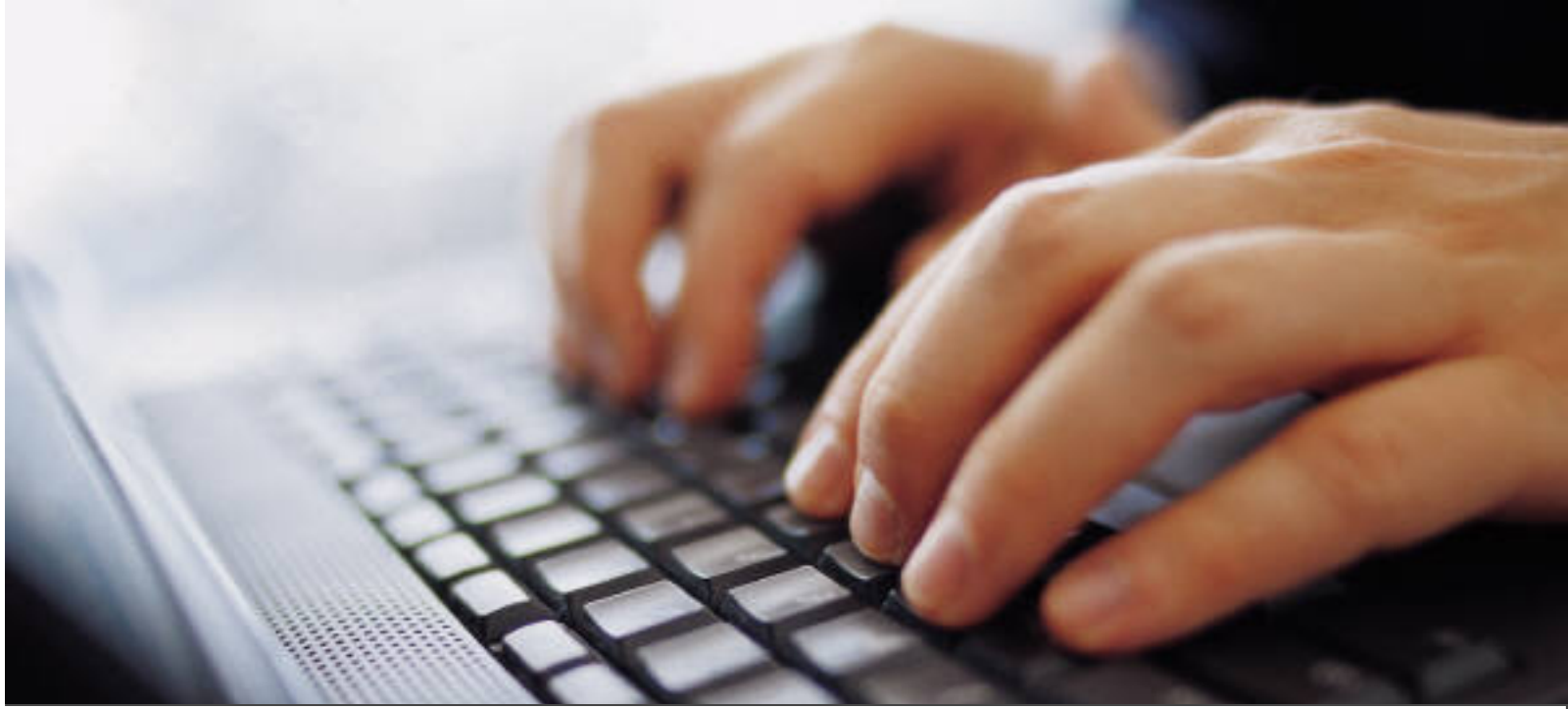
Une technologie en plein essort

- **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

Le modèle Map/Reduce

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.**

- **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.).

Exemple concret (1/8)

Imaginons qu'on nous donne un texte écrit en langue Française. On souhaite déterminer pour un travail de recherche quels sont les mots les plus utilisés au sein de ce texte (exemple Hadoop très répandu).

Ici, nos données d'entrée sont constituées du contenu du texte.

Première étape: déterminer une manière de découper (*split*) les données d'entrée pour que chacune des machines puisse travailler sur une partie du texte.

Notre problème est ici très simple – on peut par exemple décider de découper les données d'entrée ligne par ligne. Chacune des lignes du texte sera un fragment de nos données d'entrée.

Exemple concret (2/8)

Nos données d'entrée (le texte):

```
Celui qui croyait au ciel
Celui qui n'y croyait pas
[…]
Fou qui fait le délicat
Fou qui songe à ses querelles
```

(Louis Aragon, *La rose et le Réséda*, 1943, fragment)

Pour simplifier les choses, on va avant le découpage supprimer toute ponctuation et tous les caractères accentués. On va également passer l'intégralité du texte en minuscules.

Exemple concret (3/8)

Après découpage:

```
celui qui croyait au ciel
celui qui ny croyait pas
fou qui fait le delicat
fou qui songe a ses querelles
```

… on obtient 4 fragments depuis nos données d'entrée.

Exemple concret (4/8)

On doit désormais déterminer la clef à utiliser pour notre opération MAP, et écrire le code de l'opération MAP elle-même.

Puisqu'on s'intéresse aux occurrences des mots dans le texte, et qu'à terme on aura après l'opération REDUCE un résultat pour chacune des clefs distinctes, la clef qui s'impose logiquement dans notre cas est: le mot-lui même.

Quand à notre opération MAP, elle sera elle aussi très simple: on va simplement parcourir le fragment qui nous est fourni et, pour chacun des mots, générer le couple clef/valeur: (MOT ; 1). La valeur indique ici l'occurrence pour cette clef - puisqu'on a croisé le mot une fois, on donne la valeur « 1 ».

Exemple concret (5/8)

Le code de notre opération MAP sera donc (ici en pseudo code):

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

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

`celui qui croyait au ciel` → (celui;1) (qui;1) (croyait;1) (au;1) (ciel;1)

`celui qui ny croyait pas` → (celui;1) (qui;1) (ny;1) (croyait;1) (pas;1)

`fou qui fait le delicat` → (fou;1) (qui;1) (fait;1) (le;1) (delicat;1)

`fou qui songe a ses querelles` → (fou;1) (qui;1) (songe;1) (a;1) (ses;1) (querelles;1)

Exemple concret (6/8)

Une fois notre opération MAP effectuée (de manière distribuée), Hadoop groupera (*shuffle*) tous les couples par clef commune.

Cette opération est effectuée automatiquement par Hadoop. Elle est, là aussi, effectuée de manière distribuée en utilisant un algorithme de tri distribué, de manière récursive. Après son exécution, on obtiendra les 15 groupes suivants:

(celui;1) (celui;1)

(qui;1) (qui;1) (qui;1) (qui;1)

(croyait;1) (croyait;1)

(au;1) (ny;1)

(ciel;1) (pas;1)

(fou;1) (fou;1)

(fait;1) (le;1)

(delicat;1) (songe;1)

(a;1) (ses;1)

(querelles;1)

Exemple concret (7/8)

Il nous reste à créer notre opération REDUCE, qui sera appelée pour chacun des groupes/clef distincte.

Dans notre cas, elle va simplement consister à additionner toutes les valeurs liées à la clef spécifiée:

```
TOTAL=0
POUR COUPLE dans GROUPE, FAIRE:
  TOTAL=TOTAL+1
RENVOYER TOTAL
```

Exemple concret (8/8)

Une fois l'opération REDUCE effectuée, on obtiendra donc une valeur unique pour chaque clef distincte. En l'occurrence, notre résultat sera:

```
qui: 4
celui: 2
croyait: 2
fou: 2
au: 1
ciel: 1
ny: 1
pas: 1
fait: 1
[…]
```

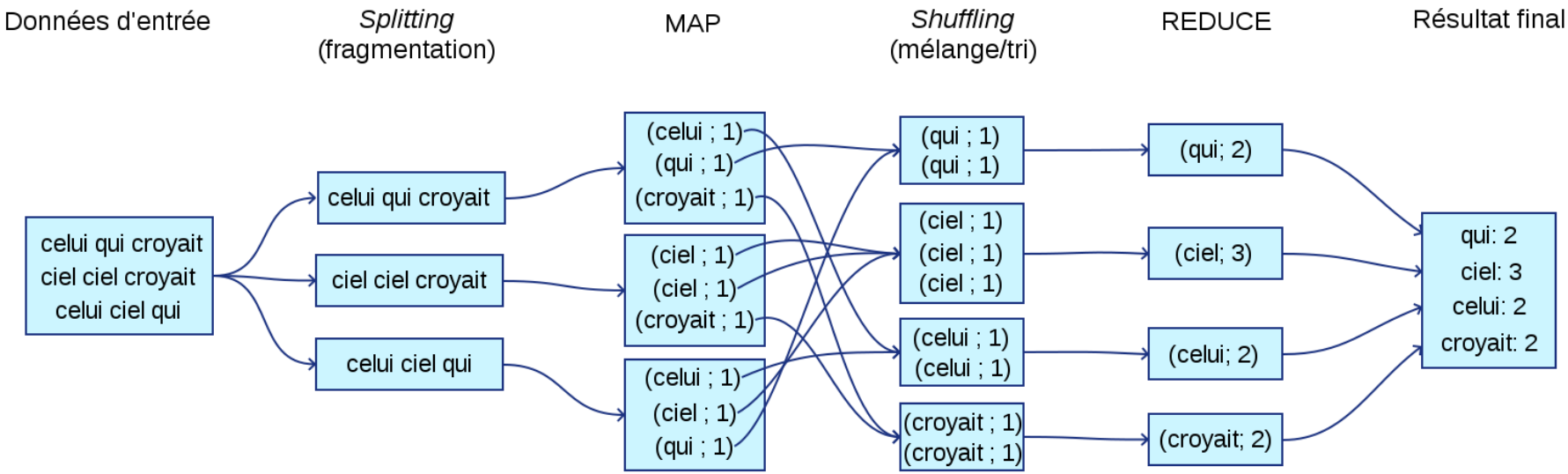
On constate que le mot le plus utilisé dans notre texte est « qui », avec 4 occurrences, suivi de « celui », « croyait » et « fou », avec 2 occurrences chacun.

Exemple concret - conclusion

Notre exemple est évidemment trivial, et son exécution aurait été instantanée même sur une machine unique, mais il est d'ores et déjà utile: on pourrait tout à fait utiliser les mêmes implémentations de MAP et REDUCE sur l'intégralité des textes d'une bibliothèque Française, et obtenir ainsi un bon échantillon des mots les plus utilisés dans la langue Française.

L’intérêt du modèle MapReduce est qu'il nous suffit de développer les deux opérations réellement importantes du traitement: MAP et REDUCE, et de bénéficier automatiquement de la possibilité d'effectuer le traitement sur un nombre variable de machines de manière distribuée.

Schéma général



Exemple – Statistiques web

Un autre exemple: on souhaite compter le nombre de visiteurs sur chacune des pages d'un site Internet.
On dispose des fichiers de logs sous la forme suivante:

```
/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=4 [24/Oct/2013:18:13:12 +0200]
/news.php?id=18 [24/Oct/2013:18:14:31 +0200]
...etc...
```

Ici, notre clef sera par exemple l'URL d'accès à la page, et nos opérations MAP et REDUCE seront exactement les mêmes que celles qui viennent d'être présentées: on obtiendra ainsi le nombre de vue pour chaque page distincte du site.

