eBook Gratuit

APPRENEZ apache-spark

eBook gratuit non affilié créé à partir des **contributeurs de Stack Overflow.**

#apache-spark

Table des matières

À propos

You can share this PDF with anyone you feel could benefit from it, downloaded the latest version from: apache-spark

It is an unofficial and free apache-spark ebook created for educational purposes. All the content is extracted from Stack Overflow Documentation, which is written by many hardworking individuals at Stack Overflow. It is neither affiliated with Stack Overflow nor official apache-spark.

Use the content presented in this book at your own risk; it is not guaranteed to be correct nor accurate, please send your feedback and corrections to info@zzzprojects.com

Chapitre 1: Démarrer avec apache-spark

Remarques

Apache Spark est une infrastructure de traitement de données volumineuses open source construite autour de la vitesse, de la facilité d'utilisation et d'analyses sophistiquées. Un développeur doit l'utiliser lorsqu'il traite une grande quantité de données, ce qui implique généralement des limitations de mémoire et / ou un temps de traitement prohibitif.

Il devrait également mentionner tous les grands sujets dans apache-spark, et établir un lien avec les sujets connexes. La documentation de apache-spark étant nouvelle, vous devrez peut-être créer des versions initiales de ces rubriques connexes.

Versions

Version	Date de sortie
2.2.0	2017-07-11
2.1.1	2017-05-02
2.1.0	2016-12-28
2.0.1	2016-10-03
2.0.0	2016-07-26
1.6.0	2016-01-04
1.5.0	2015-09-09
1.4.0	2015-06-11
1.3.0	2015-03-13
1.2.0	2014-12-18
1.1.0	2014-09-11
1.0.0	2014-05-30
0.9.0	2014-02-02
0.8.0	2013-09-25
0.7.0	2013-02-27

Version	Date de sortie
0.6.0	2012-10-15

Examples

introduction

Prototype :

> agrégat (valeur zéro, seqOp, combOp)

Description :

`aggregate()` vous permet de prendre un RDD et de générer une valeur unique d'un type différent de ce qui était stocké dans le RDD d'origine.

Paramètres :

1. `zeroValue` : La valeur d'initialisation, pour votre résultat, au format souhaité.
2. `seqOp` : opération que vous souhaitez appliquer aux enregistrements RDD. Fonctionne une fois pour chaque enregistrement dans une partition.
3. `combOp` : définit comment les objets résultants (un pour chaque partition) sont combinés.

Exemple :

> Calculez la somme d'une liste et la longueur de cette liste. Renvoie le résultat dans une paire de `(sum, length)` .

Dans un shell Spark, créez une liste de 4 éléments, avec 2 *partitions* :

```
listRDD = sc.parallelize([1,2,3,4], 2)
```

Ensuite, définissez *seqOp* :

```
seqOp = (lambda local_result, list_element: (local_result[0] + list_element, local_result[1] +
1) )
```

Ensuite, définissez *combOp* :

```
combOp = (lambda some_local_result, another_local_result: (some_local_result[0] +
another_local_result[0], some_local_result[1] + another_local_result[1]) )
```

Puis agrégé:

```
listRDD.aggregate( (0, 0), seqOp, combOp)
Out[8]: (10, 4)
```

La première partition a la sous-liste [1, 2]. Ceci applique le seqOp à chaque élément de cette liste,

ce qui produit un résultat local - Une paire de `(sum, length)` qui reflétera le résultat localement, uniquement dans cette première partition.

`local_result` obtient initialisé à la `zeroValue` paramètre `aggregate()` a été fourni avec. Par exemple, (0, 0) et `list_element` sont le premier élément de la liste:

```
0 + 1 = 1
0 + 1 = 1
```

Le résultat local est (1, 1), ce qui signifie que la somme est 1 et la longueur 1 pour la 1ère partition après avoir traité *uniquement* le premier élément. `local_result` est mis à jour de (0, 0) à (1, 1).

```
1 + 2 = 3
1 + 1 = 2
```

Le résultat local est maintenant (3, 2), qui sera le résultat final de la 1ère partition, car ils ne sont pas d'autres éléments dans la sous-liste de la 1ère partition. Faire la même chose pour les retours de 2ème partition (7, 2).

Appliquez combOp à chaque résultat local pour former le résultat final global:

```
(3,2) + (7,2) = (10, 4)
```

Exemple décrit dans 'figure':

```
            (0, 0) <-- zeroValue

[1, 2]                  [3, 4]

0 + 1 = 1               0 + 3 = 3
0 + 1 = 1               0 + 1 = 1

1 + 2 = 3               3 + 4 = 7
1 + 1 = 2               1 + 1 = 2
    |                       |
    v                       v
  (3, 2)                  (7, 2)
     \                     /
      \                   /
       \                 /
        \               /
         \             /
          \           /
           \         /
           ----------
           |  combOp  |
           ----------
                |
                v
             (10, 4)
```

Transformation vs Action

Spark utilise une **évaluation paresseuse** ; cela signifie qu'il ne fera aucun travail, à moins que ce soit vraiment nécessaire. Cette approche nous permet d'éviter l'utilisation inutile de la mémoire, ce qui nous permet de travailler avec des données volumineuses.

Une *transformation* est évaluée paresseuse et le travail réel se produit lorsqu'une *action* se produit.

Exemple:

```
In [1]: lines = sc.textFile(file)        // will run instantly, regardless file's size
In [2]: errors = lines.filter(lambda line: line.startsWith("error")) // run instantly
In [3]: errorCount = errors.count()    // an action occurred, let the party start!
Out[3]: 0                              // no line with 'error', in this example
```

Donc, dans `[1]` nous avons dit à Spark de lire un fichier dans un RDD, nommé `lines` . Spark nous a entendus et nous a dit: "Oui, je *le* ferai", mais en fait, il n'a pas *encore* lu le fichier.

Dans [2], nous filtrons les lignes du fichier, en supposant que son contenu contient des lignes avec des erreurs marquées d'une `error` au début. Nous disons donc à Spark de créer un nouveau RDD, appelé des `errors` , qui aura les éléments des `lines` RDD, qui comportaient le mot `error` au début.

Maintenant, dans `[3]` , nous demandons à Spark de *compter* les *erreurs* , c'est-à-dire de compter le nombre d'éléments que le RDD appelle des `errors` . `count()` est une **action** qui ne laisse pas le choix à Spark, mais à effectuer l'opération, afin qu'elle puisse trouver le résultat de `count()` , qui sera un entier.

Par conséquent, lorsque `[3]` est atteint, `[1]` et `[2]` seront effectivement exécutés, c'est-à-dire que lorsque nous atteignons `[3]` , alors et seulement alors:

1. le fichier va être lu dans `textFile()` (à cause de `[1]`)
2. `lines` seront `filter()` 'ed (à cause de `[2]`)
3. `count()` s'exécutera, à cause de `[3]`

Astuce de débogage: Puisque Spark ne fera pas de travail réel avant d'avoir atteint `[3]` , il est important de comprendre que si une erreur existe dans `[1]` et / ou `[2]` , elle n'apparaîtra pas tant que l'action dans `[3]` déclenche le travail de Spark. Par exemple, si vos données dans le fichier ne supportent pas le `startsWith()` j'ai utilisé, alors `[2]` sera correctement accepté par Spark et ne provoquera aucune erreur, mais quand `[3]` sera soumis et que Spark sera effectivement évalue à la fois `[1]` et `[2]` , alors et seulement alors il comprendra que quelque chose n'est pas correct avec `[2]` et produit une erreur descriptive.

En conséquence, une erreur peut être déclenchée lorsque `[3]` est exécuté, mais cela ne signifie pas que l'erreur doit se trouver dans l'instruction de `[3]` !

Notez que ni les `lines` ni les `errors` ne seront stockées en mémoire après `[3]` . Ils continueront à exister uniquement comme un ensemble d'instructions de traitement. Si plusieurs actions sont

effectuées sur l'un de ces RDD, spark lit et filtre les données plusieurs fois. Pour éviter les opérations de duplication lors de l'exécution de plusieurs actions sur un seul RDD, il est souvent utile de stocker des données en mémoire à l'aide du `cache` .

Vous pouvez voir plus de transformations / actions dans les documents Spark .

Vérifier la version Spark

En `spark-shell` :

```
sc.version
```

Généralement dans un programme:

```
SparkContext.version
```

En utilisant `spark-submit` :

```
 spark-submit --version
```

Lire Démarrer avec apache-spark en ligne: https://riptutorial.com/fr/apache-spark/topic/833/demarrer-avec-apache-spark

Chapitre 2: Appeler des emplois scala à partir de pyspark

Introduction

Ce document vous montrera comment appeler des travaux Scala depuis une application pyspark.

Cette approche peut être utile lorsque l'API Python manque certaines fonctionnalités existantes de l'API Scala ou même pour résoudre les problèmes de performances liés à l'utilisation de python.

Dans certains cas d'utilisation, l'utilisation de Python est inévitable, par exemple, vous construisez des modèles avec `scikit-learn` .

