Scala, Accumulators, DataFrames and Datasets

Plan

- Scala
    - k-means in Spark Scala
- Spark ML pipelines
- Decision trees in Spark
- Gradient descendant:
    - from batch to ADAGRAD
    - implementation in Spark RDD
    - application to linear regression
- TensorFlow

Plan for Scala

- The basics
- Control structures and functions
- Collections
- Case classes
- Structure of a Scala program

Interactive mode - the Scala interpreter

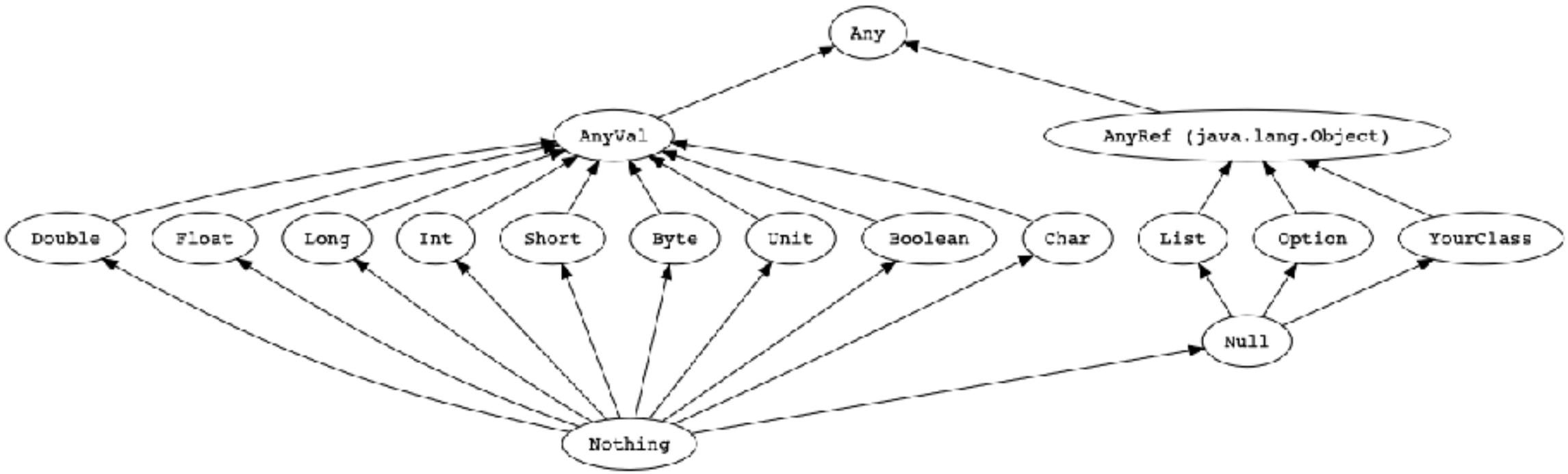
- Instructions for installing Scala can be found here http://horstmann.com/scala/install/

- This link is related to the book I am referring in these slides, considered by Martin Odersky as the best book for quickly learning the essentials of Scala

Interactive mode - the Scala interpreter

```
scala> 8 * 5 + 2
res0: Int = 42

scala> 0.5 * res0
res57: Double = 21.0


scala> "Hello, " + res0
res58: String = Hello, 42


scala> 1 to 10
res0: scala.collection.immutable.Range.Inclusive = Range 1 to 10

scala>
```

Scala types



Scala Type Hierarchy

https://docs.scala-lang.org/tour/unified-types.html

Focus on range collections

```
scala> 1 to 10
res1: scala.collection.immutable.Range.Inclusive = Range 1 to 10

scala> 1 until 10
res2: scala.collection.immutable.Range = Range 1 until 10
```

```
scala> 1 to 10 by 2
res3: scala.collection.immutable.Range = inexact Range 1 to 10 by 2

scala> 'a' to 'c'
res4: scala.collection.immutable.NumericRange.Inclusive[Char] = NumericRange a to c

scala> 'a' until 'c'
res5: scala.collection.immutable.NumericRange.Exclusive[Char] =
NumericRange a until c

scala>
```

Range for populating sequences

```
scala> val x = (1 to 10).toList
x: List[Int] = List(1, 2, 3, 4, 5, 6, 7, 8, 9, 10)
```

```
scala>  val x = (1 to 10).toArray
x: Array[Int] = Array(1, 2, 3, 4, 5, 6, 7, 8, 9, 10)

scala> val x = (1 to 10).toSet
x: scala.collection.immutable.Set[Int] = Set(5, 10, 1, 6, 9, 2, 7, 3, 8, 4)

scala>
```

Values and Variables

```
scala> val answer = 8 * 5 + 2
answer: Int = 42

scala>  0.5 * answer
res59: Double = 21.0

scala> answer = 0
<console>:13: error: reassignment to val
       answer = 0
              ^

scala> var counter = 0
counter: Int = 0

scala> counter = 1
counter: Int = 1

scala>
```

- Names can be either of the val or variable kind
- Val is used for names associated to constants
- While variable is for names whose value may change
- Suggestion: use as much as possible val names in programs.

