Chapter 1. Introduction to Data Analysis with Spark

- Apache Spark is a cluster computing platform designed to be ***fast*** **and** ***general-****purpose*

- On the **speed side, Spark extends the popular MapReduce model to efficiently support more types of computations, including interactive queries and stream processing.**

- One of the main features Spark offers for speed is the **ability to run computations in memory, but the system is also more efficient than MapReduce for complex applications running on disk.**

- On the generality side, **Spark is designed to cover a wide range of workloads that previously required separate distributed systems, including batch applications, iterative algorithms, interactive queries and streaming.**

- By **supporting these workloads in the same engine, Spark makes it easy and inexpensive to *combine* different processing types, which is often necessary in production data analysis pipelines**.

- In addition, **it reduces the management burden of maintaining separate tools.**

A Unified Stack

- The Spark project contains **multiple closely integrated components.**

- At its core, **Spark is a "computational engine" that is responsible for scheduling, distributing, and monitoring applications** consisting of many computational tasks across many worker machines, or a *computing cluster*.

- **A philosophy of tight integration has several benefits.**

- First, **all libraries and higher-level components in the stack benefit from improvements at the lower layers.**

- For example, when Spark's core engine adds an optimization, SQL and machine learning libraries automatically speed up as well.

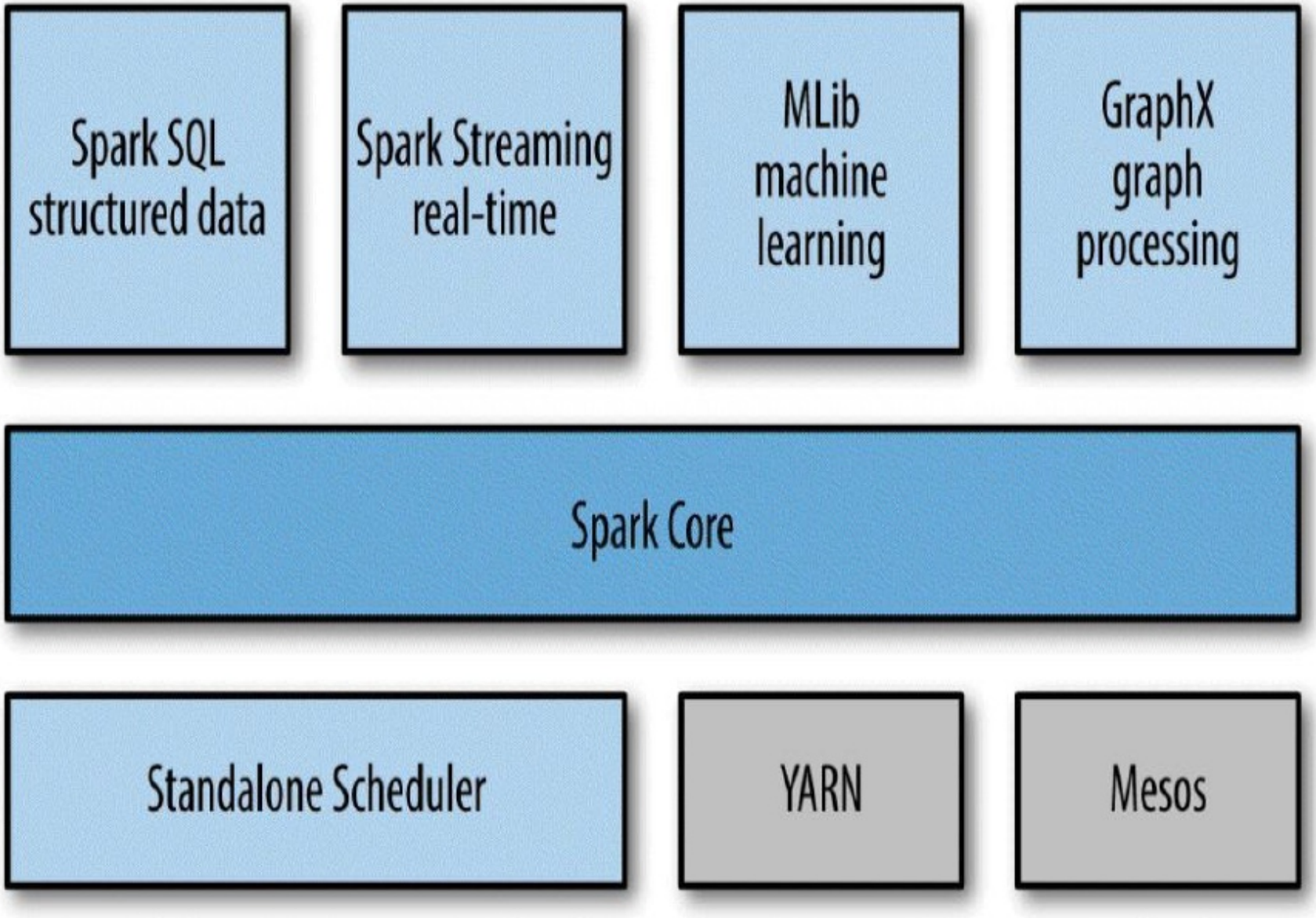
•Second, **the costs associated with running the stack are minimized, because instead of running 5–10 independent software systems, an organization needs to run only one.**

• **These costs include deployment, maintenance, testing, support, and others**

•Finally, **one of the largest advantages of tight integration is the ability to build applications that seamlessly combine different processing models.**

• For example, in Spark **you can write one application that uses machine learning to classify data in real time as it is ingested from streaming sources.**

•Simultaneously, **analysts can query the resulting data, also in real time, via SQL**



The Spark stack

Spark Core

- Spark Core contains the **basic functionality** of Spark, including components for **task scheduling, memory management, fault recovery, interacting with storage systems, and more**.

- **Spark Core is also home to the API that defines *resilient distributed datasets* (RDDs), which are Spark's main programming abstraction.**

Spark SQL

- **Spark SQL is Spark's package for working with structured data**.

- It allows **querying data via SQL as well as the Hive Query Language (HQL) — and it supports many sources of data, including Hive tables, Parquet, and JSON.**

- **Beyond providing a SQL interface to Spark**, Spark SQL allows **developers to intermix SQL queries with the programmatic data manipulations supported by RDD**s in Python, Java, and Scala, all within a single application, thus combining SQL with complex analytic

Spark Streaming

- Spark Streaming is a **Spark component that enables processing of live streams of data.**

- Examples of data streams include

- **logfiles generated by production web servers**, or
- **queues of messages containing status updates posted by users of a web service.**

- Spark Streaming **provides an API for manipulating data streams that closely matches the Spark Core's RDD API** that **manipulate data stored in memory, on disk, or arriving in real time.**

Mllib

Spark comes with a **library containing common machine learning (ML) functionality, called MLlib**.

MLlib provides multiple types of **machine learning algorithms, including classification, regression, clustering, and collaborative filtering, as well as supporting functionality such as model evaluation and data import**

GraphX

- GraphX is a **library for manipulating graphs** (e.g., a social network's friend graph) and **performing graph-parallel computations.**

- GraphX extends the Spark RDD API, allowing us to **create a directed graph with arbitrary properties attached to each vertex and edge.**

- **GraphX also provides various operators for manipulating graphs** (e.g., subgraph and mapVertices) and **a library of common graph algorithms**

Cluster Managers

- Spark is **designed to efficiently scale up from one to many thousands of compute nodes**.

- To **achieve this while maximizing flexibility, Spark can run over a variety of cluster managers, including Hadoop YARN, Apache Mesos, and a simple cluster manager included in Spark itself called the Standalone Scheduler**

Who Uses Spark, and for What?

