Réalisez des calculs distribués sur des données massives

20 heures Moyenne

Mis à jour le 08/04/2020

Familiarisez-vous avec Hadoop

Connectez-vous ou inscrivez-vous gratuitement pour bénéficier de toutes les fonctionnalités de ce cours !

Dans les chapitres précédents, nous avons vu les principes du modèle de programmation **MapReduce** et, au travers d'exemples, la logique de conception d'algorithmes selon ce modèle.

Nous allons maintenant revenir au contexte Big Data dans lequel il a tout son intérêt car il permet le passage à l'échelle de traitements sur de gros volumes de données.

Cependant, il faut pour cela qu'il soit associé à **une infrastructure logicielle dédiée** qui permette d'exécuter le schéma MapReduce de manière massivement distribuée sur un cluster de machines tout en prenant à sa charge les enjeux du calcul distribué :

- l'optimisation des transfert disques et réseau en limitant les déplacements de données (*data locality*),
- la scalabilité pour permettre d'adapter la puissance au besoin (*scalability*),
- et enfin la tolérance aux pannes (*embracing failure*).

Dans ce chapitre, nous allons donc nous intéresser au framework **Hadoop** de la fondation Apache, écrit en java, et qui constitue l'implémentation libre de référence d'une telle infrastructure. C'est un framework très largement utilisé et porté, entre autres, par les géants du web.

La petite histoire d'Hadoop

L'anecdote dit que Hadoop, au départ, c'est le nom de ce petit éléphant en peluche !

Ce petit éléphant en peluche appartenait au fils de Doug Cutting, l'un des concepteurs du framework Hadoop. Voilà pourquoi le logo de Hadoop est un éléphant !

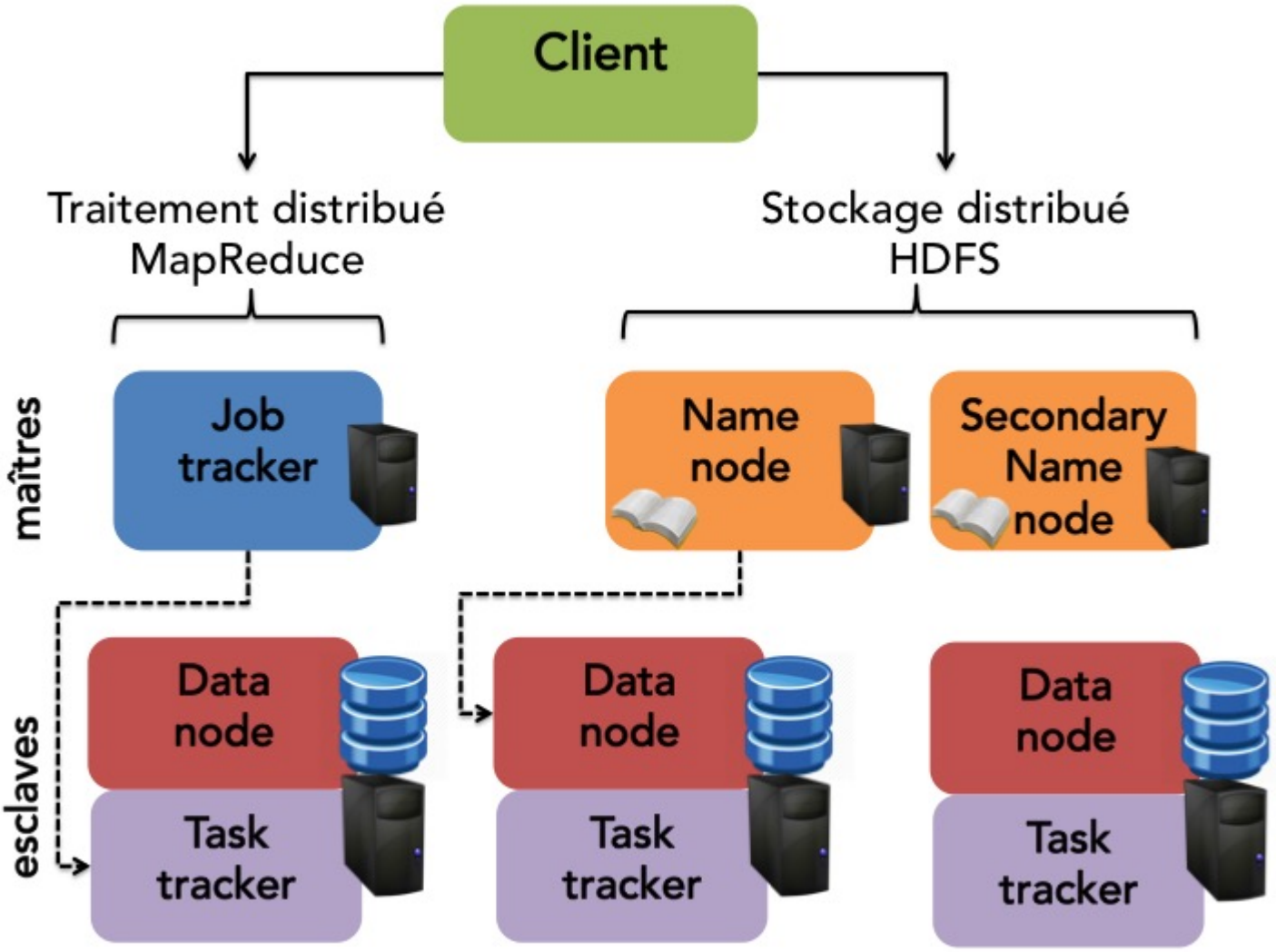
En 2002, Doug Cutting et Mike Cafarella, deux ingénieurs, décident de s'attaquer au passage à l'échelle de Lucene, le moteur de recherche open source. L'objectif était de le rendre capable d'indexer et de rechercher dans des collections de la taille du Web. C'est le projet Nutch. Pour cela, ils s'inspirent de deux articles de recherche publiés par les Google Labs. Le premier article décrit *Google File System*, un

système de fichiers distribués propriétaire developpé par Google et permettant de stocker de gros volumes de données de manière fiable sur des clusters. Nous avons déjà parlé du deuxième article, proposant le modèle de programmation MapReduce. L'architecture de Nutch, qui repose donc sur un système de fichiers distribué et sur MapReduce, est relativement générique et donnera lieu au projet Hadoop, initié en 2006. Il rejoint la fondation Apache en 2008. La version stable actuelle est la version 2.7.

Le socle technique d'Hadoop

Le socle technique d'Hadoop est composé :

- De toute l'architecture support nécessaire pour l'orchestration de MapReduce, c'est-à-dire :
  - l'ordonnancement des traitements,
  - la localisation des fichiers,
  - la distribution de l'exécution.
- D'un système de fichiers **HDFS** qui est :
  - Distribué : les données sont réparties sur les machines du cluster.
  - Répliqué : en cas de panne, aucune donnée n'est perdue.
  - Optimisé pour la colocalisation des données et des traitements.



Nous allons maintenant rentrer dans le détail des différents composants de ce socle technique.

