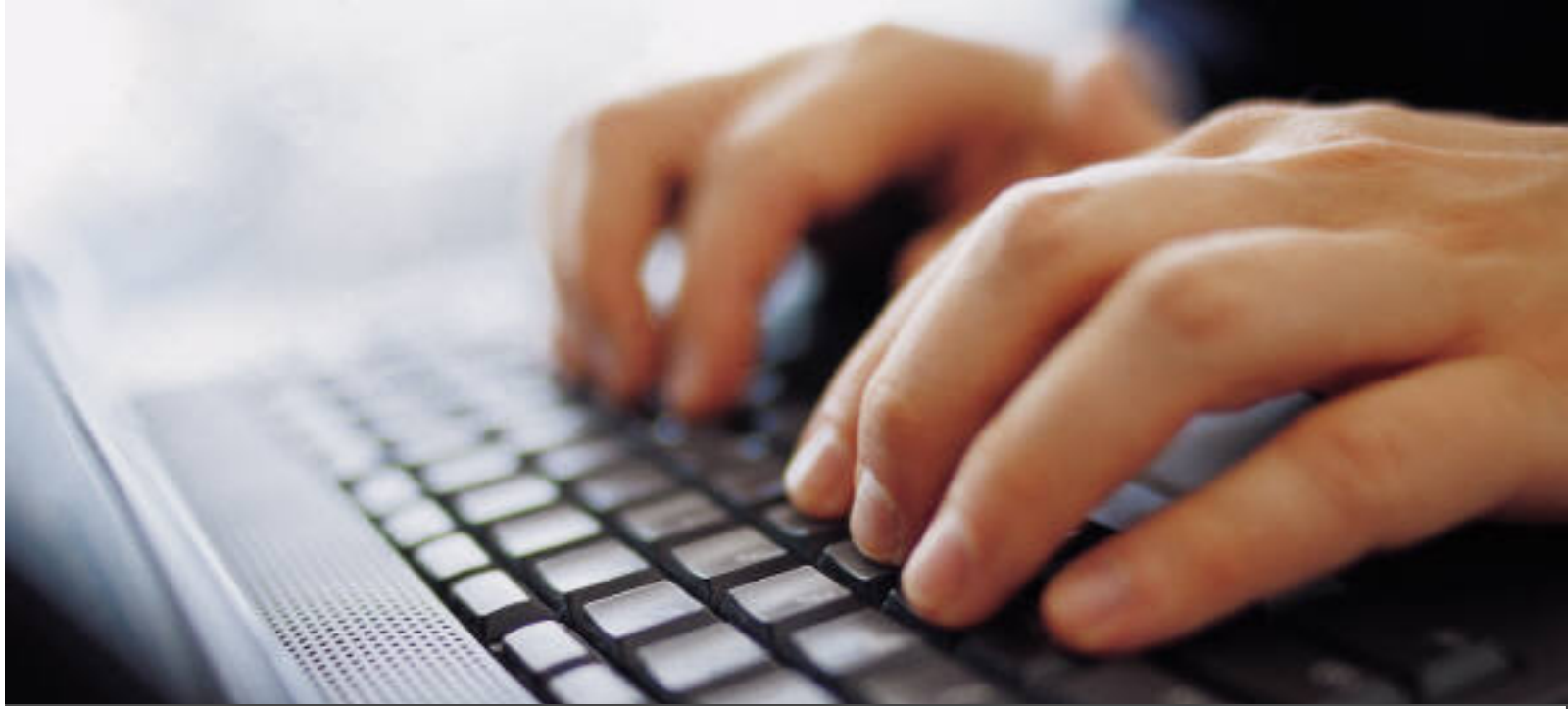
Hadoop / Big Data

Benjamin Renaut <renaut.benjamin@tokidev.fr>

MBDS

2019 - 2020



12

Apache Spark – Présentation et architecture

Présentation

- **Framework de conception et d'exécution map/reduce.**
- **Originellement (2014) un projet de l'université de Berkeley en Californie, désormais un logiciel libre de la fondation Apache.**
- **Licence Apache.**
- **Trois modes d'exécution:**
  - **Cluster Spark natif.**
  - **Hadoop (YARN).**
  - **Mesos (Spark natif + scheduler Mesos).**
- **Sensiblement plus rapide que Hadoop, notamment pour les tâches impliquant de multiples maps et/ou reduce.**

Présentation

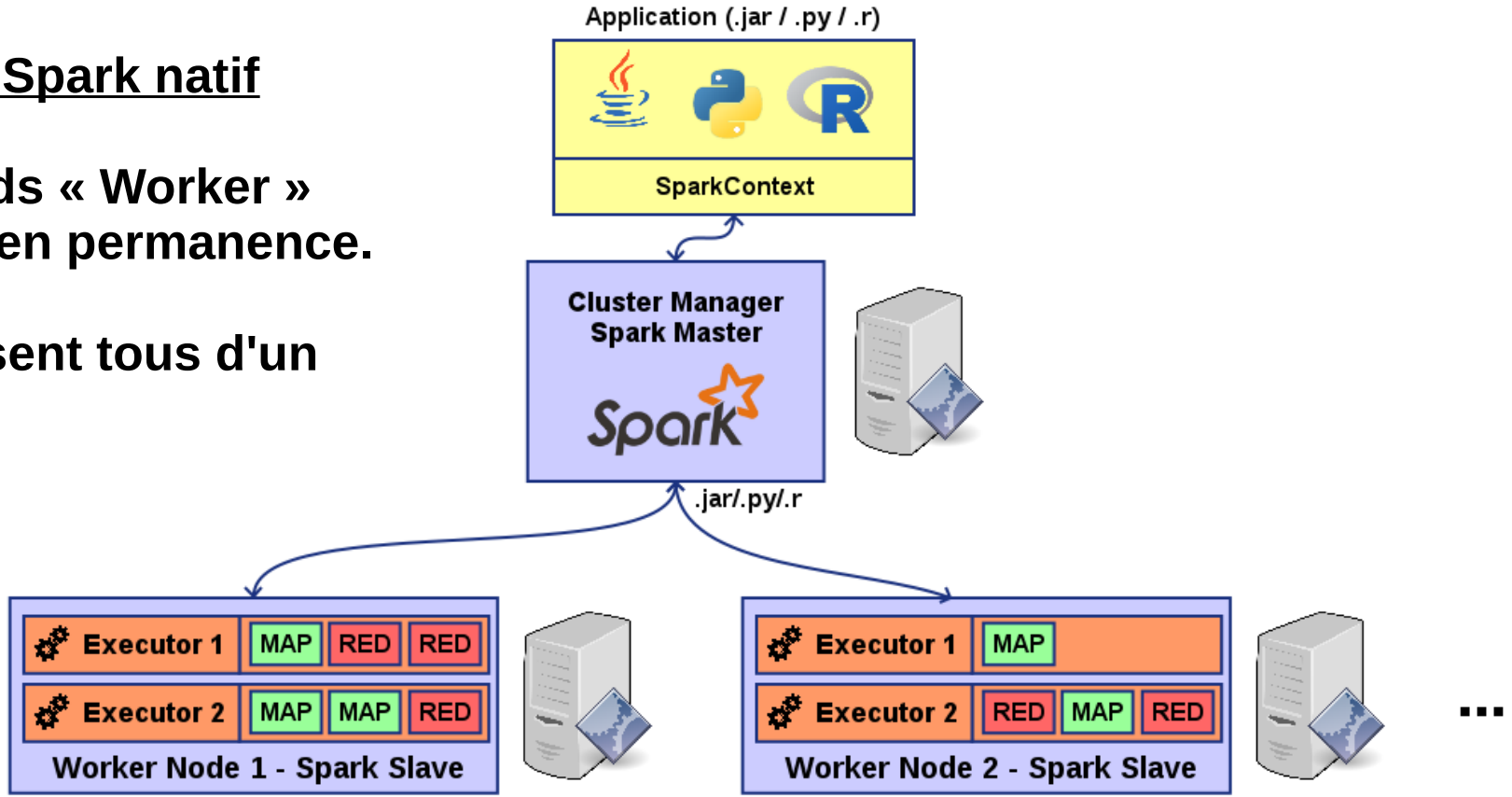
- **Développé en Scala (langage orienté objet dérivé de Java et incluant de nombreux aspects des langages fonctionnels). Quatre langages supportés:**
  - **Scala**
  - **Java**
  - **Python (PySpark)**
  - **R (SparkR)**
- **Performances et fonctionnalités équivalentes pour les 4.**

Architecture – Haut niveau

En mode Spark natif

Les nœuds « Worker » tournent en permanence.

