

Centrale Digital Lab, Université Paris-Saclay, LISN, Grand Est, Région Île de France

Big Data

Spark Programming

Stéphane Vialle & Gianluca Quercini

1

Spark Programming

1. Main objectives
2. RDD concepts
3. Operations on generic RDDs
4. Operations on RDD of *key-value* pairs

2

Spark main objectives

Spark has been designed:

- To efficiently run iterative and interactive applications
→ keeping data in-memory between operations
- To provide a low-cost fault tolerance mechanism
→ low overhead during safe executions
→ fast recovery after failure
- To be easy and fast to use in interactive environment
→ using compact *Scala* programming language
- To be « scalable »
→ able to efficiently process bigger data on larger computing clusters

3

Spark main objectives

Spark Framework

- SQL (high level)
- Stream processing

Programming	Scala	Python	R	Java	Tools
Library	Spark SQL Dataframe	ML Lib	GraphX	Streaming	
Engine	Spark Core				
Management	YARN	Mesos	Spark Scheduler		
Storage	Local	HDFS	S3	RDBMS	NoSQL

- RDD (low level)
- Transformations & Actions (Map-Reduce)
- Fault-Tolerance
- ...

Maestros: The Data Science Ecosystem

Spark design started in 2009, with the PhD thesis of Matei Zaharia at Berkeley Univ. Matei Zaharia co-founded DataBricks in 2013.

4

Spark main objectives

An essential difference between Spark and Hadoop: the speed!

Spark

Spark job: Data flows from HDFS (Hard disk) through multiple steps, each using RAM (Memory). A 'Faster!' callout points to this path.

Hadoop

Hadoop job: Data flows from HDFS (Hard disk) through multiple steps, each using HDFS (Hard disk).

5

Spark Programming

1. Main objectives
2. RDD concepts
3. Operations on generic RDDs
4. Operations on RDD of *key-value* pairs

6

RDD concepts and operations

A RDD (Resilient Distributed Dataset) is:

- an **immutable** (read only) dataset
- a **partitioned** dataset
- usually stored in a distributed file system (like HDFS)

When reading a HDFS file:

```
rdd1 = sc.parallelize(« myFile.txt »)
```

- Read each HDFS block
- Spread the blocks in memory of different **Spark Executor processes** (on ≠ nodes) ⇒ **Get a RDD**

When writing a HDFS file:

- one RDD → One HDFS file
- one RDD partition block → One HDFS file block
- each RDD partition block is replicated by HDFS

7

RDD concepts and operations

Example of a 4-blocks partition stored on 2 data nodes (no replication)

Source: <http://images.backtobasics.com/>

8

RDD concepts and operations

Initial input RDDs:

- are usually created from distributed files (like HDFS files),
- Spark processes read the file blocks that become **in-memory RDD**

Operations on RDDs:

- Transformations**: read RDDs, compute, and generate a new RDD
- Actions**: read RDDs and generate results **out** of the RDD world

Map and Reduce are parts of the operations

Source: [Stack Overflow](#)

9

RDD concepts and operations

Exemple of Transformations and Actions

Transformations	<code>map(f: T => U)</code>	: RDD[T] => RDD[U]
	<code>filter(f: T => Bool)</code>	: RDD[T] => RDD[T]
	<code>flatMap(f: T => Seq[U])</code>	: RDD[T] => RDD[U]
	<code>sample(fraction: Float)</code>	: RDD[T] => RDD[T] (Deterministic sampling)
	<code>groupByKey()</code>	: RDD[(K, V)] => RDD[(K, Seq[V])]
	<code>reduceByKey(f: (V, V) => V)</code>	: RDD[(K, V)] => RDD[(K, V)]
	<code>union()</code>	: (RDD[T], RDD[T]) => RDD[T]
	<code>join()</code>	: (RDD[(K, V)], RDD[(K, W)]) => RDD[(K, (V, W))]
	<code>cogroup()</code>	: (RDD[(K, V)], RDD[(K, W)]) => RDD[(K, (Seq[V], Seq[W]))]
	<code>crossProduct()</code>	: (RDD[T], RDD[U]) => RDD[(T, U)]
Actions	<code>mapValues(f: V => W)</code>	: RDD[(K, V)] => RDD[(K, W)] (Preserves partitioning)
	<code>sortBy(c: Comparator[K])</code>	: RDD[(K, V)] => RDD[(K, V)]
	<code>partitionBy(p: Partitioner[K])</code>	: RDD[(K, V)] => RDD[(K, V)]
	<code>count()</code>	: RDD[T] => Long
	<code>collect()</code>	: RDD[T] => Seq[T]
	<code>reduce(f: (T, T) => T)</code>	: RDD[T] => T
	<code>lookup(k: K)</code>	: RDD[(K, V)] => Seq[V] (On hash/range partitioned RDDs)
	<code>save(path: String)</code>	: Outputs RDD to a storage system, e.g., HDFS

Table 2: Transformations and actions available on RDDs in Spark. Seq[T] denotes a sequence of elements of type T.