Outlined two groups of readers - Data scientists and Engineers

Data Science Tasks

- *A data scientist is* **somebody whose main task is to analyze and model data.**

- Data scientists **may have experience with SQL, statistics, predictive modeling (machine learning), and programming**

- Data **scientists use their skills to analyze data with the goal of answering a question or discovering insights**.

- Oftentimes, their **workflow involves ad hoc analysis, so they use interactive shells**. Machine learning and data analysis is supported through the MLLib libraries.

- Sometimes, **after the initial exploration phase, the work of a data scientist will be productized or extended and tuned to become a production data processing application,** which itself is a component of a business application.

- Often **it is a different person or team that leads the process of productizing the work of the data scientists, and that person is often an engineer.**

Data Processing Applications

- The other **main use case of Spark can be described in the context of the engineer persona**.

- Engineers are **large class of software developers who use Spark to build production data processing applications**.

- For engineers, **Spark provides a simple way to parallelize these applications across clusters, and hides the complexity of distributed systems programming, network communication, and fault tolerance**.

- The **system gives them enough control to monitor, inspect, and tune applications while allowing them to implement common tasks quickly**.

Storage Layers for Spark

- Spark can **create distributed datasets from any file stored in the Hadoop distributed filesystem** (HDFS) or other storage systems supported by the Hadoop APIs (including your local filesystem, Amazon S3, Cassandra, Hive, HBase, etc.).

- It's important to remember that **Spark does not require Hadoop**;

- It simply has **support for storage systems implementing the Hadoop APIs.**

- Spark supports text files, SequenceFiles, Avro, Parquet, and any other Hadoop InputFormat.

Compare Hadoop and Spark

Feature Criteria	Apache Spark	Hadoop
Speed	100 times faster than Hadoop	Decent speed
Processing	Real-time & Batch processing	Batch processing only
Difficulty	Easy because of high level modules	Tough to learn
Recovery	Allows recovery of partitions	Fault-tolerant
Interactivity	Has interactive modes	No interactive mode except Pig & Hive

Introduction to Spark's Python and Scala Shells

- Spark comes with **interactive shells that enable ad hoc data analysis.**

- **Unlike most other shells**, however, **which let you manipulate data using the disk and memory on a single machine**, **Spark's shells allow you to interact with data that is distributed on disk or in memory across many machines**

- Spark takes care of **automatically distributing this processing**.

- **Because Spark can load data into memory** on the worker nodes, **many distributed computations can run in a few seconds**

- **Spark provides both Python and Scala Shells**

Programming with RDDs

•This chapter introduces Spark's core abstraction for working with data, the resilient distributed dataset (RDD).

•An RDD is simply a **distributed collection of elements.**

• In Spark **all work is expressed as either creating new RDDs, transforming existing RDDs, or calling operations on RDDs to compute a result.**

•Under the hood, **Spark automatically distributes the data contained in RDDs across your cluster and parallelizes the operations you perform on them**.

RDD Basics

•An RDD in Spark is simply an **immutable distributed collection** of objects.

• Each RDD is split into **multiple *partitions***, *which may be computed on different nodes of the cluster.*

•RDDs can **contain any type of Python, Java, or Scala objects, including user-defined classes**

Users create RDDs in two ways:
by loading an external dataset, OR
by distributing a collection of objects (e.g., a list or set) in their **driver program**

Creating an RDD of strings with textFile() in Python

```
>>> lines = sc.textFile("README.md")
```

•Once created, **RDDs offer two types of operations:** ***transformations and actions.***

•*Transformations* ***construct a new RDD from a previous one****. For example, one common* transformation is filtering data that matches a predicate

Calling the filter() transformation

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

•*Actions, on the other hand,* ***compute a result based on an RDD, and either return it to the*** **driver program or save it to an external storage system** (e.g., HDFS).

•One example of an action is first(), which returns the first element in an RDD

Calling the first() action

```
>>> pythonLines.first()
```

- Transformations and actions are **different because of the way Spark computes RDDs.**

- Although you can define new RDDs any time, Spark computes them only in a *lazy fashion* — that is, the first time they are used in an action

- Finally, **Spark's RDDs are by default recomputed each time you run an action on them.**

- If you would like to **reuse an RDD in multiple actions, you can ask Spark to *persist it using* RDD.persist().**

Persisting an RDD in memory

```
>>> pythonLines.persist
>>> pythonLines.count()
2
>>> pythonLines.first()
```

To summarize, every Spark program and shell session will work as follows:

1. Create some input RDDs from external data.

2. Transform them to define new RDDs using transformations like filter().

3. Ask Spark to persist() any intermediate RDDs that will need to be reused.

4. Launch actions such as count() and first() to kick off a parallel computation, which is then optimized and executed by Spark.

Creating RDDs

- Spark provides two ways to create RDDs: **loading an external dataset and parallelizing a collection in your driver program.**

- The simplest way to create RDDs is to take an existing collection in your program and **pass it to SparkContext's parallelize() method**, as shown

Example 3-5. parallelize() method in Python

```
lines = sc.parallelize([“pandas”, “i like pandas”])
```

Example 3-6. parallelize() method in Scala

```
val lines = sc.parallelize(List(“pandas”, “i like pandas”))
```

Example 3-7. parallelize() method in Java

```
JavaRDD<String> lines =
sc.parallelize(Arrays.asList(“pandas”, “i like pandas”));
```

- A more common way to **create RDDs is to load data from external storage.**

- One method that loads a text file as an RDD of strings, SparkContext.textFile(), which is shown

Example 3-8. textFile() method in Python

```
lines = sc.textFile("/path/to/README.md")
```

Example 3-9. textFile() method in Scala

```
val lines = c.textFile("/path/to/README.md")
```

Example 3-10. textFile() method in Java

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

RDD Operations

- As we've discussed, RDDs support two types of operations: ***transformations and actions.***

- **Transformations are operations on RDDs that return a new RDD, such as map() and filter().**

- **Actions are operations that return a result to the driver program or write it to storage,** and kick off a computation, such as count() and first().

- Spark treats transformations and actions very **differently, so** understanding which type of operation you are performing will be important.

- If you are ever confused whether a given function is a transformation or an action, you can look at its return type**: transformations return RDDs, whereas actions return some other data type.**

Transformations

- Transformations are operations **on RDDs that return a new RDD**.

- Transformed RDDs are **computed lazily, only when you use them in an action**

As an example, suppose that we have a logfile, *log.txt, with a number of messages, and we* want to **select only the error messages. We can use the filter() tra**nsformation

