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 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: conexión con HBase

- Se maneja con sqlContext, como acceso a repositorio externo
  - Habilitar el acceso a metadatos en `/opt/startup/hive-site.xml`

```
<property>
<name>hive.metastore.schema.verification</name>
<value>true</value>
</property>
```

  - Sobre una tabla creada en Hbase (si no existe, crearla y agregarle datos)

```
> hbase shell
> create 'tablaEj', ‘col'
> put 'tablaEj', '1', ‘col:ColumnaA', ‘datos columna A'
> put 'tablaEj', '1', ‘col:ColumnaB', ‘datos columna B’
> exit
```

  - En pyspark, acceder a la tabla HBase

```
>>> hTbl = sqlContext.read.format('org.apache.hadoop.hbase.spark’).
option('hbase.table',’tablaEj').option('hbase.columns.mapping',
'KEY_FIELD STRING :key, A STRING col:ColumnaA, B STRING
col:ColumnaB’).option('hbase.use.hbase.context',False).option('hbase
.config.resources', 'file:///etc/hbase/conf/hbase-site.xml').load()
```

Apache Spark: conexión con HBase

```
>>> hTbl.show()
+---------+-----------------+---------------+
|KEY_FIELD|                A|              B|
+---------+-----------------+---------------+
|1        | datos columna A | datos columna B|
+---------+-----------------+---------------+
>>> hTbl = sqlContext.read.format('org.apache.hadoop.hbase.spark').
option('hbase.table','stations').option('hbase.columns.mapping','NAME
STRING :key, LOCATION STRING info:location, DESCRIPTION STRING
info:description').option('hbase.use.hbase.context', False).
option('hbase.config.resources', 'file:///etc/hbase/conf/hbase-
site.xml').load()
>>> hTbl.show()
+---------+-----------+-----------+
| LOCATION|       NAME|DESCRIPTION|
+---------+-----------+-----------+
|(unknown)|010000-99999|  (unknown)|
|(unknown)|010003-99999|  (unknown)|
|(unknown)|010010-99999|  (unknown)|
...
```

Apache Spark: agregaciones y ordenamiento

- El agrupamiento u ordenamiento de datos se realiza previamente a la aplicación de funciones de suma y conteo.
- RDD.groupBy(<función>, numPartitions)
  - Retorna un RDD de elementos agrupados aplicando la función. La función puede simplemente indicar una clave para agrupar elementos, indicar una expresión para agrupar (por ejemplo: números pares o impares).
  - numPartitions permite especificar el número de particiones (por ejemplo, agrupar un RDD por los días de la semana, numPartitions=7).

- Cuidado: groupBy no es eficiente, considerar utilizar otras operaciones
  - aggregateByKey y reduceByKey (trabajan sobre pair RDDs)
  - groupBy no agrega antes de realizar shuffling, por lo cual se procesan y transmiten más datos
  - groupBy require que todos los valores para una clave entren en memoria.

Apache Spark: agregaciones y ordenamiento

- RDD.sortBy(<keyfunc>, ascending, numPartitions)
  - Ordena un RDD por la función que indica la clave para un dataset.
  - Ordena de acuerdo al tipo de dato de la clave (numérico para int, lexicográfico para strings, etc.).
  - Ordenar enteros como strings:

```
b = sc.parallelize([('t', 3),('b', 4),('c', 1)])
bSorted = b.sortBy(lambda a: a[1])
bSorted.collect()
[('c', 1),('t', 3),('b', 4)]
```

- RDD.distinct(numPartitions=None)
  - Retorna un nuevo RDD con los elementos distintos del RDD de entrada.
- Ejemplo: procesamiento de logs web
  - Retornar registros con código de respuesta único.
  - Retornar registros <código de respuesta, puerto [único]> para cada código de respuesta.

Apache Spark: agregaciones y ordenamiento

Esquema

```
# field 0 : date
# field 1 : time
# field 2 : ip
# field 3 : username
# field 4 : sitename
# field 5 : computername
# field 6 : ip
# field 7 : port
# field 8 : method
# field 9 : uri-stem
# field 10 : uri-query
# field 11 : status
# field 12 : time-taken
# field 13 : version
# field 14 : host
# field 15 : user-agent
# field 16 : referer
```

Ejemplos

```
logs = sc.textFile('file:///…/spark/data/weblogs')
logrecs = logs.map(lambda x: x.split(' '))
reqfieldsonly = logrecs.map(lambda x: (x[7], x[11]))
# [(u'80', u'200'), (u'80', u'200'), (u'80',
u'200'), ...]
distinctrecs = reqfieldsonly.distinct()
# [(u'80', u'200'), (u'443', u'200'), (u'443',
u'500'), ...]
sorted = distinctrecs.sortBy(lambda x: x[1]) \
.map(lambda x: (x[1], x[0]))
# [(u'200', u'443'), (u'200', u'80'), (u'206',
u'80'), ...]
grouped = sorted.groupBy(lambda x: x[0]) \
.map(lambda x: (x[0], list(x[1])))
# [(u'200', [(u'200', u'443'), (u'200', u'80')]),
...]
```

Apache Spark: obtener datos

- RDD.collect()
  - Retorna (al driver) una lista que contiene todos los elementos del RDD.
  - No restringe el tamaño de la salida ! Puede causar problemas de eficiencia e inclusive fallos de memoria en el driver.
  - Solo es útil para desarrollos pequeños y de verificación.

```
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: obtener datos

- RDD.take(n)
  - Retorna los primeros n elementos de un RDD.
  - Orden aleatorio: acción no-determinista (en especial en entornos distribuidos).
  - Para RDDs que abarcan más de una partición, take analiza una partición y estima el número de particiones adicionales que serán necesarias para retornar los n elementos.

```
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’]
```

  - Existe una operación RDD.takeOrdered que retorna los primeros n elementos de acuerdo a un ordenamiento dado por una función.

Apache Spark: obtener datos

- RDD.top(n,[key])
  - Retorna los primeros n elementos, considerando el ordenamiento (en orden descendente) de acuerdo al tipo de dato de la clave (numérico para int, lexicográfico para strings, etc.).

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

  - Si se proporciona una clave, se utiliza para ordenar los elementos del RDD.

```
# (numbersofrating,title,avg_rating)
newRDD = [(3,'monster',4),(4,'minions 3D',5),....]
newRDD.top(2,key=lambda x:x[2])
[(4,'minions 3D',5),(3,'monster',4)....]
```

  - takeOrdered funciona de modo similar, pero con orden ascendente

```
newRDD.takeOrdered(2,key=lambda x:-x[2])
```

Apache Spark: obtener datos

- RDD.first()
  - Retorna el primer elemento en el RDD.
  - Opera de modo similar a take y collect (pero NO como top), no considera un orden para los elementos.
  - Es una operación no determinista.
  - La diferencia entre first y take(1) es que first retorna un elemento (operación atómica) y take siempre retorna una lista, aún para n = 1.
  - La acción first es útil para inspeccionar datos en fases de desarrollo o exploración de datos.

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

Apache Spark: reducciones

- reduce() y fold(): acciones de agregación que ejecutan operaciones conmutativas [x + y = y + x] y asociativas [(x + y) + z = x + (y + z)].
  - Clave: las operaciones son independientes del orden en el cual se aplican. Fundamental en procesamiento distribuido, donde el orden de procesamiento no puede asegurarse.
- RDD.reduce(<función>): reduce los elementos del RDD aplicando la función.
  - La función opera sobre dos parámetros o valores (lambda x, y: ...) que representan valores en secuencia del RDD.

