Introduction à Map-Reduce

Vincent Leroy

Sources

- Apache Hadoop
- Yahoo! Developer Network
- Hortonworks
- Cloudera
- Practical Problem Solving with Hadoop and Pig
- Les cours seront mis en ligne sur http://lig-membres.imag.fr/leroyv/

Besoin « Big Data »

- Google, 2008
  - 20 PB/jour
  - 180 GB/job (très variable)
- Index du Web
  - 50 milliards de pages
  - 15PB
- Grand collisionneur de hadrons (LHC) du CERN : génère 15PB/an

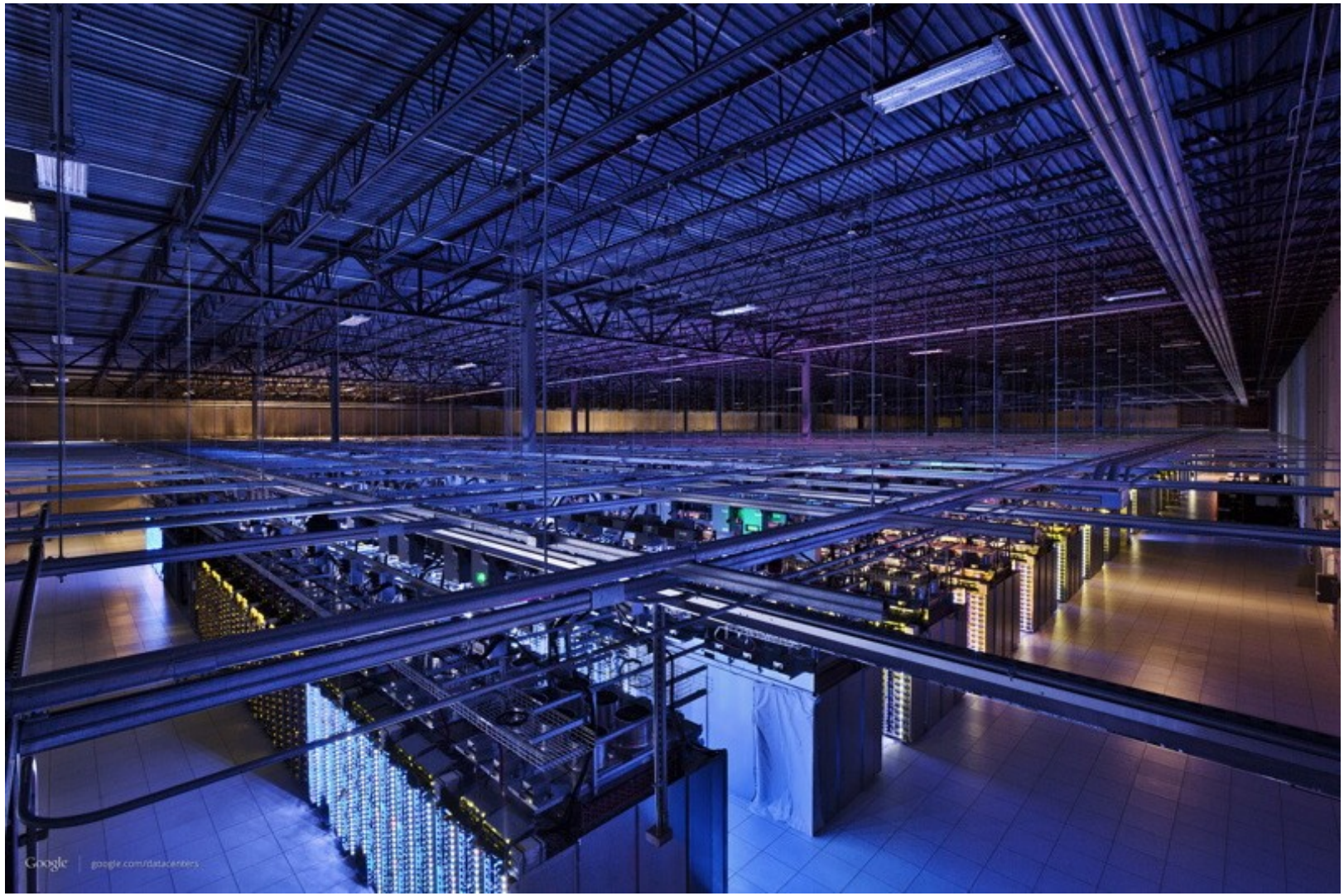
Capacité d’un (gros) serveur

- Capacité mémoire max : 256 GB
- Capacité disque max : 24TB
- Débit disque : 100MB/s

