Computación distribuida

Sergio Nesmachnow
(sergion@fing.edu.uy)

UNIVERSIDAD
DE LA REPÚBLICA
URUGUAY

UNIVERSIDAD DE LA REPÚBLICA URUGUAY

Contenido

1. Computación distribuida y computación cloud
2. Procesamiento de grandes volúmenes de datos
3. El modelo de computación Map-Reduce
4. El framework Hadoop y su ecosistema
5. Almacenamiento: HDFS y HBase.
6. Aplicaciones de Map Reduce sobre Hadoop: conteo, índice invertido, filtros
7. Procesamiento de datos con Apache Spark
8. Ejemplos de aplicaciones en Spark y el lenguaje Scala
9. Análisis de datos utilizando Spark y el lenguaje R
10. Aplicaciones iterativas: Google Pregel y Apache Giraph

Apache Spark: entrada y salida

- En modo interactivo: datos de coleciones en memoria y de archivos regulares.
- Sin embargo, son necesarias otras opciones para los casos en que los datos no entran en la memoria de una máquina.
- Entradas y salidas soportadas por Spark:
  - Interfaces InputFormat y OutputFormat de Hadoop MapReduce en diferentes formatos y desde diferentes repositorios (DFS, S3, Hbase, etc.)
  - Archivos y filesystems: locales (NFS, HDFS, etc.) o distribuidos (Amazon S3, etc.) en formatos de texto, JSON, SequenceFiles, archivos comprimidos, etc.
  - Repositorios de datos estructurados a través de Spark SQL, en formatos csv, JSON, Apache Hive, etc.
  - Bases de datos y repositorios en formato clave-valor: Cassandra, HBase, Elasticsearch, conectores JDBC, etc.

Apache Spark: entrada y salida

- Archivos de texto: textFile y saveAsTextFile
  - Python `input = sc.textFile("file:///home/sparkuser/README.md")`
  - Scala `val input = sc.textFile("file:///home/sparkuser/README.md")`
- Múltiples archivos:
  - Se pueden manejar con textFile sobre un directorio, que carga todos los archivos en un RDD. Soporta wildcards (e.g., part-*.txt).
  - Para saber a qué archivo corresponde cada parte de la entrada (por ejemplo, la fecha podría ser parte del nombre del archivo) o para procesar los archivos por separado: sc.wholeTextFiles() (para archivos pequeños) crea un pair RDD con el nombre del archivo como clave.
  - Ejemplo: promedio de ventas por año:

```
val input = sc.wholeTextFiles("file://home/sparkuser/salesFiles")
val result = input.mapValues{y =>
        val nums = y.split(" ").map(x => x.toDouble)
        nums.sum / nums.size.toDouble
}
```

Apache Spark: entrada y salida

- Archivos en formato estructurado, dos opciones:
  1. Tratarlos como texto y parsear el contenido (como se presentó previamente para archivos JSON).
  2. Usar una biblioteca de serialización para convertir valores a strings.
     - JSON: en Scala y Java se puede usar el formato predefinido de Hadoop.
     - csv: en Python la biblioteca csv, en Scala y Java la biblioteca opencsv. En Scala y Java el InputFormat CSVInputFormat de Hadoop.

```python
import csv
import StringIO
...
def loadRecord(line): """Parsear una línea del archivo CSV"""
    input = StringIO.StringIO(line)
    reader = csv.DictReader(input, fieldnames=["name", "favAnimal"])
    return reader.next()
input = sc.textFile(inputFile).map(loadRecord)
```

- Salvar archivos de texto: saveAsTextFile(outputpath)
  - Escribe multiples archivos en el directorio outputpath (desde diferentes nodos). No permite controlar qué segmentos se escriben en qué archivos.

Apache Spark: entrada y salida

- SequenceFiles: archivos con pares clave-valor que implementan la interfaz Writable de Hadoop. Pueden ser procesados eficientemente en paralelo usando marcas de sincronización de Hadoop.
- Leer datos: sc.sequenceFile(path, keyClass, valueClass, minPartitions)
  - keyClass y valueClass deben ser clases Writable.
  - Ejemplo: Cargar identificador y número de pandas vistos por personas.
  - Key Class: Text, valueClass: IntWritable

```
data = sc.sequenceFile(inFile, "org.apache.hadoop.io.Text",
        "org.apache.hadoop.io.IntWritable")
```

(Python)

```
val data = sc.sequenceFile(inFile, classOf[Text], classOf[IntWritable]).
        map{case (x, y) =>(x.toString,y.get())}
```

(Scala)

- Salvar datos a SequenceFile: saveAsSequenceFile(path)
  - Si es necesario: map sobre los datos para convertir los tipos.

```
doc = sc.textFile("file:///opt/spark/data/shakespeare.txt")
words = doc.flatMap(lambda x: x.split()).keyBy(lambda x: x)
words.saveAsSequenceFile("file:///opt/spark/data/seqfiles")
```

Spark: entrada y salida con tipos de Hadoop

- newAPIHadoopFile(path, inputFormatClass, keyClass, valueClass)
  - Ejemplo: KeyValueTextInputFormat para leer pares clave-valor. Cada línea se procesa individualmente (clave y valor separados por tabulador).

```
val input = sc.newAPIHadoopFile("Hadoop_input",
classOf[KeyValueTextInputFormat], classOf[Text],classOf[IntWritable])
```

- RDD.saveAsNewAPIHadoopFile(path, outputFormatClass, keyClass, valueClass[, keyConverter, valueConverter, conf])
  - Salva un pair RDD a HDFS usando la API de MapReduce y sus outputFormats

```
cls = "org.apache.hadoop.mapreduce.lib.output.SequenceFileOutputFormat"
doc = sc.textFile("hdfs://localhost:8020/input/shakespeare.txt")
words = doc.flatMap(lambda x: x.split()).map(lambda x: (x,1)) \
    .reduceByKey(lambda x, y: x + y)
words.saveAsNewAPIHadoopFile("hdfs://localhost:8020/output/newAPIfiles",
    outputFormatClass=cls, keyClass="org.apache.hadoop.io.Text",
    valueClass="org.apache.hadoop.io.IntWritable", keyConverter=None,
    valueConverter=None, conf=None)
```

Spark: entrada y salida con tipos de Hadoop

- saveAsNewAPIHadoopDataset: Guarda RDDs a repositorios clave-valor, como HBase (con la configuración de conexión necesaria).
- Ejemplo: guardar datos en HBase
  1. Crear una tabla en HBase, con una única familia cf1 y cargar dos registros

```
$ hbase shell
hbase> create 'people', 'cf1’
hbase> put 'people', 'userid1', 'cf1:fname', 'John'
hbase> put 'people', 'userid1', 'cf1:lname', 'Doe'
hbase> put 'people', 'userid1', 'cf1:age', '41'
hbase> put 'people', 'userid2', 'cf1:fname', 'Jeffrey'
hbase> put 'people', 'userid2', 'cf1:lname', 'Aven'
hbase> put 'people', 'userid2', 'cf1:age', '46'
hbase> put 'people', 'userid2', 'cf1:city', 'Hayward’
```

  2. Los datos se pueden visualizar con scan

```
hbase> scan 'people'
ROW COLUMN+CELL
userid1 column=cf1:age, timestamp=1461296454933, value=41
...
```

Spark: entrada y salida con tipos de Hadoop

3. Ejecutar pyspark con los jar de Hbase y Hadoop. En Cloudera:

```
pyspark --master local --driver-class-path "/usr/lib/hbase/*:
/usr/lib/hbase/lib/*:/home/cloudera/hbase-examples-1.2.0-
cdh5.10.0.jar:/usr/lib/spark/lib/spark-examples-1.6.0-cdh5.12.0-
hadoop2.6.0-cdh5.12.0.jar"
```