Examples

Créer une fonction Scala qui reçoit un RDD python

Créer une fonction Scala qui reçoit un RDD python est facile. Ce que vous devez construire est une fonction qui obtient un JavaRDD [Any]

```
import org.apache.spark.api.java.JavaRDD

def doSomethingByPythonRDD(rdd :JavaRDD[Any]) = {
    //do something
    rdd.map { x => ??? }
}
```

Sérialiser et envoyer python RDD au code scala

Cette partie du développement, vous devez sérialiser le RDD python à la JVM. Ce processus utilise le développement principal de Spark pour appeler la fonction jar.

```
from pyspark.serializers import PickleSerializer, AutoBatchedSerializer


rdd = sc.parallelize(range(10000))
reserialized_rdd = rdd._reserialize(AutoBatchedSerializer(PickleSerializer()))
rdd_java = rdd.ctx._jvm.SerDe.pythonToJava(rdd._jrdd, True)

_jvm = sc._jvm #This will call the py4j gateway to the JVM.
_jvm.myclass.apps.etc.doSomethingByPythonRDD(rdd_java)
```

Comment appeler spark-submit

Pour appeler ce code, vous devez créer le pot de votre code scala. Que vous devez appeler votre étincelle comme ceci:

```
spark-submit --master yarn-client --jars ./my-scala-code.jar --driver-class-path ./my-scala-code.jar main.py
```

Cela vous permettra d'appeler n'importe quel code scala dont vous avez besoin dans vos tâches pySpark

Lire Appeler des emplois scala à partir de pyspark en ligne: https://riptutorial.com/fr/apache-spark/topic/9180/appeler-des-emplois-scala-a-partir-de-pyspark

Chapitre 3: Comment poser la question liée à Apache Spark?

Introduction

L'objectif de cette rubrique est de documenter les meilleures pratiques lorsque vous posez des questions sur Apache Spark.

Examples

Détails de l'environnement:

Lorsque vous posez des questions sur Apache Spark, veuillez inclure les informations suivantes.

- Version Apache Spark utilisée par le client et déploiement Spark, le cas échéant. Pour les questions liées à l'API, les questions majeures (1.6, 2.0, 2.1, etc.) sont généralement suffisantes, car les questions concernant d'éventuels bogues utilisent toujours des informations complètes sur la version.
- Version Scala utilisée pour construire des binaires Spark.
- Version JDK (version `java -version`).
- Si vous utilisez le langage invité (Python, R), veuillez fournir des informations sur la version linguistique. Dans Python, utilisez des balises: python-2.x , python-3.x ou des balises plus spécifiques pour distinguer les variantes de langage.
- Créez la définition (`build.sbt` , `pom.xml`), le cas échéant, ou les versions de dépendance externes (Python, R), le cas échéant.
- Gestionnaire de cluster (`local[n]` , Spark autonome, Yarn, Mesos), mode (`client` , `cluster`) et autres options d'envoi, le cas échéant.

Exemple de données et code

Exemple de données

Essayez de fournir un exemple minimal de données d'entrée dans un format directement utilisable par les réponses, sans analyse fastidieuse et longue, par exemple un fichier d'entrée ou une collection locale avec tout le code requis pour créer des structures de données distribuées.

Le cas échéant, incluez toujours des informations sur le type:

- Dans les API basées sur RDD, utilisez les annotations de type si nécessaire.
- Dans l'API basée sur DataFrame, fournissez des informations de schéma sous la forme d'un `StrucType` ou d'une sortie de `Dataset.printSchema` .

La sortie de `Dataset.show` ou `print` peut avoir un bon aspect mais ne nous dit rien sur les types sous-jacents.

Si un problème particulier se produit uniquement à l'échelle, utilisez des générateurs de données aléatoires (Spark fournit des utilitaires utiles dans `org.apache.spark.mllib.random.RandomRDDs` et `org.apache.spark.graphx.util.GraphGenerators`

Code

Veuillez utiliser des annotations de type lorsque cela est possible. Bien que votre compilateur puisse facilement suivre les types, il n'est pas si facile pour les simples mortels. Par exemple:

```
val lines: RDD[String] = rdd.map(someFunction)
```

ou

```
def f(x: String): Int = ???
```

sont meilleurs que:

```
val lines = rdd.map(someFunction)
```

et

```
def f(x: String) = ???
```

respectivement.

Informations de diagnostic

Questions de débogage

Lorsque la question est liée au débogage d'une exception spécifique, fournissez toujours une trace appropriée. Bien qu'il soit conseillé de supprimer les sorties en double (provenant de différents exécuteurs ou tentatives), ne coupez pas les traces vers une seule ligne ou une seule classe d'exception.

Questions de performance

En fonction du contexte, essayez de fournir des détails tels que:

- `RDD.debugString` / `Dataset.explain` .
- Sortie de l'interface utilisateur Spark avec diagramme DAG si applicable dans un cas particulier.
- Messages de journal pertinents.
- Informations de diagnostic collectées par des outils externes (Ganglia, VisualVM).

Avant de demander

- Recherche Stack Overflow pour les questions en double. Il existe une classe commune de problèmes qui ont déjà été largement documentés.
- Lire Comment puis-je poser une bonne question? .
- Lire Quels sujets puis-je poser ici?
- Ressources communautaires Apache Spark

Lire Comment poser la question liée à Apache Spark? en ligne: https://riptutorial.com/fr/apache-spark/topic/8815/comment-poser-la-question-liee-a-apache-spark-

Chapitre 4: Configuration: Apache Spark SQL

Introduction

Dans cette rubrique, les utilisateurs Spark peuvent trouver différentes configurations de Spark SQL, composant le plus utilisé du framework Apache Spark.

Examples

Contrôle des partitions de lecture aléatoire Spark SQL

Dans Apache Spark, tout en effectuant des opérations de `cogroup` aléatoire telles que la `join` et le `cogroup` nombreuses données sont transférées sur le réseau. Maintenant, pour contrôler le nombre de partitions sur lesquelles se produit la lecture aléatoire, vous pouvez les contrôler à l'aide de configurations données dans Spark SQL. Cette configuration est la suivante:

```
spark.sql.shuffle.partitions
```

En utilisant cette configuration, nous pouvons contrôler le nombre de partitions des opérations de lecture aléatoire. Par défaut, sa valeur est `200` . Mais 200 partitions n'a aucun sens si nous avons des fichiers de quelques Go. Nous devrions donc les modifier en fonction de la quantité de données à traiter via Spark SQL. Comme suit:

Dans ce scénario, nous avons deux tables à joindre `employee` et `department` . Les deux tables ne contiennent que peu d'enregistrements, mais nous devons les joindre pour connaître le service de chaque employé. Donc, nous les rejoignons en utilisant Spark DataFrames comme ceci:

```
val conf = new SparkConf().setAppName("sample").setMaster("local")
val sc = new SparkContext(conf)

val employee = sc.parallelize(List("Bob", "Alice")).toDF("name")
val department = sc.parallelize(List(("Bob", "Accounts"), ("Alice", "Sales"))).toDF("name",
"department")

employeeDF.join(departmentDF, "employeeName").show()
```

Maintenant, le nombre de partitions créées lors de la jointure est de 200 par défaut, ce qui est bien sûr trop pour cette quantité de données.

Changeons donc cette valeur pour réduire le nombre d'opérations de lecture aléatoire.

```
val conf = new
SparkConf().setAppName("sample").setMaster("local").set("spark.sql.shuffle.partitions", 2)
val sc = new SparkContext(conf)

val employee = sc.parallelize(List("Bob", "Alice")).toDF("name")
val department = sc.parallelize(List(("Bob", "Accounts"), ("Alice", "Sales"))).toDF("name",
"department")
```

```
employeeDF.join(departmentDF, "employeeName").show()
```

Maintenant, le nombre de partitions shuffle est réduit à 2, ce qui réduit non seulement le nombre d'opérations de mélange mais réduit également le temps nécessaire pour joindre les DataFrames de `0.878505 s` à `0.077847 s` .

Configurez donc toujours le nombre de partitions pour les opérations de lecture aléatoire en fonction des données en cours de traitement.

Lire Configuration: Apache Spark SQL en ligne: https://riptutorial.com/fr/apache-spark/topic/8169/configuration--apache-spark-sql

Chapitre 5: Fichiers texte et opérations à Scala

Introduction

Lire des fichiers texte et effectuer des opérations sur eux.

Examples

Exemple d'utilisation

Lire le fichier texte du chemin:

```
val sc: org.apache.spark.SparkContext = ???
sc.textFile(path="/path/to/input/file")
```

Lire des fichiers en utilisant des caractères génériques:

```
sc.textFile(path="/path/to/*/*")
```

Lire des fichiers en spécifiant un nombre minimum de partitions:

```
sc.textFile(path="/path/to/input/file", minPartitions=3)
```

Joindre deux fichiers lus avec textFile ()

Se joint à Spark:

- Lire le texteFichier 1

  ```
  val txt1=sc.textFile(path="/path/to/input/file1")
  ```

  Par exemple:

  ```
  A B
  1 2
  3 4
  ```

- Lire le texteFichier 2

  ```
  val txt2=sc.textFile(path="/path/to/input/file2")
  ```

  Par exemple:

```
A C
1 5
3 6
```

- Joindre et imprimer le résultat.

```
txt1.join(txt2).foreach(println)
```

Par exemple:

```
A B C
1 2 5
3 4 6
```

La jointure ci-dessus est basée sur la première colonne.

Lire Fichiers texte et opérations à Scala en ligne: https://riptutorial.com/fr/apache-spark/topic/1620/fichiers-texte-et-operations-a-scala

Chapitre 6: Fonctions de fenêtre dans Spark SQL

Examples

introduction

Les fonctions de fenêtre sont utilisées pour effectuer des opérations (généralement des agrégations) sur un ensemble de lignes appelées collectivement comme fenêtre. Les fonctions de fenêtre fonctionnent dans Spark 1.4 ou version ultérieure. Les fonctions de fenêtre fournissent plus d'opérations que les fonctions intégrées ou UDF, telles que substr ou round (largement utilisées avant Spark 1.4). Les fonctions de fenêtre permettent aux utilisateurs de Spark SQL de calculer des résultats tels que le rang d'une ligne donnée ou une moyenne mobile sur une plage de lignes en entrée. Ils améliorent considérablement l'expressivité des API SQL et DataFrame de Spark.

