Hadoop

Généralités et points de repère (big data, hadoop)

How big is Big Data ?

Une ambiguïté sur le sens précis de cette notion :

- Données trop massives pour entrer en **RAM**
- Données trop massives pour être **stockées** sur une machine
- Données trop massives pour une gestion conventionnelle efficace

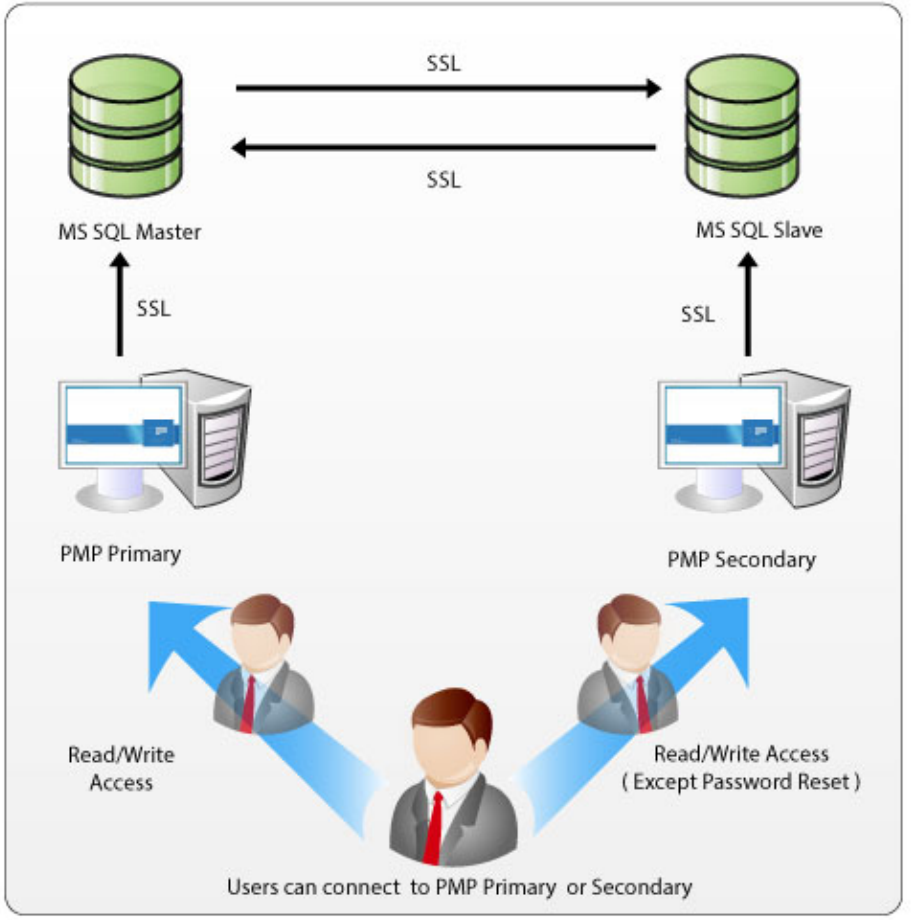
Nous verrons que cette ambiguïté provient de la collusion de plusieurs enjeux :

