Dipartimento di Ingegneria Informatica
Automatica e Gestionale Antonio Ruberti

Sapienza
Università di Roma

Laboratory of Advanced Programming

Spark/Scala Tutorial

Marco Calamo, Matteo Marinacci, Jacopo Rossi
{calamo,marinacci,j.rossi}@diag.uniroma1.it

Course Outline

- **Spark:** Introduction
  - Spark Framework
  - Spark Resource management
- **Scala:** Introduction
  - Examples (Interactive Shell)
  - Examples (IDE)
- **Processing Data**
  - Working flow
  - Dstreams
  - Transformations
  - RDD Introduction
    - Programming Examples
  - Dataframes Introduction
    - Programming Examples

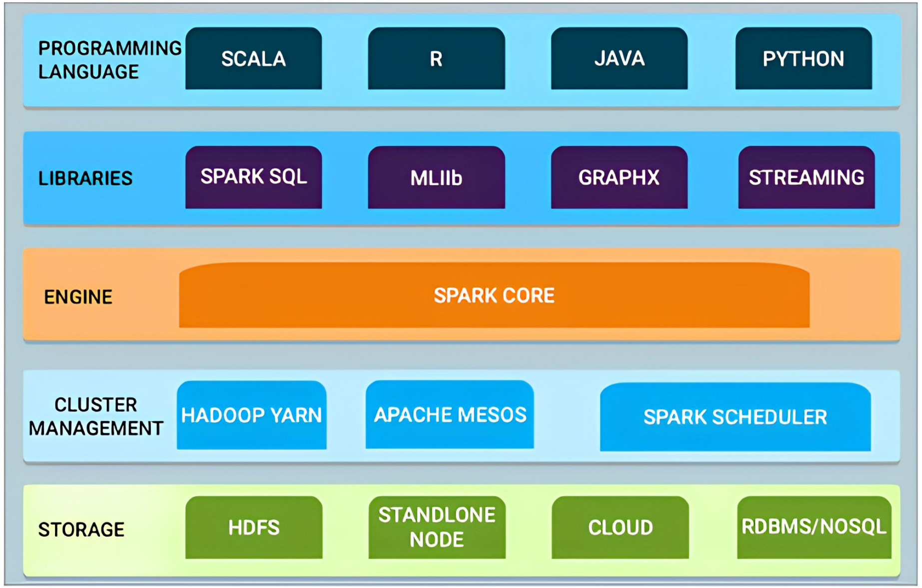
Spark Framework

What is SPARK?

- **Analytics Engine, Big Data Engine** for large-scale data processing.

- *Advantages:*
  - faster than Hadoop;
  - write Spark applications in several programming languages: Java, Python, Scala and others;
  - Platform Independent;
  - does not need any STORAGE. “Give me data, I will process it.”

Spark framework (Overview)



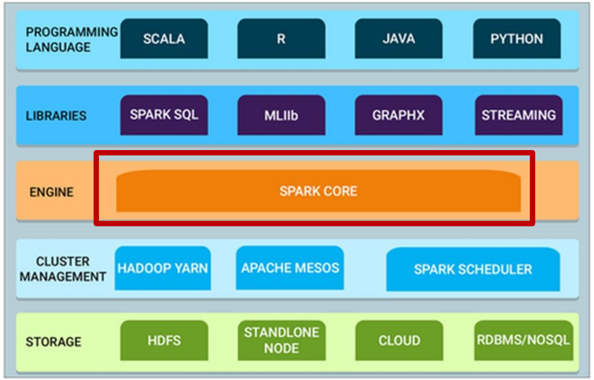
Spark framework (Spark Core)

Dipartimento di Ingegneria Informatica Automatica e Gestionale Antonio Ruberti
Sapienza Università di Roma

- Spark Core is a general-purpose, distributed data processing engine.

- On top of it sit libraries for **SQL, stream processing, machine learning, and graph computation**

- Is the base of a whole project, providing distributed task dispatching, scheduling, and basic I/O functionalities.



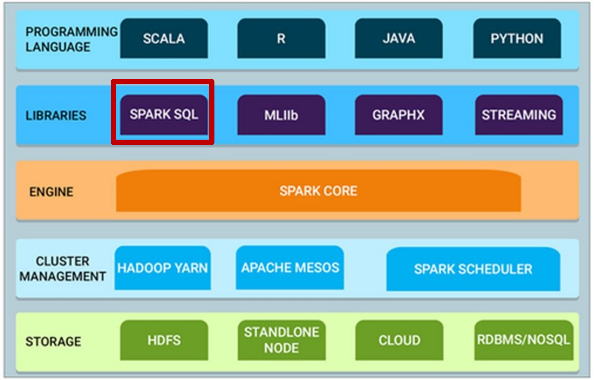
Spark framework (Spark SQL)

Dipartimento di Ingegneria Informatica Automatica e Gestionale Antonio Ruberti
Sapienza Università di Roma

- Spark SQL is the Spark module for working with structured data that supports a common way to access a variety of data sources.