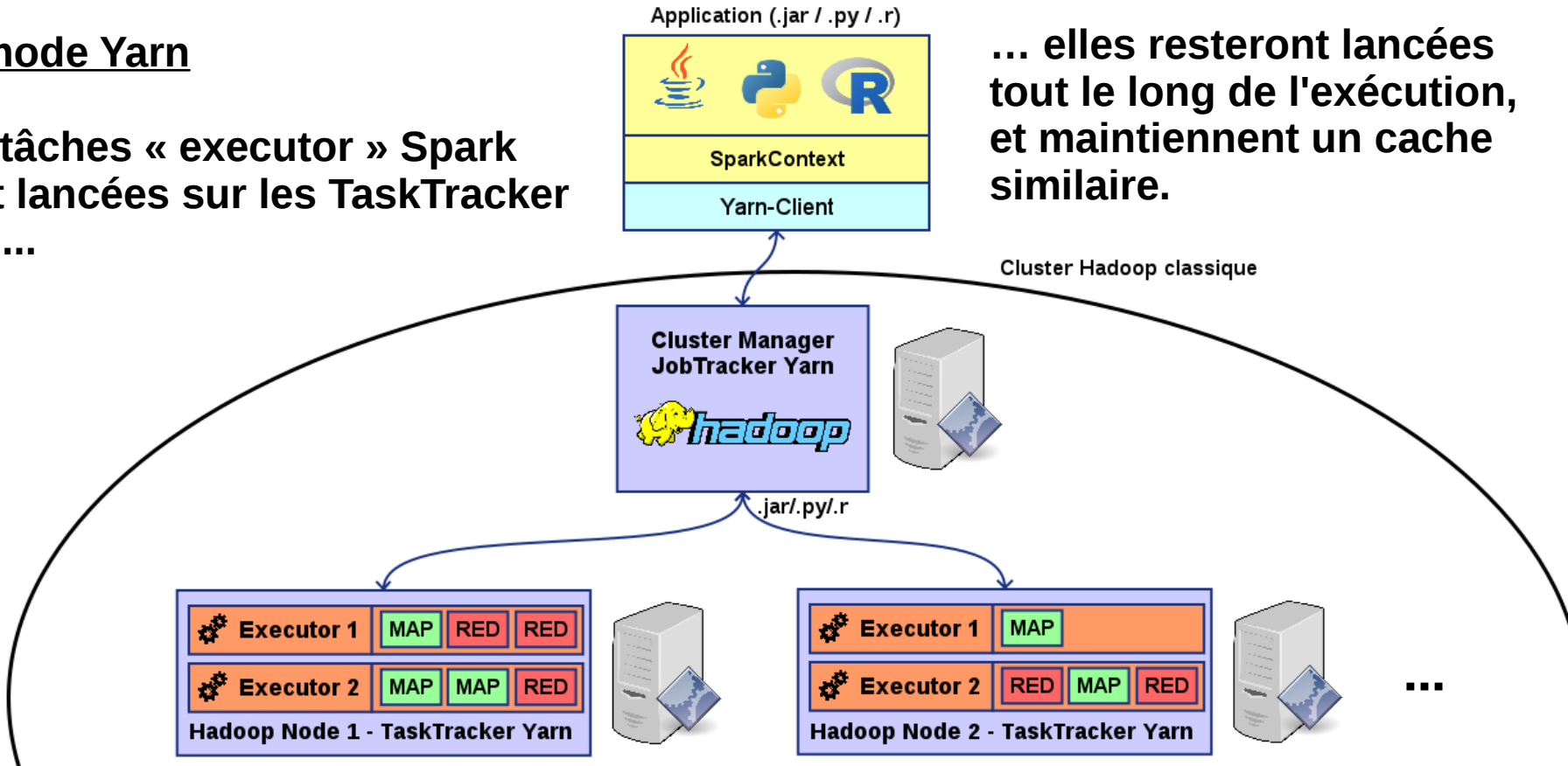
Ils disposent tous d'un cache.



Architecture – Haut niveau

En mode Yarn

Des tâches « executor » Spark Sont lancées sur les TaskTracker Yarn...

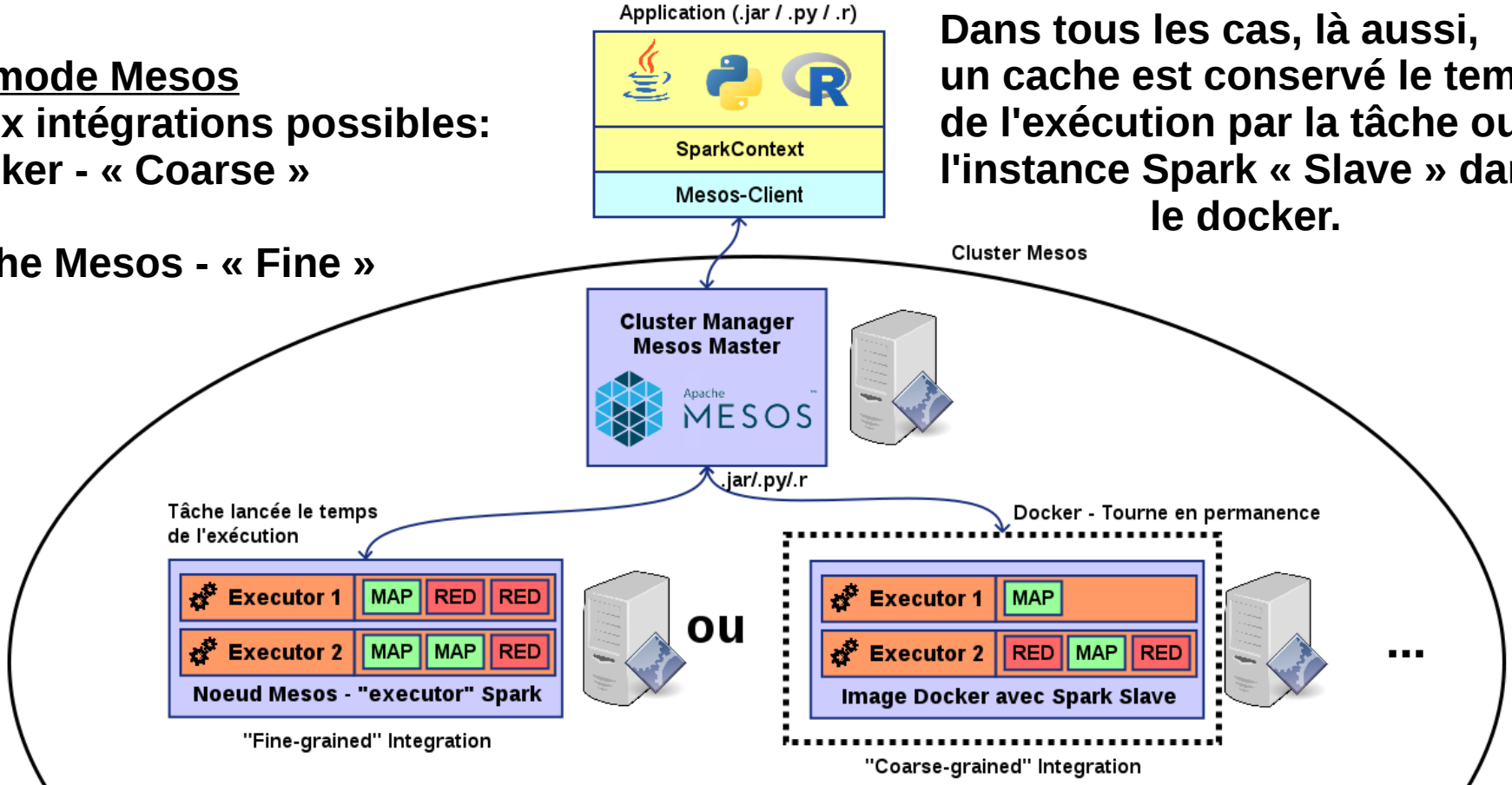


… elles resteront lancées tout le long de l'exécution, et maintiennent un cache similaire.

Architecture – Haut niveau

<u>En mode Mesos</u>
Deux intégrations possibles:
Docker - « Coarse »
Ou
Tâche Mesos - « Fine »

Dans tous les cas, là aussi, un cache est conservé le temps de l'exécution par la tâche ou l'instance Spark « Slave » dans le docker.



Remarques

- **Le mode de déploiement natif, bien qu'en théorie aussi performant que les autres, nécessite de mettre en place un système de *high availability* pour assurer la redondance du processus « Master ».**

- **Par ailleurs, Spark est souvent amené en remplacement ou en complément d'instances de Hadoop / dans des infrastructures existantes; dans les faits, le mode de déploiement *via* Yarn est à l'heure actuelle le plus courant en production.**

- **Le cache maintenu au sein des instances d'« executors » Spark fait partie des innovations apportées par Spark par rapport à Hadoop; cet aspect va être abordé plus en détail.**

Architecture – Exécution

- **Le modèle proposé par Spark a plusieurs particularités:**
  - **Il n'impose pas un modèle « map / shuffle / reduce » aussi rigoureux que Hadoop; à la place, il propose de multiples actions et transformations applicables aux données.**
  - **Dans les faits, un programme Spark sera souvent susceptible d'appeler plusieurs opérations « map », « reduce », ou « shuffle », l'une à la suite de l'autre ou en parallèle; et c'est là que Spark montre vraiment ses avantages en terme de performances.**
  - **Dans le modèle Hadoop, de telles tâches seraient réalisées *via* plusieurs exécutions de « tâches » Hadoop, impliquant une sérialisation HDFS (ou autre) après chacune de ces tâches.**

