# Computación distribuida

Sergio Nesmachnow
(sergion@fing.edu.uy)

UNIVERSIDAD
DE LA REPÚBLICA
URUGUAY

# Computación distribuida

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 masivos 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

- 'Tercera generación' de sistemas de computación distribuida para procesamiento de grandes volúmenes de infrormación.
- Objetivos: afrontar problemas fuera de línea y en tiempo real.
- Explotar eficientemente datos en memoria RAM de la misma manera que datos en disco.
- Reutilizar los componentes útiles de la generación previa:
  - Sistemas de archivos distribuidos (HDFS).
  - Manejadores de procesos y clusters distribuidos: Apache Mesos/YARN.
- Interoperable con Hadoop.

# Apache Spark

- Framework general y eficiente para procesamiento de grandes volúmenes de datos.
- Es genérico: no solo prove MapReduce, sino un conjunto amplio de operaciones (transformaciones y acciones).
- De código abierto.
- Trabaja en el espacio de usuario.
- Ejecuta sobre diversos sistemas distribuidos:
  - Hadoop (1.X, 2.X).
  - Apache Mesos.
  - Un cluster propio.
- Tiene una API simple (comparada con la API Java de MapReduce).
- APIs para Scala, Python, Java, SQL, R.

# Apache Spark

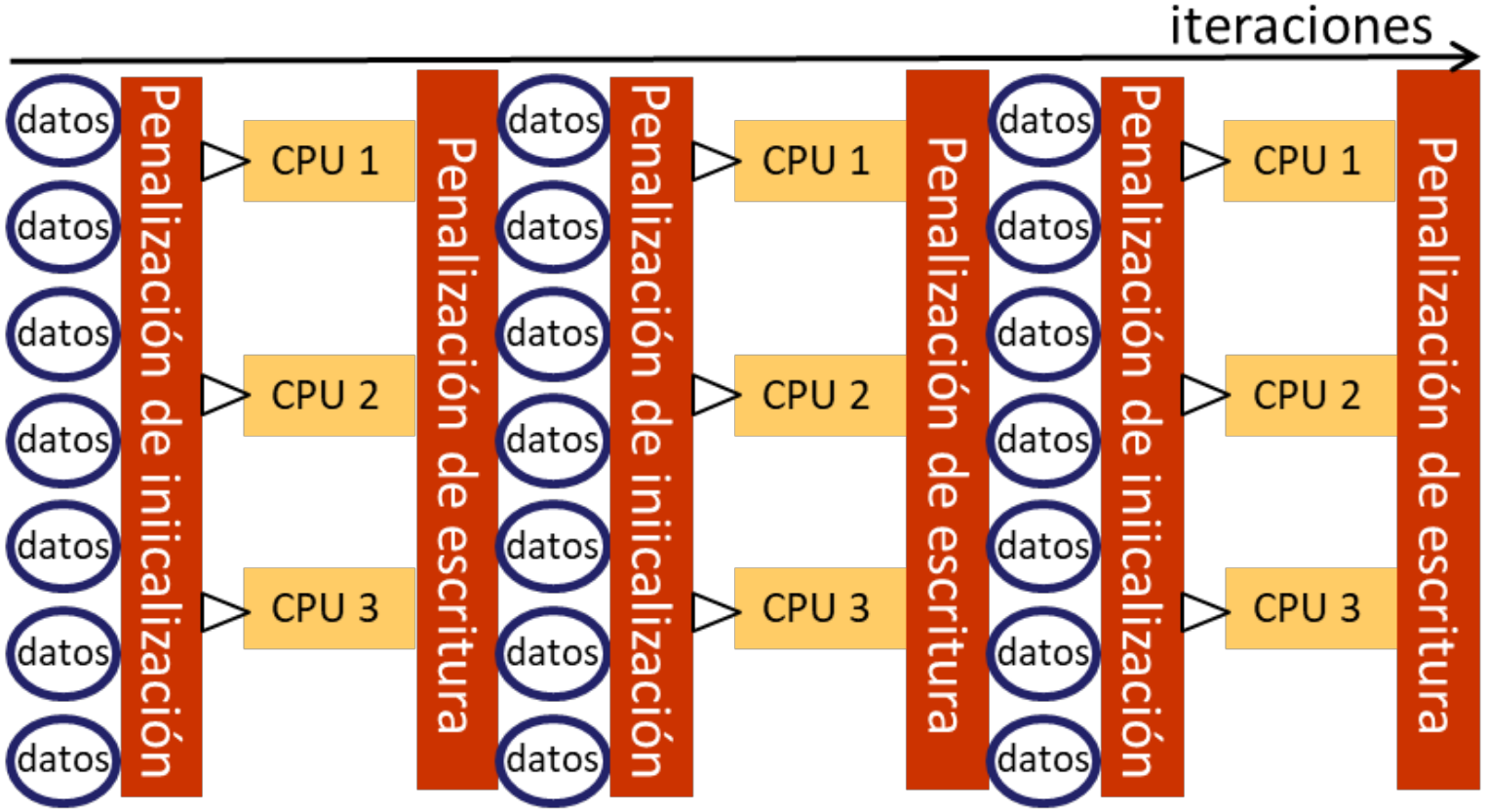
- Puede acceder a datos de Hadoop (HDFS).
- Puede usar muchos otros sistemas como fuente o destino de datos, incluyendo:
  - Sistemas de archivo locales o en red.
  - Repositorios de objetos (como Amazon S3 o Ceph).
  - Bases de datos relacionales.
  - Bases de datos NoSQL, incluyendo Apache Cassandra, HBase y otras.
  - Sistemas de mensajes (como Kafka).

# Apache Spark

- Ideado para resolver eficientemente:
  - Programas iterativos.
  - Consultas interactivas.
- Unifica procesamiento fuera de línea y streaming (tiempo real).
- Tipos de aplicaciones adaptadas para usar Spark:
  - Aplicaciones Extract-Transform-Load (ETL).
  - Análisis predictivo y aprendizaje computacional.
  - Operaciones de acceso a datos (consultas SQL, visualizaciones).
  - Procesamiento de eventos en tiempo real y streaming.
  - Procesamiento y minería de texto, aplicaciones de grafos, reconocimiento de patrones, motores de recomendación.

# Apache Spark

- Ideado para resolver eficientemente aplicaciones encadenadas e interactivas, que Hadoop no resuelve eficientemente.
  - En Hadoop las comunicaciones son muy costosas (involucran transferencia de datos) y el encadenamiento implica crear nuevos procesos.



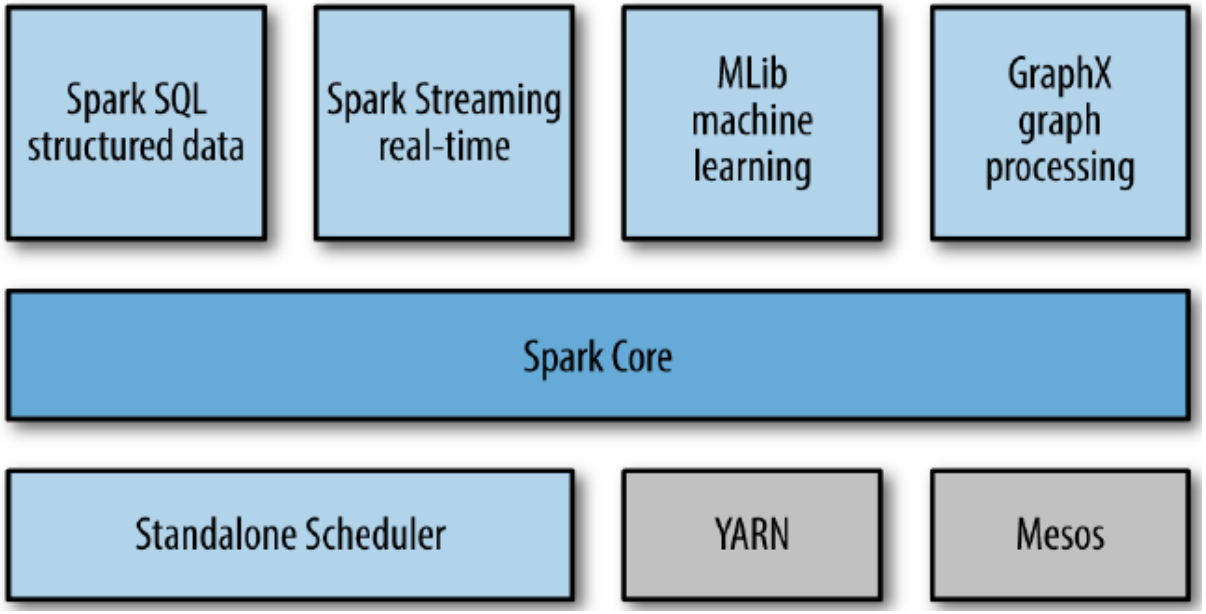
# Apache Spark

- Basado en el concepto de Resilient Distributed Dataset (RDD).
- Abstracción de datos que pueden residir en disco o en RAM.
- Particiona (distribuye/paraleliza) los datos.
- Basada en transformaciones de los datos (cada operación crea un nuevo RDD con datos transformados).
- Un RDD también se puede crear desde un archivo o repositorio externo utilizando un InputFormat de Hadoop.
- Implementa un modelo de Single Operation Multiple Data.