Example 3-11. filter() transformation in Python

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

Example 3-12. filter() transformation in Scala

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

Example 3-13. filter() transformation in 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"); }
```

Note that the **filter() operation does not mutate the existing inputRDD.**

Instead, it **returns a pointer to an entirely new RDD. inputRDD can still be reused later in the program** — for instance, to search for other words.

Let's use inputRDD again to search for lines with the word *warning in them.*

Then, we'll use another transformation, union(), to print out the number of lines that contained either *error or warning*

Example 3-14. union() transformation in Python

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

•Finally, as you **derive new RDDs from each other using transformations, Spark keeps track of the set of dependencies between different RDDs, called the *lineage graph*.**

•*It* ***uses*** **this information to compute each RDD on demand** and to recover lost data if part of a persistent RDD is lost

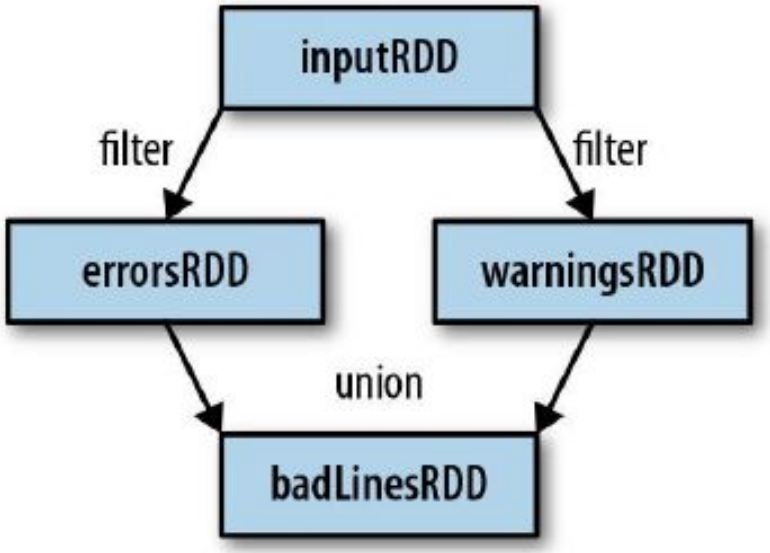


Figure 3-1. RDD lineage graph created during log analysis

Actions

•We've seen how to create RDDs from each other with transformations, but at some point, we'll want to **actually *do something with our dataset.***

• *Actions are the* ***second type of RDD*** **operation**.

•They are the operations that **return a final value to the driver program or write data to an external storage system.**

•Actions **force the evaluation of the transformations required for the RDD they were called on, since they need to actually produce output.**

•For example, we might want to **print out some information about the badLinesRDD.** To do that, we'll use two actions, count(), which returns the count as a number, and take(), which collects a number of elements from the RDD, as shown

Example 3-15. Python error count using actions

```
print “Input had “ + badLinesRDD.count() + ” concerning lines”
print “Here are 10 examples:”
for line in badLinesRDD.take(10):
print line
```

Example 3-16. Scala error count using actions

```
println(“Input had “ + badLinesRDD.count() + ” concerning lines”)
println(“Here are 10 examples:”)
badLinesRDD.take(10).foreach(println)
```

Example 3-17. Java error count using actions

```
System.out.println(“Input had “ + badLinesRDD.count() + ” concerning lines”)
System.out.println(“Here are 10 examples:”)
for (String line: badLinesRDD.take(10)) {
System.out.println(line);
}
```

In this example, we used take() to retrieve a small number of elements in the RDD at the driver program

We then iterate over them locally to print out information at the driver.

Lazy Evaluation

transformations on RDDs are lazily evaluated, meaning that Spark will not begin to execute until it sees an action.

- Lazy evaluation means that **when we call a transformation on an RDD (for instance, calling map()), the operation is not immediately performed.**

- Instead, **Spark internally records metadata to indicate that this operation has been requested.**

- Rather than **thinking of an RDD as containing specific data, it is best to think of each RDD as consisting of instructions on how to compute the data that we build up through transformations.**

- **Loading data into an RDD is lazily evaluated in the same way transformations are. So, when we call sc.textFile(), the data is not loaded until it is necessary**

Passing Functions to Spark

Most of Spark's transformations, and some of its actions, **depend on passing in functions that are used by Spark to compute data**

Passing functions in Python

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

Scala function passing