- It lets you query structured data inside Spark programs, using either **SQL or a familiar DataFrame API.**

- A server mode provides standard connectivity through **Java database connectivity or open database connectivity.**

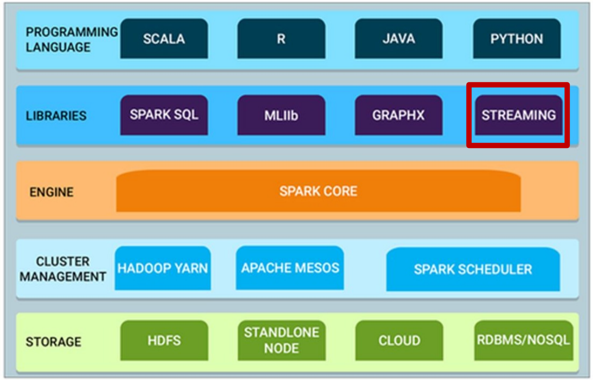


Spark framework (Spark Streaming)

Dipartimento di Ingegneria Informatica Automatica e Gestionale Antonio Ruberti
Sapienza Università di Roma

- Spark Streaming makes it easy to build scalable, fault-tolerant streaming solutions.

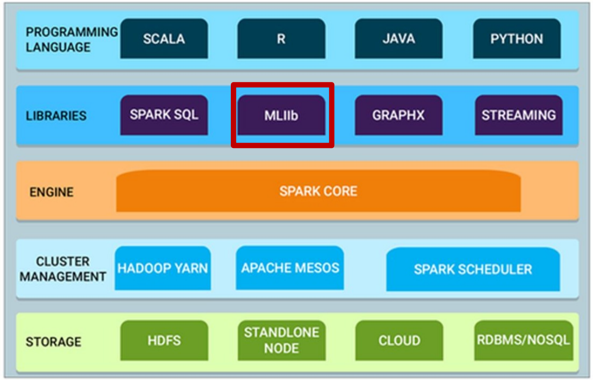
- It brings the Spark language-integrated API to **stream processing**, so you can write streaming jobs in the same way as batch jobs.



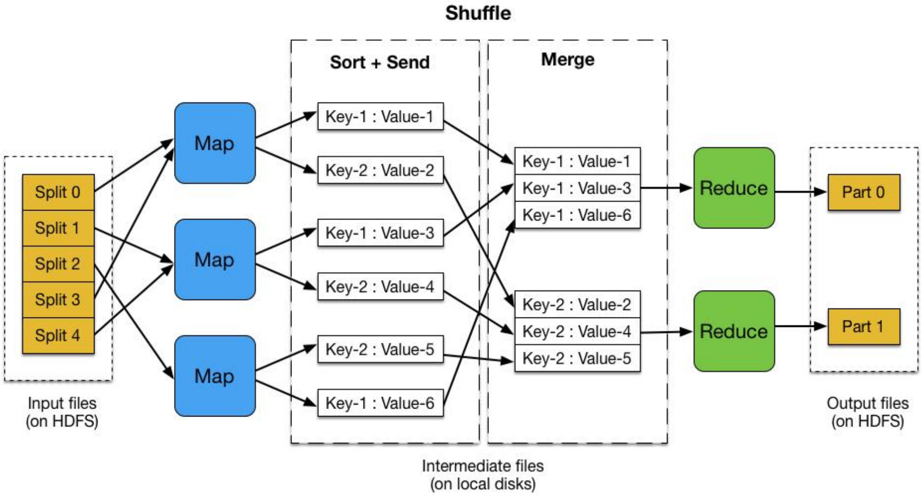
Spark framework (Spark MLlib)

Dipartimento di Ingegneria Informatica Automatica e Gestionale Antonio Ruberti
Sapienza Università di Roma

- MLlib is the Spark scalable machine learning library with tools that make practical ML scalable and easy.
- It contains many common learning algorithms, such as **classification, regression, recommendation, and clustering**.
- It also contains workflow and other utilities, including feature **transformations, ML pipeline construction, model evaluation, distributed linear algebra, and statistics**.