# Apache Spark

- Cálculos en memoria:
  - Se pueden almacenar (‘cachear’) datos de HDFS en la memoria de los workers para realizar el análisis directamente en memoria, aplicar barajado y ordenamiento, etc.
  - Permite que Spark tenga unas velocidades de procesamiento del orden de 100 veces más rápidas que las de MapReduce en Hadoop.
- Es también más eficiente que MapReduce de Hadoop para aplicaciones que ejecutan en disco.
- Provee tolerancia a fallos.

# Apache Spark: ecosistema




- Spark también se integra con Hadoop y los productos de su ecosistema

| Hadoop | Spark |
|---|---|
| Hive | SparkSQL |
| Apache Mahout | MLLib |
| Impala | SparkQSL |
| Apache Giraph | Graphax |
| Apache Storm | Spark streaming |

# Apache Spark: uso interactivo

## Interfaz de comandos (shell) de PySpark

# Apache Spark: uso interactivo

## Interfaz de comandos (shell) de Scala

```
[root@mycluster ~]# spark-shell
15/11/17 00:18:14 WARN NativeCodeLoader: Unable to load native-hadoop library fo
r your platform... using builtin-java classes where applicable
Welcome to
      ____              __
     / __/__  ___ _____/ /__
    _\ \/ _ \/ _ `/ __/  '_/
   /___/ .__/\_,_/_/ /_/\_\   version 1.3.1
      /_/

Using Scala version 2.10.4 (OpenJDK 64-Bit Server VM, Java 1.7.0_85)
Type in expressions to have them evaluated.
Type :help for more information.
Spark context available as sc.
SQL context available as sqlContext.

scala>
```

# Apache Spark: instalación y modos

1. Spark Standalone.
2. Spark sobre YARN (Hadoop): cliente (para trabajos interactivos, por ejemplo con pyspark o sparkshell).
3. Spark sobre YARN (Hadoop): cluster (para trabajos no interactivos).
4. Spark sobre Mesos.
5. Spark en infraestructuras cloud.

Cuidado: Spark tiene un uso muy intensivo de memoria. Ejecutarlo sobre YARN/Mesos alojará containers y recursos (CPU y memoria) que pueden afectar la ejecución de otras aplicaciones ejecutando en el cluster.

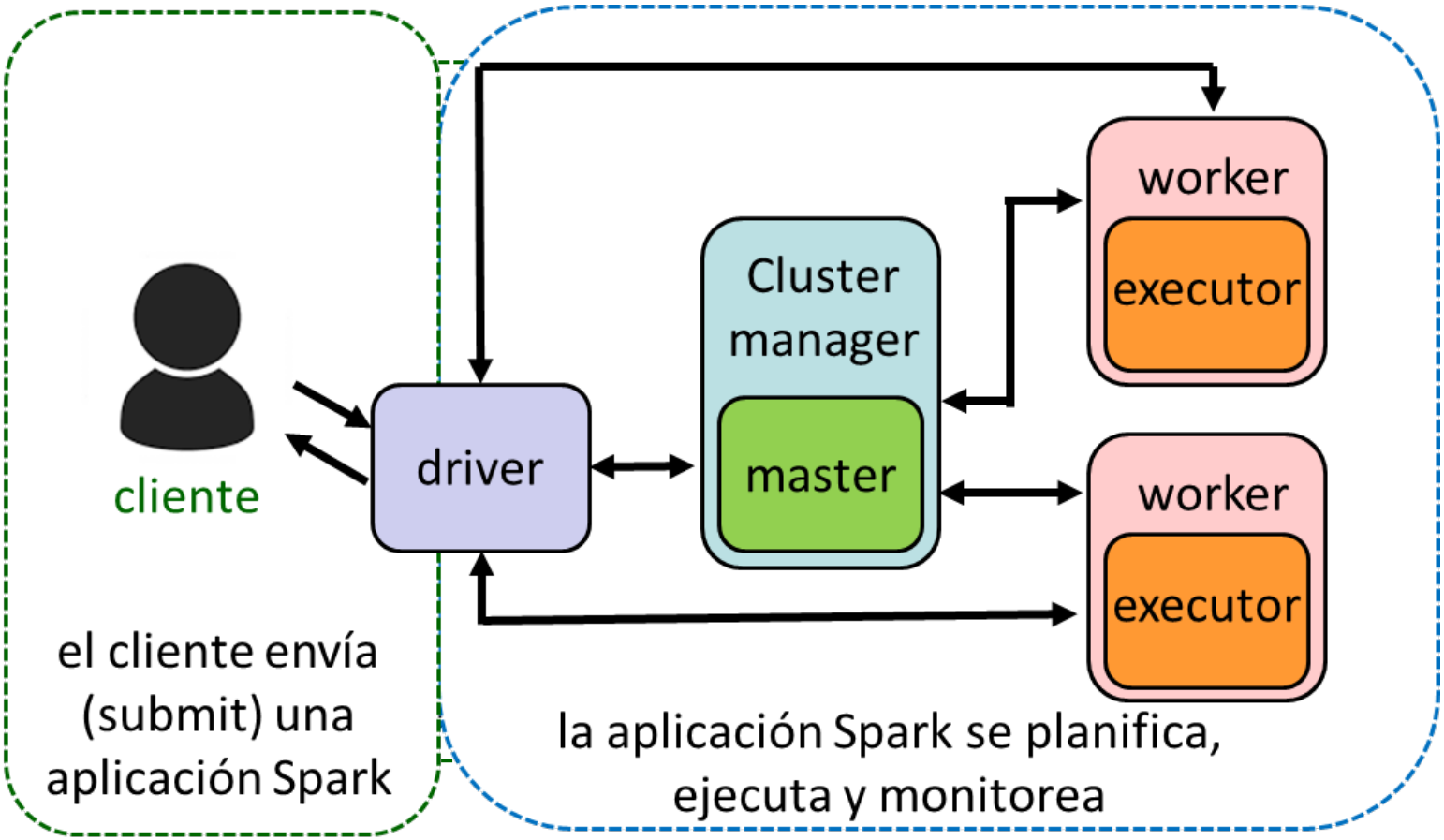
Se deben configurar propiedades del cluster (colas, planificación, etc.) para asegurar que puedan ejecutar apropiadamente todas las aplicaciones necesarias.

# Apache Spark: arquitectura y componentes

- Los componentes de una aplicación Spark son el *driver*, el *master*, el *cluster manager* y los *executors*, que ejecutan en recursos de cómputo llamados *worker nodes* o *workers*.
- Todos los componentes ejecutan en máquinas virtuales de Java (JVM), plataforma virtual que puede ejecutar instrucciones compiladas en bytecode.
- Es importante no confundir los componentes (lógicos) de una aplicación con las ubicaciones y recursos (físicos) donde ejecutan, que son diferentes para los distintos modos de desplegar Spark (standalone, YARN, Mesos, cloud).

# Apache Spark: arquitectura y componentes

- Spark Standalone



# Apache Spark: arquitectura y componentes

- Spark driver: proceso al cual los clientes envían trabajos.
  - Planifica y coordina la ejecución del programa Spark y retorna el status y resultados al cliente.
  - El driver crea el contexto (SparkContext, sc), instancia de la aplicación que representa la conexión con el master y los workers).
  - Se instancia al inicio de la aplicación y se usa durante todo el programa.
- Planificación de la aplicación y scheduling:
  - Considera las transformaciones y acciones sobre los datos y crea un DAG para planificar la ejecución de la aplicación.
  - Monitorea los recursos disponibles para la ejecución de tareas, intenta mantener la localidad de datos.
  - Coordina la ubicación y transferencia de datos entre procesos.
- Ofrece la *Application UI* en el Puerto 4040 (y sucesivos).

# Apache Spark: arquitectura y componentes

- Spark executors y workers: ejecutan las tareas de un DAG Spark.
  - Executors reservan recursos (CPU y memoria) en workers.
  - Están dedicados a una aplicación, finalizan cuando la aplicación termina.
  - Pueden ejecutar cientos o miles de tareas Spark.
  - Los workers tienen recursos limitados: número máximo de executors en un cluster (ejecutan en JVMs, con un heap asociado, su tamaño se puede especificar con spark.executor.memory o con el comando --executor-memory de pyspark, spark-shell, o spark-submit).
  - Los executors almacenan la salida de las tareas en memoria o en disco.
  - Executors y workers solo manejan las tareas que tienen asignadas (solo el driver ve el DAG por completo).
  - Los workers exponen una UI en el Puerto 8081.

# Apache Spark: arquitectura y componentes

- Spark Master y Cluster Manager: reservan, alocan y monitorean los recursos en el cluster (containers cuando se usa YARN/Mesos) donde ejecutan los executors.
- Pueden ser procesos separados o combinados (como en modo Standalone)
- Spark master: solicita recursos para el driver, negocia recursos o containers en los workers y monitorea su estado.
  - Expone una UI web en el Puerto 8080.
  - No confundir el 'master' con un proceso controlador de la ejecución de una aplicación (esta tarea la realiza el driver). El master solo controla los recursos.
- Cluster Manager: reserva y monitorea los workers requeridos por el master.
  - Al ejecutar sobre YARN el cluster manager es el ResourceManager de YARN.
  - El driver envía una aplicación al ResourceManager, que designa un ApplicationsMaster para la aplicación Spark.

