# **Table of Contents**

[Table of Contents 1](#_Toc475905913)

[1. Scala PRogramming – CourseRA 4](#_Toc475905914)

[1.1 Functional programmming Introdunction 4](#_Toc475905915)

[1.1.1 What is Functional Programming 5](#_Toc475905916)

[1.1.2 Non Primitive Expression evalution 6](#_Toc475905917)

[1.1.3 Function Parameters & Types 6](#_Toc475905918)

[1.1.4 Function Application Evaluation 6](#_Toc475905919)

[1.1.5 Substitution Model 7](#_Toc475905920)

[1.1.6 CN & CV Termination 8](#_Toc475905921)

[1.1.7 Condition Expression 8](#_Toc475905922)

[1.1.8 Reduction rules for Boolean 9](#_Toc475905923)

[1.1.9 Value Definition 9](#_Toc475905924)

[1.1.10 Block & Visibility in Scala 10](#_Toc475905925)

[1.2 Compose Function- Higer order functions 14](#_Toc475905926)

[1.2.1 Summing with Higher-Order Functions 14](#_Toc475905927)

[1.2.2 Anaonymous Functions 15](#_Toc475905928)

[1.2.3 Currying 16](#_Toc475905929)

[1.2.4 Multiple Prarameter Lists 17](#_Toc475905930)

[1.2.5 More Function Types 18](#_Toc475905931)

[1.2.6 Functions and Data (Class & Objects) 18](#_Toc475905932)

[1.2.7 Class 18](#_Toc475905933)

[1.2.8 Objects 19](#_Toc475905934)

[1.2.9 Members of an Objects 19](#_Toc475905935)

[1.2.10 Rational Arithmetic 19](#_Toc475905936)

[1.2.11 Methods 20](#_Toc475905937)

[1.2.12 Data Abstractions 21](#_Toc475905938)

[1.2.13 Self Reference 23](#_Toc475905939)

[1.2.14 Precondition 23](#_Toc475905940)

[1.2.15 Assertions 23](#_Toc475905941)

[1.2.16 Constructors 23](#_Toc475905942)

[1.2.17 Classes and Subsitutions 24](#_Toc475905943)

[**1.3** **Scala Class –Abstract Classes** 26](#_Toc475905944)

[**1.4** **Scala Cheat Sheet** 32](#_Toc475905945)

[2. Spark MapR 34](#_Toc475905946)

[2.1 Benefits of Apache Spark Over Hadoop 34](#_Toc475905947)

[2.1.1 Data Exploration-1(MAPR Lab 2\_1) 36](#_Toc475905948)

[2.1.2 Data Exploration-2 (MAPR Lab 2\_2) 37](#_Toc475905949)

[2.1.3 Data Exploration-3(MapR Lab4) 38](#_Toc475905950)

[2.1.4 Data Exploration-4(MapR Lab5) 40](#_Toc475905951)

[2.2 Spark-Sql 42](#_Toc475905952)

[2.3 Spark- Streaming 43](#_Toc475905953)

[2.3.1 Spark Streaming Use Case 1(MAPR-LAB8) 43](#_Toc475905954)

[2.4 Spark- MLlib 46](#_Toc475905955)

[2.4.1 MLLIB Use case1 46](#_Toc475905956)

[2.4.2 MLLIB Use case2 48](#_Toc475905957)

[2.5 Spark- GraphX 52](#_Toc475905958)

[2.5.1 Graphx Use case1 52](#_Toc475905959)

[2.5.1 Graphx Use case2 53](#_Toc475905960)

[2.6 Spark- Execution Model 57](#_Toc475905961)

[2.6.1 Logical Model 57](#_Toc475905962)

[2.6.2 Physical Model 58](#_Toc475905963)

[2.6.3 Data Lineage Truncation 59](#_Toc475905964)

[2.6.4 Phases During Spark Execution 59](#_Toc475905965)

[2.6.5 Spark MCS – UI 60](#_Toc475905966)

[2.7 Spark- PErformance Optimization 62](#_Toc475905967)

[2.7.1 Common Issues Leading To Slow Performance 62](#_Toc475905968)

[2.7.2 Logging 63](#_Toc475905969)

[2.7.3 Best Practices 63](#_Toc475905970)

[2.8 Spark- Code Snippet 64](#_Toc475905971)

[2.8.1 Data Load 64](#_Toc475905972)

[2.8.2 filter 64](#_Toc475905973)

[2.8.3 MAP 64](#_Toc475905974)

[3. Spark example for Transormation / Action 65](#_Toc475905975)

[3.1 Transformation Explained with Parallelize 69](#_Toc475905976)

[3.1.1 filter 69](#_Toc475905977)

[3.1.1 map, flatMap,Map,mapValues 69](#_Toc475905978)

[3.1.2 reduceByKey 71](#_Toc475905979)

[3.1.3 groupByKey 71](#_Toc475905980)

[3.1.4 Set Operator 72](#_Toc475905981)

[3.1.5 Join Operator 72](#_Toc475905982)

[3.1.6 CoGroup Operator 73](#_Toc475905983)

[3.2 Spark SQL 73](#_Toc475905984)

[3.3 Spark Streaming 74](#_Toc475905985)

[3.4 Accumulator and Broadcast Variable 78](#_Toc475905986)

[3.4.1 Accumulator 78](#_Toc475905987)

[3.4.2 BroadCast 79](#_Toc475905988)

[3.5 Certificate Preparation 80](#_Toc475905989)

[3.5.1 Standalone Application 80](#_Toc475905990)

[3.5.2 Libraries 80](#_Toc475905991)

[3.5.3 Spark Basics 80](#_Toc475905992)

[3.5.4 udf 81](#_Toc475905993)

[3.5.5 Spark Basics 81](#_Toc475905994)

[3.5.6 Spark Basics 81](#_Toc475905995)

[4. Spark Microsoft 84](#_Toc475905996)

[4.1 Data Exploration 84](#_Toc475905997)

[4.2 Classification 87](#_Toc475905998)

[4.3 PipeLine 89](#_Toc475905999)

[4.4 Classification Evaluation 91](#_Toc475906000)

[4.4.1 Tunning Model Parameters 94](#_Toc475906001)

[4.5 Regression Evaluation 97](#_Toc475906002)

[4.5.1 Cross Validation for Regression 99](#_Toc475906003)

# Scala PRogramming – CourseRA

Books :-

1. Programming in Scala by Martin Odersky, Lex Spoon and Bill Venners, 2nd editions
2. Scala for the Impatient.
3. Scala in Depth.

## Functional programmming Introdunction

Functional programming is a different paradigm from regular practice, Main programming paradigms are

1. Imperative programming (java,c)
2. Functional programming (scala)
3. Logic programming

Note: Object-Oriented programming is Orthogonal to all three.

Imperative Programming is about,

1. Modifying the mutable variables

2. Using assignments

3. Control structures like, if then else, loop, break, continue, return

Von Neumann computer has, processor, Memory and bus (32 or 64 bit)

* Mutable Variable => Memory cells
* Variable Dereferences => load instruction
* Variable assignments => Store instruction
* Control Structures => jumps

Problem here is, how can we avoid conceptualizing programs word by word, coined by John Backus. Pure imperative programming is limited by the “Von Neumann” bottleneck. “One tends to conceptualize data structures word by word.”

We need other technique for defining high-level abstractions such as collections, polynomials, shapes, strings, documents. Ideally, develop theories of collections, shapes, strings..

What is a Theory, A theory consists of,

1. One or more data types
2. Operations on these types
3. Laws that describe the relationships between values and operations.

Note: Normally a theory doesn’t describe mutations.

Consequences for Programming of this new paradigm is,

If you want to implement high-level concepts following their mathematical theories, there’s no place for mutation.

* Theories do not admit it.
* Mutation can destroy useful laws in the theories.

Therefore lets,

* Concentrate on defining theories for operators expressed as functions.
* Avoid mutations.
* Have powerful ways to abstract and compose functions.

### What is Functional Programming

It’s of two types’restricted sense and wider sense

* Restricted Sense, functional programming means programming without variables, assignments, loops and other imperative control structures.

Eg: Pure Lisp, XSLT, Xpath, etc.

* Wider Sense, functional programming means focusing on the functions.

Eg: Lisp,F#,**Scala**, Smalltalk,Ruby..

* In particular, functions can be values that are produced, consumed and composed.
* All this becomes easier in a functional language.
* Functions in FP language are first class citizens means,
  + They can be defined anywhere including inside the functions.
  + Like any other value, they can be passed as parameters to functions and returned as results.
  + As for other values, there exists a set of operators to compose functions.
* Most functional programming as REPL (read evaluate print loop) (similar like calculator)

**Functional programming benefit:**

1. Simpler reasoning principles
2. Better modularity
3. Good for exploiting parallelism for multicore and cloud computing.

### Non Primitive Expression evalution

Every non trivial programming language provides,

1. Primitive expressions
2. Combine expressions
3. Abstract expressions; introduce a name for an expression by which it can then be referred to.

Non primitive expression is evaluated as follows

1. Take the leftmost operator
2. Evaluate its operands (left before right) (2&pi)
3. Apply the operator to the operands

A name is evaluated by replacing it with the right hand side of its definition. Evolutions process stops once its results in value, let’s consider value to number (also other kind)

Eg: (2\*pi)\*radius

### Function Parameters & Types

Function parameters come with their type , which is given after a colon,

* def power(x: Double, y: Int): Double= ….

If a return type is given, it follows the parameter list.Definitions can have parameters, for instance,

* def square(x: Double) = x\*x
* square(5+4) o/p is 81
* def sumOfSquares(x: Double, y: Double) = square(x) + square(y)

Types:

It as primitive types like java but are written capitalized

1. Int, 32 bit integers.
2. Double, 64-bit floating point numbers.
3. Boolean, Boolean values true and false.

### Function Application Evaluation

Applications of parameterized functions are evaluated in asimilar way as operators

1. Evaluate all functions arguments , from left to right
2. Replace the function application by the function’s right-hand side at the same time.
3. Replace the formal parameters of the function by the actual arguments.

Eg: sumOfSquares(3, 2+2) > sumOfSuares(3,4) > square(3) + square(4) > 3\*3+square(4) > 9+square(4) > 9+4\*4 > 9+16 >25

### Substitution Model

1. This scheme of expression evaluation is called the Substitution Model.
2. The idea underlying this model is that all evaluation does is reducing an expression to a value.
3. It can be applied to all expressions, as long as they have no side effects.

Eg:

Var=1; some function{

return(var ++)

}

For the first iteration var is one for next its two and increases by one each time; expression var++ as side effect on the current state on the variable.

1. The substitution model is formalized in the Lamda(inverse y) –Calculus, which gives a foundation for the functional programming.

Does every expression reduce to value(in a finite number of steps) and answer is no, for Eg, below code forms infinite loop,

* def loop: Int = loop

loop

**Changing the evaluation strategy:**

The interpreter reduces function arguments to values before rewriting the function application (call by value). One could alternatively apply the function to unreduced arguments (call by name), for eg,

sumOfSquares(3, 2+2) > square(3) + square(2+2) > 3\*3 + square(2+2) >9 + square(2+2) >**9 + (2+2)\*(2+2)**> 9 + 4\*4 >9+16>25

Both strategies reduce to the same final values as long as,

1. The reduced expression consists of pure functions, and
2. Both evaluations terminate.

* Call by Value(CV) has the advantage that it evaluates every function argument only once.
* Call by Name(CN) has the advantage that a function argument is not evaluated if the corresponding parameter is unused in the evaluation of the function body.
* Def test(x: Int, y) = x\*x
* test (2,3) // (CV & CN)both are same
* test(3+4,8) // CV is faster
* test(7,2\*4) // CN is faster
* test(7,2\*4\*2) // both are same

### CN & CV Termination

If the termination is not guaranteed? Then,

1. If the CV evaluation of an expression ‘e’ terminates, then CN evaluation of ‘e’ terminates too,
2. Other direction is not true.

