

conceptos

- La principal característica es almacenar grandes cantidades de datos en memoria entre trabajos
- Esta capacidad permite mejorar el rendimiento de MapReduce, donde los conjuntos de datos son siempre almacenados en disco
- Spark dispone de algoritmos iterativos, de tal manera que una función es aplicada a un dataset hasta que se cumpla una condición
- Tiene también análisis interactivo, lo que permite a un usuario hacer consultas sobre un dataset

conceptos

- Dispone de Apis en tres idiomas, SCALA, Java y Python
- Dispone de múltiples módulos para completar las herramientas
 - SQL DataFrames
 - Spark Streaming
 - Mlib (Machin Learning)
 - GraphX

conceptos

- Spark utiliza como MapReduce trabajos, pero con una visión más general
- Utiliza un motor DAG (directed acyclic graph) de etapas, que permite ejecutar listas de operaciones y traducirlas en trabajos. podría asimilarse al concepto de las tareas Map y Reduce
- RDD (Resilient Distributed DataSet) es una colección de objetos que está dividido por múltiples máquinas en un cluster
- Normalmente uno o varios RDD se crean como input y mediante transformaciones se convierten en el destino

Conceptos

- Las tareas se dividen en tareas por el runtime de Spark y se pueden ejecutar en paralelo en porciones de RDD a lo largo del cluster
- Los trabajos siempre se ejecutan en el contexto de una aplicación (una instancia de SparkContext)
- Permite agrupar variables y RDDs
- Una aplicación puede ejecutar más de un trabajo en serie o en paralelo y ofrece los mecanismos para que un trabajo pueda acceder a un RDD que haya sido cacheado por otro trabajo previo en la misma aplicación.
- Una sesión del Shell es una aplicación

Spark-Shell - Ejemplo

- Es un editor de comandos en Scala
- Al iniciar el editor genera una variable de contexto “sc” que es el punto de entrada a Spark

```
scala> val lines = sc.textFile("input/ncdc/micro-tab/sample.txt")  
lines: org.apache.spark.rdd.RDD[String] = MappedRDD[1] at textFile at  
<console>:12
```

Spark-Shell - Ejemplo

- Una vez cargado un RDD podemos realizar transformaciones. La primera es dividir las líneas en campos
- Utiliza el método map para separar por el campo que queremos
- Convierte de un string a un array de string de Scala

```
scala> val records = lines.map(_.split("\t"))  
records: org.apache.spark.rdd.RDD[Array[String]] = MappedRDD[2] at map at  
<console>:14
```

Spark-Shell Ejemplo

- La siguiente transformación es quitar los elementos que no nos interesen aplicando un filtro
- El método `filter()` toma un predicado que devuelve un booleano, en este caso que la temperatura no sea válida

```
scala> val filtered = records.filter(rec => (rec(1) != "9999"  
    && rec(2).matches("[01459]")))  
filtered: org.apache.spark.rdd.RDD[Array[String]] = FilteredRDD[3] at filter at  
<console>:16
```

Spark-Shell Ejemplo

- Para poder utilizar funciones de agrupación, necesitamos que el RDD esté en formato key/value, por lo que podemos hacer otra función map() con la transformación
- Map() puede utilizar expresiones Lambda

```
scala> val tuples = filtered.map(rec => (rec(0).toInt, rec(1).toInt))  
tuples: org.apache.spark.rdd.RDD[(Int, Int)] = MappedRDD[4] at map at  
<console>:18
```


Spark-Shell Ejemplo

- Para obtener la temperatura máxima, tenemos que agrupar la información, para ello transformamos el RDD utilizando el método `ReduceByKey()`. Ahora lo podemos utilizar ya que tenemos el RDD en formato `key/value`
- La temperatura máxima se obtiene con la función de java `Math.max`

```
scala> val maxTemps = tuples.reduceByKey((a, b) => Math.max(a, b))
maxTemps: org.apache.spark.rdd.RDD[(Int, Int)] = MapPartitionsRDD[7] at
reduceByKey at <console>:21
```

Spark-Shell Ejemplo

- La salida del resultado se puede hacer por pantalla, con `println()` utilizando la función `foreach()` para recorrer la variable con el resultado.
- Esta operación hace que Spark genere otro trabajo para que procese los valores y se puedan presentar a través de la función `println()`

```
scala> maxTemps.foreach(println(_))  
(1950,22)  
(1949,111)
```

Spark-Shell Ejemplo

- Otra alternativa es guardar el resultado en un archivo utilizando el método `SaveAsTextFile()`
- Este método crea un directorio con el archivo resultante
- El método `SaveAsTextFile()` genera un trabajo de Spark, que lo escribe.

```
scala> maxTemps.saveAsTextFile("output")
```

```
% cat output/part-*  
(1950,22)  
(1949,111)
```

Scala ejemplo código

```
import org.apache.spark.SparkContext._
import org.apache.spark.{SparkConf, SparkContext}

object MaxTemperature {
  def main(args: Array[String]) {
    val conf = new SparkConf().setAppName("Max Temperature")
    val sc = new SparkContext(conf)

    sc.textFile(args(0))
      .map(_._split("\t"))
      .filter(rec => (rec(1) != "9999" && rec(2).matches("[01459]")))
      .map(rec => (rec(0).toInt, rec(1).toInt))
      .reduceByKey((a, b) => Math.max(a, b))
      .saveAsTextFile(args(1))
  }
}
```

Scala Ejemplo ejecución

- Spark-Submit ejecuta el programa, pasando la aplicación JAR, y los comandos de input con el archivo y la salida en el directorio output
- --class indica el nombre de la aplicación
- --master indica dónde se tiene que ejecutar, en este caso en una JVM en local

```
% spark-submit --class MaxTemperature --master local \  
  spark-examples.jar input/ncdc/micro-tab/sample.txt output  
% cat output/part-*
```

(1950,22)
(1949,111)

