RDD: Resilient Distributed Datasets

- Colección inmutable y distribuida de elementos que pueden manipularse en paralelo
  - El tipo de datos más básico en Spark
- Al igual que con los DataFrames, con los RDDs podemos:
  - Crearlos
  - Transformarlos en otros RDDs (o DataFrames)
  - Realizar acciones para extraer información
- Proporcionan un control más fino sobre el particionado y la distribución de datos

En Scala y Java las operaciones que se pueden hacer sobre los RDDs y los DataSets son muy parecidas

- API Python para RDDs: https://spark.apache.org/docs/latest/api/python/reference/api/pyspark.RDD.html
- API Scala para RDDs: https://spark.apache.org/docs/latest/api/scala/org/apache/spark/rdd/index.html

In [ ]:
```
from pyspark import SparkContext
from pyspark.sql import SparkSession
import os

# Elegir el máster de Spark dependiendo de si se ha definido la variable de entorno HADOOP_CONF_DIR o YARN_CONF
SPARK_MASTER: str = 'yarn' if 'HADOOP_CONF_DIR' in os.environ or 'YARN_CONF_DIR' in os.environ else 'local[*]'

# Creamos un objeto SparkSession (o lo obtenemos si ya está creado)
spark: SparkSession = SparkSession \
  .builder \
  .appName("Mi aplicacion") \
  .config("spark.rdd.compress", "true") \
  .config("spark.executor.memory", "3g") \
  .config("spark.driver.memory", "3g") \
  .master(SPARK_MASTER) \
  .getOrCreate()

sc: SparkContext = spark.sparkContext
```

Creación de RDDs

Podemos crear RDDs a partir de una colección de datos o desde ficheros

RDDs a partir de una colección

- Utilizan el SparkContext (sc en el terminal o el notebook)

In [ ]:
```
# Ejemplo en PySpark
from pyspark import RDD
from pprint import pp

rdd1: RDD[int] = sc.parallelize([1,2,3,4,5,6,7,8])
print(rdd1.collect())

import numpy as np
rdd2: RDD[np.int64] = sc.parallelize(np.array(range(100)))
pp(rdd2.collect())

# Los RDDs aceptan listas de tipos diferentes
rdd5: RDD[int | str] = sc.parallelize([1,2,"tres",4])
pp(rdd5.collect())
```

RDDs y DataSets a partir de ficheros de texto

In [ ]:
```
from pyspark import SparkFiles
from pprint import pp

# Añadimos un archivo al contexto de Spark (los descarga en cada nodo)
sc.addFile("https://raw.githubusercontent.com/dsevilla/tcdm-public/refs/heads/24-25/datos/quijote.txt.gz")

quijoteRDD: RDD[str] = sc.textFile("file://" + SparkFiles.get("quijote.txt.gz"))

pp(quijoteRDD.take(1000))
```

Particiones

El número de particiones de un RDD puede especificarse en el momento de crearse

- También se puede modificar una vez creados (`repartition` o `coalesce`)

- El método `glom` permite ver cómo se han creado las particiones

In [ ]:
```
rdd: RDD[int] = sc.parallelize([1,2,3,4], 2)
print(rdd.glom().collect())
print(rdd.getNumPartitions())
```

RDDs y DataFrames

Un DataFrame tiene un RDD subyacente, al que podemos acceder de forma simple

In [ ]:
```
from pyspark.rdd import RDD
from pyspark.sql.dataframe import DataFrame
from pyspark.sql.types import Row

sc.addFile("https://raw.githubusercontent.com/dsevilla/tcdm-public/refs/heads/24-25/datos/2015-summary.csv")

dfDatosVuelos2015: DataFrame = (spark
    .read
    .option("inferSchema", "true")
    .option("header", "true")
    .csv("file://" + SparkFiles.get("2015-summary.csv")))

rddDatosVuelos2015: RDD[Row] = dfDatosVuelos2015.rdd

rddDatosVuelos2015.take(10)
```

In [ ]:
```
# Se puede crear un DataFrame a partir de un rdd
dfNuevo: DataFrame = rddDatosVuelos2015.toDF()
dfNuevo.printSchema()
dfNuevo.show()
```

RDDs (y DataSets) simples

RDDs formados por elementos simples.