Eg:

* Def first(x: Int, y:Int) =x
* first(1,loop) //And consider the expression
* Under CN: first(1,loop) > 1
* Under CV: first(1,loop) > goes to infinite loop

In Scala normally uses call-by-value(performance and avoid the side effects), but if the type of a function parameter starts with => it uses call-by-name. eg:

* def constOne(x: Int, y: => Int) = 1
* let’s evaluate constOne(1+2, loop) & constOne(loop, 1+2)
* constOne(1+2, loop) > constOne(3, loop) >1
* constOne(loop, 1+2) > infinite loop

### Condition Expression

If-Else:To choose between alternatives use if-else, it looks like a if-else in java, but is used for expressions, not statements

Eg; def abs(x: Int) = if (x >=0) x else -x

Note: x >= 0 is a predicate, of type Boolean.

Boolean expression: say b can be composed of

True false // Costants

!b // Negation

b && b // Conjunction

b || b // Disjuntion

and of the usual comparison operations:

e <= e, e >= e,e < e,e > e,e == e,e != e

### Reduction rules for Boolean

Here are reduction rules for Boolean expressions (e is an arbitrary expression)

!true 🡪 false

!fales 🡪 true

true && e 🡪 e

false && e 🡪 false

true || e 🡪 true

fasle || e 🡪 e

Note that && and || don’t always need their right operand to be evaluated. We say expressions use “short-circuit evaluation”

### Value Definition

We have seen that functions parameters can be passed by Value or be Passed by name. The same distinction applies to definitions.

The def form is “by-name” its right hand side is evaluated on each use.

There is also a val for, which is “by-value”, right-hand side of val is evaluated at the point of the definition itself, afterwards the name refers to the value. example

Val x= 2

Val y =square(x) // so here y refers to 4 and not to the expression square(x)

Value Definition and Termination:

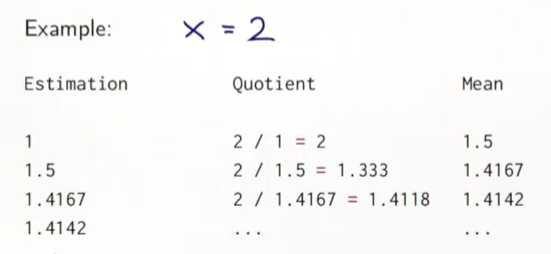
Difference b/w val and def becomes apparent when the right hand side doesn’t terminate, given example

* def loop: Boolean=loop
* by def, def x = loop > is ok for definition
* by val, val x = loop > will lead to an infinite loop

**Exercise:** create a function to find the square root by using newton’s method(successive approximation).

Method:

1. start with an initial estimate y(let’s pick y=1)
2. Repeatedly improve the estimate by taking the mean of y and x/y



**Implementation in Scala:**

First, define a function which computes one iteration step.

* Object session {
* def sqrtIter(guess: Double, x: Double): Double =

if (isGoodEnough(guess,x)) guess

else sqrtIter(improve(guess, x), x)

* def isGoodEnough(guess: Double, x: Double) =

abs(guess\*guess -x) < 0.001

//abs(guess\*guess -x) / x < 0.001

* def improve(guess: Double, x: Double)=

(guess + (x/guess) /2

* def sqrt(x: Double)=sqrtIter(1.0, x)
* }

Note: the sqrtIter is recursive, its right-hand side calls itself. Recursive functions need an explicit return type in scala. For non-recursive functions, the retrun type is optional.

### Block & Visibility in Scala

A block is delimited by braces, it contains a sequence of definitions or expressions. The last element of a block is an expression that defines its value. This return expression can be preceded by auxiliary definitions. Blocks are themselves expressions; a block may appear everywhere an expression can(including the right hand side of function definition).

Eg;

{ val x= f(3)

x\*x

}

* def sqrt(x: Double)= {
* def sqrtIter(guess: Double, x: Double): Double =

if (isGoodEnough(guess,x)) guess

else sqrtIter(improve(guess, x), x)

* def isGoodEnough(guess: Double, x: Double) =

abs(guess\*guess -x) < 0.001

//abs(guess\*guess -x) / x < 0.001

* def improve(guess: Double, x: Double)=

(guess + (x/guess) /2

sqrtIter(1.0, x)

* }

**Blocks and Visibility:**

Definitions inside a block are only visible from within the block. The definition inside a block shadow definitions of the same names outside the block.

Eg:

* val x=0
* def f(y: Int) = y+1
* val result = {

val x = f(3) *// value of x is not 0 & f(x) is visible for f(3)*

x\*x

} +x *// Value of x is 0 also block is used as a expression*

By basing our idea that value of x is visible inside the bracket we remove the redundant definition of x, also note that value of x has never changed.

* def sqrt(x: Double)= {
* def sqrtIter(guess: Double): Double =

if (isGoodEnough(guess)) guess

else sqrtIter(improve(guess))

* def isGoodEnough(guess: Double) =

abs(guess\*guess -x) < 0.001

//abs(guess\*guess -x) / x < 0.001

* def improve(guess: Double)=

(guess + (x/guess) /2

sqrtIter(1.0)

* }

Semicoln :

It’s optional, if there more than one statement on a line, they need to be separated by semicolon.

* val y = x+1; y\*y

**Semicolons and infix operators:**

One issue with Scala’s semicolon convention is how to write expressions that span several lines. It can be avoided by parentheses or plus symbol.

(someLongExpression

SomeotherExpression)

(or)

someLongExpression +

SomeotherExpression

**Review: Evaluating a function Application**

One Simple rule: one evaluates a function application f(e1,…,en)

* by evaluating the expressions e1,..,en resulting in the values v1,..vn, then
* by replacing the application with the body of the function f, in which
* the actual parameters v1,…vn replace the formal parameters of f

**Application Rewriting Rule:**

This can be formalized as a rewriting of the program itself.

* def f(x1,..,xn) = B; …f(v1,…,vn)
* def f(x1,..,xn) = B; …[v1/x1,….,vn/xn]B

Here, [v1/x1,…,vn/xn]B means:

The expression B in which all occurrences of xi have been replaced by vi.

Note:

Formal parameters are x1,..xn and function application is the implementation of the function and not the definition.

Example for GCD

* def gcd(a: Int, b: Int): Int=

if (b==0) a else gcd(b, a%b)

//gcd(14,21) goes through gcd(21,14), gcd(14,7),gcd(7,0) to get the answer 7.

Example for factorial:

* def factorial(n: Int): Int=

if(n==0) 1 else n\*factorial(n-1) // not tail recursive function.

//factorial(4) goes through 4\*factorial(3) …4\*(3\*(2\*(1\*1)))

Note: in gcd we regain the gcd form again but the factorial expression goes bigger and bigger

**Implementation Consideration:**

If the function calls itself as its last action, the function’s stack frame can be reused. This is called tail recursion. (Tail recursive functions are iterative processes.)

In general, if the last action of a function consists of calling a function(which may be the same), one stack frame would be sufficient for both functions. Such call are called tail-calls.

/\* helps avoid stack overflow error conditions\*/

Modifying factorial into tail calls:

* object exercise {
* def factorial(n: Int): Int = {

def llop(acc: Int, n: Int): Int =

if(n == 0 ) acc

else loop(acc\*n,n-1)

loop(1,n)

}

}

## Compose Function- Higer order functions

Functional language treat functions as first-class values, it means like any other values, a function can be passed as a parameter and returned as a result. This provides a flexible way to compose programs. Functions that take other functions as parameters or that return functions as results are called higher order functions.

Take the sum of the integers between a and b:

* def sumInts(a: Int, b: Int): Int =

if (a>b) 0 else a+sumInts(a+1, b)

Take the sum of the cubes of all the integers between a and b:

* def cube(x: Int): Int = x\*x\*x
* def sumCubes(a: Int, b:Int): Int=

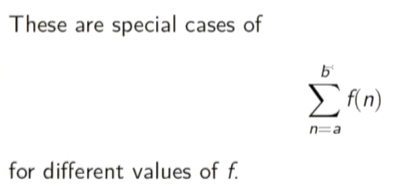
if(a>b) 0 else cube(a) + sumCubes(a+1,b)

Take the sum of factorial of all the integersbetween a and b:

* def sumFactorials(a: Int, b: Int): Int=

if (a>b) 0 else fact(a) + sumFactorials(a+1,b)

//can factor out the common pattern into a singular method.



### Summing with Higher-Order Functions

Let’s define:

* def sum(f: Int => Int, a: Int, b: Int): Int=

if (a > b) 0

else f(a) + sum(f,a+1,b)

// f is not a given function but a parameter.

**Function Type:**

The type A => B is the type of a ‘function’ that takes an argument of type A and returns a result of type B. So, Int => Int is the type of functions that map integers to integers.

We can then write:

* def sumInts(a: Int, b: Int ) = sum(id, a, b)
* def sumCubes(a: Int, b: Int ) = sum(cube, a, b)
* def sumFactorials(a: Int, b: Int ) = sum(fact, a, b)

Where // below are the auxillary functions

* def id(x: Int):Int = x
* def cube(x: Int):Int = x\*x\*x
* def fact(x: Int):Int = if (x==0) 1 else fact(x-1)
* //sum(id,1,4)
* //sumInts(1,4)

### Anaonymous Functions

Passing functions as parameters leads to the creation of many small functions. Sometimes it is tedious to have to define (and name) these functions using def.

We would like to think functions as literals say if you want to print the string you directly print. These literals are called anonymous functions.

A function that raises its argument to cube:

* (x: Int) => x\*x\*x

Here, (x: Int) is the parameter of the function, and x\*x\*x is it’s body. The type of the parameter can be omitted if it can be inferred by the compiler from the context. If there are several parameters, they are separated by commas:

* (x: Int,y:Int) => x+y

Using anonymous functions, we can write sums in a shorter way (remove the auxiallry definition):

object excercise1 {

def sum(f:Int => Int,a: Int, b: Int): Int =

if (a>b) 0

else f(a) + sum(f,a+1,b)

def sumInts(a: Int, b: Int) = sum(x=> x, a,b)

def sumCubes(a: Int, b: Int) = sum1(x=> x\*x\*x, a,b)

}

sumInts(1,4)

//sumCubes(1,4)

The sum Functions uses tail recursion function (useful for summation(if not for cube))

object excercise2 {

def sum(f: Int => Int, a: Int, b: Int):Int={

def loop(a: Int, acc: Int): Int=

if(a > b)acc

else loop(a+1,f(a)+acc)

loop(a,0)

}

def sumInts(a: Int, b: Int) = sum(x=> x, a,b)

def sumCubes(a: Int, b: Int) = sum(x=> x\*x\*x, a,b)

}

sumInts(1,4)

//sumCubes(1,4)

def sum(f: Int => Int, a: Int, b: Int):Int={

def loop(a: Int, acc: Int): Int=

if(a > b)acc

else loop(a+1,f(a)+acc)

loop(a,0)

}

//def sumInts(a: Int, b: Int) = sum(x=> x, a,b)

//def sumCubes(a: Int, b: Int) = sum(x=> x\*x\*x, a,b)

sum(x => x\*x,3,5)

### Currying

def sumInts(a: Int, b: Int) = sum(x=> x, a,b)

def sumCubes(a: Int, b: Int) = sum(x=> x\*x\*x, a,b)

def SumFactorials(a: Int,b: Int) = sum(fact,a,b)

sumInts(4,5)

Can we get rigid of the parameter a,b since they are redundant. Let’s rewrite because we have repeation of parameter a,b in both the functions.

def sum1(f1: Int => Int):(Int,Int) => Int={

def sumF(a: Int, b: Int): Int =

if (a>b) 0

else f1(a)+sumF(a+1,b)

sumF

}

Sum is a functions that returns a function, SumF is defined inside Sum and SumF returns to Sum,We can then define

def sumInts = sum(x => x)

def sumCubes = sum(x => x\*x\*x)

def sumFactorials= sum(fact)

//sumCubes(1,10)

sumInts(1,2)

How can we avoid sumInts, sumCubes,.. (middlemen)

sum(cube)(1,3)

1. Sum(cube) applies sum to cube and returns the sum of cubes functions
2. sum(cube) is therefore equvalent to sumCubes
3. This function is next applied to the arguments(1,10)

Note: Function application associates to the left,

Sum(cube)(1,10) == (sum(cube))(1,10)

### Multiple Prarameter Lists

The definition of **functions that return functions** is so useful in functional programming that there is a special syntax for it in Scala. For example, the following definition of sum is equivalent to the one with the nested sumF function, but whorter:

def cube(x: Int):Int = x

def sum(f: Int => Int)(a: Int, b: Int): Int=

if(a>b) 0 else f(a) + sum(f)(a+1,b)

//sum(cube)(1,4)

to the one with the nested , if you this then you can write sum(cube) by itself is a valid expression, you can avoid passing arguments, they can arrive in different parameter list, they arrive latter.

