EBook Gratis

APRENDIZAJE apache-spark

Free unaffiliated eBook created from
Stack Overflow contributors.

#apache-spark

Tabla de contenido

Acerca de

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

Capítulo 1: Empezando con la chispa de apache

Observaciones

Apache Spark es un marco de procesamiento de datos grandes de código abierto creado en torno a la velocidad, la facilidad de uso y el análisis sofisticado. Un desarrollador debe usarlo cuando maneja una gran cantidad de datos, lo que generalmente implica limitaciones de memoria y / o tiempo de procesamiento prohibitivo.

También debe mencionar cualquier tema grande dentro de apache-spark y vincular a los temas relacionados. Dado que la Documentación para apache-spark es nueva, es posible que deba crear versiones iniciales de esos temas relacionados.

Versiones

Versión	Fecha de lanzamiento
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

Versión	Fecha de lanzamiento
0.7.0	2013-02-27
0.6.0	2012-10-15

Examples

Introducción

Prototipo :

> agregado (valor cero, seqOp, combOp)

Descripción :

`aggregate()` permite tomar un RDD y generar un valor único que es de un tipo diferente al que estaba almacenado en el RDD original.

Parámetros :

1. `zeroValue` : el valor de inicialización, para su resultado, en el formato deseado.
2. `seqOp` : la operación que desea aplicar a los registros RDD. Se ejecuta una vez por cada registro en una partición.
3. `combOp` : define cómo se `combOp` los objetos resultantes (uno para cada partición).

Ejemplo :

> Calcule la suma de una lista y la longitud de esa lista. Devuelve el resultado en un par de `(sum, length)` .

En un shell de Spark, crea una lista con 4 elementos, con 2 *particiones* :

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

Luego define *seqOp* :

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

Luego define *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]) )
```

Luego se agrega:

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

La primera partición tiene la sublista [1, 2]. Esto aplica el seqOp a cada elemento de esa lista, que produce un resultado local: un par de `(sum, length)` que reflejará el resultado localmente, solo en esa primera partición.

`local_result` se inicializa con el parámetro `zeroValue` `aggregate()` se proporcionó. Por ejemplo, (0, 0) y `list_element` es el primer elemento de la lista:

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

El resultado local es (1, 1), lo que significa que la suma es 1 y la longitud 1 para la primera partición después de procesar *solo* el primer elemento. `local_result` se actualiza de (0, 0), a (1, 1).

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

El resultado local es ahora (3, 2), que será el resultado final de la primera partición, ya que no hay otros elementos en la lista secundaria de la primera partición. Haciendo lo mismo para la 2ª partición devuelve (7, 2).

Aplique combOp a cada resultado local para formar el resultado global final:

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

Ejemplo descrito en 'figura':

```
            (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)
```

Transformación vs acción

Spark utiliza la **evaluación perezosa** ; eso significa que no hará ningún trabajo, a menos que realmente tenga que hacerlo. Ese enfoque nos permite evitar el uso innecesario de la memoria, lo que nos permite trabajar con big data.

Una *transformación* se evalúa de forma perezosa y el trabajo real ocurre cuando se produce una *acción* .

Ejemplo:

```
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
```

Entonces, en `[1]` le dijimos a Spark que leyera un archivo en un RDD, llamado `lines` . Spark nos escuchó y nos dijo: "Sí, *lo* haré", pero en realidad *aún* no *había* leído el archivo.

En [2], estamos filtrando las líneas del archivo, asumiendo que su contenido contiene líneas con errores que están marcados con un `error` en su inicio. Entonces le pedimos a Spark que cree un nuevo RDD, llamado `errors` , que tendrá los elementos de las `lines` RDD, que tenían la palabra `error` al comienzo.

Ahora en `[3]` , le pedimos a Spark que *cuente* los *errores* , es decir, que cuente el número de elementos que tiene el RDD llamado `errors` . `count()` es una **acción** , que no deja ninguna opción a Spark, sino a realizar la operación, para que pueda encontrar el resultado de `count()` , que será un número entero.

Como resultado, cuando `[3]` se alcanza, `[1]` y `[2]` en realidad se está realizando, es decir, que cuando se llega a `[3]` , entonces y sólo entonces:

1. el archivo se leerá en `textFile()` (debido a `[1]`)
2. `lines` serán `filter()` 'ed (debido a `[2]`)
3. `count()` se ejecutará, debido a `[3]`

Consejo de depuración: como Spark no realizará ningún trabajo real hasta que se alcance `[3]` , es importante entender que si existe un error en `[1]` y / o `[2]` , no aparecerá hasta que aparezca la acción en `[3]` activa a Spark para hacer un trabajo real. Por ejemplo, si sus datos en el archivo no son compatibles con `startsWith()` que usé, entonces `[2]` será aceptado por Spark y no generará ningún error, pero cuando `[3]` se envía, y Spark en realidad evalúa tanto `[1]` como `[2]` , entonces y solo entonces entenderá que algo no es correcto con `[2]` y producirá un error descriptivo.

Como resultado, se puede desencadenar un error cuando se ejecuta `[3]` , ¡pero eso no significa que el error deba estar en la declaración de `[3]` !

Tenga en cuenta que ni las `lines` ni los `errors` se almacenarán en la memoria después de `[3]` . Continuarán existiendo solo como un conjunto de instrucciones de procesamiento. Si se realizarán varias acciones en cualquiera de estos RDD, spark leerá y filtrará los datos varias

veces. Para evitar la duplicación de operaciones cuando se realizan varias acciones en un solo RDD, a menudo es útil almacenar datos en la memoria usando la memoria `cache` .

Puedes ver más transformaciones / acciones en documentos de Spark .

Revisar la versión Spark

En `spark-shell` :

```
sc.version
```

Generalmente en un programa:

