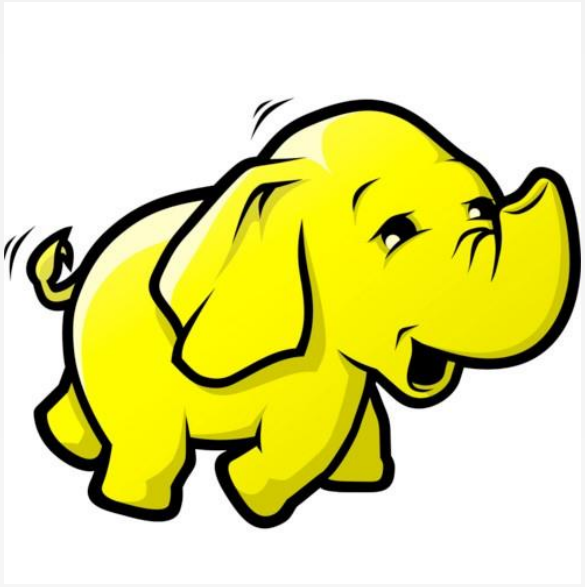
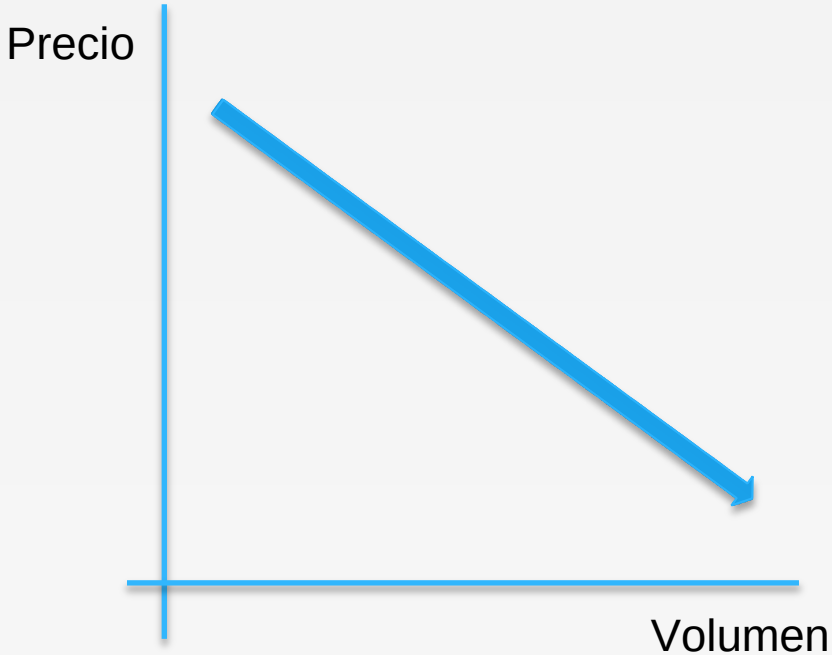
Apache Spark

FUNDAMENTOS TECNOLÓGICOS PARA BIG DATA

Isaac Esau Rubio Torres
2018

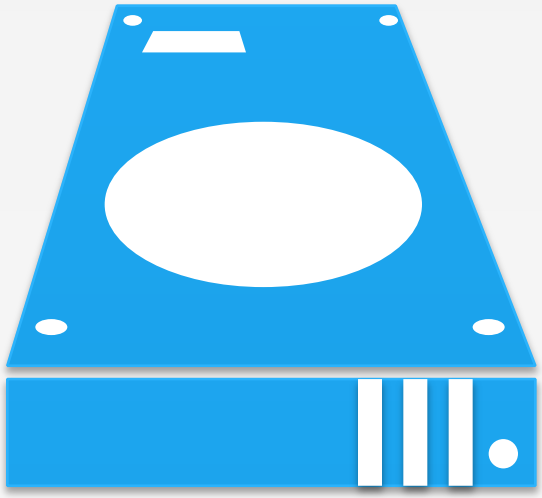
Hadoop





Apache Sparkv

Primer Problema: La Persistencia



Segundo Problema: MapReduce

<k1, v1> -> map -> <k2, v2>
<k2, v2> -> combine -> <k2, v2>
<k2, v2> -> reduce -> <k3, v3>

Tercer Problema: Cálculos iterativos

Spark

· Es un **motor** para el procesamiento en memoria de grandes volúmenes de datos.

· Se **facilita el paradigma MapReduce** (reduciendo costes y tiempos de ejecución) a gracias a los RDDs.

· Tiene API´s para **Scala, Java & Python**.

Spark

- Modelo Resilient Distributed Data Sets.
- Modelo de distribuido de memoria para computación, aplicado con apache Mesos.
- Escrito en Scala.
- En memoria 100x. ● En disco 10x.

https://spark.apache.org/

¿Quién usa Spark?


DATA
ClearStory
Now You See It™


Baidu 百度


GROUPON


TREND
MICRO™

¿Por qué Apache Spark?