Architecture – Exécution

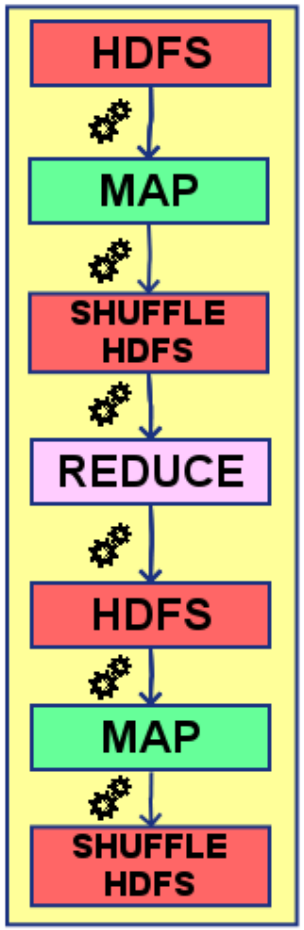
- **Là où Hadoop aurait besoin de séraliser / désérialiser les données entre chaque « tâche » map/reduce effectuée, Spark sait les conserver en mémoire tout au long de l'exécution du programme; il n'utilise le disque que quand c'est nécessaire. C'est pour cette raison principale que Spark est beaucoup plus rapide que Hadoop pour les tâches complexes, impliquant de multiples traitements à la suite.**

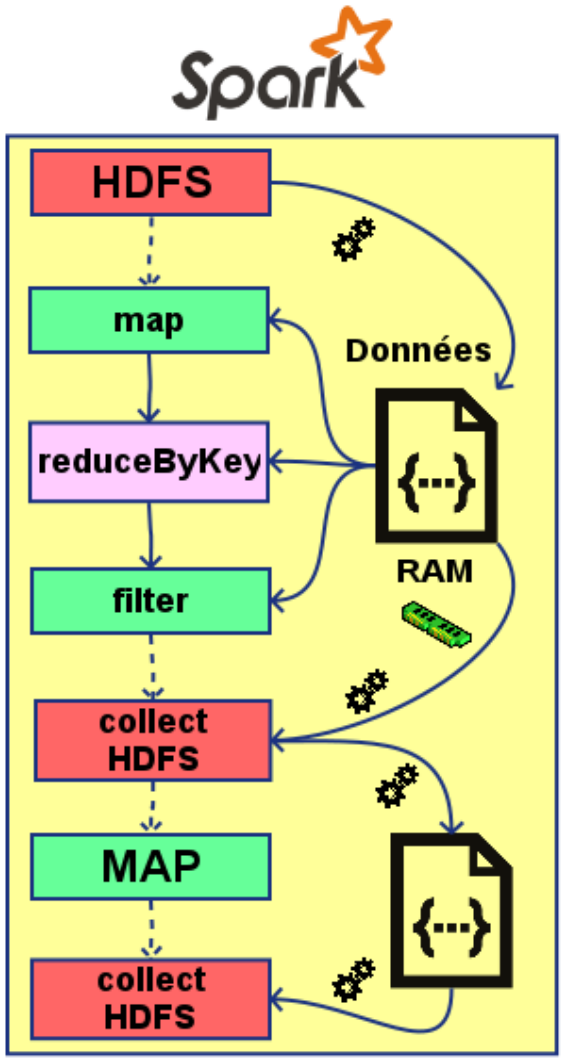
- **Même pour une simple tâche map/reduce unique, cependant, Spark aura également tendance a être plus performant, parce qu'il maintient dans la plupart de ses modes de déploiement des « executors » tournant en permanence et prêts à l'usage, là où Hadoop devra invoquer le lancement d'une nouvelle machine virtuelle Java, effectuer du *parsing* de fichiers XML, etc. au lancement d'une tâche.**

Architecture

- Spark est capable de déterminer *quand* il aura besoin de sérialiser les données / les ré-organiser; et ne le fait que quand c'est nécessaire.

- On peut également explicitement lui demander de conserver des données en ram, parce qu'on sait qu'elles seront nécessaires entre plusieurs instances d'écriture disque.





Architecture – Les RDDs

- **Au centre du paradigme employé par Spark, on trouve la notion de RDD, pour Resilient Distributed Datasets.**

- **Il s'agit de larges *hashmaps* stockées en mémoire et sur lesquelles on peut appliquer des traitements.**

- **Ils sont:**

  - **Distribués.**
  - **Partitionnés (pour permettre à plusieurs nœuds de traiter les données).**
  - **Redondés (limite le risque de perte de données).**
  - **En lecture seule; un traitement appliqué à un RDD donne lieu à la création d'un nouveau RDD.**

Architecture – Les RDDs

- **Deux types d'opérations possibles sur les RDDs:**
  - **Une transformation: une opération qui modifie les données d'un RDD. Elle donne lieu à la création d'un nouveau RDD. Les transformations fonctionnent en mode d'évaluation *lazy*: elles ne sont exécutées que quand on a véritablement besoin d'accéder aux données. « `map` » est un exemple de transformation.**
  - **Une action: elles accèdent aux données d'un RDD, et nécessitent donc son évaluation (toutes les transformations ayant donné lieu à la création de ce RDD sont exécutées l'une aprés l'autre). « `saveAsTextFile` » (qui permet de sauver le contenu d'un RDD) ou « `count` » (qui renvoie le nombre d'éléments dans un RDD) sont des exemples d'actions.**

Architecture – Les RDDs

- **Lors d'un programme Spark, les RDD contiendront les données sur lesquelles on travaille de manière parallèle; et ils ne seront sérialisés sur disque que quand cela sera nécessaire.**

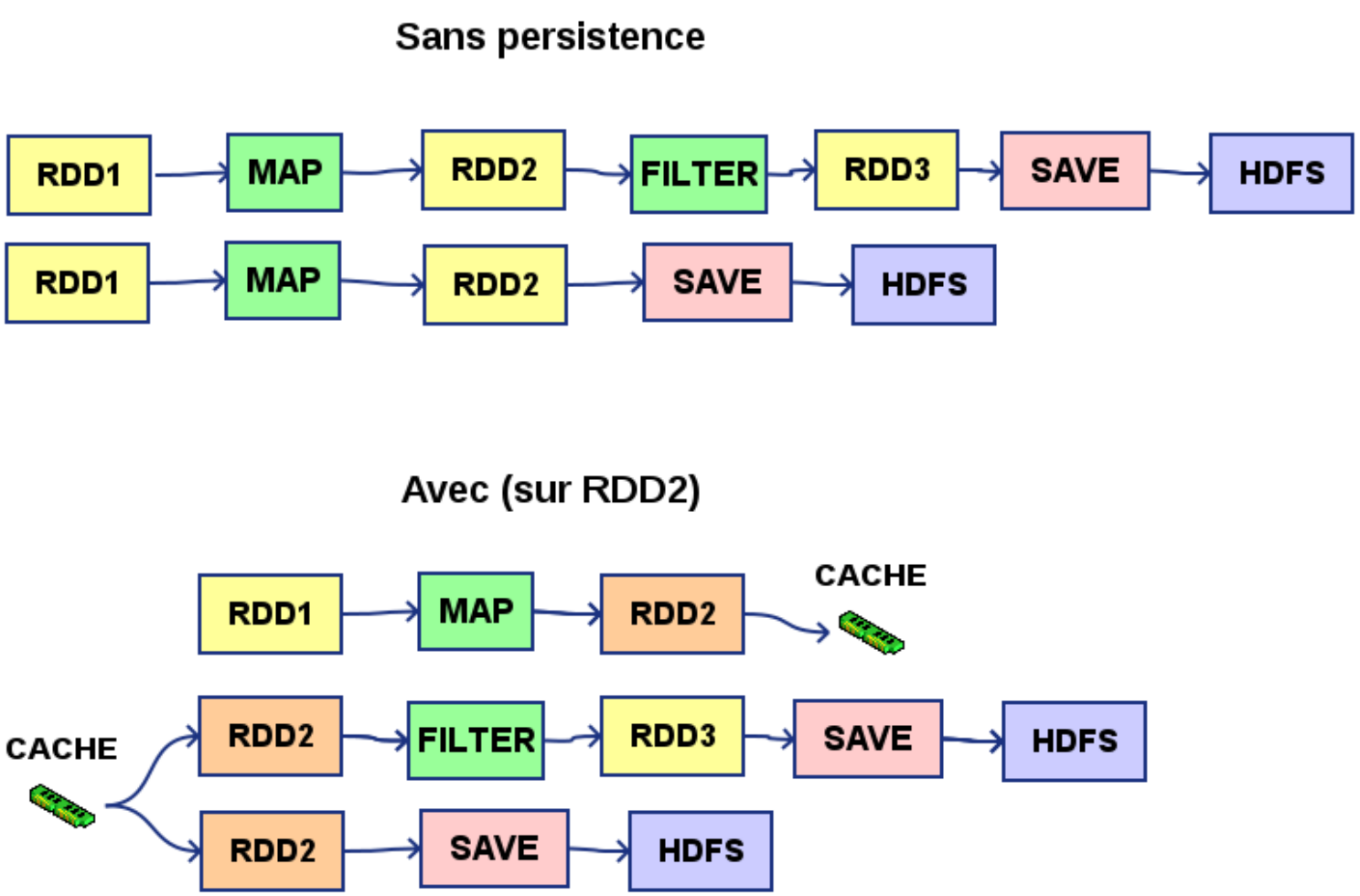
- **En revanche, une transformation donne toujours lieu à la création d'un nouveau RDD; et les transformations sont évaluées en mode « lazy ».**

- **Cela veut dire que deux tâches différentes appliquant une opération à un même RDD lui-même issu d'une transformation donnera lieu à l'exécution de cette transformation *deux* fois: une fois pour la première tâche, et une fois pour la seconde.**

- **Pour éviter cela, on peut *persister* un RDD en mémoire, de telle sorte qu'il ne soit calculé qu'une fois et que Spark le maintienne en cache pendant toute la durée du programme.**

RDDs

Sans persistence, le map entre RDD1 et RDD2 est exécuté deux fois.



