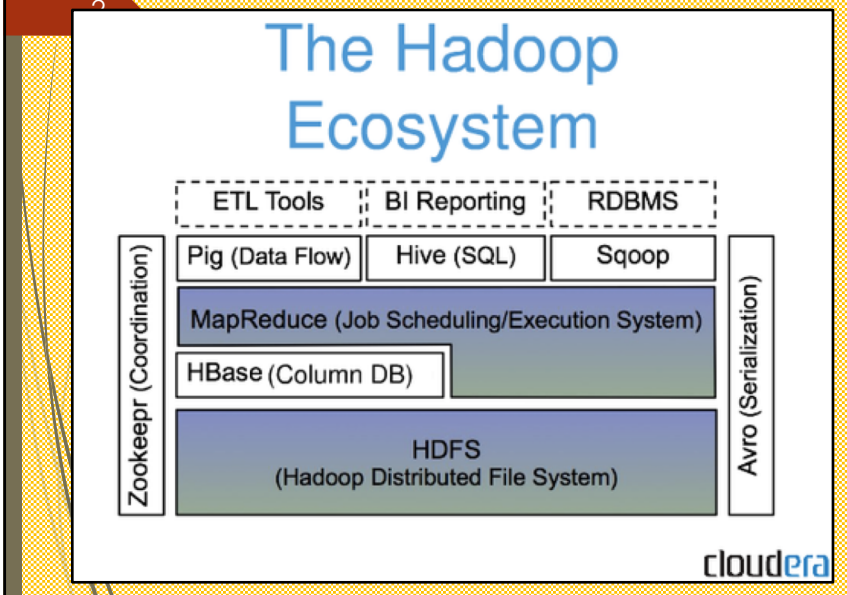
Introduction à Map Reduce

Rappel : l'écosystème Hadoop



3

MapReduce : Principe

- Lire séquentiellement *beaucoup de* données
- Map : extraire "quelque chose" d'intérêt

  map (k, v) → <k', v'>*

- Grouper par clé : Trier (Sort) et Mélanger (Shuffle)
- Reduce : agréger, résumer, filtrer ou transformer

  reduce (k', v') → <k'', v''>*
- Ecrire le résultat

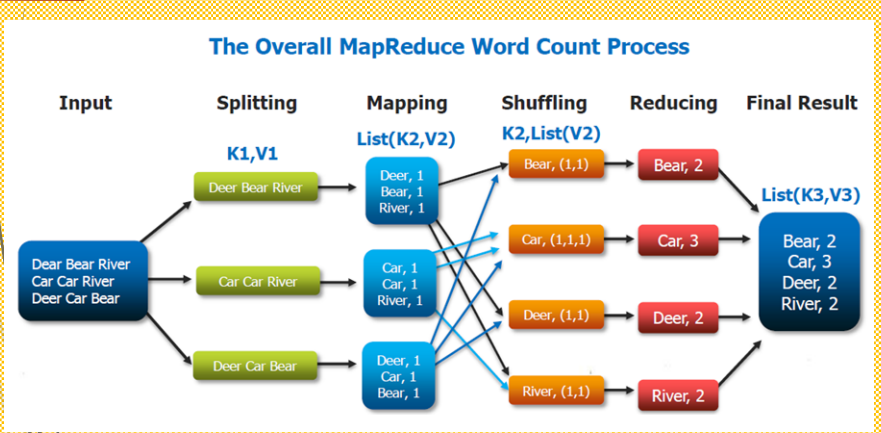
  La structure reste la même

  Map et Reduce changent pour s'adapter au problème.

Hadi Reduce

Map Reduce : comptage de mots

4



Hadi Reduce

5

MapReduce : Principe

- Les combinateurs combinent les valeurs de toutes les clés d'un même mapper
  - Beaucoup moins de données doivent être mélangées
- Fonction de partitionnement : contrôle la façon dont les clés sont partitionnées
  - Valeur par défaut : hash(key) mod R
  - Il est parfois utile de remplacer la valeur par défaut, par exemple, **hash(nomd'hôte(URL)) mod *R*** assure que les URL d'un hôte se retrouvent dans le même fichier de sortie.

Map Reduce

6

Environnement MapReduce

Il gère :

- Partitionnement des données d'entrée
- Planification de l'exécution du programme sur plusieurs machines
- Réalisation de l'étape *group by key*
- Pannes machines
- La communication inter-machines

Map Reduce

MapReduce : Flux de données

7

- L'entrée et la sortie finale sont stockées dans le *système de fichiers distribué* (DFS).
  - Le planificateur essaie de planifier les tâches Map "à proximité" de l'emplacement de stockage physique des données d'entrée.