# Apache Spark: arquitectura y componentes

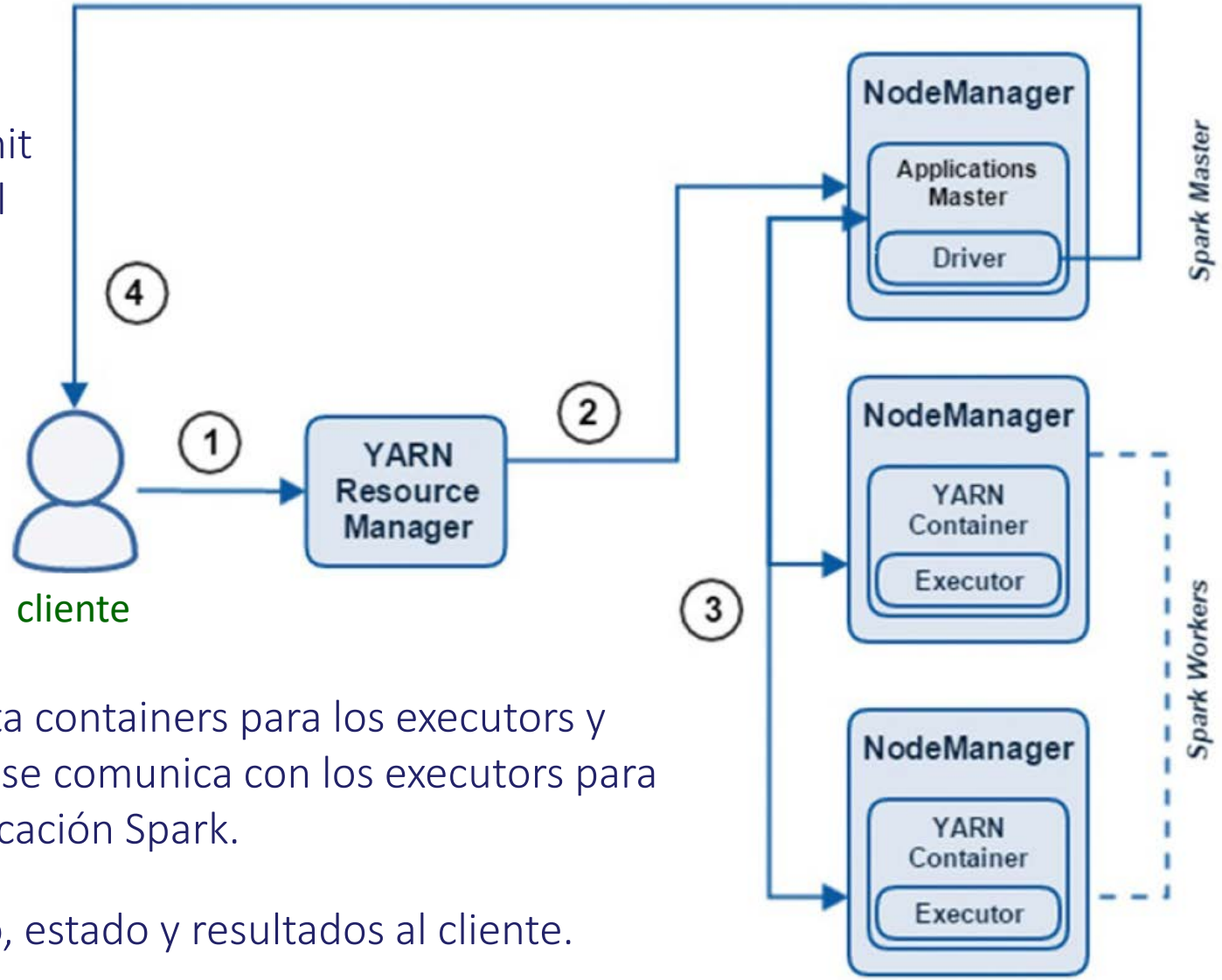
## Spark YARN-cluster

1. El cliente invoca spark-submit para enviar una aplicación al cluster manager (YARN ResourceManager).

2. El ResourceManager asigna un ApplicationsMaster (Spark master) para la aplicación. El driver se crea en el mismo nodo.

3. El ApplicationsMaster solicita containers para los executors y lanza su ejecución. El driver se comunica con los executors para ejecutar las tareas de la aplicación Spark.

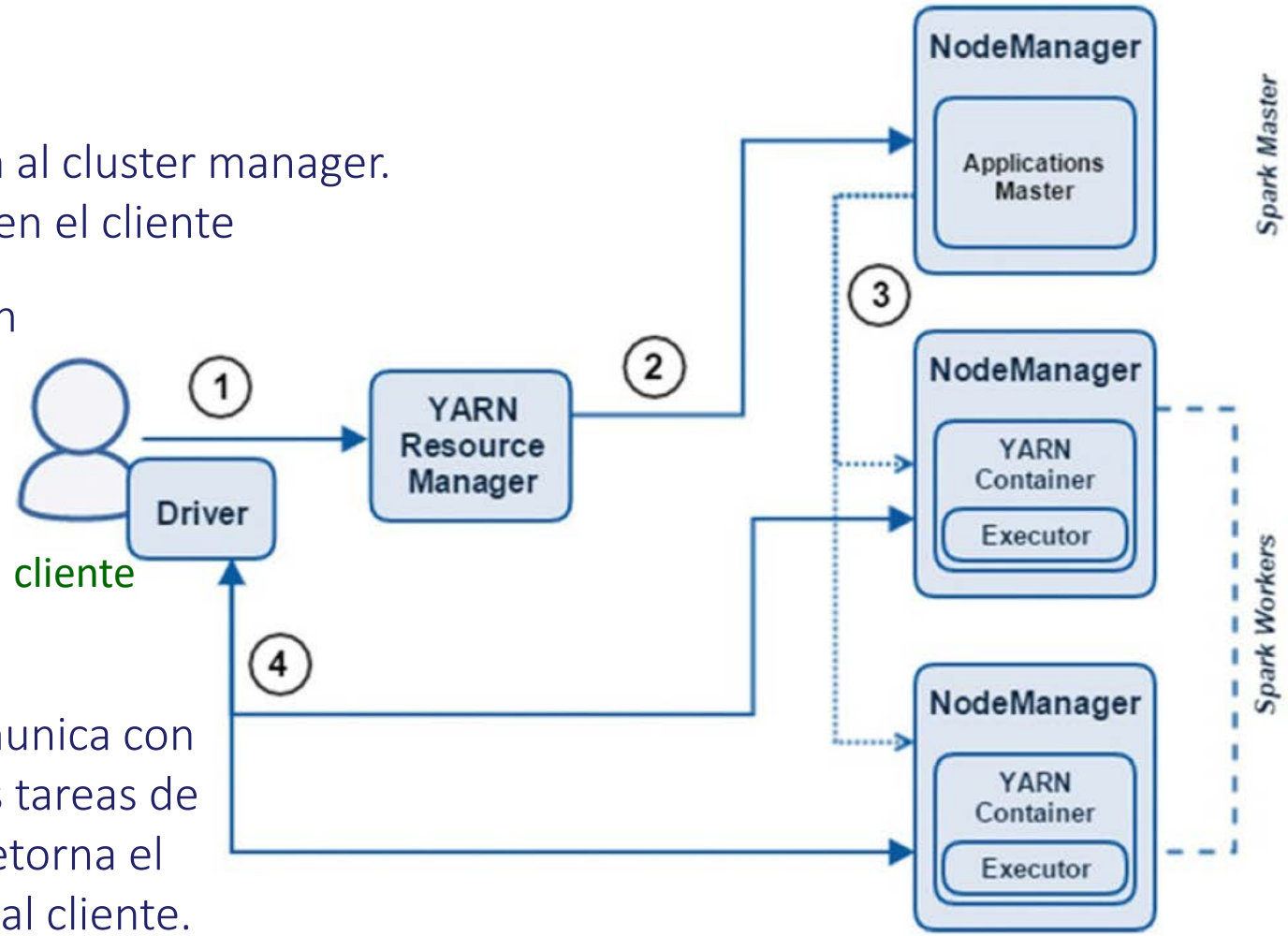
4. El driver retorna el progreso, estado y resultados al cliente.



# Apache Spark: arquitectura y componentes

## Spark YARN-client

1. El cliente envía una aplicación al cluster manager. Se crea el driver, que ejecuta en el cliente
2. El ResourceManager asigna un ApplicationsMaster (Spark master) para la aplicación.
3. El ApplicationsMaster solicita containers para los executors y lanza su ejecución.
4. El driver (en el cliente) se comunica con los executors para ejecutar las tareas de la aplicación Spark. El driver retorna el progreso, estado y resultados al cliente.




El cliente que ejecuta el driver debe estar disponible durante toda la ejecución de la aplicación.

