

Bigdata Assignment 6

Task 1

Write a simple program to show inheritance in scala.

Code:

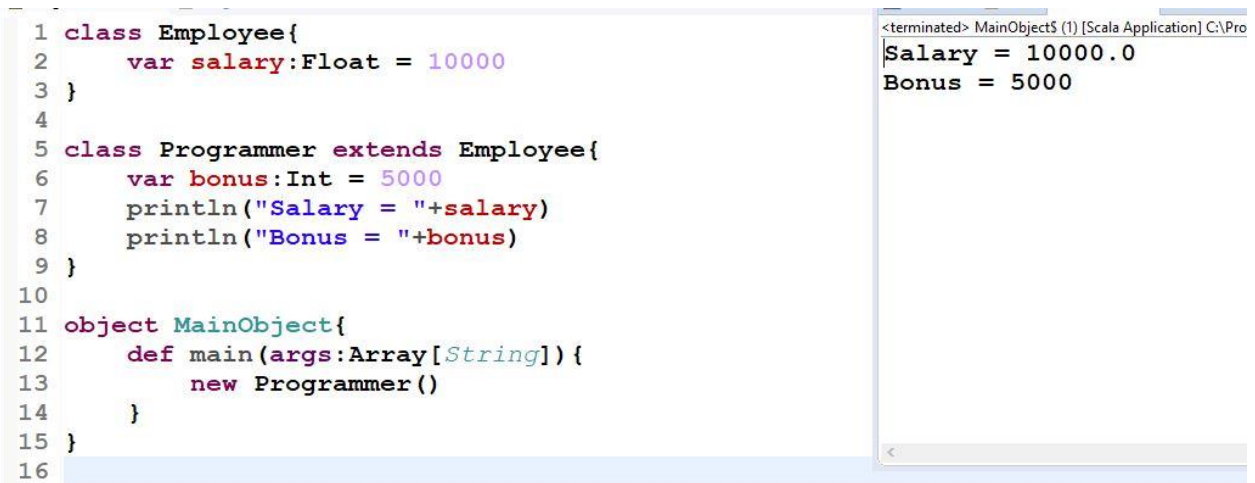
Scala Single Inheritance

```
class Employee{
    var salary:Float = 10000
}

class Programmer extends Employee{
    var bonus:Int = 5000
    println("Salary = "+salary)
    println("Bonus = "+bonus)
}

object MainObject{
    def main(args:Array[String]){
        new Programmer()
    }
}
```

Output:



```
1 class Employee{
2     var salary:Float = 10000
3 }
4
5 class Programmer extends Employee{
6     var bonus:Int = 5000
7     println("Salary = "+salary)
8     println("Bonus = "+bonus)
9 }
10
11 object MainObject{
12     def main(args:Array[String]){
13         new Programmer()
14     }
15 }
16
```

<terminated> MainObject\$ (1) [Scala Application] C:\Pro
Salary = 10000.0
Bonus = 5000

Task 2

Write a simple program to show multiple inheritance in scala

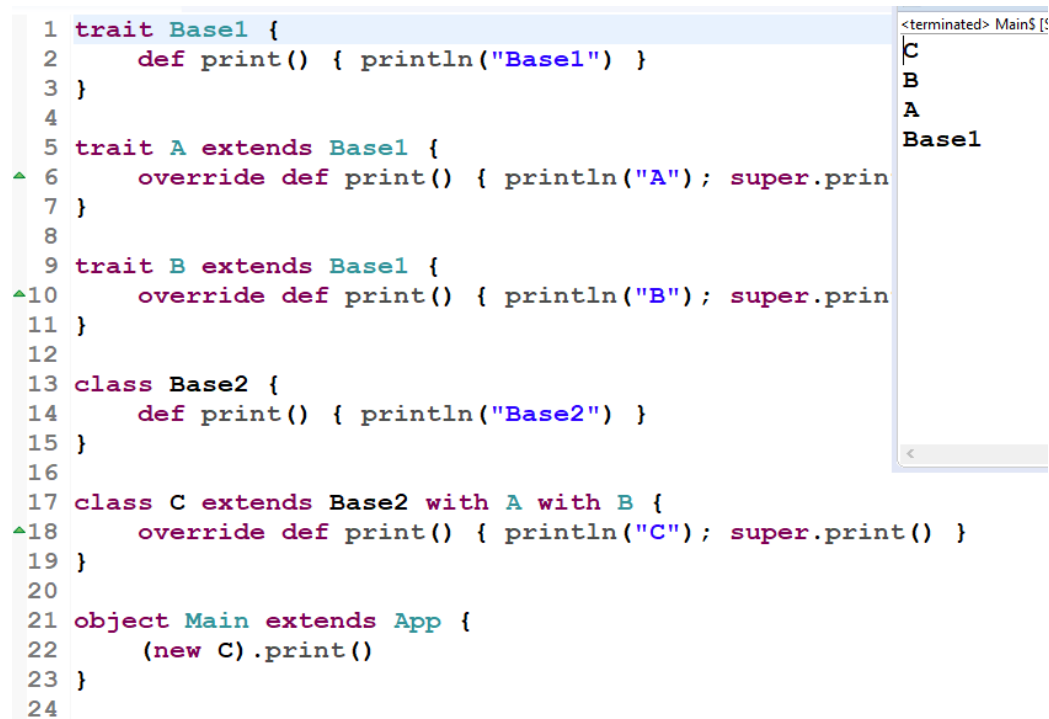
Code:

Bigdata Assignment 6

Multiple Inheritance in scala

```
trait Base1 {  
  def print() { println("Base1") }  
}  
  
trait A extends Base1 {  
  override def print() { println("A"); super.print() }  
}  
  
trait B extends Base1 {  
  override def print() { println("B"); super.print() }  
}  
  
class Base2 {  
  def print() { println("Base2") }  
}  
  
class C extends Base2 with A with B {  
  override def print() { println("C"); super.print() }  
}  
  
object Main extends App {  
  (new C).print()  
}
```

Output:



```
1 trait Base1 {  
2   def print() { println("Base1") }  
3 }  
4  
5 trait A extends Base1 {  
6   override def print() { println("A"); super.print() }  
7 }  
8  
9 trait B extends Base1 {  
10  override def print() { println("B"); super.print() }  
11 }  
12  
13 class Base2 {  
14   def print() { println("Base2") }  
15 }  
16  
17 class C extends Base2 with A with B {  
18   override def print() { println("C"); super.print() }  
19 }  
20  
21 object Main extends App {  
22   (new C).print()  
23 }  
24
```

<terminated> Main\$ [5]
C
B
A
Base1

Bigdata Assignment 6

Task 3

Write a partial function to add three numbers in which one number is constant and two numbers can be passed as inputs and define another method which can take the partial function as input and squares the result.

Code:

Task 4

Write a program to print the prices of 4 courses of Acadgild:

Android App Development -14,999 INR

Data Science - 49,999 INR

Big Data Hadoop & Spark Developer – 24,999 INR

Blockchain Certification – 49,999 INR

using match and add a default condition if the user enters any other course.

Code:

```
object coursePrice {  
  def main(args: Array[String]) {  
    println("Enter a number from 1 to 4 to know about the course fee")  
    val a=scala.io.StdIn.readInt()  
    println("The Fee for "+ matchTest(a))  
  }  
  
  def matchTest(x: Int): String = x match {  
    case 1 => " Android App Development -14,999 INR "  
    case 2 => " Data Science - 49,999 INR "  
    case 3 => " Big Data Hadoop & Spark Developer – 24,999 INR "  
    case 4 => " Blockchain Certification – 49,999 INR "  
    case _ => "Option Not found"  
  }  
}
```

Output:

```
Enter a number from 1 to 4 to know about the course fee
```

```
3
```

```
The Fee for  Big Data Hadoop & Spark Developer – 24,999 INR
```

Task 5

Bigdata Assignment 6

Create a calculator to work with rational numbers.

Requirements:

- It should provide capability to add, subtract, divide and multiply rational Numbers
- Create a method to compute GCD (this will come in handy during operations on rational)

Add option to work with whole numbers which are also rational numbers i.e. $(n/1)$

- achieve the above using auxiliary constructors

enable method overloading to enable each function to work with numbers and rational

Code:

```
import scala.util.parsing.combinator._

object Calculator extends JavaTokenParsers {

  def expr : Parser[Double] = term ~ rep (("+" | "-") ~ term) ^^ eval

  def term : Parser[Double] = factor ~ rep(("*" | "/" ) ~ term) ^^ eval

  def factor : Parser[Double] = ( floatingPointNumber ^^ (_.toDouble)
    | "(" ~> expr <~ ")"
    )

  def eval:Double ~ List[String ~ Double] => Double = {
    case v ~ list => (v /: list) {
      case (op1, "+" ~ op2) => op1 + op2
      case (op1, "-" ~ op2) => op1 - op2
      case (op1, "*" ~ op2) => op1 * op2
      case (op1, "/" ~ op2) => op1 / op2
    }
  }

  def main(args:Array[String]) {
    println(parseAll(expr, args(0)))
  }
}
```

