Bonjour

Apache Spark

Introduction

Ce cours présente le système de programmation Spark.

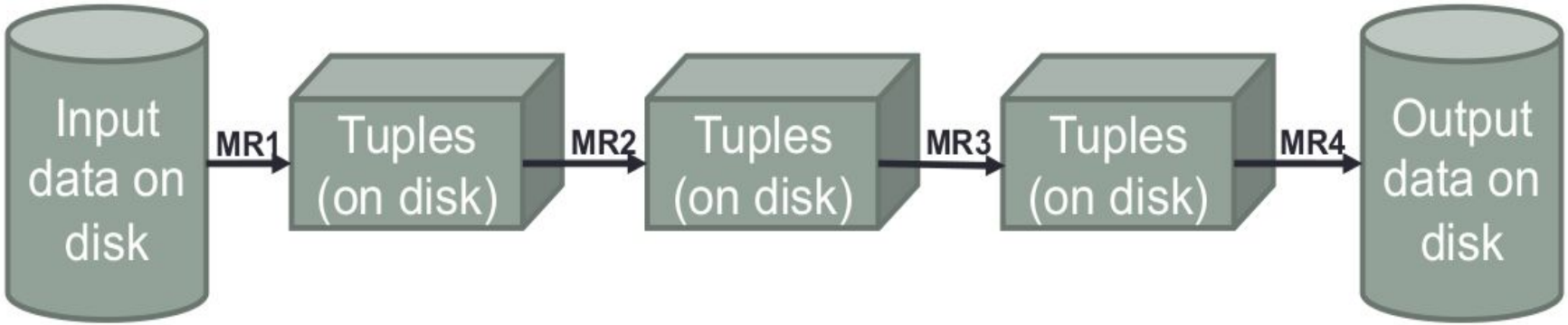
Un autre mécanisme pour écrire des programmes de type MapReduce

Il est nettement plus performant, polyvalent et facile d'utilisation comparé à Map-reduce sur Hadoop

Why Spark ?

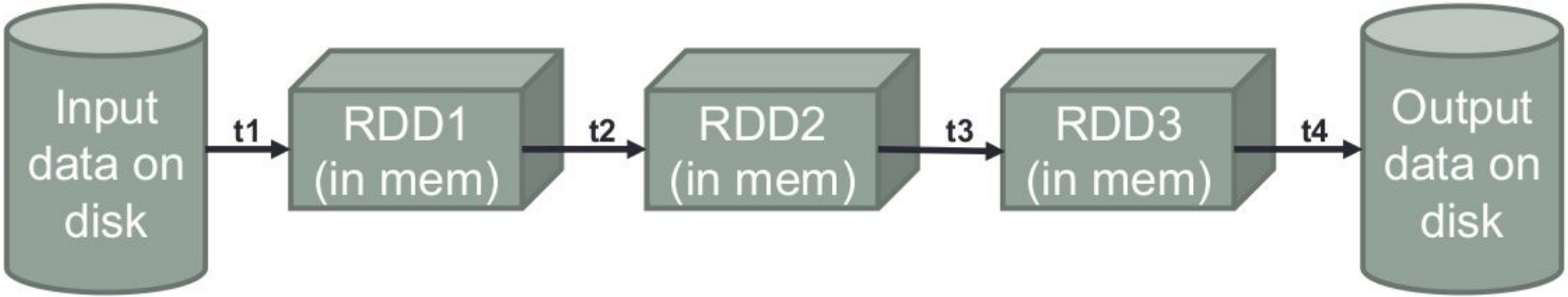
Pourquoi Spark?

Enchaînement de jobs sur Hadoop



Pourquoi Spark ?

Enchaînement de jobs sur Spark



un peu de lecture
https://www.lemondeinformatique.fr/actualites/lire-hadoop-vs-spark-apache-5-choses-a-savoir-63271.html

EPITA

Pourquoi Spark ?

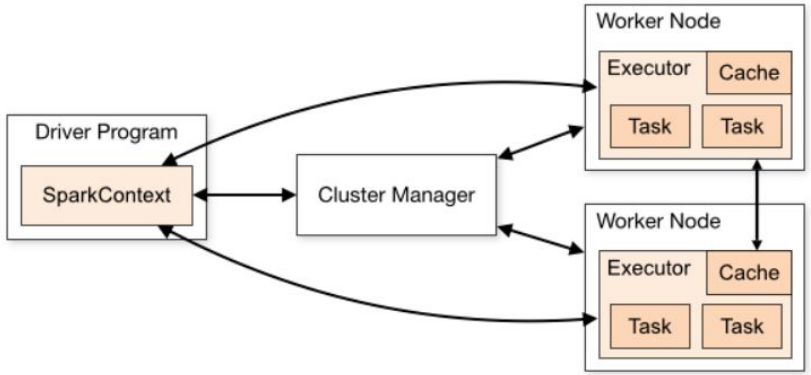
- Commence à Berkeley en 2009. Devenu un projet de la fondation Apache depuis 2013
- http://spark.apache.org/ ‘Apache Spark is a fast and general engine for large-scale data processing’
- Plus rapide que Map-reduce de Hadoop (10* sur DD) et (100* en mémoire)
- fournit des API pour Java, Scala, Python et récemment R
- S’intègre à Hadoop et son écosystème (notamment HDFS)