Generalization:

In general, a definition of a function with multiple parameter lists,

def f(args1) … (argsn)=E

//you have a function “f” with multiple arguments and body “E”

// above equation is equivalent to writing function “f” of n-1 arguments and for last //argument who create a fresh function called “g” maps it to the body E and “g” is //what that gets returned.

def f(args1) … (argsn-1)={ def g(argsn)=E;g}

// We can also write anonymous function (for short)

def f(args1) … (argsn-1)=(argsn =>E)

// if you repeat this

def f = (args1=>(args2=>… (argsn => E)…))

This style of definition and function application is called currying.

### More Function Types

Note that functional types associate to the right.

Question,

def sum(f: Int => Int)(a: Int, b: Int): Int = …

What is the type of sum ? and Answer is function type on the right i:e; Int

(Int => Int) => (Int,Int) => Int //is same as (Int => Int) => ((Int,Int) => Int)

Note that functional types associate to the right. That is to say that

Int => Int => Int // is same as Int => (Int => Int)

### Functions and Data (Class & Objects)

In this section We will learn how functionscreate and encapsulate data structure.

Example:

Rational Numbers, We want to design a package of doing rational arithmetic. A rational number (x/y) is represented by two integers:

* Its numerator x and
* Its denominator y.

Suppose we want to implement the additional of two rational numbers.

def addRationalNumerator(n1: Int, d1: Int, n2: Int,d2: Int): Int

def addRationalDenominator(n1: Int, d1: Int, n2: Int,d2: Int): Int

but it would be difficult to manage all these numerators and denominators.A better choice is to combine the numerator and denominator of a rational number in a data structure.

### Class

In Scala we do this by defining a class,

class Rational(x: Int, y: Int){

def numer = x

def denom = y

}

This definition introduces two entities,

1. A new type, named Rational.
2. A constructor Rational to create elements of this type.

Scala keeps the names of types and values in different namespaces. So there’s no conflict between the two definitions of Rational.

### Objects

We call the elements of a class type “objects”. We create an object by prefixing an application of the constructor of class with the operator “new”.Note: Type in Programming language is essentially set of values.

Example:new Rational(1,2)

object rational {

val x = new Rational(1,2)

x.numer

x.denom

}

class Rational(x: Int, y: Int){

def numer = x

def denom = y

}

### Members of an Objects

Objects of the Class Rational have two members, numer and denom. We select the member of an object with the infix operator ”.” (like in Java) i:e; ObjectName.MemberName

Example:

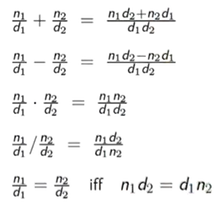
Val x = new Rational(1, 2)

x.numer

x.denom

### Rational Arithmetic

We can now define the arithmetic functions that implement the standard rules.



One thing we could do here is use Class Rational as pure data type.

object rational {

makeString(addRational(new Rational(1,2), new Rational(2,3))) // ans7/6

}

class Rational(x: Int, y: Int){

def numer = x

def denom = y

}

def addRational(r: Rational, s: Rational): Rational =

new Rational(

r.numer \* s.denom + s.numer \* r.denom,

r.denom \* s.denom)

def makeString(r: Rational) =

r.numer + "/" + r.denom

#makeString(addRational(new Rational(1,2), new Rational(2,3))) // ans7/6

### Methods

One can go further and also package functions operating on a data abstraction in the data abstraction itself. Suh functions are called “methods”

Example:

Rational numbers now would have, in addition to the functions “numer” and “denom”, the functions add, sub, mul, div, equal, toString.

object rational {

val x = new Rational(1,3)

val y = new Rational(5,7)

val z = new Rational(3,2)

//x.numer

//x.denom

//x.add(y)

x.sub(y).sub(z)

}

class Rational(x: Int, y: Int){

def numer = x

def denom = y

def add(that: Rational) =

//add function takes single parameter where is other one, left operand of add is //rational number itself, rational number for which we defined class.

new Rational (

numer \* that.denom + that.numer \* denom,

denom \* that.denom)

def neg: Rational = new Rational(-numer, denom)

def sub(that: Rational) = add(that.neg)

override def toString = numer + “/” + denom

}

### Data Abstractions

When try to find y.add(y) it produces o/p of 70/49 but not as 10/7. It needs to simplified in the simplest possible form.

object rational {

val x = new Rational(1,3)

val y = new Rational(5,7)

val z = new Rational(3,2)

//x.numer

//x.denom

//x.add(y)

//x.sub(y).sub(z)

y.add(y)

x.less(y)

x.max(y)

val strange = new Rational(1,0)

strange.add(strange)

new Rational(2)

}

class Rational(x: Int, y: Int){

require(y!=0, “denominator must be nonzero”)

def this(x: Int) = this(x,1)

private def gcd (a: Int, b: Int): Int = if (b==0) a else gcd(b, a%b)

private val g = gcd(x,y)

def numer = x / g

def denom = y / g

def less(that: Rational) = numer \* that.denom < that.numer \* denom

Note: “this” helps to refer the current rational number as whole.

def max(that: Rational) = if (this.less(that)) that else this

def add(that: Rational) =

//add function takes single parameter where is other one, left operand of add is //rational number itself, rational number for which we defined class.

new Rational (

numer \* that.denom + that.numer \* denom,

denom \* that.denom)

def neg: Rational = new Rational(-numer, denom)

def sub(that: Rational) = add(that.neg)

override def toString = numer + “/” + denom

}

There are three ways to implement the below code, however for the users it’s the same behavior for each case. This ability to choose different implementations of the data without affecting clients is called data abstraction.

1. gcd and g are private members, we ca access them from inside the Rational class. In this example, we calculate gcd immediately, so that its value can be reused in the calculations of numer and denom.

private def gcd (a: Int, b: Int): Int = if (b==0) a else gcd(b, a%b)

private val g = gcd(x,y)

def numer = x / g

def denom = y / g

1. It is possible to call gcd in the code of numer and denom: This can be advantageous if it is expected that the functions numer and denom are called in frequently.

private def gcd (a: Int, b: Int): Int = if (b==0) a else gcd(b, a%b)

def numer = x / gcd(x,y)

def denom = y / gcd(x,y)

1. It is equally possible to turn numer and denom into vals. So that they are computed only once.This can be advantageous if it is expected that the functions numer and denom are called in often

private def gcd (a: Int, b: Int): Int = if (b==0) a else gcd(b, a%b)

def numer = x / gcd(x,y)

def denom = y / gcd(x,y)

### Self Reference

On the inside of a class, the name “this” represents the object on which the current method is executed.

Note: “this” helps to refer the current rational number as whole.

def max(that: Rational) = if (this.less(that)) that else this

Note that a simple name x, which refers to another member of the class, is an abbreviation of “this.x”. Thus , an equivalent way to formulate “less” is as follows.

//def less(that: Rational) = numer \* that.denom < that.numer \* denom

def less(that: Rational) =

this.numer \* that.denom < that.numer \* this.denom

### Precondition

Let’s say our Rational class required that the denominator is Positive. We can enforce this by calling the “require” function. “require” is a predefined fuction. It takes a condition and optional message string. If the condition passed to require is false, an IllegalArgumentException is thrown with the given message string.

class Rational(x: Int, y: Int){

require(y!=0, “denominator must be nonzero”)

### Assertions

Beside “require” there is also assert. Assert also takes a condition and an optional message string as parameters. Eg.

val x = sqrt(y)

assert( x >= 0)

Like “require” , a failing assert will also throw an exception, but it’s a different one: AssertionError for assert, IllegalArgumentException for require. This reflects a difference in intent,

* “require” is used to enforce a precondition on the caller of a fuction.
* Assert is used as to check the code of the function itself.

### Constructors

In Scala, a class implicitly introduces a constructor. This one is called primary constructor of the class. The primary constructor,

* Takes the parameters of the class
* And executes all statements in the class body (such as the “require” a couple of slides back)

If you want to create second constructor of class of Rational, say which takes only one parameter,

class Rational(x: Int, y: Int){

require(y!=0, “denominator must be nonzero”)

// if the “this” is used as function position than it means constructor of class

def this(x: Int) = this(x,1)

//secondary constructor this(x:Int) calls the other constructor which takes two arguments, which is implicit primary constructor

new Rational(2)

### Classes and Subsitutions

**Classes and Substitutions:**

We previously defined the meaning of a function application using a computation model based on substitution. Now we extend this model to classes and objects.

Question: How is an instantiation of the class ‘new C(e1,.., em)’ evaluated?

Answer: The expression arguments e1,..em are evaluated like the arguments of a normal function. That’s it.

The resulting expression ,say, ‘new C(v1, … ,vm)’ is already a value.

Now that definition like given below,

Class c(x1,..,xm){ … def f(y1,…,yn)=b … }

Where

* Formal parameters of the class are x1,…,xm.
* Class defines a method f with formal parameters y1, …, yn.

(The list of function parameters can be absent. For simplicity, we have omitted the parameter types)

Question: How is the following expression evaluated ?

new C(v1, …, vm).f(w1, …, wn)

Answer: The expression ‘new C(v1, … ,vm).f(w1, …, wn) is rewritten to:

‘[w1/y1, …, wn/yn][v1/x1, …, vm/xm][new C(v1, …, vm)/this]b’

There are three substitutions at work here:

* Substitution of formal parameters y1, …, yn of the function f by the arguments w1, …, wn
* Substitution of formal parameters x1, …, xm of the class f by the arguments v1, …, vm
* Subsituation of the self reference “this” by the value of the object new C(v1, …, vn)

Operators:

In principle, the rational numbers defined by Rational are as natural as integers. But for the user of these abstractions, there is a noticeable difference.

1. We write x + y, if x and y are integers, biut
2. We write r.add(s) if r and s are rational numbers.

In Scala, we can eliminate this difference. We procede in two steps

1. Infix Notation
   1. Any method with a parameter can be used like an infix operator. It is therefore possible to write
      1. r add s r.add(s)
      2. r less s r.less(s)
      3. r max s r.max(s)
2. Relaxed Identifiers, Operators can be used as identifiers. Thus an identifier can be
   1. Alphanumeric, starting with a letter, followed by a sequence of letters or numbers
   2. Symbolic, starting with an operator symbol, followed by other operator symbols.
   3. The underscore character ‘\_’ counts as a letter.
   4. Alphanumeric identifiers can also end in an underscore, followed by some operator symbols.

Example:

X1 \* +?%& vector\_++

Precedence Rules:

The precedence of an operator is determined by its first character. The following table lists the characters in increasing order of priority precedence:

(all letters) /\* lowest precedence\*/

|

^

&

<>

=!