Architecture – Les RDDs

- **Ces RDDs persistés sont justement stockés dans le cache des nœuds executor montré précédemment. C'est la principale source des améliorations en terme de performance apportées par Spark:**

  - **Les RDDs; et le paradigme map/reduce plus souple couplé à l'empilement des traitements effectués sur les RDDs sans sérialisation entre chaque opération (il n'écrit « que quand c'est nécessaire », étant conscient du programme dans son ensemble).**

  - **La possibilité de persister certains de ces RDDs de telle sorte qu'ils ne soit pas recalculés plusieurs fois si nécessaire.**

… l'une comme l'autre sont surtout visibles sur des programmes complexes (multiples opérations sur des RDDs).

Architecture – Exécution

- **Lors de l'exécution d'un programme, Spark construit un Graphe Orienté Acyclique (DAG, Directed Acyclic Graph) représentant les *actions* et *opérations* effectuées dans le programme.**

- **Il découpe ensuite le graphe en étapes (stages), chacun contenant un certain nombre de tâches: des actions ou des opérations.**

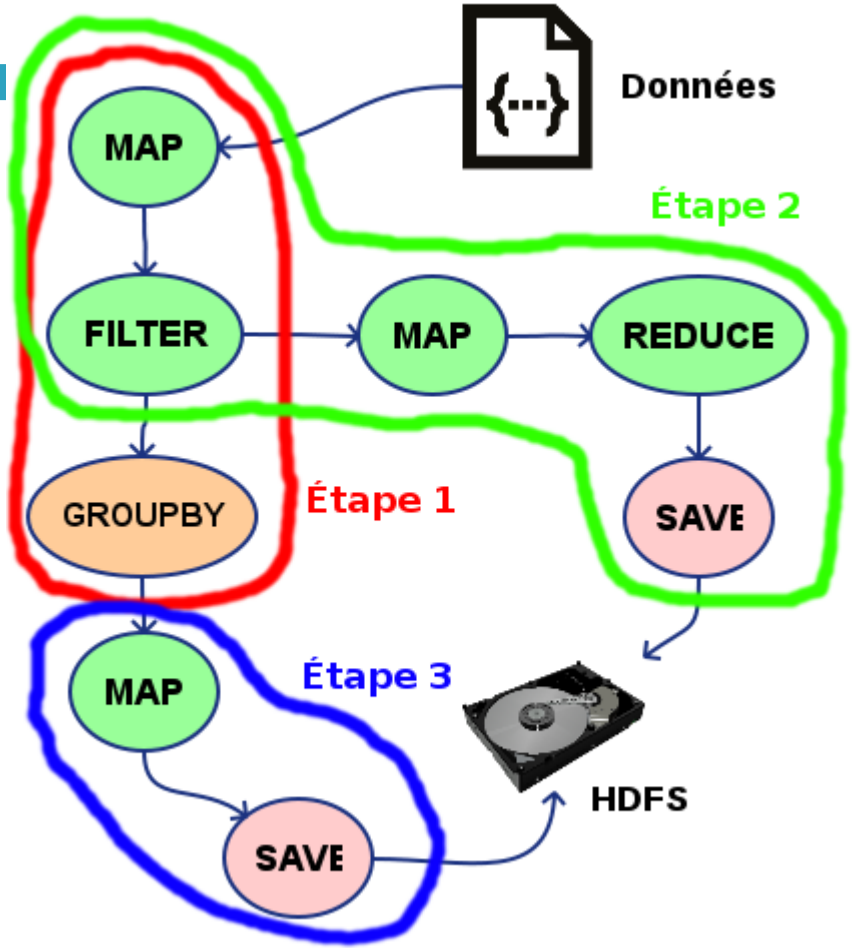
- **Le découpage en étapes s'effectue avant tout en se basant sur le besoin de ré-organiser les données des RDDs manipulés (les partitionner différemment pour que l'exécution parallèle reste efficace).**

- **Le *driver* envoie ensuite les tâches, l'une après l'autre, au *cluster manager*; celui-ci n'est pas conscient des étapes ou du programme plus général: il ne reçoit que des tâches à exécuter.**

Exécution - Exemple

- Ici, l'exécution d'un programme est découpée dans le graphe en trois étapes. On n'a persisté aucun RDD, et on remarque qu'en conséquence les opérations « map » et « filter » initiales seront exécutées deux fois.

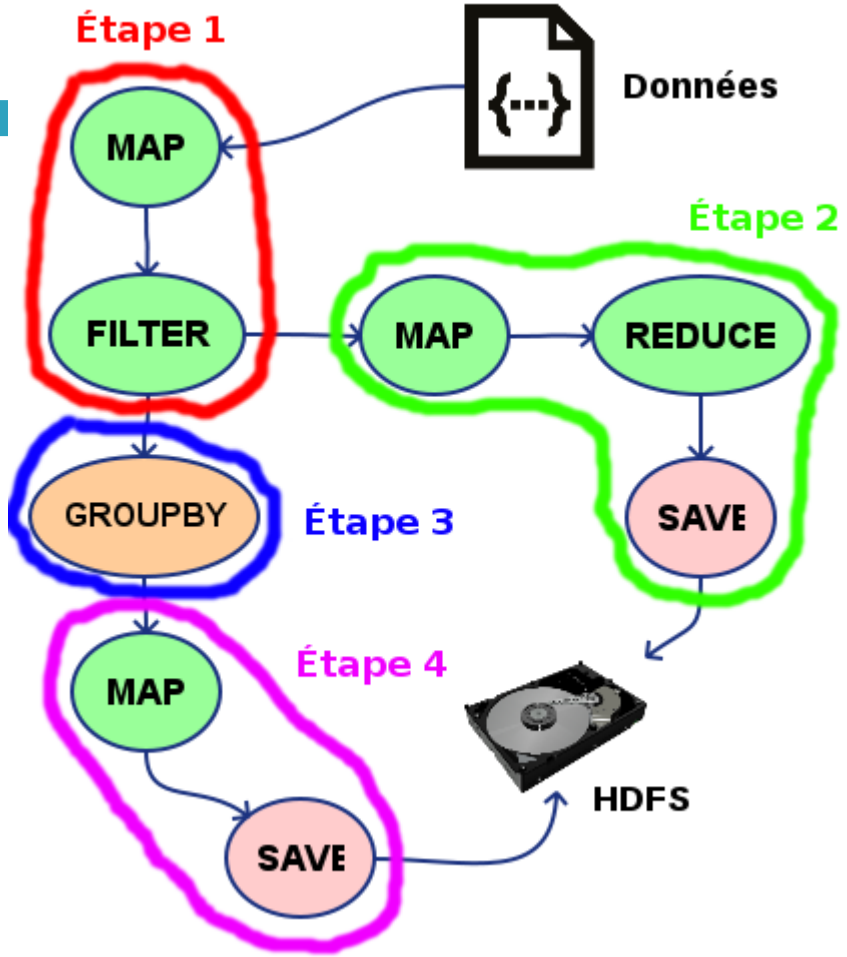
- `groupBy` est une opération Spark qui ré-organise les données entre les nœuds; elle nécessite donc un découpage en tâches. On parle dans ce cas d'opération « large » (« wide operation »).



Exécution - Exemple

- Ici, on imagine qu'on a persisté le RDD en sortie de « filter »; le découpage est par conséquent différent.

- A noter que les opérations « larges » (comme `groupBy`) sont plus coûteuses, parce qu'elles nécessitent de ré-organiser (re-partitioner) les données entre les nœuds d'exécution, ce qui peut impliquer un certain nombre d'échanges.