```
class SearchFunctions(val query: String) {

def isMatch(s: String): Boolean = {
s.contains(query)
}
def getMatchesFunctionReference(rdd: RDD[String]):
RDD[String] = {
rdd.map(isMatch)
```

Function name Method to implement Usage

Function<T, R> R call(T)
Take in one input and return one output, for use with operations like map() and filter().

Function2<T1, T2,R> R call(T1, T2)
Take in two inputs and return one output, for use with operations like
aggregate() or fold().

FlatMapFunction<T,R> Iterable<R> call(T)
Take in one input and return zero or more outputs, for use with operations like flatMap().

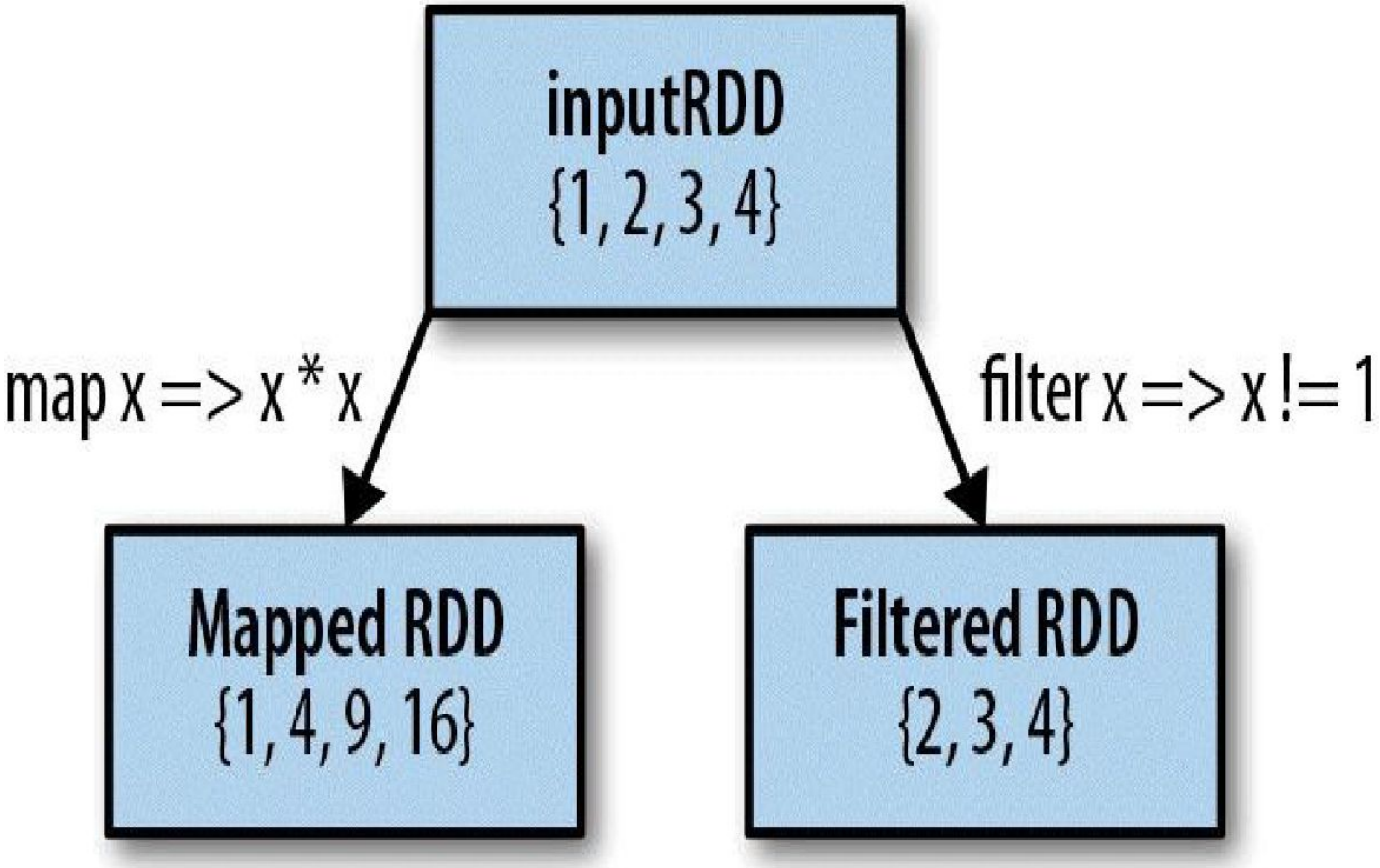
Common Transformations and Actions

Element-wise transformations

The two most **common transformations you will likely be using are map() and filter()**

The map() transformation **takes in a function and applies it to each element in the RDD with the result of the function being the new value of each element in** the resulting RDD.

The filter() transformation **takes in a function and returns an RDD that only has elements that pass the filter() function**


inputRDD
{1, 2, 3, 4}
map x => x * x
filter x => x != 1
Mapped RDD
{1, 4, 9, 16}
Filtered RDD
{2, 3, 4}

Example 3-26. Python squaring the values in an RDD

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

Example 3-27. Scala squaring the values in an RDD

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

Example 3-28. Java squaring the values in an RDD

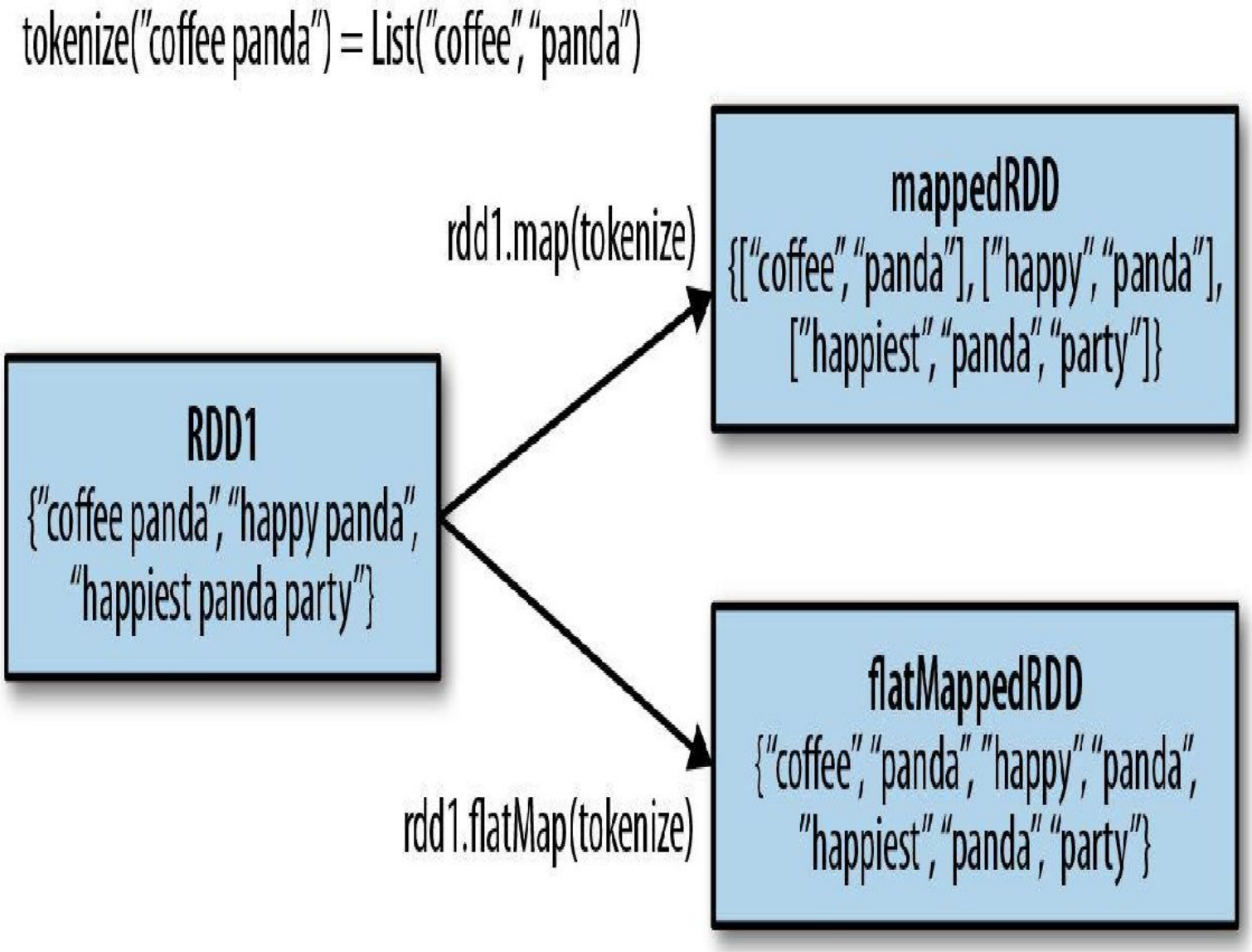
```
JavaRDD<Integer> rdd = sc.parallelize(Arrays.asList(1, 2,
3, 4));
JavaRDD<Integer> result = rdd.map(new
Function<Integer, Integer>() {
public Integer call(Integer x) { return x*x; }
});
System.out.println(StringUtils.join(result.collect(), “,”));
```

Sometimes we want to **produce multiple output elements for each input element.**

The operation to do this is called flatMap(). As with map(), the **function we provide to flatMap() is called individually for each element in our input RDD**.

Instead of returning a single element, we return an iterator with our return values.

Rather than producing an RDD of iterators, we get back an RDD that consists of the elements from all of the iterators


tokenize("coffee panda") = List("coffee", "panda")
rdd1.map(tokenize)
mappedRDD
{["coffee", "panda"], ["happy", "panda"],
["happiest", "panda", "party"]}
RDD1
{"coffee panda", "happy panda",
"happiest panda party"}
rdd1.flatMap(tokenize)
flatMappedRDD
{"coffee", "panda", "happy", "panda",
"happiest", "panda", "party"}

Example 3-29. flatMap() in Python, splitting lines into words