EPITA

Fonctionnement de Spark

1. Les applications Spark s'exécutent de manière indépendante sur un cluster, coordonné par l'objet SparkContext dans votre programme principal
2. SparkContext se connecte au Cluster Manager qui alloue des ressources entre les applications. Une fois connecté, Spark obtient des « executors » sur les nœuds du cluster
3. Les « executors » sont des processus qui réalisent des calculs et stockent des données pour votre application

https://spark.apache.org/docs/latest/cluster-overview.html



Fonctionnement de Spark

4. Spark envoie votre code (Java JAR ou fichier Python) aux « executors »
5. Finalement SparkContext envoie des tâches à exécuter aux « executors »

Spark

- ➔ Spark est une API de programmation parallèle sur des données
- ➔ L'objet principal dans Spark est le RDD : Resilient Distributed Dataset
  - ◆ C'est un dispositif pour traiter une collection de données par des algorithmes parallèles robustes
  - ◆ Un RDD ne contient pas vraiment de données, mais seulement un traitement
- ➔ Spark utilise l'évaluation paresseuse (lazy evaluation) : traitement effectué que lorsque cela est nécessaire
- ➔ Spark fait en sorte que le traitement soit distribué sur le cluster, donc calculé rapidement

Avantages de Spark

- Spark permet d'écrire des traitements plus complexes
- Les jobs Yarn sont assez longs à lancer et exécuter. Il y a des temps de latences considérables
- Au contraire, Spark utilise beaucoup mieux la RAM des machines et gère lui-même l'enchaînement des tâches

➔ Les traitements peuvent être écrits dans plusieurs langages : scala, java et Python. On utilisera Python pour sa simplicité.

Le début d'un programme Spark

Un programme pySpark commence toujours par :

```
from pyspark import SparkConf, SparkContext
nom_appli = "wordcount"
config = SparkConf().setAppName(nom_appli)
#config.setMaster("spark://master:7077")
sc = SparkContext(conf=config)
```

sc : représente le contexte Spark. C'est un objet contenant plusieurs méthodes dont celles qui instancient les RDD

La ligne commentée, config.setMaster() permet de définir l'url du spark

Master, c'est à dire le cluster sur lequel lancer l'exécution, (yarn, spark, local, etc)

Exemple

Soit le fichier de données sur les arbres remarquables à Paris.

Chaque ligne décrit un arbre : Id, position gps, arrondissement, genre, taille, etc

On souhaite afficher l'année de plantation de l'arbre le plus grand.

Let's do it !!

Principes du traitement

Voyons comment faire cela sur Spark ?

1. séparer les champs de ce fichier
2. extraire les champs souhaités (hauteur et année de plantation). On en fait une paire (clé, valeur)
3. éliminer la clé correspondant à la ligne de titre du fichier et les clés vides
4. Convertir les clés en float
5. classer les paires selon la clé dans l'ordre croissant
6. afficher la première des paires. C'est le résultat voulu

Programme PySpark

```
from pyspark import SparkConf, SparkContext
sc = SparkContext(conf=SparkConf().setAppName("arbres"))
arbres = sc.textFile('path_to/arbres.csv')
arbres_splits = arbres.map(lambda ligne: ligne.split(';'))
paires = arbres_splits.map(lambda champs: (champs[12],champs[16]))
paires_filtrees = paires.filter(lambda paire : paire[0]!='' and paire[0]!='HAUTEUR EN M')
paires2 = paires_filtrees.map(lambda paire : (float(paire[0]), paire[1]))
classement = paires2.sortByKey(ascending=False)
print (classement.first())
```

EPITA

Fonction Lambda ou fonction nommée

- Le programme précédent fait appel à une lambda function. Ce sont des fonctions sans noms
- La syntaxe est :

lambda paramètres : expression

Le but est de créer un traitement (fonction) qu'on peut appeler dans un map

Fonction Lambda ou fonction nommée

- ➔ Complexité :
  - ◆ Une lambda ne peut pas contenir un algorithme complexe. Elles sont limitées à une seule expression
  - ◆ Au contraire, une fonction peut contenir des boucles, des tests, des affectations...
- ➔ Lisibilité
  - ◆ Les lambda sont beaucoup moins lisibles que les fonctions
- ➔ Praticité
  - ◆ Les lambda sont très courtes et plus pratiques, à écrire rapidement et sur place, tandis que les fonctions doivent être définies ailleurs que là où on les emploie

Avertissement

Spark fait tourner les traitements sur des machines différentes afin d'accélérer le traitement global.