Transformaciones sobre un único RDD o DataSet

Generan un nuevo RDD a partir de uno dado modificando cada uno de los elementos del original

- `filter(func)` filtra los elementos de un RDD

In [ ]:
```
quijsRDD: RDD[str] = quijoteRDD.filter(lambda line: "Quijote" in line)
sanchsRDD: RDD[str] = quijoteRDD.filter(lambda line: "Sancho" in line)
quijssancsRDD: RDD[str] = quijsRDD.intersection(sanchsRDD)
quijssancsRDD.cache()
print("Líneas con Quijote y Sancho: {0}.".format(quijssancsRDD.count()))
for line in quijssancsRDD.takeSample(False, 10):
    pp(line)
```

In [ ]:
```
# Obtener los valores positivos de un rango de números
rdd: RDD[int] = sc.parallelize(range(-5,5))          # Rango [-5, 5)
filtered_rdd: RDD[int] = rdd.filter(lambda x: x >= 0)   # Devuelve los positivos

assert filtered_rdd.collect() == [0, 1, 2, 3, 4]
```

- `map(func)` aplica una función a los elementos de un RDD

In [ ]:
```
# Añade 1 a cada elemento del RDD
# Para cada elemento, obtiene una tupla (x, x**2)
def add1(x: int) -> int:
    return x+1

squared_rdd: RDD[tuple] = (filtered_rdd
               .map(add1)                 # Añade 1 a cada elemento del RDD
               .map(lambda x: (x, x*x)))  # Para cada elemento, obtén una tupla (x, x**2)

assert squared_rdd.collect() == [(1, 1), (2, 4), (3, 9), (4, 16), (5, 25)]
```

- `flatMap(func)` igual que `map`, pero “aplana” la salida

In [ ]:
```
squaredflat_rdd: RDD[int] = (filtered_rdd
                  .map(add1)
                  .flatMap(lambda x: (x, x*x)))  # Da la salida en forma de lista

assert squaredflat_rdd.collect() == [1, 1, 2, 4, 3, 9, 4, 16, 5, 25]
```

- `sample(withReplacement, fraction, seed=None)` devuelve una muestra del RDD o DataSet
  - `withReplacement` - si True, cada elemento puede aparecer varias veces en la muestra
  - `fraction` - tamaño esperado de la muestra como una fracción del tamaño del RDD
    - **sin reemplazo**: probabilidad de seleccionar un elemento, su valor debe ser [0, 1]
    - **con reemplazo**: En RDD número esperado de veces que se escoge un elemento, su valor debe ser >= 0
  - `seed` - semilla para el generador de números aleatorios

In [ ]:
```
srdd1: RDD[int] = squaredflat_rdd.sample(False, 0.5)
srdd2: RDD[int] = squaredflat_rdd.sample(True, 2)
srdd3: RDD[int] = squaredflat_rdd.sample(False, 0.8, 14)
print('s1={0}\ns2={1}\ns3={2}'.format(srdd1.collect(), srdd2.collect(), srdd3.collect()))
```

- `distinct()` devuelve un nuevo RDD o DataSet sin duplicados
  - El orden de la salida no está definido

In [ ]:
```
distinct_rdd: RDD[int] = squaredflat_rdd.distinct()
pp(distinct_rdd.collect())
```

- `groupBy(func)` devuelve un RDD con los datos agrupados en formato clave/valor, usando una función para obtener la clave

In [ ]:
```
from collections.abc import Iterable

grouped_rdd: RDD[tuple[int, Iterable[int]]] = distinct_rdd.groupBy(lambda x: x%3)
pp(grouped_rdd.collect())
pp([(x, sorted(y)) for (x,y) in grouped_rdd.collect()])
```

Transformaciones sobre dos RDDs o DataSets

Operaciones tipo conjunto sobre dos RDDs o DataSets

- `rdda.union(rddb)` devuelve un RDD o DataSet con la unión de los datos de los dos de partida (se comparan como set porque no se garantiza el orden del resultado)

In [ ]:
```
rdda: RDD[str] = sc.parallelize(['a', 'b', 'c'])
rddb: RDD[str] = sc.parallelize(['c', 'd', 'e'])
rddu: RDD[str] = rdda.union(rddb)
assert set(rddu.collect()) == {'a', 'b', 'c', 'c', 'd', 'e'}
```