Control stucures and functions

- Differently from Java and C++, in Scala almost all constructs have a value
- For instance
  - An if-expression has a value
  - A { ; …; ….;… } block has a value, the value of the last expression in the block

Conditional expressions

```
scala> val x=1
x: Int = 1

scala> if (x > 0) 1 else -1
res1: Int = 1

scala> val s = if (x > 0) 1 else -1
s: Int = 1

scala>
```

- The value of the if-expression is the value of the expression following if or else

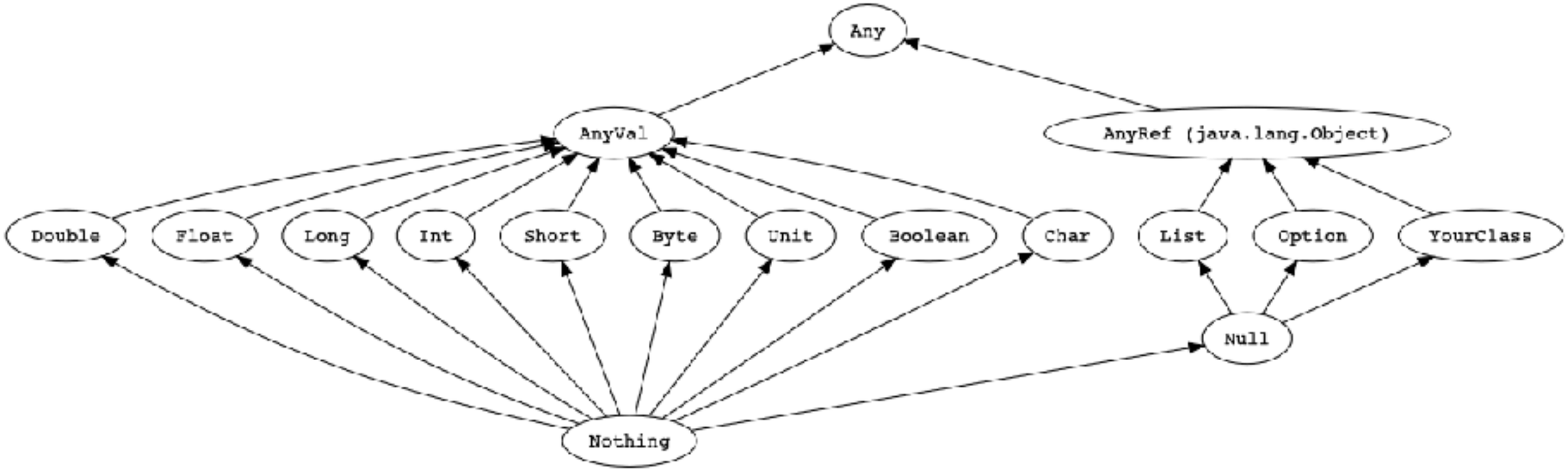
- It follows that the if epression has a type, the type of its value

- What happens if the if-expression may yeld values of different types ?

Conditional expressions

```
scala> if (x > 0) "positive" else -1
res2: Any = positive

scala>
```

- The type of the if-expression is Any, which is the *common* super-type of all types

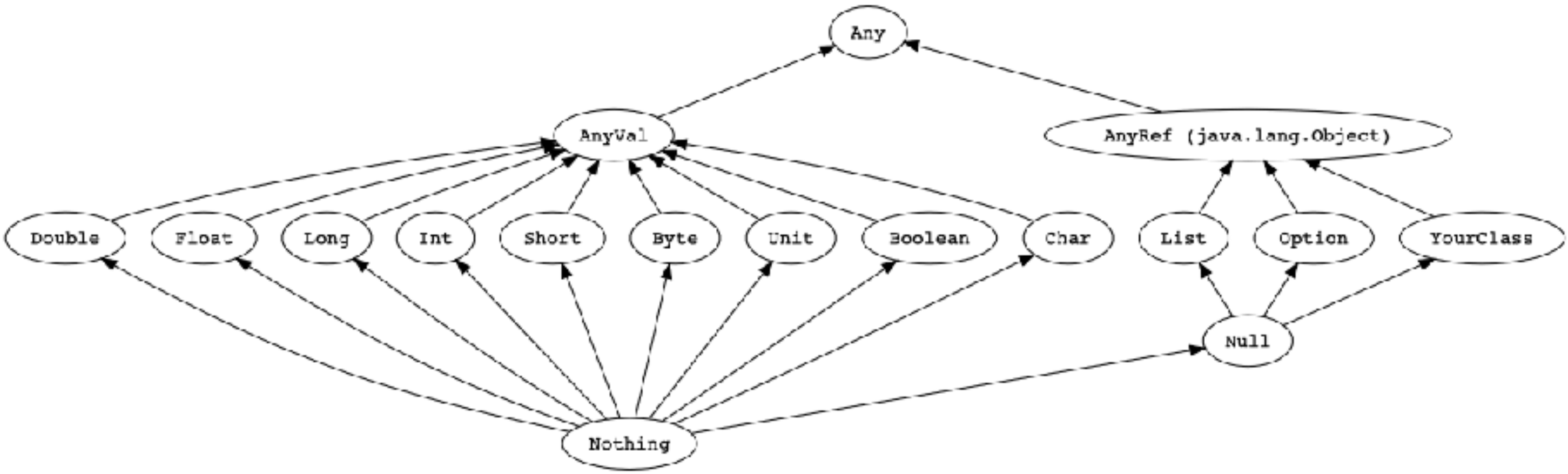


Scala Type Hierarchy

Conditional expressions

```
scala> var y=0
y: Int = 0

scala> if (y > 0) 1
res4: AnyVal = ()

scala> y=1
y: Int = 1

scala>  if (y > 0) 1
res5: AnyVal = 1
```

- An if-expression without the else-branch can yeld no value
- In this case Scala assume that a value is returned which is ()
- () denotes ‘no value’ and its type is Unit.