:

+-

\*/%

(all other special characters) /\* highest precedence\*/

Eg: Every Binary operation needs to put into parentheses, but the structure of the expression should not change.

a + b ^? C ?^ d less a ==> b | c

((a + b) ^? (c ?^ d)) less (( a ==> b)| c )

## **Scala Class –Abstract Classes**

Abstract Classes can contain members which are missing an implementation (in our case incl & contains). Consequently no instances of an abstract class can be created with operator ”new”.

A tree for the empty set.

A tree consisting of an integer and two sub-trees

Note all the values on the right is always greater than left.

Object intsets {

Println(“welcome to scala worksheet”)

# new IntSet

}

abstract class IntSet {

def incl(x: Int): IntSet

def contains(x: Int): Boolean

}

One way of doing (implementing) it would be binary tree structure. There are two types of possible trees

Implementation – Empty Tree

Class Empty extends IntSet {

def contains(x: Int): Boolean = false

def incl(x: Int): IntSet = new NonEmpty(x, new Empty, new Empty )

#override def toString = “.”

}

Class NonEmpty(elem: Int, left: IntSet, right: IntSet) extends IntSet {

def contains(x: Int): Boolean =

if (x < elem) left contains x

else if (x > elem) right contains x

else true

def incl(x: Int): IntSet =

if (x < elem) new NonEmpty(elem,left incl x, right)

else if (x > elem) new NonEmpty(elem, left, right incl x)

else this

#override def toString = “{” + left + elem + right + “}”

}

Execution:

Object intsets {

Val t1 = new NonEmpty(3, new Empty, new Empty)

Val t2 = t1 incl 4

}

Empty and NonEmpty both extends the class IntSet, means they are subclasses. This implies that the types Empty and NonEmpty conform to the type IntSet

An object of type Empty and NonEmpty can be used wherever an object of type IntSet is required.

Terminology:

IntSet is called the superclass of Empty and NonEmpty.

Empty and NonEmpty are subclasses of IntSet.

In scala, any suer-defined class extends another class.

If no superclass is given, the standard class “object” in the Java package java.lang is assumed.

The direct or indirect superclasses of a class C are called base classes of C.

So, the base classes of NonEmpty are Intset and Object.

Implementation and Overriding:

The definitions of “contains” and “incl” in the classes Empty and NonEmpty implements the abstract function in the base trait IntSet. It is also possible to redefine an existing, non-abstract definition in a subclass by using override.

Example:

abstract class Base {

def foo = 1

def bar: Int

}

class Sub extends Base {

override def foo = 2

def bar = 3

}

Object Definitions:

In the IntSet example, one could argue that there is really only a single empty Inset. So it seems overkill to have the user create many instances of it. We can express this case better with an definition:

object Empty extends IntSet {

def contains(x: Int): Boolean = false

def incl(x: Int): IntSet = new NonEmpty(x, Empty, Empty )

#override def toString = “.”

}

This defines a singleton object named Empty. No other Empty instances can be (or need to be ) created. Singleton objects are values, so Empty evaluates to itself.

Standalone applications in Scala, each such application contains an object with a main Method.

For instance, here is the “Hello World” program in scala.

Object Hello {

def main(args: Array[string]) = println(“hello World”)

}

You can complie with “scala Hello”

Dynamic Binding:

Object-oriented languages (including Scala) implement dynamic method dispatch. This means that the code invoked by a method call depends on the runtime type of the object that contains the method.

Note: Dynamic dispatch of methods is analogous to calls to higher-order functions. Can we implement one Concept in terms of the other?

Object in terms of higher-order functions?

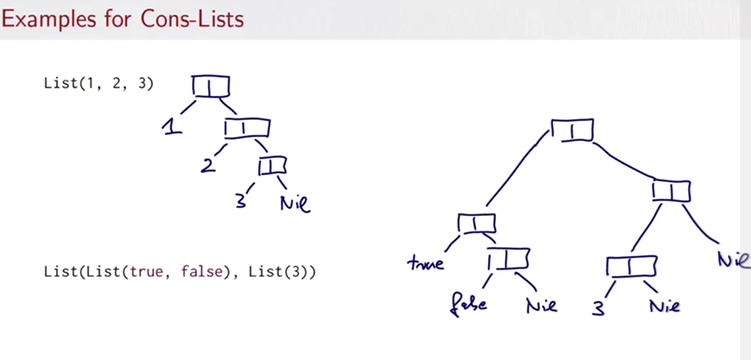
Higher-order functions in terms of objects ?

Cons-Lists in Scala:

A fundamental data structure in many functional languages is the immutable linked List. It is constructed from two building blocks.

Nil the empty list

Cons a cell containing an element and the remainder of the list.



package some\_package

trait IntList …

class Cons(val head: Int, val tail: IntList) extends IntLists …

class Nil extends IntList …

Alist is either

1. An empty list new Nil, or
2. A list new Cons(x, xs) consisting of a head element x and a tail list xs.

Value parameters:

Note that abbreviation (val head: Int, val tail: IntList) in the definition of Cons. This defines at the same time parameters and fields of a class. It is equivalent to ,

class Cons (\_head: Int, \_tail: IntList) extends IntList {

val head =\_head

tail = \_tail

}

Where \_head and \_tail are otherwise unused names.

**Generic Functions:**

Like classes, functions can have type parameters. For instances, here is a fuction that creates a lise of a single element.

def singleton[T](elem: T) = new Cons[T](elem, new Nil[T])

We can than Write:

1. Singleton[Int](1)
2. Singleton[Boolean](true)

In fact, the scala complier can usually deduce the correct type parameters from the value arguments of a function call. So, in most cases, type parameters can be left out. You could also write:

1. Singleton(1)
2. Singleton(true)

**Types and Evaluation:**

Type parameters do not affect evaluation in Scala. We can assume that all type parameters and type arguments are removed before evaluating the program. This is also called “type erasure”

EX: Java,scala,Haskell,ML

**Polymorphism:**

Polymorphism means that a function type come “in many forms”. In programming it means that

* The function can be applied to arguments of many types, or
* The type can have instances of many types.

We have seen two principal forms of polymorphism:

1. Subtyping: Instance of a subclass can be passed to a base class.
2. Generics: instances of a function or class are created by type parameterization.

Pure Object Orinetation:

A pure object-oriented language is one in which every value is an object.

If the language is based on classes, this means that the type of each value is a class.

Is Scala a Pure object-oriented language ?

Conceptually, types such as Int or Boolean do not receive special treatment in Scala. They are like the other classes, defined in the scala package scala. For reason of efficiency, the scala complier represents the values of

Of type scala. Int by 32 bit integers and calues

## **Scala Cheat Sheet**

<https://www.coursera.org/learn/progfun1/supplement/Sauv3/cheat-sheet>

<https://github.com/lrytz/progfun-wiki/blob/gh-pages/CheatSheet.md>

// Call by Value: evaluates the function arguments before calling the function

// call by name: evaluates the function first, and then evaluates the arguments if need be

def example = 2 // evaluated when called

val example = 2 // evaluated immediately

lazy val example = 2 // evaluated once when needed

def square(x: Double) // call by value

def square(x: => Double) // call by name

def myFct(bindings: Int\*) = { ... } // bindings is a sequence of int, containing a varying # of parameters

// Functions that take other functions as parameters or that return functions as a result are called higher

// order function

// First order fuction acts on the int, float, list.. and not on other function.

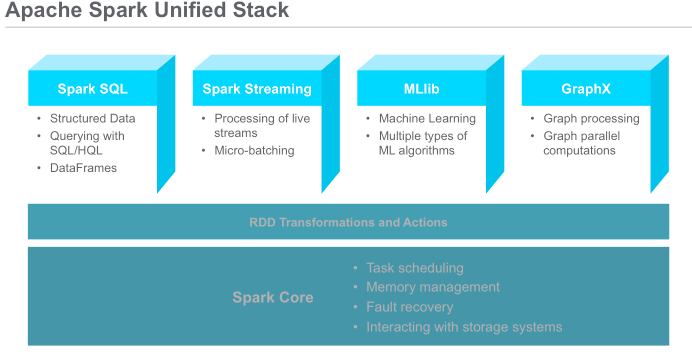
# Spark MapR

The Spark core is a computational engine that is responsible for taskscheduling, memory management, fault recovery and interacting withstorage systems. The Spark core contains the functionality of Spark.

It also contains the APIs that are used to define RDDs andmanipulate them.

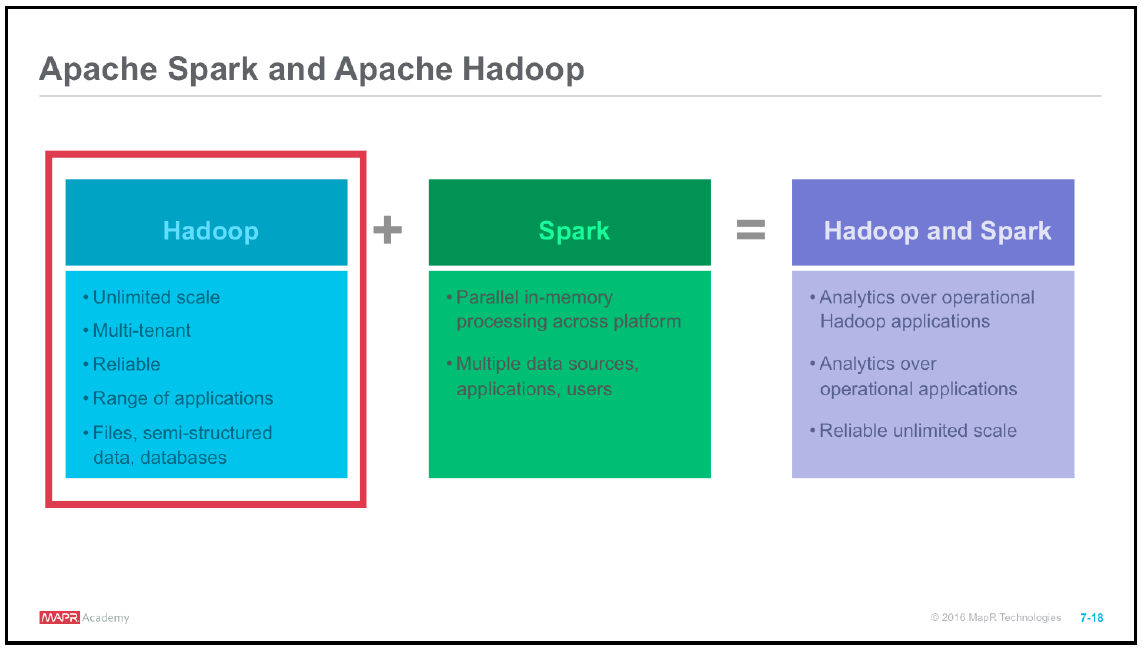
The higher level components are integrated into this stack.