RDD Resilient distributed dataset

- RDD son los elementos claves en los trabajos de Spark
- Procesos a tener en cuenta con RDDs son
 - Creación
 - Transformación
 - Persistencia
 - Serialización

RDD - Creación

- Se pueden crear de tres maneras distintas
 - A partir de un objeto en memoria
 - Obtener un conjunto de datos por ejemplo de HDFS
 - Transformando un RDD existente

RDD Creación

- A partir de un objeto en memoria es interesante para realizar procesos de computación intensiva con pequeñas cantidades de información en paralelo
- El nivel de paralelismo se indica en la propiedad `spark.default.parallelism` que tiene como valor por defecto en local el número de cores de la máquina y en cluster el de todas las máquinas
- El método `Parallelize` indica el número de procesos paralelos

```
val params = sc.parallelize(1 to 10)
val result = params.map(performExpensiveComputation)
```


RDD - Creación

- Crearlo como una referencia de un dataset externo en un directorio local o en HDFS
- Utiliza internamente TextInputFormat de la API de MapReduce para leer la información, por lo que la división del archivo es la misma que en Hadoop, por lo que en el caso de HDFS se hace una partición por cada bloque de HDFS

```
val text: RDD[String] = sc.textFile(inputPath)
```

RDD - CREACIÓN

- También se puede leer todo el archivo en memoria como pares dónde se especifica la ruta del archivo y el contenido, esto solo es posible para archivos pequeños, ya que traslada todo el contenido a la memoria

```
val files: RDD[(String, String)] = sc.wholeTextFiles(inputPath)
```

RDD - transformaciones

- Spark tiene dos categorías
 - Transformaciones que generan un nuevo RDD a partir de uno existente
 - Acciones que realiza un proceso dentro del RDD y realiza una tarea con el resultado como mostrarlo al usuario o almacenarlo en disco
- Las acciones tienen un resultado inmediato, mientras que las transformaciones esperan a que se realice una acción en el RDD transformado
- Una manera de distinguirlas es mirar el tipo devuelto, si es RDD es una transformación, en cualquier otro caso es una acción.

RDD – Transformaciones Ejemplo

- Convertimos en minúsculas un RDD

```
val text = sc.textFile(inputPath)
val lower: RDD[String] = text.map(_.toLowerCase())
lower.foreach(println(_))
```

- Esta función no se ejecuta hasta que el método foreach es llamado, generando Spark un trabajo que lee el archivo de entrada y ejecuta la función en cada línea antes de imprimirla

RDD Transformaciones

- Hay tres funciones de agrupación que trabajan con pares key/value, , a partir de un conjunto de datos obtienen un valor único para cada clave
- ReducedByKey(), FoldByKey(), AgregateByKey()

```
val pairs: RDD[(String, Int)] =  
    sc.parallelize(Array(("a", 3), ("a", 1), ("b", 7), ("a", 5)))  
val sums: RDD[(String, Int)] = pairs.reduceByKey(_+_)  
assert(sums.collect().toSet == Set(("a", 9), ("b", 7)))
```

Rdd persistencia

- Cache() permite almacenar en memoria un RDD intermedio
- Al llamar al método cache() no cachea la información hasta que realizamos una acción, simplemente lo marca como cacheable.
- BlockManagerInfo muestra información de las particiones y del número de RDD en memoria
- No tiene que volver a cargarlo desde archivo como en MapReduce
- Existen diferentes tipos de persistencia MEMORY_ONLY, MEMORY_ONLY_SER, MEMORY_AND_DISK, MEMORY_AND_DISK_SER

Rdd persistencia ejemplo

```
scala> val tuples = filtered.map(rec => (rec(0).toInt, rec(1).toInt))
tuples: org.apache.spark.rdd.RDD[(Int, Int)] = MappedRDD[4] at map at
<console>:18
```

```
scala> tuples.cache()
res1: tuples.type = MappedRDD[4] at map at <console>:18
```

```
scala> tuples.reduceByKey((a, b) => Math.max(a, b)).foreach(println(_))
INFO BlockManagerInfo: Added rdd_4_0 in memory on 192.168.1.90:64640
INFO BlockManagerInfo: Added rdd_4_1 in memory on 192.168.1.90:64640
(1950,22)
(1949,111)
```