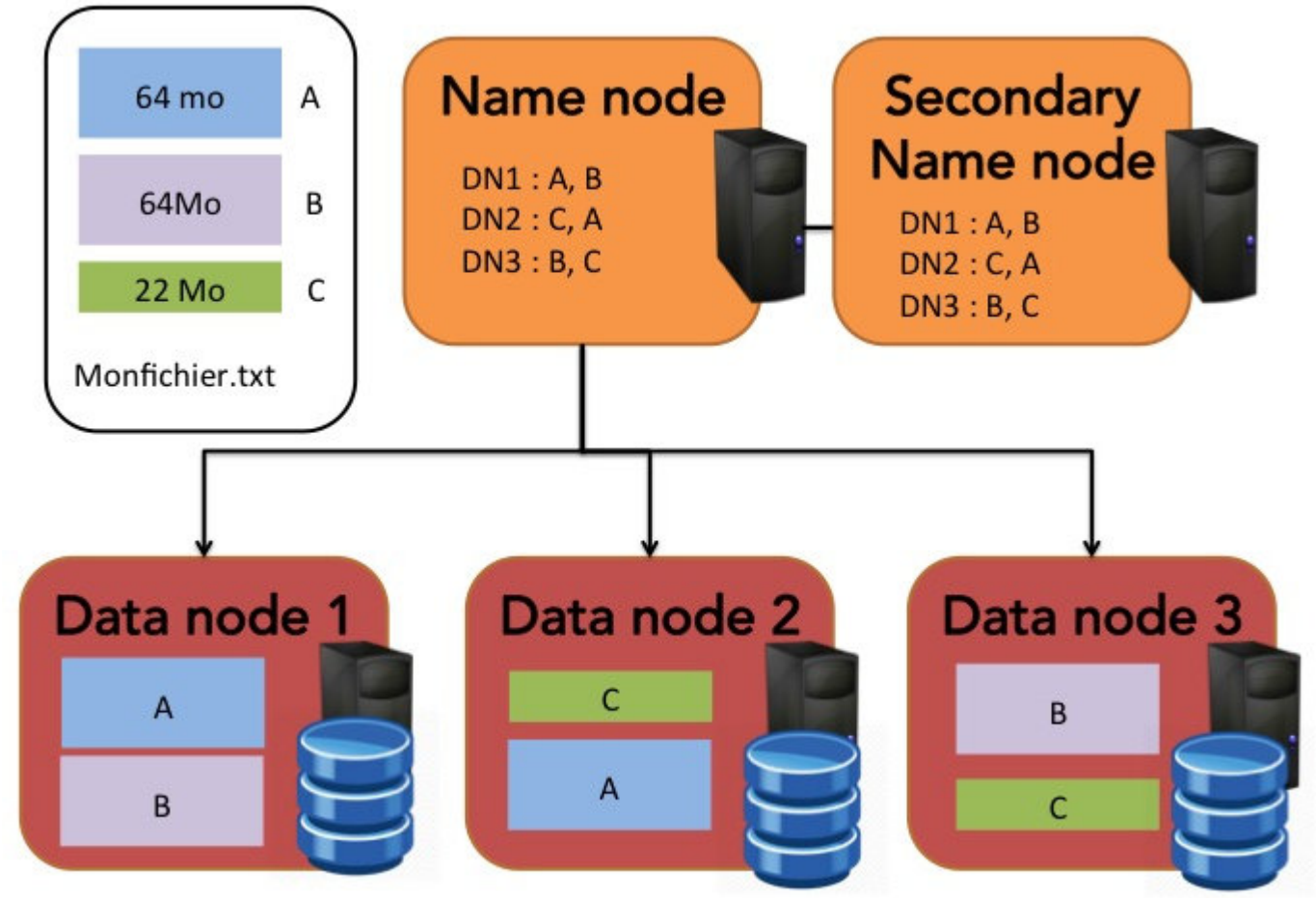
HDFS

HDFS (*Hadoop Distributed File System*) est un système de fichiers distribué et la couche native de stockage et d'accès à des données d'Hadoop. Il a été conçu pour stocker des fichiers de très grande taille et, comme son nom l'indique, dans un cadre distribué. Nous reviendrons plus en détails sur HDFS dans un prochain cours. Pour le moment, il vous suffit de savoir que dans HDFS :

- les fichiers sont physiquement **découpés en blocs d'octets de grande taille** (par défaut 64 Mo) pour optimiser les temps de transfert et d'accès ;
- ces blocs sont ensuite **répartis** sur plusieurs machines, permettant ainsi de traiter un même fichier en parallèle. Cela permet aussi de ne pas être limité par la capacité de stockage d'une seule machine pour au contraire tirer parti de tout l'espace disponible du cluster de machines ;
- enfin, pour garantir une tolérance aux pannes, les blocs de chaque fichier sont **repliqués**, de manière intelligente, sur plusieurs machines.

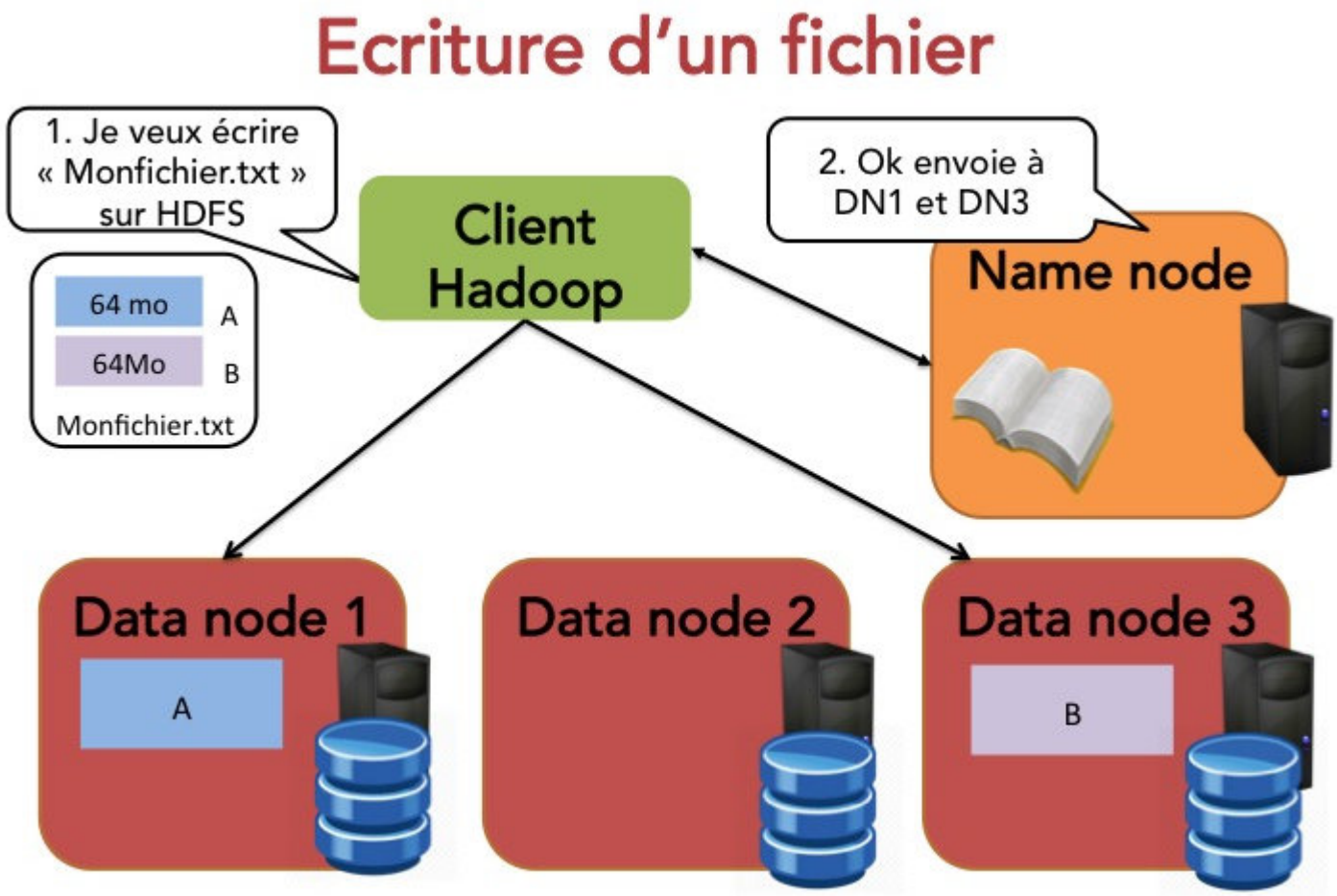
Dans Hadoop, l'architecture de stockage est une architecture *maître-esclave*.

- Le nœud maitre appelé **name node** contient et stocke tous les noms et blocs des fichiers ainsi que leur localisation dans le cluster. On peut donc le voir comme un gros annuaire.
- Une autre machine, appelée **secondary name node** sert de namenode de secours en cas de défaillance du nœud maître et il a donc pour rôle de faire des sauvegardes régulières de l'annuaire.
- Les autres nœuds, les esclaves, sont les nœuds de stockage en tant que tels. Ce sont les **data nodes** qui ont pour rôle la gestion des opérations de stockage locales (création, suppression et réplication de blocs) sur instruction du name node.