```
SparkContext.version
```

Utilizando `spark-submit` :

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

Lea Empezando con la chispa de apache en línea: https://riptutorial.com/es/apache-spark/topic/833/empezando-con-la-chispa-de-apache

Capítulo 2: ¿Cómo hacer una pregunta relacionada con Apache Spark?

Introducción

El objetivo de este tema es documentar las mejores prácticas al hacer preguntas relacionadas con Apache Spark.

Examples

Detalles del entorno:

Cuando haga preguntas relacionadas con Apache Spark, incluya la siguiente información

- Versión de Apache Spark utilizada por el cliente y la implementación de Spark, si corresponde. Para preguntas relacionadas con la API, las principales (1.6, 2.0, 2.1, etc.) son generalmente suficientes, para preguntas relacionadas con posibles errores, siempre use la información de la versión completa.
- Versión de Scala utilizada para construir binarios de Spark.
- Versión JDK (versión `java -version`).
- Si utiliza el idioma invitado (Python, R), brinde información sobre la versión del idioma. En Python, utilice las etiquetas: python-2.x , python-3.xo más específicas para distinguir entre las variantes de idioma.
- Cree la definición (`build.sbt` , `pom.xml`) si corresponde o versiones de dependencia externas (Python, R) cuando corresponda.
- Administrador de clústeres (`local[n]` , Spark standalone, Yarn, Mesos), modo (`client` , `cluster`) y otras opciones de envío, si corresponde.

Ejemplo de datos y código

Ejemplo de datos

Intente proporcionar un mínimo de datos de entrada de ejemplo en un formato que puedan ser utilizados directamente por las respuestas sin un análisis tedioso y lento, por ejemplo, un archivo de entrada o una colección local con todo el código necesario para crear estructuras de datos distribuidas.

Cuando sea aplicable siempre incluya información de tipo:

- En la API basada en RDD use anotaciones de tipo cuando sea necesario.
- En la API basada en DataFrame, proporcione información de esquema como un tipo de `StrucType` o salida de `Dataset.printSchema` .

La salida de `Dataset.show` o `print` puede verse bien pero no nos dice nada sobre los tipos

subyacentes.

Si un problema en particular ocurre solo a escala, use generadores de datos aleatorios (Spark proporciona algunas utilidades útiles en `org.apache.spark.mllib.random.RandomRDDs` y `org.apache.spark.graphx.util.GraphGenerators`

Código

Por favor, use anotaciones de tipo cuando sea posible. Si bien su compilador puede realizar un seguimiento de los tipos fácilmente, no es tan fácil para los simples mortales. Por ejemplo:

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

o

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

Son mejor que:

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

y

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

respectivamente.

Informacion diagnostica

Preguntas de depuración.

Cuando la pregunta está relacionada con la depuración de una excepción específica, proporcione siempre un rastreo relevante. Si bien es recomendable eliminar las salidas duplicadas (de diferentes ejecutores o intentos), no corte las trazas de retorno a una sola línea o clase de excepción.

Preguntas de rendimiento.

Dependiendo del contexto, trate de proporcionar detalles como:

- `RDD.debugString` / `Dataset.explain` .
- Salida de Spark UI con diagrama DAG si es aplicable en un caso particular.
- Mensajes de registro relevantes.
- Información de diagnóstico recogida por herramientas externas (Ganglia, VisualVM).

Antes de preguntar

- Búsqueda de desbordamiento de pila para preguntas duplicadas. Hay una clase común de problemas que ya han sido ampliamente documentados.
- Leer ¿Cómo hago una buena pregunta? .
- Leer ¿Qué temas puedo preguntar aquí?
- Recursos de la comunidad de Apache Spark

Lea ¿Cómo hacer una pregunta relacionada con Apache Spark? en línea: https://riptutorial.com/es/apache-spark/topic/8815/-como-hacer-una-pregunta-relacionada-con-apache-spark-

Capítulo 3: Archivos de texto y operaciones en Scala

Introducción

Leyendo archivos de texto y realizando operaciones en ellos.

Examples

Ejemplo de uso

Leer el archivo de texto de la ruta:

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

Leer archivos usando comodines:

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

Leer archivos especificando el número mínimo de particiones:

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

Unir dos archivos leídos con textFile ()

Se une en Spark:

- Leer textoArchivo 1

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

  P.ej:

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

- Leer textoArchivo 2

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

  P.ej:

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

- Únete e imprime el resultado.

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

P.ej:

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

La unión anterior se basa en la primera columna.

Lea Archivos de texto y operaciones en Scala en línea: https://riptutorial.com/es/apache-spark/topic/1620/archivos-de-texto-y-operaciones-en-scala

Capítulo 4: Configuración: Apache Spark SQL

Introducción

En este tema, los usuarios de Spark pueden encontrar diferentes configuraciones de Spark SQL, que es el componente más utilizado del marco de Apache Spark.

Examples

Control de particiones aleatorias de Spark SQL

En Apache Spark, mientras se realizan operaciones aleatorias como `join` y `cogroup` muchos datos, se transfieren a través de la red. Ahora, para controlar el número de particiones sobre las cuales se produce el orden aleatorio, se puede controlar mediante configuraciones dadas en Spark SQL. Esa configuración es la siguiente:

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

Usando esta configuración podemos controlar el número de particiones de las operaciones aleatorias. Por defecto, su valor es `200` . Pero, 200 particiones no tiene ningún sentido si tenemos archivos de pocos GB (s). Por lo tanto, debemos cambiarlos de acuerdo con la cantidad de datos que necesitamos procesar a través de Spark SQL. Me gusta como sigue:

En este escenario tenemos dos tablas para unir `employee` y `department` . Ambas tablas solo contienen pocos registros, pero necesitamos unirlos para conocer el departamento de cada empleado. Entonces, nos unimos a ellos usando Spark DataFrames como este:

```
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()
```

Ahora, la cantidad de particiones que se crean mientras se realiza la unión son 200 por defecto, lo que por supuesto es demasiado para esta cantidad de datos.

Por lo tanto, cambiemos este valor para que podamos reducir el número de operaciones aleatorias.