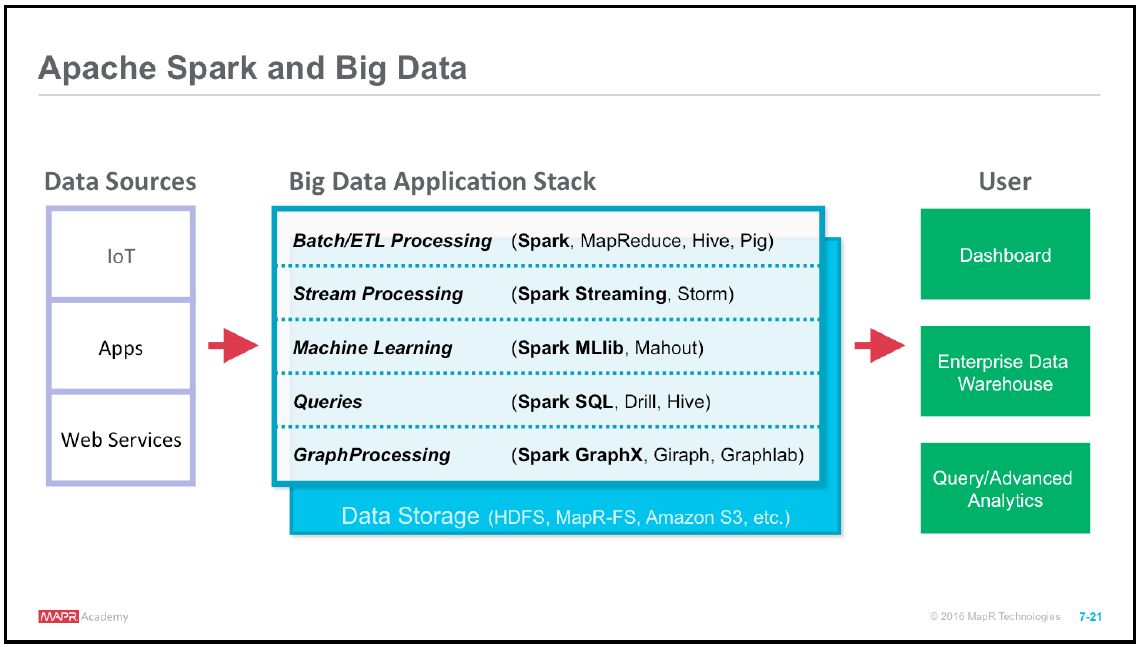
* **Spark SQL** can be used for working with structured data. You canquery this data via SQL or HiveQL. Spark SQL supports manytypes of data sources such as structured Hive tables and complexJSON data.
* **Spark streaming** enables processing of live streams of data anddoing real-time analytics.
* **MLlib** is a machine learning library that provides multiple types ofmachine learning algorithms such as classification, regression,clustering.
* **GraphX** is a library for manipulating graphs and performinggraph-parallel computations.



## Benefits of Apache Spark Over Hadoop

The combination of Spark and Hadoop takes advantage of both platforms, providing reliable, scalable, and fast parallel, in-memory processing. Additionally, you can easilycombine different kinds of workflows to provide analytics over Hadoop and otheroperational applications.





On the left we see different data sources. There are multiple ways of ingesting data,using different industry standards such as NFS or existing Hadoop tools.

The stack in the middle represents various big data processing workflows and tools that are commonly used. You may have just one of these workflows in yourapplication, or a combination of many. Any of these workflows could read/write to orfrom the storage layer.

While you can combine various workflows in Hadoop, it requires using different languages and different tools. As you can see here, with Spark, you can use Sparkfor any of the workflows. You can build applications that ingest streaming data andapply machine learning and graph analysis to the live streams. All this can be done inthe same application using one language.

The output can then be used to create real-time dashboards and alerting systems for querying and advanced analytics.

### Data Exploration-1(MAPR Lab 2\_1)

val auctionid = 0

val bid = 1

val bidtime = 2

val bidder = 3

val bidderrate = 4

val openbid = 5

val price = 6

val itemtype = 7

val daystolive = 8

*//load the data*

val auctionRDD = sc.textFile("/user/user01/data/auctiondata.csv").map(\_.split(","))

*//1. see the first element of the RDD*

auctionRDD.first

*// 2. First five element of the RDD*

auctionRDD.take(5)

*//3. What is the total number of bids?*

val totbids = auctionRDD.count()

*//totbids: Long = 10654*

*//4. What is the total number of distinct items that were auctioned?*

val totitems = auctionRDD.map(\_(auctionid)).distinct().count()

*//totitems: Long = 627*

*//5. What is the total number of item types that were auctioned?*

val itemtypes = auctionRDD.map(\_(itemtype)).distinct().count()

*//totitemtypes: Long = 3*

*//6. What is the total number of bids per item type?*

val bids\_itemtype = auctionRDD.map(x=>(x(itemtype),1)).reduceByKey((x,y)=>x+y).collect()

*//bids\_itemtype: Array[(String, Int)] = Array((palm,5917), (cartier,1953), (xbox,2784))*

*//7. What is the total number of bids per auction?*

val bids\_auctionRDD = auctionRDD.map(x=>(x(auctionid),1)).reduceByKey((x,y)=>x+y)

*//For #8, 9 - if you use Math.max, etc, then impor java.lang.Math*

import java.lang.Math

*// 8. Across all auctioned items, what is the max number of bids?*

val maxbids = bids\_auctionRDD.map(x=>x.\_2).reduce((x,y)=>Math.max(x,y))

*//maxbids: Int = 75*

*//9. Across all auctioned items, what is the minimum of bids?*

val minbids = bids\_auctionRDD.map(x=>x.\_2).reduce((x,y)=>Math.min(x,y))

*//minbids: Int = 1*

*//10. What is the average bid?*

val avgbids = totbids/totitems

*//avgbids: Long = 16*

### Data Exploration-2 (MAPR Lab 2\_2)

val sqlContext = new org.apache.spark.sql.SQLContext(sc)

import sqlContext.implicits.\_

*//Defining the Auctions case class*

case class Auctions(aucid:String, bid:Float,bidtime:Float,bidder:String,bidrate:Int,openbid:Float, price:Float,itemtype:String,dtl:Int)

*//Loading the data into RDD with split*

val inputRDD =sc.textFile("/user/user01/data/auctiondata.csv").map(\_.split(","))

*// Mapping the inputRDD to the case class*

val auctionsRDD = inputRDD.map(a=>Auctions(a(0),a(1).toFloat,a(2).toFloat,a(3),a(4).toInt, (5).toFloat,a(6).toFloat,a(7),a(8).toInt))

*// converting auctionsRDD to a DataFrame*

val auctionsDF = auctionsRDD.toDF()

*//Registering the auctionsDF as a temporary table with the same name*

auctionsDF.registerTempTable("auctionsDF")

*//8. Check the data in the DataFrame*

auctionsDF.show

*//9. TO see the schema of the DataFrame*

auctionsDF.printSchema

*//2.2.2 - Inspect Data*

*//1. total number of bids*

val totalbids = auctionsDF.count()

*//2. Number of distinct auctions*

val totalauctions = auctionsDF.select("aucid").distinct.count()

*//3. Number of distinct item types*

val itemtypes = auctionsDF.select("itemtype").distinct.count()

*//4. Count of bids per auction and item type*

auctionsDF.groupBy("itemtype","aucid").count().show()

*//(you could also use take(n))*

*//5.For each auction item and item type, want the min, max and average number of bids*

auctionsDF.groupBy("itemtype", "aucid").count.agg(min("count"), avg("count"), max("count")).show

*//6. For each auction item and item type- min*

bid, max bid, avg bid

auctionsDF.groupBy("itemtype", "aucid").agg(min("bid"), max("bid"), avg("bid")).show

*//7. Return the count of all auctions with final price greater than 200*

auctionsDF.filter(auctionsDF("price")>200).count()

*//8. Getting DF just for xboxes*

val xboxes = sqlContext.sql("SELECT aucid,itemtype,bid,price,openbid FROM auctionsDF WHERE itemtype='xbox'")

*// compute basic statistics on price across all auctions on xboxes*

xboxes.describe("price").show

### Data Exploration-3(MapR Lab4)

*/\*\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\*/*

*/\*\*\*\*\*\*\*\*\*\*\*\*Lab 4.1.2 - Load Data into Apache Spark \*\*\*\*\*\*\*\*\*\*/*

*//Map input variables*

val IncidntNum = 0

val Category = 1

val Descript = 2

val DayOfWeek = 3

val Date = 4

val Time = 5

val PdDistrict = 6

val Resolution = 7

val Address = 8

val X = 9

val Y = 10

val PdId = 11

*//Load SFPD data into RDD*

val sfpdRDD = sc.textFile("/user/user01/data/sfpd.csv").map(line=>line.split(","))

/\*\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\*/

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*LAB 4.2.1 Explore Data Using RDD Operations\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

*//1. How do you see the first element of the inputRDD?*

sfpdRDD.first()

*//2.What do you use to see the first 5 elements of the RDD?*

sfpdRDD.take(5)

*//3. What is the total number of incidents?*

val totincs = sfpdRDD.count()

*//4. What is the total number of distinct resolutionss?*

val totres = sfpdRDD.map(inc=>inc(Resolution)).distinct.count

*//4. List all the Districts.*

val dists = sfpdRDD.map(inc=>inc(PdDistrict)).distinct

dists.collect

/\*\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\*/

/\*\*\*\*\*\*\*\*\*\*\*\*\*LAB 4.3.1 - Create PAIR RDDs & apply pairRDD operations\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

*//1. Which five districts have the highest incidents?*

val top5Dists = sfpdRDD.map(incident=>(incident(PdDistrict),1)).reduceByKey((x,y)=>x+y).map(x=>(x.\_2,x.\_1)).sortByKey(false).take(5)

*//2. Which five addresses have the highest number of incidents?*

val top5Adds = sfpdRDD.map(incident=>(incident(Address),1)).reduceByKey((x,y)=>x+y).map(x=>(x.\_2,x.\_1)).sortByKey(false).take(5)

*//3. What are the top three catefories of incidents?*

val top3Cat = sfpdRDD.map(incident=>(incident(Category),1)).reduceByKey((x,y)=>x+y).map(x=>(x.\_2,x.\_1)).sortByKey(false).take(3)

*//4. What is the count of incidents by district?*

val num\_inc\_dist = sfpdRDD.map(incident=>(incident(PdDistrict),1)).countByKey()

/\*\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\*/

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*Lab 4.3.2. Join PairRDD\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

*//5. Load each dataset into separate pairRDDs with â€œaddressâ€ being the key?*

val catAdd = sc.textFile("/user/user01/data/J\_AddCat.csv").map(x=>x.split(",")).map(x=>(x(1),x(0)))

val distAdd = sc.textFile("/user/user01/data/J\_AddDist.csv").map(x=>x.split(",")).map(x=>(x(1),x(0)))

*//6. List the incident category and district for for those addresses that have both category and district information. Verify that the size estimated earlier is correct.*

val catJdist = catAdd.join(distAdd)

catJdist.collect

catJdist.count

catJdist.take(10)

*//7. List the incident category and district for all addresses irrespective of whether each address has category and district information. Verify that the size estimated earlier is correct.*

val catJdist1 = catAdd.leftOuterJoin(distAdd)

catJdist1.collect

catJdist.count

*//8. List the incident district and category for all addresses irrespective of whether each address has category and district information. Verify that the size estimated earlier is correct.*

val catJdist2 = catAdd.rightOuterJoin(distAdd)

catJdist2.collect

catJdist2.count

/\*\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\*/

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*Lab 4.4.1. Join PairRDD\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

*//1. How many partitions are there in the sfpdRDD?*

sfpdRDD.partitions.size

*//2. How do you find the type of partitioner?*

sfpdRDD.partitioner

*//3. Create a pair RDD - INcidents by district*

val incByDists = sfpdRDD.map(incident=>(incident(PdDistrict),1)).reduceByKey((x,y)=>x+y)

*//How many partitions does this have?*

incByDists.partitions.size

//What is the type of partitioner?