Complex branches

```
scala> if (n > 0) { r = r * n; n -= 1 }

scala> if (n > 0) {
     |   r = r * n
     |   n -= 1
     | }

scala>
```

- Many Scala programmers prefer the second style

Block expressions and assignements

```
scala> { r = r * n; n -= 1 }

scala> var c = { r = r * n; n -= 1 }
c: Unit = ()

scala>

scala> var x0,y0=0.1
x0: Double = 0.1
y0: Double = 0.1

scala> { val dx = x - x0; val dy = y - y0; sqrt(dx * dx + dy * dy) }
res40: Double = 1.2727922061357855

scala>
```

- The value/type of a block expression is the value/type of its last expression

Input and output

```
scala> val name = scala.io.StdIn.readLine("Your name: ")
Your name: name: String = toto

scala> print("Your age: ") ; val age = io.StdIn.readInt()
Your age: age: Int = 9

scala> printf("Hello, %s! Next year, you will be %d.\n", name, age + 1)
```

To read a numeric, Boolean, or character value, use **readInt, readDouble, readByte, readShort, readLong, readFloat, readBoolean,** or **readChar.**

Only readline takes a prompt string (e.g., "Your name: “)

Loops

```
…… n and r are initialized …

scala> while (n > 0) { r=r* n
     | n -= 1
     | }


scala> for (i <- 1 to n)
     | r=r* i
```

- In case of for the variable does not need to be initialized, differently from the while case
- For other advanced uses see the documentation
  https://docs.scala-lang.org/tour/for-comprehensions.html

```
scala> val s = "Hello"
s: String = Hello

scala> var sum = 0
sum: Int = 0

scala> for (i <- 0 until s.length) // Last value for i is s.length - 1
     | sum += s(i)

scala>
```

Functions

```
scala> def abs(x: Double) = if (x >= 0) x else -x
abs: (x: Double)Double

scala> def fac(n : Int) = {
     | var r = 1
     | for (i <- 1 to n) r = r * i
     | r
     | }
fac: (n: Int)Int

scala>
```

```
scala> def fac(n: Int):Int =
     | if (n <= 0) 1 else n * fac(n - 1)
fac: (n: Int)Int

scala>
```

- Scala has functions in addition to methods.
- A method operates on an object, but a function doesn't
- While types of input parameters must be declared, the output type could not
- With the exception of recursive functions

Pattern matching

```
scala> def fact(n: Int): Int = n match {
     |   case 0 => 1
     |   case n => n * fact(n - 1)
     | }
fact: (n: Int)Int
```

Pattern matching

```
scala> def f(x: Any): String = x match {
     |   case i:Int => "integer: " + i
     |   case _:Double => "a double"
     |   case s:String => "I want to say " + s
     |   case _: Any => "I do not know"}
f: (x: Any)String

scala> f(3)
res2: String = integer: 3

scala> f(())
res3: String = I do not know

scala> f(8.6)
res4: String = a double

scala> f("Hey")
res5: String = I want to say Hey

scala>
```

Arrays

```
scala> val nums = new Array[Int](10)
nums: Array[Int] = Array(0, 0, 0, 0, 0, 0, 0, 0, 0, 0)

scala> val a = new Array[String](10)
a: Array[String] = Array(null, null, null, null, null,
null, null, null, null, null)

scala> val s = Array("Hello", "World")
s: Array[String] = Array(Hello, World)
```

```
scala> s(0) = "Goodbye"

scala> s
res15: Array[String] = Array(Goodbye, World)


scala>
```

- They can be either of fixed or variable lenght
- These examples relate to fixed-legnth arrays

Variable-Length Arrays

```
scala> import scala.collection.mutable.ArrayBuffer

scala> val b = ArrayBuffer[Int]()
b: scala.collection.mutable.ArrayBuffer[Int] = ArrayBuffer()

scala> b += 1
res17: b.type = ArrayBuffer(1)

scala> b += (1, 2, 3, 5)
res18: b.type = ArrayBuffer(1, 1, 2, 3, 5)

scala> b ++= Array(8, 13, 21)
res19: b.type = ArrayBuffer(1, 1, 2, 3, 5, 8, 13, 21)

scala> b.trimEnd(5)

scala> b
res27: scala.collection.mutable.ArrayBuffer[Int] = ArrayBuffer(1, 1, 2)

scala>
```

Traversing Arrays

```
scala> b ++= Array(8, 13, 21)
res19: b.type = ArrayBuffer(1, 1, 2, 3, 5, 8, 13, 21)

scala> b.trimEnd(5)


scala> for (i <- 0 until b.length)
     | println(i + ": " + b(i))
0: 1
1: 1
2: 2

scala> for (elem <- b)
     | println(elem)
1
1
2

scala>
```

Transforming Arrays

```
scala> b ++= Array(8, 13, 21)
res19: b.type = ArrayBuffer(1, 1, 2, 3, 5, 8, 13, 21)

scala> b.trimEnd(5)


scala> for (elem <- b if elem % 2 == 0) yield 2 * elem
res25: scala.collection.mutable.ArrayBuffer[Int] = ArrayBuffer(4)

scala> b.filter(x => x % 2 == 0 ).map(x => 2 * x)
res26: scala.collection.mutable.ArrayBuffer[Int] = ArrayBuffer(4)


scala> b.filter(_ % 2 == 0).map(2 * _)
res27: scala.collection.mutable.ArrayBuffer[Int] = ArrayBuffer(4)

scala>
```