```
lines = sc.parallelize(["hello world", "hi"])
words = lines.flatMap(lambda line: line.split(" "))
words.first() # returns "hello"
```

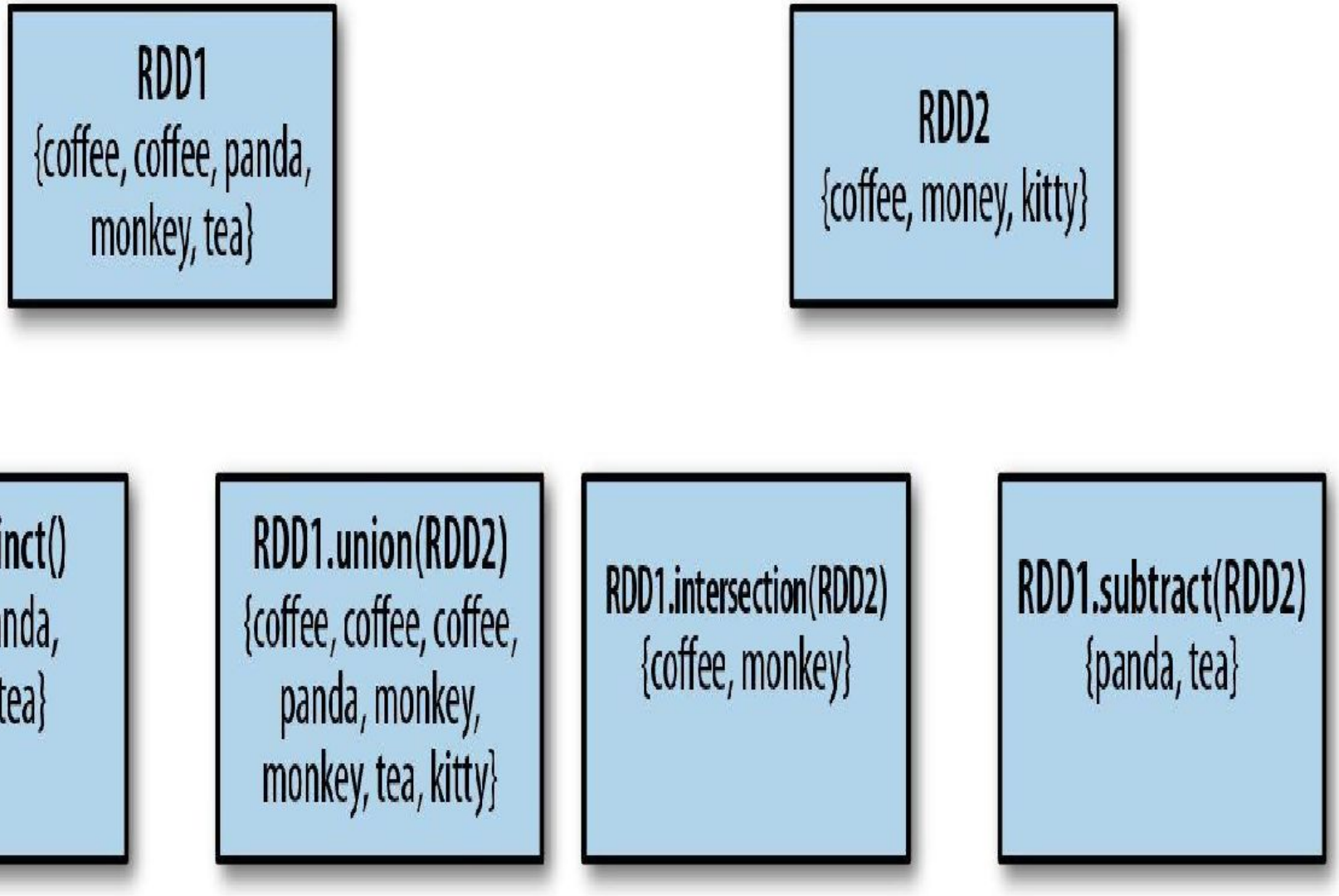
Example 3-30. flatMap() in Scala, splitting lines into multiple words

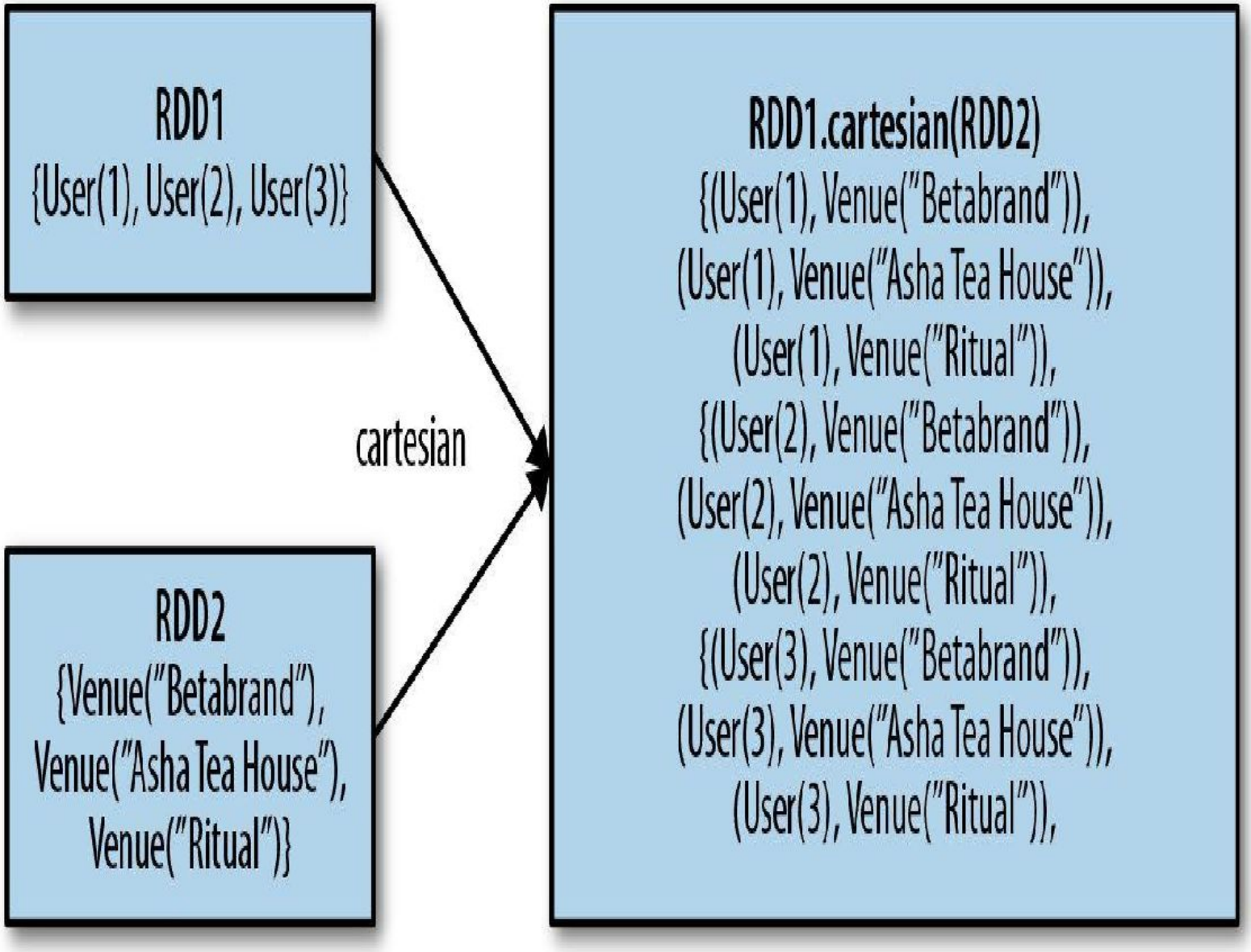
```
val lines = sc.parallelize(List("hello world", "hi"))
val words = lines.flatMap(line => line.split(" "))
words.first() // returns "hello"
```

Example 3-31. flatMap() in Java, splitting lines into multiple words

```
JavaRDD<String> lines = sc.parallelize(Arrays.asList("hello world",
"hi"));
JavaRDD<String> words = lines.flatMap(new
FlatMapFunction<String, String>() {
public Iterable<String> call(String line) {
return Arrays.asList(line.split(" "));
}
```

Pseudo set operations




RDD1
{User(1), User(2), User(3)}
RDD2
{Venue("Betabrand"),
Venue("Asha Tea House"),
Venue("Ritual")}
cartesian
RDD1.cartesian(RDD2)
{(User(1), Venue("Betabrand")),
(User(1), Venue("Asha Tea House")),
(User(1), Venue("Ritual")),
{(User(2), Venue("Betabrand")),
(User(2), Venue("Asha Tea House")),
(User(2), Venue("Ritual")),
{(User(3), Venue("Betabrand")),
(User(3), Venue("Asha Tea House")),
(User(3), Venue("Ritual")),

Basic RDD transformations on an RDD containing {1, 2, 3, 3}

map()

Apply a function to each element in the RDD and return an RDD of the result.

rdd.map(x => x + 1) {2, 3, 4, 4}

flatMap()

Apply a function to each element in the RDD and return an RDD of the contents of the iterators returned. Often used to extract words.

rdd.flatMap(x => x.to(3)) {1, 2, 3, 2, 3, 3, 3}

filter()

Return an RDD consisting of only elements that pass the condition passed to filter().

rdd.filter(x => x != 1) {2, 3, 3}

distinct()

Remove duplicates.

rdd.distinct() {1, 2, 3}

Two-RDD transformations on RDDs containing {1, 2, 3} and {3, 4, 5}

Function name Purpose Example Result

union()- Produce an RDD containing elements from both RDDs.
rdd.union(other)
{1, 2, 3, 3, 4, 5}

intersection() - RDD containing only elements found in both RDDs.
rdd.intersection(other) {3}

subtract() -Remove the contents of one RDD (e.g., remove training data).
rdd.subtract(other) {1, 2}

cartesian() -Cartesian product with the other RDD.
rdd.cartesian(other)
{(1, 3), (1, 4), … (3,5)}

What are accumulators?

Accumulators are **variables that are used for aggregating information across the executors**

For example, this information can pertain to data or API diagnosis like **how many records are corrupted or how many times a particular library API was called.**

To understand why we need accumulators, let's see a small example.