	Hadoop World Record	Spark 100 TB *	Spark 1 PB
Data Size	102.5 TB	100 TB	1000 TB
Elapsed Time	72 mins	23 mins	234 mins
# Nodes	2100	206	190
# Cores	50400	6592	6080
# Reducers	10,000	29,000	250,000
Rate	1.42 TB/min	4.27 TB/min	4.27 TB/min
Rate/node	0.67 GB/min	20.7 GB/min	22.5 GB/min
Sort Benchmark Daytona Rules	Yes	Yes	No
Environment	dedicated data center	EC2 (i2.8xlarge)	EC2 (i2.8xlarge)

* not an official sort benchmark record

http://databricks.com/blog/2014/10/10/spark-petabyte-sort.html @abxda

Spark

La “magia”:
Resilient **D**istributed **D**atasets

¿Qué es un RDD?

· Colecciones lógicas, inmutables y particionadas de los registros a lo largo del clúster.

RDD - Beneficios

· La consistencia se vuelve más sencilla gracias a la **inmutabilidad**.

· **Tolerante a fallos**: a través del “Lineage” los RDDs se pueden reconstruir si alguna partición se pierde.

· A pesar de que Batch Processing es un modelo restringido a una serie de casos de uso por defecto, gracias a los RDDs se puede utilizar en **multitud de aplicaciones**.

· Es bueno para **algoritmos iterativos**.

· Más rápido que Hadoop.

Operaciones sobre RDDs

Transformaciones		Acciones

Map ***Filter*** ***Sample*** ***Union*** ***groupByKey*** ***reduceByKey*** ***Join*** ***Cache*** ***…***	***Reduce*** ***Collect*** ***Count*** ***Save*** ***lookupKey*** ***…***

Operaciones sobre RDDs

Transformación	Significado
map(*func*)	Devolución de un nuevo conjunto de datos distribuido formado a través de aplicar a cada elemento de la fuente la función func .
filter(*func*)	Devuelve un nuevo conjunto de datos formado por la selección de aquellos elementos de la fuente en la que func devuelve true.
flatMap(*func*)	Al igual que en map, pero cada elemento de entrada se puede asignar a 0 o más elementos de salida (modo func debe devolver una Sec en lugar de un solo elemento) .

mapPartitions(*func*)	Similar a map, pero corre por separado en cada partición (bloque) de la RDD, por lo func debe ser de tipo Iterator < T> = > Iterator < U> cuando se ejecuta en un RDD de tipo T.
mapPartitionsWithIndex(*func*)	Similar a mapPartitions, pero también proporciona func con un valor entero que representa el índice de la partición , así que func debe ser de tipo (Int , Iterator < T>) = > Iterator < U> cuando se ejecuta en un RDD de tipo T.
sample(*withReplacement*, *fraction*, *seed*)	Muestra una fracción de los datos, con o sin sustitución, utilizando una semilla generador de números aleatorios dado.
union(*otherDataset*)	Devuelve un nuevo conjunto de datos que contiene la unión de los elementos en el conjunto de datos de origen y el argumento pasado.

Operaciones sobre RDDs

intersection(*otherDataset*)	Devolución de un nuevo RDD que contiene la intersección de elementos en el conjunto de datos de origen y el argumento.
distinct([*numTasks*]))	Devuelve un nuevo conjunto de datos que contiene los elementos distintos del conjunto de datos de origen.
groupByKey([*numTasks*])	Cuando se llama sobre un conjunto de datos de (K, V) pares, devuelve un conjunto de datos de (K , iterable < V >) pares . Nota : Si se está agrupando a fin de realizar una agregación (como una suma o promedio) sobre cada clave, usando reduceByKey o combineByKey se obtendrá mucho mejor rendimiento. Nota: Por defecto , el nivel de paralelismo en la salida depende del número de particiones de la RDD padre. Puede pasar un argumento numTasks opcionales para establecer un número diferente de tareas .

reduceByKey(*func*, [*numTasks*])	Cuando se llama a un conjunto de datos de (K , V) tuplas, devuelve un conjunto de datos de (K , V) tupla donde los valores para cada tulla se agregan utilizando el dado reducir la función func, que debe ser del tipo (V , V) = > V. Como en groupByKey, el número de reduciones en las tareas se puede configurar a través de un segundo argumento opcional.
aggregateByKey(*zeroValue*)(*seqOp*, *combOp*, [*numTasks*])	Cuando se llama a un conjunto de datos de (K, V) pares, devuelve un conjunto de datos de (K , T) pares donde los valores para cada clave se agregan utilizando las funciones dadas de combinación y un neutral valor "cero" . Permite un tipo de valor agregado que es diferente que el tipo de valor de entrada, evitando al mismo tiempo las asignaciones innecesarias. Al igual que en groupByKey, el número de tareas de reducción se pueden configurar a través de un segundo argumento opcional.
sortByKey([*ascending*], [*numTasks*])	Cuando se llama sobre un conjunto de datos de (K, V) tullas, donde K estable el orden, devuelve un conjunto de datos de (K, V) pares ordenados por claves en orden ascendente o descendente , según se especifica en el argumento ascendente boolean .