À la base, une fonction de fenêtre calcule une valeur de retour pour chaque ligne d'entrée d'un tableau en fonction d'un groupe de lignes, appelé Frame. Chaque ligne d'entrée peut être associée à une image unique. Cette caractéristique des fonctions de fenêtre les rend plus puissantes que d'autres fonctions. Les types de fonctions de fenêtre sont

- Fonctions de classement
- Fonctions analytiques
- Fonctions d'agrégat

Pour utiliser les fonctions de fenêtre, les utilisateurs doivent indiquer qu'une fonction est utilisée comme fonction de fenêtre

- Ajouter une clause `OVER` après une fonction prise en charge dans SQL, par exemple `avg(revenue) OVER (...);` ou
- Appel de la méthode over sur une fonction prise en charge dans l'API DataFrame, par exemple `rank().over(...)` Over `rank().over(...)` .

Cette documentation vise à démontrer certaines de ces fonctions avec un exemple. On suppose que le lecteur a quelques connaissances sur les opérations de base sur Spark DataFrame, telles que: ajouter une nouvelle colonne, renommer une colonne, etc.

Lecture d'un échantillon de données:

```
val sampleData = Seq(
("bob","Developer",125000),("mark","Developer",108000),("carl","Tester",70000),("peter","Developer",185
```

Liste des instructions d'importation requises:

```
import org.apache.spark.sql.expressions.Window
import org.apache.spark.sql.functions._
```

La première instruction importe la `Window Specification` . Une spécification de fenêtre contient des conditions / spécifications indiquant quelles lignes doivent être incluses dans la fenêtre.

```
scala> sampleData.show
+------+---------+------+
|  Name|     Role|Salary|
+------+---------+------+
|   bob|Developer|125000|
|  mark|Developer|108000|
|  carl|   Tester| 70000|
| peter|Developer|185000|
|   jon|   Tester| 65000|
| roman|   Tester| 82000|
| simon|Developer| 98000|
|  eric|Developer|144000|
|carlos|   Tester| 75000|
| henry|Developer|110000|
+------+---------+------+
```

Moyenne mobile

Pour calculer la moyenne mobile du salaire des employeurs en fonction de leur rôle:

```
val movAvg = sampleData.withColumn("movingAverage", avg(sampleData("Salary"))
             .over( Window.partitionBy("Role").rowsBetween(-1,1)) )
```

- `withColumn()` crée une nouvelle colonne nommée `movingAverage` , effectuant une `average` sur la colonne `Salary`
- `over()` est utilisé pour définir la spécification de la fenêtre.
- `partitionBy()` partitionne les données sur la colonne `Role`
- `rowsBetween(start, end)` Cette fonction définit les lignes à inclure dans la fenêtre. Les paramètres (`start` et `end`) prennent des entrées numériques, `0` représente la ligne actuelle, `-1` la ligne précédente, `1` la ligne suivante, etc. La fonction inclut toutes les lignes entre `start` et `end` . Ainsi, dans cet exemple, trois lignes (-1,0,1) sont incluses dans la fenêtre.

```
    scala> movAvg.show
+------+---------+------+------------------+
|  Name|     Role|Salary|     movingAverage|
+------+---------+------+------------------+
|   bob|Developer|125000|          116500.0|
|  mark|Developer|108000|139333.33333333334|
| peter|Developer|185000|130333.33333333333|
| simon|Developer| 98000|142333.33333333334|
|  eric|Developer|144000|117333.33333333333|
| henry|Developer|110000|          127000.0|
|  carl|   Tester| 70000|           67500.0|
|   jon|   Tester| 65000| 72333.33333333333|
| roman|   Tester| 82000|           74000.0|
|carlos|   Tester| 75000|           78500.0|
+------+---------+------+------------------+
```

Spark ignore automatiquement les lignes précédentes et suivantes, si la ligne en cours est respectivement la première et la dernière ligne.

Dans l'exemple ci-dessus, movingAverage de la première ligne est la moyenne de la ligne actuelle et de la ligne suivante uniquement, car la ligne précédente n'existe pas. De même, la dernière ligne de la partition (c'est-à-dire la 6ème ligne) est la moyenne de la ligne actuelle et précédente, car la ligne suivante n'existe pas.

Somme cumulée

Pour calculer la moyenne mobile du salaire des employeurs en fonction de leur rôle:

```
val cumSum = sampleData.withColumn("cumulativeSum", sum(sampleData("Salary"))
             .over( Window.partitionBy("Role").orderBy("Salary")))
```

- `orderBy()` trie la colonne de salaire et calcule la somme cumulée.

```
scala> cumSum.show
+------+---------+------+-------------+
|  Name|     Role|Salary|cumulativeSum|
+------+---------+------+-------------+
| simon|Developer| 98000|        98000|
|  mark|Developer|108000|       206000|
| henry|Developer|110000|       316000|
|   bob|Developer|125000|       441000|
|  eric|Developer|144000|       585000|
| peter|Developer|185000|       770000|
|   jon|   Tester| 65000|        65000|
|  carl|   Tester| 70000|       135000|
|carlos|   Tester| 75000|       210000|
| roman|   Tester| 82000|       292000|
+------+---------+------+-------------+
```

Fonctions de fenêtre - Tri, Lead, Lag, Rank, Analyse des tendances

Cette rubrique montre comment utiliser des fonctions telles que withColumn, lead, lag, Level, etc. avec Spark. Spark dataframe est une couche abstraite SQL sur les fonctionnalités de base. Cela permet à l'utilisateur d'écrire du code SQL sur des données distribuées. Spark SQL prend en charge les formats de fichiers hétérogènes, notamment JSON, XML, CSV, TSV, etc.

Dans ce blog, nous avons un aperçu rapide de la façon d'utiliser SQL et des dataframes d'étincelles pour les cas d'usage courants dans SQL world. Le fichier a quatre champs, employeeID, employeeName, salaire, salaireDate

```
1,John,1000,01/01/2016
1,John,2000,02/01/2016
1,John,1000,03/01/2016
1,John,2000,04/01/2016
1,John,3000,05/01/2016
1,John,1000,06/01/2016
```

Enregistrez ce fichier sous le nom emp.dat. Dans un premier temps, nous allons créer une image

de référence à l'aide du paquetage CSV, spark, à partir des databricks.

```
val sqlCont = new HiveContext(sc)
//Define a schema for file
val schema = StructType(Array(StructField("EmpId", IntegerType, false),
          StructField("EmpName", StringType, false),
          StructField("Salary", DoubleType, false),
          StructField("SalaryDate", DateType, false)))
//Apply Shema and read data to a dataframe
val myDF = sqlCont.read.format("com.databricks.spark.csv")
          .option("header", "false")
          .option("dateFormat", "MM/dd/yyyy")
          .schema(schema)
          .load("src/resources/data/employee_salary.dat")
//Show dataframe
myDF.show()
```

myDF est le dataframe utilisé dans les exercices restants. Comme myDF est utilisé de manière répétée, il est recommandé de le persister pour qu'il ne soit pas nécessaire de le réévaluer.

```
 myDF.persist()
```

Sortie de dataframe show

```
+-----+-------+------+----------+
|EmpId|EmpName|Salary|SalaryDate|
+-----+-------+------+----------+
| 1| John|1000.0|2016-01-01|
| 1| John|2000.0|2016-02-01|
| 1| John|1000.0|2016-03-01|
| 1| John|2000.0|2016-04-01|
| 1| John|3000.0|2016-05-01|
| 1| John|1000.0|2016-06-01|
+-----+-------+------+----------+
```

Ajouter une nouvelle colonne à dataframe

Comme les diagrammes d'étincelles sont immuables, l'ajout d'une nouvelle colonne créera un nouveau cadre de données avec une colonne ajoutée. Pour ajouter une colonne, utilisez withColumn (columnName, Transformation). Dans l'exemple ci-dessous, la colonne empName est formatée en majuscule.

```
withColumn(columnName,transformation)
myDF.withColumn("FormatedName", upper(col("EmpName"))).show()


+-----+-------+------+----------+------------+
|EmpId|EmpName|Salary|SalaryDate|FormatedName|
+-----+-------+------+----------+------------+
| 1| John|1000.0|2016-01-01| JOHN|
| 1| John|2000.0|2016-02-01| JOHN|
| 1| John|1000.0|2016-03-01| JOHN|
| 1| John|2000.0|2016-04-01| JOHN|
| 1| John|3000.0|2016-05-01| JOHN|
| 1| John|1000.0|2016-06-01| JOHN|
```

```
+-----+-------+------+----------+------------+
```

Trier les données en fonction d'une colonne

```
val sortedDf = myDF.sort(myDF.col("Salary"))
sortedDf.show()


+-----+-------+------+----------+
|EmpId|EmpName|Salary|SalaryDate|
+-----+-------+------+----------+
| 1| John|1000.0|2016-03-01|
| 1| John|1000.0|2016-06-01|
| 1| John|1000.0|2016-01-01|
| 1| John|2000.0|2016-02-01|
| 1| John|2000.0|2016-04-01|
| 1| John|3000.0|2016-05-01|
+-----+-------+------+----------+
```

Trier par ordre décroissant

desc ("Salary")

```
 myDF.sort(desc("Salary")).show()


+-----+-------+------+----------+
|EmpId|EmpName|Salary|SalaryDate|
+-----+-------+------+----------+
| 1| John|3000.0|2016-05-01|
| 1| John|2000.0|2016-02-01|
| 1| John|2000.0|2016-04-01|
| 1| John|1000.0|2016-06-01|
| 1| John|1000.0|2016-01-01|
| 1| John|1000.0|2016-03-01|
+-----+-------+------+----------+
```

