FUN FRANCE UNIVERSITÉ NUMÉRIQUE

Université Nice Sophia Antipolis

Introduction à Hadoop & MapReduce

Cours 1

Benjamin Renaut <renaut.benjamin@tokidev.fr>

MOOC / FUN

2014 - 2015

1 Introduction: le paradigme Map/Reduce

Plan du cours

- **Deux cours de ~1h chacun.**
- **Accompagné d'exercices / quizzs pour validation.**
- **Deux sujets abordés:**
  - **Paradigme Map/Reduce; divers exemples pour assimiler la méthode.**
  - **Apache Hadoop: installation et utilisation; programmation Map/Reduce pour Hadoop.**
- **Constitue une introduction. Se référer au guide de lecture pour approfondir le sujet.**

Map/Reduce

- **Map/Reduce: un *paradigme* (une méthode) pour le développement logiciel.**
- **Trouve ses origines dans les méthodes map et reduce/fold des langages fonctionnels.**
- **Basé sur trois opérations principales:**
  - **map: une fonction appliquée à chaque partie des données d'entrée, pour générer une série de couples (clef;valeur).**
  - **shuffle: le regroupement des couples (clef;valeur) par clef distincte.**
  - **reduce: une autre fonction appliquée à chacun des groupes ainsi généré (donc par clef distincte).**

Map/Reduce

- **Méthodologie qui, sans être réellement « nouvelle », a été formalisée pour la première fois au sein d'un article de Google en 2004:**

  MapReduce: Simplified Data Processing on Large Clusters
  https://static.googleusercontent.com/media/research.google.com/en//archive/mapreduce-osdi04.pdf

- **Intérêt de l'approche: permet, du moment que la méthodologie est suivie, de rendre parallélisable l'exécution du programme sur autant de machines que nécessaire; et par conséquent de traiter des volumes de données aussi larges que nécessaire (plus de données, plus de machines).**

- **Il suffit de développer deux fonctions simples (map et reduce), pour rendre un programme parallélisable et pour l'exécuter sur des volumes même très importants.**

Pourquoi Map/Reduce

- **De plus en plus de données générées, par de plus en plus de systèmes. Traiter des volumes de données sans cesse grandissant nécessite une forte puissance de calcul.**

- **La cadence des processeurs stagne depuis le début des années 2000; l'évolution se tourne désormais vers la multiplication des coeurs et le parallélisme.**

- **Là où le développement de programmes parallélisables restait autrefois une exception/un besoin spécialisé, c'est désormais un besoin croissant et répandu.**

- **C'est là la motivation à l'origine de Map/Reduce: formaliser une méthodologie commune, encadrée, pour le développement et l'exécution facile de programmes parallélisables pour le traitement de volumes de données de plus en plus massifs.**

Pourquoi Map/Reduce

- **Map/Reduce est donc une révolution nécessaire dans le développement logiciel par rapport à des besoins qui évoluent, et à des contraintes matérielles.**

- **Evolution portée aussi bien par des grands groupes privés (Google, Yahoo) que par des fondations du logiciel libre ou de l'Open Source (fondation Apache).**

- **Principale implémentation d'un *framework* de développement et d'exécution de tâches Map/Reduce: Hadoop, projet de la fondation Apache.**

Quand utiliser Map/Reduce / Hadoop

- **On utilise Hadoop / Map/Reduce quand:**
  - **On souhaite travailler sur des volumes de données massifs (de l'ordre du GB/TB).**

    et/ou

  - **On souhaite résoudre un problème complexe, très consommateur de ressources processeur, rapidement (plus rapidement que sur une machine unique).**

… et d'une manière générale lorsqu'un problème *doit* être parallélisé car son exécution sur une machine unique n'est pas réaliste (trop lente/impossible).

Quand utiliser Map/Reduce / Hadoop

- **On n'utilise *pas* cette approche si:**
  - **Le volume de données reste limité, et l'exécution sur une machine unique est viable (dans ce cas, utiliser Hadoop / map/reduce peut même être contre-productif !).**
  - **Les données d'entrée ne sont pas segmentables / découpables efficacement en plusieurs fragments pour la parallélisation.**
- **Le second critère est important: impossibilité de paralléliser efficacement une tâche *via* map/reduce si on ne peut pas facilement distribuer des fragments des données d'entrée à plusieurs machines; autrement dit si la tâche ne peut pas être effectuée indépendamment par des machines différentes sans se référer aux données traitées par les autres machines. Dans les faits, ce type de problème constitue une minorité.**

