**MapReduce**

1. **Introduction**

Hadoop est un environnement logiciel "open source" de la fondation Apache, dédié au stockage et au traitement distribués de larges volumes de données.

Il repose sur deux composantes essentielles :

1. Un système de fichiers distribué HDFS (Hadoop Distributed File System)

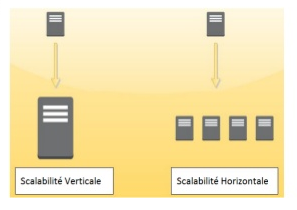
2. Une implémentation de l’algorithme MapReduce

Le principe de Hadoop est la subdivision des données, leur sauvegarde sur une collection de machines appelée cluster puis leur traitement là où elles sont stockées et non sur le serveur à l'aide d'un ensemble de logiciels (**Hadoop Common, MapReduce, Hadoop YARN, BD noSql,…)**

**Raisons**

Faire coopérer des machines simples au lieu d'un gros serveur. Ce qui a pour intérêt :

La réduction des coûts, une meilleure tolérance aux pannes (résilience) et la scalabilité (montée en charge verticale et horizontale).



|  |  |
| --- | --- |
| Vertical scaling | Horizontal scaling |

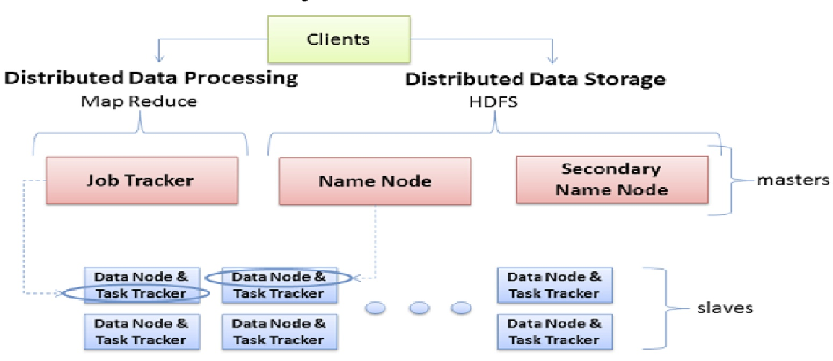
Cluster (grappe) de serveurs : Les nœuds sont groupés sous un même nom = nom du cluster qui permet de le désigner et le manipuler.

* Chaque fichier à stocker est subdivisé en blocs et chaque bloc est répliqué sur 3 noeuds ou plus.
* Si un des nœuds « tombe en panne», les données sont automatiquement répliquées sur d'autres.

**La taille peut être un problème**

|  |  |  |
| --- | --- | --- |
| **Taille**  • 100M mots  •1000M mots  •100MM mots  •1000MM mots  •Plus encore! | **Problème**  •Pas de problèmes  •Mémoire insuffisante  •Processeur insuffisant  •1 ordinateur insuffisant  •Réseau insuffisant, contrôleur surchargé | **Solution**  •Naïve avec 1 seul ordinateur  •Utiliser le disque, Fenêtre glissante  •Multithreading,  •Distribuer le calcul  •MapReduce (ou solution du même ordre comme MPI, etc.) |

**Pb : Distribuer le stockage et distribuer le traitement**



Une architecture Hadoop est basée sur les deux principaux rôles maître/esclave. Des sous-rôles relatifs au système de fichiers et à l'exécution de tâches distribuées sont associés à chaque machine.

* **JobTracker** : Il s'agit de la responsabilité pour une machine maître de lancer des tâches distribuées, en coordonnant les esclaves. Le JobTracker, planifie les exécutions, gère l'état des machines esclaves et agrège les résultats de calculs dans le cadre de MapReduce.
* **TaskTracker** : Il s'agit du rôle permettant à un esclave d'exécuter une tâche MapReduce sur les données qu'elle héberge. Le TaskTracker est piloté par le JobTracker d'une machine maître qui lui envoie la tâche à  exécuter.

1. **MapReduce**

**Objectif**

Définit un modèle de développement permettant de programmer des calculs parallèles et distribués de manière simple.

**Principe**

Au lieu de parcourir le fichier séquentiellement (beaucoup de temps, pb de mémoire, …) on le découpe en blocs puis les parcourir en parallèle.

MapReduce est un Framework de traitement de données en clusters composé des fonctions Map et Reduce. Il permet de répartir les tâches de traitement de données entre différents ordinateurs, pour ensuite réduire les résultats en une seule synthèse.

**Ecrire un programme Map Reduce:**

1 Choisir une manière de découper les données afin que Map soit parallélisable

2 Choisir la clé à utiliser pour le problème (c'est le nombre de blocs de 64Mo)

3 Écrire le programme pour l’opération Map

4 Écrire le programme pour l’opération Reduce

**Principe général**



**MapReduce** adresse deux choses.

* La première concerne le modèle de programmation MapReduce.
* La seconde concerne le framework d'implémentation.