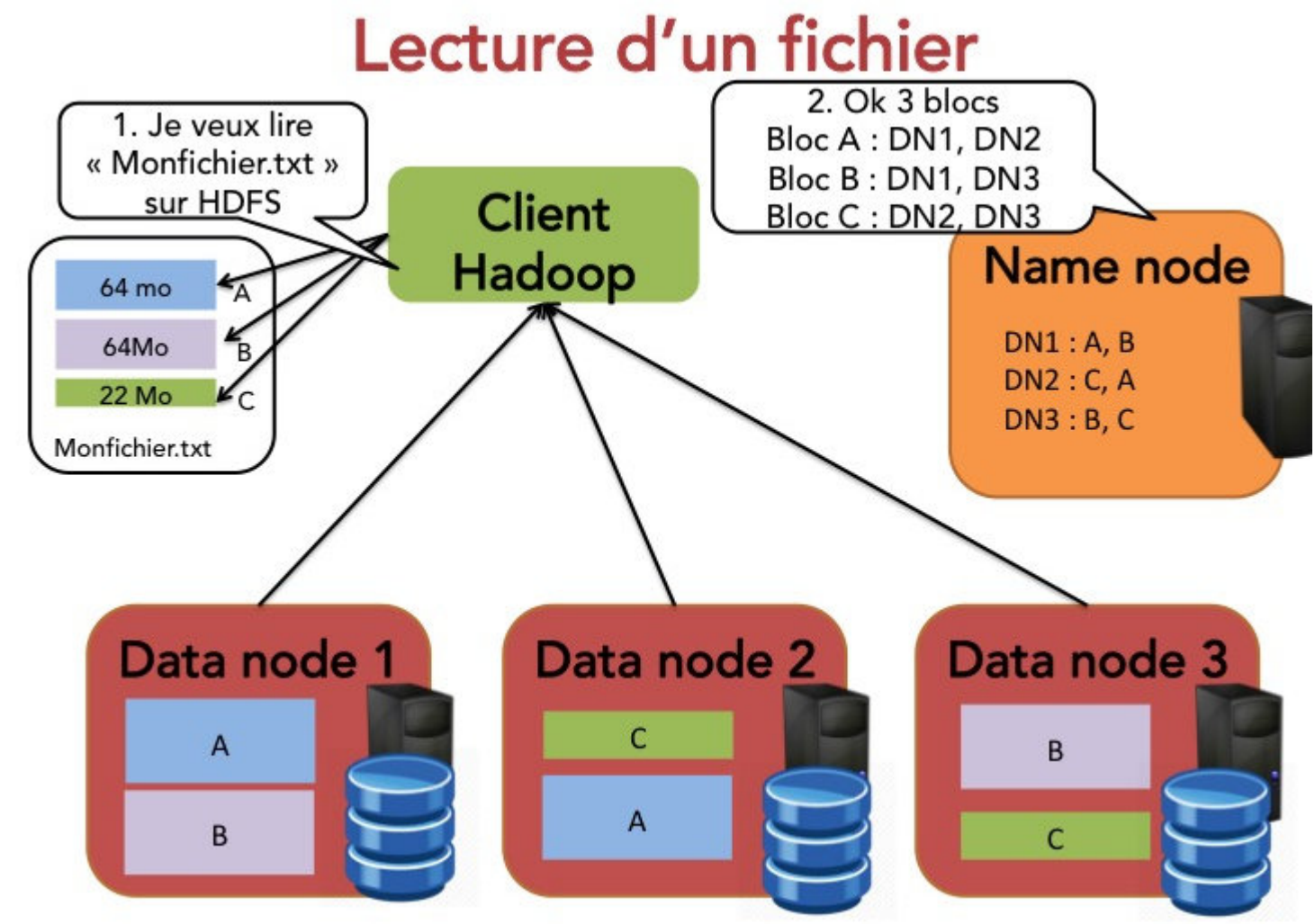
Si on souhaite écrire un fichier dans HDFS, on utilise un client Hadoop. Le principe est assez simple :

1. Le client indique au name node qu'il souhaite écrire un bloc.
2. Le name node indique le data node à contacter.
3. Le client envoie le bloc au data node.
4. Les data nodes répliquent les blocs entre eux.
5. Le cycle se répète pour le bloc suivant.



Si on souhaire lire un fichier dans HDFS, c'est également assez simple.

1. Le client indique au name node qu'il souhaite lire un fichier.
2. Le name node indique sa taille ainsi que les différents data nodes contenant les blocs.
3. Le client récupère chacun des blocs sur l'un des data nodes.
4. Si le data node est indisponible, le client en contacte un autre.



Manipuler HDFS

Il y a deux possibilités pour manipuler HDFS :

- Soit via l'API Java, que nous ne décrirons pas ici.
- Soit directement depuis un terminal via les commandes

sh

```
$ hdfs dfs <commande hdfs="" />
```

ou :

sh

```
$ hadoop fs <commande HDFS>
```

En particulier, les commandes principales sont :

python

```
hdfs dfs -help
hdfs dfs -ls <path>
hdfs dfs -mv <src><dst>
hdfs dfs -cat <src>
hdfs dfs -copyFromLocal <localsrc> ... <dst>
```

```
hdfs dfs -put <localsrc> ... <dst>
hdfs dfs -mkdir <path>
hdfs dfs -copyToLocal <src><localdst>
hdfs dfs -rm -f -r <path>
```

N'hésitez pas à vous référer à la **documentation officielle** pour comprendre ces différentes commandes. Et pour aller plus loin vous pouvez consulter cet **article sur le fonctionnement de HDFS**.

Hadoop MapReduce

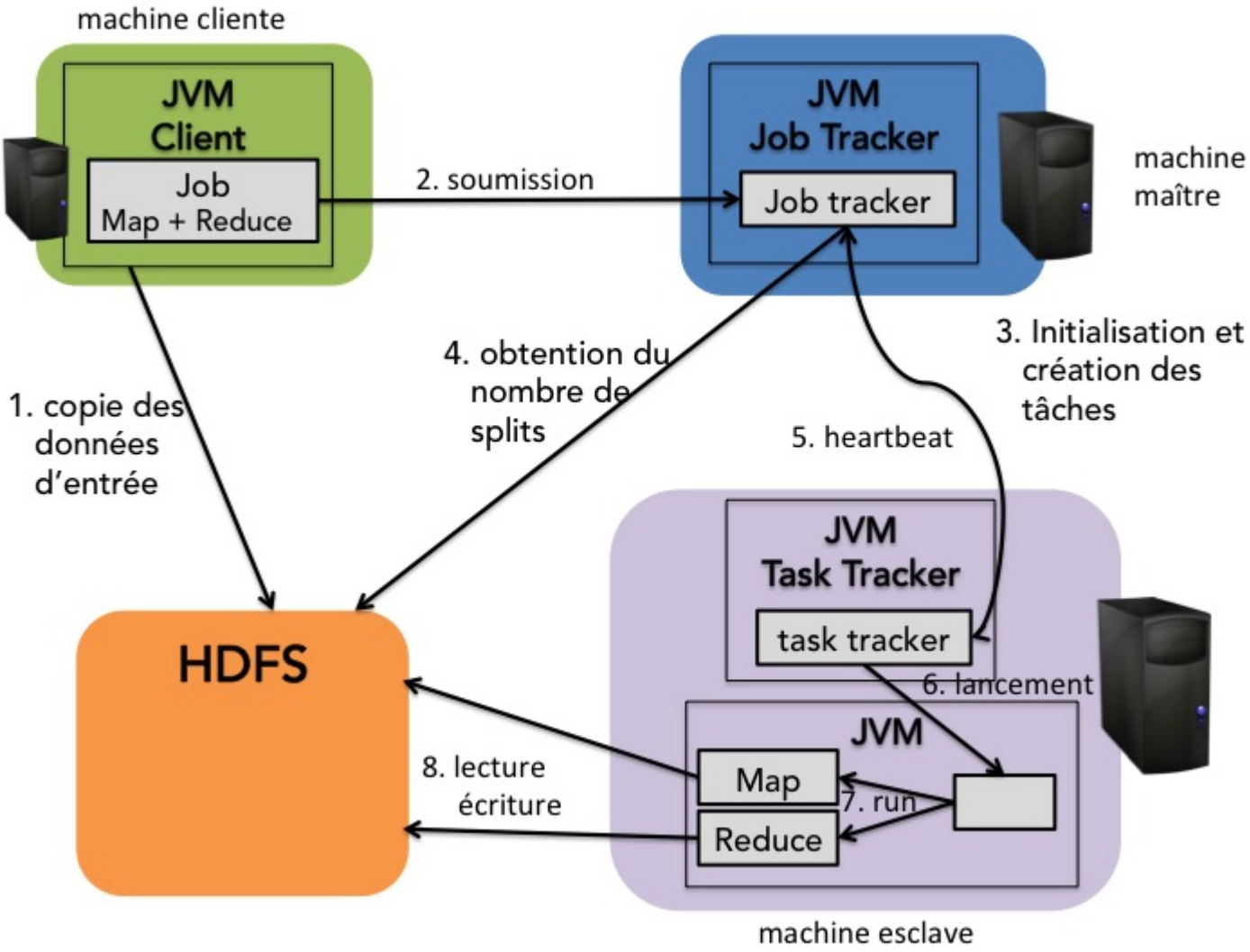
Dans les chapitres précédents concernant le modèle de programmation MapReduce, nous avons volontairement laissé quelques questions importantes en suspens. En particulier, à ce stade du cours, nous savons reformuler une tâche en MapReduce. Si nous disposons de données massives sur lesquelles appliquer cette tâche et un cluster, nous savons maintenant aussi répartir le stockage de ces données sur le cluster via HDFS. C'est très bien, tout cela mais :

- Comment ordonnancer les traitements ?
- Comment distribuer l'exécution sur les différents nœuds du cluster ?
- Comment connaître l'emplacement des fichiers à traiter ?