Bigdata Assignment 6

Task 6

Given a list of numbers - List[Int] (1, 2, 3, 4, 5, 6, 7, 8, 9, 10)

- find the sum of all numbers
- find the total elements in the list
- calculate the average of the numbers in the list
- find the sum of all the even numbers in the list
- find the total number of elements in the list divisible by both 5 and 3

Code:

```
object listOperation{
  def main(args:Array[String]){
    val list = List[Int] (1, 2, 3, 4, 5, 6, 7, 8, 9, 10)
    println("The Sum of list is: ")
    print(list.sum)
    println()
    println("Total elements in the list: ")
    print(list.length)
    println()
    println("The Average of the numbers in the list is: ")
    print((list.sum)/(list.length))
    println()
    println("The Sum of all the even numbers in the list: ")
    print(list.filter(_%2==0).sum)
    println()
    println("The list of numbers divisible by both 3 and 5: ")
    print(list.filter(t=>(t%3==0)|| (t%5==0)))
    println()
  }
}
```

Output:

```
The Sum of list is:
55
Total elements in the list:
10
The Average of the numbers in the list is:
5
The Sum of all the even numbers in the list:
```

Bigdata Assignment 6

30

The list of numbers divisible by both 3 and 5:

```
List(3, 5, 6, 9, 10)
```

Task 7

1) Pen down the limitations of MapReduce.

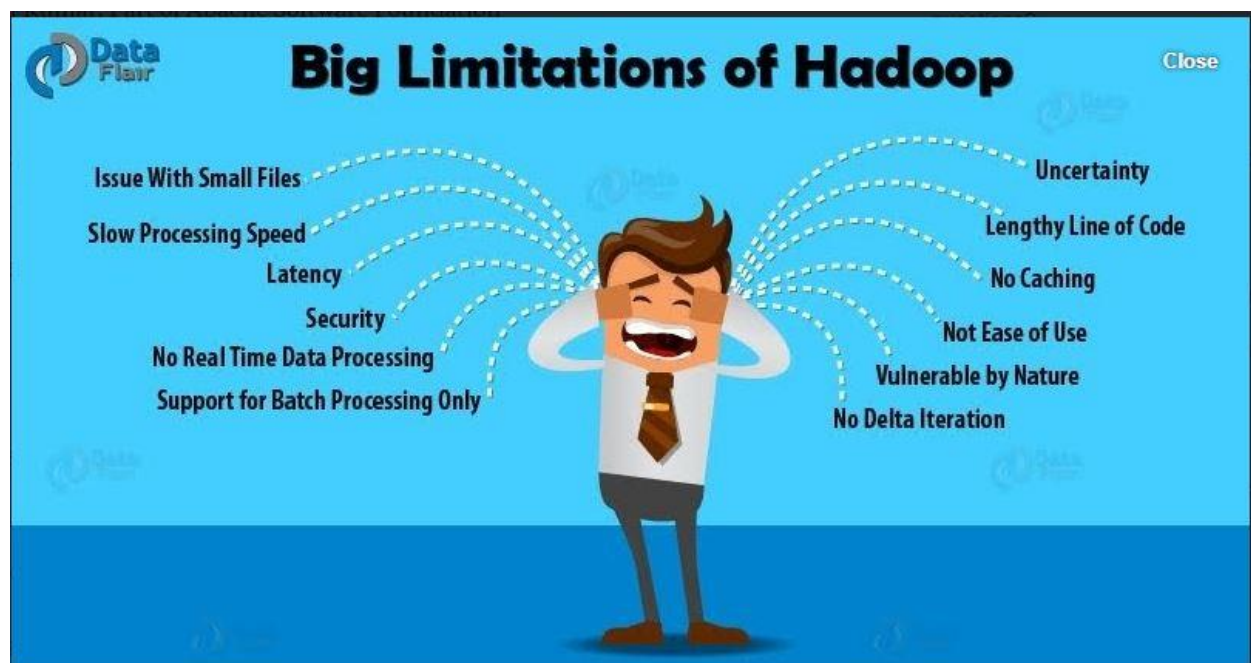
Ans: Here are some of the limitations of MapReduce:

While development of Hadoop's MapReduce the vision was pretty limited, it was developed just to handle 1 problem: **Batch processing**.