Source: *Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing*. **Matei Zaharia et al.** Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation. San Jose, CA, USA, **2012**

10

RDD concepts and operations

Exemple of Transformations and Actions

Transformations	<code>map(f: T => U)</code>	: RDD[T] => RDD[U]
	<code>filter(f: T => Bool)</code>	: RDD[T] => RDD[T]
	<code>flatMap(f: T => Seq[U])</code>	: RDD[T] => RDD[U]
	<code>sample(fraction: Float)</code>	: RDD[T] => RDD[T] (Deterministic sampling)
	<code>groupByKey()</code>	: RDD[(K, V)] => RDD[(K, Seq[V])]
	<code>reduceByKey(f: (V, V) => V)</code>	: RDD[(K, V)] => RDD[(K, V)]
	<code>union()</code>	: (RDD[T], RDD[T]) => RDD[T]
	<code>join()</code>	: (RDD[(K, V)], RDD[(K, W)]) => RDD[(K, (V, W))]
	<code>cogroup()</code>	: (RDD[(K, V)], RDD[(K, W)]) => RDD[(K, (Seq[V], Seq[W]))]
	<code>cartesianProduct()</code>	: (RDD[T], RDD[U]) => RDD[(T, U)]
Actions	<code>mapValues(f: V => W)</code>	: RDD[(K, V)] => RDD[(K, W)] (Preserves partitioning)
	<code>sortBy(c: Comparator[K])</code>	: RDD[(K, V)] => RDD[(K, V)]
	<code>partitionBy(p: Partitioner[K])</code>	: RDD[(K, V)] => RDD[(K, V)]
	<code>count()</code>	: RDD[T] => Long
	<code>collect()</code>	: RDD[T] => Seq[T]
	<code>reduce(f: (T, T) => T)</code>	: RDD[T] => T
	<code>lookup(k: K)</code>	: RDD[(K, V)] => Seq[V] (On hash/range partitioned RDDs)
	<code>save(path: String)</code>	: Outputs RDD to a storage system, e.g., HDFS

Table 2: Transformations and actions available on RDDs in Spark. Seq[T] denotes a sequence of elements of type T.

Source: *Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing*. **Matei Zaharia et al.** Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation. San Jose, CA, USA, **2012**

11

Spark Programming

1. Main objectives
2. RDD concepts
3. Operations on generic RDDs
4. Operations on RDD of *key-value* pairs

12

Transformations on generic RDDs

Transformations applied on one RDD: `rdd : {1, 2, 3, 3}`

```
Python: rdd.map(lambda x: x+1) → rdd: {2, 3, 4, 4}
Scala : rdd.map(x => x+1)      → rdd: {2, 3, 4, 4}

Scala : rdd.map(x => x.to(3)) → rdd: {(1,2,3), (2,3), (3), (3)}
Scala : rdd.flatMap(x => x.to(3)) → rdd: {1, 2, 3, 2, 3, 3, 3}

Scala : rdd.filter(x => x != 1) → rdd: {2, 3, 3}
Scala : rdd.distinct()         → rdd: {1, 2, 3}

Some sampling functions exist:
Scala : rdd.sample(false, 0.5) → rdd: {1} or {2,3} or ...
      with replacement = false
```

Sequence of transformations:

```
Scala: rdd.filter(x => x != 1).map(x => x+1) → rdd: {3, 4, 4}
```

13

Transformations on generic RDDs

Transformations applied on two RDDs: `rdd : {1, 2, 3}`
`rdd2 : {3, 4, 5}`

```
Scala : rdd.union(rdd2)      → rdd: {1, 2, 3, 3, 4, 5}
Scala : rdd.intersection(rdd2) → rdd: {3}
Scala : rdd.subtract(rdd2)   → rdd: {1, 2}
Scala : rdd.cartesian(rdd2)  → rdd: {(1,3), (1,4), (1,5),
      (2,3), (2,4), (2,5),
      (3,3), (3,4), (3,5)}
```

14

Actions on generic RDDs

Actions applied on a RDD: `rdd : {1, 2, 3, 3}`

```
Scala : rdd.collect()      → (1, 2, 3, 3)
Scala : rdd.count()        → 4
Scala : rdd.countByValue() → ((1,1), (2,1), (3,2))
Scala : rdd.take(2)        → (1, 2)           the first elts
Scala : rdd.top(2)         → (3, 3)           the higher elts
Scala : rdd.takeOrdered(3, Ordering[Int].reverse) → (3,3,2)
Scala : rdd.takeSample(false, 2) → (?,?)
      takeSample(withReplacement, NbEltToGet, [seed])

Scala : var sum = 0
      rdd.foreach(sum += _) → does not return any value
      println(sum)         → 9
```