4. Leer los datos de la tabla HBase

```
conf = {"hbase.zookeeper.quorum": "localhost", "hbase.mapreduce.inputtable":
            "people"}
keyConv = "org.apache.spark.examples.pythonconverters.
            ImmutableBytesWritableToStringConverter"
valueConv =
"org.apache.spark.examples.pythonconverters.HBaseResultToStringConverter"
hbase_rdd =
sc.newAPIHadoopRDD("org.apache.hadoop.hbase.mapreduce.TableInputFormat",
"org.apache.hadoop.hbase.io.ImmutableBytesWritable","org.apache.hadoop.hbase
.client.Result",keyConverter=keyConv, valueConverter=valueConv, conf=conf)
hbase_rdd.collect()
```

Salida: `[(u'userid1',u'{"qualifier":"age","timestamp": ...)..]`

Spark: entrada y salida con tipos de Hadoop

5. Crear un RDD de usuario y salvar el contenido a la tabla people de HBase

```
conf2 = {"hbase.zookeeper.quorum":"localhost","hbase.mapred.outputtable":
"people","mapreduce.outputformat.class":"org.apache.hadoop.hbase.mapreduce.
TableOutputFormat","mapreduce.job.output.key.class":"org.apache.hadoop.hbase
.io.ImmutableBytesWritable","mapreduce.job.output.value.class":
"org.apache.hadoop.io.Writable"}
keyConv2 = "org.apache.spark.examples.pythonconverters.
    StringToImmutableBytesWritableConverter"
valueConv2 =
    "org.apache.spark.examples.pythonconverters.StringListToPutConverter"
newpeople = sc.parallelize([('userid3',['userid3','cf1','fname','NewUser'])])
newpeople.saveAsNewAPIHadoopDataset(conf=conf2,keyConverter=keyConv2,
valueConverter=valueConv2)
```

6. Verificar que el valor se agregó, con scan sobre la tabla people

```
hbase> scan 'people'
ROW COLUMN+CELL
userid1 column=cf1:age, timestamp=1461296454933, value=41
...
userid3 column=cf1:fname, timestamp=146..., value=NewUser
```

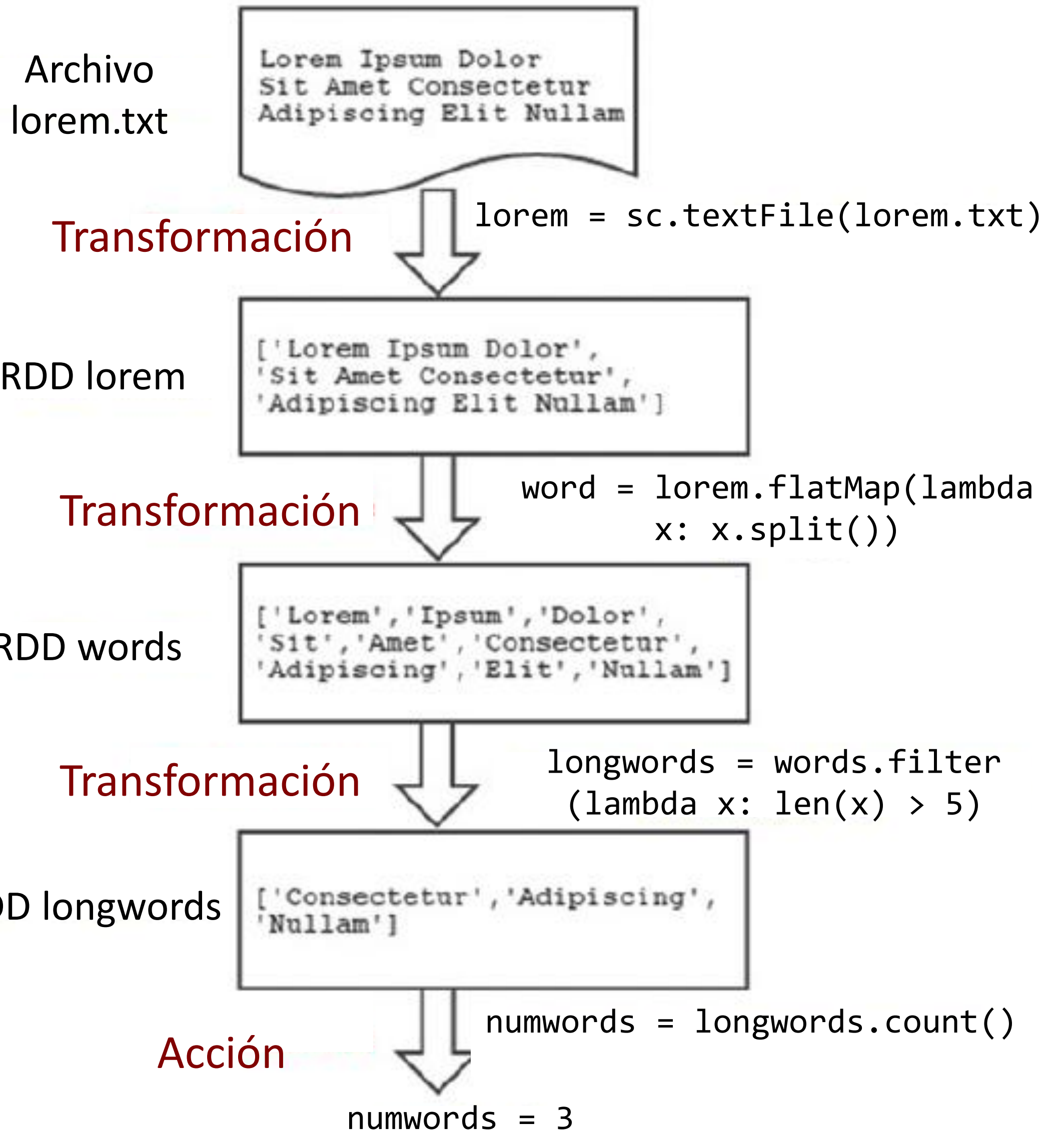
Spark: entrada y salida con tipos de Hadoop

- Existen conectores para HBase desde Spark usando la API de Scala, por ejemplo la disponible en https://github.com/nerdammer/spark-hbase-connector y otras.
- También es possible interactuar con otros sistemas de almacenamiento NoSQL, incluyendo Cassandra, DynamoDB, MongoDB y otras.
- La mayoría de estos sistemas tienen implementaciones de clases InputFormat y OutputFormat en Hadoop que permiten leer y escribir datos de RDD de Spark
  - Por ejemplo, las clases DynamoInputFormat y DynamoDBOutputFormat disponibles en Amazon.

Apache Spark: persistencia

- Persistir RDDs para evitar su reevaluación: permite mejorar el desempeño.
- Permite incrementar la eficiencia hasta en un factor de 10×
- Niveles de almacenamiento: los RDDs no siempre residen en memoria en los worker nodes, existen varios niveles para persistencia.
- Recordar un concepto clave: RDD lineage.
  – Spark planea la ejecución de un programa como un DAG (tareas y dependencias). Algunas operaciones (map) pueden ser completamente paralelizadas y otras (reduceByKey) requieren shuffle, generando dependencias.
  – Lineage: serie de transformaciones realizadas a un RDD, permite la reevaluación del RDD en caso de falla.

Apache Spark: persistencia



Resumen del plan de ejecución con toDebugString()

```
print(longwords.toDebugString())
(1) PythonRDD[6] at collect at
<stdin>:1 []
| MapPartitionsRDD[1] at textFile at
..[]
| file://lorem.txt HadoopRDD[0] at
textFile at ..[]
```

La acción longwords.count() fuerza la evaluación de cada ancestro de longwords. SI esta acción (u otra como take() o collect()) se invoca, el lineage complete es reevaluado.