Architecture – Exécution

- **Il faudra ainsi faire attention:**
  - **Au partitionnement des données dans les RDDs (celui-ci doit être optimal par rapport au nombre de nœuds du *cluster*).**
  - **A la persistance (persister le bon RDD peut radicalement changer les performances).**
  - **A l'emploi ou non d'opérations « larges » (qui « cassent » le partitionnement et sont plus coûteuses).**

- **Là aussi, on remarque que Spark verra surtout des avantages lors de son usage sur des programmes complexes, effectuant des traitements sur les données dépassant le simple cadre figé du « map / shuffle / reduce » introduit par Hadoop. Dans la pratique, cependant, beaucoup de problèmes ont une complexité qui est mieux supportée par le paradigme de Spark; et qui nécessiterait plusieurs tâches Hadoop.**

Architecture – Exécution

- **Un programme Spark fait appel à un objet, le SparkContext, qui permet d'utiliser l'API Spark et de communiquer avec un cluster Spark (ou, au choix, de tourner en mode « standalone », avec un ou plusieurs threads).**

- **On parle de *driver* pour désigner l'ensemble composé du programme, du SparkContext, et de plusieurs autres composants, parmi lesquels:**

  - **Le DagScheduler, qui est en charge de construire les graphes acycliques à partir du code du programme et de les découper en tâche.**
  - **Le TaskScheduler, qui lance l'exécution des tâches, surveille qu'elles s'exécutent bien, et relancent celles qui ont posé problème au besoin.**

Architecture – Exécution

- **Ce *driver* tourne au sein du programme lancé, sur la machine d'exécution, ou sur un des nœuds d'exécution du *cluster*. Il s'agit en fait du *main* de l'application, s'exécutant et créant un SparkContext pour communiquer avec le *cluster manager*. Faire tourner le *driver* localement n'est pas recommandé: il est préférable de le soumettre au *cluster*.**

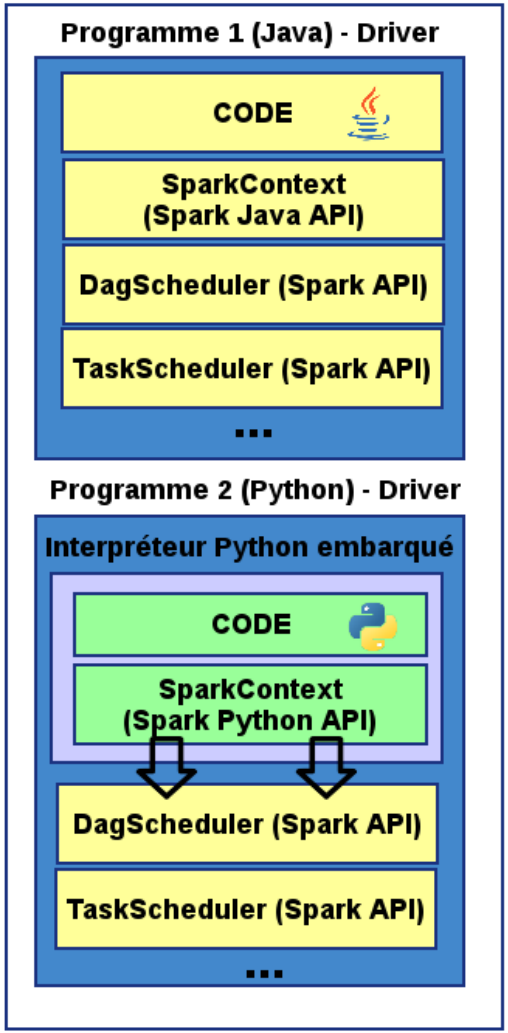
- **La plupart des opérations Spark prennent en paramètre une fonction à appliquer aux données; cette fonction est indépendante de l'environnement et ne référence aucune variable externe (elle se contente de travailler sur ses arguments); on parle de *closure*.**

- **Lors de l'exécution du programme, le TaskScheduler du *driver* enverra en fait aux nœuds d'exécution ces fonctions (*closures*), avec une référence aux données sur lesquelles elles doivent s'appliquer. Chacun des nœuds n'effectue donc que des tâches simples; c'est le *driver* qui orchestre l'exécution.**

Architecture – Exécution

- **Dans les faits, un programme Spark prêt à l'exécution est un fichier .jar (Scala/Java), .py (Python) ou .r (R).**

- **Dans le cas de Python et R, leur impact non-natif sur les performances est minimal car le programme lui-même, et toutes les *closures* qu'il émet sur le *cluster*, sont bien exécutés en Python/R natif: le programme se contente d'utiliser un SparkContext via une librairie adaptée, ce qui lui permet de communiquer avec le *cluster manager* pendant son exécution.**



Avantages

- **Performances supérieures à celles de Hadoop pour une large quantité de problèmes; et presque universellement au moins équivalentes pour le reste.**

- **API simple et bien documentée; très simple à utiliser. Paradigme plus souple qui permet un développement conceptuellement plus simple.**

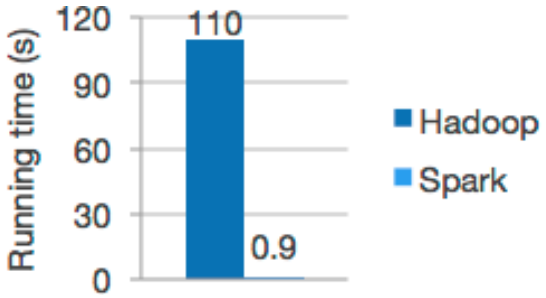
- **Très intégrable avec d'autres solutions; peut très facilement lire des données depuis de nombreuses sources, et propose des couches d'interconnexion très faciles à utiliser pour le reste (API dédiée Spark Streaming, Spark SQL).**

- **APIs dédiées pour le traitement de problèmes en *machine learning* (Spark MLlib) et graphes (Spark GraphX).**

Performances

- **Fortement dépendantes du problème mais d'une manière générale supérieures à Hadoop. Dans le cas de problèmes complexes, effectuant de nombreuses opérations sur les données et notamment sur les *mêmes* données antérieures, fortement supérieures (jusqu'à 100x plus rapide).**

- **Même sans travailler plusieurs fois sur les mêmes données ou sans persistance explicite particulière,
le paradigme de programme parallélisé en graphe acyclique couplé aux RDDs et leurs propriétés permet des améliorations notables (~10x) pour de nombreux problèmes dépassant le cadre rigide du simple map/shuffle/reduce lancé une fois.**



Inconvénients

- **Spark consomme beaucoup plus de mémoire vive que Hadoop, puisqu'il est susceptible de garder une multitude de RDDs en mémoire. Les serveurs nécessitent ainsi plus de RAM.**

- **Il est moins mature que Hadoop.**

- **Son *cluster manager* (« Spark Master ») est encore assez immature et laisse à désirer en terme de déploiement / haute disponibilité / fonctionnalités additionnelles du même type; dans les faits, il est souvent déployé *via* Yarn, et souvent sur un cluster Hadoop existant.**

Usage

- **Sur une machine où Spark est installé, on peut accéder à un *shell* interactif *via* les commandes:**

```
spark-shell
pyspark
sparkR
```

… pour respectivement lancer le shell interactif Scala, Python, ou R.

- **Ce *shell* permet de rapidement lancer et tester de petits programmes Spark (comme le *shell* Pig ou Python traditionnel).**

- **Par défaut, il fonctionne en mode local: il agit comme driver et simule la présence d'un *cluster*.**

Usage

- **On peut également soumettre un programme entier à un cluster par le biais de la commande:**

```
spark-submit
```

- **A la différence du *shell*, l'exécution n'est alors pas interactive: chaque ligne du programme est exécutée séquentiellement.**

- **Par défaut, là aussi, la commande fonctionne en mode local: il agit comme driver et simule la présence d'un *cluster*.**

- **Le code du programme lui-même est cependant susceptible de se connecter ailleurs; et on peut également outrepasser cette directive via des arguments.**

Usage

- **Pour les deux commandes, deux arguments importants existent:**

  - **`--master URL`**
    Permet de spécifier l'URL du *cluster manager*. Celle-ci peut être au modèle « spark://... » (pour spécifier un *cluster manager* Spark natif), « yarn » (pour utiliser le *cluster* hadoop local défini *via* des variables d'environnement), « mesos://... » (pour spécifier un *cluster manager* Mesos), « local » (pour indiquer le mode local), ou « local[N] » (idem mais avec N *workers* simulés). Par exemple:

```
spark-submit --master spark://10.0.0.1/
spark-submit --master yarn
spark-submit --master local[2]
spark-submit --master mesos://10.0.0.42:5050/
```

Usage

- `--deploy-mode client|cluster`
  Permet de spécifier si le code du *driver* (et ses fonctionnalités associées: DagScheduler, etc.) doit tourner sur la machine courante, là où est lancée la commande (« client »), ou bien si elle doit être lancée dans un « executor » Spark dédié (« cluster »).

  Le mode « client » est déconseillé sur une machine située loin des *executors* (en terme de latence réseau), parce qu'en tant que TaskScheduler, il devra échanger avec ces *workers* souvent. De même, utiliser le mode « client » veut dire qu'il faudra attendre la fin de la tâche sur la machine de lancement, là où le mode « cluster » permet de lancer une tâche sans se soucier de sa session sur la machine de lancement.

Usage

- **Pour soumettre un programme Spark Java ou Scala, on utilisera également les deux arguments suivants à spark-submit:**

  - **`--class CLASSPATH`**
    Le path complet de la classe main du programme.

  - **`JARFILE`**
    Le fichier jar à exécuter, par exemple via une URL de type « file:// » ou « hdfs:// ». Le fichier doit être disponible sur toutes les machines *workers* à partir de l'URL fournie.

    Par exemple:

```
spark-submit --master yarn --class org.mbds.spark.wcount \
             --deploy-mode cluster WordCount.jar
```

Usage

- **Et pour soumettre un programme Spark Python, on utilisera les deux arguments suivants à spark-submit:**

  - **`--py-files CLASSPATH`**
    (optionnel) Le chemin d'un .py ou d'un .zip contenant des dépendances Python à mettre à disposition du script sur toutes les machines.