MapReduce cannot handle:

1. Interactive Processing
2. Real-time (stream) Processing
3. Iterative (delta) Processing
4. In-memory Processing
5. Graph Processing

Let's discuss the limitations in great details:



1. Issue with Small Files

Hadoop is not suited for small data. (HDFS) Hadoop Distributed File System lacks the ability to efficiently support the random reading of small files because of its high capacity design.

2. Slow Processing Speed

Bigdata Assignment 6

In Hadoop, with a parallel and distributed algorithm, MapReduce process large data sets. There are tasks that need to be performed: Map and Reduce and, MapReduce requires a lot of time to perform these tasks thereby increasing latency. Data is distributed and processed over the cluster in MapReduce which increases the time and reduces processing speed.

3. Support for Batch Processing only

Hadoop supports batch processing only, it does not process streamed data, and hence overall performance is slower. MapReduce framework of Hadoop does not leverage the memory of the Hadoop Cluster to the maximum.

4. No Real-time Data Processing

Apache Hadoop is designed for batch processing, that means it take a huge amount of data in input, process it and produce the result. Although batch processing is very efficient for processing a high volume of data, but depending on the size of the data being processed and computational power of the system, an output can be delayed significantly. Hadoop is not suitable for Real-time data processing.

5. No Delta Iteration

Hadoop is not so efficient for iterative processing, as Hadoop does not support cyclic data flow(i.e. a chain of stages in which each output of the previous stage is the input to the next stage).

6. Latency

In Hadoop, MapReduce framework is comparatively slower, since it is designed to support different format, structure and huge volume of data. In **MapReduce**, Map takes a set of data and converts it into another set of data, where individual element are broken down into key value pair and Reduce takes the output from the map as input and process further and MapReduce requires a lot of time to perform these tasks thereby increasing latency.

7. Not Easy to Use

In Hadoop, MapReduce developers need to hand code for each and every operation which makes it very difficult to work. MapReduce has no interactive mode, but adding one such as Hive and Pig makes working with MapReduce a little easier for adopters.

8. No Caching

Hadoop is not efficient for caching. In Hadoop, MapReduce cannot cache the intermediate data in memory for a further requirement which diminishes the performance of Hadoop.

2) What is RDD? Explain few features of RDD?

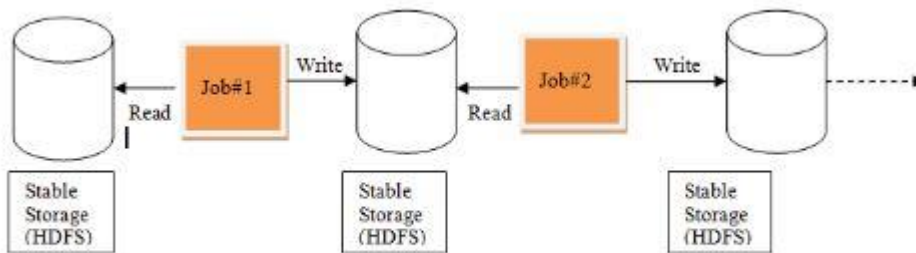
Bigdata Assignment 6

Ans: Apache Spark has already over taken Hadoop (MapReduce) in general, because of variety of benefits it provides in terms of faster execution in iterative processing algorithms such as Machine learning.

Now let's try to understand what makes spark RDDs so useful in batch analytics.

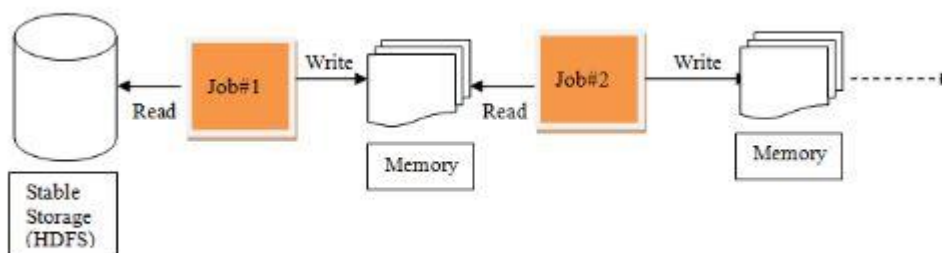
When it comes to iterative distributed computing, i.e. processing data over multiple jobs in computations such as Logistic Regression, K-means clustering, Page rank algorithms, it is fairly common to reuse or share the data among multiple jobs or you may want to do multiple ad-hoc queries over a shared data set.