- Calculatoires
- Stockage
- Pérennité des données

Les solutions classiques au problème de

Accessibilité des données

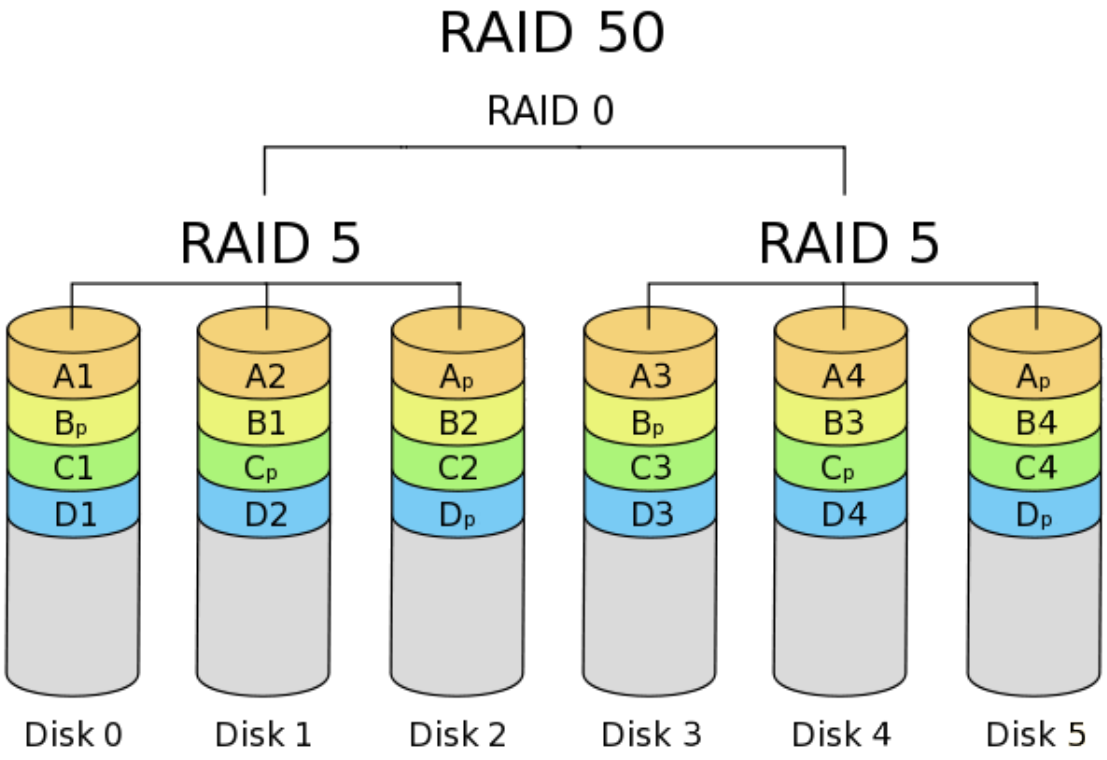
Duplication – replication / configuration master - slave



Les solutions classiques au problème de

Pérennité des données + vitesse des opérations IO

Raid (5, 10, 50, etc)



Les solutions classiques

- Coût de la maintenance, Gestion lourde
- Besoin de **matériel spécifique** et coûteux
- Paradigme très BDD relationnelle (quid du stockage d'autres ressources organisées selon une autre logique)
- Solutions propriétaires VS configurations à la mano interminables
- Problème du **goulot d'étranglement computationnel** peu ou mal adressé
- **Vitesse d'écriture** réduite à la bande passante de chaque serveur

Et hadoop fut

- Hadoop est né du constat des limitations de l’approche classique (duplication de serveur sans intégration dans un « **tout organique** »)
- Principes de bases : **clustering**, **redondance**

Hadoop, Kesako ?

Au sens strict, Hadoop est un framework pour la création d'applications distribuées écrit en Java.

Ce qui est distribué (sur plusieurs serveurs) :

- Le **stockage** des données
- Leur **traitement**

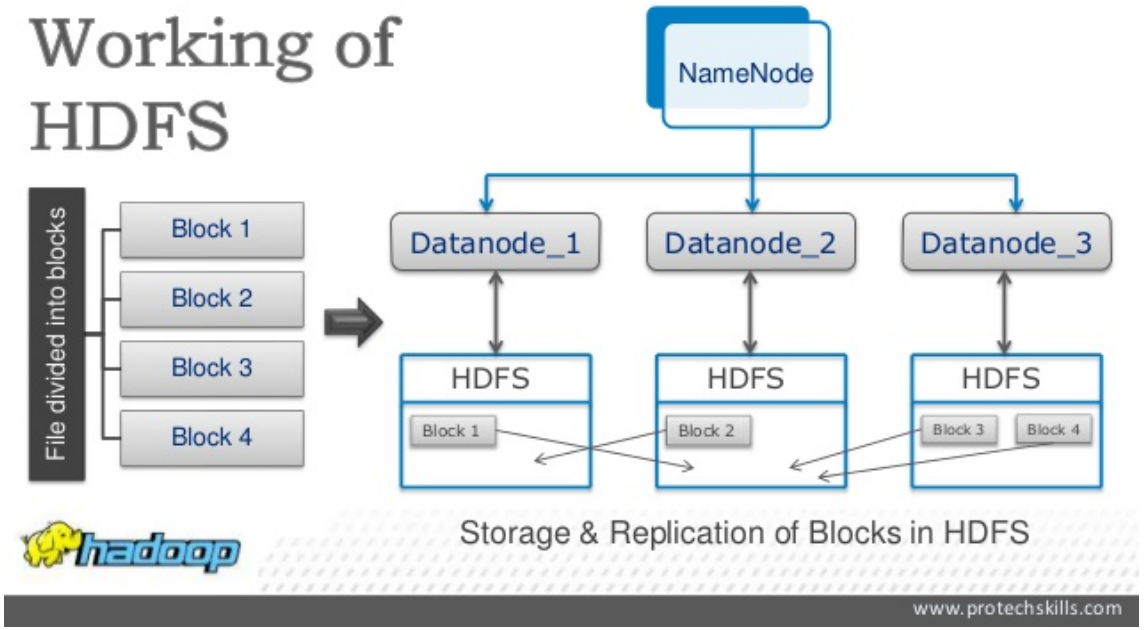
Hadoop, Kesako ?