C'est bien entendu Hadoop qui va s'occuper de tout cela pour nous, à nouveau avec une architecture de type **maître-esclave**. Dans cette architecture :

- Le **job tracker** est un processus maître qui va se charger de l'ordonnancement des traitements et de la gestion de l'ensemble des ressources du système. Il reçoit (du client) la ou les tâches MapReduce à exécuter (un .jar Java) ainsi que les données d'entrée et le répertoire où stocker les données de sorties. Il est pour cela en communication avec le **name node** d'HDFS. Le job tracker est en charge de planifier l'exécution des tâches et de les distribuer sur des **task trackers**. Comme il sait où sont situés les blocs de données, il peut optimiser la colocalisation traitements/données.
- Un **task tracker** est une unité de calcul du cluster. Il assure, en lançant une nouvelle machine virtuelle java (JVM), l'exécution et le suivi des tâches MAP ou REDUCE s'exécutant sur son nœud et qu'il reçoit du **job tracker**. Il dispose d'un nombre limité de slots d'exécution et donc un nombre limité de tâches MAP, REDUCE ou SHUFFLE pouvant s'éxécuter simultanément sur le nœud. Il est aussi en communication constante avec le **job tracker** pour l'informer de l'état d'avancement des tâches (*heartbeat call*). Et oui, nous sommes toujours confrontés au problème de la tolérance aux pannes car en cas de défaillance, le **job tracker**, informé ou sans nouvelle du task tracker, doit pouvoir ordonner la réexécution de la tâche.

Voici le schéma de soumission et d'exécution d'un job dans Hadoop MapReduce :



1. Un client hadoop copie ses données sur HDFS.
2. Le client soumet le travail à effectuer au **job tracker** sous la forme d'une archive `.jar` et des noms des fichiers d'entrée et de sortie.
3. Le **job tracker** demande au **name node** où se trouvent les blocs correspondants aux données d'entrée.
4. Il détermine alors quels sont les nœuds **Task Tracker** les plus appropriés pour exécuter les traitements (colocalisation ou proximité des nœuds). Il envoie alors au **task tracker** selectionné et pour chaque bloc de données, le travail à effectuer (Map, Reduce ou Shuffle, fichier .jar).
5. Les **task trackers** envoient régulièrement un message (*hearbeat*) au **job tracker** pour l'informer de l'avancement de la tâche et de leur nombre de slots disponibles.
6. Quand toutes les opérations envoyées aux **task trackers** sont confirmées comme étant effectuées, la tâche est considérée comme effectuée.

Mais, finalement, que reste-t-il comme travail au développeur ? Et bien effectivement, il pourra se contenter :

- d'écrire les programmes MAP et REDUCE et d'en faire une archive `.jar`
- de soumettre les fichiers d'entrée, le répertoire de sortie et le `.jar` au **job tracker**.

API Hadoop MapReduce

Hadoop est écrit en Java et fournit donc des interfaces Java pour l'écriture des programmes MapReduce. Nous verrons un peu plus loin que d'autres langages peuvent être utilisés via Hadoop Streaming.

Avec l'API java, écrire un programme MapReduce consiste à écrire trois classes :

1. Une classe `Map`, implémentant la classe `org.apache.hadoop.Mapper` d'Hadoop que l'on paramètre avec le type de la clé d'entrée (`TypeCleE`), le type de la valeur d'entrée (`TypeValE`), le type de la clé des sorties intermédiaires (`TypeCleI`) et enfin le type de la valeur des sorties intermédiaires (`TypeValI`) et qui est en charge de l'opération MAP correspondant à notre problème en surchargeant la fonction `map` de `Mapper`.
   Nous donnons ci-dessous le squelette de cette classe.

```java
package ocr.dataArchitect.cours1.hadoop.exempleMapReduce;

import org.apache.hadoop.mapreduce.Job;
import org.apache.hadoop.io.*;
import org.apache.hadoop.mapreduce.Mapper;
import java.io.IOException;

// A compléter selon le problème
public class ExempleMap extends Mapper<TypeCleE, TypeValE, TypeCleI, TypeValI> {
// Écriture de la fonction map
@Override
protected void map(TypeCleE cleE, TypeValE valE, Context context) throws IOException,InterruptedException
    {
        // À compléter selon le probleme
        // traitement : cleI = ..., valI = ...
        TypeCleI cleI = new TypeCleI(...);
        TypeValI valI = new TypeValI(...);
        context.write(cleI,valI);
    }
}
```

   Nous venons de voir que cette classe est paramètrée par 4 types. Nous ne pouvons pas utiliser pour ces types, les types standard de Java. Il faut utiliser des types spéciaux qui vont permettre la transmission efficace des données entre les différentes machines du cluster.

   - Les valeurs doivent implémenter l'interface `Writable` de l'API Hadoop qui permet la sérialisation et la désérialisation (et oui! les machines doivent s'échanger des données).
   - Les clés doivent implémenter l'interface `WritableComparable<T>`.

Bien évidemment, plusieurs types sont déjà prédéfinis dans l'API Hadoop.

type	description
`Text`	chaîne UTF8
`BooleanWritable`	booléen
`IntWritable`	entier 32 bits
`LongWritable`	entier 64 bits
`FloatWritable`	réel IEEE 32 bits
`DoubleWritable`	réel IEEE 64 bits

2. Une classe `Reduce`, implémentant la classe `org.apache.haddop.Reducer` d'Hadoop que l'on paramètre avec 4 types comme pour `Mapper` (deux types étant même identiques) et qui est en charge de l'opération REDUCE correspondant à notre problème en surchargeant la fonction `reduce` de `Reducer`.

   Nous donnons ci-dessous le squelette de cette classe.

java

```java
package ocr.dataArchitect.cours1.hadoop.exempleMapReduce;

import org.apache.hadoop.mapreduce.Job;
import org.apache.hadoop.io.*
import org.apache.hadoop.mapreduce.Reducer;
import java.io.IOException;
import jave.io.Iterable;

// A compléter selon le problème
public class ExempleReduce extends Reducer<TypeCleI,TypeValI,TypeCleS,TypeValS> {
    // Écriture de la fonction reduce
    @Override
    protected void reduce(TypeCleI cleI, Iterable<TypeValI> listevalI, Context context) throws
IOException,InterruptedException
    {
        // À compléter selon le probleme
        TypeCleS cleS = new TypeCleS(...);
        TypeValS valS = new TypeValS(...);
        for (TypeValI val: listevalI) {
            // traitement cleS.set(...), valS.set(...)
        }
        context.write(cleS,valS);
    }
}
```

3. Une classe `MyProgram` (ici `ExempleMapReduce`) qui contient la fonction `main` du programme et qui va permettre de :
    - Récupérer la configuration générale du cluster.
    - Créer un job.
    - Préciser quelles sont les classes Map et Reduce du programme.
    - Préciser les types de clés et de valeur correspondant à notre problème (attention souvent des types propres à Hadoop).
    - Indiquer où sont les données d'entrée et de sortie dans HDFS.
    - Lancer l'exécution de la tâche

java

```java
package ocr.dataArchitect.cours1.hadoop.exempleMapReduce;

import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.conf.Configured;
import org.apache.hadoop.fs.FileSystem;
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.mapreduce.Job;
import org.apache.hadoop.mapreduce.lib.input.FileInputFormat;
import org.apache.hadoop.mapreduce.lib.input.TextInputFormat;
import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;
import org.apache.hadoop.mapreduce.lib.output.TextOutputFormat;
import org.apache.hadoop.util.GenericOptionsParser;
import org.apache.hadoop.util.Tool;
import org.apache.hadoop.util.ToolRunner;

public class ExempleMapReduce extends Configured implements Tool {

    public int run(String[] args) throws Exception {
        if (args.length != 2) {
            System.out.println("Usage: [input] [output]");
            System.exit(-1);
        }

        // Création d'un job en lui fournissant la configuration et une description textuelle de la tâche
        Job job = Job.getInstance(getConf());
        job.setJobName("notre probleme exemple");

        // On précise les classes MyProgram, Map et Reduce
        job.setJarByClass(ExempleMapReduce.class);
        job.setMapperClass(ExampleMap.class);
        job.setReducerClass(ExempleReduce.class);

        // Définition des types clé/valeur de notre problème
        job.setMapOutputKeyClass(TypecleI.class);
        job.setMapOutputValueClass(TypevalI.class);

        job.setOutputKeyClass(TypeCleS.class);
        job.setOutputValueClass(TypeValS.class);
```

```
39
40          // Définition des fichiers d'entrée et de sorties (ici considérés comme des arguments à
    préciser lors de l'exécution)
41          FileInputFormat.addInputPath(job, new Path(ourArgs[0]));
42          FileOutputFormat.setOutputPath(job, new Path(ourArgs[1]));
43
44          //Suppression du fichier de sortie s'il existe déjà
45          FileSystem fs = FileSystem.newInstance(getConf());
46          if (fs.exists(outputFilePath)) {
47              fs.delete(outputFilePath, true);
48          }
49
50          return job.waitForCompletion(true) ? 0: 1;
51      }
52
53      public static void main(String[] args) throws Exception {
54          ExempleMapReduce exempleDriver = new ExempleMapReduce();
55          int res = ToolRunner.run(exempleDriver, args);
56          System.exit(res);
57      }
58  }
```

HADOOP2 et YARN

Avant de parler de Yarn, essayons de faire un petit bilan sur ce que nous avons vu !