Solution: Parallélisme

- 1 serveur
  - 8 disque
  - Lire le Web : 230 jours
- Cluster Hadoop Yahoo
  - 4000 serveurs
  - 8 disques/serveur
  - Lire le Web en parallèle : 1h20

Data center Google



Problèmes de la programmation parallèle « classique »

- Synchronisation
  - Mutex, sémaphores …
- Modèles de programmation
  - Mémoire partagée (multicores)
  - Passage de messages (MPI)
- Difficultés
  - Programmer / débugger (deadlocks …)
  - Optimiser
  - Rendre élastique (nombre arbitraire de machines)
  - Coûteux
  - Peu réutilisable

Tolérance aux pannes

- Un serveur tombe en panne de temps en temps
- 1000 serveurs …
  - MTBF (temps moyen entre 2 pannes) < 1 jour
- Un gros job prend plusieurs jours
  - Il y aura des pannes, c'est **normal**
  - Le calcul doit finir dans un délais prédictible
    → On ne relance pas tout pour une panne !
- Checkpointing, réplication
  - Difficile à faire à la main

MODÈLE DE PROGRAMMATION MAP-REDUCE

Que sont Map et Reduce ?

- 2 fonctions simples inspirées de la programmation fonctionnelle
  - map (*2) [1,2,3] = [2,4,6]
  - reduce (+) [2,4,6] = 12
- Fonctions « génériques »
- Leur combinaison permet de modéliser énormément de problèmes
- Le développeur fournit l'opérateur appliqué

Map-Reduce sur des clés/valeurs

- Map-Reduce manipule des paires clé/valeur
  - Map est appliqué **indépendamment** à chaque paire clé/valeur
    map(clé, valeur) → liste(clé,valeur)
  - Reduce est appliqué à **toutes** les valeurs associées à la même clé
    reduce(clé,liste(valeur)) → liste(clé,valeur)
  - Les clés/valeurs en sortie ne sont pas forcément du même type que les entrées

Exemple : Compter la fréquence de mots

- Input : un fichier de 2 lignes
  - 1, "a b c aa b c"
  - 2, "a bb cc a cc b"
- Output
  - a, 3
  - b, 3
  - c, 2
  - aa, 1
  - bb, 1
  - cc, 2