MapReduce Paradigm



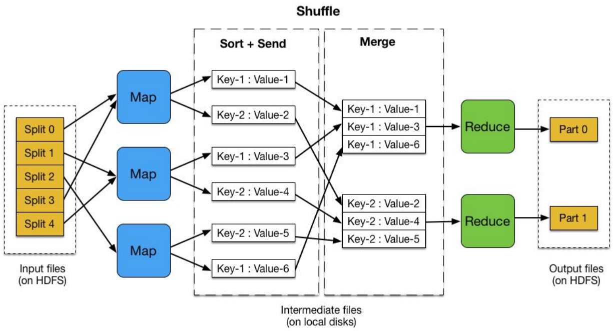
MapReduce Paradigm

Map

The map function, also referred to as the map task, processes a single key/value input pair and produces a set of intermediate key/value pairs.

Reduce

The reduce function, also referred to as the reduce task, consists of taking all key/value pairs produced in the map phase that share the same intermediate key and producing zero, one, or more data items.



Dipartimento di Ingegneria Informatica
Automatica e Gestionale Antonio Ruberti
Sapienza Università di Roma

Resource Management

- Resources are taken from the cluster manager (e.g. yarn)
    - **Executor:** a combination of CPU and RAM.

- How much RAM do you need? How much CPU?
- You can set these information up.
- **Which is the correct way to setup them?**

Resource Management (Example)

50 Gigabyte File

We want to analyze this big data file.
How much resources we need?
How many executors? We have several options.

- **A. Single Executor.**
  - All the workload on a single executor. This is not a good idea…

Resource Management (Example)

Dipartimento di Ingegneria Informatica
Automatica e Gestionale Antonio Ruberti
Sapienza
Università di Roma

- 50 Gigabyte File

  We want to analyze this big data file.
  How much resources we need?
  How many executors? We have several options.
- **B. 5 Executors:** ex-1, ex-2, ex-3, ex-4 and ex-5.
  - Each executor: 10 GB (RAM) and 10 CPU Cores.
- Then you can partition the file: 50 GB File -> 50 partitions
- Then you can distribute the workload as:

  ex-1 partitions from 1 to 10

  ex-2 partitions from 11 to 20

  ex-3 partitions from 21 to 30

  ex-4 partitions from 31 to 40

  ex-5 partitions from 41 to 50
- Each CPU core has its partition!

Resource Management (Example)

- 50 Gigabyte File

  We want to analyze this big data file.
  How much resources we need?
  How many executors? We have several options.

- **C. 50 Executors!**

- Data can move between executors. Having too many executors could be a problem.

- **The idea is not to have too less or too more executors.**
  There is no magical number, it depends on your design choices!

Setup Executors via command line

(Three major releases. For this course I will use version 2.3.1)

- `> spark-shell` -> launch interactive shell using Scala on top of Spark

And then add these options:

- `--num-executors NUM` -> set how many executors
- `--executor-memory MEM` -> set RAM for each executor
- `--executor-cores NUM` -> set CPU cores for each executor

E.g.: 5 executors, 2GB (RAM) and 2 CPU cores for each one of them.

- `> spark-shell --num-executors 5 --executor-memory 2G --executor-cores 2`

Dynamic Resource Allocation (DRA)

Dipartimento di Ingegneria Informatica
Automatica e Gestionale Antonio Ruberti
Sapienza
Università di Roma

- **DRA** automatically and dynamically adjusts resources for your Spark application. It frees up resources when they are no longer needed.
- Disabled by default.
- Available on all cluster managers (yarn, mesos, standalone…).

In order to enable it, add this option at launch:

```
--conf spark.dynamicAllocation.enabled=true
```

Dipartimento di Ingegneria Informatica
Automatica e Gestionale Antonio Ruberti

Sapienza
Università di Roma

Scala Programming language

What is Scala?

- **Java** -> works on (almost) every platform, complex, difficult to learn.
- **Python** -> scripting language, easy to learn, it suffers of some optimization problems.

- **Scala** pick the best of both languages. It combines object-oriented and functional programming in one concise, high-level language.

- It uses the same **Java Virtual Machine (JVM)** used by Java to run their programs. JVM can be installed on almost every SW/HW combination.

- Where you can run JVM, you are able to run a Scala program!

Scala

- Scala provides an interactive shell.
- Once installed run scala on your terminal.
- It also supports IDE like Eclipse or Intellij.
- *For this course we will use **Visual Studio Code***

Dynamic Type Inference

- Scala can automatically figures out data types.
- It automatically “guess” the data type. Similar to Python, you don’t have to declare data types.

- Launch your interactive shell and try this command lines:
  - ```
    scala> 2
    val res0: Int = 2
    ```

- ***“Did you just type 2? Ok, 2 for me is an Integer”***
    ```

Dynamic Type Inference

Other examples:

- ```
  scala> 10.3
  val res1: Double = 10.3
  ```

- ```
  scala> 'a'
  val res2: Char = a
  ```

- ```
  scala> "test"
  val res3: String = test
  ```
  ```

Dynamic Type Inference

- You can also directly declare the datatype. Then Scala does not infer it by itself.
  - `scala > val z: Int = 100`

- This is a better practice. You are sure of the data type!