```
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()
```

Ahora, el número de particiones aleatorias se reduce a solo 2, lo que no solo reducirá el número de operaciones de orden aleatorio, sino que también reducirá el tiempo necesario para unir los marcos de `0.878505 s` de `0.878505 s 0.077847 s` .

Por lo tanto, siempre configure el número de particiones para las operaciones aleatorias de acuerdo con los datos que se procesan.

Lea Configuración: Apache Spark SQL en línea: https://riptutorial.com/es/apache-spark/topic/8169/configuracion--apache-spark-sql

Capítulo 5: El mensaje de error 'sparkR' no se reconoce como un comando interno o externo o '.binsparkR' no se reconoce como un comando interno o externo

Introducción

Esta publicación es para aquellos que tenían problemas para instalar Spark en su máquina Windows. Principalmente utilizando la función sparkR para la sesión R.

Observaciones

Referencia utilizada de r-bloggers

Examples

detalles para configurar Spark para R

Use la siguiente URL para obtener los pasos para descargar e instalar: https://www.r-bloggers.com/installing-and-starting-sparkr-locally-on-windows-os-and-rstudio-2/ Agregue la ruta de la variable de entorno para su ruta 'Spark / bin', 'spark / bin', R y Rstudio. He agregado la ruta siguiente (las iniciales variarán según el lugar donde descargó los archivos) C: \ spark-2.0.1 C: \ spark-2.0.1 \ bin C: \ spark-2.0.1 \ sbin C: \ Archivos de programa \ R \ R-3.3.1 \ bin \ x64 C: \ Archivos de programa \ RStudio \ bin \ x64

Para configurar la variable de entorno, siga los siguientes pasos: Windows 10 y Windows 8 En la búsqueda, busque y luego seleccione: Sistema (Panel de control) Haga clic en el enlace Configuración avanzada del sistema. Haga clic en la pestaña Opciones avanzadas en Propiedades de sistema Haga clic en Variables de entorno. En la sección Variables del sistema, busque la variable de entorno PATH y selecciónela. Haga clic en Editar. Si la variable de entorno PATH no existe, haga clic en Nuevo. En la ventana Editar variable del sistema (o Nueva variable del sistema), especifique el valor de la variable de entorno PATH. Haga clic en Aceptar. Cierre todas las ventanas restantes haciendo clic en Aceptar. Vuelva a abrir la ventana de solicitud de comando y ejecute sparkR (no es necesario cambiar el directorio).

Windows 7 Desde el escritorio, haga clic derecho en el ícono de la computadora. Elija Propiedades en el menú contextual. Haga clic en el enlace Configuración avanzada del sistema. Haga clic en Variables de entorno. En la sección Variables del sistema, busque la variable de entorno PATH y selecciónela. Haga clic en Editar. Si la variable de entorno PATH no existe, haga clic en Nuevo. En la ventana Editar variable del sistema (o Nueva variable del sistema), especifique el valor de la variable de entorno PATH. Haga clic en Aceptar. Cierre todas las

ventanas restantes haciendo clic en Aceptar. Vuelva a abrir la ventana de solicitud de comando y ejecute sparkR (no es necesario cambiar el directorio).

Lea El mensaje de error 'sparkR' no se reconoce como un comando interno o externo o '.binsparkR' no se reconoce como un comando interno o externo en línea: https://riptutorial.com/es/apache-spark/topic/9649/el-mensaje-de-error--sparkr--no-se-reconoce-como-un-comando-interno-o-externo-o---binsparkr--no-se-reconoce-como-un-comando-interno-o-externo

Capítulo 6: Funciones de ventana en Spark SQL

Examples

Introducción

Las funciones de ventana se utilizan para realizar operaciones (generalmente agregación) en un conjunto de filas llamadas colectivamente como ventana. Las funciones de la ventana funcionan en Spark 1.4 o posterior. Las funciones de la ventana proporcionan más operaciones que las funciones integradas o UDF, como substr o round (utilizadas ampliamente antes de Spark 1.4). Las funciones de la ventana permiten a los usuarios de Spark SQL calcular resultados como el rango de una fila dada o un promedio móvil en un rango de filas de entrada. Mejoran significativamente la expresividad de las API de SQL y DataFrame de Spark.

En su núcleo, una función de ventana calcula un valor de retorno para cada fila de entrada de una tabla basada en un grupo de filas, llamado Marco. Cada fila de entrada puede tener un marco único asociado a ella. Esta característica de las funciones de ventana las hace más poderosas que otras funciones. Los tipos de funciones de ventana son

- Funciones de clasificación
- Funciones analiticas
- Funciones agregadas

Para usar las funciones de la ventana, los usuarios deben marcar que una función se usa como una función de la ventana ya sea por

- Agregar una cláusula `OVER` después de una función compatible en SQL, por ejemplo, `avg(revenue) OVER (...);` o
- Llamar al método over en una función admitida en la API DataFrame, por ejemplo, `rank().over(...)` Over `rank().over(...)` .

Esta documentación pretende demostrar algunas de esas funciones con ejemplo. Se supone que el lector tiene algún conocimiento sobre las operaciones básicas en Spark DataFrame como: agregar una nueva columna, cambiar el nombre de una columna, etc.

Leyendo un conjunto de datos de muestra:

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

Lista de declaraciones de importación requeridas:

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

La primera declaración importa la `Window Specification` . Una especificación de ventana contiene condiciones / especificaciones que indican qué filas deben incluirse en la ventana.

```
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|
+------+---------+------+
```

Media móvil

Para calcular la media móvil del salario de los empleadores en función de su función:

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

- `withColumn()` crea una nueva columna llamada `movingAverage` , que realiza un `average` en la columna `Salary`
- `over()` se utiliza para definir la especificación de la ventana.
- `partitionBy()` particiona los datos sobre la columna `Role`
- `rowsBetween(start, end)` Esta función define las filas que se incluirán en la ventana. Los parámetros (`start` y `end`) toman entradas numéricas, `0` representa la fila actual, `-1` es la fila anterior, `1` es la fila siguiente y así sucesivamente. La función incluye todas las filas entre el `start` y el `end` , por lo que en este ejemplo se incluyen tres filas (-1,0,1) en la ventana.