```
numbers = sc.parallelize([1,2,3,4,5,6,7,8,9])
numbers.reduce(lambda x, y: x + y)
```

  - Diferente a reduceByKey, que veremos más adelante:
    - reduceByKey es una transformación (retorna un RDD), reduce es una acción (retorna un valor).
    - Reduce agrega particiones localmente, reduceByKey puede forzar reparticionar (los datos con la misma clave deben ser reducidos por el mismo nodo).

Apache Spark: reducciones

- RDD.fold(zeroValue,<función>)
  - Agrega los elementos de cada partición de un RDD y luego aplica la función (asociativa y conmutativa) y el valor cero indicado.

```
numbers = sc.parallelize([1,2,3,4,5,6,7,8,9])
numbers.fold(0, lambda x, y: x + y)
```

  - El valor cero se aplica al inicio y al final de la aplicación de la función.
  - Permite operar sobre un RDD vacío ! (RDD.reduce() produce una excepción en ese caso).

```
result = zeroValue + (1 + 2) + 3 . . + zeroValue
```

  - Comparación de RDD.fold() y RDD.reduce()

```
empty = sc.parallelize([])
empty.reduce(lambda x, y: x + y)
# ValueError: Can not reduce() empty RDD
empty.fold(0, lambda x, y: x + y)
# 0
```

Apache Spark: reducciones

- RDD.foreach(<función>)
  - Aplica una función a todos los elementos de un RDD.
  - Es una función (no una transformación): permite aplicar funciones que no es posible usar en tranformaciones, por ejemplo una función print.
  - Las funciones lambda de Python no permiten utilizar un print directamente, pero es posible utilizar una función nominada para ejecutar el print.

```
def printfunc(x): print(x)
lorem = sc.textFile('file:///opt/spark/data/lorem.txt')
words = lorem.flatMap(lambda x: x.split())
words.foreach(lambda x: printfunc(x))
# lorem
# ipsum
# dolor
# sit
# amet
# consectetur
# adipiscing
# elit
# nullam
```

Foreach es una acción, por lo cual produce la evaluación de todo el lineage del RDD.
Si esta no es la idea, la función map puede ser una mejor opción.

Apache Spark: datos clave-valor

- Clave-valor: tipo de dato muy común para varias operaciones en Spark.
  - Los RDDs de pares clave-valor exponen operaciones específicas (conteo, agrupamiento, etc.)
- RDD clave-valor (*pair RDDs*) son el bloque básico de aplicaciones en Spark
- Las claves pueden ser objetos simples (entero, string) o complejos como tuplas. Los valores pueden ser escalares, listas, tuplas, diccionarios o conjuntos.
- Es un tipo de dato utilizado para el almacenamiento y análisis de datos sobre sistemas NoSQL.
- Usualmente se extraen campos de un RDD y se usan como clave en las operaciones.

Apache Spark: datos clave-valor

- Una funcionalidad muy importante: particionamiento.
  - Permite controlar cómo se distribuyen los RDD entre nodos.
  - Permiten acceder a datos locales, reducir las comunicaciones y mejorar el desempeño.
  - Elegir la partición apropiada para un RDD es similar a elegir una estructura de datos correcta: simplifica la programación y mejora el desempeño.
- Operaciones específicas (y muy útiles):
  - reduceByKey(): agrega datos de manera separada para cada clave.
  - join() combina dos RDDs agrupando elementos con la misma clave.

Apache Spark: datos clave-valor

- Crear pair RDDs
  - Métodos de carga de datos pueden generar pair RDDs.
  - Función map(): convertir RDDs regulares en pair RDD.
  - Ejemplo: crear pair RDD a partir de líneas de texto, la primer palabra debe ser la clave.

```python
pairs = lines.map(lambda x: (x.split(" ")[0], x))
```
(Python)

  - En Scala, para que las funciones sobre datos esten disponibles, es necesario retornar tuplas. Se puede aplicar una conversión implícita de RDDs a tuplas.

```scala
val pairs = lines.map(x => (x.split(" ")(0), x))
```
(Scala)

  - Cuando se crea un pair RDD a partir de una colección en memoria, solo debe invocarse SparkContext.parallelize() en la colección.

Apache Spark: operaciones sobre clave-valor

- Funciones de diccionario: retornan un conjunto de claves o valores
- RDD.keys(): Retorna un RDD con las claves de un par clave-valor o el primer elemento de cada tupla en un pair RDD.

```
kvpairs = sc.parallelize([('city','Hayward'),('state','CA’),
('zip',94541),('country','USA')])
kvpairs.keys().collect()
# ['city', 'state', 'zip', 'country']
```

- RDD.values(): retorna un RDD con los valores de un par clave-valor o el segundo elemento de cada tupla en un pair RDD.

```
kvpairs = sc.parallelize([('city','Hayward'),('state','CA’),
('zip',94541),('country','USA')])
kvpairs.values().collect()
# ['Hayward', 'CA', 94541, 'USA']
```

Apache Spark: operaciones sobre clave-valor

- Transformaciones funcionales: trabajan sobre la clave o valor.
- RDD.keyBy(<función>): crea tuplas clave-valor aplicando la función sobre los elementos en un RDD. El valor es la tupla de la cual se derivó la clave.
- Ejemplo: tuplas con el esquema ciudad, país, código. Para crear un pair RDD con el código como clave
  - Primer elemento: clave; segundo elemento: tupla con todos los campos

```
locations = sc.parallelize([('Hayward', 'USA', 1), ('Baumholder','Germany',
2),('Alexandria','USA', 3),('Melbourne','Australia', 4)])
bycode = locations.keyBy(lambda x: x[2])
bycode.collect()
# [(1, ('Hayward', 'USA', 1)), (2, ('Baumholder', 'Germany', 2)),
  (3, ('Alexandria', 'USA', 3)), (4, ('Melbourne', 'Australia', 4))]
```

Apache Spark: operaciones sobre clave-valor

- RDD. mapValues(<función>): aplica la función sobre los valores del pair RDD sin cambiar las claves.
  - Retorna un elemento de salida por cada elemento de entrada. La partición del RDD no cambia.
- RDD. flatMapValues(<función>): aplica la función a los valores del pair RDD y crea una lista plana.
  - Retorna cero o más elementos de salida por cada elemento de entrada. La partición del RDD no cambia.
- Ejemplo: Archivo de texto con temperaturas

```
Hayward,71|69|71|71|72
Baumholder,46|42|40|37|39
Alexandria,50|48|51|53|44
Melbourne,88|101|85|77|74
```

Apache Spark: operaciones sobre clave-valor

```
locwtemps = sc.parallelize(['Hayward,71|69|71|71|72',
'Baumholder,46|42|40|37|39’,'Alexandria,50|48|51|53|44',
'Melbourne,88|101|85|77|74’])
```

Carga datos en un RDD

```
kvpairs = locwtemps.map(lambda x: x.split(','))
# [[u'Hayward', u'71|69|71|71|72'],[u'Baumholder',
u'46|42|40|37|39’], [u'Alexandria', u'50|48|51|53|44’],
[u'Melbourne’,u'88|101|85|77|74']]
```

Crea pair RDD, strings separados por “,”

```
locwtemplist = kvpairs.mapValues(lambda x: x.split('|')) \
.mapValues(lambda x: [int(s) for s in x])
# [(u'Hayward', [71, 69, 71, 71, 72]),
(u'Baumholder', [46, 42, 40, 37, 39]),
(u'Alexandria', [50, 48, 51, 53, 44]),
(u'Melbourne', [88, 101, 85, 77, 74])]
```

Crea pair RDD [ciudad, lista de temperaturas]