- Scala requires some time to infer the data type. By declaring them you save extra time (microseconds…)

Variables in Scala

- Two types of variables:
  - **Immutable Variables** -> declare them as *'val'*
    - Intuitively, "something you can't change". You can see val as a constant.
  - **Mutable Variables** -> declare them as *'var'*
    - They can change over time.

- Similar to Java *final* and *not final* variables.

Variables in Scala

- ```
  scala> val a = "test"
  val a: String = test

  scala> a = "anotherTest"
  ```
  //ERROR! reassignment to val ERROR!

- ```
  scala> var b = "Hadoop"
  var b: String = Hadoop

  scala> b = "spark is better"
  ```
  // OK! mutated b

Static Typing

- Mutable variables can change BUT they do not allow **type mismatch**:
  - ```
    scala> b = 100  //error: type mismatch;
                     found    : Int(100)
                     required: String
    ```

- You can change mutable variables BUT only accordingly to their original data type.
    ```

Using Scala with an IDE

- We suggest to use Visual Studio Code as default editor with Scala Extension installed

Run Scala applications

- Scala Objects are the basic container of everything. They are similar to Java Static Classes.
- Create the Main method (or extend App trait)
- Compile the file using *scalac*
- Run the Bytecode with *scala*

Scala: If-else

- ```
  object example1 {
      def main(args: Array[String]){
          var fruit = "apple" //you don't need
                               semicolons as python
          if (fruit == "pear") println("red")
          else println("not an apple")
      }
  }
  ```
  ```

Scala: While loop

-
```
object example2 {
    def main(args: Array[String]){
    var i = 10
        while(i>0){
            println("Number " + i)
            i = i-1
        }
    }
}
```

Scala: For loop

- ```
  object example3 {
      def main(args: Array[String]){
        for (i <- 1 to 10)
              println(i)
      }
  }
  ```

- "x to y" is called **range**. You can also specify the **step**.
  e.g.: `for (i <- 1 to 10 by 2)` // what is the output? Try it!
  ```

Scala: block expressions

Try this piece of code:

- ```
  var add = {
      var a = 10
      var b = 20
      a - b
      a + b
  }
  println(add)
  ```

What will be printed out?
  ```

Scala: block expressions

The code will print:

- `add: Int = 30`

In block expressions. The variable will be taking the value of the last expression. Be aware of that!

Scala: block expressions

Another example:

- ```
  val x = { println("foo"); 10}
  println("bar")
  println(x)
  ```

What these 3 lines of code will print out?
  ```

Scala: lazy variables

It prints out:

- ```
  foo
  bar
  10
  ```

//block expression or not, if you have an explicit print statement, then it will print!

//val x will be the last part of the block expression

Rewrite the program in this way:

- ```
  lazy val x = { println(“foo”); 10}
  println(“bar”)
  println(x)
  ```

Scala: lazy variables

New output:

- ```
  bar
  foo
  10
  ```

Lazy variable: With lazy declarations you can delay the initialization of a variable.

"Ehi Scala this variable is lazy. Don't run this line, execute the next line."

In the example above, Scala:

1. skip the lazy val (first line),
2. prints "bar" (second line)
3. and finally when you call the print statement on the lazy variable (third line), it initializes it.
  ```

Why Scala? (Functional Programming)

- Functional programming is a programming paradigm where programs are constructed by applying and composing functions.

- It is a declarative programming paradigm in which function definitions are trees of expressions that each return a value, rather than a sequence of imperative statements which change the state of the program.

Functional Programming

Functional Programming relies on **three key concepts**:

- Immutable Values
- Pure Functions
- Functions are values

Functional Programming

Functional Programming relies on **three key concepts**:

- Immutable Values
- Pure Functions
- Functions are values

Dipartimento di Ingegneria Informatica
Automatica e Gestionale Antonio Ruberti

Sapienza
Università di Roma

Scala Exercises

Exercise 1

Write a method *scalarProd* that, given two vectors represented as a *Seq* of *Double*, evaluates their scalar product.

If the lengths of vectrs are different, limit the dot product to the range of valid common indexes.

Example: scalarProd(Seq(3,4), Seq(2,9,1)) == 3*2 + 4*9 == 42.

Hint: use *math.min(...)*

Exercise 2

Write a method *isMappedFrom*, applicable for a Vector *v* that verify if another Vector *m* is obtainable from *v* applying the function *f* to each element of *v*.

So we wanto to have this:

v:Vector[T].isMappedFrom(m:Vector[U], f: T=>U): Boolean

Hint: You need to create an object with an *implicit class*

Exercise 3

Write a method *noobSort* that, given a *Vector v* of *n generic* elements, returns the ordered version of *v*.