```
    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 ignora automáticamente las filas anteriores y siguientes, si la fila actual es la primera y la última fila respectivamente.

En el ejemplo anterior, movingAverage de la primera fila es el promedio de la fila actual y la siguiente solamente, ya que la fila anterior no existe. De manera similar, la última fila de la partición (es decir, la sexta fila) es el promedio de la fila actual y anterior, ya que la siguiente fila no existe.

Suma acumulativa

Para calcular la media móvil del salario de los empleadores en función de su función:

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

- `orderBy()` ordena la columna de salario y calcula la suma acumulada.

```
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|
+------+---------+------+-------------+
```

Funciones de la ventana: clasificación, avance, desfase, clasificación, análisis de tendencias

Este tema muestra cómo usar funciones como con Columna, plomo, retraso, Nivel, etc. usando Spark. El marco de datos de Spark es una capa abstracta de sql en las funcionalidades de Spark Core Esto permite al usuario escribir SQL en datos distribuidos. Spark SQL es compatible con formatos de archivo heterogéneos, incluidos JSON, XML, CSV, TSV, etc.

En este blog, tenemos una descripción general rápida de cómo usar spark SQL y marcos de datos para casos de uso comunes en el mundo de SQL. Por simplicidad, trataremos un solo archivo con formato CSV. El archivo tiene cuatro campos, employeeID, employeeName ,larre, sueldoFecha

```
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
```

Guarde este archivo como emp.dat. En el primer paso, crearemos un marco de datos de chispa

usando el paquete CSV de chispa de 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 es el marco de datos utilizado en el ejercicio restante. Debido a que myDF se usa repetidamente, se recomienda persistir para que no sea necesario reevaluarlo.

```
 myDF.persist()
```

Salida de muestra de datos

```
+-----+-------+------+----------+
|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|
+-----+-------+------+----------+
```

Agregar una nueva columna a dataframe

Dado que los marcos de datos de chispa son inmutables, agregar una nueva columna creará un nuevo marco de datos con una columna adicional. Para agregar una columna, use withColumn (columnName, Transformation). En la columna de ejemplo a continuación empName está formateada en mayúsculas.

```
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|
```

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

Ordenar los datos en base a una columna

```
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|
+-----+-------+------+----------+
```

Orden descendiente

desc ("salario")

```
 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|
+-----+-------+------+----------+
```

Obtener y usar la fila anterior (Lag)

LAG es una función en SQL que se utiliza para acceder a los valores de las filas anteriores en la fila actual. Esto es útil cuando tenemos casos de uso como la comparación con el valor anterior. LAG en los marcos de datos de Spark está disponible en las funciones de Windows

```
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|
+-----+-------+------+----------+------+
```

Obtener y usar la siguiente fila (Lead)

LEAD es una función en SQL que se utiliza para acceder a los valores de la siguiente fila en la fila actual. Esto es útil cuando tenemos casos como la comparación con el siguiente valor. LEAD in Spark dataframes está disponible en las funciones de Windows

```
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|
+-----+-------+------+----------+-------+
```

Análisis de tendencias con funciones de ventana Ahora, pongamos la función de ventana LAG para usar con un simple análisis de tendencias. Si el salario es menor que el mes anterior, lo marcaremos como "ABAJO", si el salario aumentó, entonces "ARRIBA". El código usa la función de la ventana para ordenar, retrasar y luego hacer una simple si no con 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|
+-----+-------+------+----------+----------+----------+-----+
```

Lea Funciones de ventana en Spark SQL en línea: https://riptutorial.com/es/apache-spark/topic/3903/funciones-de-ventana-en-spark-sql

Capítulo 7: Introducción a Apache Spark DataFrames

Examples

Spark DataFrames con JAVA

Un DataFrame es una colección distribuida de datos organizados en columnas nombradas. Conceptualmente es equivalente a una tabla en una base de datos relacional. Los DataFrames se pueden construir a partir de una amplia gama de fuentes, tales como: archivos de datos estructurados, tablas en Hive, bases de datos externas o RDD existentes.

Leyendo una tabla de Oracle RDBMS en el marco de datos de chispa ::

```
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
```

También podemos convertir este marco de datos de nuevo a rdd si es necesario:

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

Crear un marco de datos a partir de un archivo:

```
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();
    }
}
```

Ahora tenemos el marco de datos de Oracle también de un archivo. Del mismo modo podemos leer una tabla de colmena también. En el marco de datos podemos obtener cualquier columna como lo hacemos en rdbms. Como obtener un valor mínimo para una columna o valor máximo. Puede calcular una media / promedio para una columna. Algunas otras funciones como seleccionar, filtrar, agregar, agrupar también están disponibles.

Spark Dataframe explicado

En Spark, un DataFrame es una colección distribuida de datos organizados en columnas con

nombre. Conceptualmente es equivalente a una tabla en una base de datos relacional o un marco de datos en R / Python, pero con optimizaciones más ricas bajo el capó. Los DataFrames se pueden construir a partir de una amplia gama de fuentes, como archivos de datos estructurados, tablas en Hive, bases de datos externas o RDD existentes.

Formas de crear Dataframe

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

`val df = spark.read.format("com.databricks.spark.csv").load("test.txt")` en el campo de opciones, puede proporcionar encabezado, delimitador, conjunto de caracteres y mucho más

También puede crear Dataframe desde 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 quieres crear df con esquema

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

¿Por qué necesitamos Dataframe si Spark ha proporcionado RDD?

Un RDD es simplemente un conjunto de datos distribuidos resilientes que es más bien una caja negra de datos que no pueden optimizarse, ya que las operaciones que se pueden realizar en su contra no están tan limitadas.

Ningún motor de optimización incorporado: cuando se trabaja con datos estructurados, los RDD no pueden aprovechar las ventajas de los optimizadores avanzados de Spark, como el optimizador de catalizador y el motor de ejecución de Tungsten. Los desarrolladores necesitan optimizar cada RDD en función de sus atributos. Manejo de datos estructurados: a diferencia de los marcos de datos y los conjuntos de datos, los RDD no deducen el esquema de los datos ingeridos y requieren que el usuario los especifique.

Los DataFrames en Spark tienen su ejecución optimizada automáticamente por un optimizador de consultas. Antes de que se inicie cualquier cálculo en un DataFrame, el optimizador de Catalyst compila las operaciones que se usaron para construir el DataFrame en un plan físico para su ejecución. Debido a que el optimizador entiende la semántica de las operaciones y la estructura de los datos, puede tomar decisiones inteligentes para acelerar el cálculo.

Limitación de DataFrame

Tipo de seguridad de tiempo de compilación: Dataframe API no admite la seguridad de tiempo de

compilación, lo que le impide manipular datos cuando no se conoce la estructura.

Lea Introducción a Apache Spark DataFrames en línea: https://riptutorial.com/es/apache-spark/topic/6514/introduccion-a-apache-spark-dataframes

Capítulo 8: Lanzador de chispas

Observaciones

Spark Launcher puede ayudar al desarrollador a sondear el estado del trabajo de chispa enviado. Básicamente hay ocho estados que pueden ser encuestados. Se enumeran a continuación con un significado:

```
/** 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

El código de abajo es un ejemplo básico de lanzador de chispas. Esto se puede usar si el trabajo de chispa tiene que iniciarse a través de alguna aplicación.

```
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)
```

```
}
```

Lea Lanzador de chispas en línea: https://riptutorial.com/es/apache-spark/topic/8026/lanzador-de-chispas

Capítulo 9: Llamando scala empleos desde pyspark

Introducción

Este documento le mostrará cómo llamar a los trabajos de Scala desde una aplicación pyspark.

Este enfoque puede ser útil cuando a la API de Python le faltan algunas de las características existentes de la API de Scala o incluso para hacer frente a los problemas de rendimiento con Python.

En algunos casos de uso, el uso de Python es inevitable, por ejemplo, estás creando modelos con `scikit-learn` .

Examples

Creando una función de Scala que recibe un RDD de python

Crear una función de Scala que reciba un RDD de python es fácil. Lo que necesitas para construir es una función que obtenga un JavaRDD [Cualquiera]

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

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

Serializar y enviar Python RDD al código de Scala

En esta parte del desarrollo, debe serializar el RDD de Python a la JVM. Este proceso utiliza el desarrollo principal de Spark para llamar a la función 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)
```

Cómo llamar a spark-submit

Para llamar a este código, debes crear el tarro de tu código de Scala. Entonces tienes que llamar

a tu chispa enviar así:

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

Esto le permitirá llamar a cualquier tipo de código de Scala que necesite en sus trabajos de pySpark

Lea Llamando scala empleos desde pyspark en línea: https://riptutorial.com/es/apache-spark/topic/9180/llamando-scala-empleos-desde-pyspark

Capítulo 10: Manejo de JSON en Spark

Examples

Mapeo de JSON a una clase personalizada con Gson

Con `Gson` , puede leer el conjunto de datos JSON y asignarlos a una clase personalizada `MyClass` .

Como `Gson` no es serializable, cada ejecutor necesita su propio objeto `Gson` . Además, `MyClass` debe ser serializable para pasarlo entre los ejecutores.

Tenga en cuenta que el archivo (s) que se ofrece como un archivo json no es un archivo JSON típico. Cada línea debe contener un objeto JSON válido independiente y autónomo. Como consecuencia, un archivo JSON multilínea regular fallará con mayor frecuencia.

```
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 creación del objeto `Gson` resulta demasiado costosa, se puede utilizar el método `mapPartitions` para optimizarlo. Con él, habrá un `Gson` por partición en lugar de por línea:

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

Lea Manejo de JSON en Spark en línea: https://riptutorial.com/es/apache-spark/topic/2799/manejo-de-json-en-spark

Capítulo 11: Migración de Spark 1.6 a Spark 2.0

Introducción

Spark 2.0 ha sido lanzado y contiene muchas mejoras y nuevas características. Si está utilizando Spark 1.6 y ahora desea actualizar su aplicación para usar Spark 2.0, debe tener en cuenta algunos cambios en la API. A continuación se muestran algunos de los cambios en el código que deben realizarse.

Examples

Actualizar el archivo build.sbt

Actualizar build.sbt con:

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

Tenga en cuenta que al compilar con el `sbt package` , el `.jar` ahora se creará en `target/scala-2.11/` , y el nombre de `.jar` también se cambiará, por lo que el comando `spark-submit` debe actualizarse.

Actualizar ML bibliotecas de vectores

ML `Transformers` ahora genera `org.apache.spark.ml.linalg.VectorUDT` lugar de `org.apache.spark.mllib.linalg.VectorUDT` .