Obtenir et utiliser la ligne précédente (Lag)

LAG est une fonction en SQL qui permet d'accéder aux valeurs de ligne précédentes dans la ligne en cours. Ceci est utile lorsque nous avons des cas d'utilisation comme la comparaison avec la valeur précédente. LAG in data spark est disponible dans les fonctions Window

```
lag(Column e, int offset)
Window function: returns the value that is offset rows before the current row, and null if
there is less than offset rows before the current row.


import org.apache.spark.sql.expressions.Window
//order by Salary Date to get previous salary.
//For first row we will get NULL
val window = Window.orderBy("SalaryDate")
//use lag to get previous row value for salary, 1 is the offset
val lagCol = lag(col("Salary"), 1).over(window)
myDF.withColumn("LagCol", lagCol).show()
```

```
+-----+-------+------+----------+------+
|EmpId|EmpName|Salary|SalaryDate|LagCol|
+-----+-------+------+----------+------+
| 1| John|1000.0|2016-01-01| null|
| 1| John|2000.0|2016-02-01|1000.0|
| 1| John|1000.0|2016-03-01|2000.0|
| 1| John|2000.0|2016-04-01|1000.0|
| 1| John|3000.0|2016-05-01|2000.0|
| 1| John|1000.0|2016-06-01|3000.0|
+-----+-------+------+----------+------+
```

Obtenir et utiliser la ligne suivante (Lead)

LEAD est une fonction en SQL qui permet d'accéder aux valeurs de ligne suivantes dans la ligne en cours. Ceci est utile lorsque nous avons des casinos comme la comparaison avec la valeur suivante. LEAD in Spark dataframes est disponible dans les fonctions Window

```
lead(Column e, int offset)
Window function: returns the value that is offset rows after the current row, and null if
there is less than offset rows after the current row.


import org.apache.spark.sql.expressions.Window
//order by Salary Date to get previous salary. F
//or first row we will get NULL
val window = Window.orderBy("SalaryDate")
//use lag to get previous row value for salary, 1 is the offset
val leadCol = lead(col("Salary"), 1).over(window)
myDF.withColumn("LeadCol", leadCol).show()
```

```
+-----+-------+------+----------+-------+
|EmpId|EmpName|Salary|SalaryDate|LeadCol|
+-----+-------+------+----------+-------+
| 1| John|1000.0|2016-01-01| 1000.0|
| 1| John|1000.0|2016-03-01| 1000.0|
| 1| John|1000.0|2016-06-01| 2000.0|
| 1| John|2000.0|2016-02-01| 2000.0|
| 1| John|2000.0|2016-04-01| 3000.0|
| 1| John|3000.0|2016-05-01| null|
+-----+-------+------+----------+-------+
```

Analyse de tendance avec fonctions de fenêtre Maintenant, mettons la fonction de fenêtre LAG à utiliser avec une analyse de tendance simple. Si le salaire est inférieur au mois précédent, nous le marquerons comme «DOWN», si le salaire a augmenté, puis «UP». Le code utilise la fonction Window pour ordonner by, puis faire simple avec WHEN.

```
   val window = Window.orderBy("SalaryDate")
    //Derive lag column for salary
    val laggingCol = lag(col("Salary"), 1).over(trend_window)
    //Use derived column LastSalary to find difference between current and previous row
    val salaryDifference = col("Salary") - col("LastSalary")
    //Calculate trend based on the difference
    //IF ELSE / CASE can be written using when.otherwise in spark
```

```
    val trend = when(col("SalaryDiff").isNull || col("SalaryDiff").===(0), "SAME")
    .when(col("SalaryDiff").>(0), "UP")
    .otherwise("DOWN")
    myDF.withColumn("LastSalary", laggingCol)
    .withColumn("SalaryDiff",salaryDifference)
   .withColumn("Trend", trend).show()

+-----+-------+------+----------+----------+----------+-----+
|EmpId|EmpName|Salary|SalaryDate|LastSalary|SalaryDiff|Trend|
+-----+-------+------+----------+----------+----------+-----+
|  1| John|1000.0|2016-01-01| null| null| SAME|
|  1| John|2000.0|2016-02-01| 1000.0| 1000.0| UP|
|  1| John|1000.0|2016-03-01| 2000.0| -1000.0| DOWN|
|  1| John|2000.0|2016-04-01| 1000.0| 1000.0| UP|
|  1| John|3000.0|2016-05-01| 2000.0| 1000.0| UP|
|  1| John|1000.0|2016-06-01| 3000.0| -2000.0| DOWN|
+-----+-------+------+----------+----------+----------+-----+
```

Lire Fonctions de fenêtre dans Spark SQL en ligne: https://riptutorial.com/fr/apache-spark/topic/3903/fonctions-de-fenetre-dans-spark-sql

Chapitre 7: Introduction aux Apache Spark DataFrames

Examples

Spark DataFrames avec JAVA

Un DataFrame est une collection distribuée de données organisées en colonnes nommées. Il est conceptuellement équivalent à une table dans une base de données relationnelle. Les DataFrames peuvent être construits à partir d'un large éventail de sources telles que: les fichiers de données structurés, les tables dans Hive, les bases de données externes ou les RDD existants.

Lire une table Oracle RDBMS dans le bloc de données spark:

```
SparkConf sparkConf = new SparkConf().setAppName("SparkConsumer");

sparkConf.registerKryoClasses(new Class<?>[]{
        Class.forName("org.apache.hadoop.io.Text"),
        Class.forName("packageName.className")
});

JavaSparkContext sparkContext=new JavaSparkContext(sparkConf);
SQLContext sqlcontext= new SQLContext(sparkContext);

Map<String, String> options = new HashMap();
options.put("driver", "oracle.jdbc.driver.OracleDriver");
options.put("url", "jdbc:oracle:thin:username/password@host:port:orcl"); //oracle url to
connect
options.put("dbtable", "DbName.tableName");
DataFrame df=sqlcontext.load("jdbc", options);
df.show(); //this will print content into tablular format
```

Nous pouvons également convertir ce bloc de données en rdd si besoin est:

```
JavaRDD<Row> rdd=df.javaRDD();
```

Créez un dataframe à partir d'un fichier:

```
public class LoadSaveTextFile {

    //static schema class
    public static class Schema implements Serializable {

        public String getTimestamp() {
            return timestamp;
        }
        public void setTimestamp(String timestamp) {
            this.timestamp = timestamp;
        }
```

```
        public String getMachId() {
            return machId;
        }
        public void setMachId(String machId) {
            this.machId = machId;
        }
        public String getSensorType() {
            return sensorType;
        }
        public void setSensorType(String sensorType) {
            this.sensorType = sensorType;
        }

        //instance variables
        private String timestamp;
        private String machId;
        private String sensorType;
    }

    public static void main(String[] args) throws ClassNotFoundException {

        SparkConf sparkConf = new SparkConf().setAppName("SparkConsumer");

        sparkConf.registerKryoClasses(new Class<?>[]{
                Class.forName("org.apache.hadoop.io.Text"),
                Class.forName("oracle.table.join.LoadSaveTextFile")
        });

        JavaSparkContext sparkContext=new JavaSparkContext(sparkConf);
        SQLContext sqlcontext= new SQLContext(sparkContext);

        //we have a file which ";" separated
        String filePath=args[0];

        JavaRDD<Schema> schemaRdd = sparkContext.textFile(filePath).map(
                new Function<String, Schema>() {
                    public Schema call(String line) throws Exception {
                        String[] tokens=line.split(";");
                        Schema schema = new Schema();
                        schema.setMachId(tokens[0]);
                        schema.setSensorType(tokens[1]);
                        schema.setTimestamp(tokens[2]);
                        return schema;
                    }
                });

        DataFrame df = sqlcontext.createDataFrame(schemaRdd, Schema.class);
        df.show();
    }
}
```

Maintenant, nous avons aussi un bloc de données d'Oracle à partir d'un fichier. De même, nous pouvons également lire un tableau de ruche. Sur les trames de données, nous pouvons récupérer n'importe quelle colonne comme nous le faisons dans les rdbms. Comme pour obtenir une valeur min pour une colonne ou une valeur maximale. Peut calculer une moyenne / moyenne pour une colonne. Certaines autres fonctions comme select, filter, agg, groupBy sont également disponibles.

Spark Dataframe expliqué

Dans Spark, un DataFrame est une collection distribuée de données organisées en colonnes nommées. Il est conceptuellement équivalent à une table dans une base de données relationnelle ou un bloc de données dans R / Python, mais avec des optimisations plus riches sous le capot. Les DataFrames peuvent être construits à partir d'un large éventail de sources telles que des fichiers de données structurés, des tables dans Hive, des bases de données externes ou des RDD existants.

Façons de créer Dataframe

```
val data= spark.read.json("path to json")
```

`val df = spark.read.format("com.databricks.spark.csv").load("test.txt")` dans le champ d'options, vous pouvez fournir en-tête, délimiteur, jeu de caractères et bien plus encore

vous pouvez également créer un Dataframe à partir d'un RDD

```
val rdd = sc.parallelize(
  Seq(
    ("first", Array(2.0, 1.0, 2.1, 5.4)),
    ("test", Array(1.5, 0.5, 0.9, 3.7)),
    ("choose", Array(8.0, 2.9, 9.1, 2.5))
  )
)

val dfWithoutSchema = spark.createDataFrame(rdd)
```

Si vous voulez créer df avec le schéma

```
def createDataFrame(rowRDD: RDD[Row], schema: StructType): DataFrame
```

Pourquoi nous avons besoin de Dataframe si Spark a fourni RDD

Un RDD est simplement un ensemble de données distribué résilient qui est davantage une boîte noire de données qui ne peut pas être optimisée car les opérations qui peuvent être effectuées sur ce dernier ne sont pas aussi limitées.