Hint: Generate all the permutation of indexes from 0 *until* n and for each permutation generate the permutated vector, check if is ordered and eventually return it.

Core Hint: using implicitly[Ordering[U]] allow you to compare (*lt, gt, lteq, gteq*) *generic* elements (to use it you must define U as *Ordering -> noobSort[U: Ordering])*.

Exercise 4

Create a Scala construct named *repeat* that, given an *integer n* and a *body*, executes body for n times as in the followin example:

```
repeat(5) {
 println("test")
}
```

Why is this interesting? Do you ever heard about **call-by-name** parameters?

Processing Data

Spark Streaming

WordCount Example

```
import org.apache.spark.SparkContext
import org.apache.spark.SparkConf
object wordCount{
    def main(args: Array[String]) {
    /* configure spark application */
    val conf = new SparkConf().setAppName("Spark Scala WordCount
                                    Example")
      .setMaster("local[1]")//local : Run Spark locally with one worker
                        thread (i.e. 1 == no parallelism at all)
    /* spark context*/
    val sc = new SparkContext(conf)
    /* map */
    var map = sc.textFile("data/input.txt").flatMap(line => line.split("
                               ")).map(word => (word,1))
    //map.collect().foreach(println)
    /* reduce */
    var counts = map.reduceByKey(_ + _)
    /* print */
    counts.collect().foreach(println)
    /* or save the output to file */
    //counts.saveAsTextFile("out.txt")
    sc.stop()
    }
}
```

What is Spark Streaming?

Spark Streaming is an extension of the core Spark API that enables scalable, high-throughput, fault-tolerant stream processing of live data streams.

- **Batch Processing:** You have some data collected and stored somewhere (in a dataset, in a database…) and you want to perform some operations on that data.
- **Real Time Processing (Streaming):** You process data as soon as the data arrives. E.g.: In a banking system you want to detect data frauds in real time.

Processing Data

On the official website:

- **Spark Streaming** -> the original one. Queries over **RDD** data structures.
- **Structured Streaming** -> New Approach, still not so popular. Uses SQL queries over structures called **Dataframes**.

We will see both approaches starting from the first “classic” one.

Spark Streaming - Overview

Dipartimento di Ingegneria Informatica Automatica e Gestionale Antonio Ruberti
Sapienza Università di Roma

- Spark Streaming receives live input data streams and **divides the data into batches**, which are then processed by the Spark engine to generate the final stream of results in batches.

Spark Streaming

Data Ingestion

- Data Ingestion is an **optional** step to efficiently store data.
  1. Suppose you have to process **100 entries/min** -> Ok, you can avoid data ingestion and you can directly connect Spark to the data source/s.
  2. Now suppose you have **1M entries/min** -> Probably you need some Data Ingestion mechanism.
     E.g. “Analize tweets on Twitter to discover the trending actor of the week”
- You can use services like **Kafka** or **Flume**. These are services that store data for you.
- Data Ingestion is useful to avoid missing data. If Spark crashes for some reason, you can lose part of the data. By using an Ingestion mechanism (e.g. Kafka), it never fails!

Dstreams and Batch Interval

Dipartimento di Ingegneria Informatica
Automatica e Gestionale Antonio Ruberti
Sapienza Università di Roma

- Discretized Streams (Dstreams) is the basic abstraction provided by Spark Streaming. It represents a continuous stream of data, either the input data stream received from source, or the processed data stream generated by transforming the input stream.
- Each Dstream is divided into batches, i.e. it represents a sequence of data/RDDs from a certain interval. A RDD is the Spark's abstraction of an immutable, distributed dataset.
- Streaming Context Object: It is used to configure the streaming. With this object you can configure the Batch Interval.
  - E.g.: Batch Interval = 1 second. Meaning that you process data collected every second from the data source.

Configure the Streaming Context

- *E.g.: "I want to discover how many people are searching on Twitter for a new released game".*
- In this case, I don't need to collect the users' activity from Twitter every second, it's useless. Maybe every 30 minutes.

- Configure the streaming object depending on your needs (Scala):

```
import org.apache.spark._
import org.apache.spark.streaming._
import org.apache.spark.streaming.StreamingContext._ // not necessary since Spark 1.3

// Create a local StreamingContext with two working thread and batch interval of 1 second.
// The master requires 2 cores to prevent a starvation scenario.

val conf = new SparkConf().setMaster("local[2]").setAppName("NetworkWordCount")
val ssc = new StreamingContext(conf, Seconds(1))
```

About the Streaming Context

Dipartimento di Ingegneria Informatica Automatica e Gestionale Antonio Ruberti
Sapienza Università di Roma

Points to remember:

- Once a context has been started, no new streaming computations can be set up or added to it.
- Once a context has been stopped, it cannot be restarted.
- Only one StreamingContext can be active in a JVM at the same time.
- `stop()` on StreamingContext also stops the SparkContext. To stop only the StreamingContext, set the optional parameter of `stop()` called stopSparkContext to false.
- A SparkContext can be re-used to create multiple StreamingContexts, as long as the previous StreamingContext is stopped (without stopping the SparkContext) before the next StreamingContext is created.