Il ne faut donc surtout pas affecter des variables globales dans les fonctions (elles ne pourront pas être transmises d'une machine à l'autre)

Exemple de ce qu'il ne faut pas faire

```
total = 0
def cumuler(champs):
    global total
    total += float(champ[12])
    return champ[16]
annees = tableau.map(cumuler)
```

EPITA

Lancement d'un programme Spark

Spark offre plusieurs façons de lancer le programme, dont :

- Lancement en local :
  - spark-submit programme.py
  - python3 programme.py (en ajoutant setMaster ('local[*]') : * pour utiliser tous les coeurs de la machine, sinon local[nb] pour utiliser nb coeurs
- Lancement sur un cluster de Spark Workers (Standalone Mode) :
  - spark-submit -- master spark://master:7077 programme.py
  - Indique de faire appel au cluster de machines sur lesquelles tournent des Spark Workers
  - start-master.sh , start-worker.sh spark://idir-pc:7077 (à adapter sur votre machine)
- Lancement sur YARN:
  - spark-submit -- master yarn programme.py, ce sont les esclaves Yarn qui vont exécuter le programme

EPITA

API SPARK

Principes

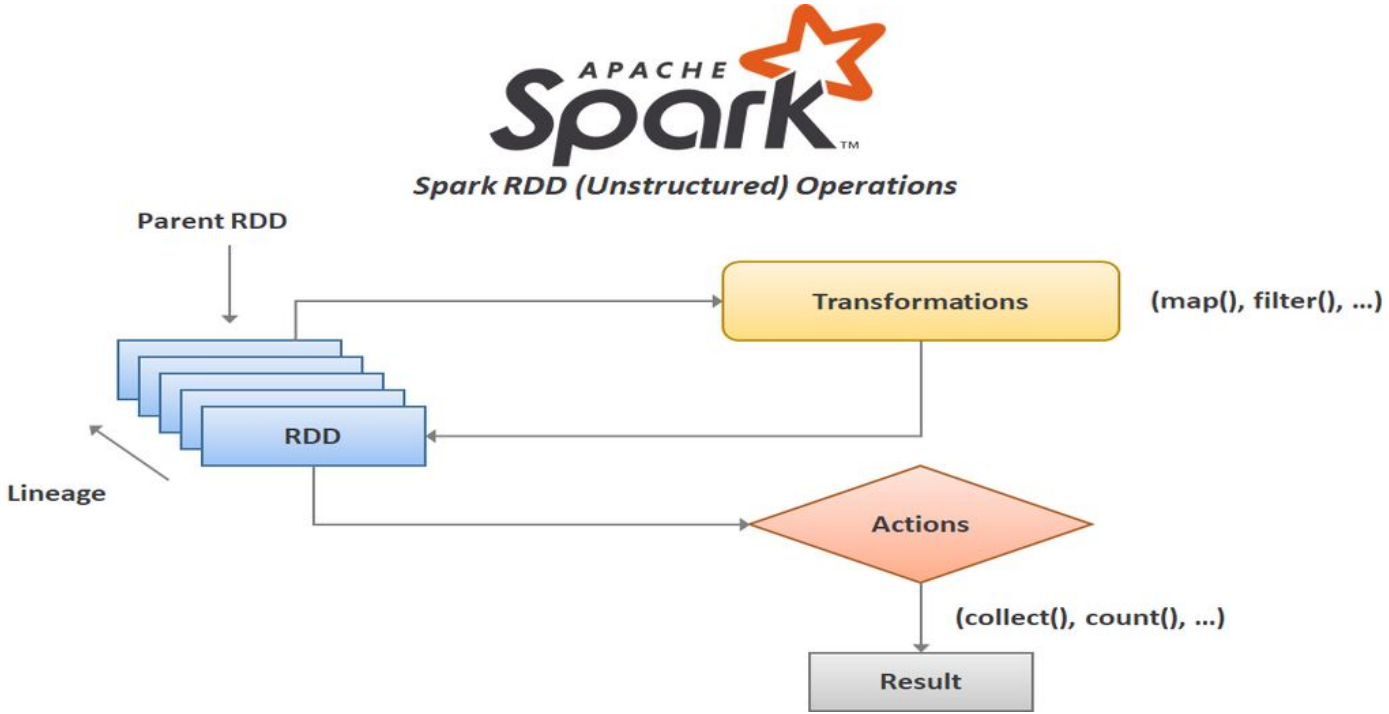
Spark est facile à prendre en main. Il repose sur des principes peu nombreux et simples. Voici la doc complète:

https://spark.apache.org/docs/latest/rdd-programming-guide.html

Principes

- ➔ Données
  - **RDD :** notion principale sur spark. Un RDD représente une collection de données distribuées, pouvant être modifié par une transformation.
  - Données partagées entre les traitements et distribuées sur le cluster de machines
- ➔ Méthodes
  - **Transformations** : ce sont des méthodes du type RDD → transformation(RDD). Créer un nouveau RDD à partir d'un RDD existant
  - **Actions** : ce sont des méthodes qui permettent d'extraire des informations des RDD, par exemple : afficher sur stdout, enregistrer, reduce, collecter, etc

Principes



EPITA

RDD

- Un RDD est une collection de données abstraite
- Un RDD est distribué, c'est à dire réparti sur plusieurs machines afin de paralléliser les traitements
- Paralléliser une collection :

```
data = [1, 2, 3, 4, 5]
distData = sc.parallelize(data)
```

EPITA

RDD

Importer un jeu de données en RDD

- Données sur HDFS

```
data = sc.textFile("hdfs:///user/idir/data/mon_dataset.txt")
```

- Données sur machine locale

```
data = sc.textFile("file:///home/idir/toto.csv")
```

Chaque ligne du fichier constitue un enregistrement. Les lignes du fichier sont distribuées sur différentes machines pour un traitement parallèle

Transformations (1) :map, filter, flatmap

- RDD.map(fonction) : chaque appel à la fonction doit retourner une valeur qui est mise dans le RDD résultant

```
RDD = sc.parallelize([1,2,3,4,5,6])
print (RDD.map(lambda n: n+1).collect())
```

- RDD.filter(fonction): la fonction doit retourner un booléen. En sortie, on aura un RDD avec que les éléments pour lesquels la fonction retourne True

```
RDD = sc.parallelize([1,2,3,4])
print (RDD.filter(lambda n: (n%2)==0).collect())
```

- RDD.flatMap(fonction) : chaque appel à la fonction doit retourner une liste et toutes les listes sont concaténées dans le RDD résultant

```
RDD = sc.parallelize([0,1,2,3,4,5])
print (RDD.flatMap(lambda n: [n*2, n*2+1]).collect())
```

EPITA

Transformations ensemblistes : union, intersection, distinct

- RDD.distinct() : retourne un seul exemplaire de chaque élément

```
RDD = sc.parallelize([1, 2, 3, 4, 6, 5, 4, 3])
print (RDD.distinct().collect())
```

- RDD1.union(RDD2): retourne la concaténation des deux RDD. Pour faire une vraie union, rajouter .distinct()

```
RDD1 = sc.parallelize([1,2,3,4])
RDD2 = sc.parallelize([6,5,4,3])
print (RDD1.union(RDD2).collect())
print (RDD1.union(RDD2).distinct().collect())
```

- RDD1.intersection(RDD2) : retourne l'intersection des deux RDD

```
RDD1 = sc.parallelize([1,2,3,4])
RDD2 = sc.parallelize([6,5,4,3])
print (RDD1.intersection(RDD2).collect())
```

Transformations sur des paires (clé, valeur)

- RDD.groupByKey(): retourne un RDD dont les éléments sont des paires (clé, liste des valeurs ayant cette clé dans le RDD)
- RDD.sortByKey(ascending) : retourne un RDD dont les clés sont triées. Mettre True ou False pour le type de tri
- RDD.reduceByKey(fonction) : regroupe les valeurs ayant la même clé et leur applique la fonction

Transformations (jointure)

Similaire aux jointures SQL.

- Rdd1.join(Rdd2) : retourne toutes les paires (k,(v1,v2)) quand v1 et v2 ont la même clé

  On peut aussi faire du leftouter, rightouter, et fullouter Join (Voir la doc)

Liste de toutes les transformations

https://spark.apache.org/docs/latest/rdd-programming-guide.html#transformations

Actions

Ce sont des méthodes qui s'appliquent à un RDD pour retourner une valeur, une collection, le premier élément, les n premiers éléments, le nombre d'éléments, etc

- liste = Rdd.collect(), retourne tout le Rdd sous forme de liste Python. Warning : attention à la taille !!
- nb = Rdd.count() : retourne le nombre d'éléments
- premier= Rdd.first() : retourne le premier élément
- premiers=Rdd.take(n) : retourne les n premiers éléments
- res= Rdd.reduce(fonction) : applique une fonction d'agrégation (commutative)

la liste de toutes les actions

https://spark.apache.org/docs/latest/rdd-programming-guide.html#actions

Merci !


EPITA
ÉCOLE D'INGENIEURS EN INFORMATIQUE