Comptage de fréquence : Mapper

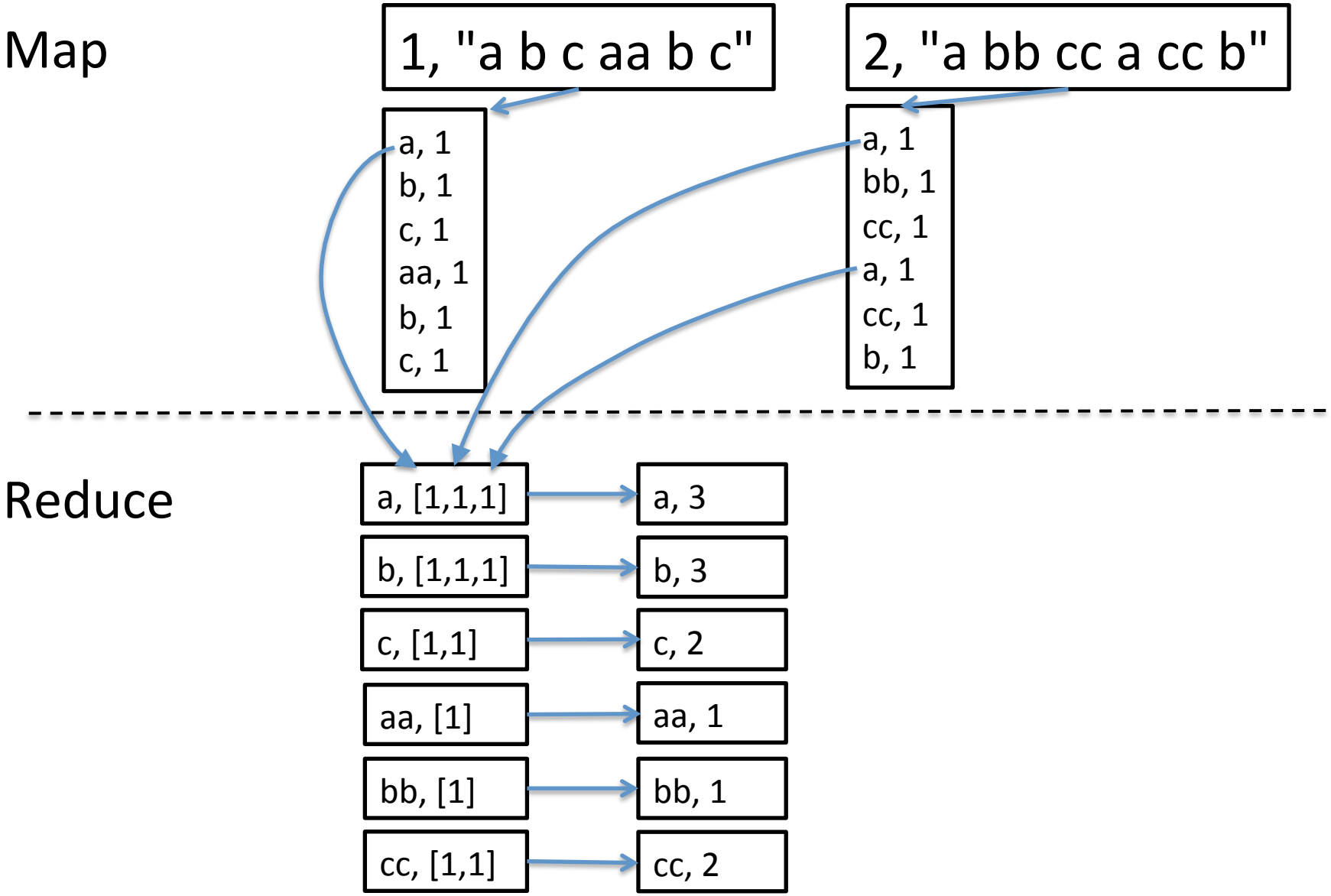
- Map traite une fraction de texte (valeur)
  - Délimiter les mots
  - Pour chaque mot, compter une occurrence
  - La clé n'est pas importante dans cet exemple
- Map(String line, Output output){
  foreach String word in line.split() {
  output.write(word, 1)
  }
  }

Comptage de fréquence : Reducer

- Pour chaque clé, Reduce traite toutes les valeurs correspondantes
  - Additionner le nombre d'occurrences
- Reduce(String word, List<Int> occurrences, Output output){

```
    int count = 0
    foreach int occ in occurrences {
        count += occ
    }
    output.write(word,count)
}
```

Schéma d'exécution



HDFS : SYSTÈME DE FICHIERS DISTRIBUÉ

Accès aléatoires / séquentiels

- Exemple
  - BDD 100M d'utilisateurs
  - 100B/utilisateur
  - Modifier 1% des enregistrements
- Accès aléatoire
  - Seek, lecture, écriture : 30mS
  - 1M d'utilisateurs → 8h20
- Accès séquentiel
  - On lit TOUT et on réécrit TOUT
  - 2 fois 10GB à 100MB/S → 3 minutes

→ Il est souvent plus efficace de tout lire et tout réécrire séquentiellement

Système de fichiers distribué (HDFS)

- Système de fichiers distribué
  - Redondance (tolérance aux pannes)
  - Performance (lecture parallèle)
- Gros fichiers
  - Lectures séquentielles
  - Écritures séquentielles
- Traitement des données « en place »
  - Stockage et traitement sur les mêmes machines
    - Meilleure utilisation des machines (pas de *filer* spécialisé)
    - Moins de saturation réseau (meilleures performances)