For other tips, see the official docs here -> *https://spark.apache.org/docs/latest/streaming-programming-guide.html#discretized-streams-dstreams*

Dynamic Resource Allocation (DRA)

Dipartimento di Ingegneria Informatica Automatica e Gestionale Antonio Ruberti
Sapienza Università di Roma

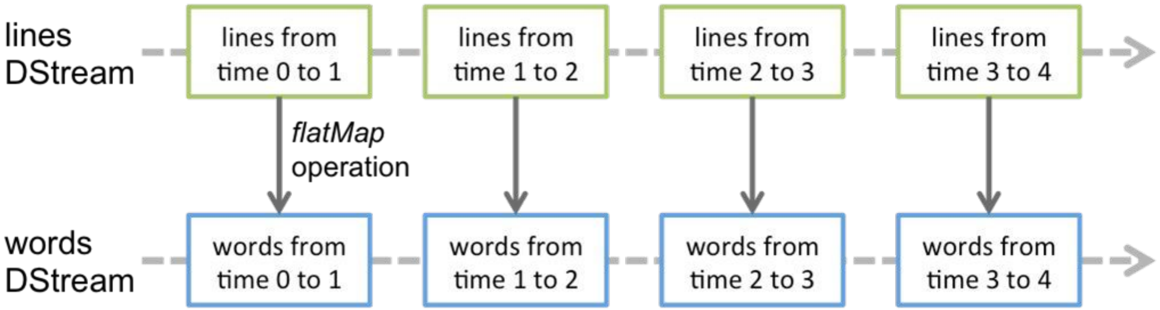
- In streaming, processing data may change over time. Then, you may need to adjust the allocated resources of the program.
- DRA automatically adjusts and scales resources (executors, RAM, CPU…).

- // enable DRA via Scala code

```
val conf = new SparkConf().setAppName("Spark dynamic
allocation demo")
.set("spark.dynamicAllocation.enabled", "true")
```

- If you are interested, you can see how DRA works in this video -> [VIDEO]

Transformations



- Any operation applied on a DStream translates these operations on the underlying RDDs. E.g., in the figure above the flatMap operation is applied on each RDD in the lines.
- These underlying RDD transformations are computed by the Spark engine. The DStream operations hide most of these details and provide the developer with a higher-level API for convenience.
- List of Transformations from official docs: https://spark.apache.org/docs/latest/streaming-programming-guide.html#transformations-on-dstreams

Resilient Distributed Datasets (RDDs)

Dipartimento di Ingegneria Informatica Automatica e Gestionale Antonio Ruberti
Sapienza Università di Roma

Spark revolves around the concept of a resilient distributed dataset (RDD), which is a fault-tolerant collection of elements that can be operated on in parallel. There are two ways to create RDDs: parallelizing an existing collection in your driver program, or referencing a dataset in an external storage system, such as a shared filesystem, HDFS, HBase, or any data source offering a Hadoop InputFormat.

- Parallelized collections are created by calling SparkContext's parallelize method on an existing collection.
  E.g.: Create a parallelized collection holding the numbers 1 to 5 in Scala:

```
val data = Array(1, 2, 3, 4, 5)
val distData = sc.parallelize(data)
```

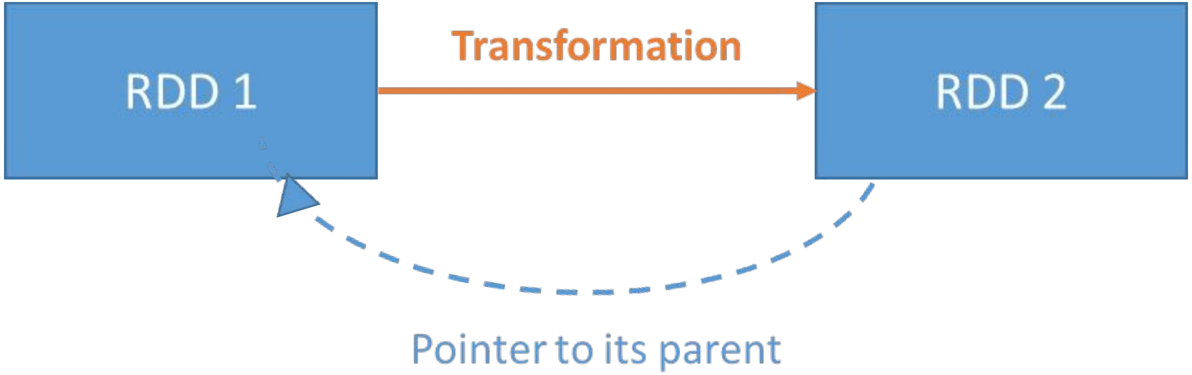
- External Datasets. Spark can create distributed datasets from any storage source. For example, Text file RDDs can be created using SparkContext's textFile method.

```
scala> val distFile = sc.textFile("data.txt")
distFile: org.apache.spark.rdd.RDD[String] = data.txt MapPartitionsRDD[10] at textFile at <console>:26
```