Au sens étendu, Hadoop c’est un écosystème

Principes d’une architecture HDFS

Hadoop Distributed File System :

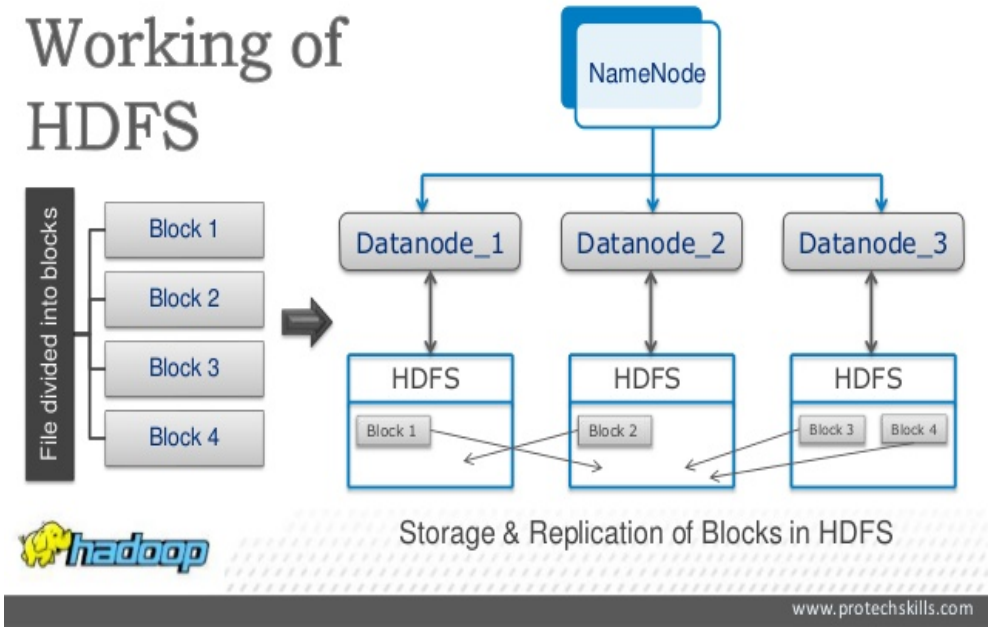
- **Cluster** : un ensemble de nodes
- **Nodes** : serveurs hadoop (bare metal, virtualisés, conteneur, etc)



Principes d’une architecture HDFS

Au moment ou un fichier est écrit sur HDFS :

- Il est découpé en **blocks** (classiquement de 64Mo)
- Chacun de ces blocks est stocké sur un data node
- Les **data nodes** renvoient leurs blocks à n autres data nodes (suivant le taux de réplication)
- Le **name node** sait où tout les blocks se trouvent