Operaciones sobre RDDs

join(*otherDataset*, [*numTasks*])	Cuando se llama sobre conjuntos de datos de tipo (k, V) y (K , W), devuelve un conjunto de datos de pares con todos los pares de elementos para cada tulla (K, (V , W)) . Las combinaciones externas son apoyadas a través leftOuterJoin, rightOuterJoin y fullOuterJoin .
cogroup(*otherDataset*, [*numTasks*])	Cuando se invoca sobre los conjuntos de datos de tipo (K , V) y (K, W), devuelve un conjunto de datos de (K, Iterable < V > , < Iterable W >) tuplas. Esta operación también se llama groupWith.

cartesian(*otherDataset*)	Cuando se invoca sobre los conjuntos de datos de los tipos T y U, devuelve un conjunto de datos de (T , U) pares (todos los pares de elementos).
coalesce(*numPartitions*)	Disminuye el número de particiones en el RDD a numPartitions . Útil para optimizar la operaciones de RDD después de filtrar por un gran conjunto de datos.
repartition(*numPartitions*)	Reordena los datos de la RDD al azar para crear, ya sea más o menos particiones, y el equilibrio que a través de ellas. Esto siempre baraja todos los datos en la red.
repartitionAndSortWithinPartitions(*partitioner*)	Reparticiona el RDD según el particionador dado y, dentro de cada partición resultante, ordena los registros por parte de sus claves. Esto es más eficiente que llamar reparto y luego la clasificación dentro de cada partición.

Operaciones sobre RDDs

- Todas las transformaciones en Spark son **perezosas**, ya que no calculan sus resultados de inmediato
- Sólo recuerda las transformaciones aplicadas a algún conjunto de datos de base (por ejemplo, un archivo)

- Las **transformaciones se calculan sólo cuando una acción requiere** un resultado para ser devuelto al programa controlador
- Este diseño permite Spark para funcionar de manera más eficiente

- Ejemplo: podemos darnos cuenta de que un conjunto de datos creado a través de un mapa se puede utilizar en una reducción y devolver sólo el resultado de la reducir al conductor, en lugar de todo el conjunto de datos

RDD

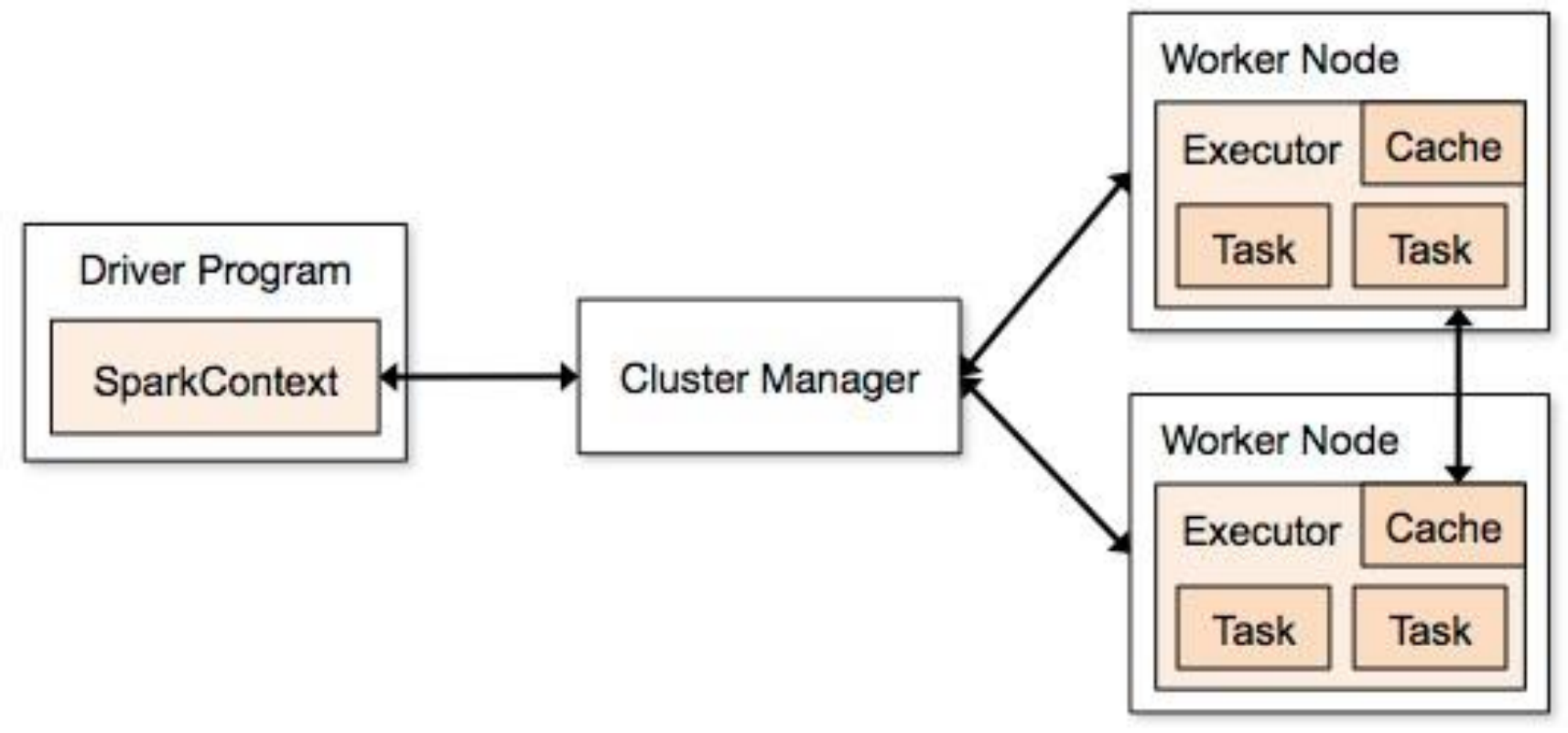
```
/* SimpleApp.scala */ import
org.apache.spark.SparkContext import
org.apache.spark.SparkContext._ import
org.apache.spark.SparkConf

object SimpleApp {
  def main(args: Array[String]) { val logFile = "YOUR_SPARK_HOME/README.md" // Should
    be some file on your system val conf = new SparkConf().setAppName("Simple
    Application") val sc = new SparkContext(conf)
```

```
    val logData = sc.textFile(logFile, 2).cache() val numAs =
    logData.filter(line => line.contains("a")).count() val numBs =
    logData.filter(line => line.contains("b")).count() println("Lines
    with a: %s, Lines with b: %s".format(numAs, numBs))
  }
}
```