- `rdda.intersection(rddb)` devuelve un RDD o DataSet con los datos comunes en los dos

In [ ]:
```
rddi: RDD[str] = rdda.intersection(rddb)
assert rddi.collect() == ['c']
```

- `rdda.subtract(rddb)` devuelve un RDD con los datos del primero menos los del segundo (sólo RDDs)

In [ ]:
```
rdds: RDD[str] = rdda.subtract(rddb)
assert set(rdds.collect()) == {'a', 'b'}
```

`rdda.cartesian(rddb)` producto cartesiano de ambos RDDs (operación muy costosa) (sólo RDDs) (de nuevo se usan conjuntos para no tener en cuenta el orden exacto si se hacen optimizaciones)

In [ ]:
```
rddc: RDD[tuple[str, str]] = rdda.cartesian(rddb)
assert set(rddc.collect()) == {('a','c'),('a','d'),('a','e'),('b','c'),('b','d'),
                               ('b','e'),('c','c'), ('c','d'), ('c','e')}
```

Acciones sobre RDDs o DatSets simples

Obtienen datos (simples o compuestos) a partir de un RDD o DataSet.

Principales acciones de agregación: `reduce` y `fold`.

- `reduce(op)` combina los elementos de un RDD o DataSet en paralelo, aplicando un operador
  - El operador de reducción debe ser un *monoide conmutativo* (operador binario asociativo y conmutativo)
  - Primero se realiza la redución a nivel de partición y luego se van reduciendo los valores intermedios

In [ ]:
```
rdd: RDD[int] = sc.parallelize(range(1,10), 2)  # rango [1, 10)
pp(rdd.glom().collect())
```

```
# Reducción con una función lambda
p = rdd.reduce(lambda x,y: x*y) # r = 1*2*3*4*5*6*7*8*9 = 362880
pp("1*2*3*4*5*6*7*8*9 = {0}".format(p))

# Reducción con un operador predefinido
from operator import add
s = rdd.reduce(add) # s = 1+2+3+4+5+6+7+8+9 = 45
pp("1+2+3+4+5+6+7+8+9 = {0}".format(s))

# Prueba con un operador no conmutativo
p = rdd.reduce(lambda x,y: x-y) # r = 1-2-3-4-5-6-7-8-9 = -43
pp("1-2-3-4-5-6-7-8-9 = {0}".format(p))

# No funciona con RDDs vacíos
#sc.parallelize([]).reduce(add)
```

- `fold(cero, op)` versión general de `reduce` (sólo con RDDs):
  - Debemos proporcionar un valor inicial `cero` para el operador
  - El valor inicial debe ser el valor identidad para el operador (p.e. 0 para suma; 1 para producto, o una lista vacía para concatenación de listas)
    - Permite utilizar RDDs vacíos
  - La función `op` debe ser un monoide conmutativo para garantizar un resultado consistente
    - Comportamiento diferente a las operaciones `fold` de lenguajes como Scala
    - El operador se aplica a nivel de partición (usando `cero` como valor inicial), y finalmente entre todas las particiones (usando `cero` de nuevo)
    - Para operadores no conmutativos el resultado podría ser diferente del obtenido mediante un `fold` secuencial

In [ ]:
```
rdd: RDD[list] = sc.parallelize([[1,2,3,4], [-10, -9, -8, -7, -6], ['a', 'b', 'c']], 2)
pp(rdd.glom().collect())

f: list[int] = rdd.fold([], lambda x,y: x+y)
pp(f)

# Se puede hacer un fold de un RDD vacío
sc.parallelize([]).fold(0, add)
```

Otras acciones de agregación:

- `aggregate` (solo RDDs)

- `aggregate(cero,seqOp,combOp)` : Devuelve una colección agregando los elementos del RDD usando dos funciones:
  1. `seqOp` - agregación a nivel de partición: se crea un acumulador por partición (inicializado a `cero`) y se agregan los valores de la partición en el acumulador
  2. `combOp` - agregación entre particiones: se agregan los acumuladores de todas las particiones
  - Ambas agregaciones usan un valor inicial `cero` (similar al caso de `fold`).
  - Versión general de `reduce` y `fold`
  - La primera función (`seqOp`) puede devolver un tipo, U, diferente del tipo T de los elementos del RDD
  - `seqOp` agregar datos de tipo T y devuelve un tipo U
  - `combOp` agrega datos de tipo U
  - `cero` debe ser de tipo U
  - Permite devolver un tipo diferente al de los elementos del RDD de entrada.