Modèle HDFS

- Données organisées en fichiers et répertoires
  → proche d'un système de fichiers classique
- Fichiers divisés en blocks (64MB par défaut) répartis sur les machines
- HDFS indique au framework Map-Reduce le placement des données
  → Si possible, exécution du programme sur la machine où sont placées les données nécessaires

Tolérance aux fautes

- Blocks des fichiers répliqués (3 par défaut) pour faire face aux pannes
- Placement suivant différents facteurs
  - Alimentation électrique
  - Équipement réseau
  - Placement varié pour augmenter la possibilité d'avoir une copie proche
- Checksum des données pour détecter les corruptions de données (présent sur les systèmes de fichiers modernes)

Architecture Master/Worker

- Un *maître*, le NameNode
  - Gère l'espace des noms de fichiers
  - Gère les droits d'accès
  - Dirige les opérations sur les fichiers, blocks …
  - Surveille le bon état du système (pannes, équilibre …)
- Beaucoup (milliers) d'esclaves, les DataNodes
  - Contient les données (blocks)
  - Effectue les opérations de lecture/écriture
  - Effectue les copies (réplication, dirigée par le NameNode)

NameNode

- Stocke les métadonnées de chaque fichier et block (*inode*)
  - Nom de fichier, répertoire, association block/ fichier, position des blocks, nombre de réplicas …
- Garde tout en mémoire (RAM)
  - Facteur limitant = nombre de fichiers
  - 60M d'objets tiennent en 16GB

DataNode

- Gère et surveille l'état des blocks stockés sur le système de fichier de l'OS hôte (souvent linux)
- Accédé directement par les clients
  → les données ne transitent pas par le NameNode
- Envoie des *heartbeats* au NameNode pour indiquer que le serveur n'est pas en panne
- Indique au NameNode si des blocks sont corrompus