RDD

```
val lines = sc.textFile("data.txt") val pairs
= lines.map(s => (s, 1)) val counts =
pairs.reduceByKey((a, b) => a + b)
```

Spark: Arquitectura



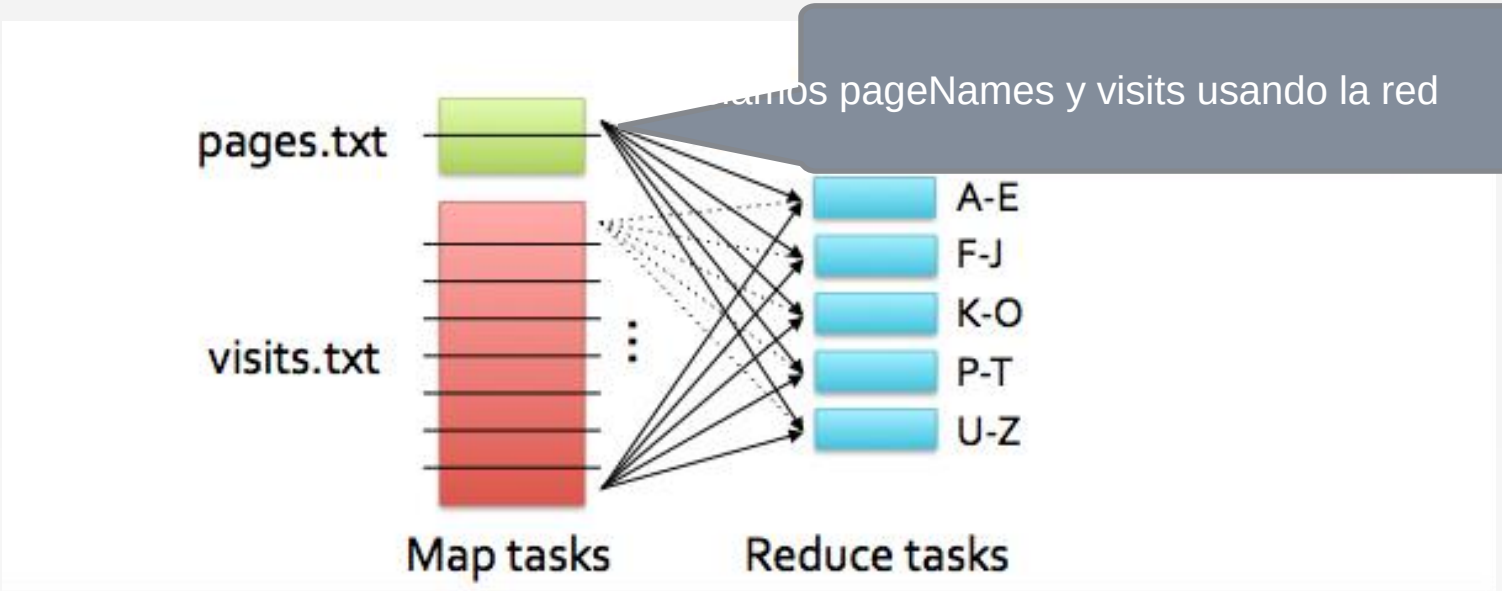
Spark - Además

- Spark es **agnóstico**.
- Si las operaciones no caben en memoria, **pagina a disco**.
- **Aplicaciones aisladas**: cada aplicación tiene su propio Executor.