  - **`PYFILE`**
    Le fichier .py à exécuter, par exemple via une URL de type « file:// » ou « hdfs:// ». Le fichier doit être disponible sur toutes les machines *workers* à partir de l'URL fournie.
    Par exemple:

```
spark-submit --master yarn --py-files libs.zip \
             --deploy-mode cluster wordcount.py
```

Usage

- **Il y a évidemment de nombreux autres arguments aux deux outils; mais ceux-ci sont les principaux.**

- **Pour plus d'informations sur leur usage, se référer à la documentation:**

  https://spark.apache.org/docs/latest/

13

Spark – API et développement

Présentation

- **L'API Spark est globalement unifiée: les fonctions et leurs synopsis sont globalement identiques d'un langage à l'autre.**
- **Le plus important est donc de connaître les différentes opérations.**
- **L'API présentée ici est celle de Python; mais les APIs Scala, Java et (dans une moindre mesure) R sont similaires.**
- **Des exemples Java et Scala seront également présentés.**
- **Comme indiqué précédemment, la plupart des opérations prennent en paramètre une fonction indépendante de l'environnement (*closure*); dans les faits, cela se traduira souvent dans le code par l'usage de fonctions lambda.**

Rappel – Les fonctions lambda

- **On peut définir en Python une fonction Lambda (closure) par le biais de la syntaxe:**

```
lambda ARG1, ARG2, ARG3, ...: RETURN_VALUE
```

Par exemple:

```
lambda x: x+1
lambda a, b: a+b
lambda mot: (mot, 1)
```

Rappel – Les fonctions lambda

- **En Java, depuis Java 8, il est également possible de faire:**

```
(PARAMS) -> expression;
```

ou

```
(PARAMS) -> { expressions };
```

Par exemple:

```
(x) -> return(x+1)
(a, b) -> { return(a+b)}
(mot) → { System.out.println(mot); return(Arrays.asList(mot,
                                                        1)) }
```

API Python

- **La classe principale permettant d'accéder à l'API Spark est SparkContext.**

- **Il est contenu dans le module pyspark. On commencera ainsi généralement un programme Spark par:**

```
from pyspark import SparkContext
```

(ou similaire)

- **C'est via cette classe qu'on instanciera un contexte de connexion à l'API Spark pour exécuter toutes les opérations. A noter que si on utilise le shell interactif « pyspark », cet import est déjà fait; et on dispose déjà une instance 'sc'.**

API Python

- **Pour instancier un contexte, on dispose de nombreux arguments et options possibles. Une forme basique:**

```
SparkContext(master=MASTER_URL, appName=DESCRIPTION)
```

L'option `master` prend le même format que l'URL de l'argument `--master` des outils Spark; et le `appName` doit contenir une description textuelle de l'application. Par exemple:

```
sc=SparkContext(master="yarn", appName="Wordcount")
sc=SparkContext("spark://10.0.0.1/")
sc=SparkContext("local[2]", "WordCount")
```

API Python – Lecture des données

- **On dispose de plusieurs fonctions pour lire des données et les charger au sein d'un premier RDD. Ces fonctions renvoient toutes un (pointeur vers un) RDD. Une liste non exhaustive (à appeler sur le SparkContext):**

  - **`textFile(FILENAME)`**
    Permet de charger un fichier texte; celui-ci sera charge comme une liste dont chacun des éléments est une ligne du texte. Dans cette fonction et toutes les autres, le nom de fichier doit être une URL Hadoop (« `hdfs://` », « `file://` ») et doit être disponible sur toutes les machines d'exécution.
    Comme il s'agit d'une URL Hadoop, on peut également spécifier ici un répertoire pour charger tous les fichiers texte qu'il contient (comme s'il s'agissait d'un fichier unique).

API Python – Lecture des données

- **`binaryFiles(FILENAME)`**
  Permet de charger un ou plusieurs fichiers binaires; même format d'URL.

- **`sequenceFile(FILENAME)`**
  Permet de charger un ou plusieurs fichiers *sequence* (les mêmes que ceux utilisés dans Hadoop, Sqoop, ou encore Pig); une conversion de type aura lieu vers le Python si elle est possible (certains arguments permettent de l'orienter). Même format d'URL.

- **`wholeTextFiles(FILENAME)`**
  Permet de charger des fichiers textes entiers: le retour contiendra une liste où chaque élément contient l'intégralité du contenu d'un fichier.

API Python – Lecture des données

- **`pickleFile(FILENAME)`**
  Permet de charger un ou plusieurs fichiers sérialisés Python « pickle »; souvent utilisé pour charger des données sauvegardées de la même manière. Même format d'URL.

- **<u>Remarque</u>: toutes les fonctions ci-dessus disposent d'un autre argument, « `numPartitions` », permettant d'indiquer la quantité minimum de partitions qu'on souhaite avoir au sein du RDD pendant et après chargement des données; ce qui peut avoir un effet sur le parallélisme plus tard lors de l'exécution.**

  Certaines des fonctions sus-citées ont également d'autres arguments non décrits ici pour ajuster leur fonctionnement.

API Python – Sauvegarde des données

- **Il est également possible de sauvegarder les données d'un RDD sur le système de fichier (HDFS ou disque). Comme il s'agit de données issues de l'exécution d'un programme parallélisé, elles seront stockées dans un répertoire, pas un fichier; et en plusieurs fragments (nommés `part-XXXXX`, où `XXXXX` est un numéro incrémental).**

- **Une liste non exhaustives de fonctions permettant la sauvegarde:**

  - **`saveAsTextFile(DIRNAME)`**
    Sauvegarde les résultats dans un ou plusieurs fichiers texte dans le répertoire indiqué. Le format d'URL à utiliser est le même que précédemment (« hdfs:// », « file:// »).

API Python – Sauvegarde des données

- **`saveAsPickleFile(DIRNAME)`**
  Sauvegarde les résultats dans un ou plusieurs fichiers python sérialisés « pickle » dans le répertoire indiqué. Le format d'URL à utiliser est le même que précédemment (« hdfs:// », « file:// »). Souvent utilisé pour un futur chargement via `pickleFile()`.

- **`saveAsSequenceFile(DIRNAME)`**
  Sauvegarde les résultats dans un ou plusieurs fichiers séquence dans le répertoire indiqué. Le format d'URL à utiliser est le même que précédemment (« hdfs:// », « file:// »). Souvent utilisé pour un futur chargement via Java/Scala (Hadoop / Spark / autres).

API Python – Créer des données

- **Au delà d'un chargement depuis un support externe, on peut aussi créer un RDD. Pour ce faire, on dispose d'une méthode appelable sur le SparkContext:**

```
parallelize(LIST, numSlices=NUM_PARTITIONS)
```

La méthode convertit la liste passée en paramètre en RDD (en effectuant le partitionnement; avec une valeur par défaut ou la valeur spécifiée), et renvoie un *handle* vers ce RDD. Par exemple:

```
rdd=sc.parallelize([0, 1, 2, 3, 4], numSlices=5)
rdd=sc.parallelize(['item-#%d' % (x) for x in xrange(0, 100)])
```

API Python – Transformations

- **On dispose de nombreuses transformations applicables aux RDD. Une liste non exhaustive (toutes s'appellent sur un RDD):**

  - `coalesce(numPartitions)`
    Renvoie un RDD résultant du re-partitionnement du RDD avec le nombre de partitions indiquées, qui doit être inférieur au nombre de partitions actuel.

  - `distinct(numPartitions=None)`
    Renvoie un RDD où tous les éléments en double du RDD d'origine sont supprimés.

  - `filter(FONCTION(1 arg))`
    Applique la fonction spécifiée sur chacun des éléments et si elle renvoie `true`, garde l'élément dans le RDD de retour.

API Python – Transformations

Par exemple:

```
rdd=sc.parallelize([1, 2, 3, 4, 5, 6])
rdd2=rdd.filter(lambda x: x>3)
rdd2.collect()
[4, 5, 6]
```

Ou encore (sans fonction Lambda):

```
def filtre(x):
  return(x%2==0)
rdd=sc.parallelize([1, 2, 3, 4, 5, 6])
rdd2=rdd.filter(filtre)
rdd2.collect()
[2, 4, 6]
```

API Python – Transformations

- **`map(FONCTION(1 arg))`**
  Renvoie un RDD résultant de l'exécution de la fonction spécifiée sur chaque élément du RDD source (MAP). Exemples:

```
rdd=sc.parallelize(["celui", "ciel", "celui", "qui"])
rdd2=rdd.map(lambda x: (x, 1))
rdd2.collect()
[('celui', 1), ('ciel', 1), ('celui', 1), ('qui', 1)]

rdd=sc.parallelize([1, 2, 3, 4])
rdd2=rdd.map(lambda x: x*2)
rdd2.collect()
[2, 4, 6, 8]
```

API Python – Transformations

- **`flatMap(FONCTION(1 arg))`**
  Renvoie un RDD résultant de l'exécution de la fonction spécifiée sur chaque élément du RDD source (MAP), aprés que chacune des valeurs de retour ait été dépliée (comme le FLATTEN de Pig); c'est à dire que toute liste renvoyée par la fonction map est dépliée et ses membres deviennent des éléments du RDD. Cela permet à la fonction map de renvoyer plusieurs valeurs. Par exemple:

```
rdd=sc.parallelize(["un mot", "deux"])
rdd2=rdd.map(lambda x: x.split())
rdd3=rdd.flatMap(lambda x: x.split())
rdd2.collect()
[['un', 'mot'], ['deux']]     # Map normal
rdd3.collect()
['un', 'mot', 'deux']     # Map déplié
```