  On peut spécifier un répertoire où se trouvent les fichiers d'entrée en utilisant *MultipleInputs.addInputPath*
- Les résultats intermédiaires sont stockés sur le *FS local* des workers Map et de Reduce.
- La sortie est souvent l'entrée d'une autre tâche MapReduce

DFS : Distributed File System

Map Reduce

Coordination : Master Node

8

- Le nœud maître assure la coordination :
  - Statut de la tâche : (inactif, en cours, terminé)
  - *Les tâches inactives* sont planifiées au fur et à mesure que les workers deviennent disponibles.
  - Lorsqu'une tâche Map est terminée, elle envoie au maître l'emplacement et la taille de ses fichiers *R* intermédiaires, un pour chaque reducer.
  - Le maître transmet cette information aux reducers.
- Le Master Node envoie périodiquement des pings aux workers pour détecter les défaillances

Map Reduce

Défaillances

9

- Mapper
  - Les tâches Map terminées ou en cours chez le worker sont mises en position idle (repos)
  - Les Reducers sont notifiés lorsque les tâches sont reprogrammées sur un autre worker.
- Reducer
  - Seules les tâches en cours sont remises à l'état idle.
- Master Node
  - La tâche MapReduce est annulée et le client est notifié

Combien de tâches Map /Reduce ?

10

- *M* map , *R* reduce
- **Règle générale :**
  - Choisir *M* beaucoup plus grand que le nombre de nœuds dans le cluster
  - Un « bout » de DFS par Map.
  - Améliore le load-balancing dynamique et accélère la récupération en cas de défaillance des workers.

  Remarque : on peut augmenter le nombre de tâches de la map en modifiant l'option

  Conf. setNumMapTasks(int num) de JobConf.
- **Habituellement, *R* est inférieur à *M***
  - La sortie est répartie sur R fichiers

11

Contraintes et inconnues

- Contrôle limité des flux de données et d'exécution
  - Tous les algorithmes doivent être exprimés en m, r
- On ne sait pas:
  - où les mappers et les reducers s'exécutent
  - quand un mapper ou un reducer commence ou finit
  - quelle entrée un mapper est en train de traiter
  - quelle clé intermédiaire un reducer est en train de traiter

Conception d'algorithmes MapReduce

12

- Nécessité de s'adapter à un modèle de calcul restreint
- Objectifs MapReduce
  - Passage à l'échelle : l'ajout de machines accélérera le fonctionnement de l'algo MapReduce
  - Efficacité : les ressources ne sont pas gaspillées
- La traduction de certains algorithmes en MapReduce n'est pas toujours évidente
  - Mais il existe des Design Patterns qui peuvent aider

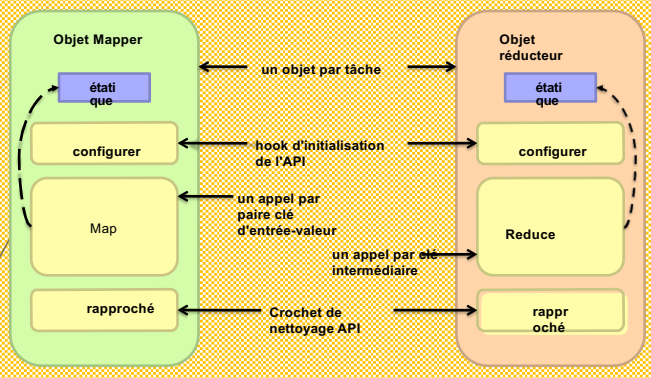
13

Outils de synchronisation

- Structures de données intelligentes
  - Regrouper des résultats partiels
- Tri sur les clés intermédiaires
  - Contrôle l'ordre de traitement des clés par les reducers
- Partitionneur
  - Contrôler quel réducteur traite quelles clés
- Préservation de l'état des mappers et des recducers
  - Capture les dépendances entre plusieurs clés et valeurs

14

Préservation de l'état



Vers des algorithmes Hadoop évolutifs

15

- Éviter la création d'objets
  - Fonctionnement coûteux
  - Garbage collector
- Éviter la bufferisation
  - Taille du tas (heap) limitée
  - Fonctionne pour les petits ensembles de données, mais ne s'adapte pas à l'échelle !