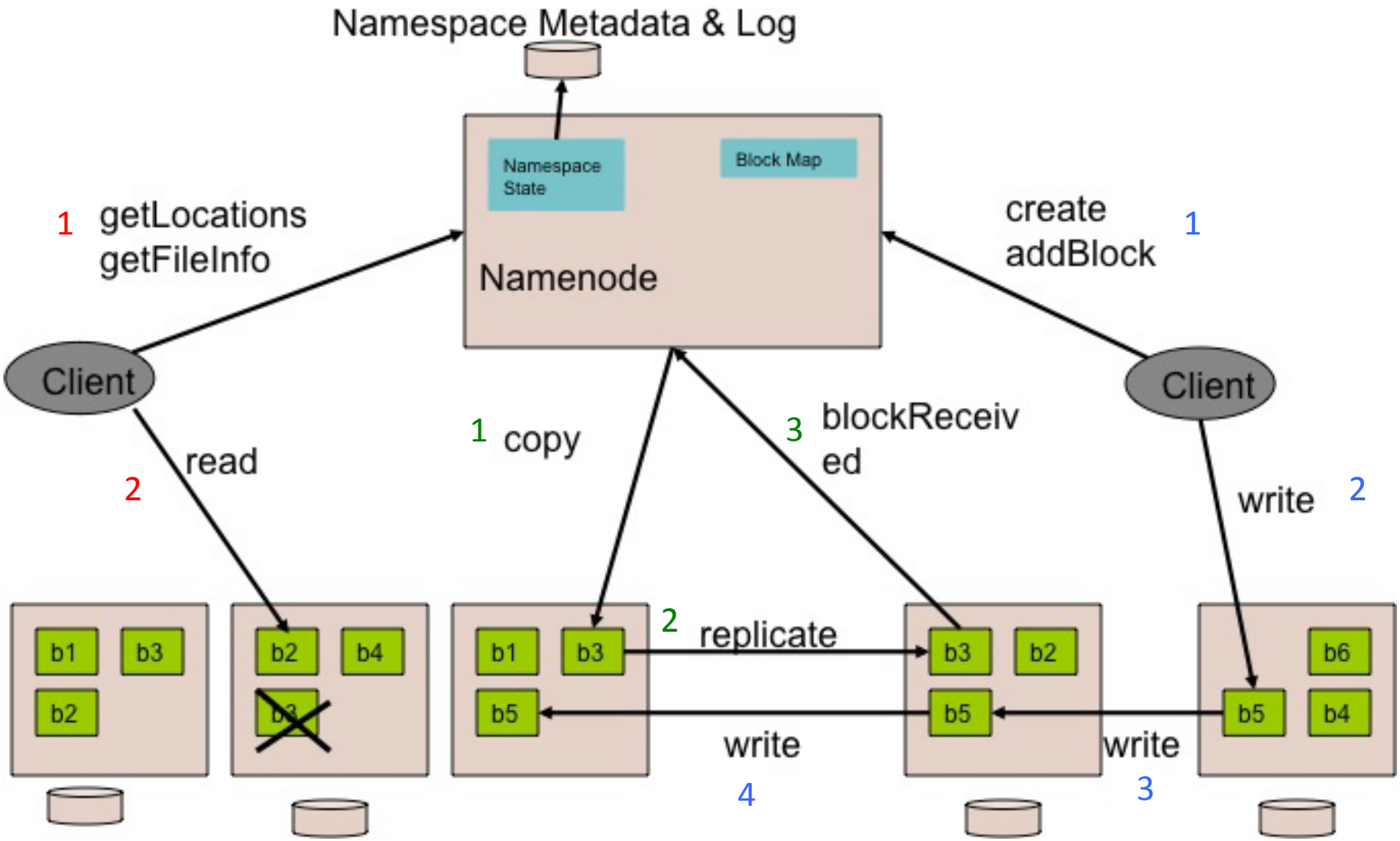
Ecriture d'un fichier

- Le client fait une requête au NameNode pour créer un nouveau fichier
- Le NameNode vérifie
  - les permissions du client
  - si le nom du fichier n'existe pas déjà
- Des DataNodes sont choisis pour stocker les blocs du fichier et des répliques
  - "pipeline" de DataNodes
- Des blocks sont alloués sur ces DataNodes
- Le flux des données du client est dirigé sur le 1er DataNode du pipeline
- Chaque DataNode forwarde les données reçues aux DataNode suivant du pipeline

Lecture d'un fichier

- Le client fait une requête au NameNode pour lire un fichier
- Le NameNode vérifie que le fichier existe et construit la liste des DataNodes contenant les premiers blocs
- Pour chacun de ces blocs, le NameNode renvoie les adresse des DataNodes les contenant
  - cette liste est triée par ordre de proximité au client
- Le client se connecte au DataNode le plus proche contenant le 1er bloc du fichier
- Lecture d'un bloc terminée :
  - Connexion au DataNode coupée
  - Nouvelle connexion au DataNode contenant le bloc suivant
- Quand tous les premiers blocs lus :
  - Requête au NameNode pour avoir l'ensemble de blocs suivants

Structure d'HDFS



Commandes HDFS (répertoires)

- Créer répertoire rep
  $ hdfs dfs –mkdir rep
- Lister contenu HDFS
  $ hdfs dfs -ls
- Effacer répertoire rep
  $ hdfs dfs –rm –r rep

Commandes HDFS (fichiers)

- Copier fichier local toto.txt dans HDFS rep/
  $ hdfs dfs -put toto.txt rep/toto.txt
- Copier fichier HDFS sur le disque local
  $ hdfs dfs -get rep/toto.txt ./
- Voir fichier rep/toto.txt
  $ hdfs dfs –cat rep/toto.txt
- Effacer fichier rep/toto.txt
  $ hdfs dfs –rm rep/toto.txt

APACHE HADOOP : FRAMEWORK MAP-REDUCE

Objectifs du framework Map-Reduce

- Offrir un modèle de programmation simple et générique : fonctions map et reduce
- Déployer automatiquement l'exécution
- Prendre en charge la tolérance aux pannes
- Passage à l'échelle jusqu'à plusieurs milliers de machines
- La performance pure est importante mais n'est pas prioritaire
  - L'important est de finir dans un temps raisonnable
  - Si c'est trop lent, ajoutez des machines !
    Kill It With Iron (KIWI principle)

Que fait le développeur ?

- Implémente les opérations Map et Reduce (Java, C++ …)
  - Dépend du programme
- Définit ses types de données (clés / valeurs)
  - Si non standards (Text, IntWritable … fournis)
  - Méthodes pour sérialiser
- C'est tout.