```
Kolkata Central Avenue Groceries 233.65

Kolkata Bowbazar Hair Care 198.99
Bad data packet
Kolkata Amherst Street Beverages 0
```

There are 4 fields,

Field 1 -> City

Field 2 -> Locality

Field 3 -> Category of item sold

Field 4 -> Value of item sold

However, the logs can be corrupted. For example, the second line is a blank line, the fourth line reports some network issues and finally the last line shows a sales value of zero (which cannot happen!).

We can use accumulators to analyse the transaction log to find out the number of blank logs (blank lines), number of times the network failed, any product that does not have a category or even number of times zero sales were recorded.

```
var blankLines: Int = 0

sc.textFile("some log file", 4)
    .foreach { line =>
        if (line.length() == 0) blankLines += 1
    }

println(s"Blank Lines=$blankLines")
```

The problem with the above code is that when the **driver prints the variable *blankLines* its value will be zero**.

This is because when **Spark ships this code to every executor the variables become local to that executor and its updated value is not relayed back to the driver**.

To avoid this problem **we need to make *blankLines* an accumulator such that all the updates to this variable in every executor is relayed back to the driver.**

So the above code should be written as,

```
val blankLines = sc.accumulator(0, "Blank Lines")

sc.textFile("some log file", 4)
    .foreach { line =>
        if (line.length() == 0) blankLines += 1
    }

println(s"\tBlank Lines=${blankLines.value}")
```

This guarantees that the accumulator *blankLines* is updated across every executor and the updates are relayed back to the driver.

Actions

The most common action on basic RDDs you will likely use is **reduce(), which takes a function that operates on two elements of the type in your RDD and returns a new element of the same type**

Example 3-32. reduce() in Python

```
sum = rdd.reduce(lambda x, y: x + y)
```

Example 3-33. reduce() in Scala

```
val sum = rdd.reduce((x, y) => x + y)
```

Example 3-34. reduce() in Java

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

Reducebykey()

```
val data = Seq(("Project", 1),
  ("Gutenberg's", 1),
  ("Alice's", 1),
  ("Adventures", 1),
  ("in", 1),
  ("Wonderland", 1),
  ("Project", 1),
  ("Gutenberg's", 1),
  ("Adventures", 1),
  ("in", 1),
  ("Wonderland", 1),
  ("Project", 1),
  ("Gutenberg's", 1))

  val  rdd=spark.sparkContext.parallelize(data)
```

As you see the data here, it's in key/value pair.

Key is the work name and value is the count.

Below is the syntax of the Spark RDD reduceByKey() transformation

```
val rdd2=rdd.reduceByKey(_ + _)
  rdd2.foreach(println)
```

```
('Project', 3)
('Gutenberg's', 3)
('Alice's', 1)
('in', 2)
('Adventures', 2)
('Wonderland', 2)
```

- Similar to reduce() is **fold(), which also takes a function with the same signature as needed for reduce(), but in addition takes a "zero value" to be used for the initial call on each partition**.

- The **zero value you provide should be the identity element for your operation**; that is, applying it multiple times with your function should not change the value (**e.g., 0 for +, 1 for *, or an empty list for concatenation**).

Fold in spark

```
def fold[T](acc:T)((acc,value) => acc)
```

1. T is the data type of RDD
2. acc is accumulator of type T which will be return value of the fold operation
3. A function , which will be called for each element in rdd with previous accumulator.

Finding max in a given RDD

Let's first build a RDD

```
val employeeData = List(("Jack",1000.0),("Bob",2000.0),("Carl",7000.0))
```

`val employeeRDD = sparkContext.makeRDD(employeeData)`

Now we want to find an employee, with maximum salary. We can do that using fold.

To use fold we need a start value. The following code defines a dummy employee as starting accumulator.

`val dummyEmployee = ("dummy",0.0);`

Now **using fold,** we can **find the employee with maximum salary**.

```
val maxSalaryEmployee =
employeeRDD.fold(dummyEmployee)((acc,employee) => {
if(acc._2 < employee._2) employee else acc})
println("employee with maximum salary is"+maxSalaryEmployee)
```

Both **fold() and reduce() require that the return type** of our **result be the same type as that of the elements in the RDD** we are operating over.

This works well for operations **like sum**, but sometimes we want to return a different type.

The **aggregate() function frees us from the constraint of having the return be the same type as the RDD we are working on**.

With **aggregate(), like fold(), we supply an initial zero value of the type we want to return.**

We then **supply a function to combine the elements from our RDD with the accumulator.**

Finally, **we need to supply a second function to merge two accumulators**, given that each node accumulates its own results

RDD aggregate() Syntax

Since RDD's are partitioned, the aggregate takes full advantage of it by first aggregating elements in each partition and then aggregating results of all partition to get the final result. and the result could be any type than the type of your RDD.

aggregate[U](zeroValue: U)(seqOp: (U, T) ⇒ U, combOp: (U, U) ⇒ U)

This takes the following arguments –

`zeroValue` – Initial value to be **used for each partition in aggregation, this value would be used to initialize the accumulator**. we mostly use `0` for integer and `Nil` for collections.

`seqOp` – This operator is used to accumulate the results of each partition, and stores the running accumulated result to U,
`combOp` – This operator is used to combine the results of all partitions U.

```
  val listRdd =
spark.sparkContext.parallelize(List(1,2,3,4,5,3,2))
  def param0= (accu:Int, v:Int) => accu + v
  def param1= (accu1:Int,accu2:Int) => accu1 +
accu2
  val result = listRdd.aggregate(0)(param0,param1)
  println("output 1 =>" + result)
Output= 20
```

```
cala> val inputrdd = sc.parallelize(
List(
(“maths”, 21),
(“english”, 22),
(“science”, 31)
),    )
scala> val result = inputrdd.aggregate(0) (

(acc, value) => (acc + value._2),

/*
* This is a combOp for mergining two U’s
* (ie 2 Int)
*/
(acc1, acc2) => (acc1 + acc2)
)
result: Int = 75
```

The result is calculated as follows,
Partition 1 : Sum(all Elements) + (Zero value)
Partition 2 : Sum(all Elements) + (Zero value)
Partition 3 : Sum(all Elements) + (Zero value)
Result = Partition1 + Partition2 + Partition3 +
So we get 21 + 22 + 31 = 75.

We can use aggregate() to compute the average of an RDD, avoiding a map() before the fold(), as shown in Examples 3-35 through 3-37.

Example 3-35. aggregate() in 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])))))
return sumCount[0] / float(sumCount[1])
```

Example 3-36. aggregate() in 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
```

Example 3-37. aggregate() in Java

```
class AvgCount implements Serializable {
public AvgCount(int total, int num) {
this.total = total;
this.num = num;
}
public int total;
public int num;
public double avg() {
return total / (double) num;
}
} Function2<AvgCount, Integer, AvgCount> addAndCount =
new Function2<AvgCount, Integer, AvgCount>() {
public AvgCount call(AvgCount a, Integer x) {
a.total += x;
a.num += 1;
return a;
} };
Function2<AvgCount, AvgCount, AvgCount> combine =
new Function2<AvgCount, AvgCount, AvgCount>() {
public AvgCount call(AvgCount a, AvgCount b) {
a.total += b.total;
a.num += b.num;
return a; } };
```

