TP2 : Spark Core/RDD

© Mourad Ouziri
mourad.ouziri@u-paris.fr

Programme-objectifs :

- Traitement de données massives.
- Programmation Big Data avec l'API Spark *Core*.

Documentation :

Spark Scala API doc : https://spark.apache.org/docs/latest/api/scala
Scala API doc : https://www.scala-lang.org/api/2.11.10/#package
Java API doc : https://docs.oracle.com/javase/7/docs/api/

1. Téléchargement de Apache Spark

Manipulation 1: Télécharger Apache Spark 3.4.2 à partir de : https://spark.apache.org/downloads.html
Manipulation 2: Spark est compatible avec les versions Java de 11 à 17 (https://community.cloudera.com/t5/Community-Articles/Spark-and-Java-versions-Supportability-Matrix/ta-p/383669)
Manipulation 3: Décompresser Spark dans un répertoire local n'ayant aucun espace ni de caractère accentué dans le nom de tous les répertoires parents.

2. Démarrage de l'environnement Spark

Manipulation 4: (seulement si déploiement de Spark sur Hadoop) S'assurer que les services de Hadoop (*Namenode* et *Yarn*) sont bien démarrés.
Manipulation 5: (Pour un déploiement de Spark en mode *standalone*) Démarrer (dans un terminal de lignes de commandes) le *Master* de Spark : *./bin/spark-class org.apache.spark.deploy.master.Master*
Manipulation 6: Consulter la page Web du Master, accessible sur le port pa défaut 8080.
Manipulation 7: (Pour un déploiement de Spark en mode *standalone*) Démarrer trois Workers (dans trois terminaux de lignes de commandes) : *./bin/spark-class org.apache.spark.deploy.worker.Worker spark://{host}:{port} -m 1G -c 2*
Manipulation 8: Démarrer le client interactif de Spark (dans un terminal de lignes de commandes à part) : *spark-shell --master {spark://*host:port *ou yarn} --name TP1-Spark*
Manipulation 9: Vérifier que *spark-shell* a bien démarré dans Yarn ou Spark Master.
Manipulation 10: Consulter les commandes disponibles. Utiliser *:help*.
Manipulation 11: Vérifier la bonne création de *SparkContext* : variable *sc* et son type avec *:type*
Manipulation 12: Consulter les packages automatiquement inclus dans la session Spark. Utiliser *:imports*.

3. Partitionnement par défaut des données

Manipulation 13: Créer la liste de nombres de 0 à 10. Utiliser *List* ou *Array* ou *(0 to 10).toList*.
Manipulation 14: Créer une RDD en distribuant cette liste sur les *workers* du cluster Spark.

Utiliser la fonction Spark *parallelize*.
Manipulation 15: Afficher la RDD. Utiliser *foreach* et *println*.
Manipulation 16: Expliquer pourquoi rien ne s'affiche. Corriger à l'aide de la fonction *collect*.
Manipulation 17: Consulter le nombre de partitions de la RDD. Utiliser *getNumPartitions*.
Manipulation 18: Consulter le nombre de partitions par défaut avec : *sc.defaultParallelism*.
Manipulation 19: Expliquer le choix par défaut du nombre de partitions.
Manipulation 20: (*optionnel*) Vous pouvez changer le nombre de cœurs par défaut alloués à l'application *spark-shell* avec les options *--executor-cores* et *--total-executor-cores*
Manipulation 21: Afficher les partitions du RDD. Utiliser *glom* et *collect*.
Manipulation 22: Consulter la (ou les) partition(s) attribuée(s) à chaque *worker* en consultant leur sortie standard. Utiliser l'interface WebUI des *workers*.

4. Traitements basiques de RDD numérique

Manipulation 23: Afficher la taille (nombre d'éléments) de la RDD. Utiliser *count*.
Manipulation 24: Afficher les 3 premiers éléments de la liste. Utiliser la fonction *take*. Quelle est le type (transformation/action) de la fonction *take* ?
Manipulation 25: Multiplier tous les nombres du RDD par un facteur donné. Utiliser la fonction *map*, une fois avec une fonction *lambda* et une fois avec la fonction *fMultiplier (nb, facteur)* à coder.
Manipulation 26: Multiplier les nombres pairs du RDD par un facteur donné et ajouter un certain nombre donné (appelé *delta*) aux nombres impairs. Utiliser la fonction *map*, une fois avec la fonction *fModifier (nb, facteur, delta)* à coder.
Manipulation 27: Filtrer les éléments du RDD pour ne retenir que les nombres pairs. Utiliser la fonction *filter*. Afficher le RDD obtenu. Quelle est le type de retour de la fonction *filter* ?
Manipulation 28: Calculer la somme de tous les éléments du RDD. Utiliser *sum* (puis *reduce*).
Manipulation 29: Calculer le produit de tous les éléments du RDD. Utiliser *reduce*.
Manipulation 30: Calculer la somme des nombres pairs seulement. Utiliser *filter* et *sum*.
Manipulation 31: Calculer la somme des éléments par parité. Le résultat doit être un RDD contenant deux éléments : [("pairs", 20), ("impairs", 25)]. Utiliser *map* (ou *keyBy*) et *reduceByKey*.
Manipulation 32: Compter le nombre d'éléments par parité. Le résultat doit être un RDD contenant deux éléments : [("pairs", 4), ("impairs", 5)]. Utiliser *map* (ou *keyBy*) et *reduceByKey*.
Manipulation 33: Compter le nombre d'éléments par parité en utilisant cette fois-ci la fonction *countByKey* au lieu de *reduceByKey*. Quelle différence ? Laquelle est la plus adaptée au Big Data ?