```
locwtemps = kvpairs.flatMapValues(lambda x: x.split('|')) \
.map(lambda x: (x[0], int(x[1])))
# [(u'Hayward', 71), (u'Hayward', 69),
(u'Hayward', 71), (u'Hayward', 71),
(u'Hayward', 72)]
```

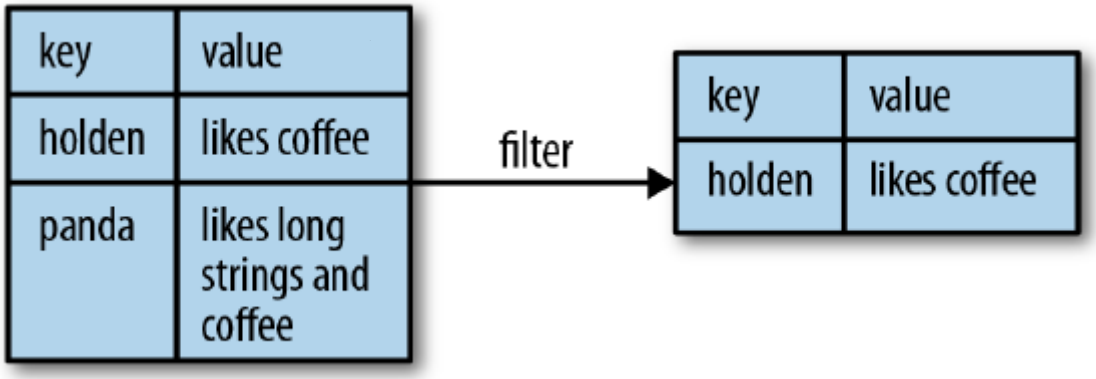
Crea pair RDD [ciudad, temperatura] para cada temperatura de la ciudad

– mapValues crea un elemento por ciudad, flatMapValues crea cinco

Apache Spark: operaciones sobre clave-valor

- Pair RDDs son RDDs: se aplican las funciones habituales de RDDs.

```
result = pairs.filter(lambda keyValue:len(keyValue[1])<20)   (Python)
pairs.filter{case (key, value) => value.length < 20}         (Scala)
```



- Para acceder solamente a los valores de un pair RDD se utiliza la función mapValues(función), que corresponde a aplicar map{case (x, y): (x, func(y))}.

Agregaciones sobre pares clave-valor

- RDD.groupByKey(numPartitions, partitionFunc): agrupa los valores para cada clave en un pair RDD en una secuencia.
  - numPartitions: particiones (grupos) a crear (por defecto, el valor es spark.default.parallelism).
  - Las particiones se crean usando la función partitionFunc.
  - Ejemplo: calcular la temperatura promedio por ciudad. Agrupar por ciudad y calcular el promedio.

```
# locwtemps = [('Hayward', 71), ('Hayward', 69), ('Hayward', 71),...]
grouped = locwtemps.groupByKey()
# [('Melbourne', <pyspark.resultiterable.ResultIterable
object...>),...]
avgtemps = grouped.mapValues(lambda x: sum(x)/len(x))
# [('Melbourne',85),('Baumholder',40),('Alexandria',49),
('Hayward',70)]
```

Agregaciones sobre pares clave-valor

- groupByKey retorna un objeto iterable (ResultIterable) con los valores agrupados. Un objeto iterable (en Python) es una secuencia que puede ser recorrida con un ciclo: (K, V) → [K, Iterable[V]].
- Muchas funciones en Python aceptan iterables como entrada (como sum y len, usadas en el ejemplo).

- Cuando se agrupan valores solamente para agregación, es más eficiente utilizar reduceByKey o foldByKey en lugar de groupByKey, porque los resultados de la función de agregación se combinan antes del shuffle, que involucra menos datos.
  - rdd.reduceByKey(func) produce el mismo RDD que rdd.groupByKey().mapValues(value => value.reduce(func)), pero es más eficiente, porque evita crear una lista de valores para cada clave.

Agregaciones sobre pares clave-valor

- RDD.reduceByKey(<función>, [numPartitions], [partitionFunc]): combina los valores del RDD usando una función asociativa (v_n, $v_{n+1} \rightarrow v_{RESULT}$)

```
x = sc.parallelize([("a", 1), ("b", 1), ("a", 1), ("a", 1), ..., ("b", 1),
("b", 1), ("b", 1), ("b", 1)], 3)
```

  - Con una función lambda

```
y = x.reduceByKey(lambda accum, n: accum + n)
y.collect()
# [('b', 5), ('a', 3)]
```

  - Con una función nominada

```
def sumFunc(accum, n):
    return accum + n
y = x.reduceByKey(sumFunc)
y.collect()
# [('b', 5), ('a', 3)]
```

Agregaciones sobre pares clave-valor

- RDD.reduceByKey(<función>, [numPartitions], [partitionFunc]): combina los valores del RDD usando una función asociativa (v_n, $v_{n+1} \rightarrow v_{RESULT}$)
  - Promedio: función no asociativa.
  - Crear tuplas con las sumas y el contador de elementos para cada clave (ambas son operaciones asociativas y conmutativas) y luego calcular el promedio.

```
locwtemps = [('Hayward', 71), ('Hayward', 69), ('Hayward', 71), ...]
temptups = locwtemps.mapValues(lambda x: (x, 1))
# [('Hayward', (71, 1)), ('Hayward', (69, 1)), ('Hayward', (71,1)), ...]
inputstoavg = temptups.reduceByKey(lambda x, y: (x[0]+y[0], x[1]+y[1]))
# [('Melbourne', (425, 5)), ('Baumholder', (204, 5)), ...]
averages = inputstoavg.map(lambda x: (x[0], x[1][0]/x[1][1]))
# [('Melbourne',85),('Baumholder',40),('Alexandria',49),('Hayward',70)]
```

Agregaciones sobre pares clave-valor

- reduceByKey() es similar a reduce() utilizado con un combiner.
  - Ejecuta multiples reducciones en paralelo, una para cada clave del dataset.
  - El combiner combina elementos localmente en cada worker.
  - Se aplica shuffle y la operación final se realiza en un worker.
  - Combiner y reducer usan la misma función (asociativa y conmutativa). En el ejemplo: sumar lista de sumas en lugar de sumar muchos valores.
  - Menos datos se envían en la faase de shuffle, reduceByKey con una función de suma es más eficiente que groupByKey seguido de una suma.
- Como los datasets pueden tener muchas claves, reduceByKey() no se implementa como una acción que retorna un valor, sino que retorna un nuevo RDD con cada clave y el valor reducido para esa clave.

Agregaciones sobre pares clave-valor

- Ejemplo: Calcular promedios contabilizando suma y número de ocurrencias usando reduceByKey() y mapValues() de modo similar a como se calcula el promedio para todo el RDD usando fold() y map():

  - En Python

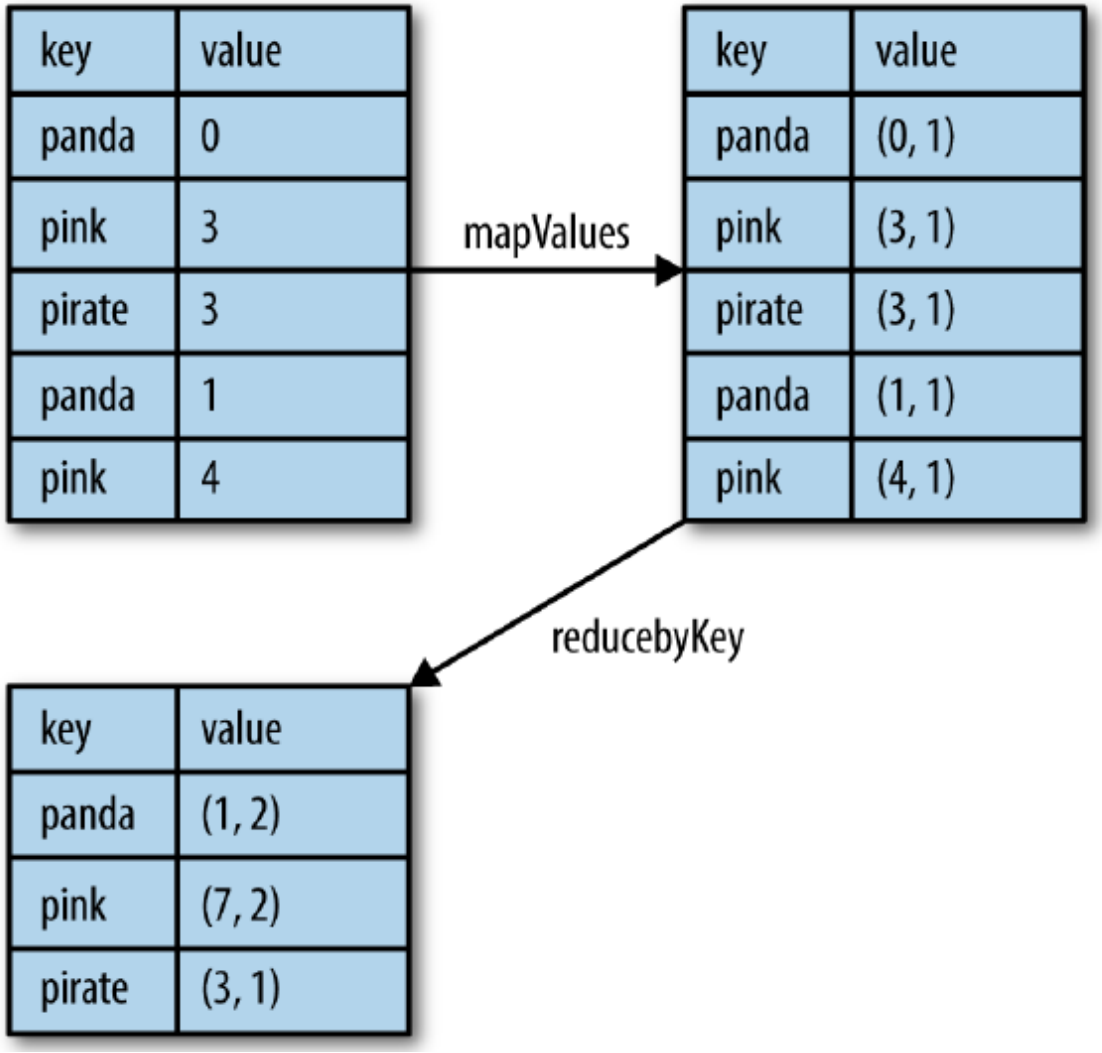
```
rdd.mapValues(lambda x:(x,1).reduceByKey(lambda
        x,y:(x[0]+y[0],x[1]+ y[1]))
```

  - En Scala