También se asignan localmente a las subclases de `org.apache.spark.ml.linalg.Vector` . Estos no son compatibles con la antigua API de MLLib que se está moviendo hacia la desaprobación en 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
```

Lea Migración de Spark 1.6 a Spark 2.0 en línea: https://riptutorial.com/es/apache-spark/topic/6506/migracion-de-spark-1-6-a-spark-2-0

Capítulo 12: Modo cliente y modo clúster

Examples

Explicación del modo Spark Client y Cluster.

Intentemos ver las diferencias entre el modo cliente y el modo de clúster de Spark.

Cliente : cuando se ejecuta Spark en el modo cliente, el programa SparkContext and Driver se ejecuta de manera externa al clúster; por ejemplo, desde tu laptop. El modo local es solo para el caso en el que no desea utilizar un clúster y, en cambio, desea ejecutar todo en una sola máquina. Por lo tanto, Driver Application y Spark Application están en la misma máquina que el usuario. El controlador se ejecuta en un servidor dedicado (nodo maestro) dentro de un proceso dedicado. Esto significa que tiene todos los recursos disponibles a su disposición para ejecutar el trabajo. Debido a que el nodo maestro tiene recursos dedicados propios, no necesita "gastar" recursos de trabajo para el programa Driver. Si el proceso del controlador muere, necesita un sistema de monitoreo externo para restablecer su ejecución.

Clúster: el controlador se ejecuta en uno de los nodos Trabajadores del clúster. Se ejecuta como un proceso independiente y dedicado dentro del Trabajador. Cuando trabaje en modo Cluster, todos los JAR relacionados con la ejecución de su aplicación deben estar disponibles públicamente para todos los trabajadores. Esto significa que puede colocarlos manualmente en un lugar compartido o en una carpeta para cada uno de los trabajadores. Cada aplicación tiene sus propios procesos de ejecutor, que permanecen activos durante toda la aplicación y ejecutan tareas en varios subprocesos. Esto tiene la ventaja de aislar las aplicaciones entre sí, tanto en el lado de la programación (cada controlador programa sus propias tareas) como en el lado del ejecutor (las tareas de diferentes aplicaciones se ejecutan en diferentes JVM).

Tipos de administrador de clúster

Apache Mesos: un administrador de clúster general que también puede ejecutar Hadoop MapReduce y brindar servicio a las aplicaciones. Hadoop YARN - el administrador de recursos en Hadoop.
Kubernetes- container-centric Infraestructure.it es experimental todavía.

Lea Modo cliente y modo clúster en línea: https://riptutorial.com/es/apache-spark/topic/10808/modo-cliente-y-modo-cluster

Capítulo 13: Operaciones con estado en Spark Streaming

Examples

PairDStreamFunctions.updateStateByKey

`updateState` by key se puede usar para crear un `DStream` estado basado en los próximos datos. Requiere una función:

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

que toma una secuencia de los valores `current` , una `Option` del estado anterior y devuelve una `Option` del estado actualizado. Poniendo todo esto junto:

```
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` , al igual que `updateState` , se puede usar para crear un DStream con estado basado en los próximos datos. Requiere `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)
    }
  )
}
```

que toma clave `key` , `value` actual y `State` acumulado y devuelve nuevo estado. Poniendo todo esto junto:

```
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()
  }
}
```

Finalmente se espera la salida:

```
-------------------------------------------
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))
```

Lea Operaciones con estado en Spark Streaming en línea: https://riptutorial.com/es/apache-spark/topic/1924/operaciones-con-estado-en-spark-streaming

Capítulo 14: Particiones

Observaciones

El número de particiones es crítico para el rendimiento de una aplicación y / o la terminación exitosa.

Un conjunto de datos distribuido resistente (RDD) es la abstracción principal de Spark. Un RDD se divide en particiones, lo que significa que una partición es una parte del conjunto de datos, una porción de ella, o en otras palabras, una parte de ella.

Cuanto mayor sea el número de particiones, menor será el tamaño de cada partición.

Sin embargo, tenga en cuenta que un gran número de particiones ejerce mucha presión sobre el Sistema de archivos distribuidos de Hadoop (HDFS), que debe mantener una cantidad significativa de metadatos.

El número de particiones está relacionado con el uso de la memoria, y un problema de MemoryOverhead se puede relacionar con este número (experiencia personal).

Un **error común** para los nuevos usuarios es transformar su RDD en un RDD con una sola partición, que generalmente se ve así:

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

¡Por lo general, es una muy mala idea, ya que le está diciendo a Spark que ponga **todos los datos en** una sola partición! Recuérdalo:

> Una etapa en Spark operará en una partición a la vez (y cargará los datos en esa partición en la memoria).

Como resultado, le dice a Spark que maneje todos los datos a la vez, lo que generalmente resulta en errores relacionados con la memoria (memoria insuficiente, por ejemplo), o incluso una excepción de puntero nulo.

Por lo tanto, a menos que sepa lo que está haciendo, ¡evite volver a particionar su RDD en una sola partición!

Examples

Particiones Intro

> ¿Cómo se divide un RDD?

De forma predeterminada, se crea una partición para cada partición HDFS, que de forma

predeterminada es de 64 MB. Lea más aquí .

> ¿Cómo equilibrar mis datos a través de particiones?

Primero, eche un vistazo a las tres formas en que uno puede *repartir* sus datos:

1. Pase un segundo parámetro, el número *mínimo* deseado de particiones para su RDD, en textFile () , pero tenga cuidado:

   En [14]: líneas = sc.textFile ("datos")

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

   En [16]: líneas = sc.textFile ("datos", 500)

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

   En [18]: líneas = sc.textFile ("datos", 5000)

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

Como puede ver, `[16]` no hace lo que uno esperaría, ya que la cantidad de particiones que tiene el RDD ya es mayor que la cantidad mínima de particiones que solicitamos.

2. Use repartition () , así:

   En [22]: líneas = líneas. Reparto (10)

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

Advertencia: Esto invocará un orden aleatorio y debe usarse cuando desee **aumentar** el número de particiones que tiene su RDD.

De la documentación :

> El orden aleatorio es el mecanismo de Spark para redistribuir los datos de modo que se agrupen de manera diferente entre las particiones. Por lo general, esto implica copiar datos a través de ejecutores y máquinas, lo que hace que el orden aleatorio sea una operación compleja y costosa.

3. Use coalesce () , así:

   En [25]: líneas = lines.coalesce (2)

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

Aquí, Spark sabe que va a reducir el RDD y se aprovecha de él. Lea más acerca de repartition () vs coalesce () .

¿Pero todo esto **garantizará** que sus datos estarán perfectamente equilibrados en sus particiones? Realmente no, como lo experimenté en ¿Cómo equilibrar mis datos en las

particiones?

Particiones de un RDD

Como se mencionó en "Comentarios", una partición es una parte / sector / segmento de un RDD. A continuación se muestra un ejemplo mínimo sobre cómo solicitar un número mínimo de particiones para su RDD:

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

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

Observe en [1] cómo pasamos 2 como segundo parámetro de `parallelize()` . Ese parámetro dice que queremos que nuestro RDD tenga al menos 2 particiones.

Repartir una RDD

A veces queremos repartir una RDD, por ejemplo, porque proviene de un archivo que no creamos nosotros, y la cantidad de particiones definidas por el creador no es la que queremos.

Las dos funciones más conocidas para lograrlo son:

```
repartition(numPartitions)
```

y:

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

Como regla general, use el primero cuando desee volver a particionar su RDD en un mayor número de particiones y el segundo para reducir su RDD, en un número menor de particiones. Spark - repartition () vs coalesce () .

Por ejemplo:

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

disminuirá el número de particiones del RDD llamado 'datos' a 100, dado que este RDD tiene más de 100 particiones cuando fue leído por `textFile()` .

Y de manera similar, si desea tener más del número actual de particiones para su RDD, podría hacerlo (dado que su RDD se distribuye en 200 particiones, por ejemplo):

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

Regla de oro sobre el número de particiones

Como regla general, uno querría que su RDD tenga tantas particiones como el producto del número de ejecutores por el número de núcleos utilizados por 3 (o quizás 4). Por supuesto, eso es una heurística y realmente depende de su aplicación, conjunto de datos y configuración de clúster.

Ejemplo:

```
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)
```

Mostrar contenidos RDD

Para mostrar el contenido de un RDD, se debe imprimir:

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

Para limitar el número de filas impresas:

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

Lea Particiones en línea: https://riptutorial.com/es/apache-spark/topic/5822/particiones

Capítulo 15: Pruebas unitarias

Examples

Prueba de unidad de conteo de palabras (Scala + JUnit)

Por ejemplo, tenemos `WordCountService` con el método `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
    }
}
```

Este servicio parece muy feo y no está adaptado para pruebas unitarias. SparkContext debe ser inyectado a este servicio. Puede alcanzarse con su marco DI favorito, pero por simplicidad se implementará utilizando el constructor:

```
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
    }
}
```

Ahora podemos crear una prueba JUnit simple e inyectar sparkContext comprobable a 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)
    }
}
```

Lea Pruebas unitarias en línea: https://riptutorial.com/es/apache-spark/topic/3333/pruebas-unitarias

Capítulo 16: Se une

Observaciones

Una cosa a tener en cuenta son sus recursos frente al tamaño de los datos a los que se une. Aquí es donde su código de Spark Join puede fallar y le da errores de memoria. Por este motivo, asegúrese de configurar sus trabajos Spark realmente bien dependiendo del tamaño de los datos. A continuación se muestra un ejemplo de una configuración para una unión de 1.5 millones a 200 millones.

Usando Spark-Shell

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

Usando Spark Submit

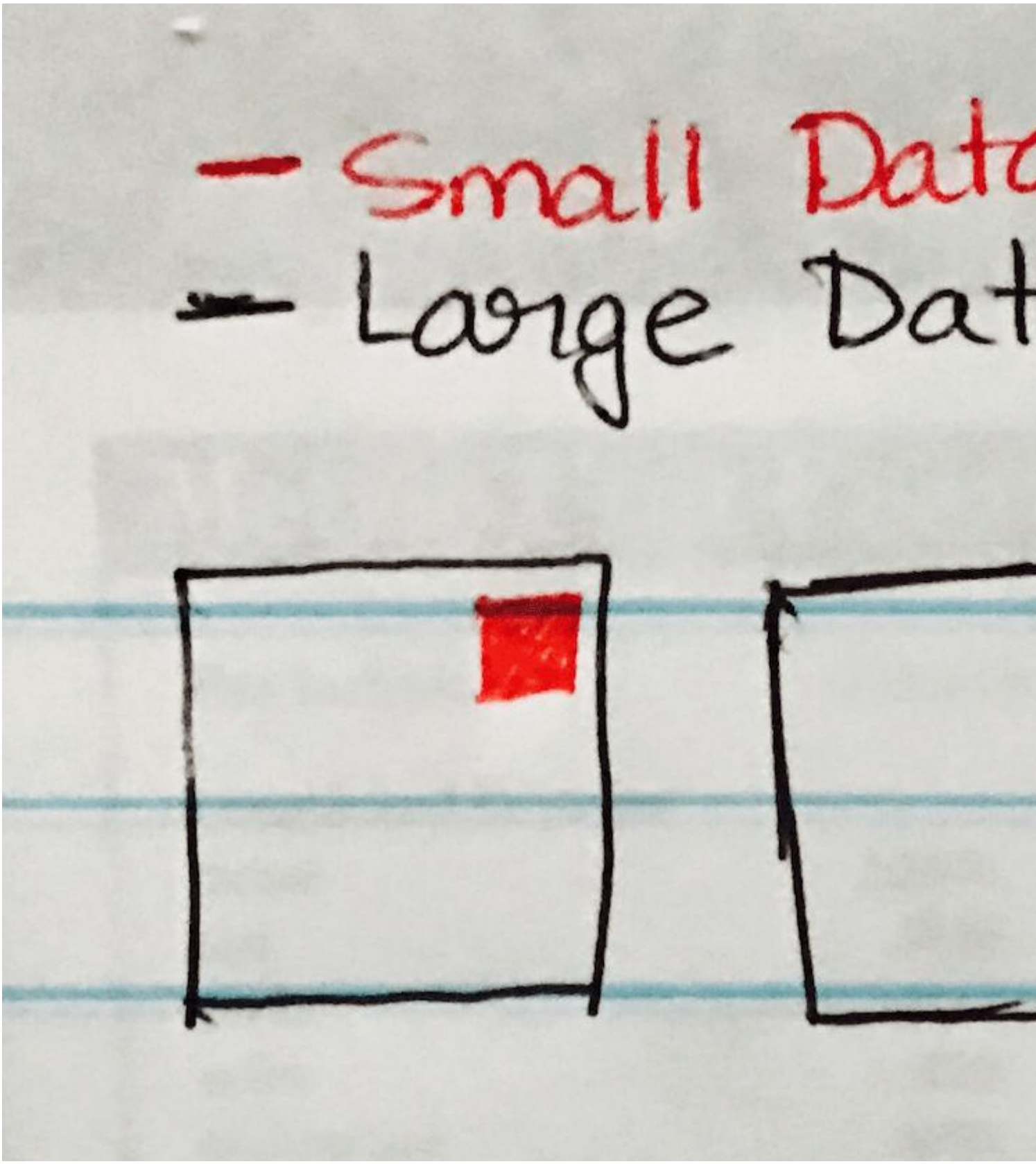
```
spark-submit   --executor-memory 32G   --num-executors 80  --driver-memory 10g --executor-
cores 10 code.jar
```

Examples

Broadcast Hash Join en Spark

Una combinación de difusión copia los datos pequeños en los nodos de trabajo, lo que conduce a una combinación altamente eficiente y súper rápida. Cuando nos unimos a dos conjuntos de datos y uno de los conjuntos de datos es mucho más pequeño que el otro (por ejemplo, cuando el conjunto de datos pequeño puede caber en la memoria), entonces deberíamos usar una combinación de hash de difusión.

La siguiente imagen visualiza una Emisión de hash de difusión cuando el conjunto de datos pequeño se transmite a cada partición del Conjunto de datos grande.



A continuación se muestra un ejemplo de código que puede implementar fácilmente si tiene un escenario similar de una combinación de datos grande y pequeña.