Impacta en el desempeño

Apache Spark: persistencia

- RDDs se almacenan en particiones en varios nodos del cluster (YARN/Mesos, etc.).
- Seis niveles de almacenamiento:
  - MEMORY_ONLY (por defecto): las particiones se almacenan en memoria únicamente.
  - MEMORY_AND_DISK: las particiones que no entran en memoria se almacenan en disco.
  - MEMORY_ONLY_SER: las particiones se almacenan en memoria como objetos serializados (ahorra memoria)
  - MEMORY_AND_DISK_SER: las particiones se almacenan como objetos serializados en memoria y las que no entran se almacenan en disco.
- Existen opciones con replicación, que almacenan las particiones en diferentes nodos del cluster (mejora tolerancia a fallos, pero consume más espacio y puede afectar el desempeño)

Apache Spark: persistencia

- Sugerencias sobre niveles de almacenamiento:
  - Si los RDDs entran en memoria, usar el valor por defecto (MEMORY_ONLY), que prove la opción más eficiente.
  - En caso contrario, intentar con MEMORY_ONLY_SER con una biblioteca de serialización eficiente que permita reducir el tamaño de los objetos pero a la vez que sean razonablemente rápidos de accede (en Scala y Java).
  - No almacenar en disco salvo que las funciones que computan los RDDs sean realmente costosas o filtran grandes volumes de datos. En otro caso, posiblemente sea más eficiente recomputar una partición que leerla de disco.
  - Utilizar los niveles con replicación para proveer tolerancia a fallos (por ejemplo, al servir pedidos en una aplicación web). Todos los niveles de almacenamiento permiten recalcular datos perdidos, pero los que incluyen replicación permiten continuar ejecutando tareas sin tener que esperar a recalcular particiones.

Apache Spark: persistencia

- RDD.persist(storageLevel=StorageLevel.MEMORY_ONLY_SER)
  - Especifica el nivel de almacenamiento para el RDD, que será realizado la primera vez que el RDD sea evaluado.
  - Si no es posible (por ejemplo no hay memoria suficiente para persistir el RDD), Spark revierte al comportamiento normal y retiene solamente las particiones necesarias en memoria.
  - storageLevel se expresa como constante estática o como conjunto de flags.

```
myrdd.persist(StorageLevel.MEMORY_AND_DISK_SER_2)
myrdd.persist(StorageLevel(True, True, False, False, 2))
```

- El nivel de almacenamiento por defecto es MEMORY_ONLY_SER.
- RDDs persistidos se ven en la Spark Application UI (Storage Tab).
- Ejemplo:

```
from pyspark.storagelevel import StorageLevel
rdd = sc.parallelize([1,2,3,4,5,6])
rdd.persist(StorageLevel.MEMORY_AND_DISK)
```

Apache Spark: persistencia

- Cahear RDDs: persiste los datos en memoria, para ser reutilizados sin tener que recalcular todo el lineage.
- RDD.cache(): no activa el cálculo del RDD, es una sugerencia para cuando se realice una acción. Si no hay memoria suficiente para cachear, el RDD se reevaluará en cada acción que se realice.
- Cache nunca persiste a disco, solamente a memoria (MEMORY_ONLY_SER storage level).

```
doc = sc.textFile("file:///opt/spark/data/shakespeare.txt")
words = doc.flatMap(lambda x: x.split()).map(lambda x: (x,1)) \
        .reduceByKey(lambda x, y: x + y)
words.cache()
words.count() # computa el RDD
# 33505
words.take(3) # no require computar el RDD
# [(u'fawn', 12), (u'mustachio', 1), (u'Debts', 1)]
words.count() # no require computar el RDD
# 33505
```

Apache Spark: persistencia

- RDD.unpersist(): elimina la persistencia de un RDD que ya no se utilizará.
  - Es necesario aplicarlo para cambiar opciones de almacenamiento (sinó, ocurre la excepción "Cannot change storage level of an RDD after it was already assigned a level").
- Ejemplo de persistencia:

```
doc = sc.textFile("file:///opt/spark/data/shakespeare.txt")
words = doc.flatMap(lambda x: x.split()).map(lambda x: (x,1)) \
        .reduceByKey(lambda x, y: x + y)
words.persist()
words.take(3)
print(words.toDebugString())
(1) PythonRDD[8] at ... [Memory Serialized 1x Replicated]
| MapPartitionsRDD[5] at ... [Memory Serialized 1x Replicated]
| ShuffledRDD[4] at ... [Memory Serialized 1x Replicated]
+-(1) PairwiseRDD[3] at ... [Memory Serialized 1x Replicated]
| PythonRDD[2] at ...[Memory Serialized 1x Replicated]
| MapPartitionsRDD[1] ... [Memory Serialized 1x Replicated]
| HadoopRDD[0] ... [Memory Serialized 1x Replicated]
```

Apache Spark: persistencia

- Checkpointing: salva datos a un archivo.
- Diferencia principal con persistir a disco: el archivo persiste aún cuando el driver de la aplicación haya finalizado. Puede ser usado para recuperar ante un fallo o como entrada para otros programas.
- Checkpointing permite olvidar el lineage (complicado en programas complejos). Usar un filesystem (por ejemplo, HDFS) prove mayor nivel de tolerancia a fallos. Checkpointing solo se realiza luego de una acción.
- RDD.checkpoint(): marca el RDD para checkpointing, que se realizará con la primera acción ejecutada en el RDD. Los archivos se almacenarán en el directorio especificado previamente con setCheckpointDir().
- Luego de que el checkpointing finaliza exitosamente, todo el lineage del RDD se elimina.

Apache Spark: persistencia

- Checkpointing: ejemplo de salvar datos a un archivo.

```
sc.setCheckpointDir('file:///opt/spark/data/checkpoint')
doc = sc.textFile("file:///opt/spark/data/shakespeare.txt")
words = doc.flatMap(lambda x: x.split()).map(lambda x: (x,1)) \
        .reduceByKey(lambda x, y: x + y)
words.checkpoint()
words.count()
...
... Done checkpointing RDD to ..., new parent is RDD 8
33505
words.isCheckpointed()
True
words.getCheckpointFile()
u'file:/.../d7a8269e-cbed-422f-bf1a-87f3853bd218/rdd-6'
```

Spark: paralelismo personalizado

- En Spark, cada RDD tiene un número fijo de particiones que determinan el nivel de paralelismo en las operaciones sobre el RDD.
- Como se vio, en las agregaciones y agrupamientos puede especificarse un número de particiones a utilizar al crear el RDD agrupado o agregado.

```
data = [("a", 3), ("b", 4), ("a", 1)]
```

  – Paralelismo por defecto

```
sc.parallelize(data).reduceByKey(lambda x, y: x + y)
```

  – Paralelismo personalizado, crea 10 particiones para procesar independientemente

```
sc.parallelize(data).reduceByKey(lambda x, y: x + y, 10)
```