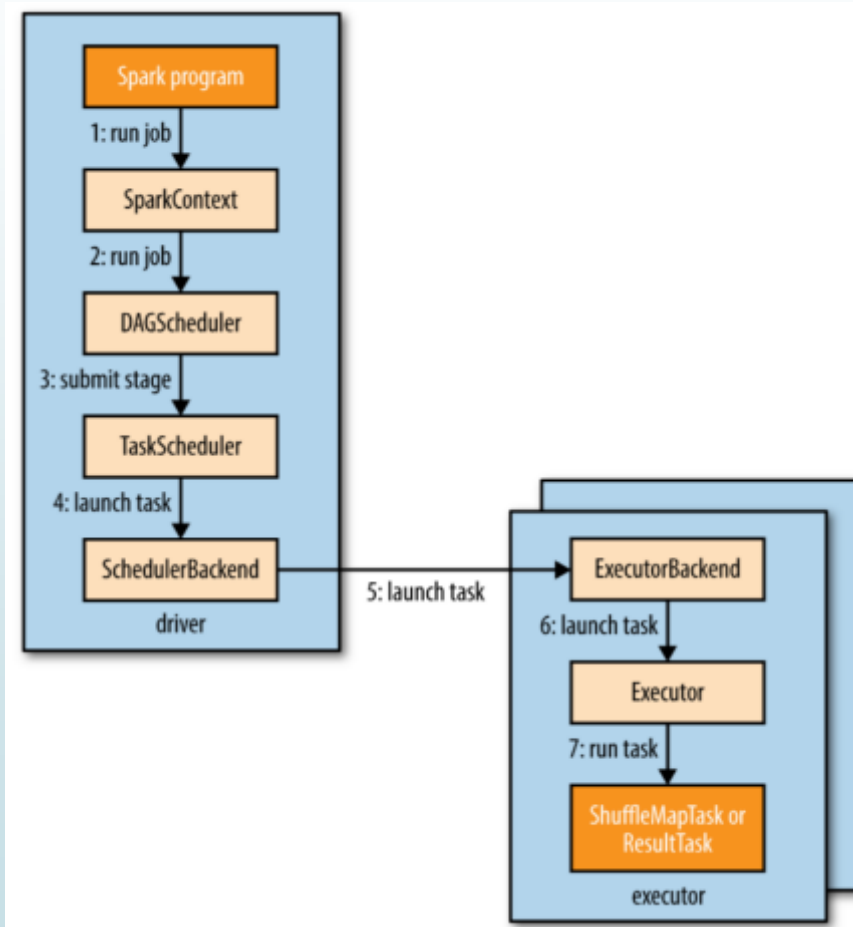
RDD Serialización

- Tenemos que tener en cuenta serialización de datos y serialización de funciones
- Serialización de datos utiliza por defecto la implementación `java.io.Serializable` o `java.io.Externalizable`
 - Se pueden utilizar otras serializaciones como Kryo, hay que modificar la configuración
 - `conf.set("spark.serializer", "org.apache.spark.serializer.KryoSerializer")`
- La serialización de funciones la realiza de forma automática Spark

Funcionamiento de los trabajos en spark

- Existen dos entidades independientes
 - Driver: que contiene la aplicación en ejecución. (sparkContext) y organizar la ejecución de las tareas
 - Executor: que es exclusivo de la aplicación y ejecuta las tareas
- Cuando se ejecuta un trabajo, se programa la ejecución y se divide en los procesos en etapas DAG y el programador que gestiona la ejecución de las etapas

Funcionamiento de los trabajos en spark

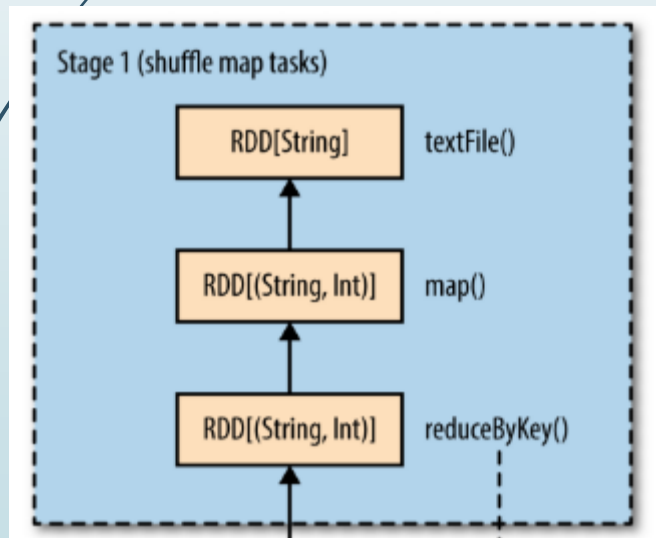


Funcionamiento de los trabajos en spark

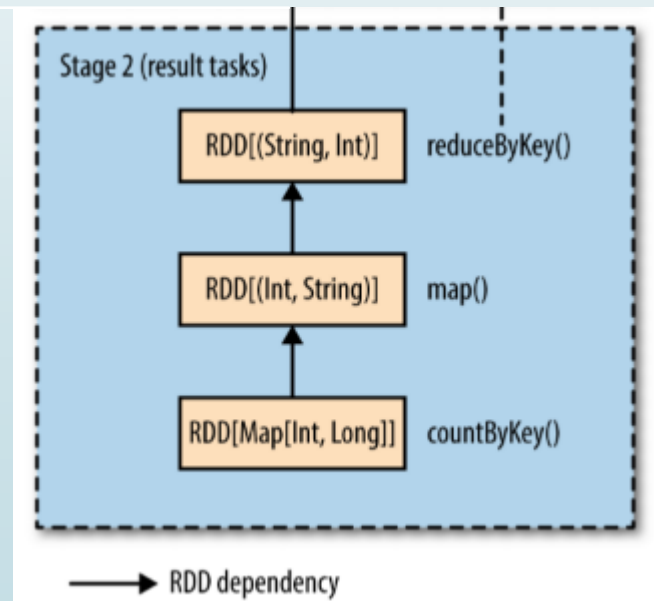
- Programador DAG tiene la función de dividir los trabajos en etapas.
- Hay dos tipos de tareas
 - Tareas mapa mezclado (shuffle), similares a las tareas Map, ejecutan procesos en un RDD y el resultado lo escriben en una nueva partición en RDD. Estas tareas se ejecutan en todas las etapas menos en la final
 - Tareas de resultado, se ejecutan en la etapa final, que devuelve el resultado al programa

Funcionamiento de los trabajos en spark

```
val hist: Map[Int, Long] = sc.textFile(inputPath)
  .map(word => (word.toLowerCase(), 1))
  .reduceByKey((a, b) => a + b)
  .map(_._swap)
  .countByKey()
```



Introducción al Big Data con Hadoop



Funcionamiento de los trabajos en spark

- Programación de tareas, cuando al programador de tareas le envía un conjunto de tareas, usa su lista de executors y construye una mapa de ejecución para las tareas
- Luego asigna las tareas a cada executor que tiene cores libres hasta que las tareas están completas
- El orden de asignación de las tareas a cada executor es primero las tareas locales, luego tareas del nodo y por último tareas remotas