Imports

```
import java.io.IOException ;
import java.util.* ;

import org.apache.hadoop.fs.Path ;
import org.apache.hadoop.io.IntWritable ;
import org.apache.hadoop.io.LongWritable ;
import org.apache.hadoop.io.Text ;
import org.apache.hadoop.mapreduce.Mapper ;
import org.apache.hadoop.mapreduce.Reducer ;
import org.apache.hadoop.mapreduce.JobContext ;
import
org.apache.hadoop.mapreduce.lib.input.FileInputFormat ;
import
org.apache.hadoop.mapreduce.lib.output.FileOutputFormat ;
import org.apache.hadoop.mapreduce.Job ;
```

Attention nouvelle / ancienne API

Mapper

```
 // type clé input, type valeur input, type clé output, type valeur output
public class WordCountMapper extends Mapper<LongWritable, Text, Text, IntWritable> {

    @Override
    protected void map(LongWritable key, Text value, Context context) throws IOException, InterruptedException {
        for (String word : value.toString().split("\\s+")) {
            context.write(new Text(word), new IntWritable(1));
        }
    }
}
```

Reducer

```
 // type clé input, type valeur input, type clé output, type valeur output
public class WordCountReducer extends Reducer<Text, IntWritable, Text, LongWritable> {

    @Override
    protected void reduce(Text key, Iterable<IntWritable> values, Context context) throws IOException, InterruptedException {
        long sum = 0;
        for (IntWritable value : values) {
            sum += value.get();
        }
        context.write(key, new LongWritable(sum));
    }
}
```

Main

```
public class WordCountMain {
    public static void main(String [] args) throws Exception {
        Configuration conf = new Configuration();
        String[] otherArgs = new GenericOptionsParser(conf,
args).getRemainingArgs();
        Job job = Job.getInstance(conf, "word count");
        job.setJarByClass(WordCountMain.class);
        job.setMapOutputKeyClass(Text.class);
        job.setMapOutputValueClass(IntWritable.class);
        job.setOutputKeyClass(Text.class);
        job.setOutputValueClass(LongWritable.class);
        job.setMapperClass(WordCountMapper.class);
        job.setReducerClass(WordCountReducer.class);
        job.setInputFormatClass(TextInputFormat.class);
        job.setOutputFormatClass(TextOutputFormat.class);
        FileInputFormat.addInputPath(job, new Path(otherArgs[0]));
        FileOutputFormat.setOutputPath(job, new Path(otherArgs[1]));
        System.exit(job.waitForCompletion(true) ? 0 : 1);
}
```

Terminologie

- Programme Map-Reduce = job
- Les jobs sont soumis au jobtracker
- Un job est divisé en plusieurs tasks
  - un map est une task
  - un reduce est une task
- Les tasks sont surveillées par des tasktrackers
  - une task lente est appelée straggler
- Dans Map-Reduce, barrière entre les maps et les reduce (shuffle & sort)
  - il faut attendre le map le plus lent avant de commencer les reduce

Exécution d'un job

- *$ hadoop jar wordcount.jar org.myorg.WordCount inputPath(HDFS) outputPath(HDFS)*
- Les paramètres sont vérifiés
  - un répertoire d'output t'il été spécifié ?
  - le répertoire d'output existe t'il déjà ?
  - un répertoire d'input a t'il été spécifié ?
- Les splits sont calculés
- Le job (code Map-Reduce), sa configuration et les splits sont copiés avec une forte réplication
- Un objet pour suivre la progression des tâches est créé sur le jobtracker
- Pour chaque split, un mapper est créé
- Le nombre de reducer par défaut est créé