# Apache Spark: uso no interactivo

- Las aplicaciones se envían (submit) utilzando el comando spark-submit.

```
$SPARK_HOME /bin/spark-submit \
  --class <main-class> --master <master-url> \
  --deploy-mode <deploy-mode> --conf <key>=<value> \
  ... # otras opciones
  <application-jar> [application-arguments]
```

  - class: punto de entrada a la aplicación
  - master: master URL del cluster (e.g. spark://23.195.26.187:7077)
  - deploy-mode: desplegar el driver en los worker nodes (modo cluster) o localmente (modo cliente, por defecto).
  - application-jar: jar con la aplicación y dependencias
  - application-arguments: parámetros de la aplicación

# Apache Spark: uso no interactivo

- Spark: uso no interactivo con spark-submit.
  - Ejecución local en 8 cores:

```
./bin/spark-submit --class org.apache.spark.examples.SparkPi \
  --master local[8] path/to/examples.jar 100
```

  - Ejecución standalone en cluster en modo cliente (100 cores, 20 GB RAM):

```
./bin/spark-submit --class org.apache.spark.examples.SparkPi \
  --master spark://207.184.161.138:7077 --executor-memory 20G \
  --total-executor-cores 100 /path/to/examples.jar 1000
```

  - Ejecución en cluster YARN

```
export HADOOP_CONF_DIR=XXX
./bin/spark-submit --class org.apache.spark.examples.SparkPi \
  --master yarn --deploy-mode cluster --executor-memory 20G \
  --num-executors 50 /path/to/examples.jar 1000
```

# Apache Spark: modo local

- Modo local: para desarrollo y verificación
- Driver, master y executor ejecutan en una única JVM.
- No distribuye el procesamiento y por tanto no escala.
- Ejemplo 1: estimar pi en modo local (código en slide 26).

```
spark-submit --class org.apache.spark.examples.SparkPi \
--master local $SPARK_HOME/lib/spark-examples*.jar 10
```

- Ejemplo 2: PySpark shell en modo local: `pyspark --master local`
- Ejemplo 3: modo local especificado en código

```
from pyspark import SparkContext
sc = SparkContext("local[*]")
[CÓDIGO]
```

- En modo local la UI de aplicación está disponible en http://localhost:4040 (master y worker UI no están disponibles).

# Spark y Hadoop

## HDFS como fuente de datos para Spark

- Spark incluye soporte para leer y escribir desde/hacia HDFS en varios formatos (Text, Sequence, Parquet y otros).
- Los procedimientos son muy simples:
  - Lectura: `textfile = sc.textFile("hdfs://mycluster/data/file.txt")`
  - Escritura: `myRDD.saveAsTextFile("hdfs://mycluster/data/output.txt")`

## YARN como planificador y manejador de recursos de Spark

- Cuando se usa HDFS, YARN planifica las tareas para tomar ventaja de la localidad de datos y minimizer el impacto de las transferencias.

# Apache Spark: programación

## Resilient Distributed Datasets (RDDs)

- Un RDD es una colección inmutable y distribuida de objetos.
- Cada RDD se divide en particiones que se procesan en diferentes nodos del cluster Spark (‘distributed’).
- Los RDD pueden incluir cualquier tipo de objetos de Python, Java, o Scala, incluyendo clases definidas por el Usuario.
- Los RDDs se crean de dos maneras: cargando un conjunto de datos externo o distribuyendo una colección de objetos (en el driver).
  - Ejemplo: Crear un RDD de strings en Python

```
lines = sc.textFile(“File")
```

- Una vez creados, los RDDs permiten dos tipos de operaciones: transformaciones y acciones.

# Apache Spark: ejemplo

- Ejemplo: Cálculo de pi en Spark, programado en Python

```
# Estimar π (tarea de cómputo intensivo).
# Algoritmo (método Monte Carlo):
#     Seleccionar aleatoriamente puntos en el cuadrado unitario (0,0) a (1,1)
#     Determinar cuántos puntos caen en el círculo unitario.
#     La fracción es un estimador del valor de π / 4
from random import random

def sample(p):
    x, y = random(), random()
    return 1 if x*x + y*y < 1 else 0

NUM_SAMPLES = 10000
# El método “parallelize” crea un dataset (RDD) de Spark
count = sc.parallelize(xrange(0,NUM_SAMPLES)).map(sample) \
             .reduce(lambda a,b: a + b)
print “Estimador de pi in %s mode: %f" % (sc.master,(4.0 * count/ NUM_SAMPLES))
```

- `sc.master` retorna el tipo de master utilizado, en el caso del ejemplo previo ‘local’.

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

# Apache Spark: ejemplo

- Ejemplo: Cálculo de pi en Spark, programado en Python

```
import sys
from random import random
from operator import add

if __name__ == "__main__":
    """
        Usage: pi [partitions]
    """“

from pyspark import SparkConf, SparkContext

conf = (SparkConf().setMaster("local").setAppName("estimacion de pi")
        .set("spark.executor.memory", "1g"))
sc = SparkContext(conf = conf)

partitions = int(sys.argv[1]) if len(sys.argv) > 1 else 2
n = 100000 * partitions

def f(_):
        x = random() * 2 - 1
        y = random() * 2 - 1
        return 1 if x ** 2 + y ** 2 <= 1 else 0

count = sc.parallelize(range(1, n + 1), partitions).map(f).reduce(add)
print "Pi es aproximadamente %f" % (4.0 * count / n)
```

# Apache Spark: ejemplo

- Ejemplo: word count
  - Cargar el archivo y aplicar flatMap() para parsear la línea leída.
  - map() para producir un RDD de pares (palabra, #ocurrencias).
  - reduceByKey() para sumar.
- En Python

```
inputrdd = sc.textFile("s3://..." o "hdfs://..." o "file://...")
words = inputrdd.flatMap(lambda x: x.split(" "))
result = words.map(lambda x: (x, 1)).reduceByKey(lambda x, y: x + y)
```

- En Scala

```
val inputrdd = sc.textFile("s3://...")
val words = inputrdd.flatMap(x => x.split(" "))
val result = words.map(x => (x, 1)).reduceByKey((x, y) => x + y)
```

- Implementación más simple utilizando función de conteo countByValue() en el primer RDD: input.flatMap(x =>x.split(" ")).countByValue().

# Apache Spark: RDDs

- Una transformación construye un nuevo RDD a partir de otro.
  - Ejemplo: filtrar datos (Python)

```
pythonLines = lines.filter(lambda line: "Python" in line)
```

- Una acción calcula un resultado aplicando una función sobre un RDD, y lo retorna al driver o lo almacena en un repositorio externo (e.g., HDFS).
  - Ejemplo, retornar el primer elemento de una lista (Python)

```
pythonLines.first()
```

- Las transformaciones se computan perezosamente (lazy evaluation): se calculan la primera vez que se utilizan en una acción.
  - Motivo: eficiencia. Si el RDD definido por `lines = sc.textFile(...)`, se calculara y almacenara se desperdiciaría tiempo y espacio.
- Spark analiza las transformaciones y solo calcula los datos que necesita. Por ejemplo, para calcular first() no es necesario leer todo el archivo !

# Apache Spark: operaciones

- Las operaciones sobre RDDs son de grano grueso (se aplican sobre todos los elementos del dataset).
  - Concepto similar al modelo MapReduce implementado en Hadoop.
- Transformaciones
  - Tienen como resultado la creación de un nuevo RDD.
  - Ejemplos comunes: map, filter, etc.

```python
originalrdd = sc.parallelize([0, 1, 2, 3, 4, 5, 6, 7, 8])
newrdd = originalrdd.filter(lambda x: x % 2)
```
*Python*

```python
inputRDD = sc.textFile("log.txt")
errorsRDD = inputRDD.filter(lambda x: "error" in x)
```
*Python*

```scala
val inputRDD = sc.textFile("log.txt")
val errorsRDD = inputRDD.filter(line => line.contains("error"))
```
*Scala*

```java
JavaRDD<String> inputRDD = sc.textFile("log.txt");
JavaRDD<String> errorsRDD = inputRDD.filter(
new Function<String, Boolean>() {
public Boolean call(String x) { return x.contains("error"); }});
```
*Java*

# Apache Spark: operaciones

## Transformaciones

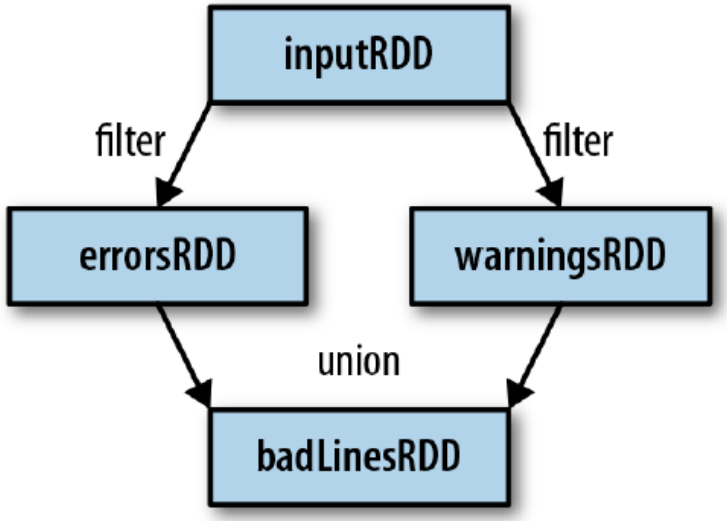
- Ejemplo: union() en Python

```
errorsRDD = inputRDD.filter(lambda x: "error" in x)
warningsRDD = inputRDD.filter(lambda x: "warning" in x)
badLinesRDD = errorsRDD.union(warningsRDD)
```

- union() opera sobre dos RDDs de entrada.
  - Una mejor opción sería filtrar directamente por error o warning.
- Spark toma en cuenta las dependencias (*lineage graph*) para calcular RDDs a demanda y para recuperar partes de un RDD en caso de fallo.
- Con la información genera un DAG de ejecución con las dependencias.

# Apache Spark: lineage

- Lineage: secuencia de operaciones para llegar a un RDD.
- Las dependencias pueden ser *narrow* o *wide*.