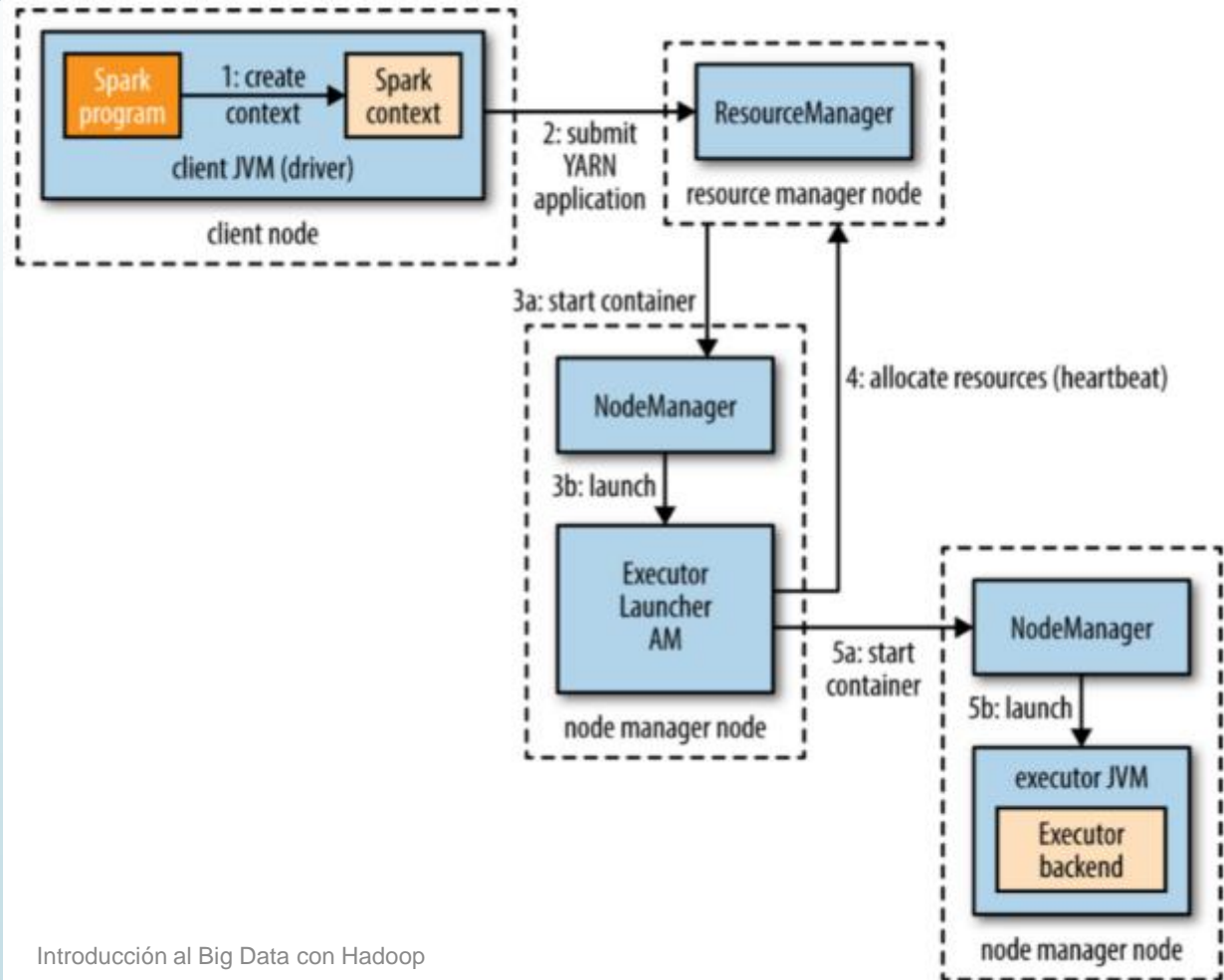
Funcionamiento de los trabajos en spark

- Ejecución de tareas
- El executor ejecuta las tareas siguiendo un orden
 - Primero se asegura que el JAR y los archivos de dependencias están actualizados
 - Deserializa el código de las tareas, que se le envió junto con la tarea
 - Se ejecuta el código de la tarea en la misma JVM que el executor