Tasktracker

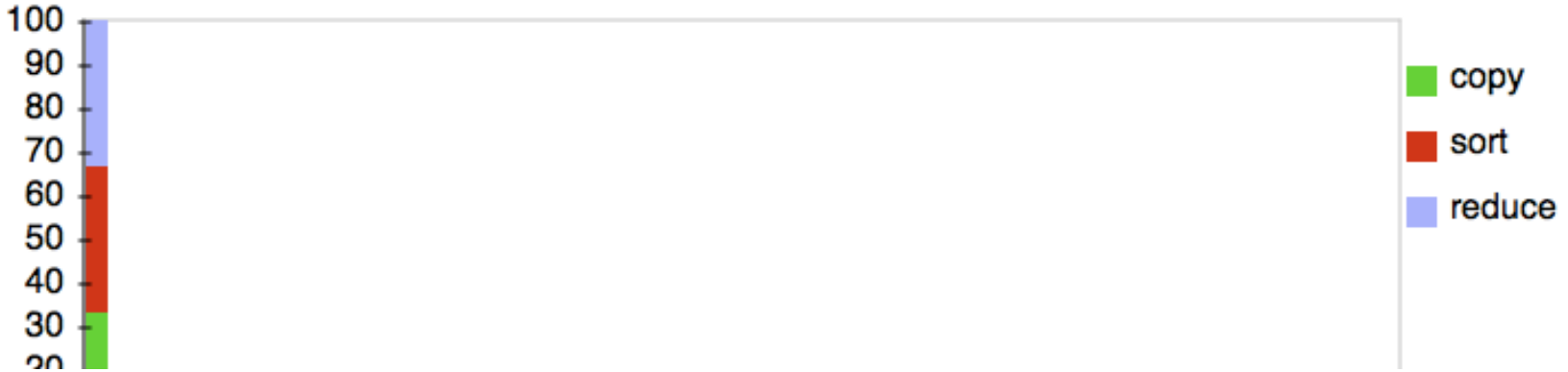
- Le tasktracker envoie périodiquement un signal au jobtracker
  - indique que le noeud fonctionne toujours
  - indique si le tasktracker est prêt à accepter une nouvelle task
- Un tasktracker est en général responsable d'un noeud
  - nombre fixé de slots pour des tasks map
  - nombre fixé de slots pour des tasks reduce
  - tasks peuvent être de jobs différents
- Chaque task tourne sur sa propre JVM
  - permet d'isoler l'exécution des tâches au sein du tasktracker

Suivi de la progression

- Une task map connaît son état d'avancement, i.e. la proportion du split qu'il a traitée
- Pour une task reduce, trois phases pour l'état d'avancement:
  - copie
  - tri
  - reduce
- Ces informations sont passées au TaskTracker
- Toutes les 5 secondes (ou plus), l'information d'avancement est envoyée au JobTracker
- Le JobTracker peut fournir ces informations au client, ou à l'interface web

Suivi de la progression

Reduce Completion Graph - close



Fin du Job

- Les outputs de chaque reduce sont écrits dans un fichier
- Le jobtracker envoie un message au client, qui affiche les compteurs du job

14/10/28 11:54:25 INFO mapreduce.Job: Job job_1413131666506_0070 completed successfully

Job Counters

Launched map tasks=392

Launched reduce tasks=88

Data-local map tasks=392

[...]

Map-Reduce Framework

Map input records=622976332

Map output records=622952022

Reduce input groups=54858244

Reduce input records=622952022

Reduce output records=546559709

[...]