Spark: paralelismo personalizado

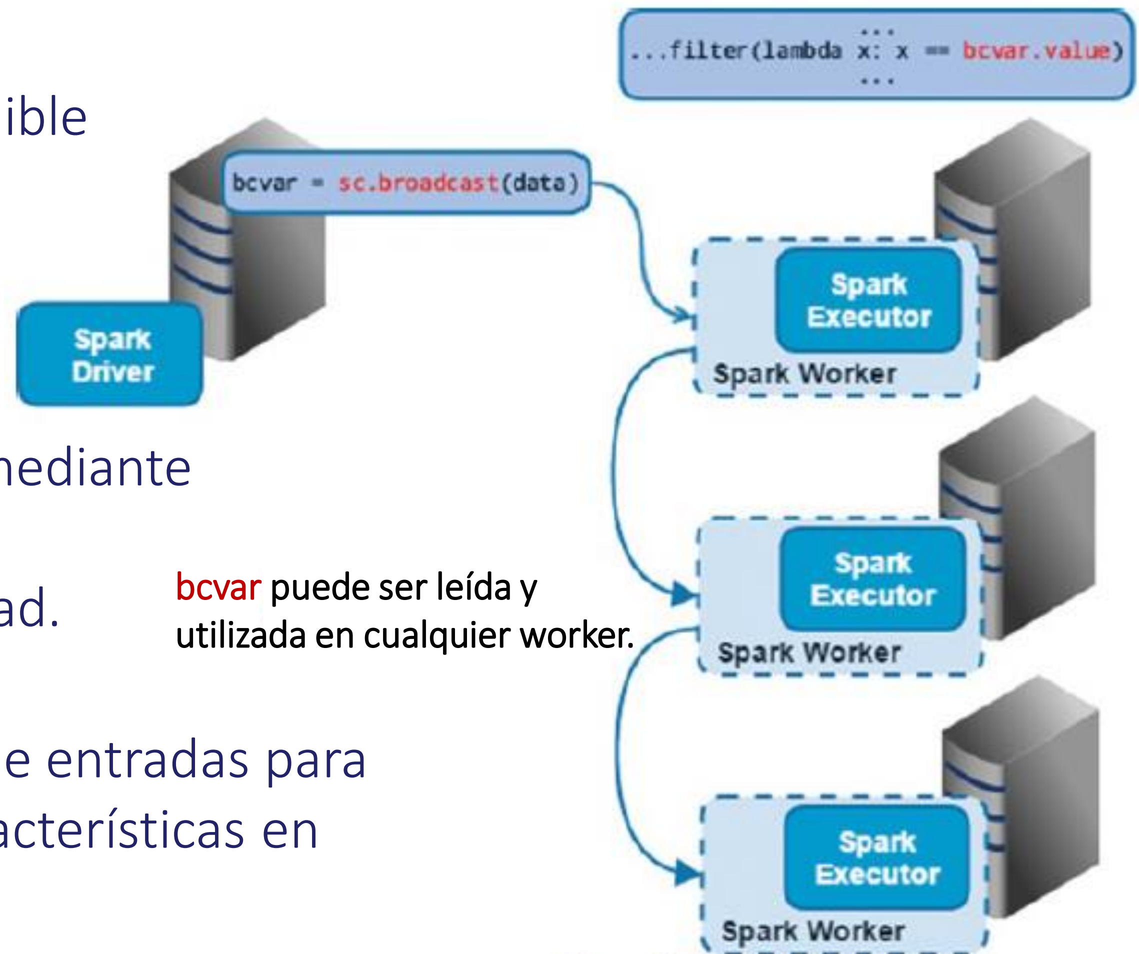
- repartition(): modifica el particionamiento de un RDD por fuera de una operación de agregación o agrupamiento. Realiza un shuffle de datos (a través de la red) creando nuevas particiones. Debe tenerse en cuenta que es una operación costosa en términos de desempeño.
- coalesce(): version optimizada de repartition() para reducer el número de particiones sin realizar movimiento de datos.
  - Chequear el número de particiones de un RDD con rdd.partitions.size() en Scala/Java o rdd.getNumPartitions() en Python y ejecutar coalesce() con seguridad, para reducir el número de particiones.

Apache Spark: variables compartidas

- Una función que se pasa a Spark (e.g., map(), reduce() o filter()) se ejecuta en un nodo remoto del cluster utilizando copias locales de las variables en la función (definidas en el driver).
  - Las variables se copian en cada host y no se actualiza ningún valor ni se propaga al driver, porque mantener variables compartidas para lectura/escritura entre tareas es ineficiente.
  - Spark automáticamente envía todas las variables en los closures a los workers. Inconveniente: una variable que se utiliza en multiples operaciones paralelas es enviada multiples veces (por separado) para cada operación.
- Spark prove soporte para dos tipos de variables compartidas: broadcast y acumuladores.
  - Permiten distribuir información de modo eficiente.
  - Cubren dos patrones muy frecuentemente usados en las aplicaciones distribuidas: difusión de datos y agregaciones.

Apache Spark: broadcast y acumuladores

- Broadcast: variables de *solo lectura* creadas por el driver de Spark para enviar información de modo eficiente a todos los workers en el cluster.
  - La información queda disponible para todas las tareas en los executors en cada worker.
  - Las variables se comparten mediante un protocolo P2P eficiente, permitiendo gran escalabilidad.
  - Útiles para enviar una tabla de entradas para búsqueda o un vector de características en un método de aprendizaje.



Spark: broadcast

- SparkContext.broadcast(value): crea una instancia de un objeto broadcast en el contexto. El valor se serializa y encapsula en el objeto broadcast. La variable queda disponible para todas las tareas en la aplicación.

```
stations = sc.broadcast({'83':'Mezes Park', '84':'Ryland Park’})
stations
<pyspark.broadcast.Broadcast object at 0x...>
```

- Broadcast también pueden crearse con el contenido de un archivo (local, en red o en filesystem distribuido).

```
# stations.csv data:
# 83,Mezes Park,37.491269,-122.236234,15,Redwood City,2/20/2014
# 84,Ryland Park,37.342725,-121.895617,15,San Jose,4/9/2014
stationsfile = '/home-cloudera/stations.csv' // Cargar archivo csv
stationsdata = dict(map(lambda x:(x[0],x[1]), map(lambda x: x.split(',’), \
        open(stationsfile)))) // Crear diccionario de pares (id, nombre)
stations = sc.broadcast(stationsdata)
stations.value["83"] // stationsdata es accesible para todos los workers
'Mezes Park'
```

Spark: broadcast

- Broadcast.value(): retorna el valor de una variable broadcast, que puede ser usado en expresiones (funciones lambda, map, etc. en el programa).
- Broadcast.unpersist(blocking=False): remueve una variable broadcast de la memoria de todos los workers del cluster donde fue definida.
- Blocking indica si la operación se bloquea hasta que la variable se elimina de todos los nodos o si se ejecuta en modo asincrónico.

```
stations = sc.broadcast({'83':'Mezes Park', '84':'Ryland Park'})
stations.unpersist()
```

- Opciones de configuración para variables broadcast

  spark.broadcast.compress: indica si comprimir la variable broadcast antes de enviarla a los workers. Por defecto es true (recomendado).

  spark.broadcast.factory: implementación de broadcast a utilizar. Por defecto es TorentBroadcastFactory (recomendado).

  spark.broadcast.blockSize: tamaño de bloque de variables broadcast. Por defecto 4 MB.

  spark.broadcast.port: puerto en que escucha el servidor broadcast HTTP del driver.

Spark: broadcast

- Ventajas de usar broadcast:
  - Método eficiente para compartir datos en tiempo de ejecución.
  - Permiten eliminar shuffles costosos.
  - Utilizan un protocolo P2P eficiente y escalable.
  - Replican los datos una vez por worker (y no una vez por tarea): 10s vs 1000s.
  - Pueden ser reutilizados por multiples tareas.
  - Son objetos serializados, su lectura es eficiente.

- Ejemplo: combinar datos de sistemas de bicicletas: stations (dataset de búsqueda, pequeño) y status (fuente de datos de eventos, grande) por clave station_id.
  - Stations: station_id, nombre, latitud, longitud, docks instalados, ciudad, fecha de instalación
  - Status: station_id, bicicletas disponibles, docks disponibles, fecha-hora

Spark: broadcast

1. Combinar con un join en la aplicación

```
status = sc.textFile('home/cloudera/status.csv’) \
  .map(lambda x: x.split(',’)) \
  .keyBy(lambda x: x[0])

stations = sc.textFile('home/cloudera/stations.csv’) \
  .map(lambda x: x.split(',’)) \
  .keyBy(lambda x: x[0])

joined = status.join(stations)
```

– Retorna [(clave,([registro 1],[registro 2]),(clave,([registro 1],[registro 2]), ...]

```
joined.map(lambda x:(x[1][0][3],x[1][1][1],x[1][0][1],x[1][0][2])).take(3)
```

– Retorna [(fecha-hora, nombre, bicicletas disponibles, docks disponibles)]

```
[("2015-01-09 17:02:02", 'Embarcadero at Sansome', '7', '8'),
("2014-09-01 00:00:03", 'Embarcadero at Sansome', '7', '8'),
("2014-09-01 00:01:02", 'Embarcadero at Sansome', '7', '8’)]
```

– Implica una operación de shuffle muy costosa.