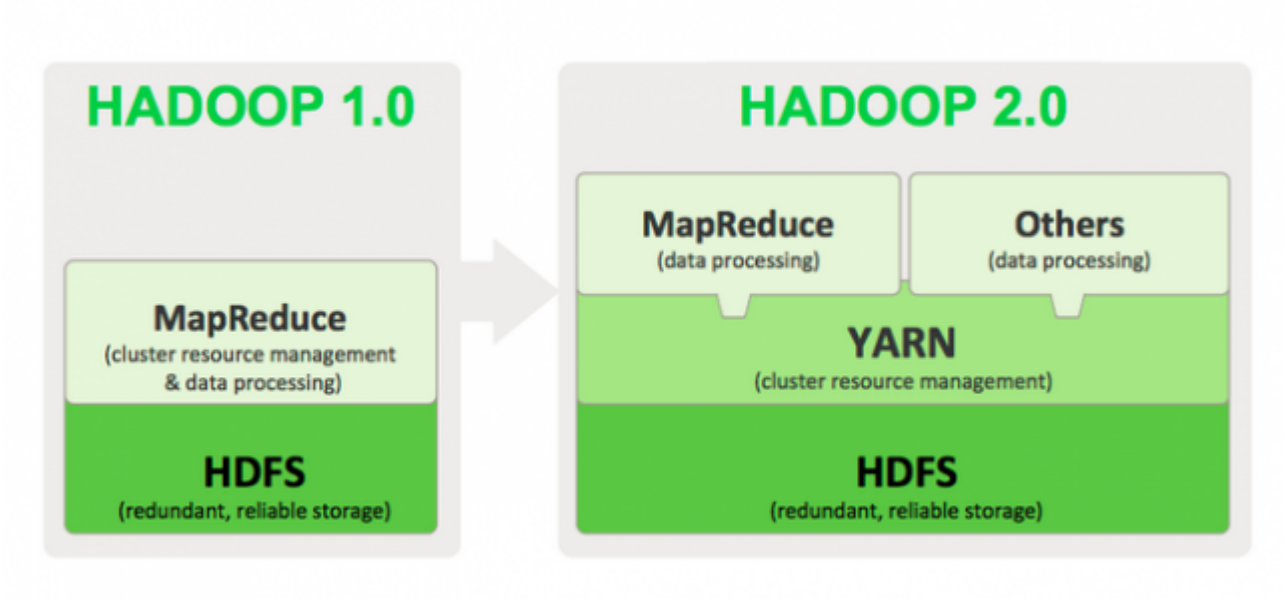
Nous avons vu que l'algorithme MapReduce permet d'implémenter de nombreux types de traitement en vue de leur parallélisation. Pour autant, tous les problèmes rentrent-ils dans le moule MapReduce, très rigide ?

- Non pas forcement et très souvent, si c'est le cas, cela demande beaucoup d'efforts de transformer un algorithme en MapReduce.
- De même, pour traiter des problèmes complexes, les deux étapes MAP et REDUCE ne suffisent pas, il est très souvent nécessaire d'enchaîner des séquences de MapReduce ce qui est très coûteux car cela nécessite de demarrer un job MapReduce à chaque fois.
- Enfin, si on s'interesse un peu plus à l'architecture d'Hadoop, on remarque que le job tracker a une double responsabilité :
    1. Il doit gérer les ressources du cluster.
    2. Il doit ordonnancer les jobs. Que se passe-t-il si le Job tracker est défaillant ?

Pour répondre à ces différents problèmes, plusieurs améliorations ont été apportées à Hadoop (version 2.x). Notamment, l'architecture d'Hadoop a été modifiée pour introduire YARN : *Yet Another Ressource Negociator*, un framework permettant d'exécuter n'importe quel type d'application distribuée sur un cluster Hadoop, pas uniquement les applications MapReduce.

YARN propose en effet de séparer la gestion des ressources du cluster et la gestion des jobs MapReduce, permettant ainsi de généraliser cette gestion des ressources à d'autres applications. L'idée

principale est de considérer que les nœuds ont des ressources (mémoire et CPU) qui seront allouées aux applications quand elles le demandent.



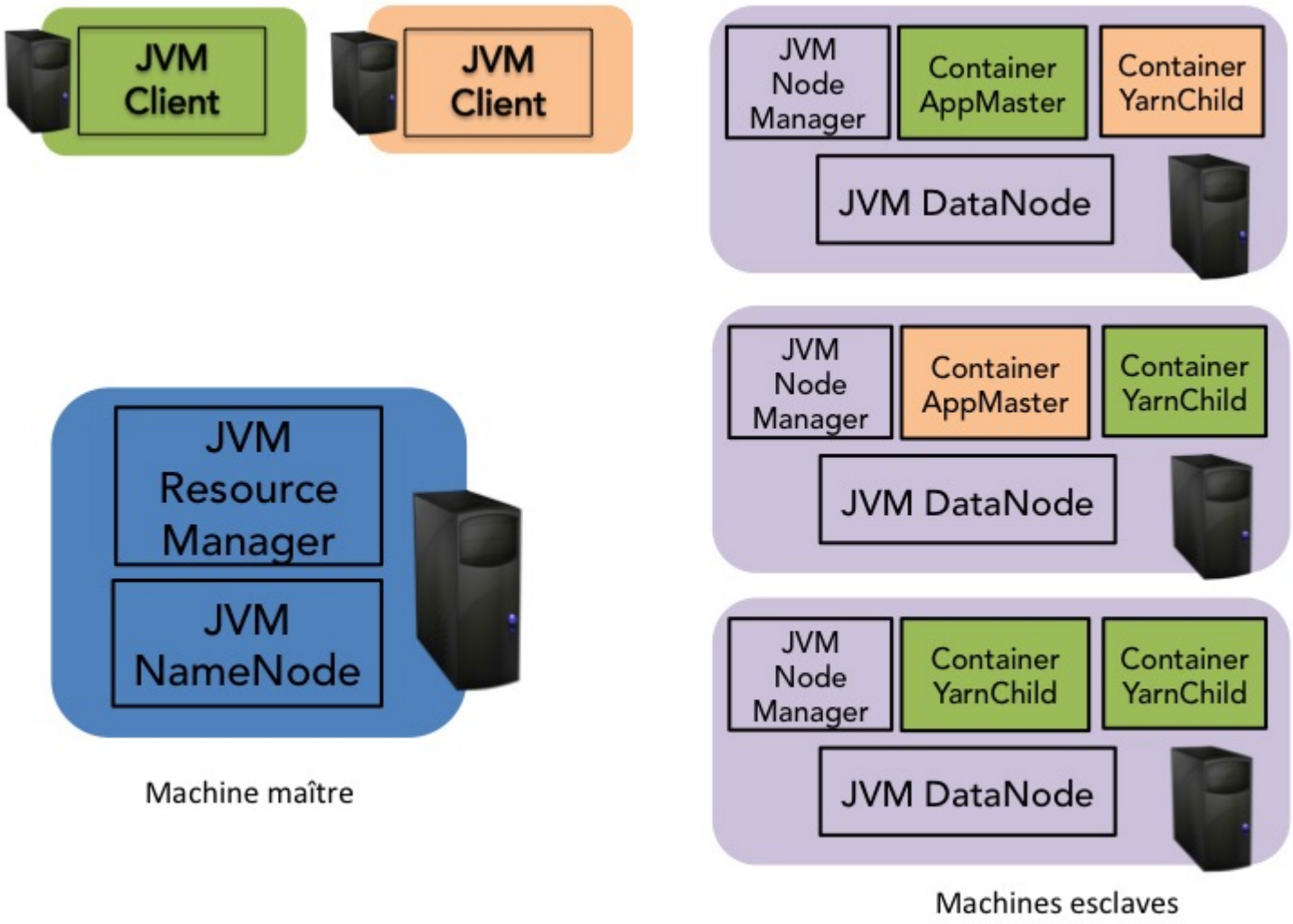
D'Hadoop 1.X à Hadoop 2.X avec YARN. La gestion des ressources est généralisée à d'autres applications que MapReduce (Source : HortonWorks)

En particulier, dans YARN, les fonctionnalités du job tracker sont réparties entre :

- Le **resource manager** qui est le chef d'orchestre des ressources du cluster. Il ordonnance les requêtes clients et pilote le cluster par l'intermédiaire de **node managers** qui s'exécutent sur chaque nœud de calcul. Il a donc pour rôle de contrôler toutes les ressources du cluster et l'état des machines qui le constituent. Il gère donc le cluster en maximisant l'utilisation de ressources.
- L'**application master (AM)** qui est un processus s'exécutant sur toutes les machines esclaves et qui gère, en discussion avec le **resource manager**, les ressources nécéssaires au travail soumis.