```
AvgCount initial = new AvgCount(0, 0);
AvgCount result = rdd.aggregate(initial,
addAndCount, combine);
System.out.println(result.avg());
```

collect(), which returns the entire RDD's contents. collect() is commonly used in unit tests where the entire contents of the RDD are expected to fit in memory, as that makes it easy to compare the value of our RDD with our expected result.

take(n) returns *n* elements from the RDD and attempts to minimize the number of partitions it accesses, so it may represent a biased collection.

If there is an ordering defined on our data, we can also extract the top elements from an RDD using top(). top() will use the default ordering on the data, but we can supply our own comparison function to extract the top elements

Table 3-4. Basic actions on an RDD containing {1, 2, 3, 3}

Function name	Purpose	Example	Result
`collect()`	Return all elements from the RDD.	`rdd.collect()`	`{1, 2, 3, 3}`
`count()`	Number of elements in the RDD.	`rdd.count()`	`4`
`countByValue()`	Number of times each element occurs in the RDD.	`rdd.countByValue()`	`{(1, 1), (2, 1), (3, 2)}`
`take(num)`	Return `num` elements from the RDD.	`rdd.take(2)`	`{1, 2}`
`top(num)`	Return the top `num` elements the RDD.	`rdd.top(2)`	`{3, 3}`

`takeOrdered(num)(ordering)`	Return `num` elements based on provided ordering.	`rdd.takeOrdered(2)(myOrdering)`	`{3, 3}`
`takeSample(withReplacement, num, [seed])`	Return `num` elements at random.	`rdd.takeSample(false, 1)`	Nondeterministic
`reduce(func)`	Combine the elements of the RDD together in parallel (e.g., `sum`).	`rdd.reduce((x, y) => x + y)`	9
`fold(zero)(func)`	Same as `reduce()` but with the provided zero value.	`rdd.fold(0)((x, y) => x + y)`	9
`aggregate(zeroValue)(seqOp, combOp)`	Similar to `reduce()` but used to return a different type.	`rdd.aggregate((0, 0)) ((x, y) => (x._1 + y, x._2 + 1), (x, y) => (x._1 + y._1, x._2 + y._2))`	`(9, 4)`

`foreach(func)`	Apply the provided function to each element of the RDD.	`rdd.foreach(func)`	Nothing

Persistence (Caching)

Spark RDDs are lazily evaluated, and sometimes **we may wish to use the same RDD multiple times.**

If we do this naively, Spark will **recompute the RDD and all of its dependencies each time we call an action on the RDD.**

This can be **especially expensive for iterative algorithms, which look at the data many times.**

Another trivial example would be **doing a count and then writing out the same RDD, as shown**

```
val result = input.map(x => x*x)
println(result.count())
println(result.collect().mkString(“,”))
```

To **avoid computing an RDD multiple times**, we can ask Spark to **persist the data**.

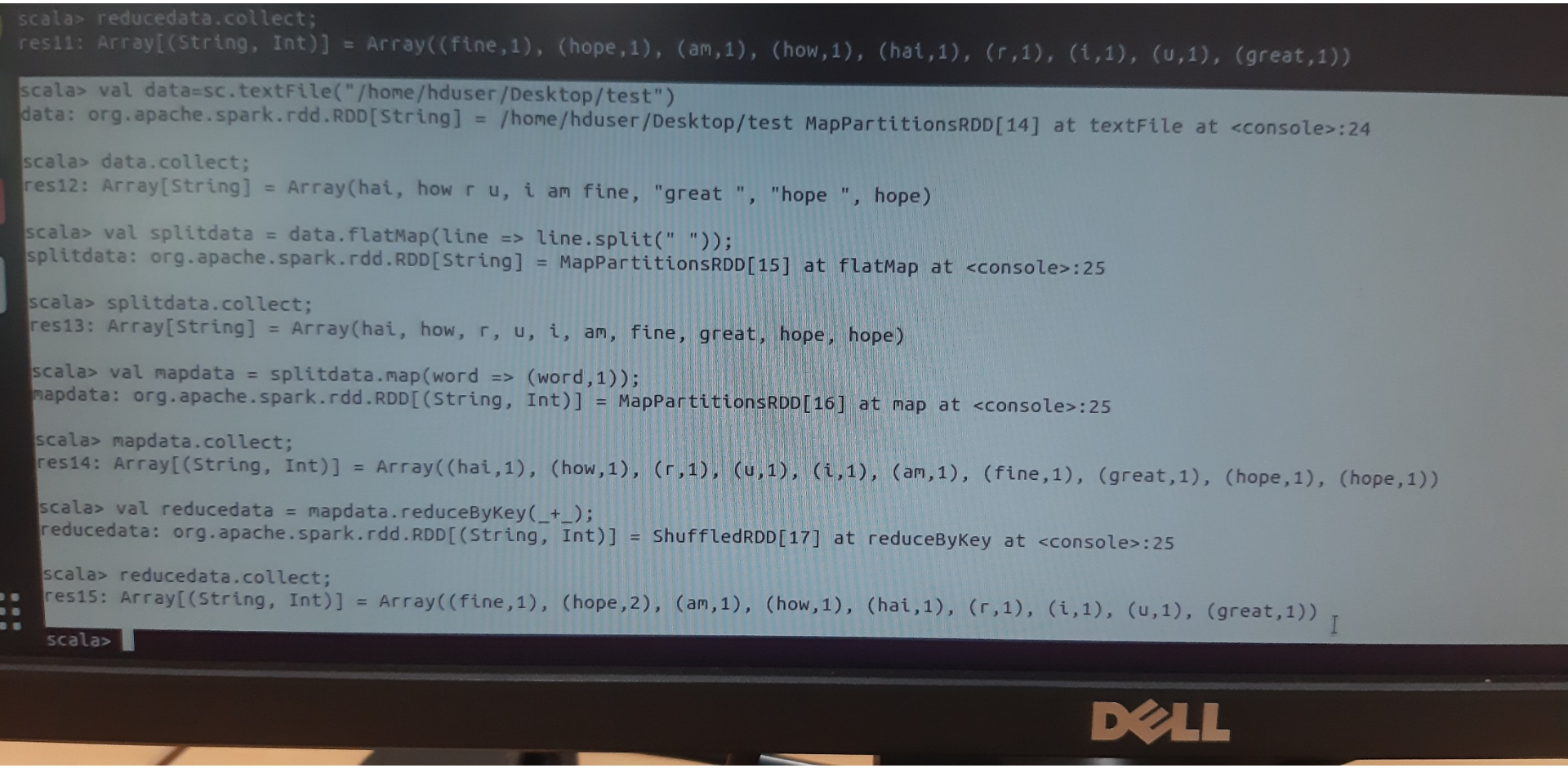
When we ask **Spark to persist an RDD, the nodes that compute the RDD store their partitions.**

Table 3-6. Persistence levels from org.apache.spark.storage.StorageLevel and pyspark.StorageLevel; if desired we can replicate the data on two machines by adding _2 to the end of the storage level

Level	Space used	CPU time	In memory	On disk	Comments
MEMORY_ONLY	High	Low	Y	N	
MEMORY_ONLY_SER	Low	High	Y	N	
MEMORY_AND_DISK	High	Medium	Some	Some	Spills to disk if there is too much data to fit in memory.
MEMORY_AND_DISK_SER	Low	High	Some	Some	Spills to disk if there is too much data to fit in memory. Stores serialized representation in memory.
DISK_ONLY	Low	High	N	Y	

Program to run wordcount on scala shell
Note- Create a textfile sparkdata.txt locally and give appropriate path while loading the data using sc.textFile