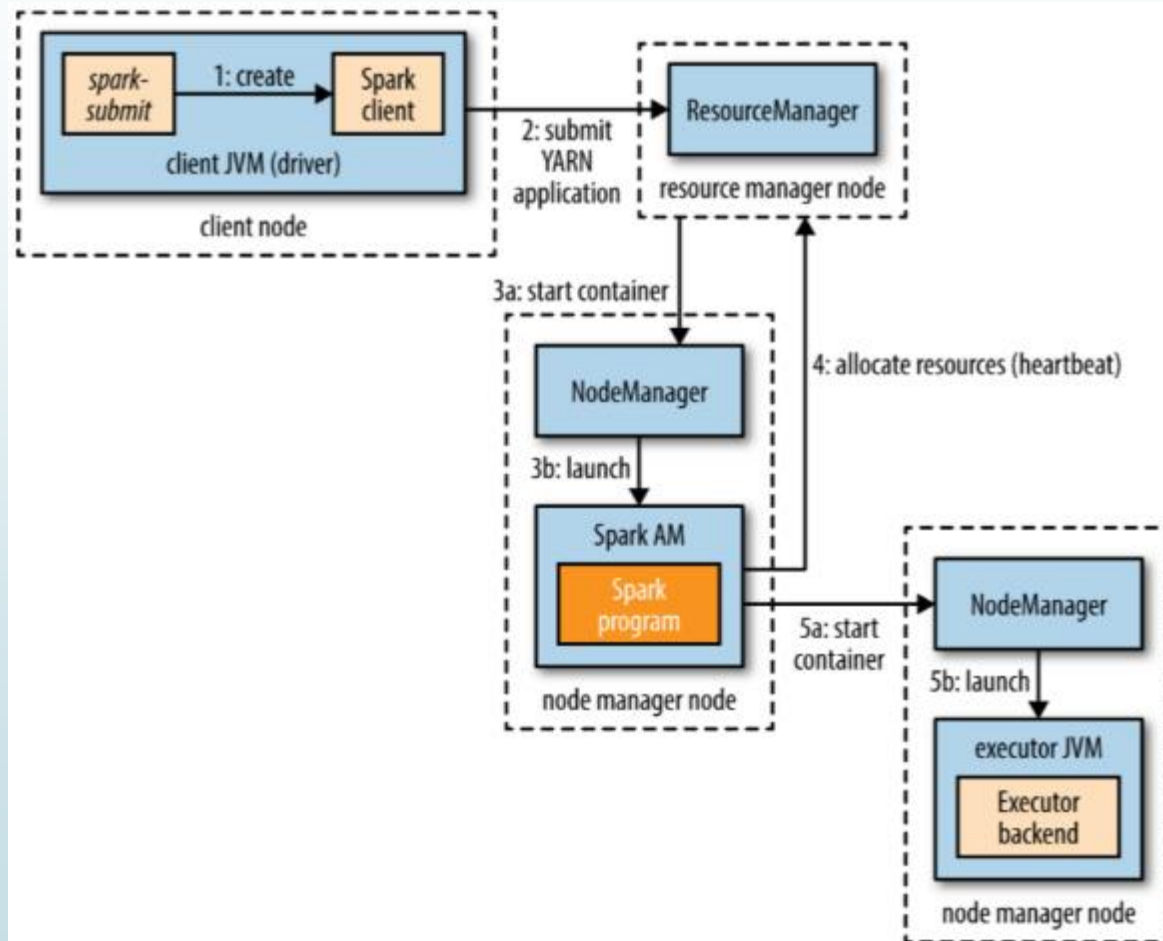
Spark con YARN

- ▶ Permite la mayor integración con otras herramientas Hadoop
- ▶ YARN Client Mode, cuando el driver se ejecuta en el cliente
 - ▶ Se utiliza en clientes que utilizan modos interactivos como Spark-Shell
- ▶ YARN cluster Mode, cuando el driver se ejecuta en el master application del cluster
 - ▶ Adecuado para entornos de producción, ya que la aplicación se ejecuta completamente en el servidor

Spark con YARN Client



Spark con YARN cluster




SPARK - Conclusiones

- Framework de desarrollo con kernel propio para trabajar con HDFS y YARN directamente sin depender de MapReduce
- Complementarlo con conjunto de herramientas como Spark Streaming


SPARK SQL, DataFrames y DataSet

- RDD es la abstracción más básica de Spark
 - Dependencias, indica como construir el RDD
 - Particiones, divide el trabajo para paralelizar el procesamiento
 - Funciones integradas
- Spark Structured APIs
 - Patrones para el proceso de datos. Filtros, selecciones, contar, agregaciones...
 - Java, Python, Spark R, SQL
 - Aplicación de esquema a los datos



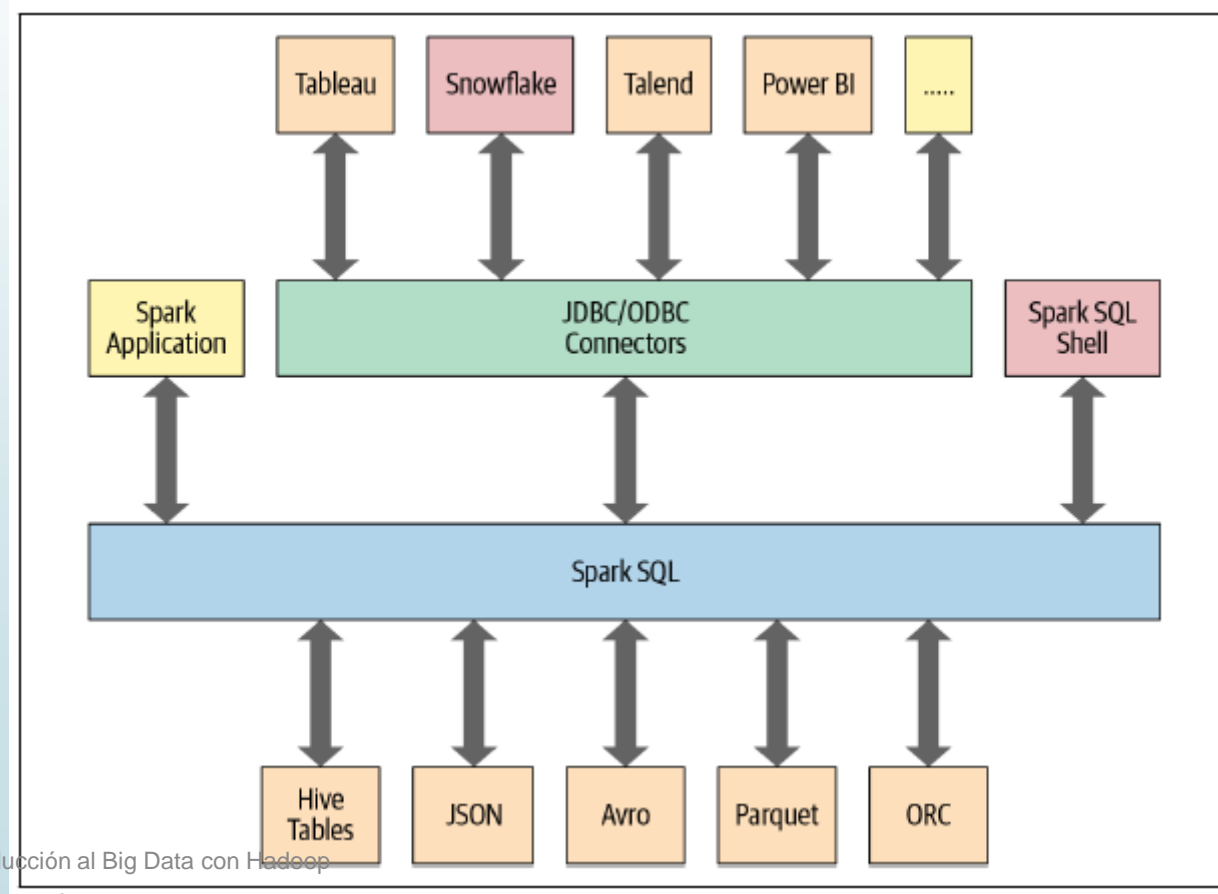
```
# In Python
# Create an RDD of tuples (name, age)
dataRDD = sc.parallelize([("Brooke", 20), ("Denny", 31), ("Jules", 30),
    ("TD", 35), ("Brooke", 25)])
# Use map and reduceByKey transformations with their lambda
# expressions to aggregate and then compute average

agesRDD = (dataRDD
    .map(lambda x: (x[0], (x[1], 1)))
    .reduceByKey(lambda x, y: (x[0] + y[0], x[1] + y[1]))
    .map(lambda x: (x[0], x[1][0]/x[1][1])))
```