De même, les fonctionnalités du **task tracker** sont aussi réparties sur une même machine entre:

- Des **containers** qui sont des abstractions de ressources sur un nœud dédiées soit à l'exécution de tâches comme Map et Reduce, soit à l'exécution d'un **application master**.
- Un **node manager** qui héberge des **containers** et gére donc les ressources du nœud. Il est en communication via un *heartbeat* avec le ressource manager.



Architecture maître-esclave de Hadoop avec YARN

Le schéma de soumission et d'exécution d'un job dans cette nouvelle architecture est donc le suivant :

1. Un client hadoop copie ses données sur HDFS.
2. Le client soumet le travail à effectuer au **resource manager** sous la forme d'une archive `.jar` et des noms des fichiers d'entrée et de sortie.
3. Le **resource manager** alloue alors un container pour l'**application master** sur un **node manager**.
4. L'**application master** demande au **resource manager** un ou plusieurs containers avec des préférences de localisation dépendant de la localité des données d'entrée du travail.
5. Le **resource manager** alloue alors un ou plusieurs containers (child) à l'**application master**.
6. L'**application master** choisit parmi la liste des tâches (par exemple Map et Reduce) et demarre une instance de la tâche choisie dans un des **containers** qui lui a été alloué. Il collabore alors avec le **node manager** pour utiliser les ressources acquises. Il communique aussi souvent avec le **resource manager** (message *heartbeat*) pour la tolérance aux pannes.

Le schéma ci-dessous illustre le schéma simplifié de soumission et d'exécution d'un travail dans Hadoop 2.X avec YARN.

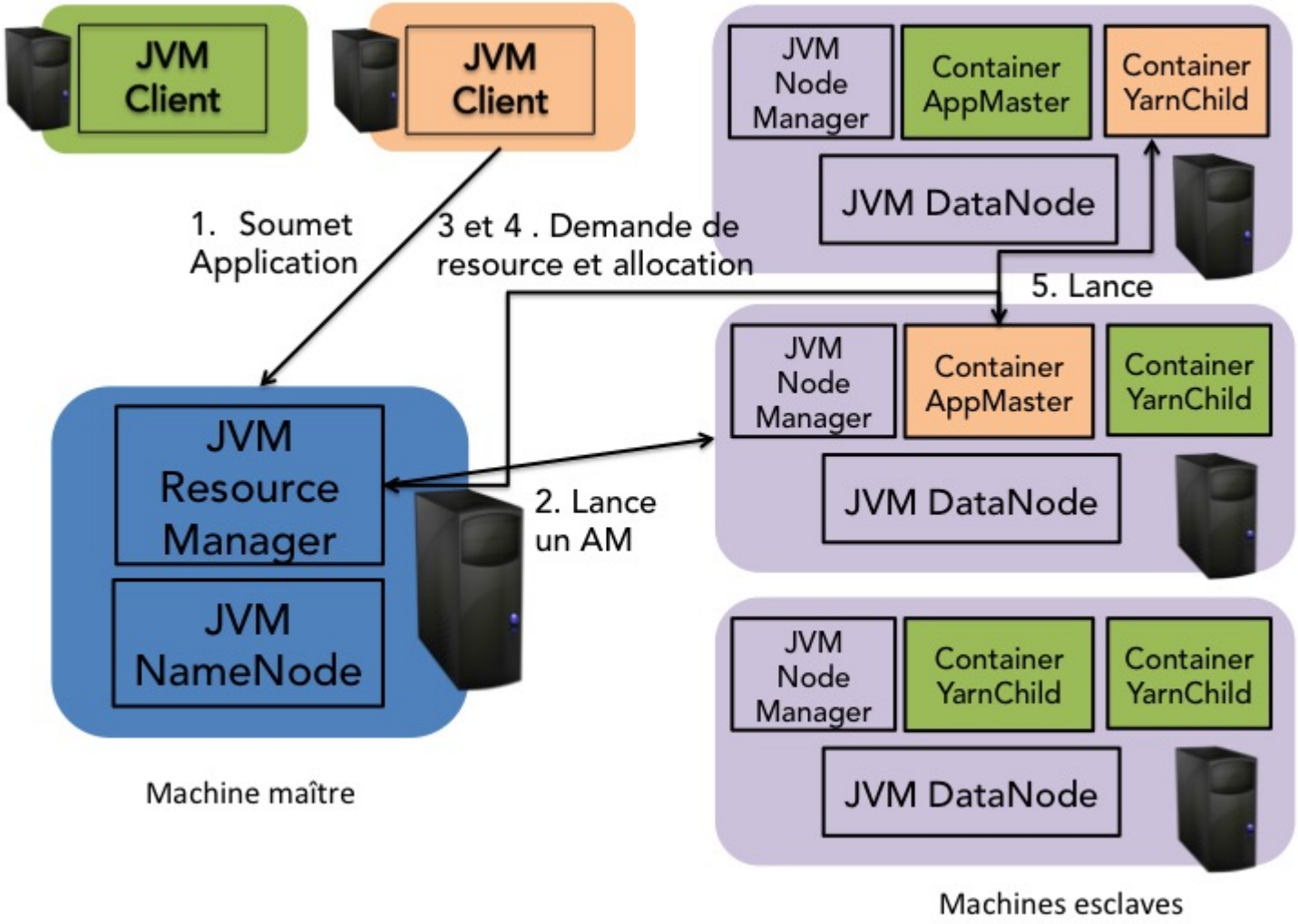
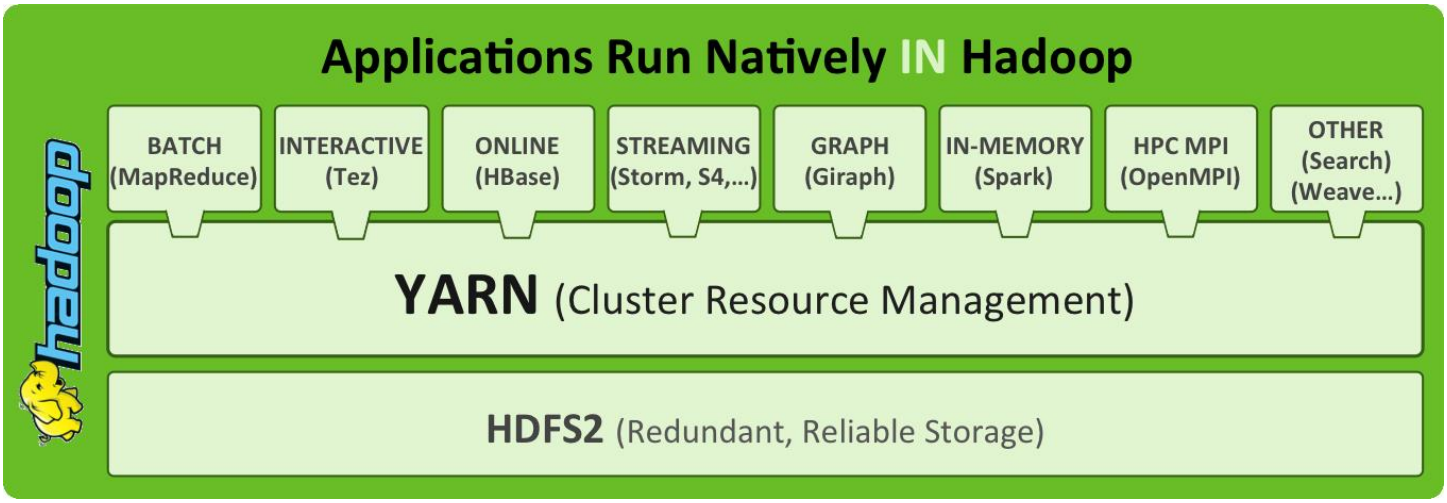


Schéma simplifié de l'exécution d'un travail dans Hadoop 2.X avec YARN.