Spark: broadcast

2. Versión mejorada: definir una variable (Python) en el driver para stations, que estará disponible como variable en tiempo de ejecución para tareas Spark que implementan operaciones map (no requiere shuffle).

```
stationsfile = 'stations.csv'
sdata = dict(map(lambda x: (x[0],x[1]), \
  map(lambda x: x.split(','),open(stationsfile))))
status = sc.textFile('home/cloudera/status.csv’) \
  .map(lambda x: x.split(',')) \
  .keyBy(lambda x: x[0])
status.map(lambda x: (x[1][3], sdata[x[0]], x[1][1], x[1][2])).take(3)
#[("2015-01-09 17:02:02", 'Embarcadero at Sansome', '7', '8'),
# ("2014-09-01 00:00:03", 'Embarcadero at Sansome', '7', '8'),
# ("2014-09-01 00:01:02", 'Embarcadero at Sansome', '7', '8’)]
```

– Es mejor que la opción 1 pero no es escalable. La variable es parte de un closure en la función que la referencia, implicando transferencias innecesarias y duplicación de datos en los workers.

Spark: broadcast

3. La mejor opción: inicializar una variable broadcast variable para la tabla stations (pequeña). Se utiliza replicación P2P para que la variable quede disponible a todos los workers y la única copia puede ser usada por todas las tareas en todos los executors de la aplicación que ejecuta en el cluster.

```
stationsfile = 'stations.csv'
sdata = dict(map(lambda x: (x[0],x[1]), \
         map(lambda x: x.split(','),open(stationsfile))))
stations = sc.broadcast(sdata)
status = sc.textFile('home/cloudera/status.csv’) \
         .map(lambda x: x.split(',')).keyBy(lambda x: x[0])
status.map(lambda x: (x[1][3], stations.value[x[0]], x[1][1], \
         x[1][2])).take(3)
#[("2015-01-09 17:02:02", 'Embarcadero at Sansome', '7', '8'),
# ("2014-09-01 00:00:03", 'Embarcadero at Sansome', '7', '8'),
# ("2014-09-01 00:01:02", 'Embarcadero at Sansome', '7', '8’)]
```

Spark: broadcast

- Optimización de broadcast:
  - Cuando se necesita realizar broadcast de grandes variables, es importante seleccionar un formato de serialización que sea compacto y eficiente.
  - El envío del broadcast sobre la red no debe ser un cuello de botella.
  - La serialización de Java (Java Serialization) se utiliza por defecto en las APIs de Spark para Scala y Java, puede ser muy ineficiente out of the box para todos los tipos de datos que no son primitivos (o son arrays).
  - Es posible optimizar el broadcast seleccionando un mecanismo de serialización diferente (por ejemplo, Kyro, una biblioteca de serialización eficiente), implementando un método de serialización ad-hoc para los tipos de datos utilizados (e.g., usando la interfaz java.io.Externalizable para serialización eficiente o usando el método reduce() para definir serializaciones personalizadas para la biblioteca pickle de Python).

Apache Spark: broadcast

- Ejemplo sobre caso de estudio: log de radioaficionados.
  - Referencias de llamadas, (cada país tiene un rango), ubicación física (permite determinar distancias entre operadores, etc.)
  - Ejemplo de entrada (formato JSON, algunos campos removidos)

```
{"address":"address here",
"band":"40m","callsign":"KK6JLK","city":"SUNNYVALE",
"contactlat":"37.384733","contactlong":"-122.032164",
"county":"Santa Clara","dxcc":"291","fullname":"MATTHEW McPherrin",
"id":57779,"mode":"FM","mylat":"37.751952821","mylong":"-
122.4208688735",...}
```

  - Ejemplo de uso de variables compartidas para contar condiciones de error no fatal y distribuir una tabla de búsqueda de gran dimensión.
  - Cuando una tarea demanda un tiempo considerable para su configuración (por ejemplo, crear una conexión con una base de datos) es conveniente realizarla una única vez y compartir la variable correspondiente.

Spark: broadcast

- Procesar log de radioaficionados, identificar países en indicativos (call sign).
  - Indicativo: PPxSSS
  - Los prefijos no tienen largo fijo: EA para España, F para Francia.

```
# Analizar los países de los indicativos en el RDD contactCounts.
# Se carga una lista de prefijos (códigos de país)
# para considerer en la búsqueda.
signPrefixes = loadCallSignTable()
def processSignCount(sign_count, signPrefixes):
    country = lookupCountry(sign_count[0], signPrefixes)
    count = sign_count[1]
    return (country, count)
countryContactCounts = (contactCounts.map(processSignCount) \
    .reduceByKey((lambda x, y: x+ y)))
```

  - Si la tabla fuera más larga (por ejemplo, direcciones IP en lugar de indicativos, o inclusive más larga), signPrefixes podría ser de varios MB y seria ineficiente enviar el array desde el master a cada tarea. Si se utiliza el mismo objeto signPrefixes luego (sobre file2.txt), será enviado *nuevamente* a cada nodo.

Spark: broadcast

- Solución: signPrefixes como variable broadcast (en Python).

```python
signPrefixes = sc.broadcast(loadCallSignTable())
def processSignCount(sign_count, signPrefixes):
    country = lookupCountry(sign_count[0], signPrefixes.value)
    count = sign_count[1]
    return (country, count)
countryContactCounts = (contactCounts.map(processSignCount) \
    .reduceByKey((lambda x, y: x+ y)))
countryContactCounts.saveAsTextFile(outputDir + "/countries.txt")
```

- En Scala

```scala
val signPrefixes = sc.broadcast(loadCallSignTable())
val countryContactCounts = contactCounts.map{case (sign, count) =>
val country = lookupInArray(sign, signPrefixes.value)
(country, count)
}.reduceByKey((x, y) => x + y)
countryContactCounts.saveAsTextFile(outputDir + "/countries.txt")
```