Aucun moteur d'optimisation intégré: Lorsque vous travaillez avec des données structurées, les RDD ne peuvent pas tirer parti des optimiseurs avancés de Spark, notamment de l'optimiseur de catalyseur et du moteur d'exécution Tungsten. Les développeurs doivent optimiser chaque RDD en fonction de ses attributs. Gestion des données structurées: contrairement aux Dataframe et aux jeux de données, les RDD ne déduisent pas le schéma des données ingérées et exigent que l'utilisateur le spécifie.

Les DataFrames dans Spark ont leur exécution optimisée automatiquement par un optimiseur de requête. Avant tout calcul sur un DataFrame, l'optimiseur Catalyst compile les opérations utilisées pour créer le DataFrame dans un plan physique à exécuter. Comme l'optimiseur comprend la sémantique des opérations et la structure des données, il peut prendre des décisions intelligentes

pour accélérer le calcul.

Limitation de DataFrame

Sécurité du type au moment de la compilation: l'API Dataframe ne prend pas en charge la sécurité au moment de la compilation, ce qui vous empêche de manipuler des données lorsque la structure n'est pas connue.

Lire Introduction aux Apache Spark DataFrames en ligne: https://riptutorial.com/fr/apache-spark/topic/6514/introduction-aux-apache-spark-dataframes

Chapitre 8: Joint

Remarques

Une chose à noter est vos ressources par rapport à la taille des données que vous rejoignez. C'est là que votre code Spark Join peut échouer, ce qui entraîne des erreurs de mémoire. Pour cette raison, assurez-vous de bien configurer vos travaux Spark en fonction de la taille des données. Voici un exemple de configuration pour une jointure de 1,5 million à 200 millions.

Utiliser Spark-Shell

```
spark-shell   --executor-memory 32G   --num-executors 80  --driver-memory 10g --executor-cores
10
```

Utiliser Spark Submit

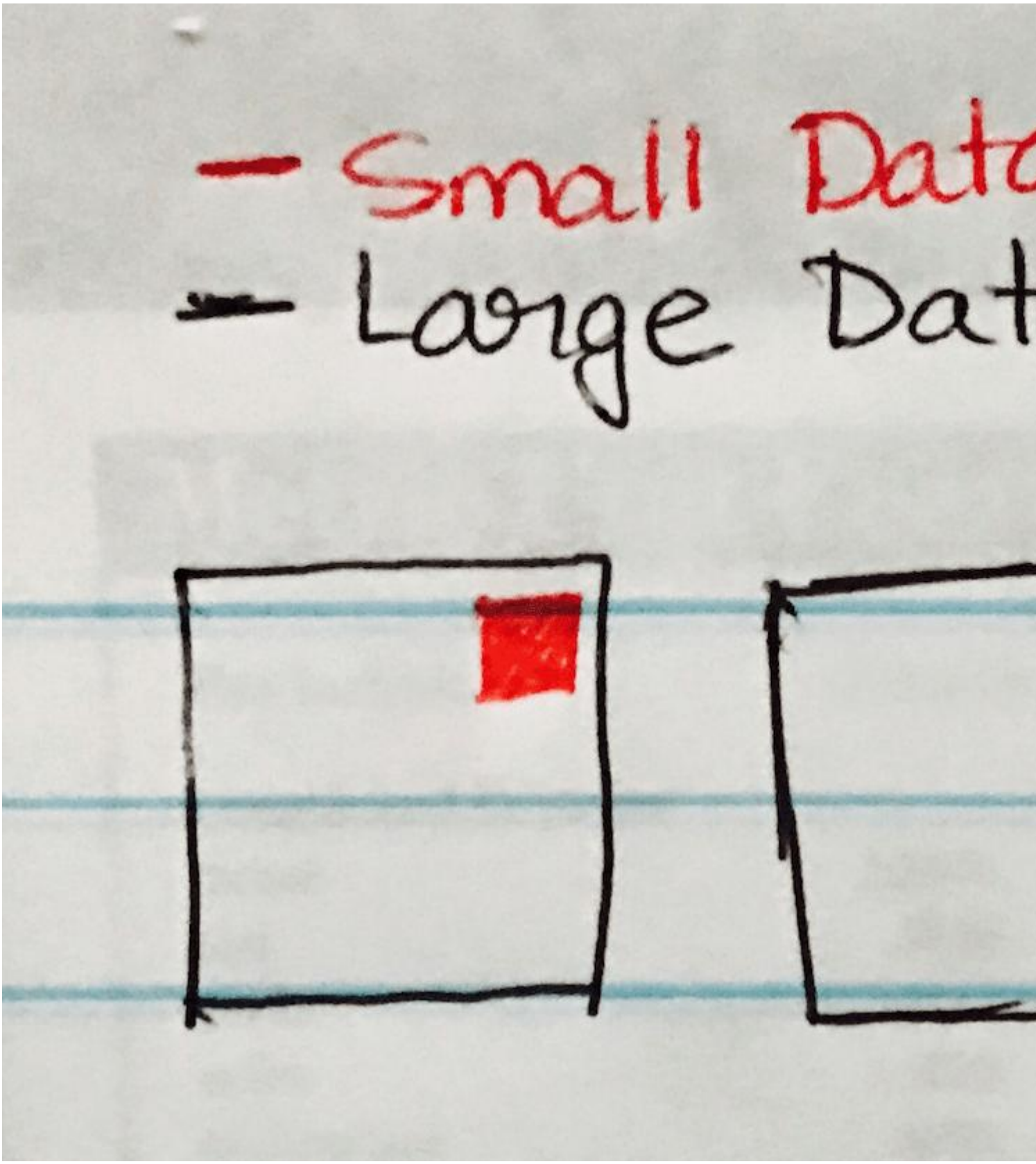
```
spark-submit   --executor-memory 32G   --num-executors 80  --driver-memory 10g --executor-
cores 10 code.jar
```

Examples

Broadcast Hash Rejoignez Spark

Une jointure de diffusion copie les petites données vers les nœuds de travail, ce qui conduit à une jointure extrêmement efficace et extrêmement rapide. Lorsque nous rejoignons deux jeux de données et que l'un des jeux de données est beaucoup plus petit que l'autre (par exemple, lorsque le petit jeu de données peut tenir dans la mémoire), nous devons utiliser une jointure par hachage de diffusion.

L'image suivante visualise une jointure par hachage de diffusion lorsque le petit ensemble de données est diffusé sur chaque partition du grand ensemble de données.



Vous trouverez ci-dessous un exemple de code que vous pouvez facilement implémenter si vous avez un scénario similaire de jointure de jeu de données volumineux et petit.

```
case class SmallData(col1: String, col2:String, col3:String, col4:Int, col5:Int)

val small = sc.textFile("/datasource")
```

```
val df1 = sm_data.map(_.split("\\|")).map(attr => SmallData(attr(0).toString,
attr(1).toString, attr(2).toString, attr(3).toInt, attr(4).toInt)).toDF()

val lg_data = sc.textFile("/datasource")

case class LargeData(col1: Int, col2: String, col3: Int)

val LargeDataFrame = lg_data.map(_.split("\\|")).map(attr => LargeData(attr(0).toInt,
attr(2).toString, attr(3).toInt)).toDF()


val joinDF = LargeDataFrame.join(broadcast(smallDataFrame), "key")
```

Lire Joint en ligne: https://riptutorial.com/fr/apache-spark/topic/7828/joint

Chapitre 9: Lanceur d'étincelles

Remarques

Spark Launcher peut aider le développeur à interroger le statut du travail d'étincelle soumis. Il y a fondamentalement huit statuts qui peuvent être interrogés. Ils sont énumérés ci-dessous avec leur signification ::

```
/** The application has not reported back yet. */
UNKNOWN(false),
/** The application has connected to the handle. */
CONNECTED(false),
/** The application has been submitted to the cluster. */
SUBMITTED(false),
/** The application is running. */
RUNNING(false),
/** The application finished with a successful status. */
FINISHED(true),
/** The application finished with a failed status. */
FAILED(true),
/** The application was killed. */
KILLED(true),
/** The Spark Submit JVM exited with a unknown status. */
LOST(true);
```

Examples

SparkLauncher

Le code ci-dessous est un exemple de base du lanceur d'étincelles. Il peut être utilisé si une application d'étincelle doit être lancée via une application.

```
val sparkLauncher = new SparkLauncher
//Set Spark properties.only Basic ones are shown here.It will be overridden if properties are
set in Main class.
sparkLauncher.setSparkHome("/path/to/SPARK_HOME")
  .setAppResource("/path/to/jar/to/be/executed")
  .setMainClass("MainClassName")
  .setMaster("MasterType like yarn or local[*]")
  .setDeployMode("set deploy mode like cluster")
  .setConf("spark.executor.cores","2")

// Lauch spark application
val sparkLauncher1 = sparkLauncher.startApplication()

//get jobId
val jobAppId = sparkLauncher1.getAppId

//Get status of job launched.THis loop will continuely show statuses like RUNNING,SUBMITED
etc.
while (true) {
    println(sparkLauncher1.getState().toString)
```

```
}
```

Lire Lanceur d'étincelles en ligne: https://riptutorial.com/fr/apache-spark/topic/8026/lanceur-d-etincelles

Chapitre 10: Le message d'erreur 'sparkR' n'est pas reconnu en tant que commande interne ou externe ou '.binsparkR' n'est pas reconnu en tant que commande interne ou externe

Introduction

Ce post s'adresse à ceux qui ont du mal à installer Spark dans leur machine Windows. Principalement en utilisant la fonction sparkR pour la session R.