name	avg(age)
Brooke	22.5
Jules	30.0
TD	35.0
Denny	31.0

Spark SQL, DataFrames, DATASET



Data type	Value assigned in Python	API to instantiate
ByteType	int	DataTypes.ByteType
ShortType	int	DataTypes.ShortType
IntegerType	int	DataTypes.IntegerType
LongType	int	DataTypes.LongType
FloatType	float	DataTypes.FloatType
DoubleType	float	DataTypes.DoubleType
StringType	str	DataTypes.StringType
BooleanType	bool	DataTypes.BooleanType
DecimalType	decimal.Decimal	DecimalType

Data type	Value assigned in Python	API to instantiate
BinaryType	bytearray	BinaryType()
TimestampType	datetime.datetime	TimestampType()
DateType	datetime.date	DateType()
ArrayType	List, tuple, or array	ArrayType(dataType, [nullable])
MapType	dict	MapType(keyType, valueType, [nullable])
StructType	List or tuple	StructType([fields])
StructField	A value type corresponding to the type of this field	StructField(name, dataType, [nullable])

```
from pyspark.sql.types import *  
schema = StructType([StructField("author", StringType(), False),  
                      StructField("title", StringType(), False),  
                      StructField("pages", IntegerType(), False)])
```

```
schema = "author STRING, title STRING, pages INT"
```

```
// Create a DataFrame by reading from the JSON file  
// with a predefined schema  
val blogsDF = spark.read.schema(schema).json(jsonFile)  
// Show the DataFrame schema as output  
blogsDF.show(false)
```

Dataframe col - row

- Col, devuelve un tipo Column
 - .WithColumns() permite trabajar con columnas
- Row, devuelve un objeto con una o varias columnas
 - Permite recorrer la colección de columnas con un índice
 - Puede usarse como método para crear DataFrames

```
# In Python
rows = [Row("Matei Zaharia", "CA"), Row("Reynold Xin", "CA")]
authors_df = spark.createDataFrame(rows, ["Authors", "State"])
authors_df.show()
```


DAtaFrame Operaciones comunes

- DataFrameReader. Interfaz para leer información
- DataFrameWriter. Interfaz para escribir información en múltiples formatos

```
# Use the DataFrameReader interface to read a CSV file  
sf_fire_file = "/databricks-datasets/learning-spark-v2/sf-fire/sf-fire-calls.csv"  
fire_df = spark.read.csv(sf_fire_file, header=True, schema=fire_schema)
```

SPARK DataSet

- Siempre necesitamos tener el esquema de los datos. DataFrames permite trabajar con datos sin estructura.
- Utilizado por Java y Scala
- Dispone de funciones a alto nivel para realizar operaciones

Ejemplos

- 9 Ejecución de aplicación con SQL DataFrames y Databricks
- 10 Integración con herramientas de BI

SPARK SQL – Instrucciones SQL

- Crea las tablas en la base de datos que se esté utilizando

- Utilizar

```
spark.sql("CREATE DATABASE learn_spark_db")  
spark.sql("USE learn_spark_db")
```

- Tiene el equivalente en DataFrame API

```
spark.sql("CREATE TABLE managed_us_delay_flights_tbl (date STRING, delay INT,  
distance INT, origin STRING, destination STRING)")
```

```
# In Python  
# Path to our US flight delays CSV file  
csv_file = "/databricks-datasets/learning-spark-v2/flights/departuredelays.csv"  
# Schema as defined in the preceding example  
schema="date STRING, delay INT, distance INT, origin STRING, destination STRING"  
flights_df = spark.read.csv(csv_file, schema=schema)  
flights_df.write.saveAsTable("managed_us_delay_flights_tbl")
```

Spark SQL UDF (User Defined functions)

- Funciones de usuario.
- Se generan para la sesión activa y no perduran en el tiempo
- Se crean con “def” <nombre> = <codigo spark>
- @udf, permite definir una función y convertirla a udf en un solo paso

```
states = ["CA", "TX", "NY", "WA"]

@udf(returnType=StringType())
def random_state():
    return str(random.choice(states))
```

Spark Streaming

- Extensión del core de Spark
- Ofrece procesamiento de datos de live stream con tolerancia a fallos y alta escalabilidad
- Admite una gran cantidad de orígenes de datos Kafka, Flume, Kinesis o TCP sockets
- Permite procesar la información utilizando algoritmos de alto nivel como map, reduce, join y window
- La salida se puede redirigir al sistema de archivos, bases de datos

Conceptos



Conceptos

- Spark streaming recibe los datos en live streaming
- Divide la tabla en bloques
- Estos bloques son procesados por el motor de Spark Stream
- Generan bloques de información procesada
- Utiliza un nivel alto de abstracción para la serilización llamado Discretized Stream (Dstream)