Spark - Ejemplo join

```
// Load RDD of (URL, name) pairs
val pageNames = sc.textFile("pages.txt").map(...)

// Load RDD of (URL, visit) pairs
```

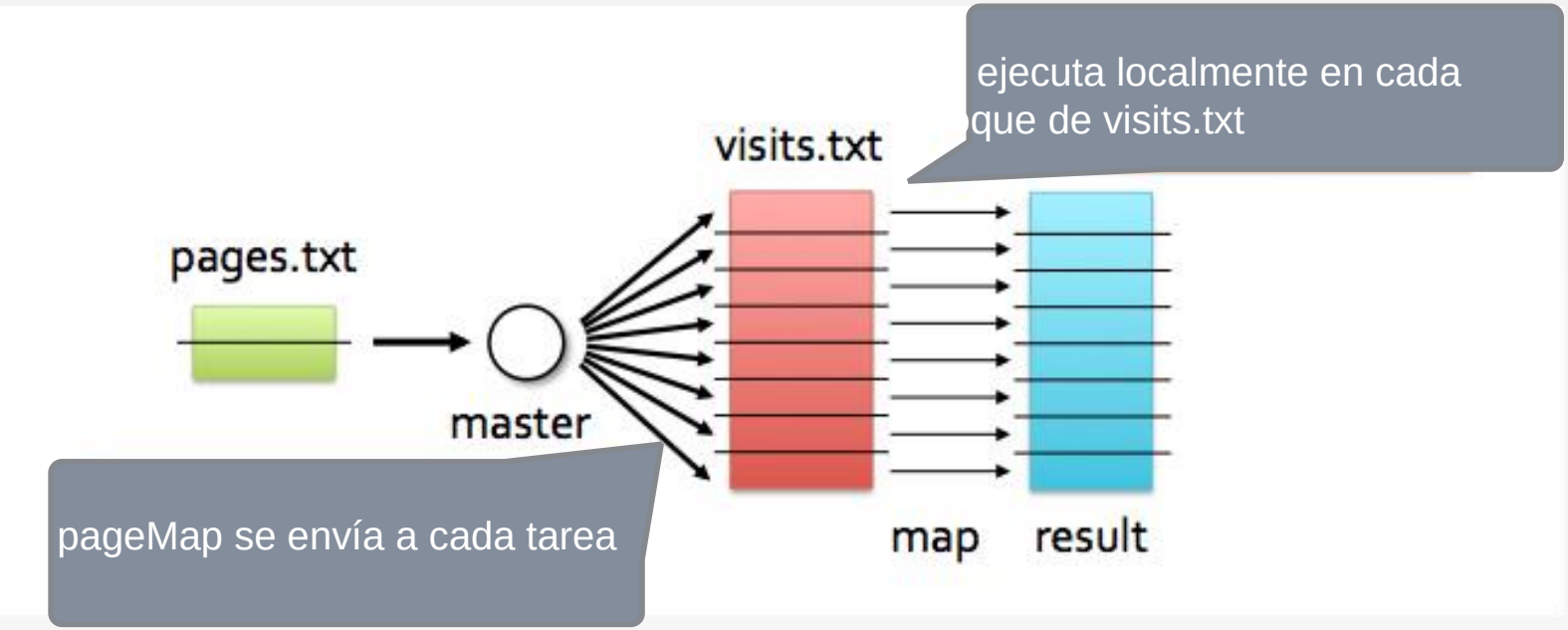
```
val visits = sc.textFile("visits.txt").map(...)
val joined = visits.join(pageNames)
```



Spark - Ejemplo join (tablas pequeñas)

```
val pageNames = sc.textFile("pages.txt").map(...)
val pageMap = pageNames.collect().toMap() val
visits = sc.textFile("visits.txt").map(...)
```

```
val joined = visits.map(v => (v._1, (pageMap(v._1), v._2)))
```