There is an underlying problem with data reuse or data sharing in existing distributed computing systems (such as MapReduce) and that is , you need to store data in some intermediate stable distributed store such as HDFS or Amazon S3. This makes the overall computations of jobs slower since it involves multiple IO operations, replications and serializations in the process.



Iterative Processing in MapReduce

RDDs , tries to solve these problems by enabling fault tolerant distributed In-memory computations.



Iterative Processing in Spark

Now, let's understand what exactly RDD is and how it achieves fault tolerance –

Bigdata Assignment 6

RDD – Resilient Distributed Datasets

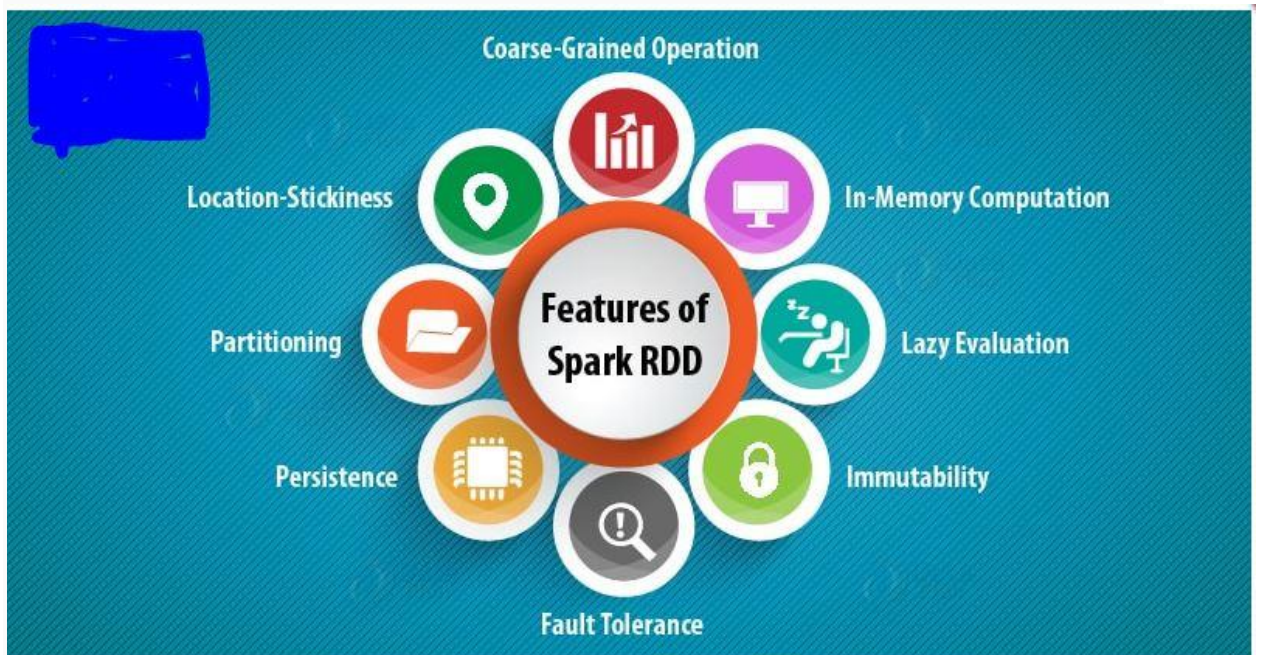
RDDs are Immutable and partitioned collection of records, which can only be created by *coarse grained operations* such as map, filter, group by etc. By coarse grained operations, it means that the operations are applied on all elements in a datasets. RDDs can only be created by reading data from a stable storage such as HDFS or by transformations on existing RDDs.

Features:

Several features of Apache Spark RDD are:

1. In-memory Computation

Spark RDDs have a provision of **in-memory computation**. It stores intermediate results in distributed memory(RAM) instead of stable storage(disk).



2. Lazy Evaluations

All transformations in Apache Spark are lazy, in that they do not compute their results right away. Instead, they just remember the transformations applied to some base data set.

Spark computes transformations when an action requires a result for the driver program. Follow this guide for the deep study of **Spark Lazy Evaluation**.