Conceptos



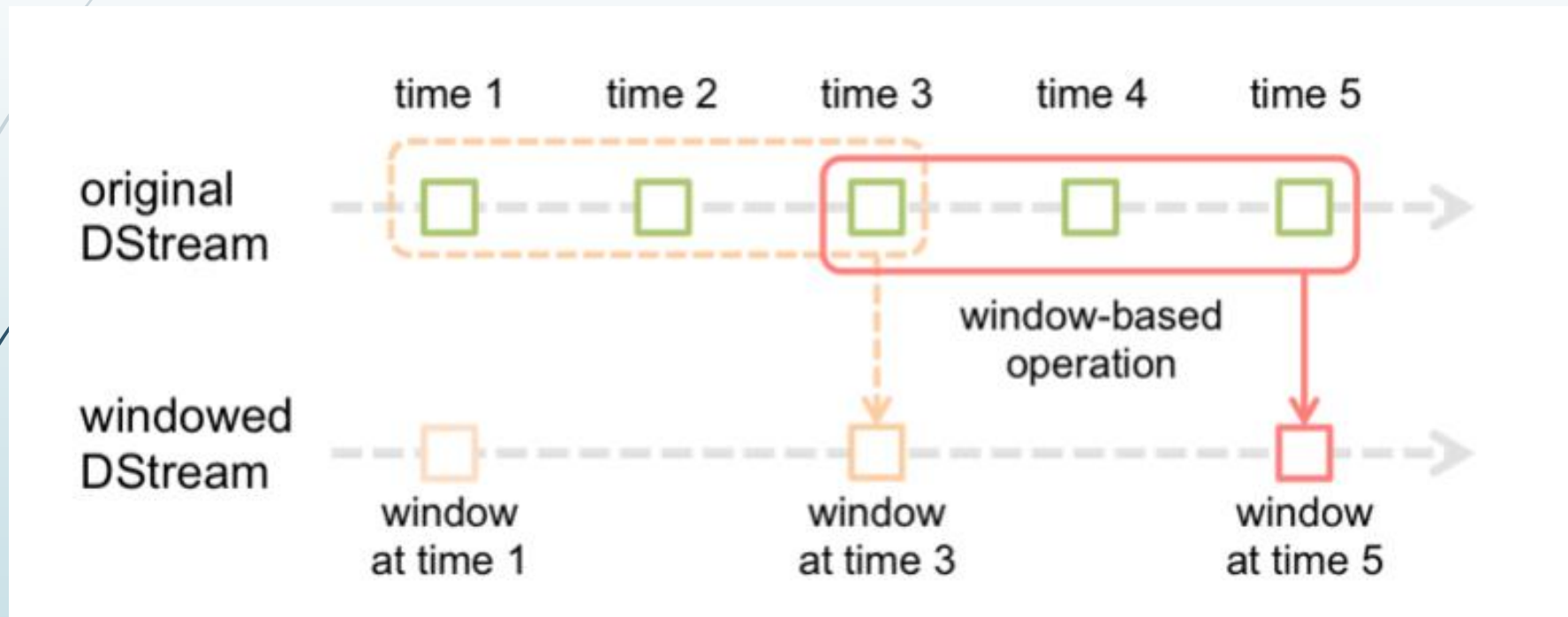
Elementos de Programación

- De forma similar a otros entornos, lo primero es crear un objeto contexto de la clase `SparkConf`. El espacio de nombres es `org.apache.spark.streaming`
- Definir el origen de datos para `Dstream`
- Definir las tareas de computación y transformación para `Dstream`
- Iniciar la recepción de los datos con `streamingContext.start()`
- Esperar que el proceso acabe (manualmente o por error) `streamingContext.stop()` o `streamingContext.awaitTermination()`

Procesamiento Window

- Permite establecer procesos y transformaciones sobre una ventana de datos en movimiento
- Cada vez que la ventana se mueve por el stream, los RDDs se combinan y se operan para producir el RDD de la ventana
- Hay que definir la longitud de la ventana y cada cuanto se agrupa la información para procesarla.
- Estos dos parámetros tienen que ser múltiplos del que define la división en bloques del stream