5. Traitement de données structurées

Manipulation 34: Créer la liste de villes suivante dans une variable ("Paris FR 5", "Stuttgart DE 0.9", "Lyon FR 1.5", "Londres UK 8", "Berlin DE 2", "Marseille FR 3", "Madrid ES 5.5", "Liverpool UK 1.5", "Munich DE 1", "Barcelone ES 4.5","Seville ES 2"). Utiliser (la classe abstraite scala) *List* ou *Array*. Une ville est décrite par son nom, son pays et sa population exprimée en millions.
Manipulation 35: Partitionner cette liste sous forme de RDD, nommée *villesRDD*. Utiliser *parallelize*.
Manipulation 36: Afficher la RDD. Utiliser *collect* ou *foreach* et *println*.

Manipulation 37: Extraire le nom des villes seulement puis le pays seulement. Utiliser *map*.
Manipulation 38: Sélectionner et afficher les villes françaises seulement. Utiliser *filter*.
Manipulation 39: Refaire l'extraction du nom des villes (ou des pays) à l'aide de *map* qui prend cette fois-ci la fonction scala nommée : *extractVillePays(chaineVillePays :String, info :String) :String*. *Info* ∈ {"ville", "pays", "population"}
Manipulation 40: Refaire la sélection de villes d'un pays à l'aide de *filter* qui prend cette fois-ci la fonction nommée à écrire : *isVilleDuPays (chaineVillePays :String, pays :String) :Boolean*.
Manipulation 41: Calculer la population totale de tous les pays. Utiliser *map* et *reduce/sum*.
Manipulation 42: Calculer la population par pays. Utiliser *keyBy* et *reduceByKey*.
Manipulation 43: Afficher le nom des pays ayant à la fois des villes d'une population inférieure à 2.5 millions d'habitants et des villes d'une population supérieures à 2.5 millions d'habitants. Utiliser *keyBy, filter, intersect* et *distinct*.
Manipulation 44: Afficher le nom des pays n'ayant que des villes d'une population inférieure à 2.5 millions d'habitants. Utiliser *keyBy, filter, substract* et *distinct*.

6. Programmation de RDD d'objets

Manipulation 45: Coder la classe : *Personne (numero : Int, nom : String, prenom : String, dateNaiss : LocalDate, ville : String)*. Utiliser *case class*.
Manipulation 46: Charger le fichier *Personnels.csv* dans Spark. Utiliser *textFile*.
Manipulation 47: Créer une RDD de personnes (collection distribuée de type RDD [Personne]) à partir de la RDD précédente. Utiliser *map* et la fonction scala à coder : *textToObjetPersonne (lignePersonne :String):Personne* permettant de créer un objet *Personne* à partir d'une ligne du fichier *csv*. Utiliser la méthode *parse()* de *LocalDate* et *DateTimeFormatter* pour convertir le texte en *LocalDate*.
Manipulation 48: Afficher seulement le nom et la date de naissance des personnes.
Manipulation 49: Sélectionner les personnes nées avant 2000 et habitant une ville donnée.
Manipulation 50: Calculer le nombre de personnes par ville. Utiliser *reduceByKey*.
Manipulation 51: Calculer le nombre de personnes par décennie (années 80, 90, 2000, etc.) de naissance.
Manipulation 52: Afficher le nom et l'âge des personnes. Utiliser *map* et écrire une fonction scala pour le calcul d'âge.
Manipulation 53: Sélectionner les personnes ayant un certain âge minimum/maximum.
Manipulation 54: Calculer l'âge moyen par ville.

7. Jointure de RDD

Manipulation 55: Charger le fichier *Employeurs.csv* dans un RDD d'objets *Employeur* à coder.
Manipulation 56: Calculer la masse salariale des employés.
Manipulation 57: Faire la jointure des fichiers *Personnels.csv* et *Employeurs.csv*. Utiliser *join* après avoir constitué les paires *(clé_de_jointure, valeur_à_joindre)* avec *keyBy*.
Manipulation 58: Afficher le résultat puis le stocker dans un fichier local/HDFS.
Manipulation 59: Calculer le salaire maximum et minimum par ville.
Manipulation 60: Calculer le salaire moyen par ville.
Manipulation 61: Calculer le salaire minimum, maximum et moyen par décennie (années 80, 90, 2000, etc.) de naissance.

Quelques éléments de syntaxe du langage Scala

Déclarer une variable :

var <nom-de-variable-*mutable*> : <type-optionnel> = <valeur-initiale> ;

val <nom-de-variable-*constante*> : <type-optionnel> = <valeur-initiale> ;

Déclarer une fonction :

def <nom-de-fonction> (<nom-argument> : <type-argument>) : <type-de-retour-optionnel> =
    <bloc-d'instructions> ;

*Déclarer une classe **case** :*

case class <nom-de-la-classe> (<nom-attribut>:<type-attribut>, …) // attributs de type **val**

val o1 = <nom-de-la-classe> (<valeur-initiale-attribut1>, <valeur-initiale-attribut2>, …)

Déclarer une classe :

class <nom-de-la-classe> (**var/val** <nom-attribut-constructeur>:<type-attribut>, …) **{**

  var/val <nom-attribut-supplémentaire> = <valeur-initiale>

  def <nom-de-méthode> (<nom-argument>:<type-argument>) : <type-de-retour> =
      <bloc-d'instructions> ;

}

val o2 = **new** <nom-de-la-classe> (<valeur-initiale-attribut1>, <valeur-initiale-attribut2>, …)

charger un fichier de code Scala dans l'interpréteur de commandes :

:load <chemin-vers-le-fichier>/<nom-fichier-scala>