In [ ]:
```
from collections.abc import Callable

lista_enteros: list[int] = [1, 2, 3, 4, 5, 6, 7, 8]
rdd: RDD[int] = sc.parallelize(lista_enteros)

# acc es una tupla de tres elementos (List, Double, Int)
# En el primer elemento de acc (lista) le concatenamos los elementos del RDD al cuadrado
# en el segundo, acumulamos los elementos del RDD usando multiplicación
# y en el tercero, contamos los elementos del RDD
seqOp: Callable[[tuple, int], tuple]  = (lambda acc, val: (acc[0]+[val*val],
                                                           acc[1]*val,
                                                           acc[2]+1))
# Para cada partición se genera una tupla tipo acc
# En esta operación se combinan los tres elementos de las tuplas
combOp: Callable[[tuple, tuple], tuple] = (lambda acc1, acc2: (acc1[0]+acc2[0],
                                                               acc1[1]*acc2[1],
                                                               acc1[2]+acc2[2]))

a: tuple = rdd.aggregate(([], 1., 0), seqOp, combOp)

pp(a)

assert a[1] == 8.*7.*6.*5.*4.*3.*2.*1.
```

```
assert a[2] == len(lista_enteros)
```

Acciones para contar elementos sobre RDDs

- `count()` devuelve un entero con el número exacto de elementos del RDD (también DataSets)
- `countApprox(timeout, confidence=0.95)` versión aproximada de `count()` que devuelve un resultado potencialmente incompleto en un tiempo máximo, incluso si no todas las tareas han finalizado. (Experimental).
  - `timeout` es un entero largo e indica el tiempo en milisegundos
  - `confidence` probabilidad de obtener el valor real. Si `confidence` es 0.90 quiere decir que si se ejecuta múltiples veces, se espera que el 90% de ellas se obtenga el valor correcto. Valor [0,1]
- `countApproxDistinct(relativeSD=0.05)` devuelve una estimación del número de elementos diferentes del RDD. (Experimental).
  - `relativeSD` – exactitud relativa (valores más pequeños implican menor error, pero requieren más memoria; debe ser mayor que 0.000017).

In [ ]:
```
rdd: RDD[int] = sc.parallelize([i % 20 for i in range(10000)], 16)
print("Número total de elementos: {0}.".format(rdd.count()))
print("Número de elementos distintos: {0}.".format(rdd.distinct().count()))

print("Número total de elementos (aprox.): {0}.".format(rdd.countApprox(1, 0.4)))
print("Número de elementos distintos (approx.): {0}.".format(rdd.countApproxDistinct(0.5)))
```

- `countByValue()` devuelve el número de apariciones de cada elemento del RDD como un mapa (o diccionario) de tipo clave/valor
  - Las claves son los elementos del RDD y cada valor, el número de ocurrencias de la clave asociada al mismo

In [ ]:
```
rdd: RDD[str] = sc.parallelize(list("abracadabra")).cache()
mimapa: dict[str, int] = rdd.countByValue()

pp(mimapa.items())
```

Acciones para obtener valores

- Estos métodos deben usarse con cuidado, si el resultado esperado es muy grande puede saturar la memoria del *driver*

- `collect()` devuelve una lista con todos los elementos del RDD o DataSet
- `show()` muestra los elementos del DataSet como una tabla

In [ ]:
```
lista: list[str] = rdd.collect()
print(lista)
```

- `take(n)` devuelve los `n` primeros elementos del RDD o las n primeras filas del DataSet
- `takeAsList(n)` devuelve las `n` primeras filas del DataSet como una lista
- `takeSample(withRep, n, [seed])` devuelve `n` elementos aleatorios del RDD
  - `withRep` : si True, en la muestra puede aparecer el mismo elemento varias veces
  - `seed` : semilla para el generador de números aleatorios

In [ ]:
```
t: list[str] = rdd.take(4)
print(t)
s: list[str] = rdd.takeSample(False, 4)
print(s)
```