```
rdd.mapValues(x=>(x,1)).reduceByKey
        ((x,y)=>(x._1+y._1,x._2+y._2))
```

Agregaciones sobre pares clave-valor

- Ejemplo



Agregaciones sobre pares clave-valor

- RDD.foldByKey(zeroValue, <función>, [numPartitions], [partitionFunc])
  - Similar a fold()): utiliza un valor cero (del mismo tipo de dato del RDD) al aplicar la función de combinación.
- Ejemplo:
  - Hallar el máximo valor por clave:

```
# locwtemps = [('Hayward', 71), ('Hayward', 69), ('Hayward', 71),...]
maxbycity = locwtemps.foldByKey(0, lambda x, y: x if x > y else y)
maxbycity.collect()
# [('Melbourne',101),('Baumholder',46),('Alexandria',53),('Hayward',72)]
```

- RDD.aggregateByKey() es similar, pero genera un valor de diferente tipo.

Agregaciones sobre pares clave-valor

- RDD.sortByKey(ascending=True, [numPartitions],[keyfun]): ordena un pair RDDde acuerdo al tipo de la clave (numérico, string, etc.)
  - A diferencia de sort(), que requiere que se identifique la clave por la cual se ordenará, sortByKey ya conoce la clave.
  - Ascending indica el orden (por defecto es true); keyfunc permite derivar una clave diferente a partir de la predefinida, e.g., keyfunc=lambda k:k.lower().
- Ordenamiento basado en clave (nombre de ciudad, orden alfabético):

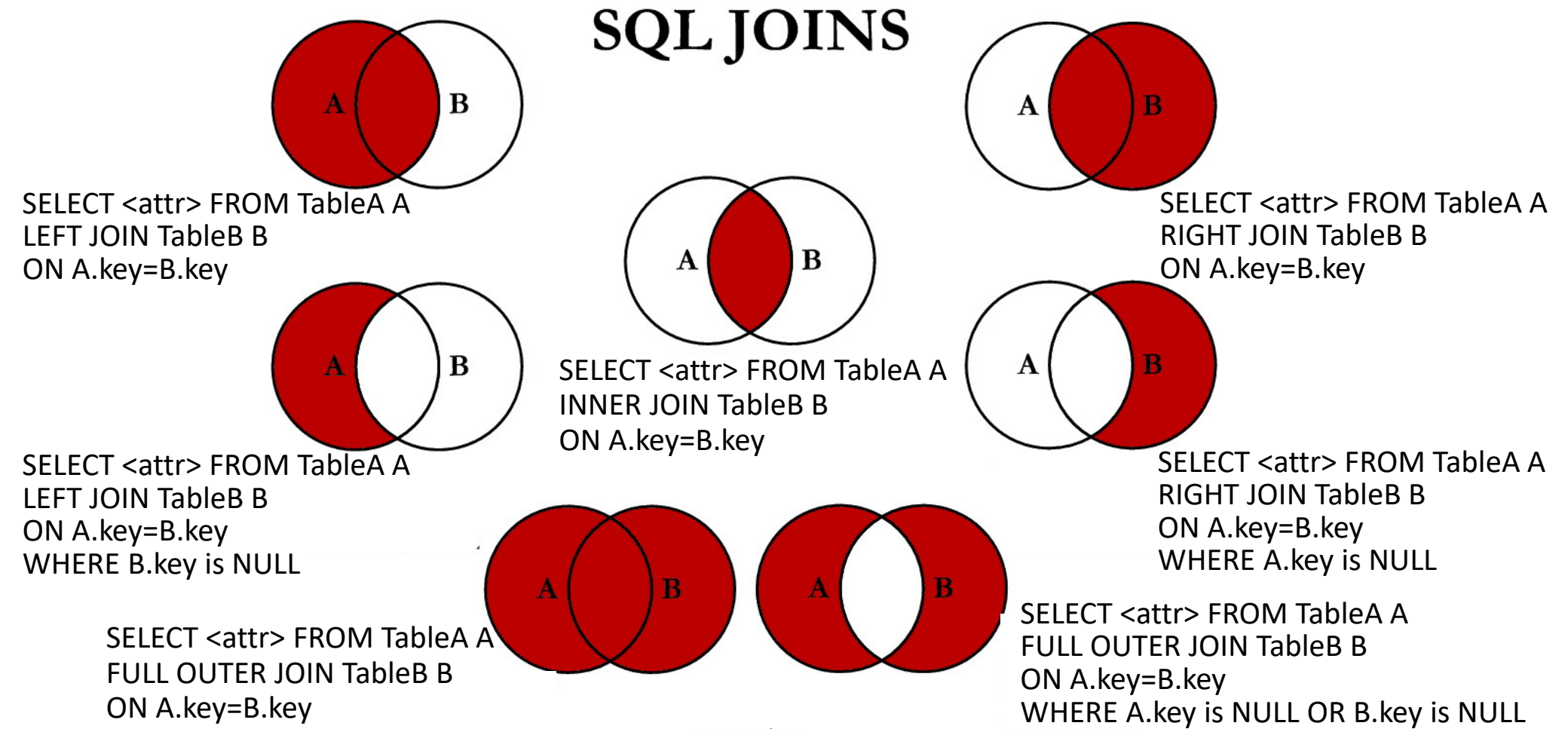
```
# locwtemps = [('Hayward', 71), ('Hayward', 69), ('Hayward', 71), ...]
sortedbykey = locwtemps.sortByKey()
# [('Alexandria', 50), ('Alexandria', 48), ('Alexandria', 51), ...]
```

- Invertir clave y valor para que temperatura sea clave y listarlas en orden numérico descendiente:

```
sortedbyval = locwtemps.map(lambda x: (x[1],x[0])).sortByKey(ascending=False)
# [(101, 'Melbourne'), (88, 'Melbourne'), (85, 'Melbourne'), ...]
```

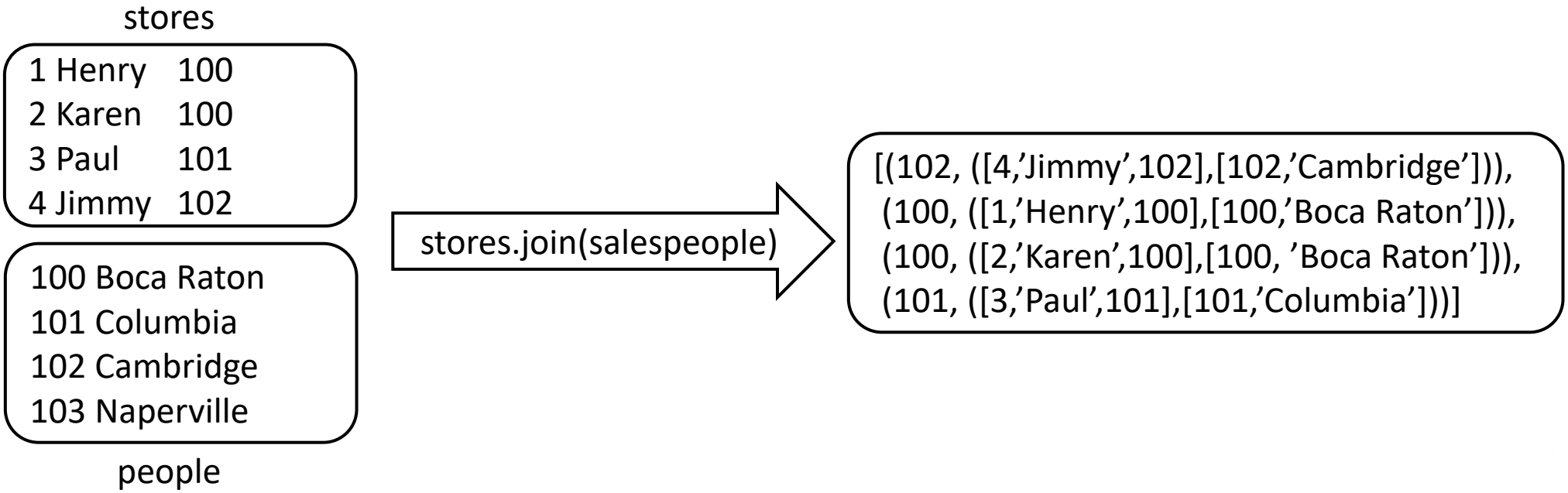
Operaciones de (seudo)conjuntos

- Join: combinar registros de RDD por una clave común.
- Similar a joins de SQL.



Operaciones de (seudo)conjuntos

- RDD.join(<otroRDD>, [numPartitions]): transformación que implementa el inner join. Por defecto usa numPartitions = spark.default.parallelism
- Retorna un RDD que contiene la clave común y un valor que es una tuple con los registros correspondientes de ambos RDDs como un objeto lista.
  - A diferencia de joins SQL, que retornan una lista de columnas de las dos tablas.
- Ejemplos: sobre lista de vendedores y tiendas.



Operaciones de (seudo)conjuntos

- Join: ejemplo en Spark

```
stores = sc.parallelize(['100\tBoca Raton','101\tColumbia','102\tCambridge',
'103\tNaperville']).map(lambda x: x.split('\t')).keyBy(lambda x: x[0])
people = sc.parallelize(['1\tHenry\t100','2\tKaren\t100', '3\tPaul\t101',
'4\tJimmy\t102','5\tJanice\t']).map(lambda x:x.split('\t'))\
.keyBy(lambda x: x[2])
people.join(stores).collect()
#[('102', (['4', 'Jimmy', '102'], ['102', 'Cambridge'])),
# ('100', (['1', 'Henry', '100'], ['100', 'Boca Raton'])),
# ('100', (['2', 'Karen', '100'], ['100', 'Boca Raton'])),
# ('101', (['3', 'Paul', '101'], ['101', 'Columbia']))]
```

  - El RDD resultado contiene información duplicada.
  - Usualmente se aplica una transformación map para eliminar campos o proyectar solamente aquellos campos requeridos para el procesamiento.

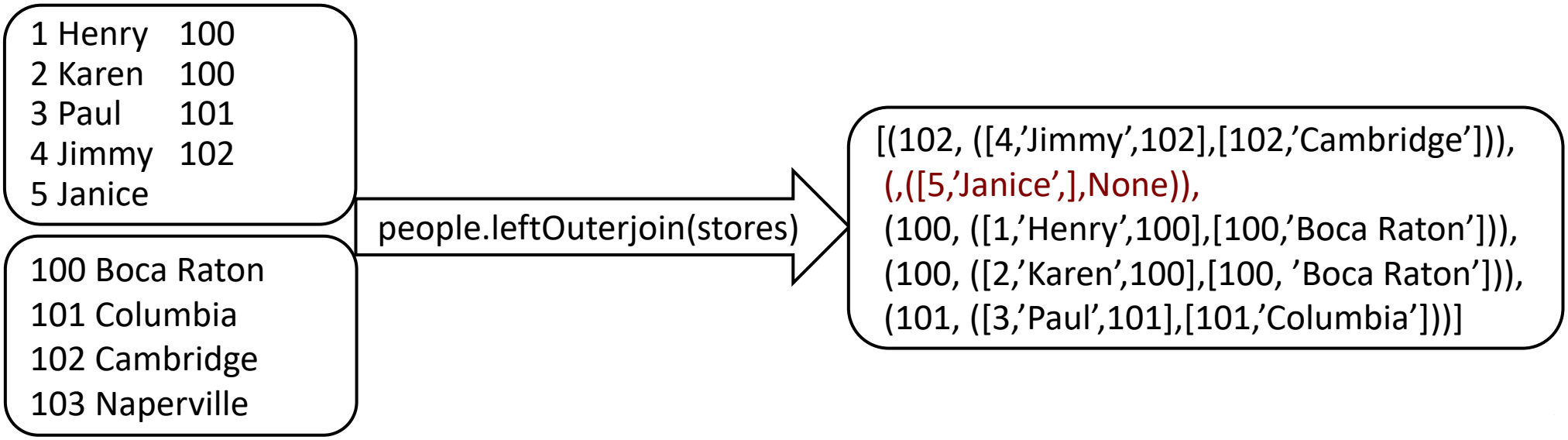
Recordar: join desde la tabla mayor a la tabla menor !

Maximiza paralelismo

Minimiza transferencia de datos

Operaciones de (seudo)conjuntos

- LeftOuterJoin: registros en el RDD (left) y si la clave está en el RDD (right) el registro correspondiente o sinó vacío.
- Ejemplo en Spark: vendedores sin tiendas




```
people.leftOuterJoin(stores).filter(lambda x: x[1][1] is None) \
.map(lambda x: "salesperson " + x[1][0][1] + " has no store").collect()
# ['salesperson Janice has no store']
```

Operaciones de (seudo)conjuntos

- RDD.cogroup(<otroRDD>, [partitions]): agrupa múltiples RDDs por clave.
- Conceptualmente similar a fullOuterJoin, con algunas diferencias:
  - Retorna un objeto iterable (fullOuterJoin crea salidas separadas con elementos para cada clave).
  - cogroup puede agrupar tres o más RDDs (usando la API de Scala o groupWith).
- El resultado de A.cogroup(B) con clave K es [K, Iterable(K,VA, ...), Iterable(K,VB, ...)]. Si un RDD no tiene elementos para una clave que está en el otro RDD, el iterable es vacío.