Hadoop MapReduce

Vers des algorithmes Hadoop évolutifs

16

- Caractéristiques idéales du passage à l'échelle:
  - Deux fois plus de données, deux fois plus de runtime
  - Deux fois plus de ressources, la moitié du runtime
- Pourquoi c'est difficile ?
  - La synchronisation exige la communication
  - La communication tue la performance
- Donc.... Eviter la communication !
  - *Réduire les données intermédiaires par agrégation locale*
  - Les combinateurs peuvent vous aider

Hadoop MapReduce

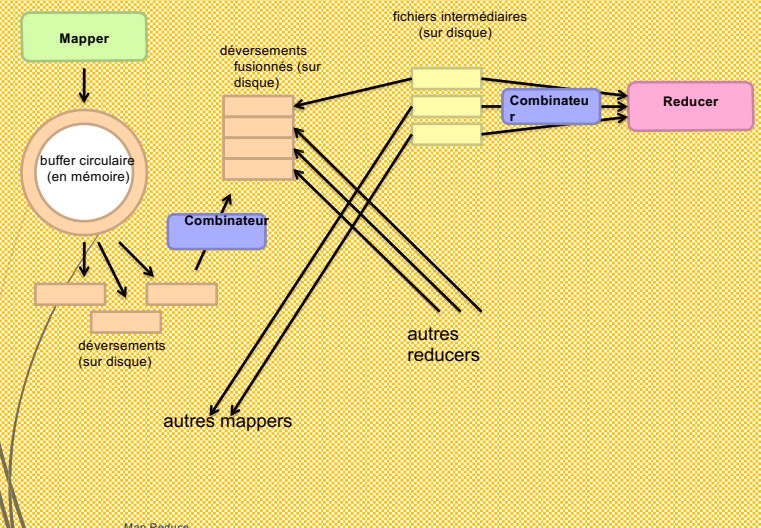
Shuffle&Sort dans Hadoop

17

- L'aspect le plus complexe de MapReduce !
- Côté Map
  - Les sorties du Map sont mises en mémoire tampon dans un buffer circulaire.
  - Lorsque le buffer atteint le seuil, le contenu est "déversé" sur le disque
  - Les déversements sont fusionnés dans un seul fichier partitionné (trié à l'intérieur de chaque partition) : le combinateurs s'exécute ici.
- Côté Reduce
  - Tout d'abord, les *sorties d'un Map sont copiées sur la machine Reducer.*
  - *"Sort" est une fusion multi-passes des outputs Map (se passe en mémoire et sur disque)* : le combinateurs s'exécute ici.
  - La fusion finale va directement au réducteur

18

Shuffle & Sort



Map Reduce

19

MR DESIGN PATTERNS

20

Stratégie : Agrégation locale

- Utiliser des combinateurs
- Agrégation à l'intérieur des *mappers*

21

Nombre de mots : Base de référence

```
1: class Mapper
2:     method Map(docid a, doc d)
3:         for all term t ∈ doc d do
4:             Emit(term t, count 1)

1: class Reducer
2:     method Reduce(term t, counts [c1, c2, ...])
3:         sum ← 0
4:         for all count c ∈ counts [c1, c2, ...] do
5:             sum ← sum + c
6:         Emit(term t, count s)
```

Supposons que la collection ait un total de *n* termes et de *d* termes distincts.
Quels sont les coûts de communication sans combinateurs ?

Quels sont les coûts de communication avec un combinateur ?

22

Nombre de mots : Agrégat dans Mapper

```
1: class Mapper
2:     method Map(docid a, doc d)
3:         H ← new AssociativeArray
4:         for all term t ∈ doc d do
5:             H{t} ← H{t} + 1                ▷ Tally counts for entire document
6:         for all term t ∈ H do
7:             Emit(term t, count H{t})
```

Les combinateurs sont-ils toujours nécessaires ?

Map Reduce

23

Nombre de mots : Agrégat dans Mapper (v 2)

```
1: class Mapper
2:     method Initialize
3:         H ← new AssociativeArray
4:     method Map(docid a, doc d)
5:         for all term t ∈ doc d do
6:             H{t} ← H{t} + 1                ▷ Tally counts across documents
7:     method Close
8:         for all term t ∈ H do
9:             Emit(term t, count H{t})
```

Key: preserve state across input key-value pairs!

Les combinateurs sont-ils toujours nécessaires ?