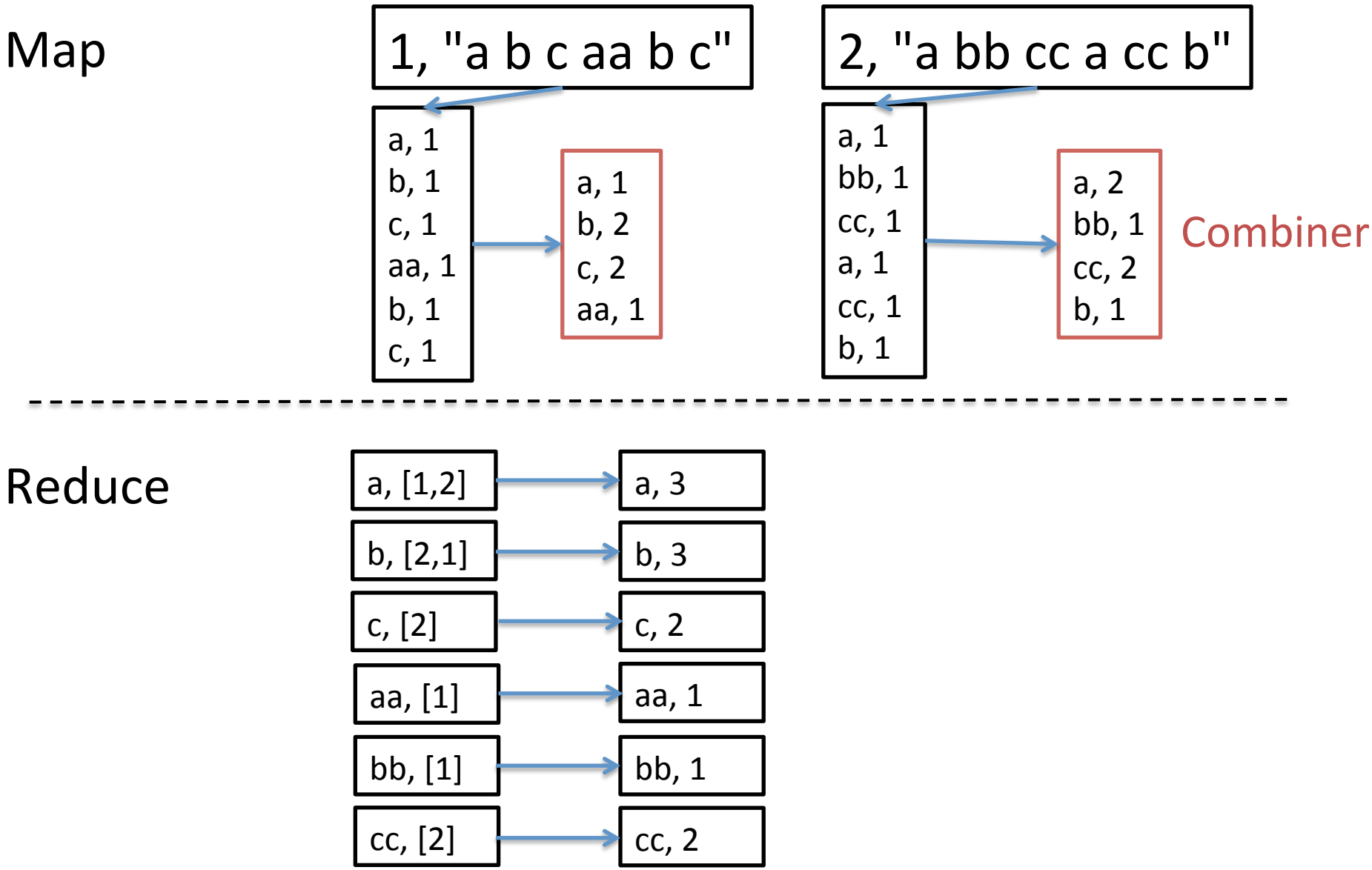
Panne d'un nœud durant un job

- Bug dans une task
  - JVM de la task crashe → JVM du tasktracker notifiée
  - task supprimée de son slot
- task ne répond plus
  - timeout de 10 minutes
  - task supprimée de son slot
- Chaque task est réessayée N fois (défaut 7)

Combiner

- Problème possible d'un map : beaucoup de couples clé/valeurs en output
- Ces couples doivent être copiés au reducer, voire transmis sur le réseau : coûteux
- Combiner : mini-reducer qui se place à la sortie du map et réduit le nombre de couples
- Types d'input du combiner = types d'output du combiner = types de sortie du map
- Combiner utilisé optionnellement par Hadoop
  - la correction du programme ne doit pas en dépendre
- conf.setCombiner(…)

Combiner


Map
1, "a b c aa b c"
2, "a bb cc a cc b"
a, 1
b, 1
c, 1
aa, 1
b, 1
c, 1
a, 1
b, 2
c, 2
aa, 1
a, 1
bb, 1
cc, 1
a, 1
cc, 1
b, 1
a, 2
bb, 1
cc, 2
b, 1
Combiner
Reduce
a, [1,2]
a, 3
b, [2,1]
b, 3
c, [2]
c, 2
aa, [1]
aa, 1
bb, [1]
bb, 1
cc, [2]
cc, 2