Exemple – Graphe social

Un autre exemple: on administre un réseau social comportant des millions d'utilisateurs.
Pour chaque utilisateur, on a dans notre base de données la liste des utilisateurs qui sont ses amis sur le réseau (*via* une requête SQL).

On souhaite afficher quand un utilisateur va sur la page d'un autre utilisateur une indication « Vous avez *N* amis en commun ».

On ne peut pas se permettre d'effectuer une série de requêtes SQL à chaque fois que la page est accédée (trop lourd en traitement). On va donc développer des programmes MAP et REDUCE pour cette opération et exécuter le traitement toutes les nuits sur notre base de données, en stockant le résultat dans une nouvelle table.

Exemple – Graphe social

Ici, nos 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
```

Puisqu'on est intéressé par l'information « amis en commun entre deux utilisateurs » et qu'on aura à terme une valeur par clef, on va choisir pour clef la concaténation entre deux utilisateurs.
Par exemple, la clef « A-B » désignera « les amis en communs des utilisateurs A et B ».
On peut segmenter les données d'entrée là aussi par ligne.

Exemple – Graphe social

Notre opération MAP va se contenter de prendre la liste des amis fournie en entrée, et va générer toutes les clefs distinctes possibles à partir de cette liste. La valeur sera simplement la liste d'amis, telle quelle.

On fait également en sorte que la clef soit toujours triée par ordre alphabétique (clef « B-A » sera exprimée sous la forme « A-B »).

Ce traitement peut paraître contre-intuitif, mais il va à terme nous permettre d'obtenir, pour chaque clef distincte, deux couples (clef;valeur): les deux listes d'amis de chacun des utilisateurs qui composent la clef.

Exemple – Graphe social

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])
```

Par exemple, pour la première ligne: `A => B, C, D`

On obtiendra les couples (clef;valeur):

`("A-B"; "B C D")` `("A-C"; "B C D")` `("A-D"; "B C D")`

Exemple – Graphe social

Pour la seconde ligne: `B => A, C, D, E`

On obtiendra ainsi:

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

Pour la troisième ligne: `C => A, B, D, E`

On aura:

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

...et ainsi de suite pour nos 5 lignes d'entrée.

Exemple – Graphe social

Une fois l'opération MAP effectuée, Hadoop va récupérer les couples (clef;valeur) de tous les fragments et les grouper par clef distincte. Le résultat sur la base de nos données d'entrée:

```
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"
```

… on obtient bien, pour chaque clef « USER1-USER2 », deux listes d'amis: les amis de USER1 et ceux de USER2.

Exemple – Graphe social

Il nous faut enfin écrire notre programme REDUCE.
Il va recevoir en entrée toutes les valeurs associées à une clef. Son rôle va être très simple: déterminer quels sont les amis qui apparaissent dans les listes (les valeurs) qui nous sont fournies. 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 EGALEMENT PRESENT DANS VALEURS[1], ALORS:
      // Présent dans les deux listes d'amis, on l'ajoute.
      LISTE_AMIS_COMMUNS+=AMI
RENVOYER LISTE_AMIS_COMMUNS
```

Exemple – Graphe social

Après exécution de l'opération REDUCE pour les valeurs de chaque clef unique, on obtiendra donc, pour une clef « A-B », les utilisateurs qui apparaissent dans la liste des amis de A et dans la liste des amis de B. Autrement dit, on obtiendra la liste des amis en commun des utilisateurs A et B. Le résultat:

```
"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"
```

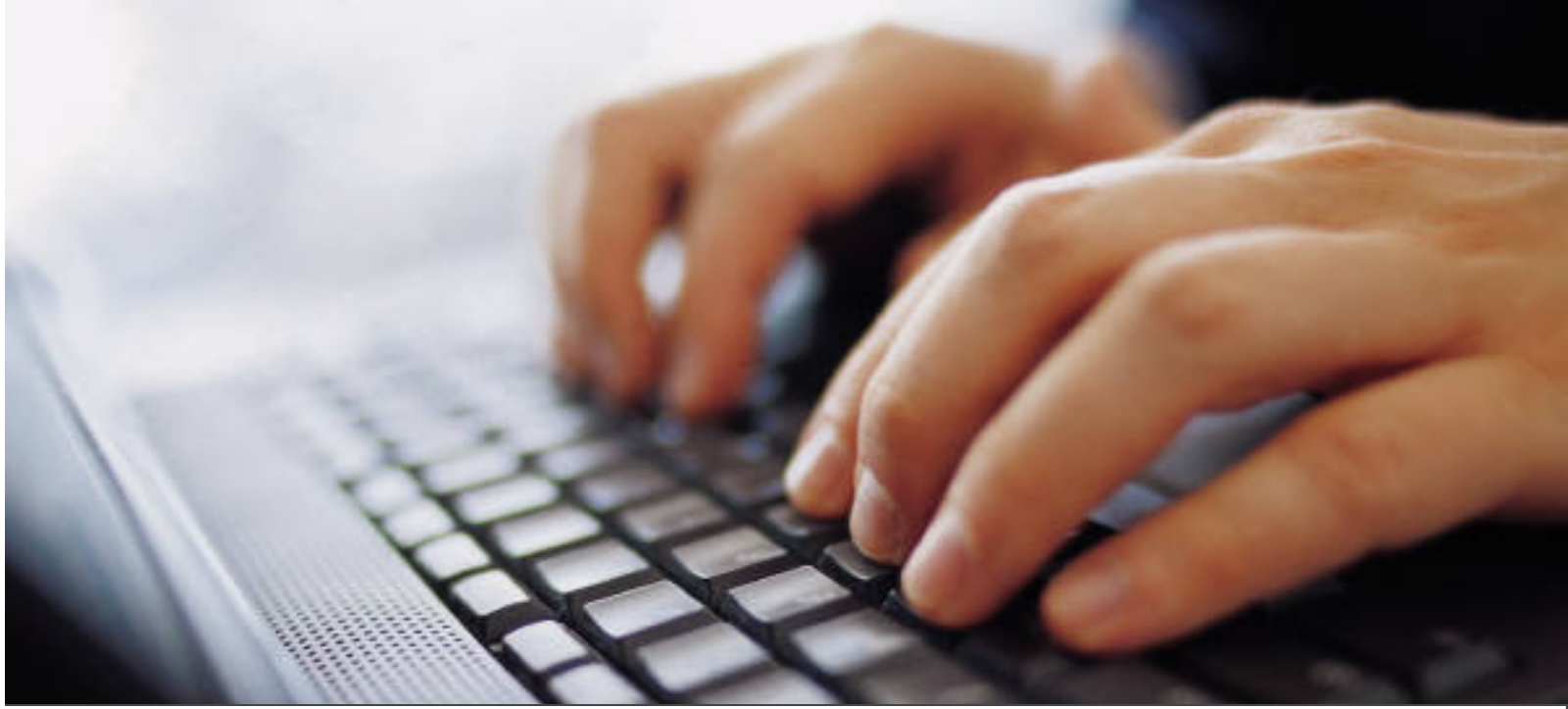
On sait ainsi que A et B ont pour amis communs les utilisateurs C et D, ou encore que B et C ont pour amis communs les utilisateurs A, D et E.

Conclusion

En utilisant le modèle MapReduce, on a ainsi pu créer deux programmes très simples (nos programmes MAP et REDUCE) de quelques lignes de code seulement, qui permettent d'effectuer un traitement somme toute assez complexe.

Mieux encore, notre traitement est parallélisable: même avec des dizaines de millions d'utilisateurs, 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.

Pour notre réseau social, il suffira d'effectuer ce traitement toutes les nuits à heure fixe, et de stocker les résultats dans une table.
Ainsi, lorsqu'un utilisateur visitera la page d'un autre utilisateur, un seul SELECT dans la base de données suffira pour obtenir la liste des amis en commun – avec un poids en traitement très faible pour le serveur.



4

HDFS

Présentation

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.

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. 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).**

Présentation

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.

Architecture

HDFS repose sur deux serveurs (des *daemons*):

- **Le *NameNode*, 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...**

Par défaut, les données sont divisées en blocs de 64MB (configurable). Hadoop est inspiré de GFS, un système de fichier distribué conçu par Google. L'implémentation de HDFS a son origine dans un *whitepaper* issu du département de recherche de Google (« *The Google File System* », 2003).

Écriture d'un fichier

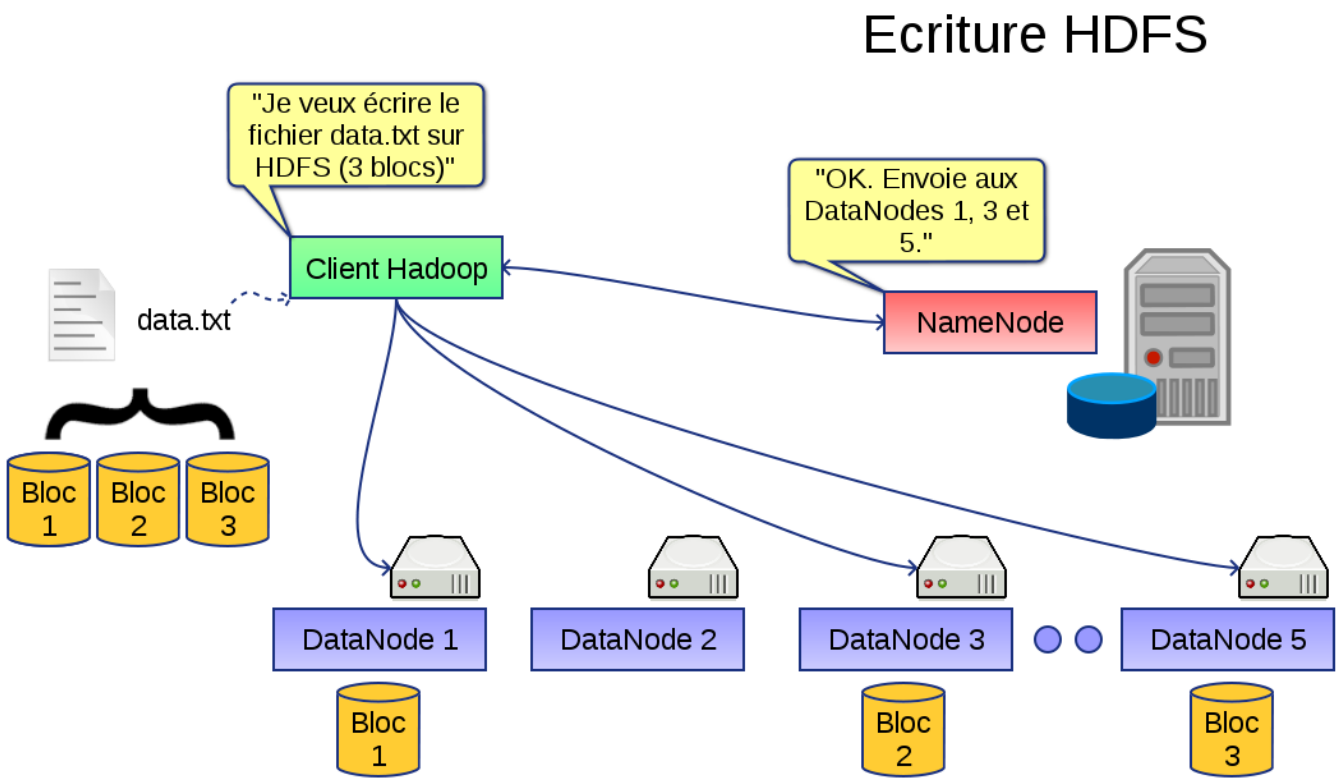
Si on souhaite écrire un fichier au sein de HDFS, on va utiliser la commande principale de gestion de Hadoop: `hadoop`, avec l'option `fs`. Mettons qu'on souhaite stocker le fichier page_livre.txt sur HDFS.

Le programme va diviser le fichier en blocs de 64MB (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 ».

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.

Écriture d'un fichier



- 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.

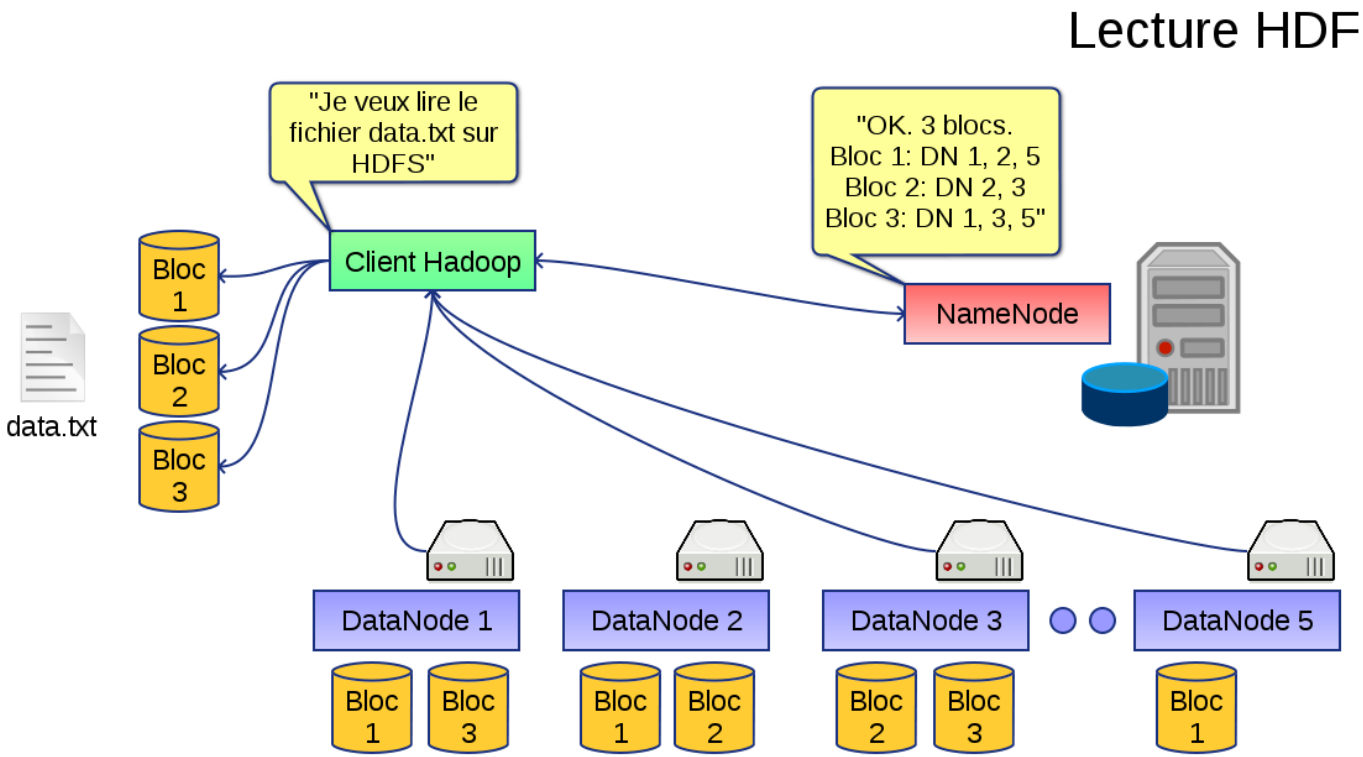
Lecture d'un fichier

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 d'un fichier



- 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`.
  - **`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...

Remarques

- **La gestion du stockage est assurée par les *daemons* Hadoop – on a pas à se soucier d'où sont stockées les données.**

- **Hadoop réplique lui-même les données: les fichiers sont disponibles à tout moment sur plusieurs DataNodes, et si une machine tombe en panne, on a toujours accés aux données grâce à la réplication.**

- **Si les données proviennent d'une base de données (par exemple dans le cas de l'exemple des « amis en commun » vu précédemment), on sera obligé avant d'exécuter le traitement Hadoop d'extraire les données de la base *via* des requêtes SQL et de les stocker sur HDFS. De même, pour récupérer les résultats au sein d'une base de données, on devra extraire le ou les fichiers des résultats depuis HDFS, et les ré-importer dans la base de données. C'est dans ce cas que les *bridges* de connexion aux bases de données évoquées plus haut prennent tout leur sens.**

Inconvénients

- **Le NameNode est unique – on ne peut pas en avoir plusieurs. En conséquence, si la machine hébergeant le NameNode tombe en panne, le cluster est incapable d'accéder aux fichiers le temps que le problème soit corrigé. Ce problème est partiellement mitigé dans les versions récentes (beta) de Hadoop, avec un BackupNameNode qui peut être basculé manuellement en cas de problème. Une implémentation est en cours pour un basculement automatique.**

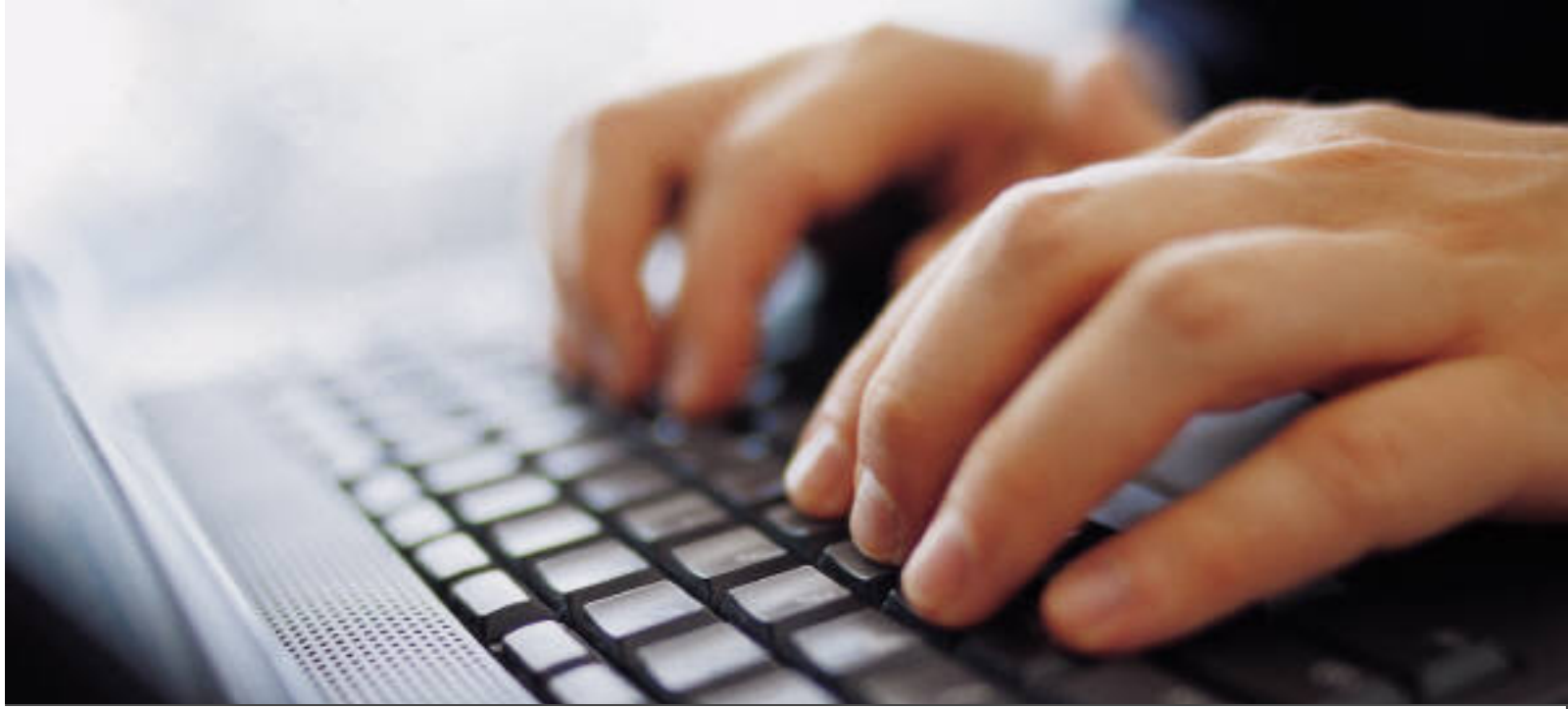
- **Le système de fichier ne peut pas être monté de manière « classique ». Pour y accéder, on est obligé de passer par l'utilitaire `hadoop` (ou par des APIs). Remarque: ce n'est plus entièrement vrai – plusieurs projets FUSE existent désormais pour assurer des points de montage classiques.**

- **Enfin, HDFS est optimisé pour des lecture concurrentes: écrire de manière concurrente est nettement moins performant.**

Alternatives

On a indiqué plus haut qu'on peut également brancher Hadoop directement par le biais de ponts sur une base de données pour en extraire des données d'entrée/y stocker des résultats. Il existe également des alternatives à HDFS qui conservent une logique de systèmes de fichiers. Par exemple:

- **Amazon S3 filesystem: le système de fichier Amazon utilisé sur les solutions de cloud de amazon.com. N'est pas conscient du positionnement par rapport aux racks.**
- **CloudStore: une alternative proposée par Kosmix. Conscient des racks.**
- **FTP: une alternative haut niveau qui transfère les blocs manuellement via FTP. Peu performant.**
- **HTTP/HTTPS en lecture seule.**



5

L'architecture Hadoop

Présentation

On distingue dans les slides qui suivent deux architectures:

- **La première, constitue du *JobTracker* et du *TaskTracker*, correspond à la première version du moteur d'exécution map/reduce (« MRv1 », Hadoop 1.x) et est présentée à titre informatif et pour des raisons historiques.**

- **A partir de la version 2.x, Hadoop intègre un nouveau moteur d'exécution: Yarn (« MRv2 »); cette architecture d'exécution, l'actuelle, est également présentée dans un second temps.**

Présentation - « MRv1 »

Comme pour HDFS, la gestion des tâches de Hadoop se basait jusqu'en version 2 sur deux serveurs (des *daemons*):

- **Le *JobTracker*, qui va directement recevoir la tâche à exécuter (un .jar Java), ainsi que les données d'entrées (nom des fichiers stockés sur HDFS) et le répertoire où stocker les données de sortie (toujours sur HDFS). Il y a un seul *JobTracker* sur une seule machine du cluster Hadoop. Le *JobTracker* est en communication avec le *NameNode* de HDFS et sait donc où sont les données.**

- **Le *TaskTracker*, qui est en communication constante avec le *JobTracker* et va recevoir les opérations simples à effectuer (MAP/REDUCE) ainsi que les blocs de données correspondants (stockés sur HDFS). Il y a un *TaskTracker* sur chaque machine du cluster.**

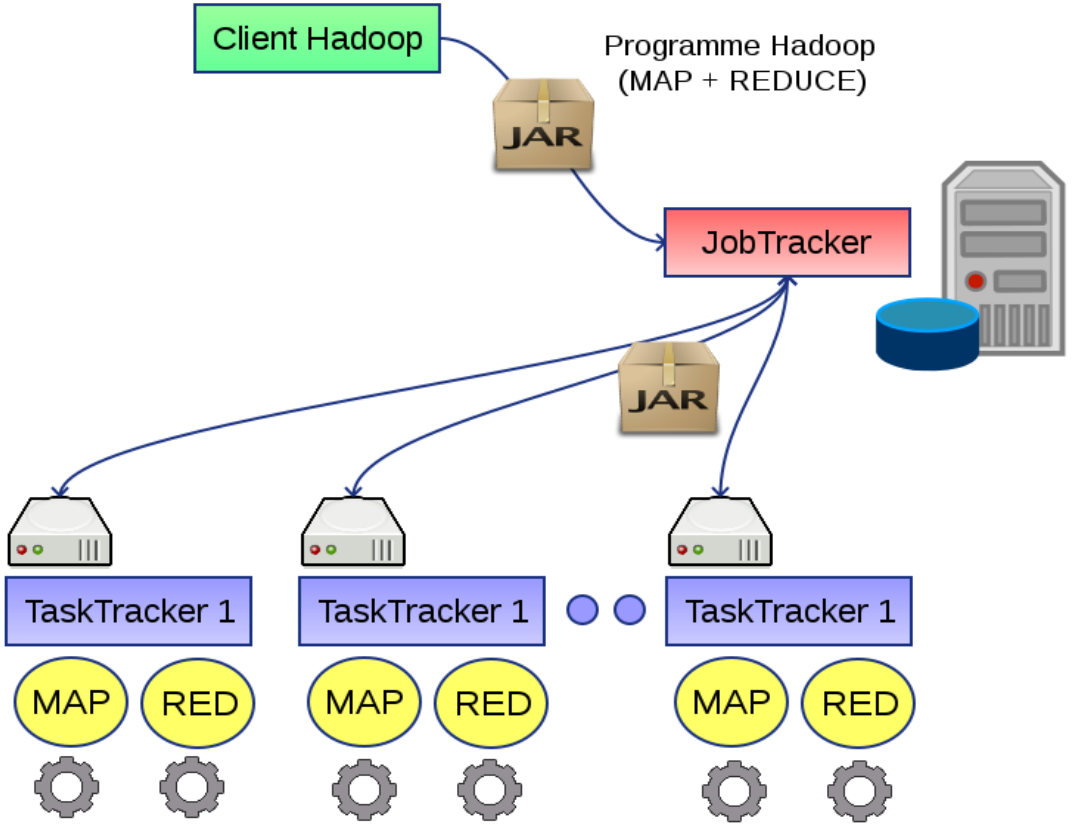
Présentation - « MRv1 »

- **Comme le *JobTracker* est conscient de la position des données (grâce au *NameNode*), il peut facilement déterminer les meilleures machines auxquelles attribuer les sous-tâches (celles où les blocs de données correspondants sont stockés).**

- **Pour effectuer un traitement Hadoop, on va donc stocker nos données d'entrée sur HDFS, créer un répertoire où Hadoop stockera les résultats sur HDFS, et compiler nos programmes MAP et REDUCE au sein d'un .jar Java.**

- **On soumettra alors le nom des fichiers d'entrée, le nom du répertoire des résultats, et le .jar lui-même au *JobTracker*: il s'occupera du reste (et notamment de transmettre les programmes MAP et REDUCE aux serveurs *TaskTracker* des machines du cluster).**

Présentation - « MRv1 »



« MRv1 » - Le JobTracker

Le déroulement de l'exécution d'une tâche Hadoop suit les étapes suivantes du point de vue du *JobTracker*:

1. Le client (un outil Hadoop console) va soumettre le travail à effectuer au JobTracker: une archive java .jar implémentant les opérations Map et Reduce. Il va également soumettre le nom des fichiers d'entrée, et l'endroit où stocker les résultats.

2. Le JobTracker communique avec le NameNode HDFS pour savoir où se trouvent les blocs correspondant aux noms de fichiers donnés par le client.

3. Le JobTracker, à partir de ces informations, détermine quels sont les nœuds TaskTracker les plus appropriés, c'est à dire ceux qui contiennent les données sur lesquelles travailler sur la même machine, ou le plus proche possible (même rack/rack proche).

« MRv1 » - Le JobTracker

4. Pour chaque « morceau » des données d'entrée, le JobTracker envoie au TaskTracker sélectionné le travail à effectuer (MAP/REDUCE, code Java) et les blocs de données correspondant.

5. Le JobTracker communique avec les nœuds TaskTracker en train d'exécuter les tâches. Ils envoient régulièrement un « heartbeat », un message signalant qu'ils travaillent toujours sur la sous-tâche reçue. Si aucun heartbeat n'est reçu dans une période donnée, le JobTracker considère la tâche comme ayant échouée et donne le même travail à effectuer à un autre TaskTracker.

6. Si par hasard une tâche échoue (erreur java, données incorrectes, etc.), le TaskTracker va signaler au JobTracker que la tâche n'a pas pû être exécuté. Le JobTracker va alors décider de la conduite à adopter: redonner la sous-tâche à un autre TaskTracker, demander au même TaskTracker de ré-essayer, marquer les données concernées comme invalides, etc. il pourra même blacklister le TaskTracker concerné comme non-fiable dans certains cas.

« MRv1 » - Le JobTracker

7. Une fois que toutes les opérations envoyées aux TaskTracker (MAP + REDUCE) ont été effectuées et confirmées comme effectuées par tous les nœuds, le JobTracker marque la tâche comme « effectuée ». Des informations détaillées sont disponibles (statistiques, TaskTracker ayant posé problème, etc.).

Par ailleurs, on peut également obtenir à tout moment de la part du *JobTracker* des informations sur les tâches en train d'être effectuées: étape actuelle (MAP, SHUFFLE, REDUCE), pourcentage de complétion, etc.

La soumission du .jar, l'obtention de ces informations, et d'une manière générale toutes les opérations liées à Hadoop s'effectuent avec le même unique client console vu précédemment: `hadoop` (avec d'autres options que l'option `fs` vu précédemment).

« MRv1 » - Le TaskTracker