- `top(n)` devuelve una lista con los primeros `n` elementos del RDD ordenados en orden descendente
- `takeOrdered(n,[orden])` devuelve una lista con los primeros `n` elementos del RDD en orden ascendente (opuesto a `top`), o siguiendo el orden indicado en la función opcional

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

print(f"4 elementos más grandes: {rdd.top(4)}")

print(f"4 elementos más pequeños: {rdd.takeOrdered(4)}")

print(f"4 elementos más grandes: {rdd.takeOrdered(4, lambda x: -x)}")
```

RDDs con pares clave/valor (aka *Pair RDDs*)

- Tipos de datos muy usados en Big Data (MapReduce)

- Spark dispone de operaciones especiales para su manejo

Creación de *Pair RDDs*

Los RDDs clave/valor pueden crearse a partir de una lista de tuplas, a partir de otro RDD o mediante un zip de dos RDDs.

- A partir de una lista de tuplas

In [ ]:
```
prdd: RDD[tuple[str, int]] = sc.parallelize([('a',2), ('b',5), ('a',3)])
print(prdd.collect())

prdd = sc.parallelize(zip(['a', 'b', 'c'], range(3)))
print(prdd.collect())
```

- A partir de otro RDD

In [ ]:
```
# Ejemplo usando un fichero
# Para cada línea ontenemos una tupla, siendo el primer elemento
# la primera palabra de la líneas, y el segundo la línea completa
linesrdd: RDD[str] = sc.textFile("file://" + SparkFiles.get("quijote.txt.gz"))
prdd: RDD[tuple[str, str]] = linesrdd.map(lambda x: (x.split(" ")[0], x))

pp(f"Par (1ª palabra, línea): {prdd.takeSample(False, 2)}")
```

In [ ]:
```
# Usando keyBy(f): Crea tuplas de los elementos del RDD usando f para obtener la clave.
nrdd: RDD[int] = sc.parallelize(range(2,5))
prdd: RDD[tuple[int, int]] = nrdd.keyBy(lambda x: x*x)

print(prdd.collect())
```

In [ ]:
```
# zipWithIndex(): Zipea el RDD con los índices de sus elementos.
rdd: RDD[str] = sc.parallelize(['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h'], 3)
prdd: RDD[tuple[str, int]] = rdd.zipWithIndex()
print(rdd.glom().collect())

print(prdd.collect())

# Este método dispara un Spark job cuando el RDD tiene más de una partición.
```

In [ ]:
```
# zipWithUniqueId(): Zipea el RDD con identificadores únicos (long) para cada elemento.
# Los elementos en la partición k-ésima obtienen los ids k, n+k, 2*n+k,... siendo n = nº de particiones
# No dispara un trabajo Spark
rdd: RDD[str] = sc.parallelize(['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h'], 3)
print("Particionado del RDD: {0}".format(rdd.glom().collect()))
prdd: RDD[tuple[str, int]] = rdd.zipWithUniqueId()

print(prdd.collect())
```

- Mediante un zip de dos RDDs
  - Los RDDs deben tener el mismo número de particiones y el mismo número de elementos en cada partición

In [ ]:
```
rdd1: RDD[int] = sc.parallelize(range(0, 5), 2)
rdd2: RDD[int] = sc.parallelize(range(1000, 1005), 2)
prdd: RDD[tuple[int, int]] = rdd1.zip(rdd2)

print(prdd.collect())
```

Transformaciones sobre un único RDD clave/valor

Sobre un único RDD clave/valor podemos efectuar transformaciones de agregación a nivel de clave y transformaciones que afectan a las claves o a los valores

Transformaciones de agregación

- `reduceByKey(func)` / `foldByKey(func)`
  - Devuelven un RDD, agrupando los valores asociados a la misma clave mediante `func`
  - Similares a `reduce` y `fold` sobre RDDs simples

In [ ]:
```
from operator import add
prdd: RDD[tuple[str, int]] = sc.parallelize([('a', 2), ('b', 5), ('a', 8), ('b', 6), ('b', 2),('c',13)],2).cach
print(prdd.glom().collect())
redrdd: RDD[tuple[str, int]] = prdd.reduceByKey(add)

print(redrdd.sortByKey().collect())
```

- `groupByKey()` agrupa valores asociados a misma clave
  - Operación muy costosa en comunicaciones
  - Mejor usar operaciones de reducción

In [ ]:
```
from collections.abc import Iterable