*incByDists.partitioner*

*//4. Now add a map transformation*

val inc\_map = incByDists.map(x=>(x.\_2,x.\_1))

*//Is there a change in the size?*

inc\_map.partitions.size

*//What about the type of partitioner?*

inc\_map.partitioner

*//5.Add sortByKey()*

val inc\_sort=incByDists.incByDists.map(x=>(x.\_2,x.\_1)).sortByKey(false)

*//type of partitioner*

inc\_sort.partitioner

*//6. Add groupByKey*

val inc\_group = sfpdRDD.map(incident=>(incident(PdDistrict),1)).groupByKey()

//type of partitioner

*//7. specify partition size in the transformation*

val incByDists = sfpdRDD.map(incident=>(incident(PdDistrict),1)).reduceByKey((x,y)=>x+y,10)

//number of partitions

incByDists.partitions.size

*//8. Create 2 pairRDD*

val catAdd = sc.textFile("/user/user01/data/J\_AddCat.csv").map(x=>x.split(",")).map(x=>(x(1),x(0)))

val distAdd = sc.textFile("/user/user01/data/J\_AddDist.csv").map(x=>x.split(",")).map(x=>(x(1),x(0)))

*//9. join and specify partitions- then check the number of partitions and the partitioner*

val catJdist=catAdd.join(distAdd,8)

catJdist.partitions.size

catjDist.partitioner

### Data Exploration-4(MapR Lab5)

/\*\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\*/

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*Lab 5.1 \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

import sqlContext.\_

import sqlContext.implicits.\_

val sfpdRDD = sc.textFile("/user/user01/data/sfpd.csv").map(inc=>inc.split(","))

case class Incidents(incidentnum:String, category:String, description:String, dayofweek:String, date:String, time:String, pddistrict:String, resolution:String, address:String, X:Float, Y:Float, pdid:String)

val sfpdCase=sfpdRDD.map(inc=>Incidents(inc(0),inc(1), inc(2),inc(3),inc(4),inc(5),inc(6),inc(7),inc(8),inc(9).toFloat,inc(10).toFloat, inc(11)))

val sfpdDF=sfpdCase.toDF()

sfpdDF.registerTempTable("sfpd")

/\*\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\*/

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*Lab 5.2 Explore data in DataFrames \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

*//1. Top 5 Districts*

val incByDist = sfpdDF.groupBy("pddistrict").count.sort($"count".desc).show(5)

val topByDistSQL = sqlContext.sql("SELECT pddistrict, count(incidentnum) AS inccount FROM sfpd GROUP BY pddistrict ORDER BY inccount DESC LIMIT 5")

*//2. What are the top ten resolutions?*

val top10Res = sfpdDF.groupBy("resolution").count.sort($"count".desc)

top10Res.show(10)

val top10ResSQL = sqlContext.sql("SELECT resolution, count(incidentnum) AS inccount FROM sfpd GROUP BY resolution ORDER BY inccount DESC LIMIT 10")

*//3. Top 3 categories*

val top3Cat = sfpdDF.groupBy("category").count.sort($"count".desc).show(3)

val top3CatSQL=sqlContext.sql("SELECT category, count(incidentnum) AS inccount FROM sfpd GROUP BY category ORDER BY inccount DESC LIMIT 3")

*//4. Save the top 10 resolutions to a JSON file.*

top10ResSQL.toJSON.saveAsTextFile("/user/user01/output")

/\*\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\*/

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*Lab 5.3 User Defined Functions \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

*//5.3.1 - UDF with SQL*

*//1. define funciton*

def getyear(s:String):String = {

val year = s.substring(s.lastIndexOf('/')+1)

year

}

*//2. register the function as a udf*

sqlContext.udf.register("getyear",getyear \_)

*//3. count inc by year*

val incyearSQL=sqlContext.sql("SELECT getyear(date), count(incidentnum) AS countbyyear FROM sfpd GROUP BY getyear(date) ORDER BY countbyyear DESC")

incyearSQL.collect.foreach(println)

*//4. Category, resolution and address of reported incidents in 2014*

val inc2014 = sqlContext.sql("SELECT category,address,resolution, date FROM sfpd WHERE getyear(date)='14'")

inc2014.collect.foreach(println)

*//5. Vandalism only in 2014 with address, resolution and category*

val van2015 = sqlContext.sql("SELECT category,address,resolution, date FROM sfpd WHERE getyear(date)='15' AND category='VANDALISM'")

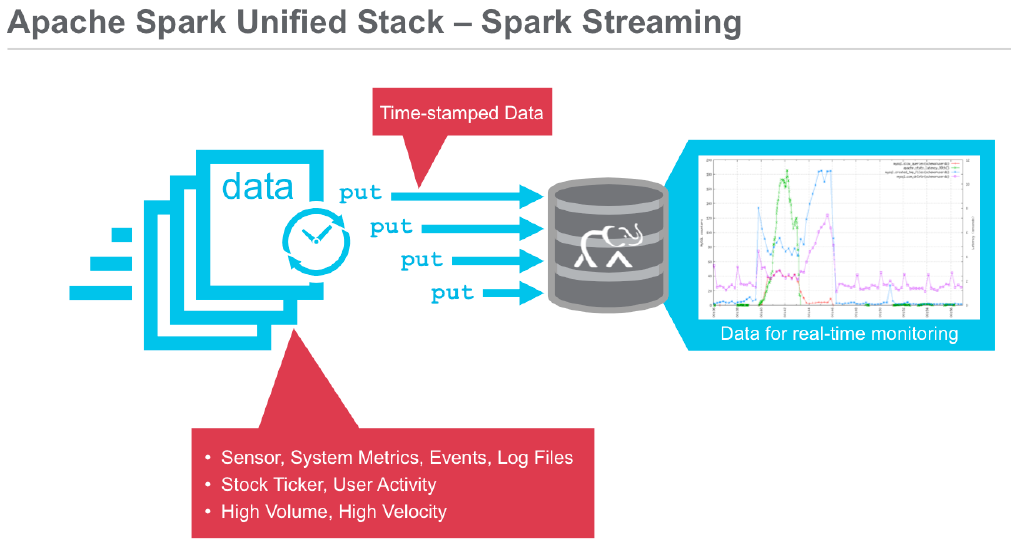
van2015.collect.foreach(println)

van2015.count

## Spark-Sql

Spark SQL is used to process structured data.Spark provides native support for SQL queries. You can query data via SQL orHiveQL. Spark SQL supports many types of data sources, such as structured Hivetables and complex JSON data. The primary abstraction of Spark SQL is SparkDataFrames.

## Spark- Streaming



Spark streaming enables processing of live streams of data.When performing analytics on the live data streams, Spark streaming uses a microbatchexecution model.

Many applications need to process streaming data with the following requirements:

* The results should be in near-real-time
* Be able to handle large workloads and
* Have latencies of few seconds

Batch processing on the other hand processes large volumes of data collected over aperiod of time, where the latency is in terms of minutes.Common examples of streaming data include activity stream data from a web or

mobile application, time stamped log data, transactional data and event streams fromsensor device networks.

Real-time applications of stream processing include website monitoring, networkmonitoring, fraud detection, web clicks and advertising.

### Spark Streaming Use Case 1(MAPR-LAB8)

Lab 8.1: Load and Inspect Data using the Spark Shell

Lab 8.2: Use Spark Streaming with the Spark Shell

Lab 8.3: Build and Run a Spark Streaming Application

Lab 8.4: Build and Run a Streaming Application with SQL

Lab 8.5: Build and Run a Streaming Application with Windows and SQL

**Lab 8.1: Load and Inspect Data using the Spark Shell**

*// SQLContext entry point for working with structured data*

val sqlContext = new org.apache.spark.sql.SQLContext(sc)

*// this is used to implicitly convert an RDD to a DataFrame.*

import sqlContext.implicits.\_

*// Import Spark SQL data types and Row.*

import org.apache.spark.sql.\_

*// load the data into a new RDD*

val textRDD = sc.textFile("/user/user01/data/sensordata.csv")

*// Return the first element in this RDD*

textRDD.take(1)

*//define the schema using a case class*

case class Sensor(resid: String, date: String, time: String, hz: Double, disp: Double, flo: Double,

sedPPM: Double, psi: Double, chlPPM: Double)

*// function to parse line of sensor data into Sensor class*

def parseSensor(str: String): Sensor = {

val p = str.split(",")

Sensor(p(0), p(1), p(2), p(3).toDouble, p(4).toDouble, p(5).toDouble, p(6).toDouble,

p(7).toDouble, p(8).toDouble)

}

*// create an RDD of sensor objects*

val sensorRDD= textRDD.map(parseSensor)

*// The RDD first() action returns the first element in the RDD*

sensorRDD.take(1)

*// Return the number of elements in the RDD*

sensorRDD.count()

*// create an alert RDD for when psi is low*

val alertRDD = sensorRDD.filter(sensor => sensor.psi < 5.0)

*// print some results*

alertRDD.take(3).foreach(println)

*//Now we will get some statistics on the data*

*// transform into an RDD of (key, values) to get daily stats for psi*

val keyValueRDD=sensorRDD.map(sensor => ((sensor.resid,sensor.date),sensor.psi))

*// print out some data*

keyValueRDD.take(3).foreach(kv => println(kv))

*// use StatCounter utility to get statistics for sensor psi*

val keyStatsRDD = keyValueRDD.groupByKey().mapValues(psi => StatCounter(psi))

*// print out some data*

keyStatsRDD.take(5).foreach(println)

*//A DataFrame is a distributed collection of data organized into named columns. Spark SQL supports*

*//automatically converting an RDD containing case classes to a DataFrame with the method toDF():*

*// change to a DataFrame*

val sensorDF = sensorRDD.toDF()

**Explore the data set with queries**

*// group by the sensorid, date get average psi*

sensorDF.groupBy("resid", "date").agg(avg(sensorDF("psi"))).take(5).foreach(println)

// register as a temp table then you can query

sensorDF.registerTempTable("sensor")

*// get the max , min, avg for each column*

val sensorStatDF = sqlContext.sql("SELECT resid, date,MAX(hz) as maxhz, min(hz) as minhz, avg(hz) as

avghz, MAX(disp) as maxdisp, min(disp) as mindisp, avg(disp) as avgdisp, MAX(flo) as maxflo, min(flo)

as minflo, avg(flo) as avgflo,MAX(sedPPM) as maxsedPPM, min(sedPPM) as minsedPPM, avg(sedPPM)

as avgsedPPM, MAX(psi) as maxpsi, min(psi) as minpsi, avg(psi) as avgpsi,MAX(chlPPM) as

maxchlPPM, min(chlPPM) as minchlPPM, avg(chlPPM) as avgchlPPM FROM sensor GROUP BY

resid,date")

*// print out the results*

sensorStatDF.take(5).foreach(println)

**Lab 8.2: Use Spark Streaming with the Spark Shell**

$ mkdir stream

*//First we will import some packages*

import org.apache.spark.SparkConf

import org.apache.spark.streaming.{Seconds, StreamingContext}

import StreamingContext.\_

*//First we create a StreamingContext, the main entry point for streaming functionality, //with a 2 second batch //interval. Next, we use the StreamingContext //textFileStream(directory) method to create an input stream*

val ssc = new StreamingContext(sc, Seconds(2))

val textDStream = ssc.textFileStream("/user/user01/stream")

textDStream.print()

case class Sensor(resid: String, date: String, time: String, hz: Double, disp: Double, flo: Double, sedPPM: Double, psi: Double, chlPPM: Double)extends Serializable

*//Next we use the DStream foreachRDD method to apply processing to each RDD in this DStream.*

*// for each RDD. performs function on each RDD in DStream*

textDStream.foreachRDD(rdd=>{

val srdd =rdd.map(\_.split(",")).map(p => Sensor(p(0), p(1), p(2), p(3).toDouble, p(4).toDouble, p(5).toDouble,

p(6).toDouble, p(7).toDouble, p(8).toDouble))

srdd.take(2).foreach(println)

})

*// Start the computation*

ssc.start()

*// Wait for the computation to terminate*

ssc.awaitTermination()

**Observe Streaming Application in Web UI**

Launch the Spark Streaming UI in the browser with you sandbox or cluster ipaddress and port 4040:

http:://ipaddress:4040. Click on the Streaming tab. Note if you are running on the cluster port 4040 may already be taken by another user, you can see which port you got when the shell starts.