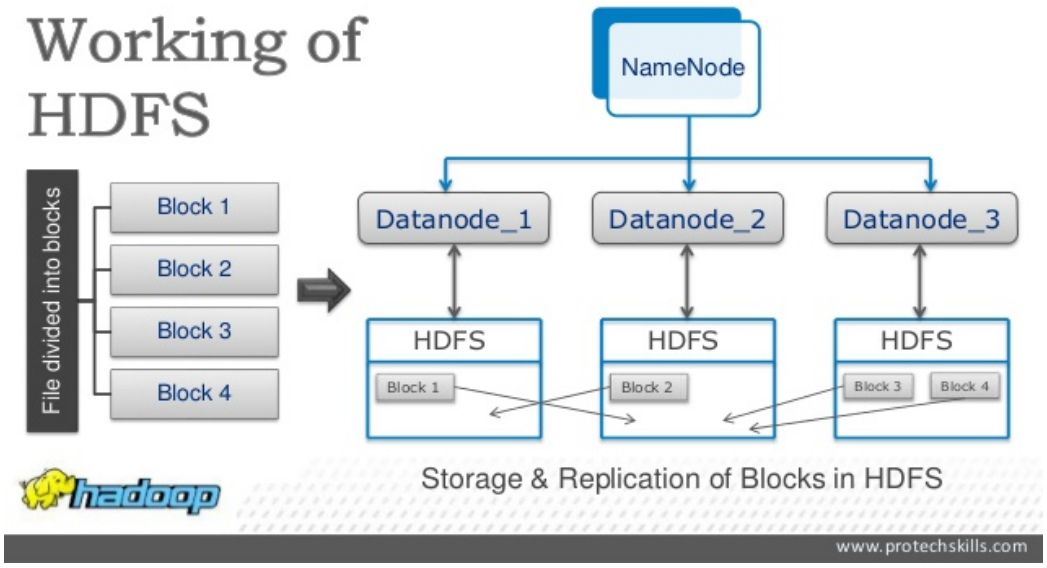
Principes d’une architecture HDFS

Que se passe t’il lorsque lorsque je tape ?

```
hdfs dfs -cat un_fichier_quelconque.txt
```

...la même chose « dans l’autre sens »

- Requête adressée au Name Node
- Identification des blocks et emplacement sur le cluster
- Agrégation (on parle aussi de désérialisation)



Principes d’une architecture HDFS

Remarques

- Un genre de **RAID** mais « between servers »
- HDFS est un **système de fichiers** au sens propre du terme (support des commandes usuelles : ls, cat, cd, droits de lectures et écriture, etc etc)
- Écriture **distribuée** : plus le cluster est gros, plus on lit / écrit vite (on parle de **scalabilité horizontale**)

Principes d’une architecture HDFS

Remarques

- Perdre un ou plusieurs **data-node** ne corrompt pas l’intégrité des données
- En revanche, la perte du **name-node** signifie la perte de l’intégralité des données (*single point of failure* → besoin de le dupliquer sur une plusieurs autre(s) machine(s))
- Un fichier stocké sur HDFS n’est pas **mutable**. Le modifier revient à le réécrire (important pour la suite)

Les avantages d’une architecture HDFS

A capacités égales, hardware moins coûteux

- Pas besoin de RAID
- Pas besoin de serveurs surpuissants : le **nombre** fait la force
- Déployable avec des **machines de monsieur tout le monde** (quatre coeurs, 8Go de RAM, 1 gros disque, 500€)
- Trois machines suffisent à déployer un cluster en une matinée