```
people.cogroup(stores).collect()
# [(102,(<pyspark.resultiterable.ResultIterable object...>,
# <pyspark.resultiterable.ResultIterable object...>)),
# (,(<pyspark.resultiterable.ResultIterable object...>,
# <pyspark.resultiterable.ResultIterable object...>)), ...]
people.cogroup(stores).mapValues(lambda x: [item for sublist in x for item
in sublist]).collect()
# [(102, [[4,'Jimmy',102], [102,'Cambridge']]), (, [[5,'Janice',]]), ...]
```

Operaciones sobre RDDs numéricos

- RDDs numéricos: contienen valores numéricos únicamente.
  - Utilizados para análisis numéricos y estadísticos.
- RDD.count(): acción que retorna el número de elementos del RDD

```
numbers = sc.parallelize([0,1,1,2,3,5,8,13,21,34])
numbers.count()
# 10
```

- RDD.min([key]) y RDD.max([key]): acciones que retornan el valor mínimo/máximo. Keyf es una función que se utiliza para la comparación (en caso que se provea)

```
numbers.min()
# 0
```

```
numbers.max()
# 34
```

RDD.mean(): calcula la media aritmética

```
numbers.mean()
# 8.8
```

RDD.sum(): suma valores de un RDD numérico

```
numbers.sum()
# 88
```

Operaciones sobre RDDs numéricos

- RDD.stdev(): acción que calcula la desviación estándar de un RDD numérico

```
numbers = sc.parallelize([0,1,1,2,3,5,8,13,21,34])
numbers.stdev()
# 10.467091286503619
```

- RDD.variance(): calcula la varianza

```
numbers.variance()
# 109.55999999999999
```

- RDD.stats(): retorna un objeto StatCounter, una estructura que contiene el número de elementos, la media, la desviación estándar, el máximo y el mínimo en una única operación.

```
numbers.stats()
# (count: 10, mean: 8.8, stdev: 10.4670912865, max: 34.0, min: 0.0)
```

- RDD.sampleVariance()/RDD.sampleStdev(): varianza/desviación estándar calculadas sobre una muestra

```
numbers.sampleVariance()
# 121.7333333333333
```

Spark: particionamiento

- Controlar particionamiento para minimizar comunicaciones costosas.
- Similar a elegir la estructura de datos correcta en programación secuencial.
- Particionamiento no es útil si un RDD se utilizará solo una vez, es útil cuando se reutiliza varias veces (por ejemplo, en joins).
- Spark no permite control explícito sobre a qué nodo irán pares con una determinada clave (para proveer tolerancia a fallos), pero si permite asegurar que los pares con una misma clave irán al mismo nodo.
- Ejemplo: hash-particionar un RDD en 100 particiones para que las claves que tienen el mismo hash (modulo 100) vayan al mismo nodo.

Agregaciones sobre pares clave-valor

- Ejemplo: aplicación que maneja info de usuarios en memoria (pares (UserID, UserInfo [tópicos a los que se suscribió el Usuario])).
  - La aplicación periódicamente combina la infomración con un archivo más pequeño que registra eventos que ocurrieron en los últimos 5 minutos (pares (UserID, LinkInfo) de los sitios visitados).
- Contar usuarios que visitaron un link que NO forma parte de sus suscripciones (en Scala):
  - join() para agrupar pares UserInfo y LinkInfo para cada clave UserID

```
// Carga info de Hadoop SequenceFile en HDFS. Distribuye los elementos
de acuerdo a los bloques HDFS. Spark no conoce en qué partición se
encuentra un UserID particular.
val sc = new SparkContext(...)
val userData = sc.sequenceFile[UserID, UserInfo]("hdfs://...").persist()

[Continúa]
```

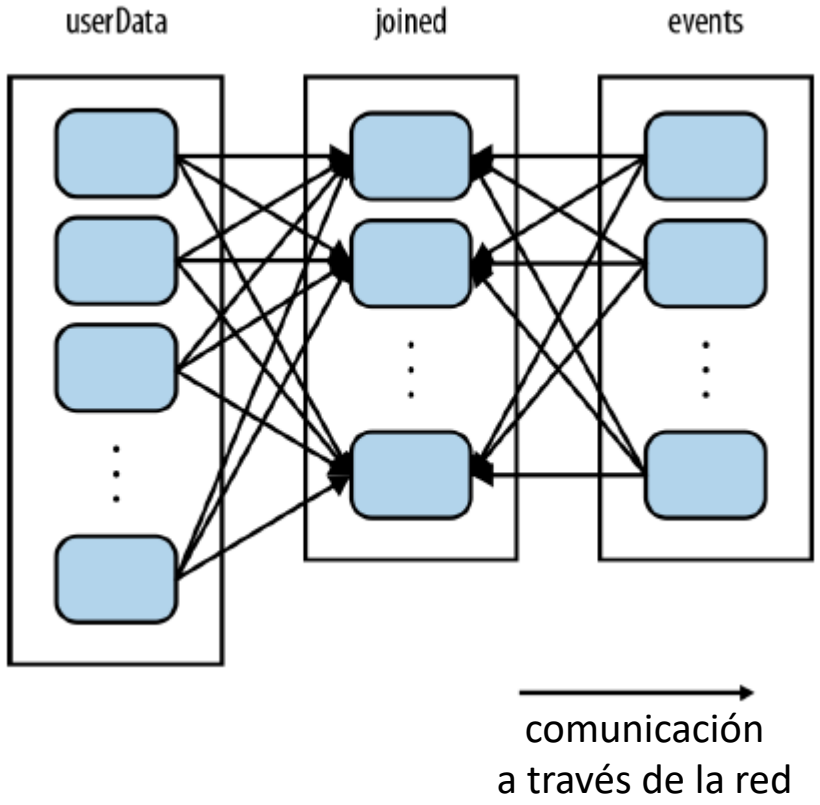
Agregaciones sobre pares clave-valor

- Contar usuarios que visitaron un link que NO forma parte de sus suscripciones:
  - join() para agrupar pares UserInfo y LinkInfo para cada clave UserID