2 Map/Reduce: explications et exemple (1)

La méthodologie Map/Reduce

- **Basée sur la stratégie algorithmique du *divide and conquer*: découper un problème complexe en plusieurs problèmes simples.**

- **En découpant les données d'entrée en plusieurs fragments distincts, et en développement un couple de tâches simples qui seront exécutées sur ces fragments séparément, on pourra ainsi facilement paralléliser l'exécution (chaque machine exécute chaque tâche sur un ensemble de fragments d'entrée).**

La méthodologie Map/Reduce

- **On distingue 4 étapes dans l'exécution d'un programme Map/Reduce:**
  - **Le découpage des données d'entrée (opération *split*).**
  - **L'opération map, une fonction exécutée indépendamment sur chacun de ces fragments, et qui génère une série de couples (clef;valeur).**
  - **Le regroupement de tous les couples (clef;valeur) ainsi générés par clef distincte (opération *shuffle*).**
  - **L'opération reduce, une fonction exécutée indépendamment sur chacun de ces groupes associés à une clef distincte, et produisant elle aussi une série de couples (clef;valeur).**
- **Conceptuellement, les données d'entrée elles-même sont considérées comme des couples (clef;valeur). D'une manière plus générale, toutes les tâches ainsi exécutées (map, reduce, shuffle) reçoivent de tels couples en entrée et produisent de tels couples en sortie.**

La méthodologie Map/Reduce

- **Du point de vue du développeur, les étapes nécessaires au développement d'une tâche map/reduce consistent uniquement à:**
  - **Déterminer une manière efficace de fragmenter les données d'entrée.**
  - **Développer la fonction map.**
  - **Développer la fonction reduce.**
- **… c'est le framework de développement et d'exécution (Hadoop ou alternative) qui s'occupe de tout le reste:**
  - **L'opération shuffle.**
  - **Les impératifs liés au *clustering*: accès concurrent aux données, redondance, assignation des tâches aux différents membres du *cluster*, etc.**

Exemple: compteur d'occurences de mots

- **L'approche map/reduce constitue une manière sensiblement différente d'aborder un problème; elle est plus facile à appréhender par l'exemple.**

- **Premier exemple simple: un compteur d'occurences de mots (le *Hello, World* du Map Reduce).**

- **On cherche à énumérer tous les mots distincts d'une source textuelle, avec pour chacun d'entre eux le nombre de fois qu'ils sont présents au sein de la source.**

- **On imagine qu'on souhaite exécuter le problème sur un fort volume de données (par exemple l'intégralité des livres d'une bibliothèque nationale): il est donc nécessaire d'avoir une approche map/reduce pour facilement paralléliser l'exécution.**

Exemple: compteur d'occurences de mots

- **Données d'exemple:**

```
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)

- **On commence par découper les données d'entrée, de telle sorte que la fragmentation résulte en des blocs de données susceptibles d'être traités indépendamment sans influence sur le résultat final.**

- **Ici, la fragmentation est simple: on prend chacune des lignes du texte d'entrée comme un « bloc » de données. On en profite ici pour « nettoyer » également le texte (tout en minuscules, suppression des accents et de la ponctuation).**

Exemple: compteur d'occurences de mots

- **Données d'exemple:**

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

(on considere conceptuellement ces 4 blocs de données d'entrée comme 4 couples (clef;valeur) pour lesquels la clef est nulle et la valeur est constituée de la ligne de texte)

- **Notre opération *map* va produire une série de couples (clef;valeur) – et ces couples seront lors de l'opération suivante (*shuffle*) regroupés par clef distincte.**

- **Notre opération *reduce* aura quand à elle pour but de *réduire* ces groupes triés par clef distincte; le choix de la clef à utiliser est donc crucial.**

Exemple: compteur d'occurences de mots

- **Ici, on génère comme clef au sein de l'opération *map* le mot lui-même; et comme valeur correspondante, la constante « 1 » (quelque soit le mot rencontré).**

- **Le rôle de la fonction *map* sera donc de parcourir la ligne d'entrée, et pour chaque mot distinct de générer un couple (clef;valeur): (*mot*;1).**

```
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: compteur d'occurences de mots

- **Aprés exécution de *map* et de *shuffle*, les couples (clef;valeur) ainsi générés seront regroupés (par le *framework* d'exécution du programme sur le *cluster*, par exemple Hadoop) par clef distincte:**

(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)