**Fonction map**

Cette étape consiste à appliquer une même fonction à tous les éléments de la liste;

map(T)[x0, ..., xn] = [T(x0), ...,T(xn)] ( avec X0=(cle, valeur)

map(\*4)[2, 3, 6] = [8, 12, 24]

**Fonction reduce**

Applique une fonction récursivement à une liste et retourne un seul résultat;

reduce(f)[y0, ..., yn] = f(y0, f(y1, f(y2, ...)))

reduce(+)[2, 3, 6,9] = (2 + (3 +(6+9)) = 20

**MapReduce agit sur des paires(clé, valeur )**

La base fondamentale pour l'automatisation de la parallélisation, est que **l'ensemble des données est représenté sous la forme de paires(clé, valeur).**

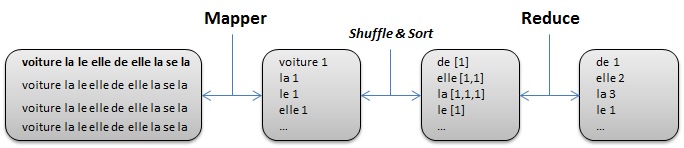
**Exemple : fréquences de lettres**

****

**Exemple :** Compteur de mots

Tous les mots sont comptabilisés à l'exception du mot « se ».

Ci-dessous, le fichier exemple.txt présente un jeu de données comportant une seule ligne : "voiture la le elle de elle la se la maison voiture"



La fonction map s'écrit de la manière suivante : map(clé1, valeur1) → List(clé2, valeur2).

À partir d'un couple clé/valeur, la fonction map retourne un ensemble de nouveaux couples clé/valeur. Cet ensemble peut être vide, d'une cardinalité un ou plusieurs.

Dans notre exemple, la *clé* d'entrée correspond au numéro de ligne dans le fichier et la *valeur* vaut  "*voiture la le elle de elle la se la maison voiture"*.

Le résultat de la fonction map est :

(voiture,1)

(la,1)

(le,1)

(elle,1)

(de,1)

(elle,1)

(la,1)

(la,1)

(maison,1)

(voiture,1)

**Implémentation Java de la fonction map**

**public** class WordCountMapper **extends** Mapper<LongWritable, Text, Text, IntWritable> {

**public** void map(LongWritable key, Text value, Context context) **throws** IOException, InterruptedException {

StringTokenizer itr = **new** StringTokenizer(value.toString());

**while** (itr.hasMoreTokens()) {

String s = itr.nextToken();

**if** (!s.equals("se")) {

Text word = **new** Text();

word.set(s);

context.write(word, **new** IntWritable(1));

}

}

}

}

**Regroupement et tri**

Avant de présenter la fonction **reduce**, deux opérations intermédiaires doivent être exécutées pour préparer la valeur de son paramètre d'entrée.

La première opération appelée **shuffle** permet de grouper les valeurs dont la clé est commune.

La seconde opération **sort** permet de trier par clé.

Les fonctions **shuffle** et **sort** sont des fonctions fournies par le framework Hadoop.

Ainsi, après l'exécution des fonctions **shuffle** et **sort** le résultat de l'exemple est le suivant.

(de,[1])

(elle,[1,1]

(la,[1,1,1])

(le,[1])

(maison,[1])

(voiture,[1,1])

Nous avons maintenant à notre disposition un ensemble de paires(clé, liste\_valeurs).

Il reste maintenant à écrire le code de l'opération reduce, selon le schéma imposé par MapReduce.

Pour WordCount, l'opération reduce va juste consister à sommer toutes les valeurs de la liste associée à une clé.

**Fonction reduce**

La fonction reduce s'écrit de la manière suivante : reduce(clé2, List(valeur2)) → List(valeur2). À partir des groupes de valeurs associées à une clé, la fonction reduce retourne généralement une valeur ou aucune, bien qu'il soit possible de retourner plus d'une valeur.

Suite à l'appel de la fonction reduce, le résultat de l'exemple est le suivant.