Lab 8.3: Build and Run a Spark Streaming Application

Lab 8.4: Build and Run a Streaming Application with SQL

Lab 8.5: Build and Run a Streaming Application with Windows and SQL

## Spark- MLlib

MLlib is the Spark machine learning library. It supports common learning algorithmssuch as classification, regressions, clustering, collaborative filtering anddimensionality reduction.There are two packages available,

* **spark.mllib** – original API that can be used on RDDs
* **spark.ml** – higher level API that can be used on DataFrames

### MLLIB Use case1

*// SQLContext entry point for working with structured data*

val sqlContext = new org.apache.spark.sql.SQLContext(sc)

*// this is used to implicitly convert an RDD to a DataFrame.*

import sqlContext.implicits.\_

*// Import Spark SQL data types*

import org.apache.spark.sql.\_

*// Import MLLIB data types*

import org.apache.spark.mllib.recommendation.{ALS, MatrixFactorizationModel, Rating}

*// define the schemas using a case classes*

*// input format MovieID::Title::Genres*

case class Movie(movieId: Int, title: String)

*// input format UserID::Gender::Age::Occupation::Zip-code*

case class User(userId: Int, gender: String, age: Int, occupation: Int, zip: String)

*// function to parse input into Movie class*

def parseMovie(str: String): Movie = {

val fields = str.split("::")

assert(fields.size == 3)

Movie(fields(0).toInt, fields(1))

}

*// function to parse input into User class*

def parseUser(str: String): User = {

val fields = str.split("::")

assert(fields.size == 5)

User(fields(0).toInt, fields(1).toString, fields(2).toInt, fields(3).toInt, fields(4).toString)

}

*// function to parse input UserID::MovieID::Rating*

*// and pass into constructor for org.apache.spark.mllib.recommendation.Rating class*

def parseRating(str: String): Rating = {

val fields = str.split("::")

Rating(fields(0).toInt, fields(1).toInt, fields(2).toDouble)

}

*// load the data into an RDD*

val ratingText = sc.textFile("/user/user01/data/ratings.dat")

val ratingsRDD = ratingText.map(parseRating).cache()

*// count number of total ratings*

val numRatings = ratingsRDD.count()

*// count number of users who rated a movie*

val numUsers = ratingsRDD.map(\_.user).distinct().count()

*// count number of movies rated*

val numMovies = ratingsRDD.map(\_.product).distinct().count()

println(s"Got $numRatings ratings from $numUsers users on $numMovies movies.")

*// load the data into DataFrames*

val moviesDF= sc.textFile("/user/user01/data/movies.dat").map(parseMovie).toDF()

val usersDF = sc.textFile("/user/user01/data/users.dat").map(parseUser).toDF()

*// create a DataFrame from ratingsRDD*

val ratingsDF = ratingsRDD.toDF()

ratingsDF.registerTempTable("ratings")

moviesDF.registerTempTable("movies")

usersDF.registerTempTable("users")

ratingsDF.select("product").distinct.count

ratingsDF.groupBy("product", "rating").count.show

ratingsDF.groupBy("product").count.agg(min("count"), avg("count"),max("count")).show

ratingsDF.select("product", "rating").groupBy("product", "rating").count.agg(min("count"), avg("count"),max("count")).show

*// Count the max, min ratings along with the number of users who have rated a movie. Display the title, max rating, min rating, number of users.*

val results =sqlContext.sql("select movies.title, movierates.maxr, movierates.minr, movierates.cntu from(SELECT ratings.product, max(ratings.rating) as maxr, min(ratings.rating) as minr,count(distinct user) as cntu FROM ratings group by ratings.product ) movierates join movies on movierates.product=movies.movieId order by movierates.cntu desc ")

*// Show the top 10 most-active users and how many times they rated a movie*

val mostActiveUsersSchemaRDD = sqlContext.sql("SELECT ratings.user, count(\*) as ct from ratings group by ratings.user order by ct desc limit 10")

mostActiveUsersSchemaRDD.take(20).foreach(println)

val results =sqlContext.sql("SELECT ratings.user, ratings.product, ratings.rating, movies.title FROM ratings JOIN movies ON movies.movieId=ratings.product where ratings.user=4169 and ratings.rating > 4 order by ratings.rating desc ")

*// Randomly split ratings RDD into training data RDD (80%) and test data RDD (20%)*

val splits = ratingsRDD.randomSplit(Array(0.8, 0.2), 0L)

val trainingRatingsRDD = splits(0).cache()

val testRatingsRDD = splits(1).cache()

val numTraining = trainingRatingsRDD.count()

val numTest = testRatingsRDD.count()

println(s"Training: $numTraining, test: $numTest.")

*// Build the recommendation model using ALS with rank=20, iterations=10*

val model = ALS.train(trainingRatingsRDD, 20, 10)

val model = (new ALS().setRank(20).setIterations(10).run(trainingRatingsRDD))

*// Make movie predictions for user 4169*

val topRecsForUser = model.recommendProducts(4169, 10)

*// get movie titles to show with recommendations*

val movieTitles=moviesDF.map(array => (array(0), array(1))).collectAsMap()

*// print out top recommendations for user 4169 with titles*

topRecsForUser.map(rating => (movieTitles(rating.product), rating.rating)).foreach(println)

*// get predicted ratings to compare to test ratings*

val predictionsForTestRDD = model.predict(testRatingsRDD.map{case Rating(user, product, rating) => (user, product)})

predictionsForTestRDD.take(10).mkString("\n")

*// prepare the predictions for comparison*

val predictionsKeyedByUserProductRDD = predictionsForTestRDD.map{

case Rating(user, product, rating) => ((user, product), rating)

}

*// prepare the test for comparison*

val testKeyedByUserProductRDD = testRatingsRDD.map{

case Rating(user, product, rating) => ((user, product), rating)

}

*//Join the test with the predictions*

val testAndPredictionsJoinedRDD = testKeyedByUserProductRDD.join(predictionsKeyedByUserProductRDD)

testAndPredictionsJoinedRDD.take(10).mkString("\n")

val falsePositives =(testAndPredictionsJoinedRDD.filter{

case ((user, product), (ratingT, ratingP)) => (ratingT <= 1 && ratingP >=4)

})

*//Evaluate the model using Mean Absolute Error (MAE) between test and predictions*

val meanAbsoluteError = testAndPredictionsJoinedRDD.map {

case ((user, product), (testRating, predRating)) =>

val err = (testRating - predRating)

Math.abs(err)

}.mean()

### MLLIB Use case2

import org.apache.spark.\_

import org.apache.spark.rdd.RDD

import org.apache.spark.util.IntParam

*// Import classes for MLLib regression labeledpoint, vectors, decisiontree, decisiontree model, MLUtils*

import org.apache.spark.mllib.regression.LabeledPoint

import org.apache.spark.mllib.linalg.Vectors

import org.apache.spark.mllib.tree.DecisionTree

import org.apache.spark.mllib.tree.model.DecisionTreeModel

import org.apache.spark.mllib.util.MLUtils

case class Flight(dofM: String, dofW: String, carrier: String, tailnum: String, flnum: Int, org\_id: String, origin: String, dest\_id: String, dest: String, crsdeptime: Double, deptime: Double, depdelaymins: Double, crsarrtime: Double, arrtime: Double, arrdelay: Double, crselapsedtime: Double, dist: Int)

*// function to parse input into Movie class*

def parseFlight(str: String): Flight = {

val line = str.split(",")

Flight(line(0), line(1), line(2), line(3), line(4).toInt, line(5), line(6), line(7), line(8), line(9).toDouble, line(10).toDouble, line(11).toDouble, line(12).toDouble, line(13).toDouble, line(14).toDouble, line(15).toDouble, line(16).toInt)

}

*//Creating and RDD with the January 2014 data to be used for training the model*

val textRDD = sc.textFile("/user/user01/data/rita2014jan.csv")

val flightsRDD = textRDD.map(parseFlight).cache()

flightsRDD.take(2)

*//Array(Flight(1,3,AA,N338AA,1,12478,JFK,12892,LAX,900.0,914.0,14.0,1225.0,1238.0,13.0,385.0,2475),*

*// Flight(2,4,AA,N338AA,1,12478,JFK,12892,LAX,900.0,857.0,0.0,1225.0,1226.0,1.0,385.0,2475))*

var carrierMap: Map[String, Int] = Map()

var index: Int = 0

flightsRDD.map(flight => flight.carrier).distinct.collect.foreach(x => { carrierMap += (x -> index); index += 1 })

carrierMap.toString

*//res2: String = Map(DL -> 5, F9 -> 10, US -> 9, OO -> 2, B6 -> 0, AA -> 6, EV -> 12, FL -> 1, UA -> 4, MQ -> 8, WN -> 13, AS -> 3, VX -> 7, HA -> 11)*

var originMap: Map[String, Int] = Map()

var index1: Int = 0

flightsRDD.map(flight => flight.origin).distinct.collect.foreach(x => { originMap += (x -> index1); index1 += 1 })

originMap.toString

*//res4: String = Map(JFK -> 214, LAX -> 294, ATL -> 273,MIA -> 175 ...*

var destMap: Map[String, Int] = Map()

var index2: Int = 0

flightsRDD.map(flight => flight.dest).distinct.collect.foreach(x => { destMap += (x -> index2); index2 += 1 })

*//- Defining the features array*

val mlprep = flightsRDD.map(flight => {

val monthday = flight.dofM.toInt - 1 // category

val weekday = flight.dofW.toInt - 1 // category

val crsdeptime1 = flight.crsdeptime.toInt

val crsarrtime1 = flight.crsarrtime.toInt

val carrier1 = carrierMap(flight.carrier) // category

val crselapsedtime1 = flight.crselapsedtime.toDouble

val origin1 = originMap(flight.origin) // category

val dest1 = destMap(flight.dest) // category

val delayed = if (flight.depdelaymins.toDouble > 40) 1.0 else 0.0

Array(delayed.toDouble, monthday.toDouble, weekday.toDouble, crsdeptime1.toDouble, crsarrtime1.toDouble, carrier1.toDouble, crselapsedtime1.toDouble, origin1.toDouble, dest1.toDouble)

})

mlprep.take(1)

*//res6: Array[Array[Double]] = Array(Array(0.0, 0.0, 2.0, 900.0, 1225.0, 6.0, 385.0, 214.0, 294.0))*

//Making LabeledPoint of features - this is the training data for the model

val mldata = mlprep.map(x => LabeledPoint(x(0), Vectors.dense(x(1), x(2), x(3), x(4), x(5), x(6), x(7), x(8))))

mldata.take(1)

*//res7: Array[org.apache.spark.mllib.regression.LabeledPoint] =* Array((0.0,[0.0,2.0,900.0,1225.0,6.0,385.0,214.0,294.0]))

*// mldata0 is %85 not delayed flights*

val mldata0 = mldata.filter(x => x.label == 0).randomSplit(Array(0.85, 0.15))(1)

*// mldata1 is %100 delayed flights*

val mldata1 = mldata.filter(x => x.label != 0)

*// mldata2 is delayed and not delayed*

val mldata2 = mldata0 ++ mldata1

mldata2.count

res27: Long = 116692

*// split mldata2 into training and test data*

val splits = mldata2.randomSplit(Array(0.7, 0.3))

val (trainingData, testData) = (splits(0), splits(1))

trainingData.count

res26: Long = 81727

*// set ranges for 0=dofM 1=dofW 4=carrier 6=origin 7=dest*

var categoricalFeaturesInfo = Map[Int, Int]()

categoricalFeaturesInfo += (0 -> 31)

categoricalFeaturesInfo += (1 -> 7)

categoricalFeaturesInfo += (4 -> carrierMap.size)

categoricalFeaturesInfo += (6 -> originMap.size)

categoricalFeaturesInfo += (7 -> destMap.size)

val numClasses = 2

*// Defning values for the other parameters*

val impurity = "gini"

val maxDepth = 9

val maxBins = 7000

val model = DecisionTree.trainClassifier(trainingData, numClasses, categoricalFeaturesInfo,

impurity, maxDepth, maxBins)

model.toDebugString

*// 0=dofM 4=carrier 3=crsarrtime1 6=origin*

res20: String =

DecisionTreeModel classifier of depth 9 with 919 nodes

If (feature 0 in {11.0,12.0,13.0,14.0,15.0,16.0,17.0,18.0,19.0,20.0,21.0,22.0,23.0,24.0,25.0,26.0,27.0,30.0})