Remarques

Référence utilisée par les r-blogueurs

Examples

détails pour configurer Spark pour R

Utilisez l'URL ci-dessous pour obtenir les étapes de téléchargement et d'installation: https://www.r-bloggers.com/installing-and-starting-sparkr-locally-on-windows-os-and-rstudio-2/ Ajoutez le chemin de la variable d'environnement pour votre chemin Spark / bin, spark / bin, R et Rstudio. J'ai ajouté le chemin ci-dessous (les initiales varieront selon l'endroit où vous avez téléchargé les fichiers) C: \ spark-2.0.1 C: \ spark-2.0.1 \ bin C: \ spark-2.0.1 \ sbin C: \ Program Files \ R \ R-3.3.1 \ bin \ x64 C: \ Program Files \ RStudio \ bin \ x64

Pour définir la variable d'environnement, procédez comme suit: Windows 10 et Windows 8 Dans Rechercher, recherchez et sélectionnez ensuite: Système (Panneau de configuration) Cliquez sur le lien Paramètres système avancés. Cliquez sur l'onglet Avancé sous Propriétés du système, cliquez sur Variables d'environnement. Dans la section Variables système, recherchez la variable d'environnement PATH et sélectionnez-la. Cliquez sur Modifier. Si la variable d'environnement PATH n'existe pas, cliquez sur Nouveau. Dans la fenêtre Modifier la variable système (ou la nouvelle variable système), spécifiez la valeur de la variable d'environnement PATH. Cliquez sur OK. Fermez toutes les fenêtres restantes en cliquant sur OK. Rouvrez la fenêtre d'invite de commande et exécutez sparkR (pas besoin de changer de répertoire).

Windows 7 Sur le bureau, cliquez avec le bouton droit sur l'icône Ordinateur. Choisissez Propriétés dans le menu contextuel. Cliquez sur le lien Paramètres système avancés. Cliquez sur Variables d'environnement. Dans la section Variables système, recherchez la variable d'environnement PATH et sélectionnez-la. Cliquez sur Modifier. Si la variable d'environnement

PATH n'existe pas, cliquez sur Nouveau. Dans la fenêtre Modifier la variable système (ou la nouvelle variable système), spécifiez la valeur de la variable d'environnement PATH. Cliquez sur OK. Fermez toutes les fenêtres restantes en cliquant sur OK. Rouvrez la fenêtre d'invite de commande et exécutez sparkR (pas besoin de changer de répertoire).

Lire Le message d'erreur 'sparkR' n'est pas reconnu en tant que commande interne ou externe ou '.binsparkR' n'est pas reconnu en tant que commande interne ou externe en ligne: https://riptutorial.com/fr/apache-spark/topic/9649/le-message-d-erreur--sparkr--n-est-pas-reconnu-en-tant-que-commande-interne-ou-externe-ou---binsparkr--n-est-pas-reconnu-en-tant-que-commande-interne-ou-externe

Chapitre 11: Manipulation de JSON dans Spark

Examples

Mapper JSON à une classe personnalisée avec Gson

Avec `Gson` , vous pouvez lire un ensemble de données JSON et les mapper à une classe personnalisée `MyClass` .

Comme `Gson` n'est pas sérialisable, chaque exécuteur a besoin de son propre objet `Gson` . De plus, `MyClass` doit être sérialisable pour pouvoir passer entre les exécuteurs.

Notez que le ou les fichiers proposés en tant que fichier json ne sont pas un fichier JSON standard. Chaque ligne doit contenir un objet JSON valide, autonome et séparé. En conséquence, un fichier JSON multi-lignes normal échouera le plus souvent.

```
val sc: org.apache.spark.SparkContext // An existing SparkContext

// A JSON dataset is pointed to by path.
// The path can be either a single text file or a directory storing text files.
val path = "path/to/my_class.json"
val linesRdd: RDD[String] = sc.textFile(path)

// Mapping json to MyClass
val myClassRdd: RDD[MyClass] = linesRdd.map{ l =>
    val gson = new com.google.gson.Gson()
    gson.fromJson(l, classOf[MyClass])
}
```

Si la création de l'objet `Gson` devient trop coûteuse, la méthode `mapPartitions` peut être utilisée pour l'optimiser. Avec lui, il y aura un `Gson` par partition au lieu de par ligne:

```
val myClassRdd: RDD[MyClass] = linesRdd.mapPartitions{p =>
    val gson = new com.google.gson.Gson()
    p.map(l => gson.fromJson(l, classOf[MyClass]))
}
```

Lire Manipulation de JSON dans Spark en ligne: https://riptutorial.com/fr/apache-spark/topic/2799/manipulation-de-json-dans-spark

Chapitre 12: Migration de Spark 1.6 vers Spark 2.0

Introduction

Spark 2.0 a été publié et contient de nombreuses améliorations et nouvelles fonctionnalités. Si vous utilisez Spark 1.6 et que vous souhaitez maintenant mettre à niveau votre application pour utiliser Spark 2.0, vous devez prendre en compte certaines modifications de l'API. Vous trouverez ci-dessous certaines des modifications à apporter au code.

Examples

Mettre à jour le fichier build.sbt

Mettre à jour build.sbt avec:

```
scalaVersion := "2.11.8" // Make sure to have installed Scala 11
sparkVersion := "2.0.0"  // Make sure to have installed Spark 2.0
```

Notez que lors de la compilation avec le `sbt package` , le `.jar` sera maintenant créé dans `target/scala-2.11/` , et le nom du `.jar` sera également modifié. La commande `spark-submit` doit également être mise à jour.

Mettre à jour les bibliothèques vectorielles ML

ML `Transformers` génère désormais `org.apache.spark.ml.linalg.VectorUDT` au lieu de `org.apache.spark.mllib.linalg.VectorUDT` .

Ils sont également associés localement aux sous-classes de `org.apache.spark.ml.linalg.Vector` . Celles-ci ne sont pas compatibles avec l'ancienne API MLLib qui évolue vers la dépréciation de Spark 2.0.0.

```
//import org.apache.spark.mllib.linalg.{Vector, Vectors} // Depreciated in Spark 2.0
import org.apache.spark.ml.linalg.Vector // Use instead
```

Lire Migration de Spark 1.6 vers Spark 2.0 en ligne: https://riptutorial.com/fr/apache-spark/topic/6506/migration-de-spark-1-6-vers-spark-2-0

Chapitre 13: Mode client et mode cluster

Examples

Le mode Spark Client et Cluster expliqué

Essayons de regarder les différences entre le mode client et le mode cluster de Spark.

Client : lors de l'exécution de Spark en mode client, le programme SparkContext et Driver s'exécute en externe sur le cluster; par exemple, à partir de votre ordinateur portable. Le mode local est uniquement utilisé lorsque vous ne souhaitez pas utiliser un cluster et que vous souhaitez tout exécuter sur un seul ordinateur. Ainsi, l'application pilote et l'application Spark sont toutes deux sur le même ordinateur que l'utilisateur. Le pilote s'exécute sur un serveur dédié (nœud maître) dans un processus dédié. Cela signifie qu'il dispose de toutes les ressources disponibles pour exécuter le travail. Étant donné que le nœud maître possède des ressources dédiées, vous n'avez pas besoin de dépenser des ressources de travail pour le programme du pilote. Si le processus du pilote meurt, vous avez besoin d'un système de surveillance externe pour réinitialiser son exécution.

Cluster: le pilote s'exécute sur l'un des noeuds Worker du cluster. Il s'exécute en tant que processus autonome dédié dans Worker. Lorsque vous travaillez en mode cluster, tous les fichiers JAR liés à l'exécution de votre application doivent être accessibles au public pour tous les utilisateurs. Cela signifie que vous pouvez les placer manuellement dans un emplacement partagé ou dans un dossier pour chacun des travailleurs. Chaque application dispose de ses propres processus d'exécution, qui restent en place pendant toute la durée de l'application et exécutent des tâches dans plusieurs threads. Cela présente l'avantage d'isoler les applications les unes des autres, tant du côté de la planification (chaque pilote planifie ses propres tâches) que du côté de l'exécuteur (les tâches provenant de différentes applications s'exécutent dans des machines virtuelles Java différentes)

Types de gestionnaire de cluster

Apache Mesos - un gestionnaire de cluster général qui peut également exécuter Hadoop MapReduce et les applications de service. Hadoop YARN - le gestionnaire de ressources dans Hadoop.
Kubernetes-infrastructure-centrée sur les conteneurs est encore expérimentale.