Resilient Distributed Datasets (RDDs)

Dipartimento di Ingegneria Informatica Automatica e Gestionale Antonio Ruberti
Sapienza Università di Roma

- Immutable collections of objects. They don’t change.
- A Transformation returns a new RDD, one or many.



1. map()

- Let’s see some examples via interactive shell.

```
spark-shell //launch the interactive shell
val x = sc.parallelize(List(“Spark”, “rdd”, “sample”))
// create an RDD
val y = x.map(x => (x,1)) //maps every entry of x to 1
y.collect   //to look the result
```

- **OUTPUT:**

```
res0: Array[(String, Int)] = Array((spark,1), (rdd,1),
(sample,1)) // new RDD y
```

2. flatMap()

- ```
  var z = sc.parallelize(List(1,2,3)).flatMap(x=>List(x,x,x))
  z.collect
  ```

  OUTPUT:
  ```
  res1: Array[Int] = Array(1, 1, 1, 2, 2, 2, 3, 3, 3)
  ```

What is the difference with map()? Let's see it:

- ```
  var z2 = sc.parallelize(List(1,2,3)).map(x=>List(x,x,x))
  z2.collect
  ```

  OUTPUT:
  ```
  res3: Array[List[Int]] = Array(List(1, 1, 1), List(2, 2, 2),
  List(3, 3, 3))
  ```

FlatMap creates one-dimensional RDDs.

3. filter()

- Returns odds and evens from a list of integers:

```
val numbers = sc.parallelize(List(1,2,3,4,5,6,7,8,9,10))
val evens = numbers.filter(_%2==0)
evens.collect
```

OUTPUT:

```
res4: Array[Int] = Array(2, 4, 6, 8, 10)
```

```
val odds = numbers.filter(_%2!=0)
odds.collect
```

OUTPUT:

```
res5: Array[Int] = Array(1, 3, 5, 7, 9)
```

4. union() and distinct()

- ```
  val num1 = sc.parallelize(List(1,2,3,4,5,6,7,8,9,10))
  val num2 = sc.parallelize(List(4,5,6,7,13,14,15))
  num1.union(num2).collect
  ```

  OUTPUT:
  ```
  res6: Array[Int] = Array(1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 4, 5, 6, 7,
  13, 14, 15)
  ```

- ```
  num1.union(num2).distinct.collect
  ```

  OUTPUT:
  ```
  res7: Array[Int] = Array(1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 13, 14, 15)
  ```
  //no duplicates

5. zip()

- ```
  val zip1 = sc.parallelize(List(1,2,3,4,5,6,7,8,9,10))
  val zip2 = sc.parallelize(List(11,12,13,14,15,16,17,18,19,20))

  val zipfinal = zip1 zip zip2
  zipfinal.collect
  ```