3. Fault Tolerance

Bigdata Assignment 6

Spark RDDs are fault tolerant as they track data lineage information to rebuild lost data automatically on failure. They rebuild lost data on failure using lineage, each RDD remembers how it was created from other datasets (by transformations like a map, join or groupBy) to recreate itself. Follow this guide for the deep study of **RDD Fault Tolerance**.

4. **Immutability**

Data is safe to share across processes. It can also be created or retrieved anytime which makes caching, sharing & replication easy. Thus, it is a way to reach consistency in computations.

5. **Partitioning**

Partitioning is the fundamental unit of parallelism in Spark RDD. Each partition is one logical division of data which is mutable. One can create a partition through some transformations on existing partitions.

6. **Persistence**

Users can state which RDDs they will reuse and choose a storage strategy for them (e.g., in-memory storage or on Disk).

7. **Coarse-grained Operations**

It applies to all elements in datasets through maps or filter or group by operation.

8. **Location-Stickiness**

RDDs are capable of defining placement preference to compute partitions. Placement preference refers to information about the location of RDD. The **DAGScheduler** places the partitions in such a way that task is close to data as much as possible. Thus, speed up computation.

3) List down few Spark RDD operations and explain each of them.

Ans: RDD in Apache Spark supports two types of operations:

- Transformation
- Actions

- Transformations

Spark **RDD Transformations** are *functions* that take an RDD as the input and produce one or many RDDs as the output. They do not change the input RDD (since RDDs are

Bigdata Assignment 6

immutable and hence one cannot change it), but always produce one or more new RDDs by applying the computations they represent e.g. `Map()`, `filter()`, `reduceByKey()` etc.

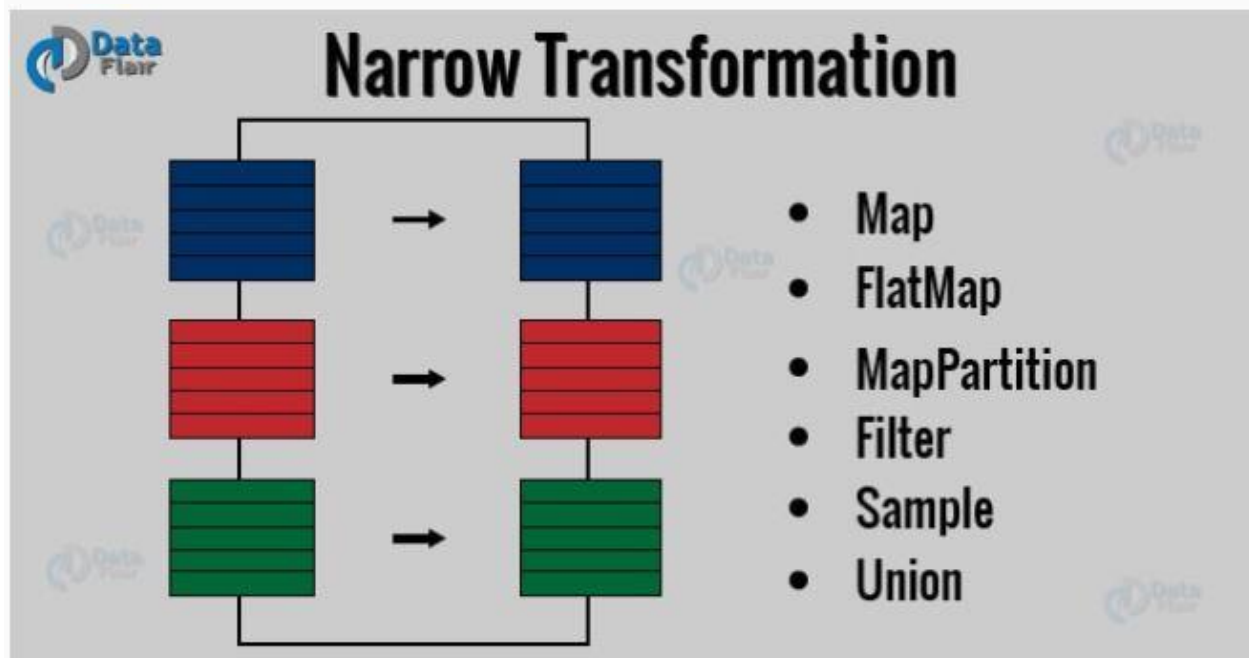
Transformations are **lazy** operations on an RDD in Apache Spark. It creates one or many new RDDs, which executes when an Action occurs. Hence, Transformation creates a new dataset from an existing one.