Tuples

```
scala> val t = (1, 3.14, "Fred")
t: (Int, Double, String) = (1,3.14,Fred)


scala> t._1
res49: Int = 1

scala> val (first, second, third) = t
first: Int = 1
second: Double = 3.14
third: String = Fred

scala> val (first, second, _) = t
first: Int = 1
second: Double = 3.14
```

Case classes

```
scala> case class Message(sender: String, recipient: String, body: String)
defined class Message

scala> val message1 = Message("guillaume@quebec.ca", "jorge@catalonia.es", "Ça va ?")
message1: Message = Message(guillaume@quebec.ca,jorge@catalonia.es,Ça va ?)

scala> println(message1.sender)
guillaume@quebec.ca

scala> val message4 = Message("julien@bretagne.fr", "travis@washington.us", "Me zo o komz gant
ma amezeg")
message4: Message = Message(julien@bretagne.fr,travis@washington.us,Me zo o komz gant ma
amezeg)
```

```
scala> val message5 = message4.copy(sender = message4.recipient, recipient =
"claire@bourgogne.fr")
message5: Message = Message(travis@washington.us,claire@bourgogne.fr, Me zo o komz gant ma
amezeg)

scala> message5.sender
res53: String = travis@washington.us

scala> message5.recipient
res54: String = claire@bourgogne.fr

scala> message5.body
res55: String = Me zo o komz gant ma amezeg

scala>
```

Pattern matching on case classes

```scala
abstract class Notification

case class Email(sender: String, title: String, body: String) extends Notification

case class SMS(caller: String, message: String) extends Notification

case class VoiceRecording(contactName: String, link: String) extends Notification


def showNotification(notification: Notification): String = {
  notification match {
    case Email(sender, title, _) =>
      s"You got an email from $sender with title: $title"
    case SMS(number, message) =>
      s"You got an SMS from $number! Message: $message"
    case VoiceRecording(name, link) =>
      s"You received a Voice Recording from $name! Click the link to hear it: $link"
  }
}
val someSms = SMS("12345", "Are you there?")
val someVoiceRecording = VoiceRecording("Tom", "voicerecording.org/id/123")

println(showNotification(someSms))  // prints You got an SMS from 12345! Message: Are you there?

println(showNotification(someVoiceRecording))  // prints You received a Voice Recording from Tom! Click the link to hear it:
voicerecording.org/id/123
```

https://docs.scala-lang.org/tour/pattern-matching.html

Hello, world!

Each Scala program must start with an object's main method of type:

Array[String] => Unit

```
object Hello {
  def main(args: Array[String]) {
    println("Hello, World!")
  }
}
```

Hello, world!++

```
object Hello extends App {
  if (args.length > 0)
    println("Hello, " + args(0))
  else
    println("Hello, World!")
}
```

```
$ scalac Hello.scala
$ scala -Dscala.time Hello Fred
Hello, Fred
[total 4ms]
```

What to use in place of numpy?

- You can rely on Nd4J

- Used in this nice post presenting a Scala implementation of linear regression

https://www.cpuheater.com/scala/machine-learning-scala-linear-regression/

Much more in the following references

- Scala for the impatient. *Cay Horstmann*
- Programming in Scala, 3rd Edition. *Martin Odersky et al.*
- Scala Cook book, Recipes for Object-Oriented and Functional Programming By *Alvin Alexander*

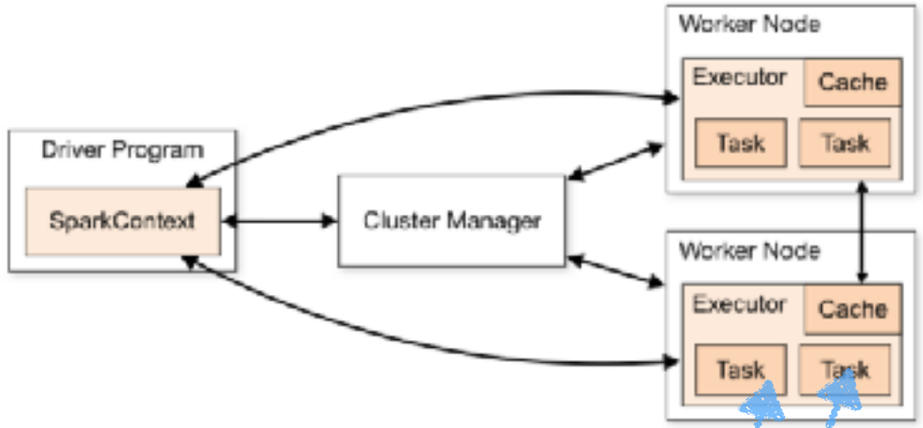
Spark

Accumulators

needed in the GCC project

Accumulators

Aggregating values from worker nodes back to the driver program.



Accumulators

Accumulable	Value
counter	45

Tasks

Index ▲	ID	Attempt	Status	Locality Level	Executor ID / Host	Launch Time	Duration	GC Time	Accumulators	Errors
0	0	0	SUCCESS	PROCESS_LOCAL	driver / localhost	2016/04/21 10:10:41	17 ms			
1	1	0	SUCCESS	PROCESS_LOCAL	driver / localhost	2016/04/21 10:10:41	17 ms		counter: 1	
2	2	0	SUCCESS	PROCESS_LOCAL	driver / localhost	2016/04/21 10:10:41	17 ms		counter: 2	
3	3	0	SUCCESS	PROCESS_LOCAL	driver / localhost	2016/04/21 10:10:41	17 ms		counter: 7	
4	4	0	SUCCESS	PROCESS_LOCAL	driver / localhost	2016/04/21 10:10:41	17 ms		counter: 5	
5	5	0	SUCCESS	PROCESS_LOCAL	driver / localhost	2016/04/21 10:10:41	17 ms		counter: 6	
6	6	0	SUCCESS	PROCESS_LOCAL	driver / localhost	2016/04/21 10:10:41	17 ms		counter: 7	
7	7	0	SUCCESS	PROCESS_LOCAL	driver / localhost	2016/04/21 10:10:41	17 ms		counter: 17	

Demo on DataBricks notebook

Some more practice with Scala and RDD

Spark

Data frames, SQL.