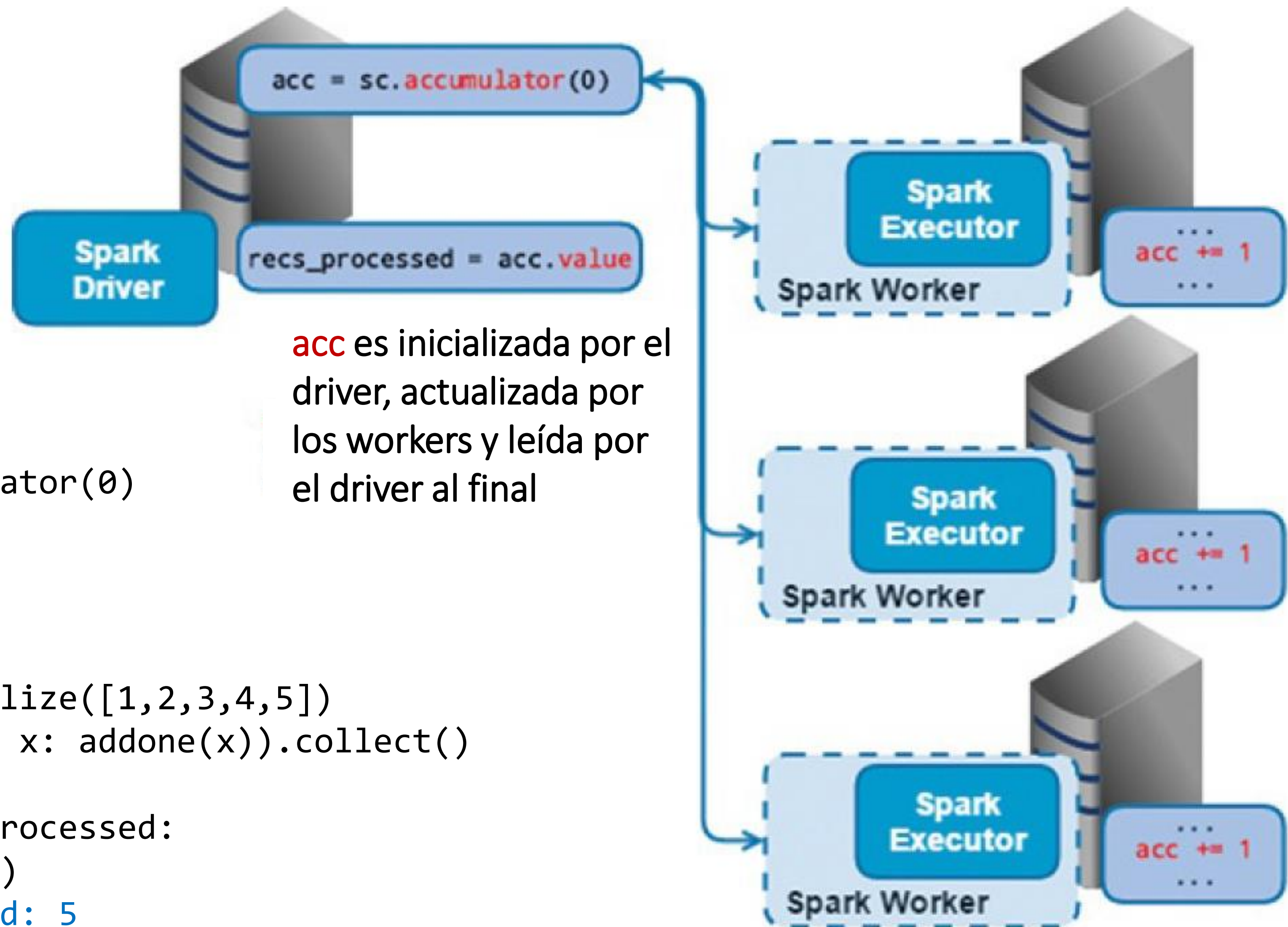
Spark: broadcast

- Resumen:
  1. Crear Broadcast[T] usando SparkContext.broadcast en un objeto de tipo T (cualquier tipo serializable).
  2. La variable se envía a cada nodo una única vez.
  3. Acceder a su valor mediante el campo value.
  4. La variable se trata como de solo lectura (si se actualiza, su valor no se propaga a otros nodos).
     – La manera más simple: broadcast de un valor primitivo o de referencia a objeto inmutable. Solamente podrá ser modificada en el driver.
     – Si el broadcaste es de un objeto modificable, el programador debe mantener la condición de solo lectura (por ejemplo, val theArray = broadcastArray.value; theArray(0) = newValue, modificando solo la variable local y no broadcastArray.value en los otros worker nodes.

Apache Spark: acumuladores

- Acumuladores: variables globales numéricas que, a diferencia de broadcast, pueden ser modificadas (incrementadas).
  - Permiten agregación y conteo mientras el programa distribuido ejecuta.
  - Son creadas por el driver y actualizadas por los executors que ejecutan tareas dentro del spark context respectivo.
  - El valor final es leído únicamente por el driver, en general al final del programa.
  - Son actualizados solamente una vez por cada tarea exitosamente completada en la aplicación Spark.
  - Los acumuladores pueden ser valores enteros o números de punto flotante.
  - Uno de los usos más comunes de los acumuladores es contar eventos que ocurren durante la ejecución de una aplicación (para verificación y debug).

Apache Spark: acumuladores



acc es inicializada por el driver, actualizada por los workers y leída por el driver al final

```
acc = sc.accumulator(0)
def addone(x):
   global acc
   acc += 1
   return x + 1
myrdd=sc.parallelize([1,2,3,4,5])
myrdd.map(lambda x: addone(x)).collect()
[2, 3, 4, 5, 6]
print("records processed:
"+str(acc.value))
records processed: 5
```

Apache Spark: acumuladores

- SparkContext.accumulator(initialValue, [accumPar=none])
  - Crea una instancia de un acumulador en el Spark context y lo inicializa con el valor indicado.
  - accumPar permite definir acumuladores personalizados.
- Accumulator.value(): accede al valor del acumulador
  - Solo es posible utilizarlo en el driver
- Acumuladores personalizados: permiten realizar agregaciones sobre tipos no numéricos.
  - Las operaciones deben ser asociativas y conmutativas.
  - Comúnmente utilizados para acumular vectores en listas o diccionarios.
  - También pueden usarse para operaciones no numéricas (por ejemplo, concatenar strings).
  - Se debe extender la clase AccumulatorParam e incluir dos funciones: addInPlace para agregar dos objetos (del tipo de datos personalizado) y retornar un resultado y zero, el valor cero para el tipo definido.

Apache Spark: acumuladores

- Ejemplo: obtener logs de llamadas de un archivo y contar líneas en blanco del archivo de entrada (pueden denotar una omisión o un error).

```
file = sc.textFile(inputFile)        # Crear acumulador[Int] inicializado a 0
blankLines = sc.accumulator(0)
def extractCallSigns(line):
    global blankLines                # Hacer accessible la variable global
    if (line == ""):
        blankLines += 1
    return line.split(" ")
callSigns = file.flatMap(extractCallSigns)
callSigns.saveAsTextFile(outputDir + "/callsigns")
print "Blank lines: %d" % blankLines.value
```

- El acumulador[Int] blankLines permite agregar 1 cada vez que se identifica una línea en blanco en el archivo de entrada.
- El valor se accede solamente luego de ejecutar la acción saveAsTextFile() porque la transformación aplicada (flatMap()) es lazy. El incremento del acumulador sucede cuando flatMap() es forzado a ejecutar por la acción saveAsTextFile().

Apache Spark: acumuladores

- Ejemplo: obtener logs de llamadas de un archivo y contar líneas en blanco del archivo de entrada (pueden denotar una omisión o un error) (en Scala).

```scala
val sc = new SparkContext(...)
val file = sc.textFile("file.txt")
val blankLines = sc.accumulator(0) // Crear acumulador[Int], inicializar a 0
val callSigns = file.flatMap(line => {
    if (line == "") {
       blankLines += 1                 // Agregar en el acumulador
    }
    line.split(" ")
})
callSigns.saveAsTextFile("output.txt")
println("Blank lines: " + blankLines.value)
```

  - Es posible agregar valores de un RDD y enviar el resultado al driver con acciones como reduce(). Los acumuladores proven una manera simple de acumular/contar mientras se transforma un RDD.
  - En el ejemplo, el acumulador permite contar las líneas en blanco mientras se cargan los datos, sin realizar una acción filter() o reduce() por separado.

Apache Spark: acumuladores

- Ejemplo: obtener logs de llamadas de un archivo y contar líneas en blanco del archivo de entrada (pueden denotar una omisión o un error) (en Java)

```
JavaRDD<String> rdd = sc.textFile(args[1]);
final Accumulator<Integer> blankLines = sc.accumulator(0);
JavaRDD<String> callSigns = rdd.flatMap(
new FlatMapFunction<String, String>() { public Iterable<String> call(String
    line) {
        if (line.equals("")) {
            blankLines.add(1);
        }
        return Arrays.asList(line.split(" "));
}});
callSigns.saveAsTextFile("output.txt")
System.out.println("Blank lines: "+ blankLines.value());
```

Spark: acumuladores

- Resumen:
  1. Crear acumulador[T] usando SparkContext.accumulator(initialValue) en un objeto de tipo T, numérico (el mismo tipo que initialValue). La variable creada es de tipo org.apache.spark.Accumulator[T].
  2. El código de los workers (en closures de Spark) puede agregar al acumulador con su método += (o add en Java).
  3. El driver puede aceder al valor mediante la propiedad value (value() en Java).

- Las tareas en los workers no pueden acceder a value(), para ellas los acumuladores son variables de solo escritura. Este mecanimos permite implementar eficientemente los acumuladores, sin que sea necesario comunicar sus actualizaciones.

Spark: acumuladores

- Acumuladores: útiles cuando se necesita contar varios valores, o un mismo valor se debe incrementar en varios lugares del programa paralelo (por ejemplo, contar cuántas veces se invocó a una rutina de parsing de archivos JSON).
- Casos de aplicación: procesamiento que tolera algunos datos corruptos o permite algunos fallos ocasionales. Se implementa con contadores para registros válidos e inválidos.
- Ejemplo: sobre archivo de datos de radioficionados, validar códigos de llamada e imprimir registros erróneos (si su número es razonable).