Lire Mode client et mode cluster en ligne: https://riptutorial.com/fr/apache-spark/topic/10808/mode-client-et-mode-cluster

Chapitre 14: Opérations avec état dans Spark Streaming

Examples

PairDStreamFunctions.updateStateByKey

`updateState` by key peut être utilisé pour créer un `DStream` état basé sur les données à venir. Il nécessite une fonction:

```
object UpdateStateFunctions {
  def updateState(current: Seq[Double], previous: Option[StatCounter]) = {
    previous.map(s => s.merge(current)).orElse(Some(StatCounter(current)))
  }
}
```

qui prend une séquence des valeurs `current` , une `Option` de l'état précédent et renvoie une `Option` de l'état mis à jour. Mettre tout cela ensemble:

```
import org.apache.spark._
import org.apache.spark.streaming.dstream.DStream
import scala.collection.mutable.Queue
import org.apache.spark.util.StatCounter
import org.apache.spark.streaming._

object UpdateStateByKeyApp {
  def main(args: Array[String]) {

    val sc = new SparkContext("local", "updateStateByKey", new SparkConf())
    val ssc = new StreamingContext(sc, Seconds(10))
    ssc.checkpoint("/tmp/chk")

    val queue = Queue(
      sc.parallelize(Seq(("foo", 5.0), ("bar", 1.0))),
      sc.parallelize(Seq(("foo", 1.0), ("foo", 99.0))),
      sc.parallelize(Seq(("bar", 22.0), ("foo", 1.0))),
      sc.emptyRDD[(String, Double)],
      sc.emptyRDD[(String, Double)],
      sc.emptyRDD[(String, Double)],
      sc.parallelize(Seq(("foo", 1.0), ("bar", 1.0)))
    )

    val inputStream: DStream[(String, Double)] = ssc.queueStream(queue)

    inputStream.updateStateByKey(UpdateStateFunctions.updateState _).print()

    ssc.start()
    ssc.awaitTermination()
    ssc.stop()
  }
}
```

PairDStreamFunctions.mapWithState

`mapWithState` , similaire à `updateState` , peut être utilisé pour créer un DStream avec état basé sur les données à venir. Il nécessite `StateSpec` :

```
import org.apache.spark.streaming._

object StatefulStats {
  val state = StateSpec.function(
    (key: String, current: Option[Double], state: State[StatCounter]) => {
      (current, state.getOption) match {
        case (Some(x), Some(cnt)) => state.update(cnt.merge(x))
        case (Some(x), None) => state.update(StatCounter(x))
        case (None, None) => state.update(StatCounter())
        case _ =>
      }

      (key, state.get)
    }
  )
}
```

qui prend clé `key` , `value` actuelle et `State` accumulé et retourne un nouvel état. Mettre tout cela ensemble:

```
import org.apache.spark._
import org.apache.spark.streaming.dstream.DStream
import scala.collection.mutable.Queue
import org.apache.spark.util.StatCounter

object MapStateByKeyApp {
  def main(args: Array[String]) {
    val sc = new SparkContext("local", "mapWithState", new SparkConf())

    val ssc = new StreamingContext(sc, Seconds(10))
    ssc.checkpoint("/tmp/chk")

    val queue = Queue(
      sc.parallelize(Seq(("foo", 5.0), ("bar", 1.0))),
      sc.parallelize(Seq(("foo", 1.0), ("foo", 99.0))),
      sc.parallelize(Seq(("bar", 22.0), ("foo", 1.0))),
      sc.emptyRDD[(String, Double)],
      sc.parallelize(Seq(("foo", 1.0), ("bar", 1.0)))
    )

    val inputStream: DStream[(String, Double)] = ssc.queueStream(queue)

    inputStream.mapWithState(StatefulStats.state).print()

    ssc.start()
    ssc.awaitTermination()
    ssc.stop()
  }
}
```

Sortie enfin attendue:

```
-------------------------------------------
Time: 1469923280000 ms
-------------------------------------------
(foo,(count: 1, mean: 5.000000, stdev: 0.000000, max: 5.000000, min: 5.000000))
(bar,(count: 1, mean: 1.000000, stdev: 0.000000, max: 1.000000, min: 1.000000))

-------------------------------------------
Time: 1469923290000 ms
-------------------------------------------
(foo,(count: 3, mean: 35.000000, stdev: 45.284287, max: 99.000000, min: 1.000000))
(foo,(count: 3, mean: 35.000000, stdev: 45.284287, max: 99.000000, min: 1.000000))

-------------------------------------------
Time: 1469923300000 ms
-------------------------------------------
(bar,(count: 2, mean: 11.500000, stdev: 10.500000, max: 22.000000, min: 1.000000))
(foo,(count: 4, mean: 26.500000, stdev: 41.889736, max: 99.000000, min: 1.000000))

-------------------------------------------
Time: 1469923310000 ms
-------------------------------------------

-------------------------------------------
Time: 1469923320000 ms
-------------------------------------------
(foo,(count: 5, mean: 21.400000, stdev: 38.830916, max: 99.000000, min: 1.000000))
(bar,(count: 3, mean: 8.000000, stdev: 9.899495, max: 22.000000, min: 1.000000))
```

Lire Opérations avec état dans Spark Streaming en ligne: https://riptutorial.com/fr/apache-spark/topic/1924/operations-avec-etat-dans-spark-streaming

Chapitre 15: Partitions

Remarques

Le nombre de partitions est critique pour les performances d'une application et / ou la résiliation réussie.

Un ensemble de données distribué résilient (RDD) est l'abstraction principale de Spark. Un RDD est divisé en partitions, ce qui signifie qu'une partition fait partie du jeu de données, une tranche ou, en d'autres termes, un morceau.

Plus le nombre de partitions est élevé, plus la taille de chaque partition est petite.

Cependant, notez qu'un grand nombre de partitions met beaucoup de pression sur le système de fichiers distribué Hadoop (HDFS), qui doit conserver une quantité importante de métadonnées.

Le nombre de partitions est lié à l'utilisation de la mémoire et un problème de memoryOverhead peut être lié à ce numéro (expérience personnelle).

Un **piège courant** pour les nouveaux utilisateurs est de transformer leur RDD en RDD avec une seule partition, qui ressemble généralement à ceci:

```
data = sc.textFile(file)
data = data.coalesce(1)
```

C'est généralement une très mauvaise idée, puisque vous dites à Spark que **toutes les données ne** sont qu'une partition! Rappelez-vous que:

> Une étape dans Spark fonctionnera sur une partition à la fois (et chargera les données dans cette partition en mémoire).

Par conséquent, vous indiquez à Spark de gérer toutes les données à la fois, ce qui entraîne généralement des erreurs liées à la mémoire (mémoire insuffisante par exemple) ou même une exception de pointeur nul.

Donc, à moins de savoir ce que vous faites, évitez de partitionner votre RDD en une seule partition!

Examples

Partitions Intro

> Comment un RDD est-il partitionné?

Par défaut, une partition est créée pour chaque partition HDFS, qui est par défaut de 64 Mo. Lire plus ici .

Comment équilibrer mes données entre partitions?

Tout d'abord, jetez un coup d'œil sur les trois façons de *répartir* ses données:

1. Transmettez un second paramètre, le nombre *minimum* de partitions souhaité pour votre RDD, dans textFile () , mais faites attention:

   In [14]: lines = sc.textFile ("data")

   Dans [15]: lines.getNumPartitions () Out [15]: 1000

   In [16]: lines = sc.textFile ("data", 500)

   Dans [17]: lines.getNumPartitions () Out [17]: 1434

   In [18]: lines = sc.textFile ("data", 5000)

   Dans [19]: lines.getNumPartitions () Out [19]: 5926

Comme vous pouvez le voir, `[16]` ne fait pas ce à quoi on s'attendrait, car le nombre de partitions que possède le RDD est déjà supérieur au nombre minimum de partitions que nous demandons.

2. Utilisez repartition () , comme ceci:

   Dans [22]: lignes = lignes.partition (10)

   Dans [23]: lines.getNumPartitions () Out [23]: 10

Attention: ceci invoquera un shuffle et devrait être utilisé quand vous voulez **augmenter** le nombre de partitions de votre RDD.

De la documentation :

> Le shuffle est le mécanisme utilisé par Spark pour redistribuer les données afin qu'elles soient regroupées différemment selon les partitions. Cela implique généralement la copie des données sur les exécuteurs et les machines, ce qui en fait une opération complexe et coûteuse.

3. Utilisez coalesce () , comme ceci:

   In [25]: lines = lines.coalesce (2)

   Dans [26]: lines.getNumPartitions () Out [26]: 2

Spark sait que vous allez réduire le RDD et en tirer parti. En savoir plus sur repartition () vs coalesce () .

Mais tout cela **garantira-t-il** que vos données seront parfaitement équilibrées sur vos partitions? Pas vraiment, comme je l'ai expérimenté dans Comment équilibrer mes données entre les partitions?

Partitions d'un RDD

Comme mentionné dans "Remarques", une partition est une partie / une tranche / un bloc d'un RDD. Voici un exemple minimal sur la façon de demander un nombre minimum de partitions pour votre RDD:

```
In [1]: mylistRDD = sc.parallelize([1, 2, 3, 4, 5, 6, 7, 8, 9, 10], 2)

In [2]: mylistRDD.getNumPartitions()
Out[2]: 2
```

Remarquez dans [1] comment nous avons passé 2 comme second paramètre de `parallelize()` . Ce paramètre dit que nous voulons que notre RDD ait au moins 2 partitions.

Repartitionner un RDD

Nous souhaitons parfois repartitionner un RDD, par exemple parce qu'il provient d'un fichier que nous n'avons pas créé, et que le nombre de partitions défini à partir du créateur n'est pas celui que nous souhaitons.

Les deux fonctions les plus connues pour y parvenir sont:

```
repartition(numPartitions)
```

et:

```
coalesce(numPartitions, shuffle=False)
```

En règle générale, utilisez le premier lorsque vous souhaitez repartitionner votre RDD dans un plus grand nombre de partitions et le second pour réduire votre RDD, dans un nombre plus restreint de partitions. Spark - repartition () vs coalesce () .

Par exemple:

```
data = sc.textFile(file)
data = data.coalesce(100) // requested number of #partitions
```

diminuera le nombre de partitions du RDD appelé 'data' à 100, étant donné que ce RDD a plus de 100 partitions lorsqu'il a été lu par `textFile()` .

Et de la même manière, si vous souhaitez avoir plus que le nombre actuel de partitions pour votre RDD, vous pouvez le faire (étant donné que votre RDD est distribué dans 200 partitions par exemple):

```
data = sc.textFile(file)
data = data.repartition(300) // requested number of #partitions
```

Règle de base sur le nombre de partitions

En règle générale, on voudrait que son RDD ait autant de partitions que le produit du nombre d'exécuteurs par le nombre de cœurs utilisés par 3 (ou peut-être 4). Bien sûr, c'est une heuristique et cela dépend vraiment de votre application, de votre jeu de données et de la configuration de votre cluster.