- **Chacun de ces groupes distincts sera passé en entrée de la fonction *reduce*.**

Exemple: compteur d'occurences de mots

- **Quand à notre opération *reduce*, elle va recevoir un groupe de couples (clef;valeur) en entrée: ceux qui correspondent à une des clefs distinctes.**

- **Son rôle va simplement être de conserver la clef unique, d'additionner les valeurs de tous les couples (clef;valeur) reçus en entrée, et générer un unique couple (clef;valeur) en sortie, composé de la clef unique et du total obtenu.**

(qui;1) (qui;1) (qui;1) (qui;1) → **(qui;4)**

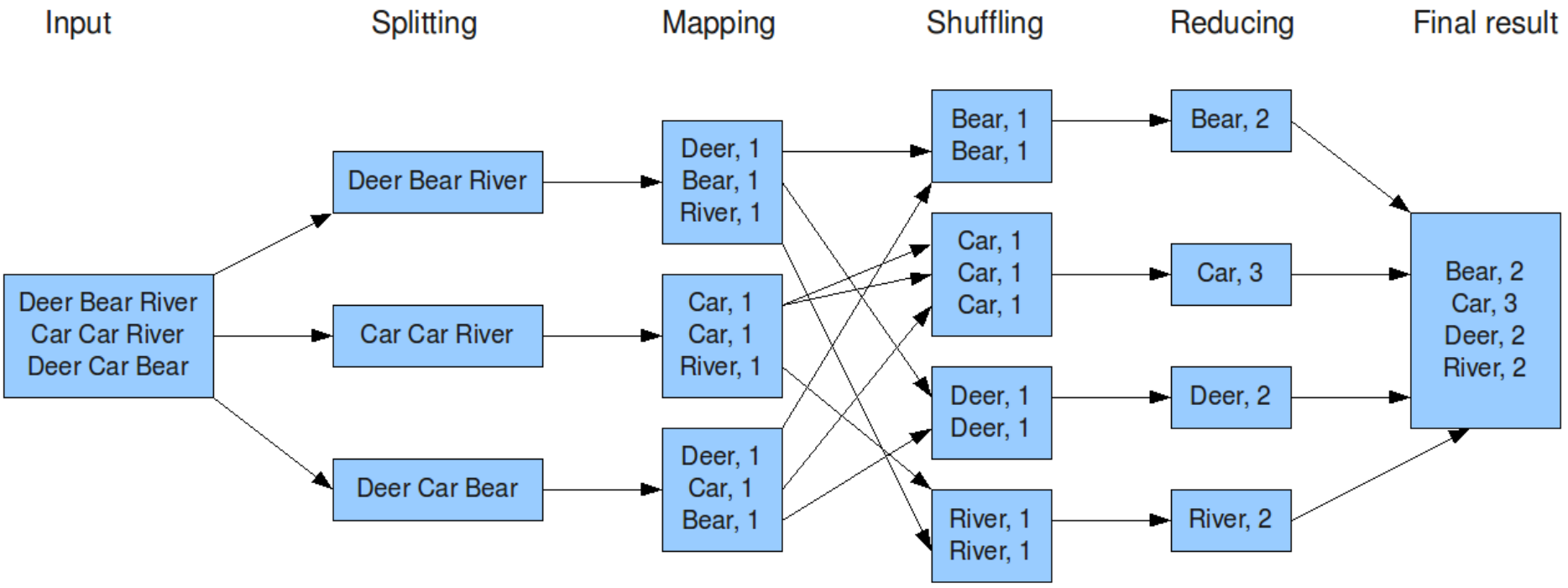
Exemple: compteur d'occurences de mots

- **Au terme de l'exécution du programme map reduce dans son ensemble, on obtiendra une série de couples (clef;valeur): un pour chacun des mots uniques présents dans le texte. Pour chacune de ces clefs, on aura le nombre d'occurences du mot dans le texte.**

- **Comme le programme a été implémenté à partir de la méthodologie map/reduce, il est parallélisable; et pourrait si on dispose d'assez de machines s'exécuter en quelques secondes même sur un texte d'entrée très volumineux.**

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

Schémas général



(source: documentation Hadoop)

3 Map/Reduce: explications et exemple (2)

Exemple: graphe social

- **Second exemple: on dispose de la base de données d'un réseau social contenant plusieurs millions d'utilisateurs. Pour chacun d'entre eux, on a une liste d'autres utilisateurs: les amis de l'utilisateur courant sur le réseau.**