Spark: acumuladores

```
validSignCount = sc.accumulator(0)
invalidSignCount = sc.accumulator(0)

def validateSign(sign):
   global validSignCount, invalidSignCount
   # formato International Telecommunication Union
   if re.match(r"\A\d?[a-zA-Z]{1,2}\d{1,4}[a-zA-Z]{1,3}\Z", sign):
      validSignCount += 1
      return True
   else:
      invalidSignCount += 1
      return False

# Contar número de veces de cada caso
validSigns = callSigns.filter(validateSign)
contactCount = validSigns.map(lambda sign:(sign,1)).reduceByKey(lambda(x, y):x+y)
# Forzar la evaluación para que los contadores se actualicen
contactCount.count()

# los chequeos deben realizarse en el driver !
if invalidSignCount.value < 0.1 * validSignCount.value:
   contactCount.saveAsTextFile(outputDir + "/contactCount")
else:
   print “Muchos errores: %d in %d" % (invalidSignCount.value,validSignCount.value)
```

Spark: acumulador personalizado

```
from pyspark import AccumulatorParam
class VectorAccumulatorParam(AccumulatorParam):

def zero(self, value):
    dict1={}
    for i in range(0,len(value)):
        dict1[i]=0
    return dict1

def addInPlace(self, val1, val2):
    for i in val1.keys():
       val1[i] += val2[i]
    return val1
rdd1 = sc.parallelize([{0: 0.3, 1: 0.8, 2: 0.4}, {0: 0.2, 1: 0.4, 2:0.2}])
vector_acc = sc.accumulator({0: 0, 1: 0, 2: 0},VectorAccumulatorParam())

def mapping_fn(x):
    global vector_acc
    vector_acc += x
    # seguir procesando el RDD ...
rdd1.foreach(mapping_fn)
print vector_acc.value
# {0: 0.50, 1: 1.20, 2: 0.60}
```

Spark: trabajo por particiones

- Permite evitar realizar tareas de configuración (setup) para cada ítem.
  - Conexión con base de datos, creación de generador de números aleatorio.
  - Versiones de map y foreach para aplicarse solo una vez por partición.
- Ejemplo: consultar una base de datos online para analizar contactos
  - Con operaciones basadas en partición se comparte una conexión a la base de datos (para no realizar multiples conexiones) y reutilizar el parser de JSON.

```
def processCallSigns(signs):
   http = urllib3.PoolManager() # Crear una conexión con la URL de cada registro
   urls = map(lambda x: "http://73s.com/qsos/%s.json" % x, signs)
   # Crear los requests (no-bloqueante)
   requests = map(lambda x: (x, http.request('GET', x)), urls)
   # Fetch de los resultados
   result = map(lambda x: (x[0], json.loads(x[1].data)), requests)
   # Eliminar resultados vacíos y retornar
   return filter(lambda x: x[1] is not None, result)

def fetchCallSigns(input):
   return input.mapPartitions(lambda callSigns : processCallSigns(callSigns))

contactsContactList = fetchCallSigns(validSigns)
```

Spark: trabajo por particiones

- En Scala

```
val contactsContactLists = validSigns.distinct().mapPartitions{
    signs =>
    val mapper = createMapper()
    val client = new HttpClient()
    client.start() // create request HTTP
    signs.map {sign =>
       createExchangeForSign(sign)
    // obtener respuestas
    }.map{ case (sign, exchange) =>
       (sign, readExchangeCallLog(mapper, exchange))
    }.filter(x => x._2 != null) // Eliminar registros vacíos
}
```

- Las funciones por partición reciben un iterador de elementos en cada partición y retornan un iterable.
  - mapPartitions(), f: (Iterator[T]) → Iterator[U]
  - foreachPartition(), f: (Iterator[T]) → Unit

Spark: trabajo por particiones

- mapPartitions() también puede aplicarse para evitar el overhead de crear objetos (e.g., para agregar, cuando el resultado tiene un tipo diferente).
  - Ejemplo: cálculo de promedio. Se puede convertir el RDD de números a un RDD de pares para tomar en cuenta el número de elementos procesados.
  - Como alternativa, se puede crear un par por partición.

Estándar

```
def combineCtrs(c1, c2):
   return (c1[0]+c2[0], c1[1]+c2[1])
def basicAvg(nums):
   """"calcular el promedio"""
   nums.map(lambda num: (num, 1)) \
      .reduce(combineCtrs)
```

Por partición

```
def partitionCtr(nums):
"""Compute sumCounter for partition"""
   sumCount = [0, 0]
   for num in nums:
      sumCount[0] += num
      sumCount[1] += 1
   return [sumCount]
def fastAvg(nums):
   """Compute the avg"""
   sumCount = nums.mapPartitions(partitionCtr)
      .reduce(combineCtrs)
   return sumCount[0] / float(sumCount[1])
```

Spark: interacción con programas externos

- Además de los bindings para Python, Scala y Java, Spark provee un mecanismo para interconectar con otros lenguajes.
- El método pipe() permite conexiones a través de streams de Unix/Linux.
- RDD.pipe(comando[, env=None, checkCode=False]) retorna un RDD creado con el resultado de un proceso externo que ejecuta el comando
  - env: diccionario (dict) de variables de entorno, checkCode: indica si se debe verificar la ejecución correcta del comando en el shell.
  - El script/programa a ejecutar debe leer de STDIN y escribir la salida a STDOUT.
- Ejemplo: parser en Perl para datos con campos de largo fijo

```
#!/usr/bin/env perl
my $format = 'A6 A8 A8 A20 A2 A5';
while (<>) {
    chomp;
    my($custid, $orderid, $date, $city, $state, $zip) = unpack($format, $_);
    print "$custid\t$orderid\t$date\t$city\t$state\t$zip";
}
```

Spark: interacción con programas externos

- Pipe de Spark para ejecutar el parser en perl

```
sc.addFile("/home/ubuntu/parsefixedwidth.pl")
fixed_width = sc.parallelize(['3840961028752220160317Hayward CA94541'])
piped = fixed_width.pipe("parsefixedwidth.pl").map(lambda x: x.split('\t'))
piped.collect()
[['384096', '10287522', '20160317', 'Hayward', 'CA', '94541’]]
```

  – La operación addFile distribuye el script perl a todos los worker nodes en el custer, previo a la ejecución.
  – Los archivos a distribuir pueden estar en el filesystem local, en HDFS/HTTP/FTP.
  – En los worker nodes, los archivos quedan en SparkFiles.getRootDirectory, o pueden ubicarse con SparkFiles.get(filename).
  – En este caso debe asegurarse que el intérprete (Perl) está disponible en el path indicado en todos los worker nodes.

Spark: interacción con programas externos

- Pipe para ejecutar programas en R
  - Script R para calcular distancias entre contactos en la lista de llamadas.
  - Los elementos en el RDD de entrada se escriben por línea (/n como separador) y cada línea emitida por el script R es un elemento (string) en el RDD resultado.
  - La entrada se parsea y reformatea: latA, lonA, latB, lonB (separador: coma).

```
#!/usr/bin/env Rscript
library("Imap")
f <- file("stdin")
open(f)
while(length(line <- readLines(f,n=1)) > 0) { # procesar línea
    contents <- Map(as.numeric, strsplit(line, ","))
    mydist <- gdist(contents[[1]][1], contents[[1]][2],contents[[1]][3],
                    contents[[1]][4], units="m", a=6378137.0,
                    b=6356752.3142, verbose = FALSE)
    write(mydist, stdout())
}
```

  - Ejecución:

```
$ ./src/R/finddistance.R
37.75889318222431,-122.42683635321838,37.7614213,-122.4240097
349.2602
```