Dario Colazzo

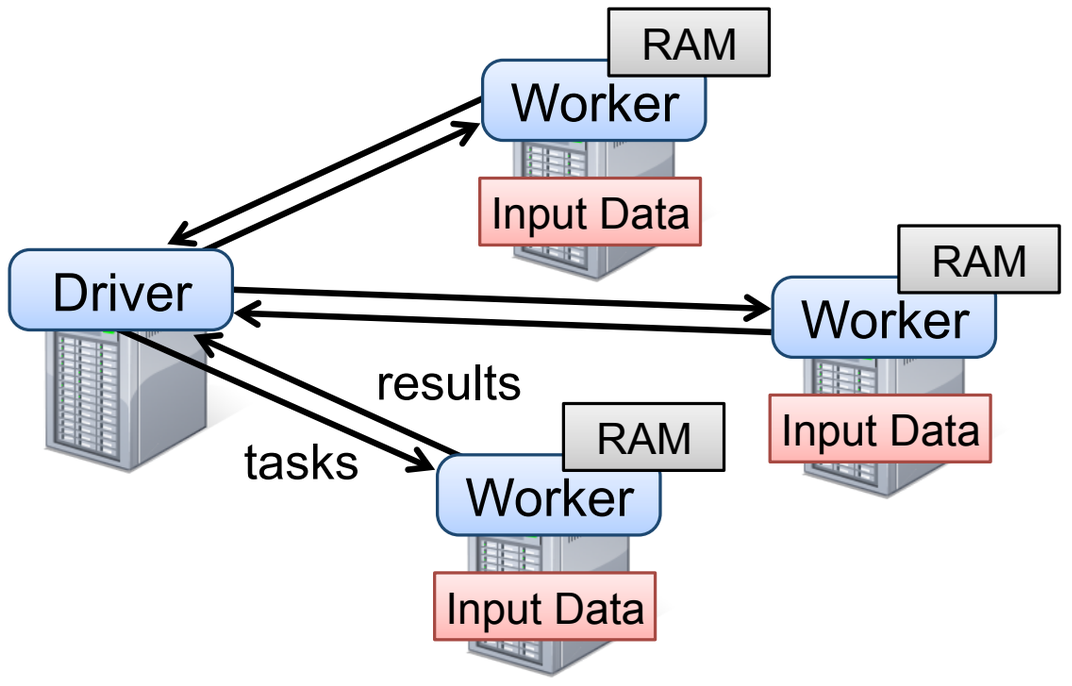
Credits: Amir H. Payberah

Spark programming model

- RDDs : collection of element values distributed over the cluster, mainly in main-memory (RAM)

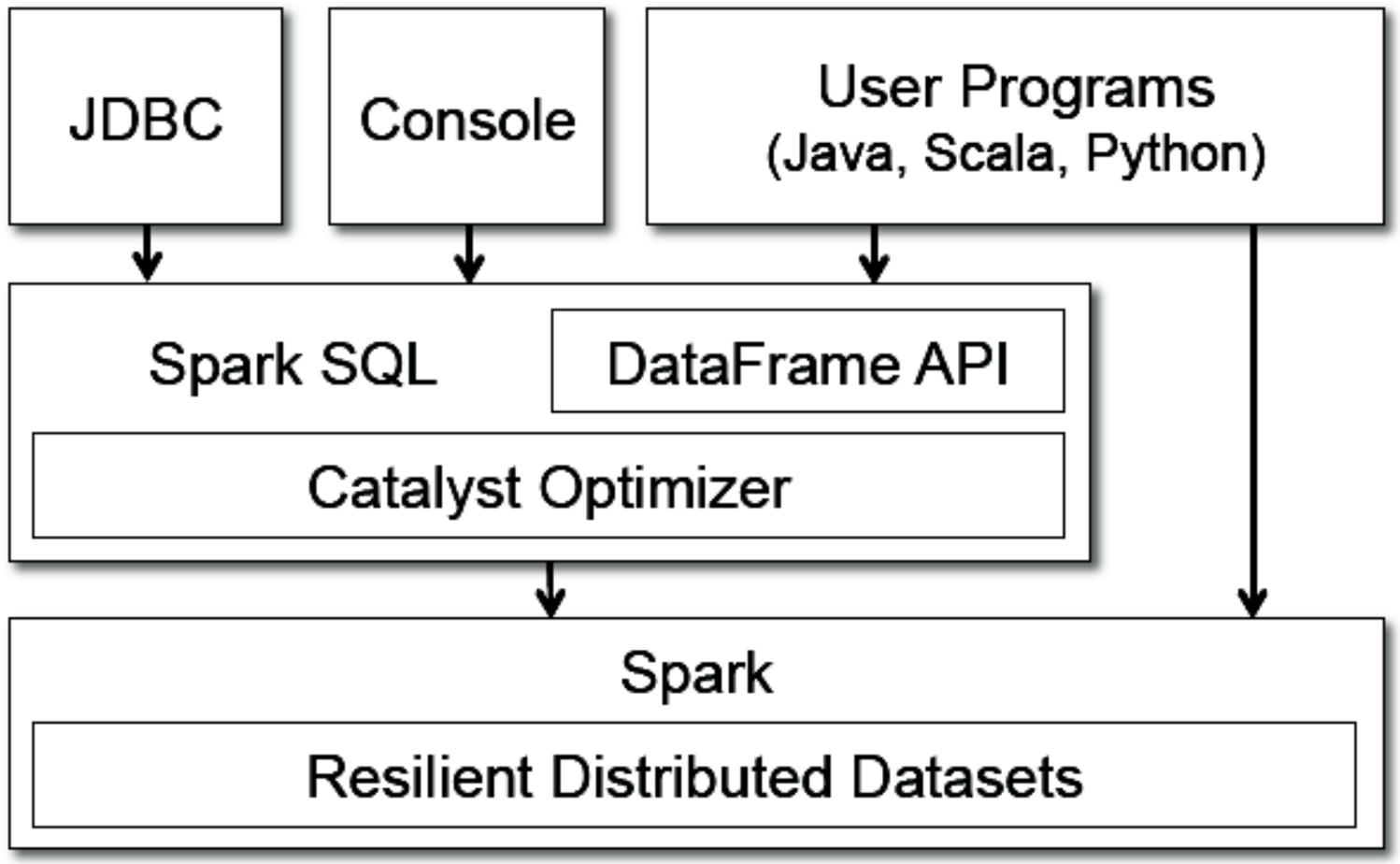
- Transformations : lazy operators that create new RDDs from RDDs.