(de,1)

(elle,2)

(la,3)

(le,1)

(maison,1)

(voiture,2)

**public** class WordCountReducer **extends** Reducer<Text, IntWritable, Text, IntWritable> {

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

int sum = 0;

**for** (IntWritable current : values) {

sum += current.get();

}

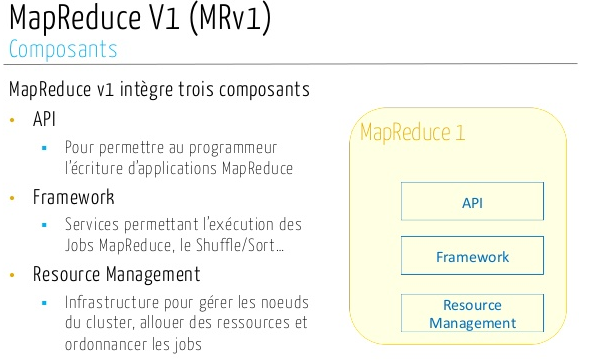
context.write(key, **new** IntWritable(sum));

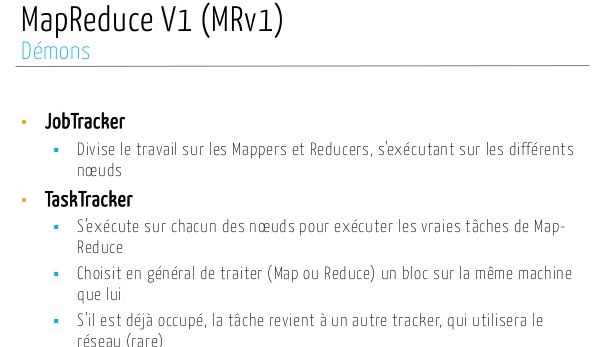
}

}

MapReduce est disponible dans plusieurs langages, dont C, C++, Java, Ruby, Perl et Python.

**Map reduce 1 et 2 :** [**https://fr.slideshare.net/LiliaSfaxi/bigdatachp2-hadoop-mapreduce**](https://fr.slideshare.net/LiliaSfaxi/bigdatachp2-hadoop-mapreduce)





<http://blog.businessdecision.com/bigdata/2015/09/tutoriel-mapreduce/>

L'exemple canonique de MapReduce compte l'apparition de chaque mot dans un ensemble de documents :

Ici, chaque document est divisé en mots et chaque mot est compté par la fonction *map* , en utilisant le mot comme clé de résultat.

Le framework rassemble toutes les paires avec la même clé et les transmet au même appel pour *réduire*.

Ainsi, cette fonction a simplement besoin d'additionner toutes ses valeurs d'entrée pour trouver les apparitions totales de ce mot.

Comme autre exemple, imaginez que pour une base de données de 1,1 milliard de personnes, on veuille calculer le nombre moyen de contacts sociaux qu'une personne a en fonction de son âge.

**unction** map(String name, String document):

*// name: document name*

*// document: document contents*

**for each** word w **in** document:

emit (w, 1)

**function** reduce(String word, Iterator partialCounts):

*// word: a word*

*// partialCounts: a list of aggregated partial counts*

sum = 0

**for each** pc **in** partialCounts:

sum += pc

emit (word, sum)

**Conclusion**

L'approche MapReduce pour le calcul distribué consiste à reformuler son problème en fonctions parallélisables avec un schéma relativement contraint.

L'implémentation concrète dans un contexte Big Data, sur un cluster de machines dépend de tout l'écho système.

Comment se fait l'ordonnancement et la distribution des calculs ?

Comment MapReduce permet-il de répondre aux principales problématiques du calcul distribué abordées dans la partie précédence à savoir :

* l'accès et le partage de données ;
* la gestion des erreurs et la tolérance aux pannes ;
* la localisation des données ?

**Java map reduce coding**

**mapper class**

public class  Mapper  {

    private final static IntWritable one = new IntWritable(1);

    private Text word = new Text();

    public void map(LongWritable **key**, Text **value**,OutputCollector**<Text**, IntWritable**>** **output**,

           Reporter **reporter**) throws IOException{

        String line = value.toString();

        StringTokenizer  tokenizer = new **StringTokenizer**(line);

        while (tokenizer.**hasMoreTokens**()){

            word.set(tokenizer.**nextToken**());

            output.collect(word, one);

        }

    }

}

**The reducer class**

    public class  Reducer   {

    public void reduce(Text **key**, Iterator<IntWritable> **values**,OutputCollector<Text,IntWritable> **output**,

     Reporter **reporter**) throws IOException {

    int sum=0;

    while (values.hasNext()) {

    sum+=values.next().get();

    }

    output.collect(key,new IntWritable(sum));

    }

    }

**The Runner class**

    public class  Runner {

        public static void main(String[] args) throws IOException{

            JobConf conf = new JobConf(Runner.class);

            conf.setJobName("WordCount");

            conf.setOutputKeyClass(Text.class);

            conf.setOutputValueClass(IntWritable.class);

            conf.setMapperClass(Mapper.class);

            conf.setCombinerClass(Reducer.class);

            conf.setReducerClass(Reducer.class);

            conf.setInputFormat(TextInputFormat.class);

            conf.setOutputFormat(TextOutputFormat.class);

            FileInputFormat.setInputPaths(conf,new Path(args[0]));

            FileOutputFormat.setOutputPath(conf,new Path(args[1]));

            JobClient.runJob(conf);

        }

    }