Procesamiento Window



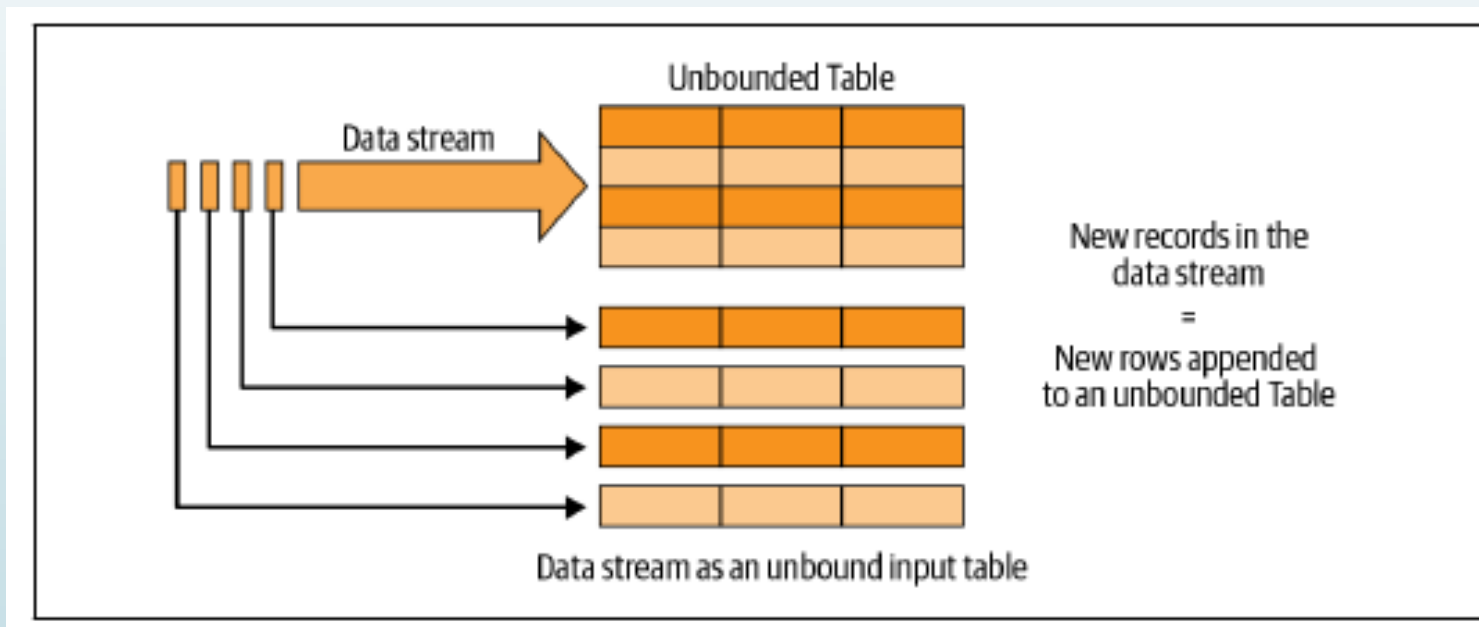
Procesamiento Window

- Spark Streaming dispone de varias funciones para procesar RDDs
 - `window(windowLength, slideInterval)` → Dstream para trabajar
 - `countByWindow(windowLength, slideInterval)` → cuenta los elementos en la ventana
 - `reduceByWindow(func, windowLength, slideInterval)` → devuelve un stream con los elementos en ese intervalo aplicando una función
 -

```
// Reduce last 30 seconds of data, every 10 seconds  
val windowedWordCounts = pairs.reduceByKeyAndWindow((a:Int,b:Int) => (a + b), Seconds(30), Seconds(10))
```

Spark Structured Streaming

- Cada registro se agrega a la tabla de datos



Fundamentos structured streaming query

- Crear DataFrames con `spark.ReadStream` para crear un `DataStreamReader`
- Procesos de transformación son los mismos
- Escribir resultados con `spark.WriteStream`
- Definir procesamiento de los datos. Trigger (default, Processing Time, Once, Continuous, CheckPoint location)
- Iniciar query

Código de ejemplo

```
# In Python
from pyspark.sql.functions import *
spark = SparkSession...
lines = (spark
    .readStream.format("socket")
    .option("host", "localhost")
    .option("port", 9999)
    .load())

words = lines.select(split(col("value"), "\\s").alias("word"))
counts = words.groupBy("word").count()
checkpointDir = "...
streamingQuery = (counts
    .writeStream
    .format("console"))
```

```
.outputMode("complete")
.trigger(processingTime="1 second")
.option("checkpointLocation", checkpointDir)
.start()
streamingQuery.awaitTermination()

// In Scala
import org.apache.spark.sql.functions._
import org.apache.spark.sql.streaming._
val spark = SparkSession...
val lines = spark
    .readStream.format("socket")
    .option("host", "localhost")
    .option("port", 9999)
    .load()

val words = lines.select(split(col("value"), "\\s").as("word"))
val counts = words.groupBy("word").count()

val checkpointDir = "...
val streamingQuery = counts.writeStream
    .format("console")
    .outputMode("complete")
    .trigger(Trigger.ProcessingTime("1 second"))
    .option("checkpointLocation", checkpointDir)
    .start()
streamingQuery.awaitTermination()
```


Data Lakes

- Solución de almacenamiento distribuida.
- Permite grandes cantidades de datos
- Apache Spark soporta diferentes Data Lakes
- Delta Lake, proyecto open source de los creadores de Apache Spark
 - Soporta funciones ACID

Delta Lake

- Cargar datos en una tabla Delta Lake, indicar en spark.write el formato “delta”

- Agregar una vista a la ta

```
spark
  .read
  .format("parquet")
  .load(sourcePath)
  .write
  .format("delta")
  .save(deltaPath)
```

```
# Create a view on the data called loans_delta
spark.read.format("delta").load(deltaPath).createOrReplaceTempView("loans_delta")
```

Delta Lake con Streams

- ▶ Permite agregar información desde archivos y streams a la misma tabla
- ▶ Se puede añadir información con múltiples streams
- ▶ Garantiza los procesos ACID (actualizar, insertar y borrar)
- ▶ Permite forzar esquemas a la hora de escribir para prevenir errores

Delta Lake Transformaciones

- Actualizaciones, método update

```
# In Python
from delta.tables import *

deltaTable = DeltaTable.forPath(spark, deltaPath)
deltaTable.update("addr_state = 'OR'", {"addr_state": "'WA'"})
```

- Borrado selectivo de información

```
# In Python
deltaTable = DeltaTable.forPath(spark, deltaPath)
deltaTable.delete("funded_amnt >= paid_amnt")
```

- Combinación datos, merge

```
# In Python
(deltaTable
 .alias("t")
 .merge(loanUpdates.alias("s"), "t.loan_id = s.loan_id")
 .whenMatchedUpdateAll()
 .whenNotMatchedInsertAll()
 .execute())
```

Delta Lake - histórico

- Guarda un registro de transacciones donde guarda los commit realizados

```
// In Scala/Python  
deltaTable.history().show()
```

version	timestamp	operation	operationParameters
5	2020-04-07	MERGE	[predicate -> (t.`loan_id` = s.`loan_id`)]
4	2020-04-07	MERGE	[predicate -> (t.`loan_id` = s.`loan_id`)]
3	2020-04-07	DELETE	[predicate -> ["(CAST(`funded_amnt` ...

Delta Lake Versiones previas

- Dispone de instantáneas de versiones anteriores con `timestampAsOf` y `versionAsOf`

```
# In Python
(spark.read
 .format("delta")
 .option("timestampAsOf", "2020-01-01") # timestamp after table creation
 .load(deltaPath))

(spark.read.format("delta")
 .option("versionAsOf", "4")
 .load(deltaPath))
```