- **On cherche à générer, pour chaque couple d'utilisateurs distincts, la liste des amis qu'ils ont en commun.**

- **On ne peut pas effectuer cette opération par le biais d'une requête sur la base de données relationnelle sans un impact immense sur le serveur du réseau social, potentiellement bloquant pour la base, et donc pour le site lui-même.**

- **Par conséquent, on va créer une tâche map/reduce pour régler le problème, et l'exécuter à intervalles réguliers sur un export de la base de données relationnelle; on imagine ensuite que les résultats seront ré-importés dans une table dédiée à cet usage.**

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
```

- **Là aussi, on peut fragmenter les données d'entrée par « ligne »/par utilisateur.**

Exemple: graphe social

- **Comme clef, on va choisir une chaîne de caractère, sous la forme:**

  « A-B »

  … cette clef, par exemple, désignera « les amis en commun des utilisateurs A et B ».

- **Le rationnel derrière ce choix vient du fait que la clef représente l'élément autour duquel sont regroupés les données lors de l'opération intermédiaire (*shuffle*); elle correspond au point commun entre les couples (clef;valeur) qui va nous permettre de *réduire* les données lors de l'opération *reduce*.**

- **Par ailleurs, on fera également en sorte que la clef soit toujours triée par ordre alphabétique (clef « B-A » sera exprimée sous la forme « A-B »).**

Exemple: graphe social

- **L'opération *map* est au coeur de la résolution du problème. Elle va prendre la liste des amis fournie en entrée, et va générer toutes les clefs distinctes possibles à partir de cette liste. Pour chacune de ces clef, elle va générer un couple (clef;valeur) dont la valeur sera la liste d'amis, telle quelle.**

- **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.**

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

- **Le pseudo-code de l'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])
```

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, l'étape de *shuffle* 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

- **… notre opération *reduce* se révèle par conséquent évidente: elle va recevoir chacun de ces groupes de deux couples (clef;valeur), parcourir les deux listes d'amis pour générer une liste d'amis communs, et renvoyer un seul couple (clef;valeur), dont la clef sera identique à la clef distincte correspondant au groupe d'entrée, et la valeur sera cette liste d'amis communs. 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

- **Les deux opérations *map* et *reduce* sont sommes toutes relativement simples, mais combinées à un *framework* d'exécution map reduce, elles effectuent un travail complexe.**

- **Par ailleurs, la tâche est parallélisable: on peut l'exécuter aussi vite que nécessaire du moment qu'on a assez de machines au sein du *cluster* d'exécution.**

- **Dans le cadre de l'exemple, on imagine que ce type de tâche serait exécuté toutes les heures sur un cluster Hadoop dédié à cet effet; permettant toutes les heures de mettre à jour la table relationnelle « classique » contenant les listes d'amis en communs avec un coût inexistant pour la base de données relationnelle.**

4 Map/Reduce: explications et exemple (3)

Exemple: anagrammes

- **Autre exemple: à partir d'une liste de mots, on cherche à déterminer lesquels sont des anagrammes.**

- **Données d'entrée:**

```
crane
imaginer
surface
metropole
migraine
structure
ancre
```

- **Là aussi, le choix de la clef est crucial.**

Exemple: anagrammes

- ***map*: renvoie un couple (clef;valeur), avec le mot pour valeur et les lettres du mot ordonnées dans l'ordre alphabétique pour clef.**

- **La clef constitue de fait un « point commun » aux anagrammes: les lettres elles-même, ordonnées. Le *shuffle* devrait donc regrouper les anagrammes ensemble.**

- **Aprés execution:**

```
("acenr";"crane")
("aegiimnr";"imaginer")
("acefrsu";"surface")
("eelmooprt";"metropole")
("aegiimnr";"migraine")
("cerrsttuu";"structure")
("acenr";"ancre")
```

Exemple: anagrammes

- **Aprés l'étape de *shuffle*:**

```
("acenr";"crane"), ("acenr";"ancre")
("aegiimnr";"imaginer"), ("aegiimnr";"migraine")
("acefrsu";"surface")
("eelmooprt";"metropole")
("cerrsttuu";"structure")
```

Exemple: anagrammes

- ***reduce*: concatène toutes les valeurs d'entrée pour la clef unique; renvoie un couple (clef;valeur) avec la clef d'entrée pour clef et la chaîne ainsi concaténée pour valeur.**

- **Aprés execution:**

```
("acenr";"crane ancre")
("aegiimnr";"imaginer migraine")
("acefrsu";"surface")
("eelmooprt";"metropole")
("cerrsttuu";"structure")
```