- Narrow: operaciones que pueden ser colapsadas en una fase única
  - Por ejemplo, map() y filter() en elementos del mismo dataset.
  - Solo un RDD hijo depende del RDD padre. No se require shuffle entre nodos. Son las operaciones más eficientes porque maximizan la ejecución paralela.
- Wide: operaciones en varias etapas, que requieren shuffling.
  - Por ejemplo join() y union(), que dependen de dos RDD de entrada.
  - Son inevitables para agrupar, unir o reducir datasets.

# Apache Spark: operaciones

## Acciones

- Retornan un resultado al driver o escriben en archivo/repositorio
  - Fuerzan la evaluación de las transformaciones sobre los RDD de entrada.

```
print “Hay " + str(badLinesRDD.count()) + " líneas sospechosas"
print “10 ejemplos:"
for line in badLinesRDD.take(10):
        print line
```

  - Se aplican las acciones count() para contar registros en un RDD y take() para acceder a un número dado de registros (para acceder a todos se utiliza collect(), debe tenerse cuidado que los datos entren en memoria).
  - Para almacenar en archivo/repositorio se utilizan saveAsTextFile(), saveAsSequenceFile() y acciones similares.
  - Para evitar ineficiencia al recalcular RDD complejos se pueden persistir en memoria (debe asegurarse que se dispone de la memoria suficiente).

# Apache Spark: RDDs

- Los RDDs son inmutables: una vez que se les asignaron datos, no pueden ser modificados.
- Por defecto, los RDD se recalculan cada vez que se aplica una acción sobre ellos. Si se desea reutilizar un RDD en multiples acciones es posible (y eficiente) persistirlo con RDD.persist().
- Luego de calcular un RDD por primera vez, Spark lo almacena en memoria, particionado en las máquinas del cluster.
- Es posible persistir un RDDs en disco lugar de en memoria (no se realiza por defecto por ser costoso para grandes datasets).
- La capapidad de recalcular un RDD le otorga la 'resiliencia' ('resilient').
  - Si un worker falla, los restantes workers del cluster pueden recalcular las particiones faltantes, de modo totalmente transparente al usuario.

# Apache Spark: persistencia

- Persistencia de RDDs
  - Los RDD se crean en la memoria de los executors. Son objetos transitorios que existen mientras son requeridos y luego se eliminan permanentemente.
  - persist() permite persistir los RDD en memoria para ser reutilizados.

```
originalrdd = sc.parallelize([0, 1, 2, 3, 4, 5, 6, 7, 8])
newrdd = originalrdd.filter(lambda x: x % 2)
noelements = newrdd.count()
# procesar newrdd
listofelements = newrdd.collect()
# reprocesar newrdd
print "Hay %s elementos in el RDD %s" %
(noelements, listofelements)
# retorna:  Hay 4 elementos en el RDD [1, 3, 5, 7]
```

```
originalrdd = sc.parallelize([0, 1, 2, 3, 4, 5, 6, 7, 8])
newrdd = originalrdd.filter(lambda x: x % 2)
newrdd.persist()
noelements = newrdd.count()
# procesar y persistir newrdd en memoria
listofelements = newrdd.collect()
# no se debe recalcular newrdd
print “Hay %s elementos in el RDD %s" %
(noelements, listofelements)
# retorna: Hay 4 elementos en el RDD [1, 3, 5, 7]
```

  - Luego de aplicar persist() o cache() el RDD queda en memoria de todos los nodos del cluster donde fue computado (luego de ejecutar la primera acción). Se puede visualizar en la UI del driver (tab ‘Storage’).

# Apache Spark: tipos de RDD

- PairRDD: RDD de pares clave-valor.
- DoubleRDD: colección de valores double, con funciones estadásticas asociadas (mean(), sum(), stdev, variance, histogram, etc.).
- DataFrame (antes SchemaRDD): datos distribuidos organizados en columnas con nombre.
- SequenceFileRDD: RDD creado desde un SequenceFile (con/sin compresión).
- HadoopRDD: RDD con funcionalidades para leer datos desde HDFS usando la API MapReduce de Hadoop v1.
- NewHadoopRDD: RDD con funcionalidades para leer datos desde HDFS con la nueva API MapReduce de Hadoop (org.apache.hadoop.mapreduce).
- CoGroupedRDD: RDD que agrupa (cogroup) pares clave-valor por clave.

# Apache Spark: tipos de RDD

- JdbcRDD: RDD que ejecuta consultas SQL via JDBC (solo en Scala).
- PartitionPruningRDD: RDD para eliminar particiones y no lanzar tareas superfluas.
  - Por ejemplo, si un RDD está particionado por rango y el DAG de la aplicación tiene un filtro, se puede evitar lanzar tareas sobre valores que no pertenecen al rango del filtro.
- ShuffledRDD: RDD resultado de un shuffle (reparticionado de datos).
- UnionRDD: RDD resultado de la operación de union de dos RDDs
- Otros: ParallelCollectionRDD, PythonRDD, etc.
- Son abstracciones (simples) de la clase base de RDDs.
- Los RDDs más comúnmente utilizados son PairRDD, DoubleRDD y DataFrame RDD.

# Apache Spark: programación

- Ejemplo: filtrar líneas de error en un log

```
# cargar archivos de log desde el filesystem local
logfilesrdd = sc.textFile("file:///var/log/hadoop/hdfs/hadoop-hdfs-*")
# filtrar registros de error
onlyerrorsrdd = logfilesrdd.filter(lambda line: "ERROR" in line)
# almacenar el RDD resultado en un archivo
onlyerrorsrdd.saveAsTextFile("file:///tmp/onlyerrorsrdd")
```

- El procesamiento se realiza en paralelo, puede tener un impacto significativo en el tiempo de ejecución al procesar grandes volúmenes de datos.
- La acción saveAsTextFile() crea un directorio conteniendo múltiples archivos de texto, como consecuencia de que el RDD está particionado y se procesa en paralelo.

# Apache Spark: crear y cargar datos en RDDs

- Crear RDDs: cargar datos desde una fuente externa o paralelizar una colección en el driver.
  - Paralelizar utilizando el método parallelize() de SparkContext se utiliza para desarrollo y verificación. No se usa en aplicaciones reales en producción porque requiere disponer de todos los datos en memoria del driver.

```
lines = sc.parallelize(["pandas", "i like pandas"]) (Python)
val lines = sc.parallelize(List("pandas", "i like pandas")) (Scala)
JavaRDD<String> lines = sc.parallelize(Arrays.asList("pandas",
"i like pandas")); (Java)
```

- Para cargar datos en RDD existen varias opciones:
  - Cargar datos desde archivo(s).
  - Cargar datos desde un repositorio de datos (SQL o NoSQL).
  - Cargar datos programáticamente.
  - Cargar datos desde un stream.

# Apache Spark: crear y cargar datos en RDDs

- Cargar datos desde archivo:
  - La API de Spark prove métodos para crear RDDs desde archivo(s)/directorio.
  - Formatos no estructurados, semiestructurados (JSON, etc.), estructurados (archivos csv, etc.), binarios serializados (SequenceFiles, etc.).
  - Provee soporte nativo para manejar archivos comprimidos (gzip, zip, etc.)
- Leer líneas desde archivo: SparkContext.textFile()
  - Cada línea se representa por un registro (sin referencia al archivo de origen).
- sc.textFile(name [, minPartitions=None, use_unicode=True])
  - Name: path o glob, incluyendo el esquema del filesystem (file://, hdfs://, s3://)
  - minPartitions: número de particiones a crear. Si se carga desde HDFS, se crea una partición por bloque (tamaño 64MB o 128MB).
  - use_Unicode: indica si codifica con Unicode o UTF-8.
  - Para leer de HDFS se debe setear la variable de entorno HADOOP_CONF_DIR en todos los nodos del cluster (export HADOOP_CONF_DIR=/etc/hadoop/conf)

# Apache Spark: crear y cargar datos en RDDs

- Ejemplos de SparkContext.textFile()

```
lines = sc.textFile("/path/to/README.md") (Python)
val lines = sc.textFile("/path/to/README.md") (Scala)
JavaRDD<String> lines = sc.textFile("/path/to/README.md"); (Java)
```

- Leer archivos de un directorio: wholeTextFiles()
- sc.wholeTextFiles(path [, minPartitions=None, use_unicode=True])
  - Cada archivo se representa por un registro cuya clave es el nombre del archivo y el valor es su contenido. Muy útil en procesamiento de eventos.
- Al usar wholeTextFiles() cada registro contiene todo el archivo, solo debe utilizarse para archivos pequeños !
  - path: path al directorio, incluyendo el esquema del filesystem.
  - minPartitions y use_Unicode como en sc.textFile().
  - Ejemplo: leer archivos de /opt/spark/licenses y analizar número, particiones, claves, etc.

# Apache Spark: crear y cargar datos en RDDs

- Crear RDDs desde archivos serializados
  - sequenceFile() y hadoopFile(): crean RDDs desde sequence files de Hadoop.
  - objectFile(): crea RDDs desde objetos de Java serializados
  - Archivos Pickle (formato de serialización de Python) y archivos JSON, que se tratan como repositorios externos.
- Crear RDDs desde repositorios externos (datasources): SQL y NoSQL.
  - Los datos se dividen en particiones en diferentes workers.
  - Se usa un context especial para manejar datos estructurados: SQLContext.
  - Permite crear DataFrames (prev. SchemaRDDs) y ejecutar consultas SQL.
- Crear RDDs desde un datasource Java Database Connectivity (JDBC)
  - JDBC: API para acceder a DBMS a través de conectores.
  - Manejan conexiones/desconexiones, consultas etc.
  - Spark ejecuta en JVMs, tiene soporte nativo para JDBC.