```
val data=sc.textFile("sparkdata.txt")

data.collect;

val splitdata = data.flatMap(line => line.split(" "));

splitdata.collect;

val mapdata = splitdata.map(word => (word,1));

mapdata.collect;

val reducedata = mapdata.reduceByKey(_+_);

reducedata.collect;
```


scala> reducedata.collect;
res11: Array[(String, Int)] = Array((fine,1), (hope,1), (am,1), (how,1), (hai,1), (r,1), (i,1), (u,1), (great,1))
scala> val data=sc.textFile("/home/hduser/Desktop/test")
data: org.apache.spark.rdd.RDD[String] = /home/hduser/Desktop/test MapPartitionsRDD[14] at textFile at <console>:24
scala> data.collect;
res12: Array[String] = Array(hai, how r u, i am fine, "great ", "hope ", hope)
scala> val splitdata = data.flatMap(line => line.split(" "));
splitdata: org.apache.spark.rdd.RDD[String] = MapPartitionsRDD[15] at flatMap at <console>:25
scala> splitdata.collect;
res13: Array[String] = Array(hai, how, r, u, i, am, fine, great, hope, hope)
scala> val mapdata = splitdata.map(word => (word,1));
mapdata: org.apache.spark.rdd.RDD[(String, Int)] = MapPartitionsRDD[16] at map at <console>:25
scala> mapdata.collect;
res14: Array[(String, Int)] = Array((hai,1), (how,1), (r,1), (u,1), (i,1), (am,1), (fine,1), (great,1), (hope,1), (hope,1))
scala> val reducedata = mapdata.reduceByKey(_+_);
reducedata: org.apache.spark.rdd.RDD[(String, Int)] = ShuffledRDD[17] at reduceByKey at <console>:25
scala> reducedata.collect;
res15: Array[(String, Int)] = Array((fine,1), (hope,2), (am,1), (how,1), (hai,1), (r,1), (i,1), (u,1), (great,1))
scala>
DELL

Using RDD and FlaMap count how many times each word appears in a file and write out a list of words whose count is strictly greater than 4 using Spark.

```
  val textFile = sc.textFile("/home/bhoom/Desktop/wc.txt")
val counts = textFile.flatMap(line => line.split(" ")).map(word => (word, 1)).reduceByKey(_ + _)
import scala.collection.immutable.ListMap
val sorted=ListMap(counts.collect.sortWith(_._2 > _._2):_*)// sort in descending order based on values
println(sorted)
for((k,v)<-sorted)
{
  if(v>4)
    {
     print(k+",")
       print(v)
       println()
     }
}
```

```
        print(v)
        println()
      }
    }
```

```
scala> val textFile = sc.textFile("/home/hduser/Desktop/test")
textFile: org.apache.spark.rdd.RDD[String] = /home/hduser/Desktop/test MapPartitionsRDD[24] at textFile at <console>:25

scala> val counts = textFile.flatMap(line => line.split(" ")).map(word => (word, 1)).reduceByKey(_ + _)
counts: org.apache.spark.rdd.RDD[(String, Int)] = ShuffledRDD[27] at reduceByKey at <console>:26

scala> import scala.collection.immutable.ListMap
import scala.collection.immutable.ListMap

scala> val sorted=ListMap(counts.collect.sortWith(_._2 > _._2):_*)
sorted: scala.collection.immutable.ListMap[String,Int] = Map(hope -> 5, fine -> 1, am -> 1, how -> 1, "" -> 1, hai -> 1, r -> 1, i -> 1, u -> 1, great -> 1)

scala> println(sorted)
Map(hope -> 5, fine -> 1, am -> 1, how -> 1,  -> 1, hai -> 1, r -> 1, i -> 1, u -> 1, great -> 1)

scala> for((k,v)<-sorted)
     | {
     |   if(v>4)
     |     {
     |       print(k+",")
     |         print(v)
     |         println()
     |       }
     | }
hope,5

scala>
```

Spark RDD Transformations with Examples

```
1. val spark:SparkSession = SparkSession.builder()
      .master("local[3]")
      .appName("SparkByExamples.com")
      .getOrCreate()

val sc = spark.sparkContext

val rdd:RDD[String] = sc.textFile("src/main/scala/test.txt")


2. val rdd2 = rdd.flatMap(f=>f.split(" "))
```

3. val rdd3:RDD[(String,Int)]= rdd2.map(m=>(m,1))

4. val rdd4 = rdd3.filter(a=> a._1.startsWith("a"))

5. val rdd5 = rdd3.reduceByKey(_ + _)

6. SortByKey() transformation is used to sort RDD elements on key.

In our example, first, we convert RDD[(String,Int]) to RDD[(Int,String]) using map transformation and apply sortByKey which ideally does sort on an integer value.

And finally, foreach with println statement prints all words in RDD and their count as key-value pair to console.

```
val rdd6 = rdd5.map(a=>(a._2,a._1)).sortByKey()
//Print rdd6 result to console
rdd6.foreach(println)
```

def reduce(f: (T, T) => T): T
RDD reduce() function takes function type as an argument and returns the RDD with the same type as input. It reduces the elements of the input RDD using the binary operator specified.

Reduce a list – Calculate min, max, and total of elements

```
val listRdd = spark.sparkContext.parallelize(List(1,2,3,4,5,3,2))
 println("output min using binary : "+listRdd.reduce(_ min _))
 println("output max using binary : "+listRdd.reduce(_ max _))
 println("output sum using binary : "+listRdd.reduce(_ + _))
```

Alternatively, you can also write the above operations as below.

```
val listRdd = spark.sparkContext.parallelize(List(1,2,3,4,5,3,2))
println("output min : "+listRdd.reduce( (a,b) => a min b))
println("output max : "+listRdd.reduce( (a,b) => a max b))
println("output sum : "+listRdd.reduce( (a,b) => a + b))
```

```
output min : 1
output max : 5
output sum : 2
```

Reduce function on Tupple RDD(String,Int)

```
 val inputRDD = spark.sparkContext.parallelize(List(("Z", 1),("A", 20),("B", 30),("C", 40),("B", 30),("B", 60)))

  println("output min : "+inputRDD.reduce( (a,b)=> ("max",a._2 min b._2))._2)
  println("output max : "+inputRDD.reduce( (a,b)=> ("max",a._2 max b._2))._2)
  println("output sum : "+inputRDD.reduce( (a,b)=> ("Sum",a._2 + b._2))._2)
```

Points to Note

reduce() is similar to fold() except reduce takes a ‘Zero value‘ as an initial value for each partition.

reduce() is similar to aggregate() with a difference; reduce return type should be the same as this RDD element type whereas aggregation can return any type.

reduce() also same as reduceByKey() except reduceByKey() operates on Pair RDD

```
package com.sparkbyexamples.spark.rdd.functions
import org.apache.spark.sql.SparkSession

object reduceExample extends App {

  val spark = SparkSession.builder()
    .appName("SparkByExamples.com")
    .master("local[3]")
    .getOrCreate()

  spark.sparkContext.setLogLevel("ERROR")

  val listRdd = spark.sparkContext.parallelize(List(1,2,3,4,5,3,2))

  println("output sum using binary : "+listRdd.reduce(_ min _))
  println("output min using binary : "+listRdd.reduce(_ max _))
  println("output max using binary : "+listRdd.reduce(_ + _))

  // Alternatively you can write
  println("output min : "+listRdd.reduce( (a,b) => a min b))
  println("output max : "+listRdd.reduce( (a,b) => a max b))
  println("output sum : "+listRdd.reduce( (a,b) => a + b))


  val inputRDD = spark.sparkContext.parallelize(List(("Z", 1),("A", 20),("B", 30),
    ("C", 40),("B", 30),("B", 60)))

  println("output max : "+inputRDD.reduce( (a,b)=> ("max",a._2 min b._2))._2)
  println("output max : "+inputRDD.reduce( (a,b)=> ("max",a._2 max b._2))._2)
  println("output sum : "+inputRDD.reduce( (a,b)=> ("Sum",a._2 + b._2))._2)     }
```