- **Le *TaskTracker* dispose d'un nombre de « slots » d'exécution. A chaque « slot » correspond une tâche exécutable (configurable). Ainsi, une machine ayant par exemple un processeur à 8 cœurs pourrait avoir 16 slots d'opération configurés.**

- **Lorsqu'il reçoit une nouvelle tâche à effectuer (MAP, REDUCE, SHUFFLE) depuis le JobTracker, le TaskTracker va démarrer une nouvelle instance de Java avec le fichier .jar fourni par le JobTracker, en appelant l'opération correspondante.**

- **Une fois la tâche démarrée, il enverra régulièrement au JobTracker ses messages heartbeats. En dehors d'informer le JobTracker qu'il est toujours fonctionnels, ces messages indiquent également le nombre de slots disponibles sur le TaskTracker concerné.**

« MRv1 » - Le TaskTracker

- **Lorsqu'une sous-tâche est terminée, le TaskTracker envoie un message au JobTracker pour l'en informer, que la tâche se soit bien déroulée ou non (il indique évidemment le résultat au JobTracker).**

« MRv1 » - Remarques

- **De manière similaire au NameNode de HDFS, il n'y a qu'un seul JobTracker et s'il tombe en panne, le cluster tout entier ne peut plus effectuer de tâches. Là aussi, des résolutions aux problème sont prévues dans la roadmap Hadoop, mais pas encore disponibles.**

- **Généralement, on place le JobTracker et le NameNode HDFS sur la même machine (une machine plus puissante que les autres), sans y placer de TaskTracker/DataNode HDFS pour limiter la charge. Cette machine particulière au sein du cluster (qui contient les deux « gestionnaires », de tâches et de fichiers) est communément appelée le nœud maître (« Master Node »). Les autres nœuds (contenant TaskTracker + DataNode) sont communément appelés nœuds esclaves (« slave node »).**

« MRv1 » - Remarques

- **Même si le JobTracker est situé sur une seule machine, le « client » qui envoie la tâche au JobTracker initialement peut être exécuté sur n'importe quelle machine du cluster – comme les TaskTracker sont présents sur la machine, ils indiquent au client comment joindre le JobTracker.**

- **La même remarque est valable pour l'accés au système de fichiers: les DataNodes indiquent au client comment accéder au NameNode.**

- **Enfin, tout changement de configuration Hadoop peut s'effectuer facilement simplement en changeant la configuration sur la machine où sont situés les serveurs NameNode et JobTracker: ils répliquent les changement de configuration sur tout le cluster automatiquement.**

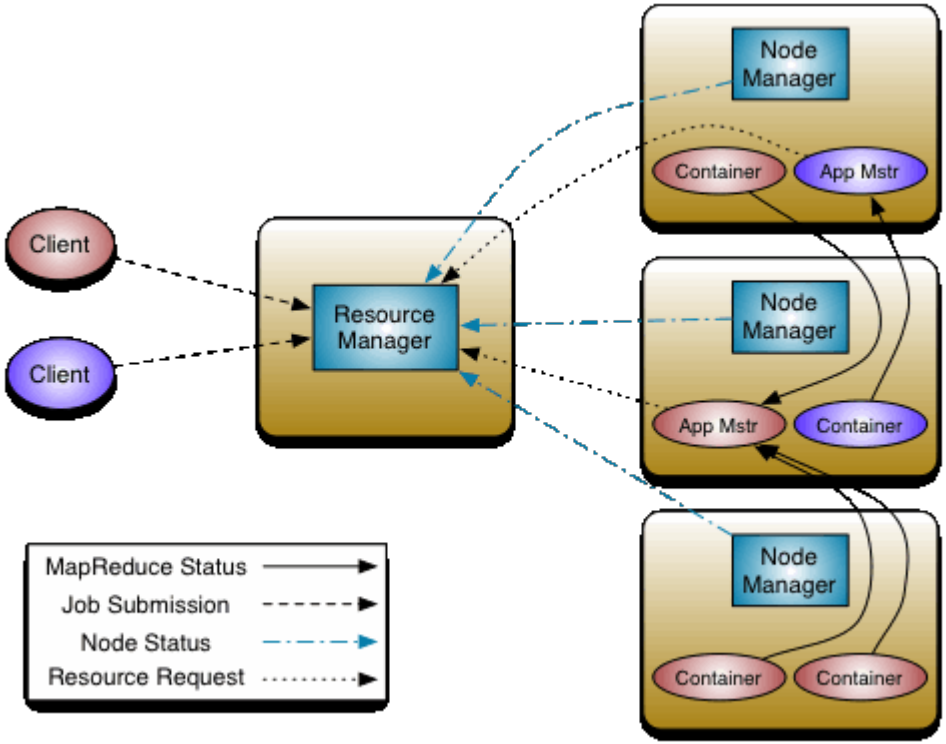
Présentation - Yarn

Le nouveau moteur d'exécution introduit dans Hadoop 2 est Yarn. Ce nouveau moteur d'exécution est:

- **Plus générique (il impose moins une logique map/reduce inflexible, et d'une manière plus générale l'interdépendance avec HDFS est moins forte).**
- **Vise à décoréler la gestion des resources (CPU, mémoire, slots d'exécution, etc.) du *scheduling* des applications dans deux démons séparés (là où JobTracker assurait ces deux tâches en MRv1).**
- **D'une manière plus générale, est plus scalable; notamment parce qu'il génère un processus maître par programme à exécuter, évitant le goulot d'étranglement provoqué par le JobTracker précédemment sur de larges *clusters*.**

Yarn – vue générale

Le démon principal est Resource Manager; il est unique sur le cluster et gère le concept de « resources ».



Yarn – ResourceManager

- **Le Resource Manager maintient une table de « resources » disponibles sur le cluster: machines, slots d'exécution, mémoire/CPU sur les machines, etc.**

- **Il est composé de deux principaux composants:**

  - **Le *scheduler*, en charge de distribuer des travaux de manière générique aux différents slots d'exécution du cluster, et en fonction des ressources disponibles et des règles liées à leur attribution.**

  - **L'*Applications Manager*, à l'écoute de nouveaux programmes à exécuter proposés par des clients, et qui lancera une première tâche lors du début de l'exécution: la tâche *Application Master*, qui constituera le « chef d'orcheste » de l'exécution du programme et soumettra par elle-même de nouvelles tâches à effectuer au *scheduler* du ResourceManager directement.**

Yarn – NodeManager

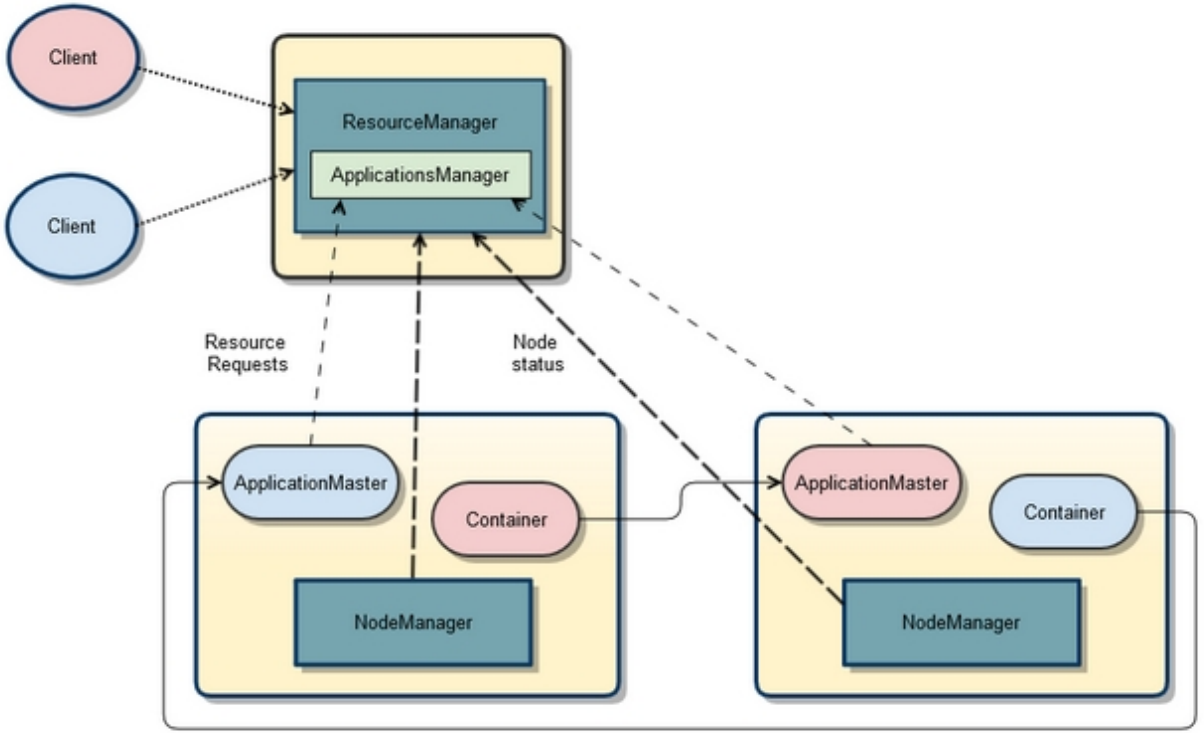
- **Du coté des nœuds, c'est un second démon, NodeManager, qui tourne sur chaque machine et qui est comparable au TaskTracker du modèle v1.**

- **Ce démon est en charge de maintenir les slots d'exécution locaux (désignés sous le terme générique de *container* d'exécution), et de recevoir et exécuter des tâches dans ces *containers* attribués aux applications par le *scheduler*.**

- **Ces tâches à exécuter peuvent être de simples opérations map ou reduce, mais aussi le coeur de l'application lui-même : la classe driver, main du programme, qui serait alors lancée au sein d'une tâche spéciale ApplicationMaster, lancée par l'ApplicationMaster du ResourceManager.**

- **Les mécaniques d'échange en terme de statut (heartbeats) restent identiques; les NodeManager mettent à jour régulierement le ResourceManager avec des mises à jour.**

Yarn – Soumission d'un programme



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’Application Manager du ResourceManager prépare un container libre pour y exécuter l’Application Master du programme (son main).**
  - **L’application Master du programme est lancé; il s’enregistre immédiatement auprés 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’Application Master sur leur statut continuellement. En cas de problème, celui-ci peut demander de nouveaux containers au Resource Manager 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’Application Master directement.**

- **Une fois toutes les tâches exécutées, l’Application Master signale le ResourceManager et s’arrête.**

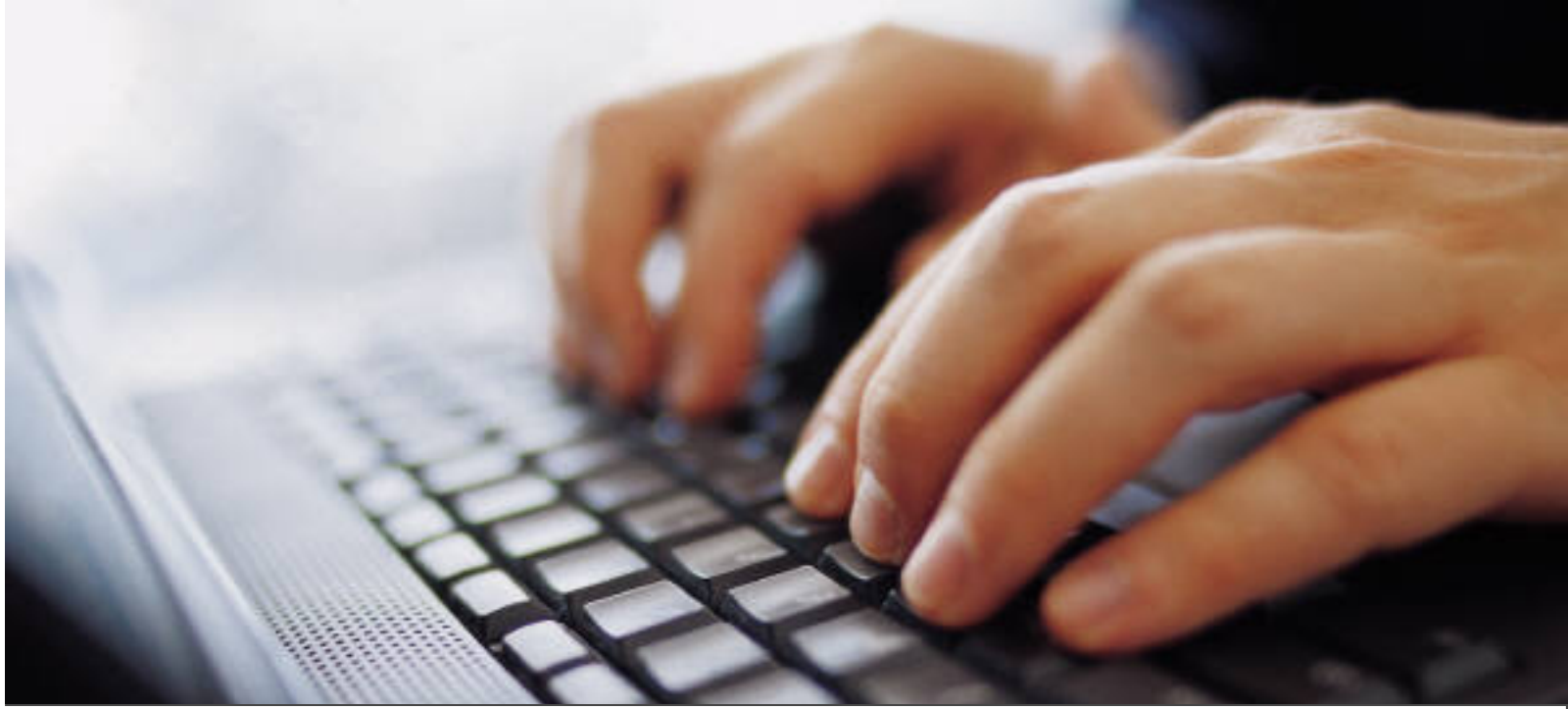
- **Le ResourceManager libère alors le container occupé restant, qui exécutait le code de l’Application Master.**

Yarn – Remarques

- **La communication directe entre l'Application Master et les Node Managers est un point de différence notable avec la version 1, et qui évite le goulot d'étranglement lié au JobTracker.**

- **D'une manière plus générale, Yarn est plus proche des moteurs d'exécution de « nouvelle génération », comme celui de Spark.**

- **En revanche, le ResourceManager reste un « single point of failure » : il est nécessaire d'assurer sa haute disponibilité via des mécanismes classiques de failover, comme pour le JobTracker dans la version 1 du moteur.**



6

Utilisation 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.**

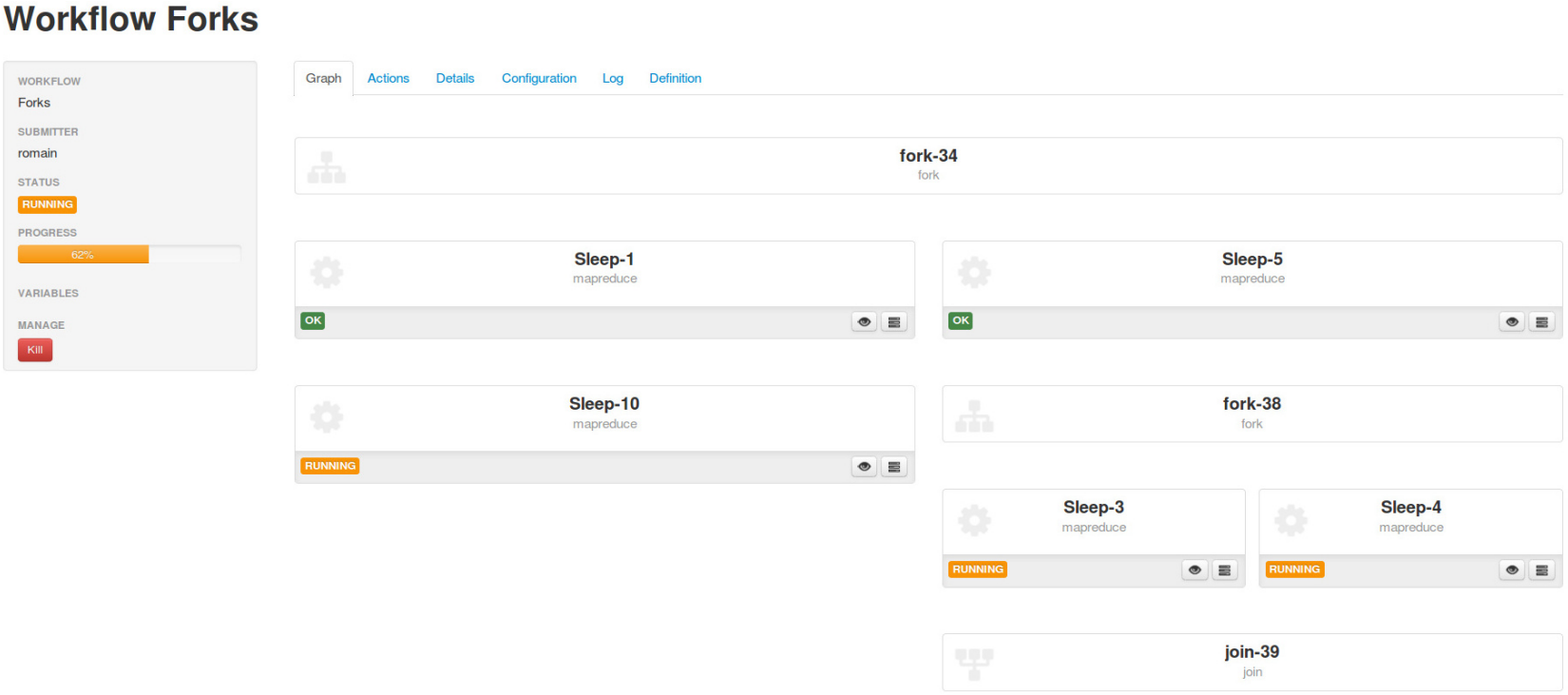
- **DataNode expose une interface web similaire, appelée par l'interface de NameNode, qui permet de lire les blocs des fichiers stockés (ou les fichiers stockés dans leur intégralité).**

- **Enfin, JobTracker offre une interface web également, qui permet de consulter les tâches en cours, d'avoir des statistiques sur l'exécution des tâches, de voir l'état des nodes TaskTracker et si oui ou non ils sont parvenus à exécuter les tâches qui leur ont été affecté, etc.**

Remarques

6-2

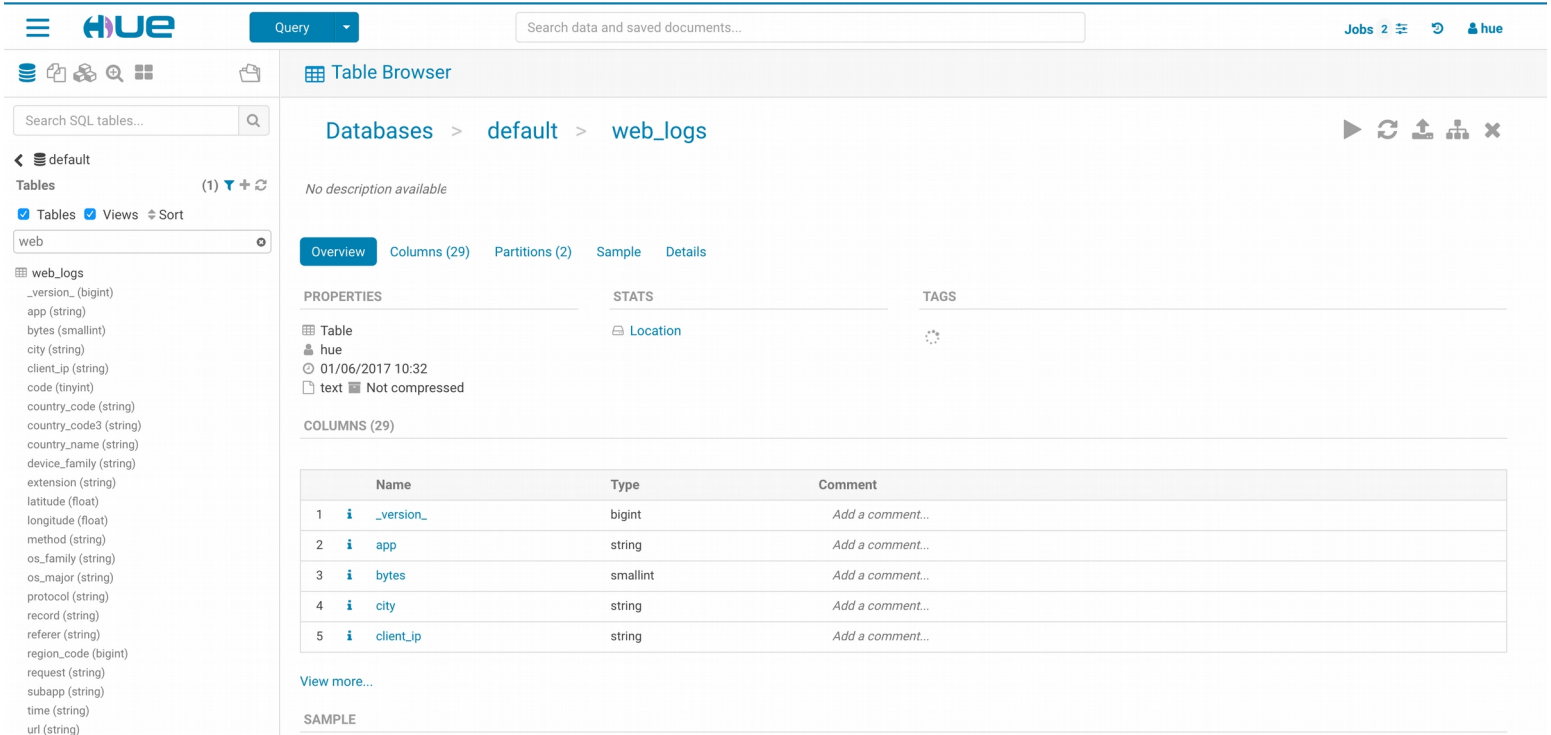
Par ailleurs, des interfaces utilisateur tierces ont été développées pour simplifier l'utilisation de Hadoop, par exemple le projet Hue (Open Source).



Remarques

6-3

Il est néanmoins vital de savoir de servir des commandes de base (qui pourraient par exemple être appelées depuis des scripts *shell*).



Programmation Hadoop

Comme indiqué précédemment, Hadoop est développé en Java. Les tâches MAP/REDUCE sont donc implémentables par le biais d'interfaces Java (il existe cependant des wrappers très simples permettant d'implémenter ses tâches dans n'importe quel langage). 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 clef/valeur utilisées, des classes se chargeant des opérations MAP et REDUCE, et des fichiers HDFS à utiliser pour les entrées/sorties.**

- **Une classe MAP (qui effectuera l'opération MAP).**

- **Une classe REDUCE (qui effectuera l'opération REDUCE).**

Programmation Hadoop – Classe Driver

La classe Driver contient le *main* de notre programme.

Au sein du *main()* en question, on va effectuer les opérations suivantes:

- **Créer un objet Configuration de Hadoop, qui est nécessaire pour permettre à Hadoop d'obtenir la configuration générale du *cluster*. L'objet en question pourrait aussi nous permettre de récupérer nous-même des options de configuration qui nous intéressent.**

- **Permettre à Hadoop de récupérer d'éventuels arguments génériques disponibles sur la ligne de commande (par exemple le nom du *package* de la tâche à exécuter si le .jar en contient plusieurs). On va également récupérer les arguments supplémentaires pour s'en servir; on souhaite que l'utilisateur puisse préciser le nom du fichier d'entrée et le nom du répertoire de sortie HDFS pour nos tâches Hadoop grâce à la ligne de commande.**

Programmation Hadoop – Classe Driver

- **Créer un nouvel objet Hadoop *Job*, qui désigne une tâche Hadoop.**
- **Utiliser cet objet *Job* pour informer Hadoop du nom de nos classes Driver, MAP et REDUCE.**
- **Utiliser le même objet pour informer Hadoop des types de données utilisés dans notre programme pour les couples (clef;valeur) MAP et REDUCE.**
- **Informer Hadoop des fichiers d'entrée/sortie pour notre tâche sur HDFS.**
- **Enfin, utiliser l'objet *Job* créé précédemment pour déclencher le lancement de la tâche *via* le cluster Hadoop.**

On reprend ici l'exemple du compteur d’occurrences de mots décrit précédemment.

Programmation Hadoop – Classe Driver

- Le prototype de notre fonction *main()*:

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

On se sert de *args* pour récupérer les arguments de la ligne de commande. Plusieurs fonctions Hadoop appelées au sein du *main* sont susceptibles de déclencher des exceptions – on l'indique donc lors de la déclaration.

- Avant toute chose, on créé dans notre main un nouvel objet Configuration Hadoop:

```
// Créé un objet de configuration Hadoop.
Configuration conf=new Configuration();
```

Le *package* à importer est:
`org.apache.hadoop.conf.Configuration`

Programmation Hadoop – Classe Driver

- **Ensuite, on passe à Hadoop les arguments de la ligne de commande pour lui permettre de récupérer ceux qui sont susceptibles de lui être adressés:**

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

On utilise pour ce faire un objet Hadoop *GenericOptionsParser*, dont le *package* est:
`org.apache.hadoop.util.GenericOptionsParser`

La fonction `getRemainingArgs()` de notre objet nous permet par ailleurs de récupérer les arguments non exploités par Hadoop au sein d'un tableau `ourArgs` – ce qui nous permettra de les utiliser par la suite.

On peut ainsi rendre paramétrable facilement une tâche Hadoop lors de l'exécution, par le biais des arguments de la ligne de commande.

Programmation Hadoop – Classe Driver

- **On créé ensuite un nouvel objet Hadoop Job:**

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

Le *package* à importer est le suivant:
`org.apache.hadoop.mapreduce.Job`

On passe au constructeur notre objet *Configuration*, ainsi qu'une description textuelle courte du programme Hadoop.

- **Ensuite, il faut indiquer à Hadoop – par le biais de l'objet *Job* nouvellement créé – quelles sont les classes Driver, Map et Reduce de notre programme Hadoop.**
Dans notre cas, il s'agira respectivement des classes `WCount`, `WCountMap` et `WCountReduce` du *package* `org.mbds.hadoop.wordcount`.

Programmation Hadoop – Classe Driver

On utilise pour ce faire les fonctions suivantes:

```
job.setJarByClass(WCount.class);
job.setMapperClass(WCountMap.class);
job.setReducerClass(WCountReduce.class);
```

- Il faut ensuite indiquer à Hadoop quels sont les types de données que l'ont souhaite utiliser pour les couples (clef;valeur) de nos opérations map et reduce. Dans le cas de notre compteur d'occurrences de mots, on souhaite utiliser des chaînes de caractères pour les clefs (nos mots) et des entiers pour nos occurrences.

  Remarque: on ne doit pas utiliser les types classiques *Int* et *String* Java pour désigner nos types, mais des classes qui leur correspondent et qui sont propres à Hadoop. Dans notre cas, les classes `IntWritable` et `Text`.

Programmation Hadoop – Classe Driver


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

Les *packages* des types en question:
`org.apache.hadoop.io.IntWritable` et `org.apache.hadoop.io.Text`
Il en existe beaucoup d'autres dans `org.apache.hadoop.io.*`.

A noter qu'on pourrait aussi décider d'utiliser un type de clef/valeur en sortie de la fonction reduce différent de celui utilisé en sortie de la fonction map; on utiliserait alors les fonctions explicites: `setMapOutputKeyClass` et `setMapOutputValueClass` (pour influencer les types en sortie de la fonction map).

Programmation Hadoop – Classe Driver

- **Ensuite, on doit indiquer où se situent nos données d'entrée et de sortie dans HDFS. On utilise pour ce faire les classes Hadoop `FileInputFormat` et `FileOutputFormat`.**
  Ces classes sont implémentées suivant un *design pattern* Singleton – il n'est pas nécessaire de les instancier dans le cas qui nous intéresse (dans des cas plus complexe, on étendra parfois la classe en question dans une nouvelle classe qui nous est propre).

Programmation Hadoop – Classe Driver

On procède de la manière suivante:

```
FileInputFormat.addInputPath(job, new Path(ourArgs[0]));
FileOutputFormat.setOutputPath(job, new Path(ourArgs[1]));
```

Les *packages* à utiliser:

`org.apache.hadoop.mapreduce.lib.input.FileInputFormat`
`org.apache.hadoop.mapreduce.lib.input.FileOutputFormat`
et
`org.apache.hadoop.fs.Path`

On utilise les arguments restants après ceux qu'a utilisé Hadoop. Le code est ici simplifié – on devrait en théorie vérifier la taille du tableau *ourArgs* pour éviter d'éventuelles erreurs.

Programmation Hadoop – Classe Driver

Enfin, il reste à lancer l'exécution de la tâche par le biais du *cluster* Hadoop. On procède ainsi:

```
if(job.waitForCompletion(true))
  System.exit(0);
System.exit(-1);
```

La fonction `waitForCompletion` de l'objet job va exécuter la tâche et attendre la fin de son exécution. Elle prend un argument: un booléen indiquant à Hadoop si oui ou non il doit donner des indications sur la progression de l'exécution à l'utilisateur sur la sortie standard (stdout).

Elle renvoie *true* en cas de succés; ici, on terminera l'exécution du programme en renvoyant 0 si tout s'est bien passé, et -1 en cas de problème (codes de retour unix standards).

Programmation Hadoop – Classe Driver

Le code complet de notre classe Driver:

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

import org.apache.hadoop.fs.Path;
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.conf.Configuration;
import org.apache.hadoop.util.GenericOptionsParser;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.io.IntWritable;

// Notre classe Driver (contient le main du programme Hadoop).
public class WCount
{
```

Programmation Hadoop – Classe Driver


```
// Le main du programme.
public static void main(String[] args) throws Exception
{
    // Créé un object de configuration Hadoop.
    Configuration conf=new Configuration();

    // Permet à Hadoop de lire ses arguments génériques,
     // récupère les arguments restants dans ourArgs.
    String[] ourArgs=new GenericOptionsParser(conf,
                          args).getRemainingArgs();

    // Obtient un nouvel objet Job: une tâche Hadoop. On
   // fourni la configuration Hadoop ainsi qu'une description
    // textuelle de la tâche.
    Job job=Job.getInstance(conf, "Compteur de mots v1.0");
```

Programmation Hadoop – Classe Driver


```
// Défini les classes driver, map et reduce.
job.setJarByClass(WCount.class);
job.setMapperClass(WCountMap.class);
job.setReducerClass(WCountReduce.class);

// Défini types clefs/valeurs de notre programme Hadoop.
job.setOutputKeyClass(Text.class);
job.setOutputValueClass(IntWritable.class);

// Défini les fichiers d'entrée du programme et le
 // répertoire des résultats. On se sert du premier et du
 // deuxième argument restants pour permettre à
 // l'utilisateur de les spécifier lors de l'exécution.
FileInputFormat.addInputPath(job, new Path(ourArgs[0]));
FileOutputFormat.setOutputPath(job, new Path(ourArgs[1]));
```

Programmation Hadoop – Classe Driver

6-17

```
        // On lance la tâche Hadoop. Si elle s'est effectuée
        // correctement, on renvoie 0. Sinon, on renvoie -1.
        if(job.waitForCompletion(true))
            System.exit(0);
        System.exit(-1);
    }
}
```

Programmation Hadoop – Classe MAP

La classe MAP va être en charge de l'opération MAP de notre programme.
Elle doit étendre la classe Hadoop `org.apache.hadoop.mapreduce.Mapper`.
Il s'agit d'une classe *générique* qui se paramétrise avec quatre types:

- **Un type *keyin*: le type de clef d'entrée.**
- **Un type *valuein*: le type de valeur d'entrée.**
- **Un type *keyout*: le type de clef de sortie.**
- **Un type *valueout*: le type de valeur de sortie.**

Le type *keyin* est notamment utile lorsqu'on utilise des fonctionnalités plus avancées, comme la possibilité d'effectuer plusieurs opérations MAP les unes à la suite des autres, auquel cas notre opération map recevra en entrée des couples (clef;valeur). Par défaut, cependant, la clef d'entrée lors de la lecture de fichiers texte par Hadoop est constituée du numéro de ligne.
Dans notre cas, nous n'utiliserons pas cette clef; on utilisera donc le type Java `Object` comme type *keyin*, ce qui évite un *cast* inutile.

Programmation Hadoop – Classe MAP

Dans notre exemple, notre classe Map sera déclarée ainsi:

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

On utilise ici comme types:

- **`Text` pour le type *valuein*, puisque notre valeur d'entrée à la fonction Map est une chaîne de caractères (une ligne de texte).**
- **`Text` pour le type *keyout*, puisque notre valeur de clef pour les couples (clef;valeur) de la fonction Map est également une chaîne de caractères (le mot dont on compte les occurrences).**
- **`IntWritable` pour le type *valueout*, puisque notre valeur pour les couples (clef;valeur) de la fonction Map est un entier (le nombre d'occurences).**

Ici aussi, on utilise les types Hadoop et non les types natifs Java (Int et String).

Programmation Hadoop – Classe MAP

Au sein de la classe `Mapper`, c'est la fonction `map` qui va s'occuper d'effectuer la tâche MAP. C'est la seule qu'on doit absolument implémenter.
Elle prends trois arguments: la clef d'entrée *keyin* (qu'on ignore dans notre cas), la valeur d'entrée *valuein* (la ligne de texte dont on souhaite compter les mots), et un `Context` Java qui représente un *handle* Hadoop et nous permettra de retourner les couples (clef;valeur) résultant de notre opération Map.

Le prototype de notre fonction map:

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

Comme pour la fonction `main`, la fonction `map` appellera des fonctions susceptibles de déclencher des exceptions (notamment concernant l'interruption de l'exécution Hadoop ou des problèmes d'accès HDFS) – on le précise donc dans sa déclaration.

Programmation Hadoop – Classe MAP

Au sein de la méthode map, on va donc effectuer la tâche MAP de notre programme MAP/REDUCE.
Dans le cadre de notre exemple, la fonction devra parcourir la ligne de texte fournie en entrée, et renvoyer un couple (clef;valeur) pour chacun des mots. Ce couple devra avoir pour clef le mot en question, et pour valeur l'entier « 1 ».

Dans la fonction map, afin d'indiquer à Hadoop qu'on souhaite renvoyer un couple (clef;valeur), on utilise la fonction `write` de notre objet `Context`. Elle peut être appelée autant de fois que nécessaire; une fois pour chacun des couples (clef;valeur) qu'on souhaite renvoyer. Par exemple:

```
context.write("ciel", 1);
```

Il faut évidemment que la clef et la valeur renvoyées ainsi correspondent aux types *keyout* et *valueout* de notre classe Mapper.

Programmation Hadoop – Classe MAP

Notre classe Map d'exemple en intégralité:

```
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;


// Notre classe MAP.
public class WCountMap extends Mapper<Object, Text, Text, IntWritable>
{
    // IntWritable contant de valeur 1.
    private static final IntWritable ONE=new IntWritable(1);
```

Programmation Hadoop – Classe MAP


```
    // La fonction MAP elle-même.
    protected void map(Object offset, Text value, Context context)
                        throws IOException, InterruptedException
    {
        // Un StringTokenizer va nous permettre de parcourir chacun des
        // mots de la ligne qui est passée à notre opération MAP.
        StringTokenizer tok=new StringTokenizer(value.toString(), " ");

        while(tok.hasMoreTokens())
        {
            Text word=new Text(tok.nextToken());
            // On renvoie notre couple (clef;valeur): le mot courant suivi
            // de la valeur 1 (définie dans la constante ONE).
            context.write(word, ONE);
        }
    }
}
```

Programmation Hadoop – Classe REDUCE

La classe REDUCE va être en charge de l'opération REDUCE de notre programme.

Elle doit étendre la classe Hadoop `org.apache.hadoop.mapreduce.Reducer`. Il s'agit là aussi d'une classe *générique* qui se paramétrise avec les mêmes quatre types que pour la classe `Mapper`: *keyin*, *valuein*, *keyout* et *valueout*.

On rappelle que l'opération REDUCE recevra en entrée une clef unique, associée à toutes les valeurs pour la clef en question.

Dans le cas du compteur d'occurrences de mots, on recevra en entrée une valeur unique pour la clef, par exemple « ciel », suivi de toutes les valeurs qui ont été rencontrées à la sortie de l'opération MAP pour la clef « ciel » (par exemple cinq fois la valeur « 1 » si le mot « ciel » était présent cinq fois dans notre texte d'exemple).

Programmation Hadoop – Classe REDUCE

Dans notre exemple, la classe Reduce sera définie comme suit:

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

On utilise pour chacun des types paramétrables:

- **`Text` pour *keyin*: il s'agit de notre clef unique d'entrée – le mot concerné.**
- **`IntWritable` pour *valuein*: le type de nos valeurs associées à cette clef (le nombre d'occurences, un entier).**
- **`Text` pour *keyout*: le type de clef de sortie. Nous ne modifierons pas la clef, il s'agira toujours du mot unique concerné – on utilise donc Text.**
- **`Text` pour *valueout*: le type de valeur de sortie. On utilise ici Text – on renverra le nombre total d'occurrences pour le mot concerné sous la forme d'une chaîne de caractères (on pourrait également utiliser IntWritable ici).**

Là aussi, on utilise les types de données propres à Hadoop.

Programmation Hadoop – Classe REDUCE

Au sein de la classe `Reducer`, c'est la fonction `reduce` qui va effectuer l'opération REDUCE. C'est la seule qu'on doive implémenter.

Elle prends trois arguments: la clef concernée, un `Iterable` java (une liste) de toutes les valeurs qui lui sont associées et qui ont été renvoyées par l'opération MAP, et enfin un objet `Context` java similaire à celui de la fonction `map` de la classe `Mapper`, et qui nous permettra de renvoyer notre valeur finale, associée à la clef.