API Python – Transformations

- **`mapValues(FONCTION(1 arg))`**
  Renvoie un RDD résultant de l'exécution de la fonction spécifiée sur chaque valeur du RDD source, sans changer la clef; suppose que le RDD source contient des couples (clef;valeur) (tuples). Exemple:

```
rdd=sc.parallelize([("qui", 20), ("ciel", 12)])
rdd.mapValues(lambda x: x*2).collect()
[('qui', 40), ('ciel', 24)]
```

  Qui est d'ailleurs équivalent à:

```
rdd.map(lambda x: (x[0], x[1]*2)).collect()
[('qui', 40), ('ciel', 24)]
```

API Python – Transformations

- **`flatMapValues(FONCTION(1 arg))`**
  Renvoie un RDD résultant de l'exécution de la fonction spécifiée sur chaque valeur du RDD source, sans changer la clef; suppose que le RDD source contient des couples (clef;valeur) (tuples); et déplie les tuples renvoyés par la fonction dans le RDD final. C'est l'équivalent de `flatMap` mais avec le comportement de `mapValues` en terme de respect de la clef. Exemple:

```
rdd=sc.parallelize([("qui", 20), ("ciel", 12)])
rdd.flatMapValues(lambda x: (x-5, 5)).collect()
[('qui', 15), ('qui', 5), ('ciel', 7), ('ciel', 5)]
```

Qui est d'ailleurs équivalent à:

```
rdd.flatMap(lambda x: [(x[0], x[1]-5), (x[0], 5)]).collect()
[('qui', 15), ('qui', 5), ('ciel', 7), ('ciel', 5)]
```

API Python – Transformations

- `reduceByKey(FONCTION(2 args))`
  Renvoie un RDD résultant d'un regroupement par clef, suivi d'une fonction reduce appliquée à chaque clef distincte et émettant une valeur unique. La fonction spécifiée est le reduce; elle prend deux arguments en paramètre (deux valeurs associées à une même clef) et n'en retourne qu'une. Cette fonction sera appliquée aux valeurs de chaque clef jusqu'à ce qu'elles soient toutes réduites. Suppose que le RDD contient des couples (clef;valeur). Exemple:

```
rdd=sc.parallelize([("qui", 1), ("ciel", 1), ("qui", 1),
                    ("ciel", 1), ("qui", 1), ("qui", 1)])
rdd.reduceByKey(lambda a,b: a+b).collect()
[('qui', 4), ('ciel', 2)]
```

C'est l'équivalent d'un shuffle + reduce Hadoop.

API Python – Transformations

- **`groupBy(FONCTION(1 arg))`**
  Renvoie un RDD résultant de l'exécution de la fonction spécifiée sur chaque valeur du RDD source, et en récupérant la valeur ainsi obtenue comme clef; la valeur source du RDD utilisée est ajoutée à une liste correspondant à cette clef dans le RDD de sortie; la valeur de retour contient un iterable sur cette liste.

```
rdd=sc.parallelize([1, 2, 3, 4, 5])
res=rdd.groupBy(lambda x: x%2).collect()
[(0, <pyspark.resultiterable...>),
 (1, <pyspark.resultiterable...>)]
print [(x, sorted(y)) for (x, y) in res]
[(0, [2, 4]), (1, [1, 3, 5])]
```

API Python – Transformations

- **`groupByKey()`**
  Regroupe les éléments du RDD par clef distincte et unique; renvoie un RDD de retour à un format similaire à celui de `groupBy()`. Suppose que le RDD source contient des couples (clef;valeur). Exemple:

```
rdd=sc.parallelize([("qui", 1), ("ciel", 1), ("qui", 1),
                    ("ciel", 1), ("qui", 1), ("qui", 1)])
res=rdd.groupByKey().collect()
[("qui", <pyspark.resultiterable...>),
 ("ciel", <pyspark.resultiterable...>)]
print [(x, sorted(y)) for (x, y) in res]
[('qui', [1, 1, 1, 1]), ('ciel', [1, 1])]
```

… c'est l'équivalent de l'étape de « shuffle » Hadoop (pré-reduce).

API Python – Transformations

- `intersection(RDD)`
  Renvoie un RDD résultant de l'intersection avec un autre RDD. Exemple:

```
rdd=sc.parallelize([1, 2, 3, 4, 5])
rdd2=sc.parallelize([2, 4, 8])
rdd.intersection(rdd2).collect()
[2, 4]
```

- `union(RDD)`
  Renvoie un RDD résultant de l'union avec un autre RDD. Exemple:

```
rdd=sc.parallelize([1, 2, 3, 4, 5])
rdd2=sc.parallelize([2, 4, 8])
rdd.union(rdd2).collect()
[1, 2, 3, 4, 5, 2, 4, 8]
```

API Python – Transformations

- `subtract(RDD)`
  Renvoie un RDD résultant de la soustraction avec un autre RDD. Exemple:

```
rdd=sc.parallelize([1, 2, 3, 4, 5])
rdd2=sc.parallelize([2, 4, 8])
rdd.subtract(rdd2).collect()
[1, 3, 5]
```

- `sortBy(FONCTION(1 arg))`
  Renvoie un RDD aprés tri du RDD source; la fonction renvoie la clef à utiliser pour le tri. Par exemple:

```
rdd=sc.parallelize([('z', 1), ('y', 2), ('m', 3), ('e', 4)])
rdd.sortBy(lambda x: x[0]).collect()
[('e', 4), ('m', 3), ('y', 2), ('z', 1)]
rdd.sortBy(lambda x: x[1]).collect()
[('z', 1), ('y', 2), ('m', 3), ('e', 4)]
```

API Python – Transformations

- `join(RDD)`
- `fullOuterJoin(RDD)`
- `leftOuterJoin(RDD)`
- `rightOuterJoin(RDD)`

  Permet d'effectuer des jointures entre le RDD courant et le RDD spécifié; les jointures sont effectuées sur les clef (la fonction suppose que les RDD contiennent des couples (clef;valeur)). Les jointures sont respectivement « INNER », « FULL OUTER », « LEFT OUTER » et « RIGHT OUTER ».

  Les fonctions de jointure renvoient des tuples dont le premier membre est une des valeurs de clef et le second un tuple composé de la valeur correspondante trouvée dans le premier RDD et de la valeur correspondante trouvée dans le second. Il peut y en avoir plusieurs (si un des RDD contenait plusieurs fois la même clef); et si une valeur n'a pas été trouvée dans l'autre RDD, `None` est indiqué.

API Python – Transformations

Par exemple:

```
rdd=sc.parallelize([('a', '1'), ('b', 4), ('c', 9)])
rdd2=sc.parallelize([('a', '0'), ('c', 2), ('d', 7)])
rdd.join(rdd2).collect()
[('a', ('1', '0')), ('c', (9, 2))]

rdd.fullOuterJoin(rdd2).collect()
[('a', ('1', '0')), ('c', (9, 2)), ('b', (4, None)), ('d',
(None, 7))]

rdd.leftOuterJoin(rdd2).collect()
[('a', ('1', '0')), ('c', (9, 2)), ('b', (4, None))]

rdd.rightOuterJoin(rdd2).collect()
[('a', ('1', '0')), ('c', (9, 2)), ('d', (None, 7))]
```

API Python – Transformations

- **`setName(STR)`**
  Permet de donner un nom au RDD; ce nom est susceptible d'être utilisé dans certains formats de sérialisation et dans les logs de Spark.

- **`cache()`**
  Permet de demander à ce que le RDD soit persisté; c'est justement cette fonction qui indique à Spark qu'il doit garder le RDD en question en cache car il est susceptible d'être évalué plusieurs fois. Cet appel ne déclenche pas immédiatement l'évaluation; il permet simplement de s'assurer que le RDD sera mis en cache / persisté la première fois que cette évaluation sera réalisée.

- **`unpersist()`**
  L'inverse de `cache()`; indique que ce RDD est temporaire et ne dois pas être persisté.

API Python – Actions

- **On dispose également d'actions applicables aux RDD; celles-ci déclenchent toutes une évaluation du RDD.**

  - `collect()`
    Permet de récupérer le contenu d'un RDD directement dans le programme source sous la forme d'une liste. Attention: les données sont mises en RAM coté driver; si beaucoup de données sont présentes dans le RDD, cet appel peut échouer pour manque de mémoire.

  - `count()`
    Renvoie le nombre d'éléments dans le RDD (un int).

API Python – Actions

- **`collectAsMap()`**
  Comme `collect()`, mais retourne une *hashmap* qui associe clefs et valeurs. Suppose que le RDD contient des tuples (clef;valeur). Même remarque que pour `collect()`.

- **`countByKey()`**
  Renvoie le nombre d'éléments dans le RDD pour chaque clef distincte, sous la forme d'une *hashmap* (clef => quantité). Suppose que le RDD contient des tuples (clef;valeur)

- **`countByValue()`**
  Renvoie le nombre d'éléments dans le RDD pour chaque valeur distincte, sous la forme d'une liste de tuples (valeur; quantité). Suppose que le RDD contient des tuples (clef;valeur)

API Python – Actions

- `first()`
  Renvoie le premier élément du RDD.

- `getNumPartitions()`
  Renvoie le nombre de partitions du RDD.