Exemple: sentiment client/twitter

- **Autre exemple: une entreprise dispose d'un compte twitter pour son service aprés vente, recevant plusieurs dizaines de milliers de tweets par jour. Elle cherche à déterminer le taux de satisfaction de ses clients à partir du compte twitter.**

- **Chaque heure, les *tweets* reçus sont exportés au sein d'un fichier texte. Données d'entrée:**

```
"@acme Votre service client est nul"
"@acme 30min d'attente... très insatisfait"
"Très satisfait par un produit super !! @acme"
"merci d'avoir RT @acme"
"@acme produit déjà cassé, super insatisfait !"
```

Exemple: sentiment client/twitter

- **Clef: un descripteur de sentiment client (« satisfait », « insatisfait » ou « attente », « inconcluant »).**
- ***map*: génère un couple (clef;valeur) par sentiment client détecté (mot correspondant à une liste prédéfinie).**
- **Si deux sentiments contradictoires détectés: renvoyer inconcluant.**
- **On renvoie un couple (clef;valeur) pour chaque fragment des données d'entrée: chaque *tweet***

Exemple: sentiment client/twitter

- **Pseudo code:**

```
WORDS_BAD = ["nul", "insatisfait", "bof", "incompétents", …]
WORDS_GOOD = ["satisfait", "super", "excellent", …]
BAD=0; GOOD=0
POUR MOT dans [TWEET], FAIRE:
  SI MOT PRESENT_DANS WORDS_BAD:
    BAD=1
  SINON SI MOT PRESENT_DANS WORDS_GOOD:
    GOOD=1
SI BAD==1 ET GOOD==0:
  RENVOYER("insatisfait",1)
SINON SI BAD==0 ET GOOD=1:
  RENVOYER("satisfait",1)
SINON:
  RENVOYER("inconcluant",1)
```

Exemple: sentiment client/twitter

- **Après exécution:**

```
"@acme Votre service client est nul"
"@acme 30min d'attente... très insatisfait"
"Très satisfait par un produit super !! @acme"
"merci d'avoir RT @acme"
"@acme produit déjà cassé, super insatisfait !"
```

```
("insatisfait";1)
("insatisfait";1)
("satisfait";1)
("inconcluant";1)
("inconcluant";1)
```

Exemple: sentiment client/twitter

- ***reduce*: additionne les valeurs associées à la clef unique; renvoie le total pour valeur (identique au *reducer* du compteur d'occurences de mots).**

- **Après exécution:**

```
("insatisfait";2)
("satisfait";1)
("inconcluant";2)
```

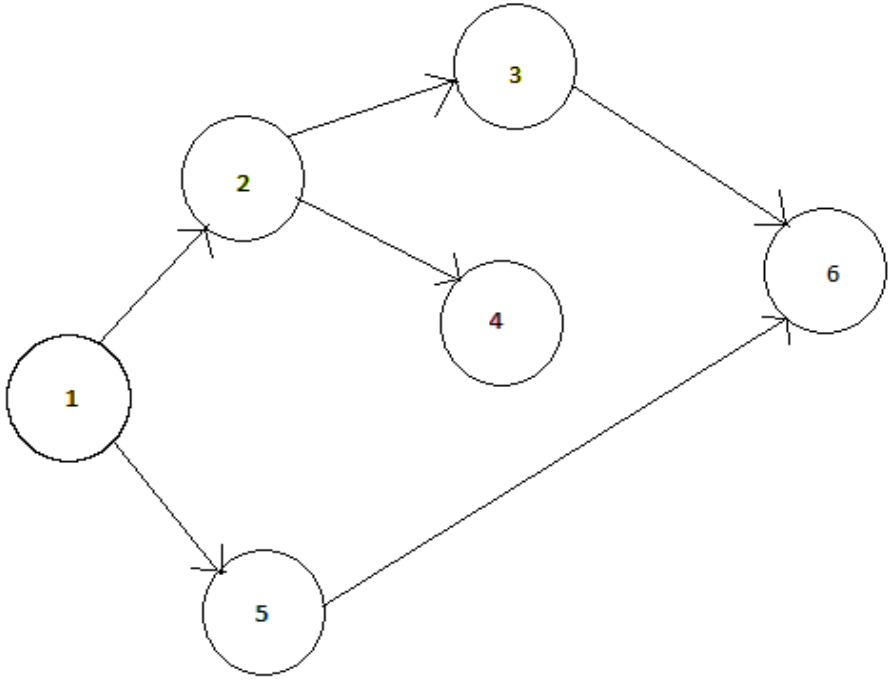
- **Conclusion: lors de la dernière heure, 33% de *tweets* exprimant de la satisfaction.**