```
[Continuación]
def processNewLogs(logFileName: String) {
// Periódicamente procesa el archivo de eventos (SequenceFile) de pares
(UserID, LinkInfo)
val events = sc.sequenceFile[UserID, LinkInfo](logFileName)
val joined = userData.join(events) // RDD de pares (UserID, (UserInfo,
LinkInfo))
val offTopicVisits = joined.filter {
   case (userId, (userInfo, linkInfo)) => // Expande la tupla en sus
   componentes
   !userInfo.topics.contains(linkInfo.topic)
}.count()
println(“Visitas a tópicos no suscritos: " + offTopicVisits)
}
```

Particionamiento

- El código previo es ineficiente
- La operación join(), utilizada cada vez que se invoca processNewLogs() no sabe cómo se particionan las claves en los datasets.
  - Por defecto, join debe hashear todas las claves de ambos datasets, enviando elementos con la misma clave a través de la red a una misma máquina y luego combinándolos en esa máquina.
  - Como la tabla de datos de usuarios es mucho más larga que el archivo de eventos, se desperdicia trabajo en hashing y shuffling de la tabla de datos a través de la red en cada invocación, aunque la tabla varíe muy poco (o nada).



Particionamiento

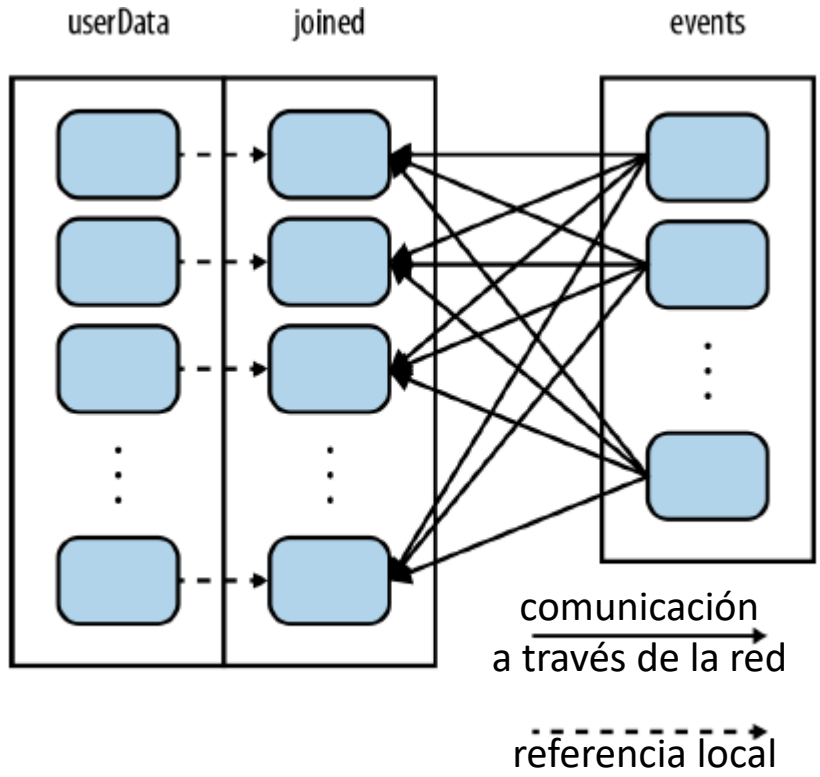
- Solución: particionar userData

```
val sc = new SparkContext(...) // Crear 100 particiones y persistir
val userData = sc.sequenceFile[UserID, UserInfo]("hdfs://...")
.partitionBy(new HashPartitioner(100)).persist()
```

– processNewLogs() no cambia. eventos es local a processNewLogs() y se usa solo una vez: no hay ventaja en particionarlo.

– Al invocar partitionBy() al construir userData, Spark conoce que está hash-particionado y al invocar a join() aprovecha esta información

– Al invocar userData.join(events), Spark solo hace shuffle del RDD de eventos y envía eventos con cada UserID particular a la máquina que contiene la partición correspondiente de userData.



Se reducen las comunicaciones y se mejora el desempeño

Particionamiento

- partitionBy() es una transformación, siempre retorna un *nuevo* RDD.
- Recordar: los RDDs no se modifican
- Es importante persistir el resultado de partitionBy() y no el sequenceFile() original.
- El parámetro de partitionBy() [100 en el ejemplo] es el número de particiones, controla cuántas tareas paralelas realizan operaciones futuras en el RDD.
  - Sugerencia: el número de cores en el cluster.

- En Python no se pasa un objeto Hash Partitioner como parámetro, simplemente se pasa el número de particiones: rdd.partitionBy(100).

Particionamiento

- Si el RDD creado con partitionBy() no se persiste, cada vez que se use se repetirá el particionamiento (ineficiente) y se reevaluará su lineage.
- Otras operaciones de Spark heredan la información de particionamiento:
  - sortByKey() y groupByKey() producen RDDs range/hash-partitioned.
  - Operaciones como map() causan que el nuevo RDD 'olvide' el particionamiento del padre, porque potencialmente pueden modificar la clave de cada registro.
- Todas las operaciones que que involucran shuffling de datos sobre la red se benefician del particionamiento: cogroup(), groupWith(), join() y sus variantes, groupByKey(), reduceByKey(), combineByKey() y lookup().
- Operaciones que actúan sobre un único RDD (e.g. reduceByKey()) cuando trabajan sobre un preparticionado causan que los valores para cada clave se computen *localmente* en un único host. Solo el valor final de la reducción local se envía del worker al master.

Particionamiento

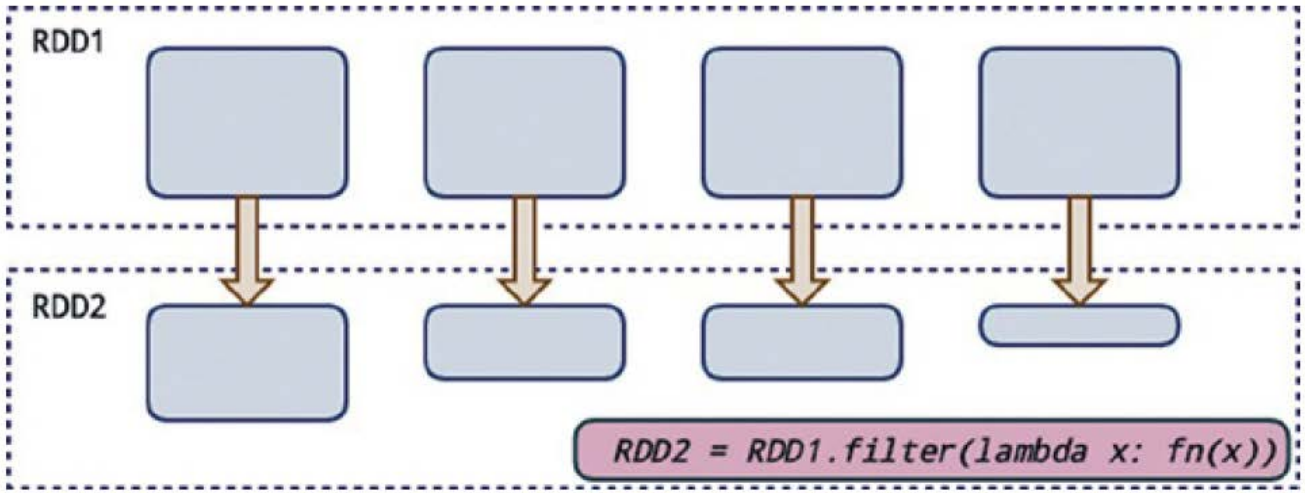
- Operaciones que actúan sobre dos RDDs (e.g., cogroup(), join()):
  - al menos para uno de los RDDs (el que tiene particionamiento conocido) no se realiza shuffle.
  - Si ambos RDDs tienen *el mismo* particionamiento y están cacheados en la misma máquina (e.g., uno se creó usando mapValues() sobre el otro, que preserva claves y particionamiento) no se realiza shuffle.
  - Si uno de los RDDs no ha sido computado, no se realiza shuffle.

Particionamiento

- Spark aprovecha la información de particionamiento.
  - Ejemplo: join() sobre dos RDDs, los elementos con la misma clave se hashean en el mismo host. Spark sabe que el resultado está hash-particionado. reduceByKey() sobre el resultado del join será significativamente más rápido.
  - Para transformaciones que no garantizan un particionamiento, el RDD no se considera particionado (e.g., map() sobre un pair RDD particionado). Spark no analiza las operaciones para determinar si retienen la clave.
  - Opciones: usar mapValues() o flatMapValues(), que no cambian la clave.
- Operaciones que retornan RDDs particionados:
  - cogroup, groupWith, groupByKey, reduceByKey, sort, partitionBy(), joins y si el RDD padre tiene particionamiento: mapValues, flatMapValues() y filter()
- Operaciones binarias: i) operandos sin particiones: particionamiento hash, con tantas particiones como niveles de paralelismo admita la operación; ii) un operando con partición: se hereda al hijo; iii) dos operandos con particiones: se hereda el particionamiento del primer padre.

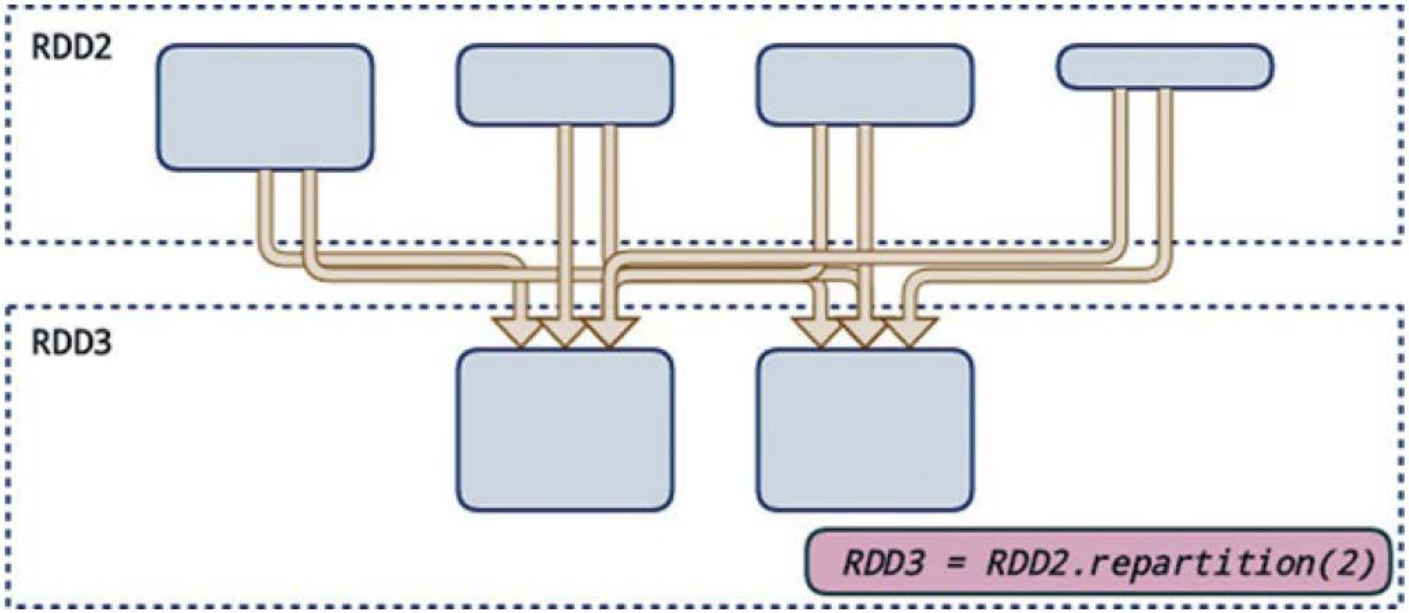
Controlando el particionamiento

- Dilema: cuántas particiones son efectivas ?
  - Pocas particiones grandes: se limita el paralelismo, pueden existir problemas de memoria en los hosts (executors).
  - Muchas particiones pequeñas: muchas tareas para entradas triviales.
  - Particiones pequeñas y grandes: ejecuciones especulativas innecesarias.
- Ejemplo: filter() crea una nueva partición por cada partición de entrada con los registros que cumplen la condición, afectando el desempeño de operaciones posteriores



Controlando el particionamiento

- Solución: reparticionar



- Determinar el número óptimo de particiones implica realizar análisis empíricos: encontrar el punto en el cual utilizar particiones adicionales comienza a degrader la eficiencia.
- Regla práctica: iniciar el análisis con dos veces el número de cores disponibles en el cluster (en todos los worker nodes).

Particionamiento avanzado

- RDD.repartitionAndSortWithinPartitions(numPartitions,partitionFunc, ascending,keyfunc)
  - Reparticiona el RDD en numPartitions de acuerdo a la función partitionFunc y los registros de cada partición se ordenan por sus claves, definidas por keyfunc, en orden ascendente.
  - Ejemplo: ordenamiento secundario de un pair RDD con una clave compuesta.
  - La primer parte de la clave se agrupa en particiones separadas y la segunda parte se ordena descendiente (glom es una función para inspeccionar particiones, la veremos a continuación).