- Actions : lunch a computation and return a value to the program driver or write data to the external storage



Dataframes

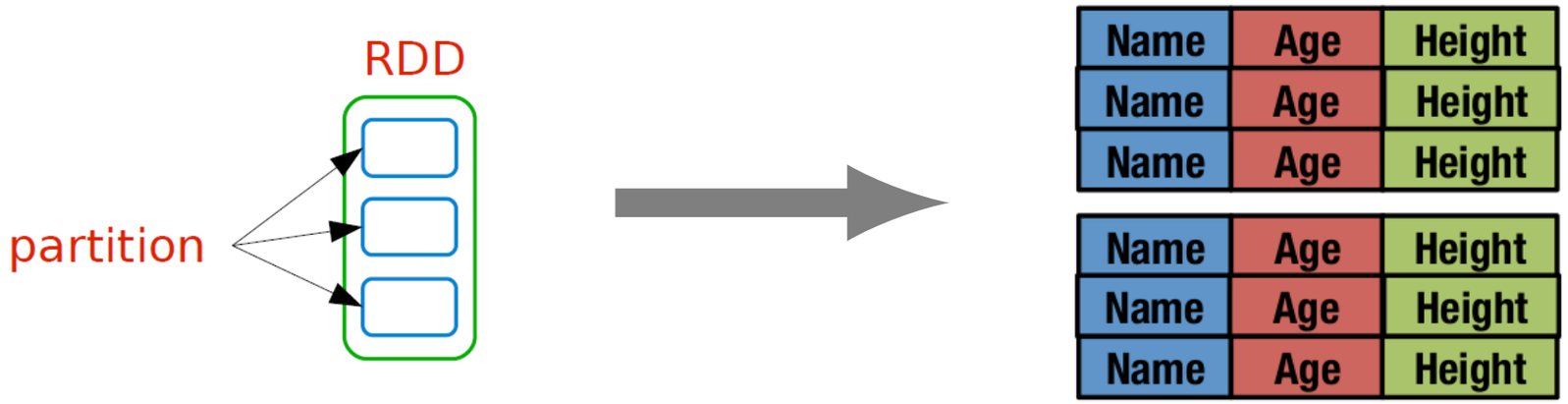
Dataframes and Spark SQL



Dataframe

- a Dataframe is a distributed collection of Row values with
- a schema
- Somewhat equivalent to a table in a relational database.

Adding schema to RDDs



RDD vs DataFrames

- Like an RDD, a DataFrame is an immutable distributed collection of data.

- Unlike an RDD, data is organized into named columns, like a table in a relational database.

- A DataFrame is a collection of values of type Row + a schema for the Row values

More on Row type

- A Row value essentially corresponds to a record beloning to a table or produced by a relational/SQL expression
- Example (from https://spark.apache.org/docs/2.1.0/api/java/org/apache/spark/sql/Row.html)

```
import org.apache.spark.sql._

 val row = Row(1, true, "a string", null)
 // row: Row = [1,true,a string,null]
 val firstValue = row(0)
 // firstValue: Any = 1
 val fourthValue = row(3)
 // fourthValue: Any = null
```

More on Row type

- Example (from https://spark.apache.org/docs/2.1.0/api/java/org/apache/spark/sql/Row.html)

```
// using the row from the previous example.
val firstValue = row.getInt(0)
// firstValue: Int = 1
val isNull = row.isNullAt(3)
// isNull: Boolean = true
```

DataFrame = Row + Schema

- Let’s see a notebook

Creating DataFrames

```
scala> val spark = SparkSession
          .builder()
          .appName("Spark SQL basic example”)
          .config()
          .getOrCreate()
```

```
val df = spark
    .read
    .json("/usr/local/Cellar/apache-spark/2.4.0/libexec/
    examples/src/main/resources/people.json")
```

Local path

https://databricks.com/blog/2016/08/15/how-to-use-sparksession-in-apache-spark-2-0.html