  OUTPUT:

  ```
  res10: Array[(Int, Int)] = Array((1,11), (2,12), (3,13), (4,14),
  (5,15), (6,16), (7,17), (8,18), (9,19), (10,20))
  ```
  ```

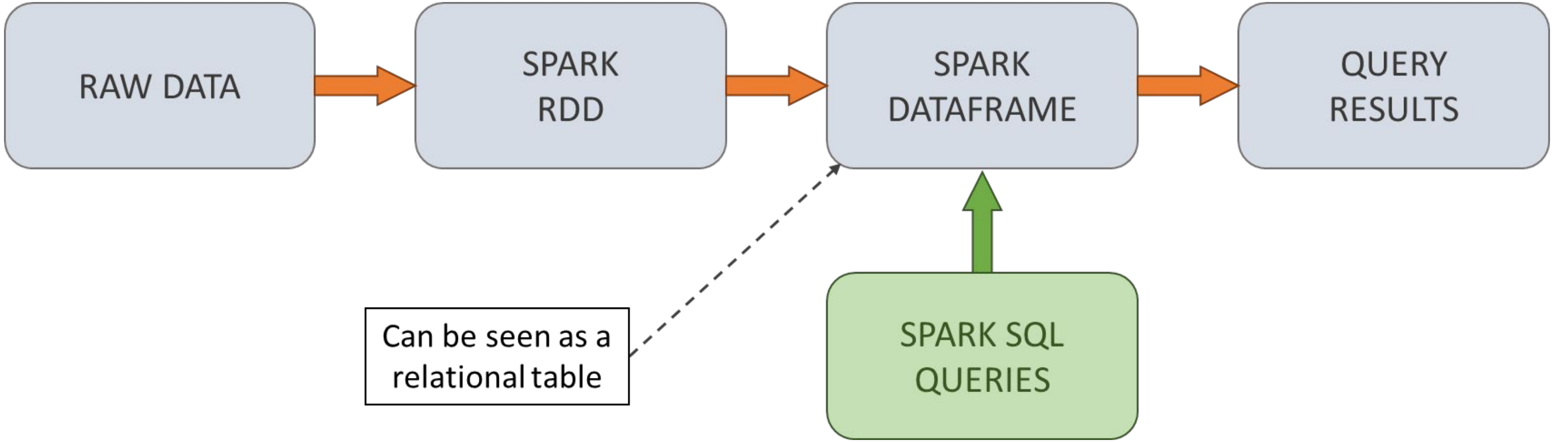
DATAFRAMES in Spark

- A DataFrame is a *Dataset* organized into named columns. It is conceptually equivalent to a table in a relational database.
- DataFrames can be constructed from a wide array of sources such as: structured data files, tables in Hive, external databases, or existing RDDs.

- E.g.: create a DataFrame from a CSV file:

```
val df = spark.read.json(“PATH_TO_CSV_FILE")
```

SQL Queries on data files



Example: Query data from a text file

DIPARTIMENTO DI INGEGNERIA INFORMATICA AUTOMATICA E GESTIONALE ANTONIO RUBERTI
SAPIENZA UNIVERSITÀ DI ROMA

E.g.: Search for word occurrences in a text file and then query over the results.

```
//read text file
val rdd = sc.textFile("PATH_TO_TEXTFILE")  -> returns a new RDD object

//perform transformations on RDD
val counts = rdd.flatMap(line => line.split("\\W+"))
                .map(word => (word,1))
                .reduceByKey(_+_)  -> returns a new RDD object (child)

//convert the RDD to a DataFrame
val df = counts.toDF()

//show it as table
df.show()
```

Pointer to its parent

Example: Query data from a text file

E.g.: Word occurrences in a text file

The table has _1 and _2 as column names. To change df's column names we have to create a new Dataframe (because they are immutable objects):

```
val df2 = df.toDF(Seq("Word","Count"): _*) //replace each column name with format _*

df2.show() -> now we have Word and Count as column names
```

Now we can perform queries on df2:

```
//order results by asc and show only top 5 elements
df2.orderBy(asc("Count")).show(5)

//show only words with more than 1 occurrence in the file
df2.filter("Count>1").show()
```

SQL query

Example: Taxi Dataset

The **yellow taxi trip records** include fields capturing pick-up and drop-off dates/times, pick-up and drop-off locations, trip distances, itemized fares, rate types, payment types, and driver-reported passenger counts. The data used in the attached datasets were collected and provided to the NYC Taxi and Limousine Commission.

What's in this Dataset?

- Rows = 112M
- Columns = 17
- Each row represents a Yellow Taxi Trip

You can download the dataset here -> [yellow.csv]

Example: Taxi Dataset

// read the csv file containing the dataset

```
val df = spark.read.format("csv").
     .option("header", true)
     .option("inferSchema", true)
     .load("PATH_TO_CSV_FILE")
```

configure
one or more
reading options

// create view on that file to refer to it as "taxidata"

```
df.createOrReplaceTempView("taxidata")
```

Example: Taxi Dataset

// count #rows in the dataset

```
spark.sql("select count(*) from taxidata") .show()
```

// get total revenue generated

```
spark.sql("select sum(total_amount) from taxidata").show()
```

// get average revenue in US Dollars:

```
spark.sql("select avg(total_amount) from taxidata").show()
```

Training a language classifier

DIPARTIMENTO DI INGEGNERIA INFORMATICA
AUTOMATICA E GESTIONALE ANTONIO RUBERTI
SAPIENZA
UNIVERSITÀ DI ROMA

Suppose we want to build a Spark's application for training a language classifier over tweets from Twitter.

1. **Collect a Dataset of Tweets -** Spark Streaming is used to collect a dataset of tweets and write them out to files.

2. **Examine the Tweets -** Spark SQL is used to examine the dataset of Tweets.

3. **Train a Model -** Then Spark MLlib is used to apply the K-Means algorithm to train a model on the data.

- If you are interested here you can find an example.