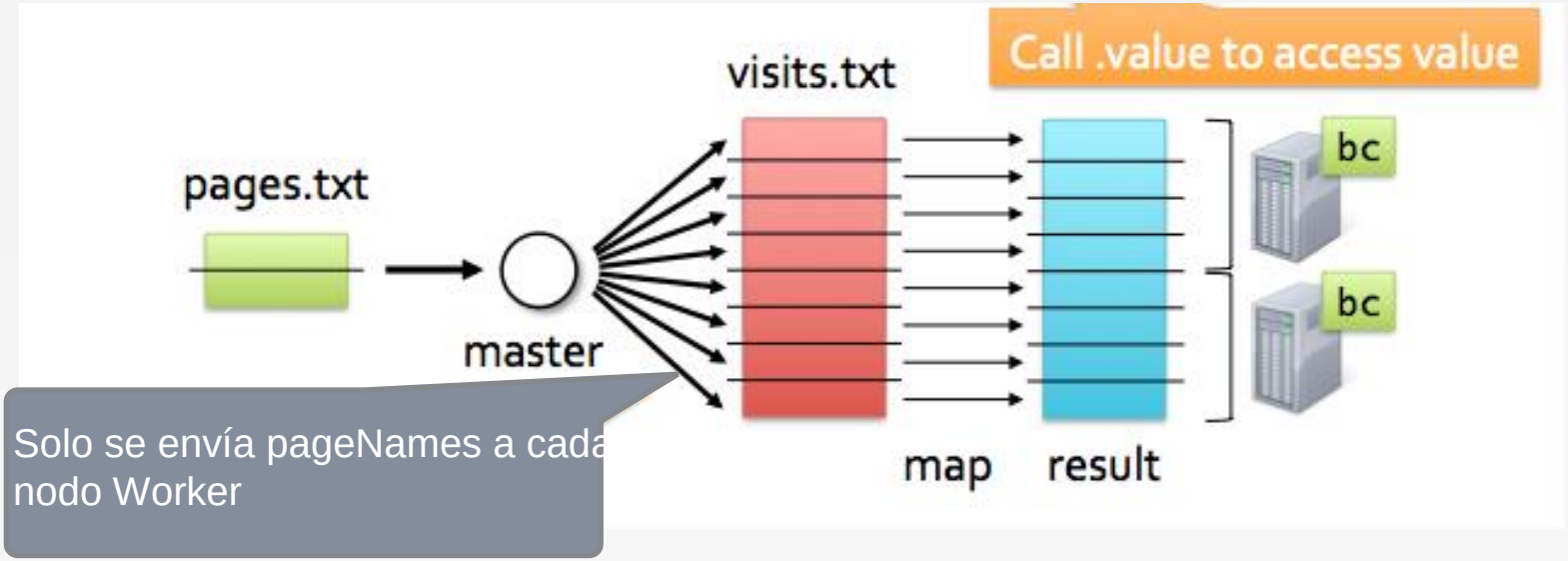


Spark - Ejemplo join (mejor solución)

```
val pageNames = sc.textFile("pages.txt").map(...)
val pageMap = pageNames.collect().toMap()

val bc = sc.broadcast(pageMap)
```

Type is Broadcast[Map[...]]

```
val visits = sc.textFile("visits.txt").map(...) val joined =
visits.map(v => (v._1, (bc.value(v._1), v._2)))
```



Análisis interactivo de datos

- Exploración de datos mediante una Shell interactiva en Scala.

Variables

Broadcast:

```
scala> val broadcastVar = sc.broadcast(Array(1,2,3)) broadcastVar:
```

org.apache.spark.broadcast.Broardcast[Array[Int]] = Broadcast(2)

scala> broadcastVar.value
res6: Array[Int] = Array(1,2,3)

Variables

Broadcast:

scala> val broadcastVar = sc.broadcast(Array(1,2,3)) broadcastVar: org.apache.spark.broadcast.Broardcast[Array[Int]] = Broadcast(2)

scala> broadcastVar.value
res6: Array[Int] = Array(1,2,3)

Accumulators:

```
scala> val accum = sc.accumulator(0) accum:
```

org.apache.spark.Accumulator[Int] = 0 scala>

sc.parallelize(Array(1,2,3,4)).foreach(x => accum +=x)

scala> accum.value
res8: Int =10

Ejemplo1: Paralelización de una colección

```
scala> val data = Array(25, 20, 15, 10, 5)
data: Array[Int] = Array(25, 20, 15, 10, 5)

scala> val distData = sc.parallelize(data)
distData: org.apache.spark.rdd.RDD[Int] = ParallelCollectionRDD[0] at parallelize at
<console>:14

scala> distData.reduce(_ + _)
res0: Int = 75
```

Ejemplo1: Paralelización de una colección

```
scala> val data = Array(25, 20, 15, 10, 5)
data: Array[Int] = Array(25, 20, 15, 10, 5)
```

```
scala> val distData = sc.parallelize(data)
distData: org.apache.spark.rdd.RDD[Int] = ParallelCollectionRDD[0] at parallelize at
<console>:14

scala>  distData.reduce(_  +  _)
res0: Int = 75
```

Ejemplo1: Paralelización de una colección

```
scala> val data = Array(25, 20, 15, 10, 5)
data: Array[Int] = Array(25, 20, 15, 10, 5)
```

```
scala> val distData = sc.parallelize(data)
distData: org.apache.spark.rdd.RDD[Int] = ParallelCollectionRDD[0] at parallelize at
<console>:14
```

```
scala> distData.reduce(_ + _)
res0: Int = 75
```

Creamos la colleción

Ejemplo1: Paralelización de una colección

```
scala> val data = Array(25, 20, 15, 10, 5)
data: Array[Int] = Array(25, 20, 15, 10, 5)

scala> val distData = sc.parallelize(data)
distData: org.apache.spark.rdd.RDD[Int] = ParallelCollectionRDD[0] at parallelize at
<console>:14
```

```
scala> distData.reduce(_ + _)
res0: Int = 75
```

Action

Ejemplo 2: Utilizar Datasets

```
scala> val distFile = sc.textFile("README.md") distFile:
org.apache.spark.rdd.RDD[String] = MappedRDD[3] at textFile at <console>:12

scala> distFile.first res5:
String = #Apache Spark
```

Ejemplo 2: Utilizar Datasets

```
scala> val distFile = sc.textFile(“README.md”) distFile:
org.apache.spark.rdd.RDD[String] = MappedRDD[3] at textFile at <console>:12
```

```
scala> distFile.first
res5: String = #Apache Spark
```

Cargar archivo en variable

Ejemplo 2: Utilizar Datasets

```
scala> val distFile = sc.textFile(“README.md”)
```

distFile: org.apache.spark.rdd.RDD[String] = MappedRDD[3] at textFile at <console>:12

```
scala> distFile.first
res5: String = #Apache Spark
```