# Apache Spark: crear y cargar datos en RDDs

- Crear RDDs desde una base de datos MySQL via JDBC (versión < 1.4)

```
pyspark --driver-class-path share/java/mysql-connector-java-5.x-bin.jar --master local
```

  - Crear un SQLContext

```
from pyspark.sql import SQLContext
sqlctx = SQLContext(sc)
```

  - Cargar datos: sqlContext.load(path, source, schema, **options) se usa para varios tipos de datos, incluyendo bases de datos. Para JDBC las opciones son un diccionario de Python (pares nombre-valor con valores de configuración).
  - Ejemplo: cargar datos de empleados

```
employeesdf = sqlctx.load(source="jdbc",url="jdbc:mysql://localhost:3306/employees?user=<u>&password=<p>",dbtable="employees", partitionColumn="emp_no", numPartitions="2", lowerBound="101", upperBound="499")
employeesdf.rdd.getNumPartitions()
```

  - Debe retornar 2 (el número de particiones especificadas)

# Apache Spark: crear y cargar datos en RDDs

- Crear RDDs desde una base de datos via JDBC (Spark 1.4 y superior)
- sqlContext.read.jdbc(url, table, column, lowerBound, upperBound, numPartitions, predicates, properties)
  - Crea un DataFrame, que se puede consultar con SQL.
  - Los parámetros permiten filtrar registros no necesarios

```
employeesdf = sqlctx.read.jdbc(url="jdbc:mysql://localhost:3306/
employees", table="employees", column="emp_no", numPartitions="2",
lowerBound="10000", upperBound="49999",
properties={"user":"<user>","password":"<pwd>"})

sqlctx.registerDataFrameAsTable(employeesdf,"employees")
df2 = sqlctx.sql("SELECT emp_no, first_name, last_name FROM
employees LIMIT 2")
df2.show()
#|emp_no|first_name|last_name|
#+------+----------+---------+
#| 10001| Georgi   | Facello |
#| 10002| Bezalel  | Simmel  |
```

# Apache Spark: crear y cargar datos en RDDs

- Dataframes: abstracción de los RDDs
  - Colección distribuida de registros, todos con el mismo esquema.
  - Análogo a una tabla distribuida de una base de datos relacional.
  - Los dataframes tienen como atributo su esquema y proven soporte nativo para funciones SQL.
  - Son RDDS: se evalúan como DAGs, de forma lazy, tienen lineage, proveen tolerancia a fallos y soportan persistencia.
  - Pueden crearse desde un RDD existente, desde archivos de texto y JSON, desde tablas en Hive, desde bases de datos relacionales externas, etc.
  - La API de Dataframe es expuesta por el objeto SQLContext.

# Apache Spark: crear y cargar datos en RDDs

- Crear RDDs desde Apache Hive
  - Configuración de Hive: copiar hive-site.xml al directorio ./conf de Spark
  - Crear un objeto HiveContext, punto de entrada para especificar consultas en Hive Query Language (HQL) en las tablas definidas.
  - Los datos se retornan como RDDs de filas.
  - Ejemplo:

```
hive> CREATE TABLE users (name VARCHAR(64), age INT, gpa DECIMAL(3, 2));
OK. Time taken: 0.096 seconds
hive> INSERT INTO TABLE users VALUES ('fred flintstone', 35, 1.28),
('barney rubble', 32, 2.32);
```

```
from pyspark.sql import HiveContext
hiveCtx = HiveContext(sc)
rows = hiveCtx.sql("SELECT name, age FROM users")
firstRow = rows.first()
print firstRow.name
'fred flintstone'
```

# Apache Spark: crear y cargar datos en RDDs

- Ejemplo: consultar datos de estaciones de San Jose

```
from pyspark import SparkConf, SparkContext
from pyspark.sql import SQLContext, HiveContext
sc = SparkContext()
hiveContext = HiveContext(sc)
sql_cmd = """SELECT name, lat, long FROM stations
                WHERE landmark = 'San Jose'"""
sqlContext.sql(sql_cmd).show()
```

```
+--------------------+---------+-----------+
| name               | lat     | long      |
+--------------------+---------+-----------+
|Adobe on Almaden    |37.331415| -121.8932 |
|San Pedro Square    |37.336721|-121.894074|
|Paseo de San Antonio|37.333798|-121.886943|
|San Salvador at 1st |37.330165|-121.885831|
|Japantown           |37.348742|-121.894715|
|San Jose City Hall  |37.337391|-121.886995|
|...                 | ...     | ...       |
+--------------------+---------+-----------+
```

# Apache Spark: crear y cargar datos en RDDs

- Crear RDDs desde archivos JSON (JavaScript Object Notation)
  - Autodescriptivos, legibles por humanos, se usan como respuesta de web services y APIs REST.
  - Objetos JSON se tratan como repositorios externos usando SQLContext.
- sqlContext.jsonFile(path, schema=None) y sqlContext.read.json(path, schema=None) en Spark 1.4 y posteriores
  - Ejemplo: crear un RDD desde un JSON con nombres y opcionalmente edad

```
from pyspark.sql import SQLContext
sqlctx = SQLContext(sc)
people = sqlctx.jsonFile("file://opt/spark/examples/src/main/resources/
                          people.json") # o desde hdfs, previo carga
people
# DataFrame[age: bigint, name: string]
```

  - Crea un DataFrame con un esquema

# Apache Spark: crear y cargar datos en RDDs

- Crear RDDs desde archivos JSON (JavaScript Object Notation)

```
people.dtypes retorna los nombres de columna y sus tipos
#[('age', 'bigint'), ('name', 'string')]
people.show()
#| age| name  |
#+----+-------+
#|null|Michael|
#| 30 | Andy  |
#| 12 | Justin|
```

  - El DataFrame se puede consultar via SQL

```
sqlctx.registerDataFrameAsTable(people, "people")
df2 = sqlctx.sql("SELECT name, age FROM people WHERE age > 20")
df2.show()
#|name|age|
#+----+---+
#|Andy| 30|
```

# Apache Spark: crear y cargar datos en RDDs

## Crear RDDs programáticamente

- A partir de datos en los programas (listas, arrays, colecciones) que se particionan y distribuyen.
- Requiere que todos los datos existan en memoria.
- sc.parallelize(c, numPart=None)
  - Paraleliza una lista o colección existente c en numPart particiones. Crea un objeto de tipo ParallelCollectionRDD.

```
parallelrdd = sc.parallelize([0, 1, 2, 3, 4, 5, 6, 7, 8])
parallelrdd
# ParallelCollectionRDD[0] at parallelize at PythonRDD.scala:423
```

`parallelrdd.min()` retorna 0

`parallelrdd.max()` retorna 8

`parallelrdd.collect()` reúne la colección paralela en una lista

```
[0, 1, 2, 3, 4, 5, 6, 7, 8]
```

# Apache Spark: crear y cargar datos en RDDs

## Crear RDDs programáticamente

- sc.range(start, end=None, step=1, numPart=None): genera un RDD con una secuencia de valores entre start y end con paso slice y la distribuye en numPart particiones

  `rangerdd = sc.range(0, 1000, 1, 2)` crea RDD con 1000 enteros, empezando en 0 con incremento de 1 y lo divide en dos particiones

  – En python el tipo es PythonRDD

  ```
  rangerdd
  # PythonRDD[1] at RDD at PythonRDD.scala:43
  ```

  `rangerdd.getNumPartitions()`, retorna 2

  `rangerdd.min()`, retorna 0

  `rangerdd.max()`, retorna 999

  `rangerdd.take(5)` retorna los 5 primeros elementos

  ```
  [0, 1, 2, 3, 4]
  ```

# Apache Spark: cómo pasar funciones

- Las transformaciones y algunas acciones requieren funciones para calcular los datos.
- Python:
  - Expresiones lambda (funciones *cortas* o *sin nombre*)

```
word = rdd.filter(lambda s: "error" in s)
```

  - Funciones de alto nivel o funciones definidas localmente (funciones *nominadas* o *con nombre*)

```
def containsError(s): return "error" in s
word = rdd.filter(containsError)
```

  - Cuidado: no debe serializarse el objeto que contiene la función !
  - Si se pasa una función que es miembro de un objeto o contiene referencias a campos (e.g., self.field), Spark envía el objeto entero a los workers (es poco eficiente y puede causar error si la clase contiene objetos que Python no maneja apropiadamente).