Creating DataFrames

```
scala> df.show()
+----+-------+
| age|   name|
+----+-------+
|null|Michael|
|  30|   Andy|
|  19| Justin|
+----+-------+


scala>
```

Using DataFrames

```
scala>  df.select($"name").show()
+-------+
|   name|
+-------+
|Michael|
|   Andy|
| Justin|
+-------+
scala> df.select($"name", $"age" + 1).show()

+-------+----------+
|   name|(age + 1)|
+-------+----------+
|Michael|      null|
|   Andy|        31|
| Justin|        20|
+-------+----------+
```

Select everybody, but increment the age by 1

Using DataFrames

```
scala> df.where($”age" > 21).show()
+---+----+
|age|name|
+---+----+
| 30|Andy|
+---+----+
scala> df.groupBy($"age").count().show()
+----+-----+
| age|count|
+——+-----+
|  19|    1|
|null|    1|
|  30|    1|
+----+-----+
```

SQL on DataFrames

Register the DataFrame as a SQL temporary view

```
scala> df.createOrReplaceTempView("people")

scala> val sqlDF = spark.sql("SELECT * FROM people")

scala> sqlDF.show()
+----+-------+
| age|   name|
+----+-------+
|null|Michael|
|  30|   Andy|
|  19| Justin|
+----+-------+


scala>
```

Converting RDDs into DataFrames

```
scala> val spark = SparkSession
          .builder()
          .appName("Spark SQL basic example”)
          .config()
          .getOrCreate()

scala> import spark.implicits._
import spark.implicits._
```

Needed for schema-inference for RDD->DF conversion when the RDD inccluded structured or semi-structured data with simple unambiguous data types

Converting RDDs into DataFrames

```
scala> val data = Array(("a",1), ("b",2) , ("a",3) , ("c",4), ("b",5))
data: Array[(String, Int)] = Array((a,1), (b,2), (a,3), (c,4), (b,5))

scala> val rdd = sc.parallelize(data)
rdd: org.apache.spark.rdd.RDD[(String, Int)] =
ParallelCollectionRDD[47] at parallelize at <console>:35

scala> val rdd_1 = rdd.reduceByKey((a, b) => a + b)

scala> rdd_1.collect
res13: Array[(String, Int)] = Array((a,4), (b,7), (c,4))



scala>
```

Converting RDDs into DataFrames

```
scala> rdd_1.collect
res13: Array[(String, Int)] = Array((a,4), (b,7), (c,4))

scala> val myDf = rdd_1.toDF("name", "val")
myDf: org.apache.spark.sql.DataFrame = [name: string, val: int]

scala> myDf.show()
+----+---+
|name|val|
+----+---+
|   a|  4|
|   b|  7|
|   c|  4|
+----+---+
scala> myDf.printSchema
root
 |-- name: string (nullable = true)
 |-- val: integer (nullable = false)
scala>
```

More details in this nice post

https://indatalabs.com/blog/convert-spark-rdd-to-dataframe-dataset

Conclusion

- Supports on a variety of data sources.

- A DataFrame can be operated on as normal RDDs or as a temporary table.
- Registering a DataFrame as a table allows you to run SQL queries over its data.
- More details on :

http://spark.apache.org/docs/latest/sql-programming-guide.html#starting-point-sparksession

Datasets

Datasets

- Datasets offers a compromise/mix between RDD and DataFrames
    - They can be used to run SQL operations
    - They preserves RDD functionalities
- A Dataset is a collection of JVM Scala objects, so objects are strongly typed
- As seen next this entails crucial benefits

Let’s see Datasets in action by means of a notebook

RDD—>DS

- A DS can also contain non case-class values, in particular when you obtain a DS from an RDD

```
scala> val wordsRDD = sc.parallelize(Seq("Spark I am your father", "May the spark be with you", "Spark I am your father”))

wordsRDD: org.apache.spark.rdd.RDD[String] = ParallelCollectionRDD[59] at parallelize at <console>:33



scala> val wordsDataset = wordsRDD.toDS()
wordsDataset: org.apache.spark.sql.Dataset[String] = [value: string]
```

```
scala> wordsDataset.printSchema
root
 |-- value: string (nullable = true)



scala> wordsDataset.show()
+--------------------+
|               value|
+--------------------+
|Spark I am your f...|
|May the spark be ...|
|Spark I am your f...|
+--------------------+
```

First example

```
scala> val groupedDataset = wordsDataset.flatMap(_.toLowerCase.split(" "))
                                        .filter(_ != "")
                                        .groupBy("value")

scala> val countsDataset = groupedDataset.count()

scala> countsDataset.show()

scala> groupedDataset.count().show
+------+-----+
| value|count|
+------+-----+
|father|    2|
|   you|    1|
|  with|    1|
|    be|    1|
|  your|    2|
|   may|    1|
| spark|    3|
```

RDD & Datasets

RDDs

```
val lines = sc.textFile("/wikipedia")

val words = lines

  .flatMap(_.split(" "))

  .filter(_ != "")
```

Datasets

```
val lines = sqlContext.read.text("/wikipedia").as[String]

val words = lines

  .flatMap(_.split(" "))

  .filter(_ != "")
```