Installation de la stack

Nous allons commencer avec une distribution légère

https://github.com/zar3bski/hadoop-sandbox

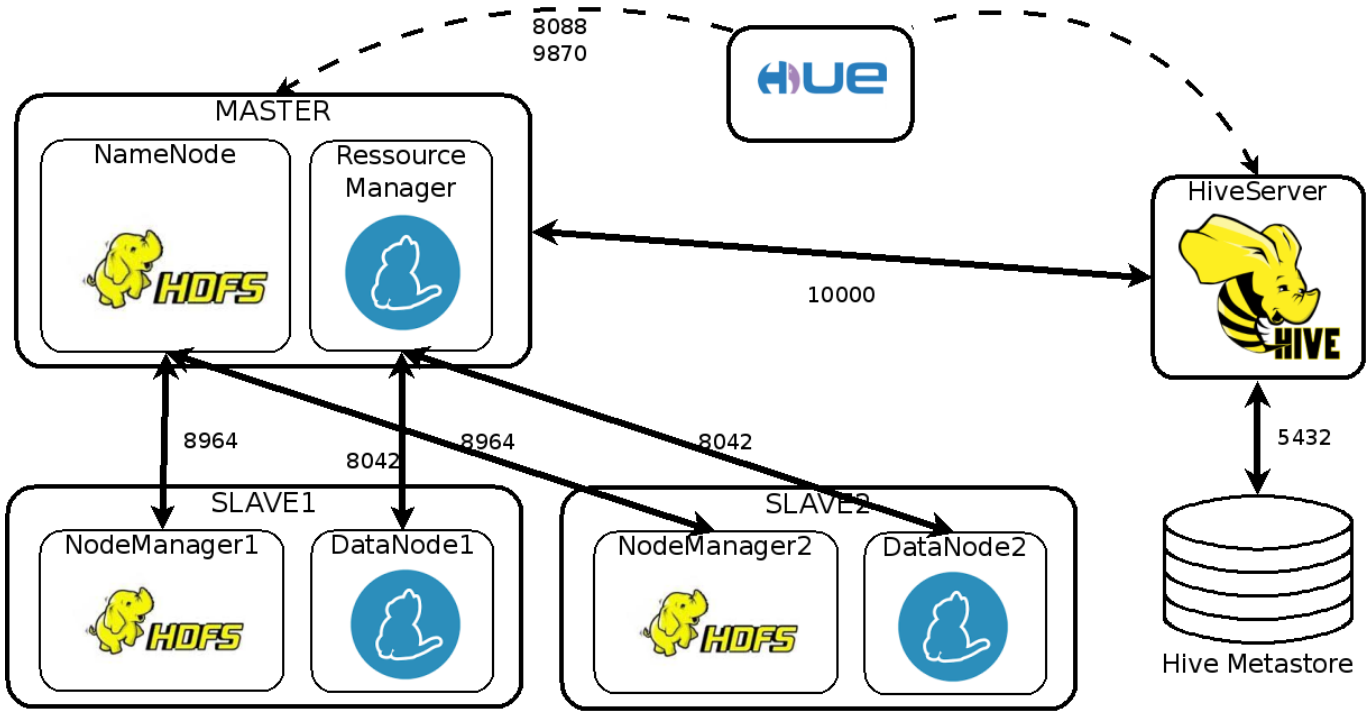
Présentation de la stack

Hadoop 3.1.1

Hue 4.4.0

Hive 2.3.2

(pour l'instant, à venir Sqoop, Spark 2, *etc*)



Prise en main des commandes HDFS de base

1) Se logger dans le NameNode

```
docker exec -it namenode bash
```

2) lister le contenu des fichiers

```
hdfs dfs -ls /
```

3) créer un dossier dans votre dossier d’utilisateur

```
hdfs dfs -mkdir /user/dav/download
```

4) téléchargez un document écrivez le sur HDFS

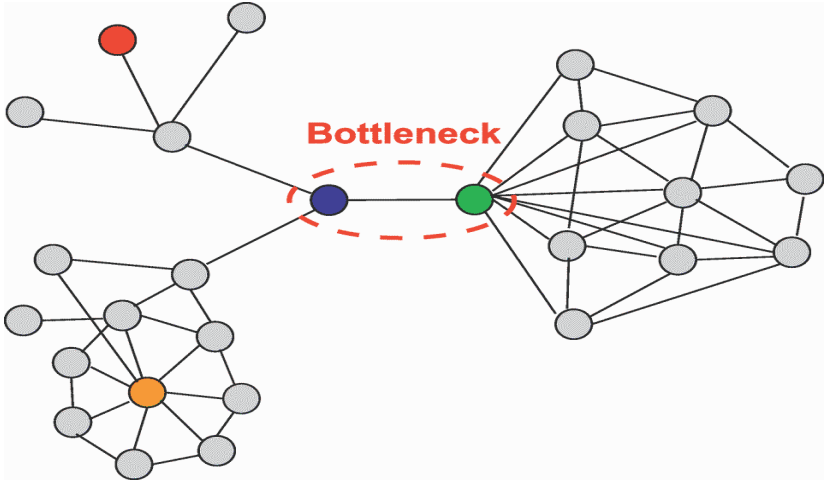
```
hdfs dfs -put fichier.txt /user/dav/download/fichier.txt
```

Exercice

- Téléchargez une ressource de votre choix et stockez en HDFS dans un dossier que seul vous (utilisateur créé par les variables d’environnements) pourrez lire, écrire et éxécuter

Distribuer le stockage, c’est bien mais quid des calculs et opérations ?