Malin ! Et le gros avantage est que tout cela ne se limite donc plus à MapReduce et avec ce principe, de nombreuses applications peuvent s'exécuter, de manière native, sur un même cluster Hadoop parmi lesquelles :

- Spark : une solution pour le traitemen et l'analyse de données massives et que l'on découvrira dans la partie suivante de ce cours.
- Giraph: une solution pour faire des calculs sur graphes.
- HBase : une base de données NoSQL reposant sur HDFS. Nous détaillerons aussi ce type de base de données dans un prochain cours.
- Tez : un cadre pour l'écriture et l'exécution de traitements modélisés sous la forme de graphes dirigés acycliques (DAG) et qui facilite l'enchaînement des traitements.
- ...



Quelques applications pouvant s'exécuter de manière native sur Hadoop
(Source : HortonWorks)

Hadoop Streaming

Java n'est pas du tout votre langage préféré et vous vous dites que, pour déployer votre algorithme MapReduce sur Hadoop vous n'allez pas pouvoir y échapper ? Et bien, bonne nouvelle, il y a Hadoop Streaming. C'est un outil distribué avec Hadoop qui permet l'exécution d'un programme écrit dans d'autres langages, comme par exemple Python, C, C++...

En fait, Hadoop Streaming est un `.jar` qui prend en arguments :

- des programmes ou scripts définissant les tâches MAP et REDUCE (dans n'importe quel langage),
- les fichiers d'entrée et le répertoire de sortie HDFS.

Hadoop Streaming - MAP

Pour écrire un programme MAP pour Hadoop mais dans un autre langage, il faut que les données d'entrée soient lues sur l'entrée standard (`stdin`) et les données de sorties doivent être envoyées sur la sortie standard (`stdout`). On écrira donc notre série de paires `(clé, valeur)` , chaque paire sur une ligne différente, au format :

text

```
1 Clé[TABULATION]Valeur
```

Hadoop Streaming - REDUCE

Le même mécanisme doit être mis en place pour le programme REDUCE. Nous avons en entrée et en sortie du programme une série de lignes au format

text

```
1 Clé[TABULATION]Valeur
```

Exécution

Une fois écrits vos programmes MAP et REDUCE avec votre langage préféré, il suffit alors d'exécuter votre application de la manière suivante :

sh

```
$ hadoop jar hadoop-streaming.jar -input [fichier entree HDFS] \
                                  -output [fichier sortie HDFS] \
                                  -mapper [programme MAP] \
                                  -reducer [programme REDUCE]
```

Et si nous mettions la main à la pâte ? Programmons notre premier job MapReduce avec Hadoop.

Hadoop propose trois modes d'exécution :

- **Mode local (standalone)** : dans ce mode, tout s'exécute au sein d'une seule JVM, en local. C'est le mode recommandé en phase de développement.
- **Mode local pseudo-distribué (pseudo-distribué)** : dans ce mode, le fonctionnement en mode cluster est simulé par le lancement des tâches dans différentes JVM exécutées localement.
- **Mode distribué (fully-distributed)**: c'est le mode d'exécution réel d'Hadoop. Il permet de faire fonctionner le système de fichiers distribué et les tâches sur un ensemble de machines.

Nous allons ici travailler en mode local (standalone ou pseudo-distribué).

Installation

Bien evidemment, la première chose est d'installer Hadoop sur votre machine (à défaut de cluster de machines!). Et pour cela plusieurs solutions s'offrent à nous.

- Une installation manuelle par le biais de paquets adaptés à la distribution ou d'un *tarball* officiel de la fondation Apache.
- Une installation par le biais d'une distribution intégrée d'Hadoop fournies par des entreprises qui vendent du service autour d'Hadoop comme Cloudera, Hortonworks ou encore MapR.

  (Concernant le cloud, la distribution de reférence est Elastic MapReduce (EMR) d'Amazon que nous verrons dans un prochain chapitre.)

L'installation manuelle d'Hadoop (en mode local) est assez simple mais ne se fait pas en un seul clic. Par défaut, le système d'exploitation est Linux. D'autres systèmes d'exploitation peuvent être utilisés mais dans ce cas l'installation est particulière à chaque SE. Il existe de nombreux tutoriaux auxquels se référer dans ce cas.

- Windows : Build and Install Hadoop 2.x or newer on Windows

- Linux : Hadoop: Setting up a Single Node Cluster
- Mac OS : Hadoop in OSX

> Si vous n'êtes pas sous Linux, vous pouvez aussi passer par une machine virtuelle ; pour cela, vous pouvez utiliser un gestionnaire machines virtuelles comme par exemple **VirtualBox**.

Installer manuellement Hadoop est un bon exercice mais si vous voulez faire l'économie de ce temps d'installation, vous pouvez aussi passer directement à une distribution packagée d'Hadoop. Par exemple, vous pouvez très facilement récupérer la machine virtuelle Hadoop de Cloudera pour VirtualBox (4.9 Go). C'est ensuite très simple ; il suffit d'ouvrir VirtualBox et dans le menu `Fichier` de cliquer sur `Import Appliance`, de sélectionner le répertoire non compressé correspondant à la distribution Hadoop, de selectionner le fichier `.ovf` et de cliquer sur les boutons `Open`, `Continue` et `Import`. La machine virtuelle apparaît alors dans la colonne de gauche et vous pouvez la lancer.

Vous pouvez cependant retenir que l'installation d'Hadoop suit toujours les mêmes étapes :

1. L'installation ou la vérification d'un ensemble de pré-requis :
   - La mise à jour de votre système.

```sh
$ sudo apt-get update
```

   - Une version Java récente (au moins 6 ou 7). Vous pouvez vérifier avec la commande suivante.

```sh
$ java -version
```

   Bien évidemment, si vous n'avez pas Java ou si vous avez une mauvaise version, vous savez ce qu'il vous reste à faire !

   - La création d'un groupe et d'un utilisateur spécifique à Hadoop (non obligatoire mais recommandé).

```sh
$ addgroup hadoop
$ adduser --ingroup hadoop hadoopuser
$ adduser hadoopuser
```

   - Configurer ssh pour permettre l'accès vers `localhost` pour l'utilisateur `hadoopuser`

```sh
$ ssh-keygen -t rsa -P ""
$ cat ~/.ssh/id_rsa.pub >> ~/.ssh/authorized_keys
$ ssh localhost
```

2. L'installation d'Hadoop
    - Téléchargement d'Hadoop à partir d'un site mirroir.
    - Décompression et installation de la distribution dans le répertoire de votre choix.
    - Placez-vous dans le répertoire d'installation d'Hadoop :

```sh
$ cd {repertoire distribution hadoop}
```

    - Indiquez l'emplacement de Java dans le fichier `etc/hadoop/hadoop-env.sh` :

```sh
export JAVA_HOME={emplacement de java}
```

    - Testez la commande suivante :

```sh
$ ./bin/hadoop
```

      Si tout s'est bien passé, votre terminal devrait vous afficher la liste des paramètres acceptés par Hadoop :

```sh
$ ./bin/hadoop
Usage: hadoop [--config confdir] [COMMAND | CLASSNAME]
  CLASSNAME            run the class named CLASSNAME
 or
  where COMMAND is one of:
  fs                   run a generic filesystem user client
  version              print the version
  jar <jar>            run a jar file
                       note: please use "yarn jar" to launch
                             YARN applications, not this command.
  checknative [-a|-h]  check native hadoop and compression libraries availability
  distcp <srcurl> <desturl> copy file or directories recursively
  archive -archiveName NAME -p <parent path> <src>* <dest> create a hadoop archive
  classpath            prints the class path needed to get the
  credential           interact with credential providers
                       Hadoop jar and the required libraries
  daemonlog            get/set the log level for each daemon
  trace                view and modify Hadoop tracing settings

Most commands print help when invoked w/o parameters.
```

3. Configuration

   Il faut maintenant définir la configuration de Hadoop et pour cela plusieurs fichiers de configurations doivent être modifiés. Dans Hadoop, les fichiers de configuration fonctionnent sur le principe de clé/valeur : la clé correspondant au nom du paramètre et valeur est celle assignée à ce paramètre, tout cela au format XML.