Word-count

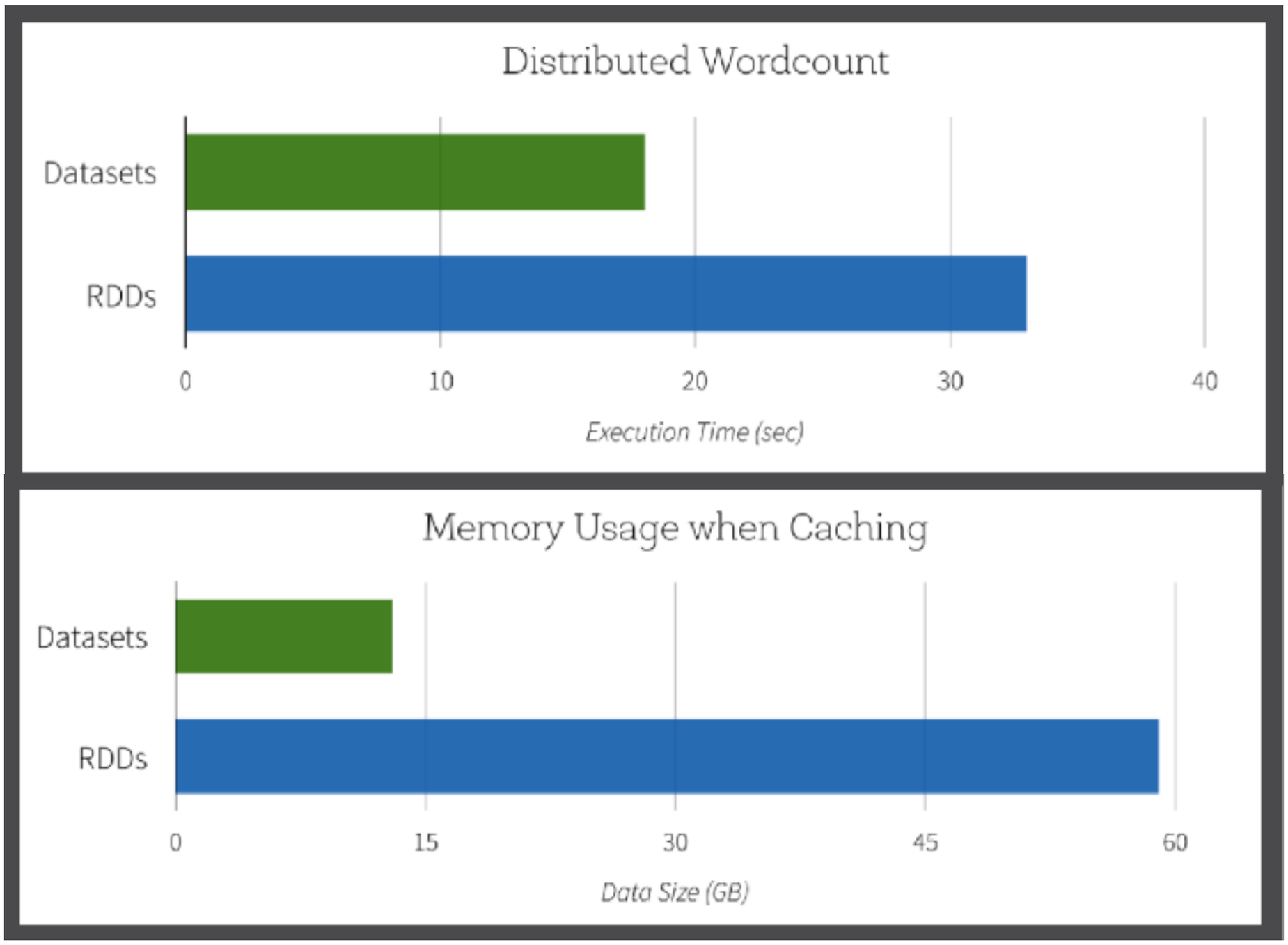
RDDs

```
val counts = words

    .groupBy(_.toLowerCase)

    .map(w => (w._1, w._2.size))
```

Datasets

```
val counts = words

    .groupBy(_.toLowerCase)

    .count()
```

Performances



https://databricks.com/fr/blog/2016/07/14/a-tale-of-three-apache-spark-apis-rdds-dataframes-and-datasets.html

Readings

- Interesting post on serialisation/deserialisation
  - https://blog.xebia.fr/2017/09/27/spark-comprendre-et-corriger-lexception-task-not-serializable/
- Datasets:
  - https://spark.apache.org/docs/2.3.0/api/java/index.html?org/apache/spark/sql/Dataset.html

RDD, DF ad DS

- When it is the case of using RDD/DF/DS?

- What are the main differences

RDD

- You need low-level transformation and actions and control on your dataset
- You want to manipulate your data with functional programming constructs, rather than domain specific expressions;
- schema is not important and/or data is unstructured, such as media streams or streams of text;
- you can give up to some optimization and performance benefits available with DF and DS
    - you implement optimizations

DF and DS

- You typically use DF and DS when you process structured (mainly relational) and semistructured (mainly JSON) data
- In these cases a schema must be available
  - this is usually the case
  - if not, the schema is inferred by Spark
- DF can be used in Python and Scala
- DS are not available in Python, you need Scala

DF vs DS

- Both give the possibility of using high-level / relational-like operators
    - Benfits: code easier to read, Spark can ensure optimizations coming from the DBMS world
- It is worth outlining, that in Machine Learning very often data is (semi-)structured
- Spark ML mainly relies on DF/DS
- As seen, with DS we can use both functional (like RDD) and declarative (like in DF) programming style
- With DS memory and execution time is much more optimized with respect to RDD and DF
- Attention: sometimes for optimization purposes you need to resort to RDD
    - as seen, in Spark, RDD/DF/DS conversions are possible