- **En important, heure par heure, les données au sein d'une base de données relationnelle, on pourra obtenir en permanence des statistiques sur la satisfaction à partir de *twitter*.**

Exemple: parcours de graphe

- **On cherche à déterminer la profondeur maximale de tous les nœuds d'un graphe à partir d'un nœud de départ (via *breadth-first search*, parcours en largeur).**

- **Le graphe:**



Exemple: parcours de graphe

- **Données d'entrée:**

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

… avec valeur sur le modèle « NOEUDS_FILS|COULEUR|PROFONDEUR ».

- **Et couleur ayant pour valeur « BLANC » pour un nœud non parcouru, « GRIS » pour un nœud en cours de parcours, et « NOIR » pour un nœud déjà parcouru. Dans nos données d'entrée, l'unique nœud GRIS indique le nœud de départ.**

Exemple: parcours de graphe

- ***map*: si le nœud du couple (clef;valeur) d'entrée est GRIS, alors:**
  - **Pour chacun de ses fils, retourner (ID_FILS;"|GRIS|PROFONDEUR+1), où PROFONDEUR est la profondeur du nœud courant.**
  - **Retourner également le couple (clef;valeur) courant, avec couleur=NOIR.**
- **Sinon: retourner le couple (clef;valeur) d'entrée.**

- **Pseudo code:**

```
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: parcours de graphe

- ***reduce*: parcourir chacune des valeurs associées à la clef unique (après *shuffle*). Renvoyer un couple (clef;NOEUD) avec un nœud dont:**
  - **La profondeur est la plus haute rencontrée parmi les valeurs associées à cette clef unique (l'identifiant du nœud).**
  - **La couleur est la plus « forte » parmi ces mêmes valeurs.**
  - **La liste des nœuds enfants est la plus longue parmi ces valeurs.**

… et pour clef la clef unique en question.

Exemple: parcours de graphe

- **Pseudo code:**

```
H_CHILDREN=""; H_PROF=-1; H_COULEUR="BLANC";

POUR CHAQUE VALEUR:
  SI VALEUR.CHILDREN.LENGTH()>H_CHILDREN.LENGTH():
    H_CHILDREN=VALEUR.CHILDREN
  SI VALEUR.PROFONDEUR>H_PROF:
    H_PROF=VALEUR.PROFONDEUR
  SI VALEUR.COULEUR>H_COULEUR:
    H_COULEUR=VALEUR.COULEUR

NODE=NOUVEAU NOEUD
NODE.COULEUR=H_COULEUR
NODE.CHILDREN=H_CHILDREN
NODE.PROFONDEUR=H_PROF
RENVOYER(CLEF;NODE)
```

Exemple: parcours de graphe

- **On exécute le programme map/reduce plusieurs fois, jusqu'à ce que tous les nœuds de la liste aient pour couleur la valeur « NOIR » <=> jusqu'à ce que tous les nœuds aient été parcourus.**

- **Ce type de logique s'implémente très facilement au sein d'un *framework* map/reduce (Hadoop).**

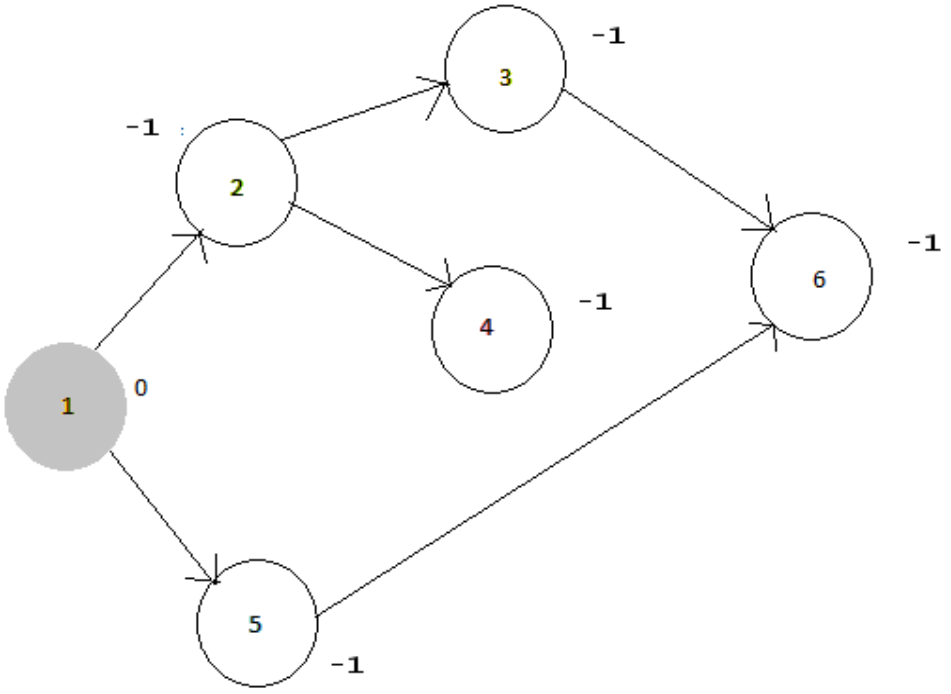
Exemple: parcours de graphe

- **Premier lancement:**

Données d'entrée:

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

Graphe:



Exemple: parcours de graphe

- **Pour ce premier lancement:**

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")
```