Certain transformations can be pipelined which is an optimization method, that Spark uses to improve the performance of computations. There are two kinds of transformations: narrow transformation, wide transformation.

1. Narrow Transformations

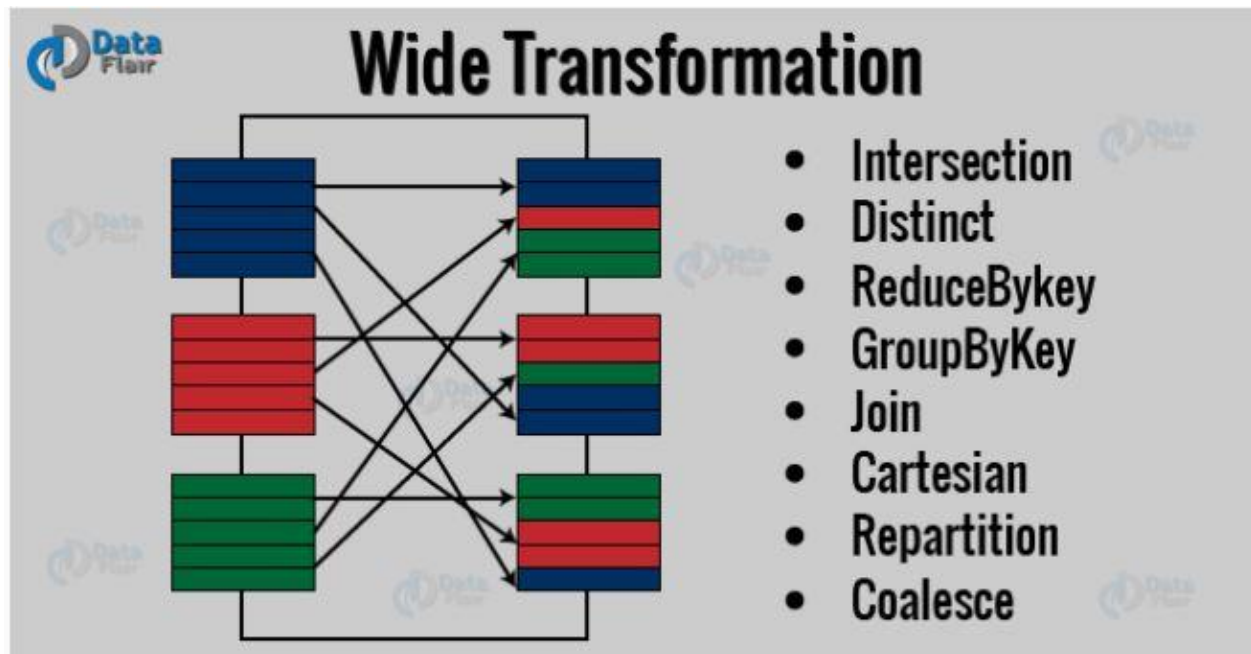
It is the result of `map`, `filter` and such that the data is from a single partition only, i.e. it is self-sufficient. An output RDD has partitions with records that originate from a single partition in the parent RDD. Only a limited subset of partitions used to calculate the result.

Spark groups narrow transformations as a stage known as **pipelining**.



2. Wide Transformations

It is the result of `groupByKey()` and `reduceByKey()` like functions. The data required to compute the records in a single partition may live in many partitions of the parent RDD. Wide transformations are also known as *shuffle transformations* because they may or may not depend on a shuffle.



- Actions

An **Action** in Spark returns final result of RDD computations. It triggers execution using lineage graph to load the data into original RDD, carry out all intermediate transformations and return final results to Driver program or write it out to file system. Lineage graph is dependency graph of all parallel RDDs of RDD.

Actions are RDD operations that produce non-RDD values. They materialize a value in a Spark program. An Action is one of the ways to send result from executors to the driver. First(), take(), reduce(), collect(), the count() is some of the Actions in spark.

Using transformations, one can create RDD from the existing one. But when we want to work with the actual dataset, at that point we use Action. When the Action occurs it does not create the new RDD, unlike transformation. Thus, actions are RDD operations that give no RDD values. Action stores its value either to drivers or to the external storage system. It brings laziness of RDD into motion.