```
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")
```

Lea Se une en línea: https://riptutorial.com/es/apache-spark/topic/7828/se-une

Capítulo 17: Spark DataFrame

Introducción

Un DataFrame es una abstracción de datos organizados en filas y columnas escritas. Es similar a los datos encontrados en bases de datos relacionales basadas en SQL. Aunque se ha transformado en solo un alias de tipo para Dataset [Fila] en Spark 2.0, todavía es ampliamente utilizado y útil para procesos de procesamiento complejos que hacen uso de la flexibilidad de su esquema y las operaciones basadas en SQL.

Examples

Creando DataFrames en Scala

Hay muchas formas de crear DataFrames. Se pueden crear a partir de listas locales, RDD distribuidos o leer de fuentes de datos.

Usando toDF

Al importar implícitos de spark sql, se puede crear un DataFrame desde un Seq, Array o RDD local, siempre que el contenido sea de un subtipo de Producto (las tuplas y las clases de casos son ejemplos bien conocidos de subtipos de Producto). Por ejemplo:

```
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")
```

Utilizando createDataFrame

Otra opción es usar el método `createDataFrame` presente en SQLcontext. Esta opción también permite la creación desde listas locales o RDD de subtipos de productos como con `toDF` , pero los nombres de las columnas no se configuran en el mismo paso. Por ejemplo:

```
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"))
))
```

Además, este enfoque permite la creación a partir de RDD de instancias de `Row` , siempre que se pase un parámetro de `schema` para la definición del esquema del marco de datos resultante. Ejemplo:

```
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)
```

Lectura de fuentes

Tal vez la forma más común de crear DataFrame es a partir de fuentes de datos. Uno puede crearlo a partir de un archivo de parquet en hdfs, por ejemplo:

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

Lea Spark DataFrame en línea: https://riptutorial.com/es/apache-spark/topic/8358/spark-dataframe

Capítulo 18: Variables compartidas

Examples

Variables de difusión

Las variables de difusión son solo objetos compartidos de lectura que se pueden crear con el método `SparkContext.broadcast` :

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

y leer usando el método de `value` :

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

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

Acumuladores

Los acumuladores son variables de solo escritura que se pueden crear con `SparkContext.accumulator` :

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

modificado con `+=` :

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

y accedido con método de `value` :

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

El uso de acumuladores se complica por la garantía de Spark de ejecutar por lo menos una vez para transformaciones. Si es necesario volver a calcular una transformación por cualquier motivo, las actualizaciones del acumulador durante esa transformación se repetirán. Esto significa que los valores del acumulador pueden ser muy diferentes de lo que serían si las tareas se ejecutaran solo una vez.

Nota:

1. Los ejecutores *no pueden* leer el valor del acumulador. Solo el programa controlador puede leer el valor del acumulador, utilizando su método de valor.

2. Es casi similar a contrarrestar en Java / MapReduce. Para que puedas relacionar los acumuladores con los contadores y entenderlos fácilmente.

Acumulador definido por el usuario en Scala

Define `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
}
```

Utilizar:

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

Acumulador definido por el usuario en Python

Define `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)
```

Lea Variables compartidas en línea: https://riptutorial.com/es/apache-spark/topic/1736/variables-compartidas

Creditos

S. No	Capítulos	Contributors
1	Empezando con la chispa de apache	4444, Ani Menon, Community, Daniel de Paula, David, gsamaras, himanshuIIITian, Jacek Laskowski, KartikKannapur, Naresh Kumar, user8371915, zero323
2	¿Cómo hacer una pregunta relacionada con Apache Spark?	user7337271
3	Archivos de texto y operaciones en Scala	Ani Menon, Community, spiffman
4	Configuración: Apache Spark SQL	himanshuIIITian
5	El mensaje de error 'sparkR' no se reconoce como un comando interno o externo o '.binsparkR' no se reconoce como un comando interno o externo	Rajesh
6	Funciones de ventana en Spark SQL	Daniel Argüelles, Hari, Joshua Weinstein, Tejus Prasad, vdep
7	Introducción a Apache Spark DataFrames	Mandeep Lohan, Nayan Sharma
8	Lanzador de chispas	Ankit Agrahari
9	Llamando scala empleos desde pyspark	eliasah, Thiago Baldim
10	Manejo de JSON en Spark	Furkan Varol, zero323

11	Migración de Spark 1.6 a Spark 2.0	Béatrice Moissinac, eliasah, Shaido
12	Modo cliente y modo clúster	Nayan Sharma
13	Operaciones con estado en Spark Streaming	zero323
14	Particiones	Ani Menon, Armin Braun, gsamaras
15	Pruebas unitarias	Cortwave
16	Se une	Adnan, CGritton
17	Spark DataFrame	Daniel de Paula
18	Variables compartidas	Community, Jonathan Taws, RBanerjee, saranvisa, spiffman, whaleberg, zero323