- Ça ne sert pas à grand-chose de distribuer nos données si leur traitement postérieur nécessite de tout agréger sur une machine
- La RAM coûte cher
- Source de bottleneck



Un premier Map Reduce

Notre document est écrit de manière distribuée sur l’ensemble de nos nœuds. Bien. Et si nous voulions compter le nombre de mots dans l’ensemble du document ? Deux manières de procéder :

- Sérielle : effectuer le décompte sur une machine
- Parallèle : compter les mots dans chaque morceaux du domument (MAP), en faire la somme par la suite (REDUCE) on appelle cela un **map reduce**

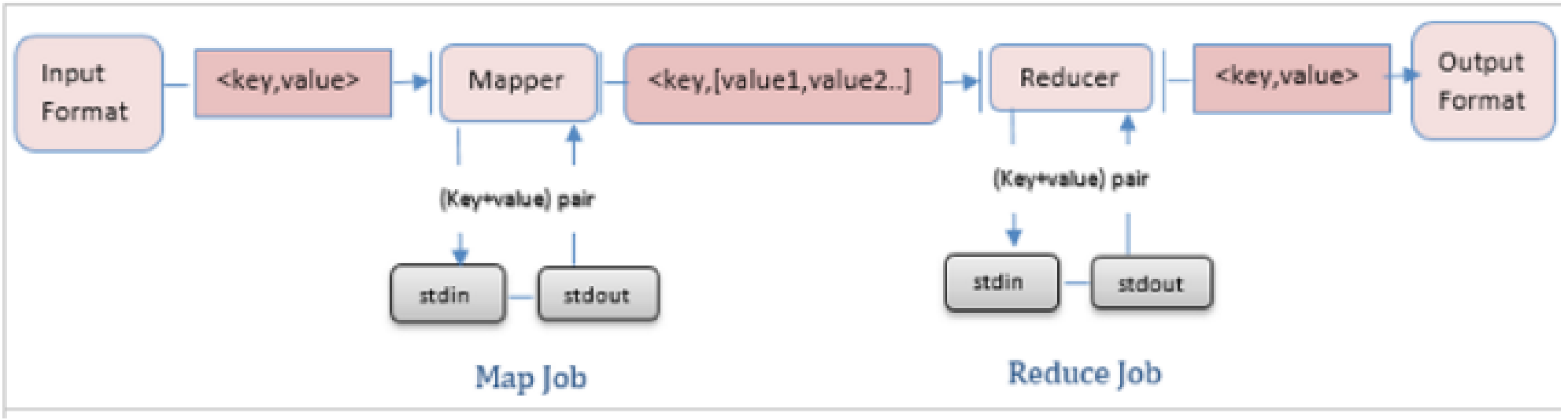
Exploiter les nombreux CPU d'un cluster : MapReduce

- La distribution de tâches nécessite un Ressource Manager
- **Yarn** (Yet Another Resource Negotiator)