Spark: interacción con programas externos

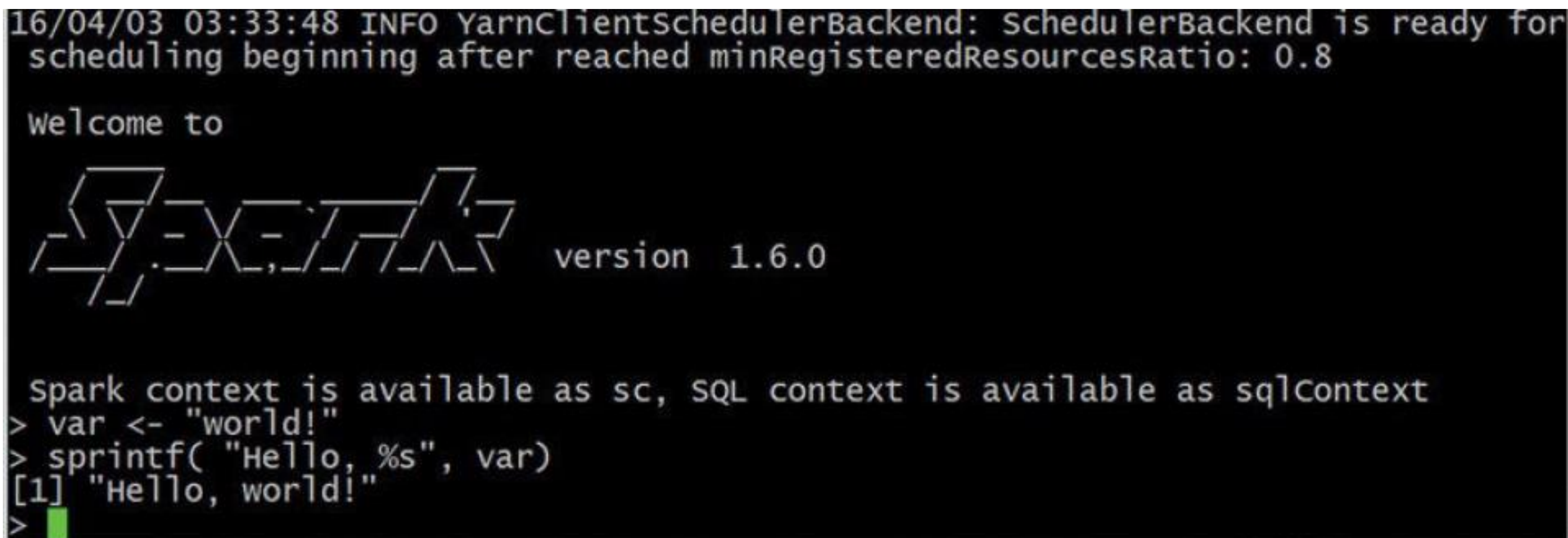
- Driver para ejecutar la tarea en Spark

```
distScript = "./src/R/finddistance.R"
distScriptName = "finddistance.R"
sc.addFile(distScript)

def hasInfo(call): """Verifica que el registro tiene los campos necesarios""
requiredFields = ["mylat", "mylong", "contactlat", "contactlong"]
return all(map(lambda f: call[f], requiredFields))

def formatCall(call): """Formatea el registro para el script R"""
return "{0},{1},{2},{3}".format(call["mylat"], call["mylong"],
    call["contactlat"], call["contactlong"])
pipeInputs = contactsContactList.values().flatMap(lambda calls:
         map(formatCall, filter(hasInfo, calls)))
distances = pipeInputs.pipe(SparkFiles.get(distScriptName))
print distances.collect()
```

  – Si findDistance recibiera el separador como parámetro, rdd.pipe(Seq(SparkFiles.get("finddistance.R"), ",")) o rdd.pipe(SparkFiles.get("finddistance.R") + " ,") invoca el comando con una secuencia posicional de argumentos (el comando está en la posición 0).

SparkR

- SparkR: frontend liviano para usar Spark desde R.
  - Disponible en $SPARK_HOME/R/lib/SparkR.
  - sparkR: shell para trabajar desde línea de comandos, inicia una sesión en R con el entorno Spark por defecto (spark.master, spark.driver.memory, etc.)
  - El SparkContext se crea como sc. Un SQLContext está disponible como sqlContext. Ambos son puntos de entrada para usar spark y data frames.



Estadística con Spark

- Ejemplo: remover outliers de un conjunto de datos
- Se operará sobre el mismo RDD dos veces (calcular la estadística y remover los outliers) se persistirá en caché.
- En el log de llamadas, remover los contactos que están 'muy lejanos'.

```
# Convertir el RDD de strings a datos numéricos para calcular
estadísticas
distanceNumerics = distances.map(lambda string: float(string))
stats = distanceNumerics.stats()
stddev = std.stdev()
mean = stats.mean()
reasonableDistances =
    distanceNumerics.filter(lambda x: math.fabs(x - mean) < 3 * stddev)
print reasonableDistances.collect()
```

Apache Spark

- Ejemplo: WordCount en Spark, programado en Scala

```
// ‘sc’ es ‘Spark context’: transforma el archivo en un RDD
val textFile = sc.textFile(“<ARCHIVO>")
// Retorna el número de items (línea) en el RDD; count() es una acción
textFile.count()
// Filtro de ejemplo. Filtro es una tranformación
val linesWithSpark = textFile.filter(line => line.contains("Spark"))
// Conteo – cuántas líneas contienen “Spark”; count() es una acción
textFile.filter(line => line.contains("Spark")).count()
// Largo de la línea con más palabras; reduce es una acción
textFile.map(line => line.split(" ").size).reduce((a,b) => if (a>b) a else b)
// WordCount – MapReduce tradicional; collect() es una acción
val wordCounts = textFile.flatMap(line => line.split(" ")).map(word =>
                 (word,1)).reduceByKey((a,b) => a + b)
wordCounts.collect()
```

Fuente: https://spark.apache.org/docs/latest/quick-start.html

Apache Spark

- Ejemplo: Regresión logística, programado en Python

```
# Algoritmo iterativo de aprendizaje computacional
# Encuentra el mejor hiperplano que separa dos conjuntos de puntos
# en un espacio multidimensional de características
# Aplica iterativamente MapReduce sobre el mismo conjunto de datos,
# por lo cual se beneficia significativamente de disponer los datos
# persistidos en memoria

points = spark.textFile(...).map(parsePoint).cache()
w = numpy.random.ranf(size = D) # Plano separador inicial
for i in range(ITERATIONS):
    gradient = points.map(lambda p:(1 / (1 + exp(-p.y * (w.dot(p.x)))) -
               1) * p.y * p.x)).reduce(lambda a, b: a + b)
    w -= gradient
print "Plano separador final: %s" % w
```

Fuente: https://spark.apache.org/docs/latest/quick-start.html
Código completo: http://elk.acis.ufl.edu/spark-2.1.0-bin-hadoop2.6/examples/src/main/python/logistic_regression.py

Apache Spark

- Ejemplo: Regresión logística, programado en Scala

```
// Algoritmo iterativo de aprendizaje computacional
// Encuentra el major hiperplano que separa dos conjuntos de puntos
// en un espacio multidimensional de características
// Aplica iterativamente MapReduce sobre el mismo conjunto de datos,
// por lo cual se beneficia significativamente de disponer los datos
// persistidos en memoria

val points = spark.textFile(...).map(parsePoint).cache()
var w = Vector.random(D) // Plano separador inicial Plano separador inicial
for (i <- 1 to ITERATIONS) {
  val gradient = points.map(p => (1 / (1 + exp(-p.y*(w dot p.x))) - 1) * p.y
                 * p.x).reduce(_ + _)
  w -= gradient
}
println(" Plano separador final: " + w)
```

Fuente: https://spark.apache.org/docs/latest/quick-start.html

Apache Spark

- Ejemplo: Regresión logística, programado en Java

```
// Algoritmo iterativo de aprendizaje computacional
// Aplica iterativamente MapReduce sobre el mismo conjunto de datos,
// por lo cual se beneficia significativamente de disponer los datos
// persistidos en memoria

class ComputeGradient extends Function<DataPoint, Vector> {
  private Vector w;
  ComputeGradient(Vector w) { this.w = w; }
  public Vector call(DataPoint p) {
    return p.x.times(p.y * (1 / (1 + Math.exp(w.dot(p.x))) - 1));
  }
}

JavaRDD<DataPoint> points = spark.textFile(...).map(new ParsePoint()).cache();
Vector w = Vector.random(D); // current separating plane
for (int i = 0; i < ITERATIONS; i++) {
  Vector gradient = points.map(new ComputeGradient(w)).reduce(new AddVectors());
  w = w.subtract(gradient);
}
System.out.println("Plano separador final: " + w);
```