15

Actions on generic RDDs

Actions applied on a RDD: `rdd : {1, 2, 3, 3}`

```
Scala : rdd.reduce(...)
```

Ex: computing the sum of the RDD values

```
Python: rdd.reduce(lambda x, y: x+y) → 9
Scala : rdd.reduce((x, y) => x+y)    → 9
```

Result is NOT a RDD

The **reduce** action is applied on 2 operands:
2 input data
or:
1 input data and 1 **reduce** result

It is defined by **only 1 associative function**:
because input and output data types must be **identical**
(will be different with action *aggregate*)

Computations are done in parallel but result is not a RDD

16

Actions on generic RDDs

Actions applied on a RDD: `rdd : {1, 2, 3, 3}`

```
Scala : rdd.reduce(...)
```

Ex: computing the sum of the RDD values

```
Python: rdd.reduce(lambda x, y: x+y) → 9
Scala : rdd.reduce((x, y) => x+y)    → 9
```

Result is NOT a RDD

Specifying the initial value of the accumulator:

```
Scala : rdd.fold(0)((accu, value) => accu+value) → 9
```

Specifying to start to accumulate from Left or from Right:

```
Scala : rdd.foldLeft(0)((accu, value) => accu+value) → 9
Scala : rdd.foldRight(0)((accu, value) => accu+value) → 9
```

17

Actions on generic RDDs

Actions applied on a RDD:

Ex. of « aggregations » to compute an average value

- Specifying the initial value of the accumulator (**0 = sum, 0 = nb**)
- Specifying a function to add a value to an accumulator (inside a rdd partition block)
- Specifying a function to add two accumulators (from two rdd partition blocks)

```
val SumNb = rdd.aggregate((0, 0)) {
  (acc, v) => (acc._1+v, acc._2+1),
  (acc1, acc2) => (acc1._1+acc2._1,
    acc1._2+acc2._2)
```

Use type inference to select the fct to use

- Division of the sum by the nb of values

```
val avg = SumNb._1/SumNb._2.toDouble
```

18

Actions on generic RDDs

Actions applied on a RDD:

Ex. of « aggregations » to compute an average value

```
Python : rdd.aggregate(acc0) ((lambda acc,v: new_acc)
                               (lambda acc1,acc2: new_acc))
Scala  : rdd.aggregate(acc0) ((acc,v) => new_acc,
                              (acc1,acc2) => new_acc)
```

The aggregate action is applied on 2 operands:

- 1 input data and 1 aggregate result
- or:
- 2 aggregate results

And is defined with 2 associative functions
because datatypes of input and aggregated data are different (otherwise: use reduce(...))

19

Spark programming

1. Main objectives
2. RDD concepts
3. Operations on generic RDDs
- 4. Operations on RDD of *key-value* pairs**

20

Operations on RDD of *key-value* pairs

Transformations for one pair RDD: `rdd : {(1, 2), (3, 3), (3, 4)}`

```
Scala : rdd.groupByKey() → rdd: {(1, [2]), (3, [3, 4])}
      Group the values associated to the same key
      Group the values of a same key in the same Spark Executor
      Move all input data → Huge network traffic in shuffle step !!
```

```
Scala : rdd.reduceByKey((x,y) => x+y) → rdd: {(1, 2), (3, 7)}
```

Reduce values associated to the same key

Limited traffic shuffle

When input data type and reduced data type are identical

21

Operations on RDD of *key-value* pairs

Transformations for one pair RDD:

```
Scala : rdd.aggregateByKey(init_acc) (
  ..., // mergeValueAccumulator fct
  ..., // mergeAccumulators fct
)
```

When input data type and reduced data type are different

```
Scala : rdd.combineByKey(
  ..., // createAccumulator fct
  ..., // mergeValueAccumulator fct
  ..., // mergeAccumulators fct
)
```

See further

The real function (used to implement the previous ones)

22

Operations on RDD of *key-value* pairs

Transformations for one pair RDD: `rdd : {(1, 2), (3, 3), (3, 4)}`

```
Scala : rdd.mapValues(x => x+1) → rdd: {(1, 3), (3, 4), (3, 5)}
```

Apply to each value (keys do not change)

```
Scala : rdd.flatMapValues(x => x to 3) → rdd: {(1,2), (1,3), (3,3)}
```

key: 1, 2 to 3 → (2, 3)	→ (1, 2), (1, 3),	} ((1,2), (1,3)), (3,3)
key: 3, 3 to 3 → (3)	→ (3, 3)	
key: 3, 4 to 3 → ()	→ nothing	