# Apache Spark: cómo pasar funciones

- Pasar una función con referencias (no hacerlo!)

```
class SearchFunctions(object):
def __init__(self, query):
self.query = query
def isMatch(self, s):
return self.query in s
def getMatchesFunctionReference(self, rdd):
return rdd.filter(self.isMatch)
def getMatchesMemberReference(self, rdd):
return rdd.filter(lambda x: self.query in x)
```

  - Problema: se referencia todo "self" en "self.isMatch“ y en "self.query"
  - Solución: extraer los campos necesarios del objeto en una variable local y pasar la variable.

```
class WordFunctions(object):
...
def getMatchesNoReference(self, rdd):
query = self.query
return rdd.filter(lambda x: query in x)
```

# Apache Spark: cómo pasar funciones

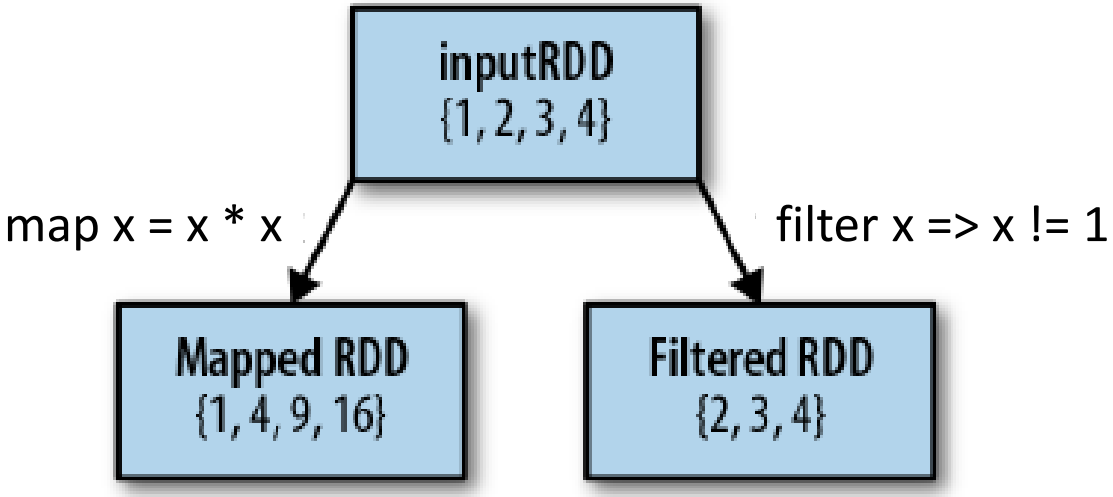
- En Scala
  - Funciones inline, referencias a métodos y funciones estáticas.
  - La función y los datos referenciados deben ser serializables.

```
class SearchFunctions(val query: String) {
def isMatch(s: String): Boolean = {s.contains(query)}
def getMatchesFunctionReference(rdd: RDD[String]): RDD[String] = {
    // Problema: "isMatch" es "this.isMatch", se pasa todo "this"
    rdd.map(isMatch)
}
def getMatchesFieldReference(rdd: RDD[String]): RDD[String] = {
    // Problem: "query" es "this.query", se pasa todo "this"
    rdd.map(x => x.split(query))
}
def getMatchesNoReference(rdd: RDD[String]): RDD[String] = {
    // Seguro: extraer el campo necesario en una variable local
    val query_ = this.query
    rdd.map(x => x.split(query_))
}}
```

  - Cuidado con referencias a métodos o campos no serializables.

# Apache Spark: operaciones

- Ejemplos de transformaciones: map() y filter()
  - map() aplica una función a cada elemento en un RDD.
  - Múltiples usos: aritméticos, recuperar un sitio web asociado a una URL, etc.
  - El tipo del resultado no tiene por qué ser el mismo tipo de entrada (ejemplo: entrada RDD[String] y resultado RDD[Double]).
  - filter() filtra los elementos que cumplen con (pasan) el filtro



# Apache Spark: operaciones

- Ejemplos de transformaciones: map()
  - En Python

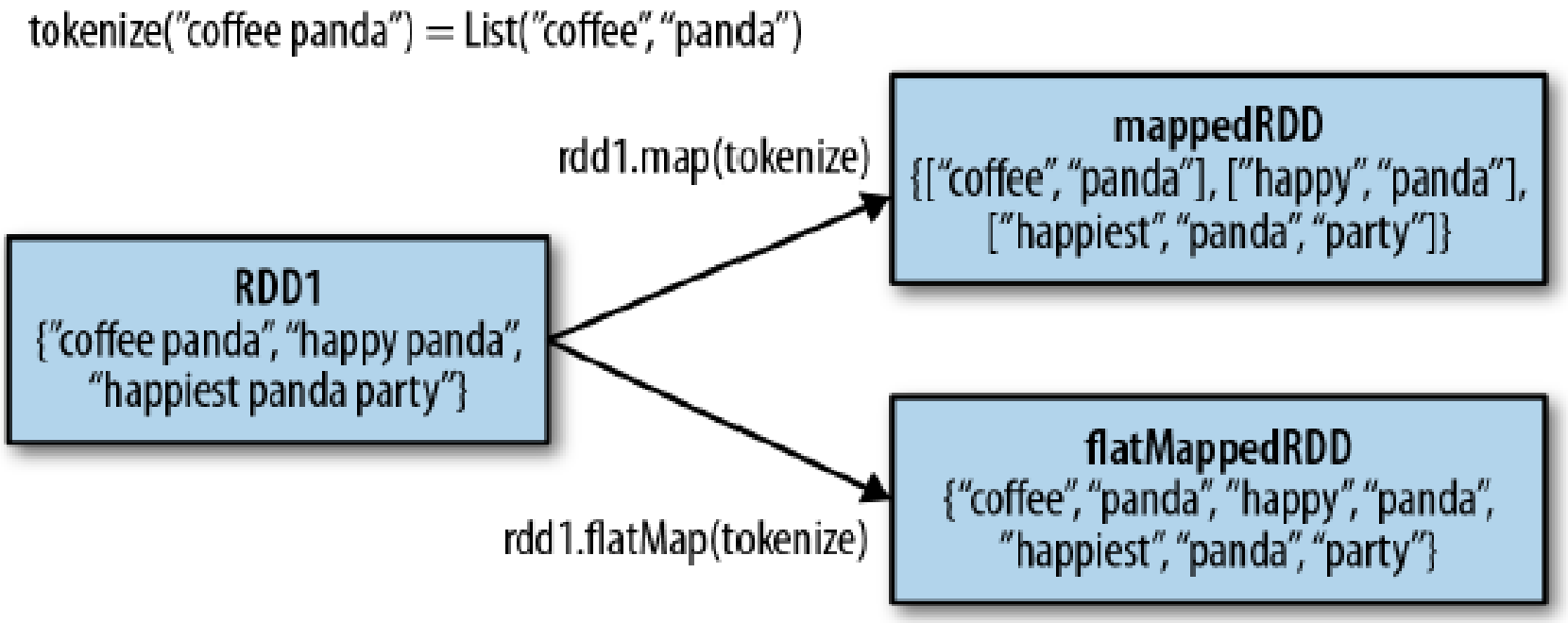
```
nums = sc.parallelize([1, 2, 3, 4])
squared = nums.map(lambda x: x * x).collect()
for num in squared:
        print "%i " % (num)
```

  - En Scala

```
val input = sc.parallelize(List(1, 2, 3, 4))
val result = input.map(x => x * x)
println(result.collect().mkString(","))
```

# Apache Spark: operaciones

- Ejemplos de transformaciones: flatmap()
  - Permite producir múltiples elementos para cada elemento de entrada.
  - flatmap() aplica una función a cada elemento de la entrada pero en lugar de retornar un elemento retorna un iterador con los valores de salida.
  - El RDD resultante contiene los elementos de todos los iteradores.



# Apache Spark: operaciones

- Ejemplos de flatmap() en Python para dividir líneas en palabras.

```
>>> lines = sc.parallelize(["hello world", "hi"])
>>> words = lines.flatMap(lambda line: line.split(" "))
>>> words.first()
'hello'
>>> words2 = lines.map(lambda line: line.split(" "))
>>> words.collect()
['hello', 'world', 'hi']
>>> words2.collect()
[['hello', 'world'], ['hi']]
```

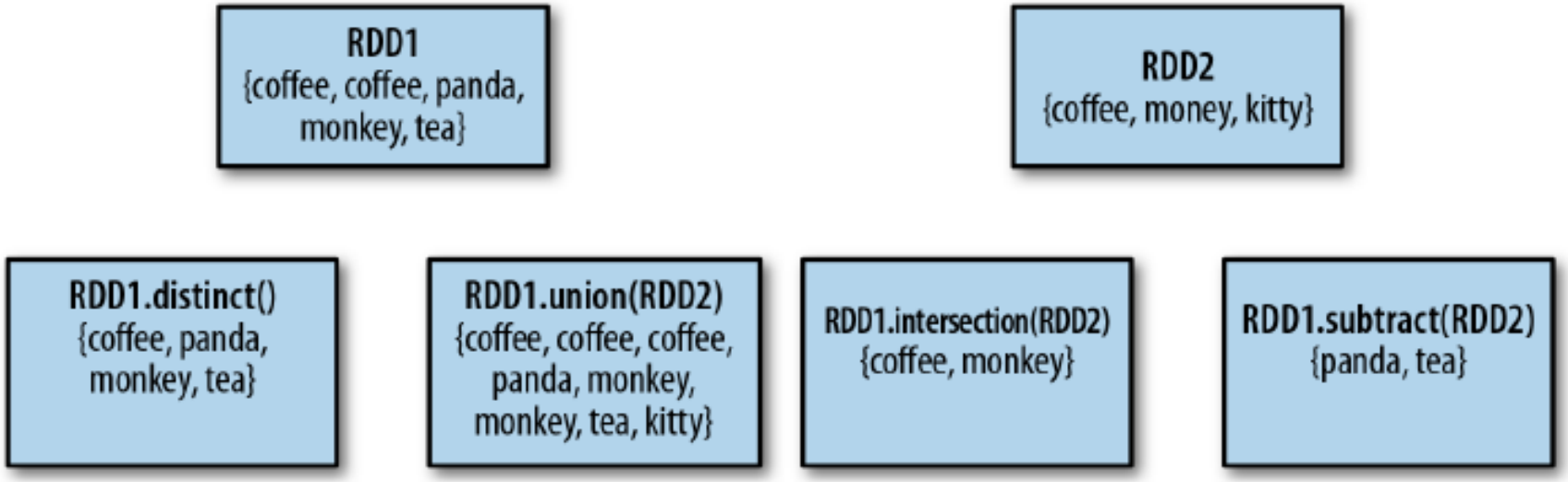
# Apache Spark: operaciones

- Ejemplos (sobre el RDD {1, 2, 3, 3})