Exemple:

```
In [1]: data  = sc.textFile(file)

In [2]: total_cores = int(sc._conf.get('spark.executor.instances')) *
int(sc._conf.get('spark.executor.cores'))

In [3]: data = data.coalesce(total_cores * 3)
```

Afficher le contenu RDD

Pour afficher le contenu d'un RDD, il faut l'imprimer:

```
myRDD.foreach(println)
```

Pour limiter le nombre de lignes imprimées:

```
myRDD.take(num_of_rows).foreach(println)
```

Lire Partitions en ligne: https://riptutorial.com/fr/apache-spark/topic/5822/partitions

Chapitre 16: Spark DataFrame

Introduction

Un DataFrame est une abstraction de données organisées en lignes et en colonnes typées. Il est similaire aux données trouvées dans les bases de données SQL relationnelles. Bien qu'il ait été transformé en un simple alias de type pour Dataset [Row] dans Spark 2.0, il est encore largement utilisé et utile pour les pipelines de traitement complexes utilisant sa flexibilité de schéma et ses opérations SQL.

Examples

Création de DataFrames dans Scala

Il existe plusieurs façons de créer des DataFrames. Ils peuvent être créés à partir de listes locales, de RDD distribués ou de lectures à partir de sources de données.

Utiliser toDF

En important les implicits spark sql, on peut créer un DataFrame à partir d'un Seq, Array ou RDD local, tant que le contenu est d'un sous-type Product (tuples et classes de cas sont des exemples bien connus de sous-types Product). Par exemple:

```
import sqlContext.implicits._
val df = Seq(
  (1, "First Value", java.sql.Date.valueOf("2010-01-01")),
  (2, "Second Value", java.sql.Date.valueOf("2010-02-01"))
).toDF("int_column", "string_column", "date_column")
```

Utiliser createDataFrame

Une autre option consiste à utiliser la méthode `createDataFrame` présente dans SQLcontext. Cette option permet également la création à partir de listes locales ou de RDD de sous-types de produits comme avec `toDF` , mais les noms des colonnes ne sont pas définis dans la même étape. Par exemple:

```
val df1 = sqlContext.createDataFrame(Seq(
  (1, "First Value", java.sql.Date.valueOf("2010-01-01")),
  (2, "Second Value", java.sql.Date.valueOf("2010-02-01"))
))
```

En outre, cette approche permet la création de fichiers RDD d'instances de `Row` , à condition qu'un paramètre de `schema` soit transmis pour la définition du schéma de DataFrame résultant. Exemple:

```
import org.apache.spark.sql.types._
val schema = StructType(List(
    StructField("integer_column", IntegerType, nullable = false),
    StructField("string_column", StringType, nullable = true),
    StructField("date_column", DateType, nullable = true)
))

val rdd = sc.parallelize(Seq(
  Row(1, "First Value", java.sql.Date.valueOf("2010-01-01")),
  Row(2, "Second Value", java.sql.Date.valueOf("2010-02-01"))
))

val df = sqlContext.createDataFrame(rdd, schema)
```

Lecture de sources

Le moyen le plus courant de créer DataFrame provient peut-être des sources de données. On peut le créer à partir d'un fichier parquet en hdfs, par exemple:

```
val df = sqlContext.read.parquet("hdfs:/path/to/file")
```

Lire Spark DataFrame en ligne: https://riptutorial.com/fr/apache-spark/topic/8358/spark-dataframe

Chapitre 17: Tests unitaires

Examples

Test d'unité de compte de mots (Scala + JUnit)

Par exemple, nous avons la méthode `WordCountService` avec `countWords` :

```
class WordCountService {
    def countWords(url: String): Map[String, Int] = {
        val sparkConf = new
SparkConf().setMaster("spark://somehost:7077").setAppName("WordCount"))
        val sc = new SparkContext(sparkConf)
        val textFile = sc.textFile(url)
        textFile.flatMap(line => line.split(" "))
                .map(word => (word, 1))
                .reduceByKey(_ + _).collect().toMap
    }
}
```

Ce service semble très laid et non adapté aux tests unitaires. SparkContext doit être injecté dans ce service. Il peut être atteint avec votre framework DI préféré mais pour plus de simplicité il sera implémenté en utilisant constructeur:

```
class WordCountService(val sc: SparkContext) {
    def countWords(url: String): Map[String, Int] = {
        val textFile = sc.textFile(url)
        textFile.flatMap(line => line.split(" "))
                .map(word => (word, 1))
                .reduceByKey(_ + _).collect().toMap
    }
}
```

Maintenant, nous pouvons créer un test JUnit simple et injecter un test sparkContext dans WordCountService:

```
class WordCountServiceTest {
    val sparkConf = new SparkConf().setMaster("local[*]").setAppName("WordCountTest")
    val testContext = new SparkContext(sparkConf)
    val wordCountService = new WordCountService(testContext)

    @Test
    def countWordsTest() {
        val testFilePath = "file://my-test-file.txt"

        val counts = wordCountService.countWords(testFilePath)

        Assert.assertEquals(counts("dog"), 121)
        Assert.assertEquals(counts("cat"), 191)
    }
}
```

Lire Tests unitaires en ligne: https://riptutorial.com/fr/apache-spark/topic/3333/tests-unitaires

Chapitre 18: Variables partagées

Examples

Variables de diffusion

Les variables de diffusion sont des objets partagés en lecture seule qui peuvent être créés avec la méthode `SparkContext.broadcast` :

```
val broadcastVariable = sc.broadcast(Array(1, 2, 3))
```

et lire en utilisant `value` méthode `value` :

```
val someRDD = sc.parallelize(Array(1, 2, 3, 4))

someRDD.map(
    i => broadcastVariable.value.apply(i % broadcastVariable.value.size)
)
```

Accumulateurs

Les accumulateurs sont des variables en écriture seule pouvant être créées avec `SparkContext.accumulator` :

```
val accumulator = sc.accumulator(0, name = "My accumulator") // name is optional
```

modifié avec `+=` :

```
val someRDD = sc.parallelize(Array(1, 2, 3, 4))
someRDD.foreach(element => accumulator += element)
```

et accessible avec `value` méthode `value` :

```
accumulator.value // 'value' is now equal to 10
```

L'utilisation des accumulateurs est compliquée par la garantie d'exécution de Spark pour les transformations. Si une transformation doit être recalculée pour une raison quelconque, les mises à jour de l'accumulateur pendant cette transformation seront répétées. Cela signifie que les valeurs de l'accumulateur peuvent être très différentes de ce qu'elles seraient si les tâches n'avaient été exécutées qu'une seule fois.

Remarque:

1. Les exécuteurs *ne peuvent pas* lire la valeur de l'accumulateur. Seul le programme pilote peut lire la valeur de l'accumulateur, en utilisant sa méthode de valeur.

2. Il est presque similaire à counter dans Java / MapReduce. Vous pouvez donc associer les accumulateurs aux compteurs pour les comprendre facilement

Accumulateur défini par l'utilisateur dans Scala

Définir `AccumulatorParam`

```
import org.apache.spark.AccumulatorParam

object StringAccumulator extends AccumulatorParam[String] {
  def zero(s: String): String = s
  def addInPlace(s1: String, s2: String)=  s1 + s2
}
```

Utilisation:

```
val accumulator = sc.accumulator("")(StringAccumulator)
sc.parallelize(Array("a", "b", "c")).foreach(accumulator += _)
```

Accumulateur défini par l'utilisateur en Python

Définir `AccumulatorParam` :

```
from pyspark import AccumulatorParam

class StringAccumulator(AccumulatorParam):
    def zero(self, s):
        return s
    def addInPlace(self, s1, s2):
        return s1 + s2

accumulator = sc.accumulator("", StringAccumulator())

def add(x):
    global accumulator
    accumulator += x

sc.parallelize(["a", "b", "c"]).foreach(add)
```

Lire Variables partagées en ligne: https://riptutorial.com/fr/apache-spark/topic/1736/variables-partagees

Crédits

S. No	Chapitres	Contributeurs
1	Démarrer avec apache-spark	4444, Ani Menon, Community, Daniel de Paula, David, gsamaras, himanshuIIITian, Jacek Laskowski, KartikKannapur, Naresh Kumar, user8371915, zero323
2	Appeler des emplois scala à partir de pyspark	eliasah, Thiago Baldim
3	Comment poser la question liée à Apache Spark?	user7337271
4	Configuration: Apache Spark SQL	himanshuIIITian
5	Fichiers texte et opérations à Scala	Ani Menon, Community, spiffman
6	Fonctions de fenêtre dans Spark SQL	Daniel Argüelles, Hari, Joshua Weinstein, Tejus Prasad, vdep
7	Introduction aux Apache Spark DataFrames	Mandeep Lohan, Nayan Sharma
8	Joint	Adnan, CGritton
9	Lanceur d'étincelles	Ankit Agrahari
10	Le message d'erreur 'sparkR' n'est pas reconnu en tant que commande interne ou externe ou '.binsparkR' n'est pas reconnu en tant que commande interne ou externe	Rajesh
11	Manipulation de JSON dans Spark	Furkan Varol, zero323

12	Migration de Spark 1.6 vers Spark 2.0	Béatrice Moissinac, eliasah, Shaido
13	Mode client et mode cluster	Nayan Sharma
14	Opérations avec état dans Spark Streaming	zero323
15	Partitions	Ani Menon, Armin Braun, gsamaras
16	Spark DataFrame	Daniel de Paula
17	Tests unitaires	Cortwave
18	Variables partagées	Community, Jonathan Taws, RBanerjee, saranvisa, spiffman, whaleberg, zero323