- Il faut tout d'abord configurer Hadoop en mode nœud unique en éditant le fichier `etc/hadoop/core-site.xml` de la manière suivante.

xml

```xml
<configuration>
    <property>
        <name>fs.defaultFS</name>
        <value>hdfs://localhost:9000</value>
    </property>
</configuration>
```

On spécifie ici le nom du système de fichier. Tous les répertoires et fichiers HDFS seront donc préfixés par `hdfs://localhost:9000` .

- Le fichier `etc/hadoop/hdfs-site.xml` contient les paramètres spécifiques au système de fichiers HDFS. Nous l'éditons de la manière suivante :

xml

```xml
<configuration>
    <property>
        <name>dfs.replication</name>
        <value>1</value>
    </property>
</configuration>
```

Nous précisons ici le nombre de réplication d'un bloc (qui vaut 1 ici).

- Il faut ensuite configurer les paramètres spécifiques à MapReduce qui sont dans le fichier `etc/hadoop/mapred-site.xml` :

xml

```xml
<configuration>
    <property>
        <name>mapreduce.framework.name</name>
        <value>yarn</value>
    </property>
</configuration>
```

Ici nous précisons que nous allons utiliser YARN comme implémentation de MapReduce.

- Enfin nous pouvons aussi paramétrer YARN via le fichier `etc/hadoop/yarn-site.xml` :

xml

```xml
<configuration>
    <property>
        <name>yarn.nodemanager.aux-services</name>
        <value>mapreduce_shuffle</value>
    </property>
</configuration>
```

Nous lui indiquons ici qu'il y aura une opération shuffle.

Hadoop est désormais correctement installé et configuré. Il reste juste à formater le système de fichiers HDFS local et à démarrer Hadoop :

sh

```sh
$ hdfs namenode -format
$ start-dfs.sh
$ start-yarn.sh
```

Et maintenant, WordCount !

Et voilà, nous pouvons vraiment faire tourner le Hello World de MapReduce, notre fameux WordCount !

Opération MAP

Nous allons commencer, à partir de l'API Hadoop de MapReduce, par écrire le code correspondant à l'opération MAP. Nous utilisons ici les types `IntWritable` , `LongWritable` et `Text` de Hadoop.

En java, nous utiliserons la classe `WordCountMapper` suivante :

java

```java
package ooc.cours1.wordcount;

import java.io.IOException;
import java.util.StringTokenizer;
import org.apache.hadoop.io.IntWritable;
import org.apache.hadoop.io.LongWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Mapper;

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

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

    @Override
    public void map(LongWritable key, Text value, Context context) throws IOException, InterruptedException {
        String line = value.toString();

        StringTokenizer tokenizer = new StringTokenizer(line);
        while (tokenizer.hasMoreTokens()) {
            word.set(tokenizer.nextToken());
            context.write(word, one);
        }
    }

    public void run(Context context) throws IOException, InterruptedException {
        setup(context);
        while (context.nextKeyValue()) {
            map(context.getCurrentKey(), context.getCurrentValue(), context);
        }
        cleanup(context);
    }
```

```
33
34 }
```

Opération REDUCE

Puis, nous écrivons la classe `WordCountReducer` qui implémente l'opération REDUCE :

java

```java
package ooc.cours1.wordcount;

import java.io.IOException;
import java.util.Iterator;

import org.apache.hadoop.io.IntWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Reducer;

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

    private IntWritable totalWordCount = new IntWritable();

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

        int sum = 0;
        Iterator<IntWritable> iterator = values.iterator();

        while (iterator.hasNext()) {
            sum += iterator.next().get();
        }

        totalWordCount.set(sum);
        // context.write(key, new IntWritable(sum));
        context.write(key, totalWordCount);
    }
}
```

Et nous pouvons ensuite écrire le code correspondant au `WordCountDriver` comme ci-dessous :

java

```java
package ooc.cours1.wordcount;

import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.conf.Configured;
import org.apache.hadoop.fs.FileSystem;
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.io.IntWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Job;
import org.apache.hadoop.mapreduce.lib.input.FileInputFormat;
import org.apache.hadoop.mapreduce.lib.input.TextInputFormat;
import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;
```

```java
import org.apache.hadoop.mapreduce.lib.output.TextOutputFormat;
import org.apache.hadoop.util.GenericOptionsParser;
import org.apache.hadoop.util.Tool;
import org.apache.hadoop.util.ToolRunner;

public class WordCountDriver extends Configured implements Tool {
    public int run(String[] args) throws Exception {
        if (args.length != 2) {
            System.out.println("Usage: [input] [output]");
            System.exit(-1);
        }
        // Creation d'un job en lui fournissant la configuration et une description textuelle de la tache
        Job job = Job.getInstance(getConf());
        job.setJobName("wordcount");

        // On precise les classes MyProgram, Map et Reduce
        job.setJarByClass(WordCountDriver.class);
        job.setMapperClass(WordCountMapper.class);
        job.setReducerClass(WordCountReducer.class);

        // Definition des types clé/valeur de notre problème
        job.setOutputKeyClass(Text.class);
        job.setOutputValueClass(IntWritable.class);

        job.setInputFormatClass(TextInputFormat.class);
        job.setOutputFormatClass(TextOutputFormat.class);

        Path inputFilePath = new Path(args[0]);
        Path outputFilePath = new Path(args[1]);

        // On accepte une entree recursive
        FileInputFormat.setInputDirRecursive(job, true);

        FileInputFormat.addInputPath(job, inputFilePath);
        FileOutputFormat.setOutputPath(job, outputFilePath);

        FileSystem fs = FileSystem.newInstance(getConf());

        if (fs.exists(outputFilePath)) {
            fs.delete(outputFilePath, true);
        }

        return job.waitForCompletion(true) ? 0: 1;
    }

    public static void main(String[] args) throws Exception {
        WordCountDriver wordcountDriver = new WordCountDriver();
        int res = ToolRunner.run(wordcountDriver, args);
        System.exit(res);
    }
}
```

N'oubliez pas de compiler votre programme ! Nous pouvons le faire de cette manière :

```sh
```

```
$ export HADOOP_CLASSPATH=$($HADOOP_HOME/bin/hadoop classpath)
$ javac -classpath $HADOOP_CLASSPATH WordCount*.java
```

Après cette compilation (sans erreur), il faut compresser ce programme au sein d'un `.jar` :

sh

```
$ mkdir -p ooc/cours1/wordcount
$ mv *.class ooc/cours1/wordcount
$ jar -cvf ooc_cours1_wordcount.jar -C . ooc
```

Et nous pouvons exécuter notre programme MapReduce par le client console hadoop.

sh

```
$ hadoop jar ooc_cours1_wordcount.jar ooc.cours1.wordcount.WordCountDriver /input/lejourseleve.txt /results
```

En Python avec Hadoop Streaming

Pour finir, juste pour le plaisir d'écrire un petit peu de code en python, voici comment nous pouvons implémenter WordCount en python avec Hadoop streaming :

`WordCountMapper.py` :

python

```
#! /usr/bin/env python3
import sys

for line in sys.stdin:
    # Supprimer les espaces
    line = line.strip()
    # recupérer les mots
    words = line.split()

    # operation map, pour chaque mot, generer la paire (mot, 1)
    for word in words:
        print("%s\t%d" % (word, 1))
```

`WordCountReducer.py` :

python

```
#! /usr/bin/env python3

import sys
total = 0
lastword = None

for line in sys.stdin:
    line = line.strip()

    # recuperer la cle et la valeur et conversion de la valeur en int
    word, count = line.split()
    count = int(count)
```

```
13
14      # passage au mot suivant (plusieurs cles possibles pour une même exécution de programme)
15      if lastword is None:
16          lastword = word
17      if word == lastword:
18          total += count
19      else:
20          print("%s\t%d occurences" % (lastword, total))
21          total = count
22          lastword = word
23
24  if lastword is not None:
25      print("%s\t%d occurences" % (lastword, total))
```

Vous pouvez ensuite exécuter ce WordCount en python de la manière suivante :

sh

```
1  $ hadoop jar hadoop-streaming.jar -input /lejourseleve.txt -output /results -mapper ./WordCountMapper.py -
   reducer ./WordCountReducer.py
```

Que pensez-vous de ce cours ?

PARCOUREZ LES PRINCIPAUX ALGORITHMES MAPREDUCE

ALLEZ AU-DELÀ DE MAPREDUCE AVEC SPARK

Les professeurs

Céline Hudelot

Professeur des Universités en Informatique à CentraleSupélec.

Régis Behmo

Expert en machine learning, développeur fullstack, grimpeur invétéré et gros, très gros amateur de nouilles chinoises.

CONTACT

EN PLUS

Français

Télécharger dans l'App Store