| Función | Propósito | Ejemplo | Resultado |
|---|---|---|---|
| `map()` | Aplica una función a cada elemento en el RDD | `rdd.map(x => x + 1)` | `{2, 3, 4, 4}` |
| `flatMap()` | Aplica una función a cada elemento en el RDD y retorna el contenido de los iteradores | `rdd.flatMap(x => x.to(3))` | `{1, 2, 3, 2, 3, 3, 3}` |
| `filter()` | Retorna el RDD con los elementos que pasan el filtro | `rdd.filter(x => x != 1)` | `{2, 3, 3}` |
| `distinct()` | Elimina duplicados | `rdd.distinct()` | `{1, 2, 3}` |
| `sample(withReplacement, fraction, [seed])` | Muestrea un RDD, con o sin duplicados | `rdd.sample(false, 0.5)` | No determinista |

# Apache Spark: operaciones

- Operaciones sobre 'seudoconjuntos'
  - RDDs no son conjuntos porque pueden tener elementos duplicados.
  - Para trabajar con conjuntos se puede utilizar la transformación RDD.distinct()
  - RDD.distinct() es una operación muy cara, ya que require una fase de shuffling (sobre la red) para recibir solamente una copia de cada elemento.
- Ejemplos
  - Las operaciones requieren que los RDDs sean del mismo tipo.



# Apache Spark: operaciones

- Ejemplos (sobre los RDD {1, 2, 3} y {3, 4, 5})

| Función | Propósito | Ejemplo | Resultado |
|---|---|---|---|
| `union()` | Unión (elementos en alguno de los RDDs) | `rdd.union(other)` | `{1, 2, 3, 3, 4, 5}` |
| `intersection()` | Intersección (elementos en ambos RDDs) | `rdd.intersection(other)` | `{3}` |
| `subtract()` | Elimina el contenido de un RDD (ejemplo, eliminar datos de entrenamiento) | `rdd.subtract(other)` | `{1, 2}` |
| `cartesian()` | Producto cartesiano con otro RDD | `rdd.cartesian(other)` | `{(1, 3), (1, 4), … (3,5)}` |

# Apache Spark: acciones

- RDD.count(): retorna el número de elementos (o registros) en un RDD.
  - Python:

```
lines = sc.textFile("README.md")
lines.count()
127
```

  - Scala:

```
scala> val lines = sc.textFile("README.md")
lines: spark.RDD[String] = MappedRDD[...]
scala> lines.count()
res0: Long = 127
```

- RDD.collect(): retorna (al driver) una lista con el contenido del RDD
  - No restringe la salida, puede ocasionar error de memoria en el driver.

```
lorem = sc.textFile('file:///opt/spark/data/lorem.txt')
words = lorem.flatMap(lambda x: x.split())
words.collect()
# [u'lorem', u'ipsum', u'dolor', u'sit', u'amet', u'consectetur’,
# u'adipiscing', u'elit', u'nullam']
```

# Apache Spark: acciones

- RDD.take(n): retorna una lista con n elementos del RDD, es no determinista.
  - La función RDD.takeOrdered retorna los primeron n elementos considerando el orden indicado por una función.

```
lorem = sc.textFile('file:///opt/spark/data/lorem.txt’)
words = lorem.flatMap(lambda x: x.split())
words.take(3)
# [u'lorem', u'ipsum', u'dolor']
```

- RDD.first(): retorna el primer elemento en el RDD, no considera orden, es no determinista.

```
lorem = sc.textFile('file:///opt/spark/data/lorem.txt')
words = lorem.flatMap(lambda x: x.split())
words.first()
# u'lorem'
```

# Apache Spark: acciones

- Una de las acciones más comunes (y relevantes) es reduce()
  - Recibe una función que opera sobre dos elementos del tipo del RDD y retorna un nuevo elemento del mismo tipo.
  - Como ejemplo, la función suma:
  - Python: `sum = rdd.reduce(lambda x, y: x + y)`
  - Scala: `val sum = rdd.reduce((x, y) => x + y)`
  - Java:

```
Integer sum = rdd.reduce(new Function2<Integer,Integer,Integer>() {
public Integer call(Integer x, Integer y) { return x + y; }
});
```

- fold() incluye un 'valor inicial' que se usa para la primer invocación en cada partición (elemento identidad para la acción)
  - E.g. 0 para la suma, 1 para el producto, lista vacía para concatenación, etc.

# Apache Spark: acciones

- fold() incluye un 'valor inicial' que se usa para la primer invocación en cada partición (elemento identidad para la acción)

```
>>> from operator import add
>>> sc.parallelize([1, 2, 3, 4, 5]).fold(0, add)
15
```

  - El resultado de fold es afectado por el número de particiones: primero hace fold de cada partición y luego fold de los resultados.
  - `sc.parallelize([1,25,8,4,2]).fold(0,lambda a,b:a+1)` no cuenta el número de elementos en el RDD: una partición vacía cuenta en el fold como un elemento !

```
>>> sc.parallelize([1,25,8,4,2], 50).fold(0,lambda a,b:a+1)
50
>>> sc.parallelize([1,25,8,4,2], 1).fold(0,lambda a,b:a+1)
1
```

  - La partición se combina al valor correcto, pero luego se combina con el 'valor cero' en el driver y retorna 1.

# Apache Spark: acciones

- reduce() y fold() requieren que los tipos de entrada y salida sean idénticos.
- Cómo calcular funciones con tipos diferentes? (por ejemplo, promedio que retorne un par (valor promedio,#elementos)).
- Usar map() para transformar cada elemento de entrada en (elemento,1) y trabajar con reduce() sobre los pares.

```
>>> a = sc.parallelize([1,25,8,4,2]).map(lambda x: (x,1)).\
    reduce(lambda x,y: (x[0]+y[0],x[1]+y[1]))
>>> (a[0]/a[1],a[1])
(8, 5)
```

# Apache Spark: acciones

- Cómo calcular funciones con tipos diferentes? (por ejemplo, moving average, que retorna un par (valor,#elementos)).
- aggregate() permite retornar resultados de diferente tipo.
  - Usa un valor inicial como fold(), una función para combinar elementos del RDD con el acumulador y una segunda función para combinar dos acumuladores.
  - Ejemplo: media de RDD sin usar map()

Python

```
sumCount = nums.aggregate(
(0, 0),
(lambda acc, value: (acc[0] + value, acc[1] + 1),
(lambda acc1, acc2: (acc1[0] + acc2[0], acc1[1] + acc2[1])))
avg = sumCount[0] / float(sumCount[1])
```

Scala

```
val result = input.aggregate((0, 0))(
(acc, value) => (acc._1 + value, acc._2 + 1),
(acc1, acc2) => (acc1._1 + acc2._1, acc1._2 + acc2._2))
val avg = result._1 / result._2.toDouble
```

# Apache Spark: otras acciones comunes

| Función | Propósito | Ejemplo | Resultado |
|---|---|---|---|
| `collect()` | Retorna todos los elementos del RDD | `rdd.collect()` | `{1, 2, 3, 3}` |
| `count()` | Número de elementos en el RDD | `rdd.count()` | `4` |
| `countByValue()` | Número de veces de cada elemento en el RDD | `rdd.countByValue()` | `{(1, 1), (2, 1), (3, 2)}` |
| `take(num)` | Retorna num elementos del RDD | `rdd.take(2)` | `{1, 2}` |
| `top(num)` | Retorna los mayores elementos del RDD | `rdd.top(2)` | `{3, 3}` |
| `takeOrdered(num)(ordering)` | Retorna num elementos de acuerdo al orden | `rdd.takeOrdered(2)(myOrdering)` | `{3, 3}` |

# Apache Spark: otras acciones comunes

| Función | Propósito | Ejemplo | Resultado |
|---|---|---|---|
| `takeSample(withReplacement, num, [seed])` | Retorna num elementos al azar | `rdd.takeSample(false, 1)` | No determinista |
| `reduce(func)` | Combina elementos del RDD en paralelo | `rdd.reduce((x, y) => x + y)` | 9 |
| `fold(zero)(func` | Ídem reduce() pero con valor inicial | `rdd.fold(0)((x, y) => x + y)` | 9 |
| `aggregate(zeroV(seqOp, combOp)` | Ídem reduce() pero con diferentes tipos | `rdd.aggregate((0, 0))`<br>`((x, y) =>`<br>`(x._1 + y, x._2 + 1),`<br>`(x, y) =>`<br>`(x._1 + y._1, x._2 + y._2))` | (9, 4) |
| `foreach(func)` | Aplica func a cada elemento del RDD | `rdd.foreach(func)` | Sin resultado |