grouprdd: RDD[tuple[str, Iterable[int]]] = prdd.groupByKey()

print(grouprdd.collect())

lista: list[tuple[str, list[int]]] = [(k, list(v)) for k, v in grouprdd.collect()]
print(lista)
```

- `combineByKey(createCombiner(func1), mergeValue(func2), mergeCombiners(func3))`
  - Método general para agregación por clave, similar a `aggregate`
  - Especifica tres funciones:
    1. `createCombiner` al recorrer los elementos de cada partición, si nos encontramos una clave nueva se crea un acumulador y se inicializa con `func1`
    2. `mergeValue` mezcla los valores de cada clave en cada partición usando `func2`
    3. `mergeCombiners` mezcla los resultados de las diferentes particiones mediante `func3`
- Los valores del RDD de salida pueden tener un tipo diferente al de los valores del RDD de entrada.

In [ ]:
```
# Para cada clave, obten una tupla que tenga la suma y el número de valores
sumCount: RDD[tuple[str, tuple[int,int]]] = prdd.combineByKey(
                            (lambda x: (x, 1)),
                            (lambda x, y: (x[0]+y, x[1]+1)),
                            (lambda x, y: (x[0]+y[0], x[1]+y[1])))

print(sumCount.collect())

# Con el RDD anterior, obtenemos la media de los valores
m: RDD[tuple[str, float]] = sumCount.mapValues(lambda v: float(v[0])/v[1])
print(m.collect())
```

Transformaciones sobre claves o valores

- `keys()` devuelve un RDD con las claves
- `values()` devuelve un RDD con los valores
- `sortByKey()` devuelve un RDD clave/valor con las claves ordenadas

In [ ]:
```
print("RDD completo: {0}".format(prdd.collect()))
print("RDD con las claves: {0}".format(prdd.keys().collect()))
print("RDD con los valores: {0}".format(prdd.values().collect()))
print("RDD con las claves ordenadas: {0}".format(prdd.sortByKey().collect()))
```

- `mapValues(func)` devuelve un RDD aplicando una función sobre los valores
- `flatMapValues(func)` devuelve un RDD aplicando una función sobre los valores y "aplanando" la salida

In [ ]:
```
mapv: RDD[tuple[str, tuple[int, int]]] = prdd.mapValues(lambda x: (x, 10*x))
print(mapv.collect())

fmapv: RDD[tuple[str , int]] = prdd.flatMapValues(lambda x: (x, 10*x))
print(fmapv.collect())
```

Transformaciones sobre dos RDDs clave/valor

Combinan dos RDDs de tipo clave/valor para obtener un tercer RDD.

`join` / `leftOuterJoin` / `rightOuterJoin` / `fullOuterJoin` realizan inner/outer/full joins entre los dos RDDs

In [ ]:
```
rdd1: RDD[tuple[str, int]] = sc.parallelize([("a", 2), ("b", 5), ("a", 8)]).cache()
rdd2: RDD[tuple[str, int]] = sc.parallelize([("c", 7), ("a", 1)]).cache()

rdd3: RDD[tuple[str, tuple[int, int]]] = rdd1.join(rdd2)

print(rdd3.collect())
```

In [ ]:
```
rdd3: RDD[tuple[str, tuple[int, int | None]]] = rdd1.leftOuterJoin(rdd2)

print(rdd3.collect())
```

In [ ]:
```
rdd3: RDD[tuple[str, tuple[int | None, int]]] = rdd1.rightOuterJoin(rdd2)
```

In [ ]:
```
rdd3: RDD[tuple[str, tuple[int | None, int]]] = rdd1.rightOuterJoin(rdd2)

print(rdd3.collect())
```

In [ ]:
```
rdd3: RDD[tuple[str, tuple[int | None, int | None]]] = rdd1.fullOuterJoin(rdd2)

print(rdd3.collect())
```

- `subtractByKey` elimina elementos con una clave presente en otro RDD

In [ ]:
```
rdd3: RDD[tuple[str, int]] = rdd1.subtractByKey(rdd2)

print(rdd3.collect())
```

- `cogroup` agrupa los datos que comparten la misma clave en ambos RDDs

In [ ]:
```
from pyspark.resultiterable import ResultIterable