```
kvrdd = sc.parallelize([((1,99),'A'),((1,101),'B’), ((2,99),'C’),
((2,101),'D')],2)
kvrdd.glom().collect()
[ [((1, 99), 'A'), ((1, 101), 'B')],[((2, 99), 'C'), ((2, 101), 'D')]]
kvrdd2 = kvrdd.repartitionAndSortWithinPartitions(numPartitions=2,
ascending=False, keyfunc=lambda x: x[1])
kvrdd2.glom().collect()
[ [((1, 101), 'B'), ((1, 99), 'A’)], [((2, 101), 'D'), ((2, 99), 'C')]]
```

Particionamiento avanzado

- RDD.repartitionAndSortWithinPartitions(numPartitions,partitionFunc, ascending,keyfunc)

```
pairs  = sc.parallelize([["a",1], ["b",2], ["c",3], ["d",3]])
pairs.collect()
# [['a', 1], ['b', 2], ['c', 3], ['d', 3]]
pairs.repartitionAndSortWithinPartitions(2).glom().collect()
# [[('a', 1), ('c', 3)], [('b', 2), ('d', 3)]]
```

  - Los datos se reordenan en dos particiones ('a' y 'c' en una y 'b' y 'd' en la otra), y las claves se ordenan
  - Reparticionamiento basado en una condición

```
pairs.repartitionAndSortWithinPartitions(2, \
        partitionFunc=lambda x: x == 'a').glom().collect()
# [[('b', 2), ('c', 3), ('d', 3)], [('a', 1)]]
```

  - Dos particiones, una con tres pares (ordenados por clave) y una con ('a',1).

Particionamiento avanzado: operaciones

- Específicos: consideran a las particiones como unidades atómicas.
- RDD.foreachPartition(func): aplica func a cada partición de un RDD

```
def f(x):
    for rec in x:
        print(rec)

kvrdd2.foreachPartition(f)
((1, 101), 'B')
((1, 99), 'A')
...
((2, 101), 'D')
((2, 99), 'C')
...
```

- foreachPartition es una ación, no una trasformación: desencadena la evaluación de todo el lineage del RDD.
- Los datos se envían al driver, su volumen puede afectar el desempeño.

Particionamiento avanzado: operaciones

- RDD.glom(): retorna un RDD que agrupa los elementos de cada partición en una lista. Útil para inspeccionar particiones de RDDs.
- RDD.lookup(key): retorna la lista de valores para la clave, utiliza el particionamiento para la búsqueda (solamente en las particiones donde la clave puede estar presente).

```
kvrdd = sc.parallelize([(1,'A'),(1,'B'),(2,'C'),(2,'D')],2)
kvrdd.lookup(1)
['A', 'B']
```

- RDD.mapPartitions(función, preservesPartitioning=False): aplica la función a cada partición de un RDD.
- Ejemplo: invertir clave y valor en cada partición

```
kvrdd = sc.parallelize([(1,'A'),(1,'B'),(2,'C'),(2,'D')],2)
def f(iterator): yield [(b, a) for (a, b) in iterator]

kvrdd.mapPartitions(f).collect()
[[('A', 1), ('B', 1)], [('C', 2), ('D', 2)]]
```