- `glom()`
  Renvoie un RDD où les données sont séparées en N listes: une pour chaque partition du RDD d'origine. Exemple:

```
rdd=sc.parallelize([1, 2, 3, 4, 5], 2)
rdd2=sc.parallelize([1, 2, 3, 4, 5], 3)
rdd.glom().collect()
[[1, 2], [3, 4, 5]]
rdd2.glom().collect()
[[1], [2, 3], [4, 5]]
```

API Python – Actions

- `isEmpty()`
  Renvoie `true` si le RDD est vide.

- `max(key=FONCTION(1 arg))`
  Renvoie l'élément maximal dans le RDD; avec optionnellement une fonction à appeler renvoyer une valeur numérique pour effectuer une comparaison.

- `min(key=FUNCTION(1 arg))`
  L'inverse de `max()`.

- `repartition(numPartitions)`
  Retourne un RDD repartitionné avec le nombre de partitions indiquées. Provoque un *shuffle* entier (au sens Spark) sur toutes les données; coûteux. Si on souhaite *diminuer* le nombre de partitions, il est possible d'utiliser `coalesce` (déjà discuté) qui ne provoque pas de re-shuffle global.

API Python – Actions

- `reduce(FONCTION(2 args))`
  Applique un reduce sur le RDD et renvoie une valeur unique; la fonction à spécifier prends deux arguments correspondant à deux éléments de la liste d'origine (et doit en renvoyer un). A noter qu'il s'agit bien d'une action et pas d'une transformation. Exemple:

```
rdd2=sc.parallelize([1, 2, 3, 4, 5])
print rdd2.reduce(lambda a,b: a+b)
15
def red(a, b):
  if a>b:
    return(a)
  return(b)
print rdd2.reduce(red)
5
```

API Python – Actions

- `foreach(FONCTION(1 arg))`
  Applique une fonction sur chacun des éléments du RDD (par exemple, print).

- `foreachPartition(FONCTION(1 arg))`
  Applique une fonction sur chacune des partitions du RDD; l'argument passé à la fonction est un iterable de tous les éléments de la partition.

- `saveAsTextFile`
- `saveAsPickleFile`
- `SaveAsSequenceFile`
  Déjà discutées dans la partie « Sauvegarde de données ».

API Python – Remarque

- **Il y a évidemment (quelques) autres transformations et actions, mais les plus importantes ont été vues.**

- **Les APIs Java et Scala sont extrêmement proches (mêmes noms de fonctions, mêmes arguments); de même, dans une moindre mesure, pour l'API R.**

- **Pour plus d'informations, se référer à la documentation:**

  https://spark.apache.org/docs/1.2.0/api/python/

API Python – Exemples

- **Le compteur d'occurrences de mots, version Spark/Python:**

```
from pyspark import SparkContext

sc = SparkContext("local[2]", "Wordcount")
lines=sc.textFile('hdfs:///poeme.txt')
words=lines.flatMap(lambda x: x.split())
tuples=words.map(lambda x: (x, 1))
res=tuples.reduceByKey(lambda a,b: a+b)
res.saveAsTextFile('hdfs:///res')
```

API Python – Exemples

- **Le détecteur d'anagrammes (noter l'usage de `.cache()`):**

```
from pyspark import SparkContext

sc = SparkContext("local[2]", "Anagrammes")
words=sc.textFile('hdfs:///common_words_en_subset.txt')
tuples=words.map(lambda x: (''.join(sorted(list(x))), x))
grouped=tuples.groupByKey().mapValues(lambda x: list(x))
grouped.cache()
filtered=grouped.filter(lambda x: len(x[1])>1)
res=filtered.reduceByKey(lambda a,b: "%s, %s" % (a, b))
res.saveAsTextFile('hdfs:///res-filtered')
res2=grouped.reduceByKey(lambda a,b: "%s, %s" % (a, b))
res2.saveAsTextFile('hdfs:///res-unfiltered')
```

API Java

- **Le compteur d'occurrences de mots, version Spark/Java:**

```
public class WordCount {
  public static void main(String[] args)
  {
    SparkConf conf = new SparkConf().setAppName("Wordcount");
    JavaSparkContext sc = new JavaSparkContext(conf);
    JavaRDD<String> lines=sc.textFile("file:///poeme.txt");

    JavaRDD<String> words=lines.flatMap(new
FlatMapFunction<String, String>() {
public Iterator<String> call(String s)
{ return(Arrays.asList(s.split(" ")).listIterator()); }
          });
```

API Java


```
    JavaPairRDD<String, Integer> tuples=words.mapToPair(new
PairFunction<String, String, Integer>() {
     public Tuple2<String, Integer> call(String s)
{ return(new Tuple2(s, 1)); }
          });
      JavaPairRDD<String, Integer> res=tuples.reduceByKey(new
Function2<Integer, Integer, Integer>() {
              public Integer call(Integer a, Integer b)
{ return(a+b); }
          });
      res.saveAsTextFile("hdfs:///res-java");
  }
}
```

API Scala

- **Le compteur d'occurrences de mots, version Spark/Scala:**

```
object WordCount {
  def main(args: Array[String]) {
    val conf=new SparkConf().setAppName("Wordcount")
    val sc=new SparkContext(conf)
    val lines=sc.textFile("hdfs:///poeme.txt")
    val words=lines.flatMap(line => line.split())
    val tuples=words.map(word => (word, 1))
    val res=words.reduceByKey(_+_)
    res.saveAsTextFile("hdfs:///res")
  }
}
```

Variables *broadcast* et Accumulateurs

- **Quand Spark exécute un programme, qu'il soit en Python ou en Java/Scala, il envoie le script ou le .jar du programme aux *workers*, qui travailleront dessus chacun indépendamment (et chacun sur une partition différente des RDDs).**

- **Cela veut dire que si on déclare par exemple une variable globale au sein du programme et qu'on l'utilise ou qu'on la modifie dans un des traitements *workers* – une des fonctions *closures* utilisées dans les transformations - alors la variable ne sera *pas* modifiée correctement: on accède en effet lors de l'exécution à une copie de la variable d'origine du driver; on est en effet dans un des *workers* et la variable est donc la copie créée lors de l'exécution Python coté *worker*.**

- **L'effet est un peu similaire aux contraintes induites par un *fork* au sein d'un programme; les variables globales ne sont plus les mêmes d'un coté et de l'autre.**

Les variables *broadcast*

- **Pour permettre au *driver* de distribuer efficacement des données à tous les *workers*, Spark introduit le concept de variables *broadcast*; il s'agit de variables définies dans le programme qui ne seront pas distribuées aux *workers* avec les tâches dés que c'est nécessaire; à la place, elles seront envoyées dés *broadcast* à tous les *workers* qui garderont la variable en question en cache. Elles seront cependant en lecture seule.**

- **L'idée est d'éviter d'envoyer à plusieurs reprises les mêmes données volumineuses aux *workers*; et le mécanisme est utile principalement quand on cherche à utiliser des données de taille importante au sein de fonctions *closure* utilisées dans les opérations dans plusieurs « étapes » (stages) différentes du programme.**

Les variables *broadcast*

- **Pour créer une variable *broadcast*, on utilise:**

```
sc.broadcast(VARIABLE)
```

… la fonction renvoie la variable de broadcast (un *wrapper* autour de la variable d'origine); on peut accéder à sa valeur via `.value`. Par exemple:

```
years=[1953, 1962, 1981, 1999, 2011]
years_bc=sc.broadcast(years)
print years_bc.value
```

… ici, on utiliserait ensuite désormais `years_bc` au sein des opérations, et plus la variable originale `years`; ce qui aura pour effet d'éviter que `years_bc` soit envoyée plusieurs fois aux *workers*. On ne devra plus modifier la variable en question.

Les accumulateurs

- **De même, Spark introduit le concept d'*accumulateurs*: des variables spécifiques qui sont incrémentables par les *workers* et dont la valeur peut être lue par le *driver*.**

- **Ce mécanisme vient lui aussi essayer d'aider à passer outre la limitation due au parallélisme / au fait que l'exécution se produise dans des environnements séparés en terme de mémoire; il permet aux *workers* de remonter une information au *driver.***

- **Il n'y a pas de conflit en terme d'accès concurrents puisque les *workers* se contentent d'incrémenter la variable et que seul le *driver* peut la lire.**

- **Couramment utilisé pour remonter des informations de progression, ou des informations simples permettant au *driver* de modifier le comportement du programme.**

Les accumulateurs

- **On pourra créer un accumulateur en faisant:**

  ```
  sc.acumulator(VALUE)
  ```

  … en indiquant une valeur initiale. La fonction renvoie l'accumulateur; et celui-ci est incrémentable *via* la fonction `.add()`.

- **Au sein d'une fonction utilisée dans un opérateur (donc exécutée sur les *workers*), on pourra donc appeler `.add()` sur cette variable pour incrémenter l'accumulateur.**

- **Le driver pourra de son coté lire la valeur de l'accumulateur en accédant au membre `.value` de la variable ainsi créée.**

Les accumulateurs

- **Exemple:**

```
nbwords=sc.acumulator(0)
words=sc.textFile("hdfs:///common_words.txt")
def mapfunc(x):
  global nbwords
  nbwords.add(1)
  return(len(x))
lenwords=words.map(mapfunc)
# Ici, nbwords vaut toujours 0 car pas encore d'évaluation.
lenwords.saveToTextFile("hdfs:///len-words")
# Désormais, nbwords a pour valeur le nombre de mots.
```