Action

Ejemplo 3: Utilizar Datasets de HDFS

scala> val distFile = sc.textFile("hdfs://mi_carpeta:9000/README.txt") distFile: org.apache.spark.rdd.RDD[String] = MappedRDD[5] at textFile at <console>:12

scala> distFile.first res2: String = For the latest information about Hadoop, please visit our website at:

Ejemplo 3: Utilizar Datasets de HDFS

```
scala> val distFile = sc.textFile(“hdfs://mi_carpeta:9000/README.txt”) distFile:
org.apache.spark.rdd.RDD[String] = MappedRDD[5] at textFile at <console>:12
```

scala> distFile.first res2: String = For the latest information about Hadoop, please visit our website at:

Cargar archivo de HDFS en variable

Ejemplo 3: Utilizar Datasets de HDFS

```
scala> val distFile = sc.textFile("hdfs://mi_carpeta:9000/README.txt")
distFile: org.apache.spark.rdd.RDD[String] = MappedRDD[5] at textFile at <console>:12
```

```
scala> distFile.first
res2: String = For the latest information about Hadoop, please visit our website at:
```

Action

RDD como una relación

```
// Definimos un esquema usando case class.
case class Person(name: String, age: Int)

// Creamos un RDD como Person, almacenado como una tabla.
val people =
  sc.textFile("examples/src/main/resources/people.txt")
     .map(_.split(","))
     .map(p => Person(p(0),  p(1).trim.toInt))

people.registerAsTempTable("people")
```

Consultas SQL

```
//  Una consulta SQL  dentro de un RDD
val teenagers =
  sql("SELECT name FROM people WHERE age >= 13 AND age <= 19")

// El resultado de la consulta como una acción en un RDD
val nameList = teenagers.map(t => "Name: " + t(0)).collect()

val teenagers =
 people.where("age >= 10").where("age <= 19").select("name")
```

Ejemplo 4: MapReduce Wordcount

```
val file = sc.textFile(“hdfs://…”)
```

```
val counts = file.flatMap(line => line.split(“ ”))
                 .map(word => (word, 1))
                 .reduceByKey(_ + _)
counts.saveAsTextFile(“hdfs://….”)
```

Ejemplo 4: MapReduce Wordcount

```
val file = spark.textFile(“hdfs://…”)
val counts = file.flatMap(line => line.split(“ ”))
                 .map(word => (word, 1))
                 .reduceByKey(_ + _)
counts.saveAsTextFile(“hdfs://….”)
```

Leemos de HDFS

Ejemplo 4: MapReduce Wordcount

```
val file = spark.textFile(“hdfs://…”)
val counts = file.flatMap(line => line.split(“ ”))
                 .map(word => (word, 1))
                 .reduceByKey(_ + _)
counts.saveAsTextFile(“hdfs://….”)
```

“MapReducimos” el texto

Ejemplo 4: MapReduce Wordcount

```
val file = spark.textFile("hdfs://…") val
counts = file.flatMap(line => line.split(" "))
                .map(word => (word, 1))
                .reduceByKey(_ + _)
counts.saveAsTextFile("hdfs://….")
```

Guardamos en HDFS

Ejemplo en Spark

```
val myFile = sc.textFile("s3n://bigdata-mex/XXXXX")
// wordcount
val counts = myFile.flatMap(line =>
line.toLowerCase().replace(".",   "   ").replace(",",   "   ").split("
")).map(word => (word, 1L)).reduceByKey(_ + _)
// creamos tuplas de palabras
val sorted_counts = counts.collect().sortBy(wc => -wc._2)
// imprimimos los primeros 10 elementos
sorted_counts.take(10).foreach(println)
// almacenamos el resultado en S3 bucket
sc.parallelize(sorted_counts).saveAsTextFile("s3n://bigdatauem/XXXX")
// si queremos almacenar en formato csv
val csvResults = sorted_counts map { case (key, value) =>
Array(key,  value).mkString(",\t")  }  //  guardamos  en
formato csv en S3
sc.parallelize(results).saveAsTextFile("s3n://bigdata-mex/XXX")
```

Ejemplo Spark /SQL

```
    val sqlContext = new
org.apache.spark.sql.SQLContext(sc) val wikiData =
    sqlContext.parquetFile("s3n://bigdata-
uem/data/wiki_parquet")
    wikiData.count()
    wikiData.registerTempTable("wikiData")
    val countResult = sqlContext.sql("SELECT COUNT(*)
FROM wikiData").collect() val sqlCount =
    countResult.head.getLong(0)
    sqlContext.sql("SELECT username, COUNT(*) AS cnt FROM
wikiData WHERE username <> '' GROUP BY username ORDER BY
cnt DESC LIMIT 10").collect().foreach(println)
```