Aprés *shuffle*:

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

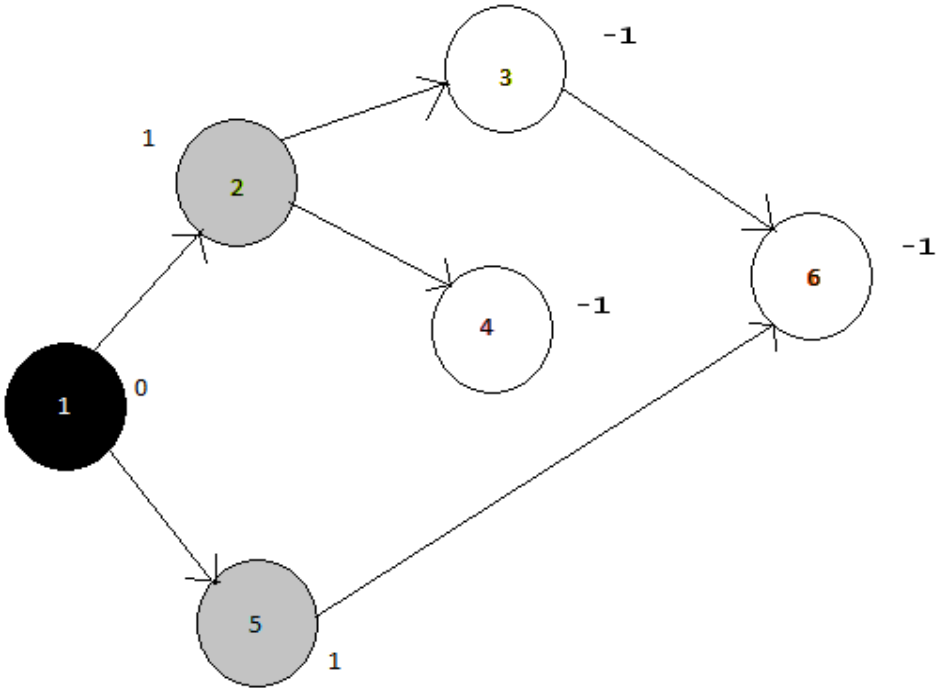
Exemple: parcours de graphe

- **Lancement 2:**

Données:

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

Graphe:



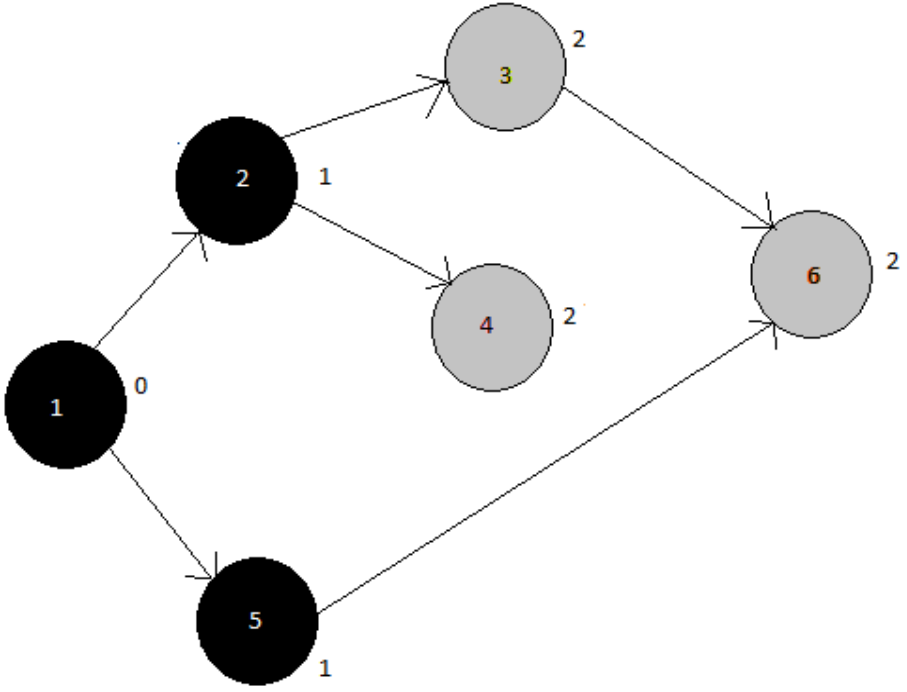
Exemple: parcours de graphe

- **Lancement 3:**

Données:

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

Graphe:



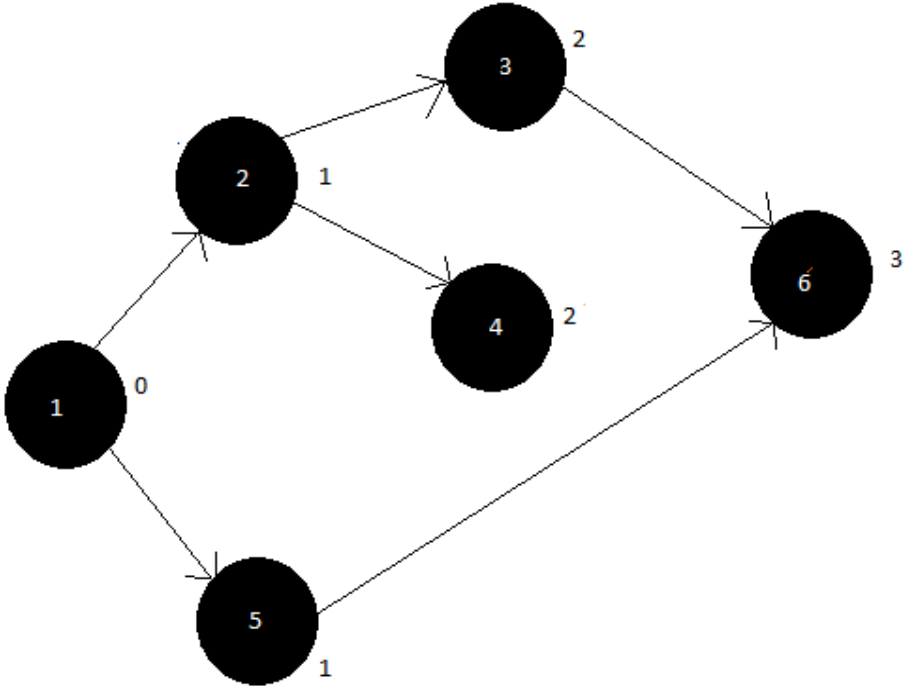
Exemple: parcours de graphe

- **Après le lancement 3:**

Données de sortie:

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

Graphe:



Exemple: parcours de graphe

- **Tous les nœuds ont tous désormais pour couleur la valeur « NOIR » ; le programme en charge de lancer la tâche map/reduce à répétition sur les données s’arrête.**

- **On a ainsi effectué un parcours en largeur sur le graphe, parallélisable (*parallel breadth-first search*). A chaque étape de profondeur dans le parcours correspond l'exécution d'une tâche map/reduce – avec une fonction *map* executée pour chaque nœud à chaque étape.**

Conclusion

- **Applications courantes de map/reduce et Hadoop:**
  - **Analyse de *logs* et données en général, validation de données, recoupements, filtrage, etc. => traitements sur des volumes de données massifs.**
  - **Exécution de tâches intensives en CPU de manière distribuée (simulations scientifiques, encodage vidéo, etc.).**

… et bien souvent les deux: tâches intensives en CPU sur des volumes de données massifs (entraînement de modèles en *machine learning*, etc.)

- **Plus complexe, voire contre-productif à appliquer sur tout problème où la fragmentation des données d'entrée pose problème (en fait sur tout problème où la stratégie du *diviser pour régner* n'est pas viable).**