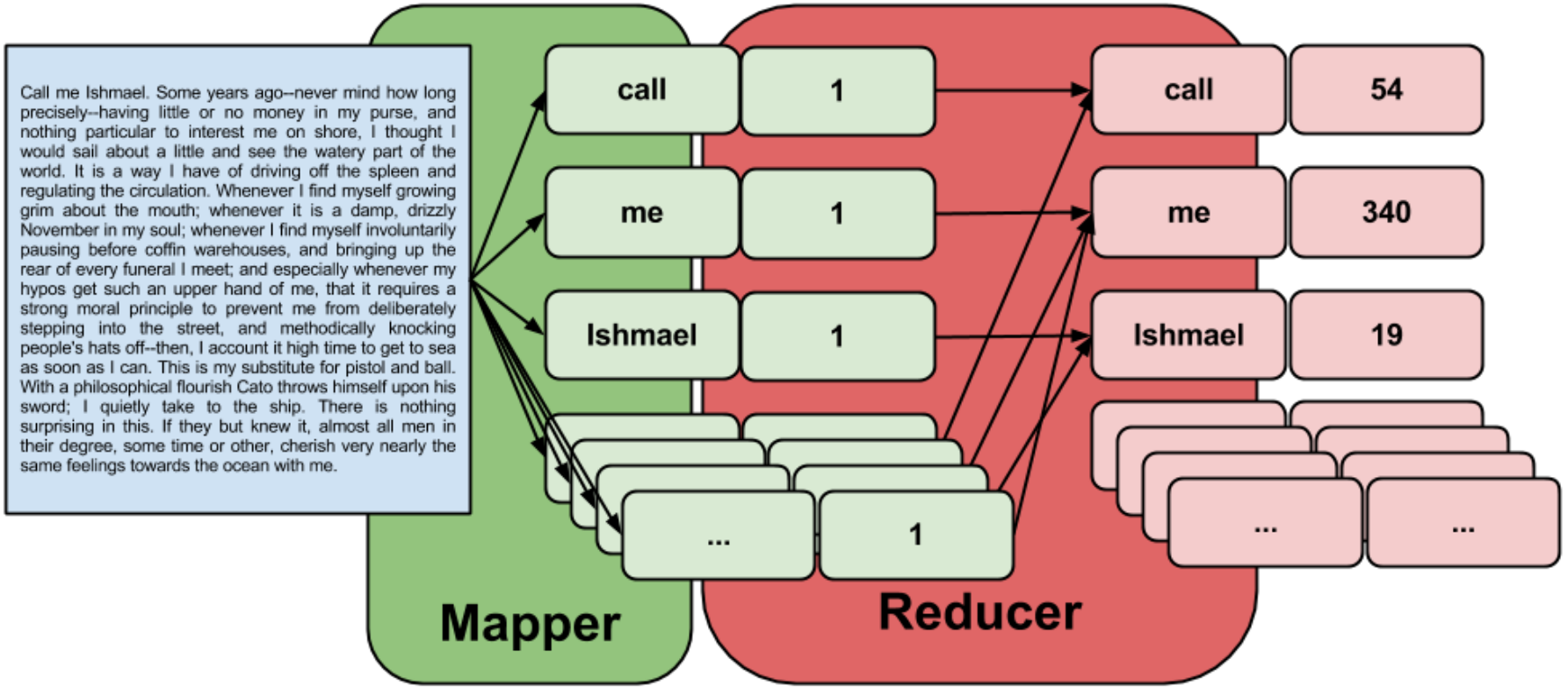
<u>Inconvénients principaux</u>

- Nécessite du code en java pour support natif
- Possibilité de pipeline mais avec un peu de bidouillage
- Nécessite d'intégrer **mappers** et **reducers** dans le design pattern (tous les devs n'y sont pas habitués)

Map Reduce : principe générale



MapReduce : un bon vieux word counter



MapReduce : un bon vieux word counter

http://www.science.smith.edu/dftwiki/index.php/Hadoop_Tutorial_2_--_Running_WordCount_in_Python

Tester son programme en local

```
cat salammbo.txt | python3 mapper.py | python3 reducer.py
```

Lancement d’un map reduce

```
hadoop jar /opt/hadoop-3.1.1/share/hadoop/tools/lib/hadoop-streaming-3.1.1.jar
-file /home/dav/mapper.py
-mapper mapper.py
-file /home/dav/reducer.py
-reducer reducer.py
-input /user/dav/salammbo.txt
-output test
```

Erreur commune

Error: java.lang.RuntimeException: PipeMapRed.waitOutputThreads(): subprocess failed with code 127

→ à l’étape map : Le mapeur est mort en cours de route

Causes possibles :

- Il manque une librairie sur les workers
- Formats de stout/stdin

Voir les logs pour comprendre ce qui cloche

```
yarn logs -applicationId application_1563877728284_0002
```

(grep est votre meilleur ami)

MapReduce : allons plus loin

https://www.kaggle.com/cityofoakland/oakland-crime-911-calls-gun-incidents

De vraies données avec de vrais enjeux, de vrais questions

nous allons travailler sur prr-20055-2010.csv

MapReduce : allons plus loin

A l’aide de MapReduce:

- Dénombrer chaque type d’incident (Incident Type Id)
- Pour chaque aire géographique (Area Id), quel type d’incident est le plus courant ?
- Combien avons-nous d’incidents par aire géographique ?
- En moyenne (*attention, des maths*) est-ce qu’un aire géographique fait l’objet d’une priorité (Priority) particulière ? Cela tient-il à la nature des incidents ou est-ce décorrélé ?
- Agréger les noms de rues par aire géographique (ça risque d’être intéressant)

Ingestion des données avec Sqoop

Pour Postgreql

sqoop import --connect
'jdbc:postgresql://ip.ou.domaine:port/nom_de_la_BDD’
--username 'nom_de_l’utilisateur' -P --table 'nom_de_la_table'
--target-dir ‘dossier_de_stockage' --split-by
'colonne_contenant_un_identifiant_unique'

Pour MySQL (ou autre), il suffirait d’utiliser un autre jdbc (e.g. jdbc:mysql, jdbc:oracle) pour peu que celui-ci ait été ajouté aux librairies de sqoop (demander à son admin-sys préféré ;-)