Ejemplo Spark /SQL

```
import org.apache.spark.sql._ val people =
sc.textFile("s3n://bigdata-uem/data/people.txt") val
schemaString = "name age"
val schema =
  StructType(schemaString.split(" ").map(fieldName =>
StructField(fieldName, StringType, true)))
val rowRDD = people.map(_.split(",")).map(p => Row(p(0), p(1).trim))
val peopleSchemaRDD= sqlContext.applySchema(rowRDD, schema)
peopleSchemaRDD.registerTempTable("people")
val results = sqlContext.sql("SELECT name FROM people WHERE age >= 13
AND age <= 19") results.map(t => "Name: " +
t(0)).collect().foreach(println)
```

Ejemplo Spark /SQL

```
// sc is an existing SparkContext. val sqlContext = new
org.apache.spark.sql.SQLContext(sc) // Importing the SQL
context gives access to all the public SQL functions and
implicit conversions. import sqlContext._ val people:
RDD[Person] = ... // An RDD of case class objects, from the
first example.

// The following is the same as 'SELECT name FROM people
WHERE age >= 10 AND age <= 19'
val teenagers = people.where('age >= 10).where('age <=
19).select('name)
teenagers.map(t => "Name: " +
t(0)).collect().foreach(println)
```

Ejemplo Spark / MongoDB

```
import com.stratio.deep.mongodb.config.MongoDeepJobConfig
import
com.stratio.deep.mongodb.extractor.MongoNativeDBObjectExtractor
import com.stratio.deep.core.context.DeepSparkContext import
```

```
com.mongodb.DBObject import org.apache.spark.rdd.RDD import
com.mongodb.QueryBuilder import com.mongodb.BasicDBObject val host =
"localhost:27017"

val database = "test" val
inputCollection = "input";
val deepContext: DeepSparkContext = new DeepSparkContext(sc)
val inputConfigEntity: MongoDeepJobConfig[DBObject] = new MongoDeepJobConfig[DBObject](classOf[DBObject])

val query: QueryBuilder = QueryBuilder.start();
query.and("number").greaterThan(27).lessThan(30);

inputConfigEntity.host(host).database(database).collection(inputCollection).filterQuery(query).setExtractorI
m plClass(classOf[MongoNativeDBObjectExtractor]) val inputRDDEntity: RDD[DBObject] =
deepContext.createRDD(inputConfigEntity)
```

Ejemplo Spark /JSON

```
/ sc is an existing SparkContext. val sqlContext = new
org.apache.spark.sql.SQLContext(sc)

// 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 = "examples/src/main/resources/people.json" // Create a SchemaRDD from
the file(s) pointed to by path val people = sqlContext.jsonFile(path)
```

```
// The inferred schema can be visualized using the printSchema() method.
people.printSchema()
// root
//  |-- age: IntegerType
//  |-- name: StringType

// Register this SchemaRDD as a table.
people.registerTempTable("people")

// SQL statements can be run by using the sql methods provided by sqlContext. val
teenagers = sqlContext.sql("SELECT name FROM people WHERE age >= 13 AND age <= 19")

// Alternatively, a SchemaRDD can be created for a JSON dataset represented by
// an RDD[String] storing one JSON object per string. val anotherPeopleRDD =
sc.parallelize(
  """{"name":"Yin","address":{"city":"Columbus","state":"Ohio"}}""" :: Nil)
val anotherPeople = sqlContext.jsonRDD(anotherPeopleRDD)
```

Aplicaciones auto contenidas en Spark

- Pasos para empaquetar una aplicación autónoma utilizando la API de Spark.
- Scala (con SBT), Java (con Maven) y Python.

```
/* SimpleApp.scala */ import
org.apache.spark.SparkContext import
org.apache.spark.SparkContext._ import
org.apache.spark.SparkConf
```

```
object SimpleApp {
  def main(args: Array[String]) {
    val logFile = "s3n://bigdata-uem/warAndPeace.txt" val conf =
    new SparkConf().setAppName("Simple Application") val sc =
    new SparkContext(conf)
    val logData = sc.textFile(logFile, 2).cache() val numAs =
    logData.filter(line => line.contains("a")).count() val numBs =
    logData.filter(line => line.contains("b")).count() println("Lineas
    con a: %s, Lineas con b: %s".format(numAs, numBs))
  }
}
```

Aplicaciones auto contenidas en Spark

- Nuestra aplicación depende de la API de Spark
- Incluir un archivo de configuración sbt , simple.sbt que explica que Spark es una dependencia.
- Este archivo también añade las dependencias de repositorio que Spark

- `name := "Simple Project"`
- `version := "1.0"`
- `scalaVersion := "2.10.4"`
- `libraryDependencies += "org.apache.spark" %% "sparkcore" % "1.2.0"`

Aplicaciones auto contenidas en Spark

```
#El contenido del directorio actual tiene que parecerse a
$ find .
.
./simple.sbt
./src
./src/main
./src/main/scala
./src/main/scala/SimpleApp.scala

# Empaquetamos en jar nuestra aplicación
$ sbt package
...
[info] Packaging {..}/{..}/target/scala-2.10/simple-project_2.10-1.0.jar

# Usamos spark-submit para ejecutar la aplicación
```

```
$ spark-submit \
  --class "SimpleApp" \ --
  master yarn-client \
  target/scala-2.10/simple-project_2.10-1.0.jar
...
Lineas con a: 46, Lineas con b: 23
```