Apply to each value (keys do not change) and flatten

23

Operations on RDD of *key-value* pairs

Transformations applying to one pair RDD: `rdd : {(1, 2), (3, 3), (3, 4)}`

```
Scala : rdd.keys() → rdd: {1, 3, 3}
```

Return an RDD containing only the keys

```
Scala : rdd.values() → rdd: {2, 3, 4}
```

Return an RDD containing only the values

```
Scala : rdd.sortByKey() → rdd: {(1, 2), (3, 3), (3, 4)}
```

Return a pair RDD sorted by the keys

24

Operations on RDD of *key-value* pairs

Transformations applying on two *pair RDDs*

```
rdd : {(1, 2), (3, 4), (3, 6)}
rdd2: {(3, 9)}
```

Scala : `rdd.subtractByKey(rdd2)` → `rdd: {(1, 2)}`
Remove pairs with key present in the 2nd pair RDD

Scala : `rdd.join(rdd2)` → `rdd: {(3, (4, 9)), (3, (6, 9))}`
Inner Join between the two pair RDDs

Scala : `rdd.cogroup(rdd2)` → `rdd: {(1, ([2], [])), (3, ([4, 6], [9]))}`
Group data from both RDDs sharing the same key

25

Operations on RDD of *key-value* pairs

Standard transformations applied on a *pair RDD*

```
rdd : {(1, 2), (3, 4), (3, 6)}
```

A pair RDD remains a RDD of tuples (key, values)
 → Classic transformations can be applied

Scala : `rdd.filter(case (k,v) => v < 5)` → `rdd: {(1, 2), (3, 4)}`

Scala : `rdd.map(case (k,v) => (k,v*10))` → `rdd: {(1, 20), (3, 40), (3, 60)}`

26

Operations on RDD of *key-value* pairs

Actions applying on a *pair RDD*

```
rdd : {(1, 2), (3, 4), (3, 6)}
```

Scala : `rdd.countByKey()` → `((1, 1), (3, 2))`
Return a tuple of couple, counting the number of pairs per key

Scala : `rdd.collectAsMap()` → `Map{(1, 2), (3, 4), (3, 6)}`
Return a 'Map' datastructure containing the RDD

Scala : `rdd.lookup(3)` → `[4, 6]`
Return an array containing all values associated with the provided key

27

Quiz

Q1: What does the RDD "r" at the end of the following code contain?

```
words = 'Technology is best when it brings people together'\
      .split(' ')
r = sc.parallelize(words)\
  .filter(lambda x: len(x) >= 3)\
  .map(lambda x: (x[0].lower(), x.lower()))\
  .reduceByKey(lambda w,v: w if len(w)>len(v) else v)
```

28

Quiz

Q2: One or more Spark-Workers work on each step of this code?

```
def f(x):
  if x > 0:
    print(x)

r = sc.parallelize(data)\
  .filter(lambda t: t[0] == 10)\
  .mapValues(lambda v: v*10)\
  .reduceByKey(lambda w,v: w+v)\
  .values()\
  .collect()\
  .foreach(f)
```

29

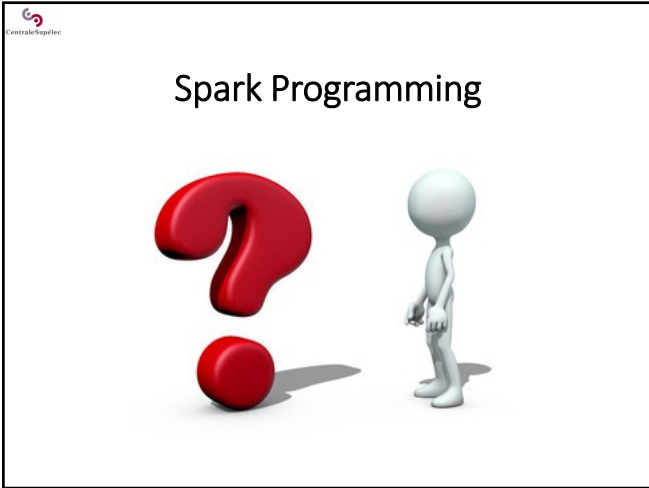
Quiz

Q3: What is the output ?

```
data : {( 'a',(12,1)), ('b',(13,1)), ('a',(9,2)), ('c',(18,4)), ('b',(13,1)), ('b',(15,2))}
```

```
res = sc.parallelize(data)\
  .filter(lambda t: t[0] <= 'z' and t[0] >= 'a') \
  .reduceByKey(lambda w,v: (w[0]+v[0],w[1]+v[1]))\
  .mapValues(lambda v: float(v[0])/float(v[1]))
print(res.collect())
```

30



31