Dans notre exemple, la déclaration de la fonction reduce:

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

Comme pour map, la fonction fait appel à des fonctions Hadoop susceptibles de provoquer des exceptions – on l'indique ici aussi.

Programmation Hadoop – Classe REDUCE

Au sein de la fonction `Reduce`, on pourra renvoyer un couple (clef;valeur) en résultat exactement de la même manière que pour la fonction `map`, par le biais d'un appel à la fonction `write` de notre objet `Context`.

Par exemple:

```
context.write("ciel", "5 occurences");
```

Contrairement à la fonction `map`, en revanche, on cherche à ne produire qu'une et une seule valeur de retour pour la clef concernée. On n'appellera donc la fonction `write` qu'une seule fois.

Remarque: en théorie et dans des cas plus complexes, on pourrait là aussi appeler la fonction à plusieurs reprises, pour renvoyer plusieurs couples (clef;valeur). En revanche, Hadoop n'appliquera aucun traitement dessus et les considérera simplement comme plusieurs résultats finals.

Programmation Hadoop – Classe REDUCE

Notre classe Reduce d'exemple en intégralité:

```
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;


// Notre classe REDUCE - paramétrée avec un type Text pour la clef, un
// type de valeur IntWritable, et un type de retour (le retour final de
// la fonction Reduce) Text.
public class WCountReduce extends Reducer<Text, IntWritable, Text, Text>
{
```

Programmation Hadoop – Classe REDUCE


```
    // La fonction REDUCE elle-même. Les arguments: la clef key, un
    // Iterable de toutes les valeurs qui sont associées à la clef en
    // question, et le contexte Hadoop (un handle qui nous permet de
    // renvoyer le résultat à Hadoop).
    public void reduce(Text key, Iterable<IntWritable> values,
                       Context context)
                       throws IOException, InterruptedException
    {
        // Pour parcourir toutes les valeurs associées à la clef fournie.
        Iterator<IntWritable> i=values.iterator();
        int count=0;  // Notre total pour le mot concerné.
        while(i.hasNext())   // Pour chaque valeur...
            count+=i.next().get();    // ...on l'ajoute au total.
        // On renvoie le couple (clef;valeur) constitué de notre clef key
        // et du total, au format Text.
        context.write(key, new Text(count+" occurences."));
    }
}
```

Remarques

- **Hadoop est une plate-forme récente, en développement constant: l'API Java change régulièrement. Il faut toujours s'informer sur les dernières modifications afin de rester à jour des dernières modifications. Par ailleurs, la distribution Hadoop comprend de nombreux exemples de programmes Map/Reduce, mis à jour afin d'utiliser l'API la plus récente.**

- **Même si l'API change régulièrement, les anciennes manières de procéder restent généralement disponibles pendant plusieurs mois après une mise à jour.**

- **Enfin, il existe de nombreuses autres possibilités offertes par l'API: extraire des paramètres de configurations globaux du *cluster*, implémenter sa propre classe FileInputFormat pour des traitements plus complexes sur les données d'entrée, etc... d'une manière générale, se référer à la documentation de l'API Hadoop:**

https://hadoop.apache.org/docs/

Compilation

- **Pour compiler un programme Hadoop, il suffit d'utiliser le compilateur `javac` comme pour tout autre programme Java.**

- **Il est cependant nécessaire d'inclure les librairies appropriées. Elles sont disponibles avec les distributions de Hadoop, sous la forme de fichiers .jar. Les librairies principales à utiliser:**

  - **`hadoop-common.jar`**
  - **`hadoop-mapreduce-client-core.jar`**
  - **`hadoop-mapreduce-client-common.jar`**
  - **`commons-cli.jar`**

  … il en existe d'autres pour des cas d'utilisation plus complexes, à inclure en fonction des besoins.

- **Enfin, après compilation, on *package* le programme Hadoop à l'intérieur d'un fichier .jar pour son exécution.**

Exécution

- **Pour exécuter un programme Hadoop, on utilise là aussi le programme en ligne de commande `hadoop`. La syntaxe est la suivante:**

```
hadoop jar [JAR FILE] [DRIVER CLASS] [PARAMETERS]
```

Par exemple pour notre compteur d'occurrences de mots:

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

… la commande demandera à Hadoop d'exécuter le programme Hadoop de la classe Driver `org.mbds.hadoop.wordcount` du fichier `wcount_mbds.jar`, en lui indiquant qu'il doit lire son texte dans le fichier « input/poeme.txt » sur HDFS, et stocker les résultats dans le répertoire « results » sur HDFS.

Remarques

- **Hadoop stocke les résultats dans une série de fichiers `part-r-xxxx`, où `xxxx` est un compteur incrémental. L'idée est ici de stocker de grandes quantités de résultats dans de nombreux fichiers différents, qui seront en conséquences distribués sur tout le *cluster* HDFS. Le nom du fichier est paramétrable dans la configuration Hadoop.**

- **On a un fichier « part-r » par opération REDUCE exécutée. Le « r » au sein du nom signifie « Reduce ». On peut également demander à Hadoop d'effectuer uniquement les opérations MAP (par exemple pour du *debug*), auquel cas on aura une série de fichiers « part-m ».**

- **Un fichier _SUCCESS (vide) est également créé dans le répertoire des résultats en cas de succès. Cela permet de contrôler que tout s'est bien passé rapidement (avec un simple `hadoop fs -ls` sur HDFS).**

Autres langages: streaming

Au delà de Java, Hadoop permet également l'exécution d'un programme Hadoop écrit dans d'autres langages, par exemple en C, en Python ou encore en *bash* (shell scripting).

Pour ce faire, un outil est distribué avec Hadoop: streaming. Il s'agit d'un .jar qui est capable de prendre en argument des programmes ou scripts définissant les tâches MAP et REDUCE, ainsi que les fichiers d'entrée et le répertoire de sortie HDFS, et d'exécuter ainsi la tâche spécifiée sur le *cluster*.

Ce .jar est disponible dans le répertoire d'installation Hadoop et porte le nom `hadoop-streaming-VERSION.jar`, où `VERSION` est la version de Hadoop concernée.

Il s'agit en réalité d'un programme Hadoop Java « classique », mais qui appelle les tâches MAP et REDUCE par le biais du système.

Streaming - MAP

Lorsqu'on développe un programme MAP Hadoop dans un autre langage pour son utilisation avec l'outil streaming, les données d'entrée doivent être lues sur l'entrée standard (`stdin`) et les données de sorties doivent être envoyées sur la sortie standard (`stdout`).

- **En entrée du script ou programme MAP, on aura une série de lignes: nos données d'entrée (par exemple dans le cas du compteur d'occurrences de mots, des lignes de notre texte).**

- **En sortie du script ou programme MAP, on doit écrire sur `stdout` notre série de couples (clef;valeur) au format:**

```
CLEF[TABULATION]VALEUR
```

… avec une ligne distincte pour chaque (clef;valeur).

Streaming - REDUCE

Lorsqu'on développe un programme REDUCE Hadoop dans un autre langage pour son utilisation avec l'outil streaming, les données d'entrée et de sortie doivent être là aussi lues/écrites sur `stdin` et `stdout` (respectivement).

- **En entrée du script ou programme REDUCE, on aura une série de lignes: des couples (clef;valeur) au format:**
  `CLEF[TABULATION]VALEUR`
  Les couples seront triés par clef distincte, et la clef 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 !

- **En sortie du script ou programme REDUCE, on doit écrire des couples (clef;valeur), toujours au format `CLEF[TABULATION]VALEUR`.**
  Remarque: d'ordinaire, on écrira évidemment un seul couple (clef;valeur) par clef distincte.

Exemple (Python)

Occurences de mots version Python – opération MAP:

```
import sys

# Pour chaque ligne d'entrée.
for line in sys.stdin:
  # Supprimer les espaces autour de la ligne.
  line=line.strip()
  # Pour chaque mot de la ligne.
  words=line.split()
  for word in words:
    # Renvoyer couple clef;valeur: le mot comme clef, l'entier "1" comme
    # valeur.
    # On renvoie chaque couple sur une ligne, avec une tabulation entre
    # la clef et la valeur.
    print "%s\t%d" % (word, 1)
```

Exemple (Python)

Occurences de mots version Python – opération REDUCE:

```
import sys

total=0;  # Notre total pour le mot courant.
# Contient le dernier mot rencontré.
lastword=None

# Pour chaque ligne d'entrée.
for line in sys.stdin:
  # Supprimer les espaces autour de la ligne.
  line=line.strip()

  # Récupérer la clef et la valeur, convertir la valeur en int.
  word, count=line.split('\t', 1)
  count=int(count)
```

Exemple (Python)


```
  # On change de mot (test nécessaire parce qu'on est susceptible d'avoir
  # en entrée plusieurs clefs distinctes différentes pour une seule et
  # même exécution du programme - Hadoop triera les couples par clef
  # distincte).
  if word!=lastword and lastword!=None:
    print "%s\t%d occurences" % (lastword, total)
    total=0;
  lastword=word

  total=total+count  # Ajouter la valeur au total

# Ecrire le dernier couple (clef;valeur).
print "%s\t%d occurences" % (lastword, total)
```

Exemple (Bash)

Occurences de mots version Bash – opération MAP:

```
# Pour chaque ligne d'entrée.
while read line; do
  # Supprimer les espaces autour de la ligne.
  line=$(echo "$line"|sed 's/^\s*\(.\+\)\s*$/\1/')
  # Pour chaque mot de la ligne.
  for word in $line; do
    # Renvoyer couple clef;valeur: le mot comme clef, l'entier "1" comme
    # valeur.
    # On renvoie chaque couple sur une ligne, avec une tabulation entre
    # la clef et la valeur.
    echo -e $word"\t"1
  done
done
```

Exemple (Bash)

Occurences de mots version Bash – opération REDUCE:

```
lastword=""
total=0

# Pour chaque ligne d'entrée.
while read line; do
  # Supprimer les espaces autour de la ligne.
  line=$(echo "$line"|sed 's/^\s*\(.\+\)\s*$/\1/')

  # Recuperer mot et occurrence (IE, clef et valeur)
  word=$(echo "$line"|awk -F "\t" '{print $1}');
  nb=$(echo "$line"|awk -F "\t" '{print $2}');
```

Exemple (Bash)


```
  # On change de mot (test nécessaire parce qu'on est susceptible d'avoir
  # en entrée plusieurs clefs distinctes différentes pour une seule et
  # même exécution du programme - Hadoop triera les couples par clef
  # distincte).
  if [ "$word" != "$lastword" ] && [ "$lastword" != "" ]; then
    echo -e $lastword"\t"$total" occurences."
    total=0;
  fi
  lastword="$word"

  total=$[ $total + $nb ]   # Ajouter la valeur au total.
done

# Ecrire le dernier couple (clef;valeur).
echo -e $lastword"\t"$total" occurences."
```

Exécution

Streaming est un programme Hadoop standard, on l'exécutera donc avec la même commande `hadoop jar` qu'un programme Hadoop Java habituel. C'est dans ses options de ligne de commande qu'on va indiquer les scripts ou programmes Hadoop map et reduce à utiliser. Syntaxe:

```
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
```

Après quoi la tâche Hadoop s'exécutera exactement comme une tâche standard.