rdd3: RDD[tuple[str, tuple[ResultIterable[int], ResultIterable[int]]]] = rdd1.cogroup(rdd2)

print(rdd3.collect())

map: dict[str, list[list[int]]] = rdd3.mapValues(lambda v: [list(l) for l in v]).collectAsMap()

print(map)
```

Acciones sobre RDDs clave/valor

Sobre los RDDs clave/valor podemos aplicar las acciones para RDDs simples y algunas adicionales.

- `collectAsMap()` obtiene el RDD en forma de mapa

In [ ]:
```
prdd: RDD[tuple[str, int]] = sc.parallelize([("a", 7), ("b", 5), ("a", 8)]).cache()

Mapa: dict[str, int] = prdd.collectAsMap()
print(Mapa)
```

- `countByKey()` devuelve un mapa indicando el número de ocurrencias de cada clave

In [ ]:
```
countMap: dict[str, int] = prdd.countByKey()

print(countMap)
```

- `lookup(key)` devuelve una lista con los valores asociados con una clave

In [ ]:
```
listA: list[int] = prdd.lookup('a')

print(listA)
```

RDDs numéricos

Funciones de estadística descriptiva implementadas en Spark

Método	Descripción
stats()	Resumen de estadísticas
mean()	Media aritmética
sum(), max(), min()	Suma, máximo y mínimo
variance()	Varianza de los elementos
sampleVariance()	Varianza de una muestra
stdev()	Desviación estándar
sampleStdev()	Desviación estándar de una muestra
histogram()	Histograma

In [ ]:
```
import numpy as np
from pyspark.statcounter import StatCounter

# Un RDD con datos aleatorios de una distribución normal
nrdd: RDD[float] = sc.parallelize(np.random.normal(size=100000)).cache()

# Resumen de estadísticas
```

```
sts: StatCounter = nrdd.stats()

print("Resumen de estadísticas:\n {0}\n".format(sts))
```

In [ ]:
```
from math import fabs

# Filtra outliers
stddev = sts.stdev()
avg = sts.mean()

frdd: RDD[float] = nrdd.filter(lambda x: fabs(x - avg) < 3*stddev).cache()

print("Número de outliers: {0}".format(sts.count() - frdd.count()))
```

In [ ]:
```
import matplotlib.pyplot as plt; plt.rcdefaults()

# Obtiene un histograma con 10 grupos
x,y = frdd.histogram(10)

# Limpia la gráfica
plt.gcf().clear()

plt.bar(x[:-1], y, width=0.6)
plt.xlabel(u'Valores')
plt.ylabel(u'Número de ocurrencias')
plt.title(u'Histograma')

plt.show()
```

Salvar RDDs

Los RDDs se pueden salvar a disco como ficheros de texto, ficheros Sequence y, en general, tipo de ficheros que pueda escribir Hadoop.

In [ ]:
```
# Salva como fichero de texto
# Se crea un fichero por partición
nrdd.saveAsTextFile("file:///tmp/nrdd")
```

In [ ]:
```
%%sh
ls -lh /tmp/nrdd
head /tmp/nrdd/part-00000
rm -rf /tmp/nrdd
```

In [ ]:
```
# Salva como fichero de texto comprimido
nrdd.saveAsTextFile("file:///tmp/nrdd-bzip", "org.apache.hadoop.io.compress.BZip2Codec")
```

In [ ]:
```
%%sh
ls -lh /tmp/nrdd-bzip
rm -rf /tmp/nrdd-bzip
```

Guardar como fichero Sequence

Tiene que ser un RDD clave valor

In [ ]:
```
rdd: RDD[tuple[str, int]] = sc.parallelize([("a",2), ("b",5), ("a",8)], 2)

# Salvamos el RDD clave valor como fichero Sequence
rdd.saveAsSequenceFile("file:///tmp/rddsequence")
```

In [ ]:
```
%%sh
ls -lh /tmp/rddsequence
```

In [ ]:
```
# Recuperamos en otro rdd
rdd2: RDD = sc.sequenceFile("file:///tmp/rddsequence",
                        "org.apache.hadoop.io.Text",
                        "org.apache.hadoop.io.IntWritable")

print("Contenido del RDD {0}".format(rdd2.collect()))
```

In [ ]:
```
%%sh
rm -rf /tmp/rddsequence
```