If (feature 4 in {0.0,1.0,2.0,3.0,4.0,5.0,6.0,7.0,8.0,9.0,10.0,11.0,13.0})

If (feature 3 <= 1603.0)

If (feature 0 in {11.0,12.0,13.0,14.0,15.0,16.0,17.0,18.0,19.0})

If (feature 6 in {0.0,1.0,2.0,3.0,4.0,5.0,6.0,7.0,8.0,10.0,11.0,12.0,13.0...

testData.take(1)

*//res21: Array[org.apache.spark.mllib.regression.LabeledPoint] =* Array((0.0,[18.0,6.0,900.0,1225.0,6.0,385.0,214.0,294.0]))

val labelAndPreds = testData.map { point =>

val prediction = model.predict(point.features)

(point.label, prediction)

}

labelAndPreds.take(3)

res33: Array[(Double, Double)] = Array((0.0,0.0), (0.0,0.0), (0.0,0.0))

val wrongPrediction =(labelAndPreds.filter{

case (label, prediction) => ( label !=prediction)

})

wrongPrediction.count()

res35: Long = 11040

val ratioWrong=wrongPrediction.count().toDouble/testData.count()

ratioWrong: Double = 0.3157443157443157

testData.count

*//res28: Long = 34965*

*// find delay predicted when there was no delay*

val falsePositives =(labelAndPreds.filter{

case (label, prediction) => ( label==0 && prediction == 1)

})

falsePositives.count

*//res24: Long = 5489*

val falseNegatives =(labelAndPreds.filter(r => ( r.\_2==0 && r.\_1== 1) ))

falseNegatives.count

*//res34: Long = 5551*

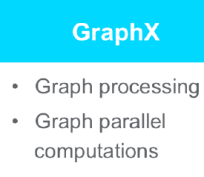
val fpRatio=falsePositives.count.toDouble/ testData.count()

fpRatio: Double = 0.156985556985557

val fnRatio=falseNegatives.count.toDouble/ testData.count()

fnRatio: Double = 0.15875875875875875

## Spark- GraphX

GraphX is a library for manipulating graphs and performing graph-parallelcomputations.

* The primary abstraction is a graph that extends the Spark RDD.
* A Graph is a directed multigraph wherein each vertex and edge has properties attached.
* GraphX has a number of operators that can be used for graph computations including an optimized version of Pregel.
* GraphX also provides graph algorithms and builders for graph analytics.
* We can work seamlessly with both graphs and collections; view same data as both graphs and collections without duplication or movement of data. We can also write custom iterative graph algorithms

### Graphx Use case1

import org.apache.spark.graphx.\_

import org.apache.spark.rdd.RDD

*//airlines*

val vertices=Array((1L, ("SFO")),(2L, ("ORD")),(3L,("DFW")))

val vRDD= sc.parallelize(vertices)

val edges = Array(Edge(1L,2L,1800),Edge(2L,3L,800),Edge(3L,1L,1400))

val eRDD= sc.parallelize(edges)

*// Array(Edge(1,2,1800), Edge(2,3,800))*

val nowhere = ("nowhere")

val graph = Graph(vRDD,eRDD, nowhere)

graph.vertices.collect.foreach(println)

*//(2,ORD)*

*//(1,SFO)*

*//(3,DFW)*

graph.edges.collect.foreach(println)

*//Edge(1,2,1800)*

*//Edge(2,3,800)*

*//Edge(3,1,1400)*

graph.triplets.collect.foreach(println)

*//((1,SFO),(2,ORD),1800)*

*//((2,ORD),(3,DFW),800)*

*//((3,DFW),(1,SFO),1400)*

println(graph.inDegrees)

*//VertexRDDImpl[83] at RDD at VertexRDD.scala:57*

val numairports = graph.numVertices

*//numairports: Long = 3*

val numroutes = graph.numEdges

*//numroutes: Long = 3*

*//How many routes distance greater than 1000?*

graph.edges.filter { case Edge(src, dst, prop) => prop > 1000 }.count

res17: Long = 2

*//which routes have distance greater than 1000?*

graph.edges.filter { case Edge(src, dst, prop) => prop > 1000 }.collect.foreach(println)

*//Edge(1,2,1800)*

*//Edge(3,1,1400)*

*//Sort and print out the longest distance routes*

graph.triplets.sortBy(\_.attr, ascending=false).map(triplet =>

"Distance " + triplet.attr.toString + " from " + triplet.srcAttr + " to " + triplet.dstAttr + ".").collect.foreach(println)

*//Distance 1800 from SFO to ORD.*

*//Distance 1400 from DFW to SFO.*

*//Distance 800 from ORD to DFW.*

*//What are the most important airports according to PageRank*

val ranks = graph.pageRank(0.1).vertices

ranks.take(3)

*//res0: Array[(org.apache.spark.graphx.VertexId, Double)] = Array((2,0.47799375), (1,0.47799375), (3,0.47799375))*

val impAirports = ranks.join(vRDD).sortBy(\_.\_2.\_1, false).map(\_.\_2.\_2)

*//ranksAndAirports: org.apache.spark.rdd.RDD[String] = MapPartitionsRDD[286] at map at <console>:44*

impAirports.collect.foreach(println)

*//ORD*

*//SFO*

*//DFW*

### Graphx Use case2

import org.apache.spark.\_

import org.apache.spark.rdd.RDD

import org.apache.spark.util.IntParam

*// import classes required for using GraphX*

import org.apache.spark.graphx.\_

import org.apache.spark.graphx.util.GraphGenerators

case class Flight(dofM:String, dofW:String, carrier:String, tailnum:String, flnum:Int, org\_id:Long, origin:String, dest\_id:Long, dest:String, crsdeptime:Double, deptime:Double, depdelaymins:Double, crsarrtime:Double, arrtime:Double, arrdelay:Double,crselapsedtime:Double,dist:Int)

def parseFlight(str: String): Flight = {

val line = str.split(",")

Flight(line(0), line(1), line(2), line(3), line(4).toInt, line(5).toLong, line(6), line(7).toLong, line(8), line(9).toDouble, line(10).toDouble, line(11).toDouble, line(12).toDouble, line(13).toDouble, line(14).toDouble, line(15).toDouble, line(16).toInt)

}

*//Create RDD with the January 2014 data*

val textRDD = sc.textFile("/user/user01/data/rita2014jan.csv")

val flightsRDD = textRDD.map(parseFlight).cache()

val airports = flightsRDD.map(flight => (flight.org\_id, flight.origin)).distinct

airports.take(1)

*// Array((14057,PDX))*

*// Defining a default vertex called nowhere*

val nowhere = "nowhere"

val routes = flightsRDD.map(flight => ((flight.org\_id, flight.dest\_id), flight.dist)).distinct

*// Array(((14869,14683),1087), ((14683,14771),1482))*

routes.cache

routes.take(1)

*//res79: Array[((Long, Long), Int)] = Array(((10299,10926),160))*

*// AirportID is numerical - Mapping airport ID to the 3-letter code*

val airportMap = airports.map { case ((org\_id), name) => (org\_id -> name) }.collect.toList.toMap

*//airportMap: scala.collection.immutable.Map[Long,String] = Map(13024 -> LMT, 10785 -> BTV, 14574 -> ROA, 14057 -> PDX, 13933 -> ORH, 11898 -> GFK, 14709 -> SCC, 15380 -> TVC,*

*// Defining the routes as edges*

val edges = routes.map { case ((org\_id, dest\_id), distance) => Edge(org\_id.toLong, dest\_id.toLong, distance) }

edges.take(1)

*//res80: Array[org.apache.spark.graphx.Edge[Int]] = Array(Edge(10299,10926,160))*

*//Defining the Graph*

val graph = Graph(airports, edges, nowhere)

*// LNumber of airports*

val numairports = graph.numVertices

*// numairports: Long = 301*

graph.vertices.take(2)

graph.edges.take(2)

*// res6: Array[org.apache.spark.graphx.Edge[Int]] = Array(Edge(10135,10397,692), Edge(10135,13930,654))*

*// which routes > 1000 miles distance?*

graph.edges.filter { case ( Edge(org\_id, dest\_id,distance))=> distance > 1000}.take(3)

*// res9: Array[org.apache.spark.graphx.Edge[Int]] = Array(Edge(10140,10397,1269), Edge(10140,10821,1670), Edge(10140,12264,1628))*

*// Number of routes*

val numroutes = graph.numEdges

*// numroutes: Long = 4090*

*// The EdgeTriplet class extends the Edge class by adding the srcAttr and dstAttr members which contain the source and destination properties respectively.*

graph.triplets.take(3).foreach(println)

*//((10135,ABE),(10397,ATL),692)*

*//((10135,ABE),(13930,ORD),654)*

*//((10140,ABQ),(10397,ATL),1269)*

*//Sort and print out the longest distance routes*

graph.triplets.sortBy(\_.attr, ascending=false).map(triplet =>

"Distance " + triplet.attr.toString + " from " + triplet.srcAttr + " to " + triplet.dstAttr + ".").take(10).foreach(println)

Distance 4983 from JFK to HNL.

Distance 4983 from HNL to JFK.

Distance 4963 from EWR to HNL.

Distance 4963 from HNL to EWR.

Distance 4817 from HNL to IAD.

Distance 4817 from IAD to HNL.

Distance 4502 from ATL to HNL.

Distance 4502 from HNL to ATL.

Distance 4243 from HNL to ORD.

Distance 4243 from ORD to HNL.

*// Define a reduce operation to compute the highest degree vertex*

def max(a: (VertexId, Int), b: (VertexId, Int)): (VertexId, Int) = {

if (a.\_2 > b.\_2) a else b

}

*// Compute the max degrees*

val maxInDegree: (VertexId, Int) = graph.inDegrees.reduce(max)

*// maxInDegree: (org.apache.spark.graphx.VertexId, Int) = (10397,152)*

val maxOutDegree: (VertexId, Int) = graph.outDegrees.reduce(max)

*// maxOutDegree: (org.apache.spark.graphx.VertexId, Int) = (10397,153)*

val maxDegrees: (VertexId, Int) = graph.degrees.reduce(max)

*// maxDegrees: (org.apache.spark.graphx.VertexId, Int) = (10397,305)*

airportMap(10397)

*// res70: String = ATL*

*// we can compute the in-degree of each vertex (defined in GraphOps) by the following:*

*// which airport has the most incoming flights?*

graph.inDegrees.collect.sortWith(\_.\_2 > \_.\_2).map(x => (airportMap(x.\_1), x.\_2))

*//res46: Array[(String, Int)] = Array((ATL,152), (ORD,145), (DFW,143), (DEN,132), (IAH,107), (MSP,96), (LAX,82), (EWR,82), (DTW,81), (SLC,80),*

val maxIncoming = graph.inDegrees.collect.sortWith(\_.\_2 > \_.\_2).map(x => (airportMap(x.\_1), x.\_2)).take(3)

maxIncoming.foreach(println)

(ATL,152)

(ORD,145)

(DFW,143)

*// which airport has the most outgoing flights?*

graph.outDegrees.join(airports).sortBy(\_.\_2.\_1, ascending=false).take(1)

val maxout= graph.outDegrees.join(airports).sortBy(\_.\_2.\_1, ascending=false).take(3)

maxout.foreach(println)

(10397,(153,ATL))

(13930,(146,ORD))

(11298,(143,DFW))

val maxOutgoing = graph.outDegrees.collect.sortWith(\_.\_2 > \_.\_2).map(x => (airportMap(x.\_1), x.\_2)).take(3)

maxOutgoing.foreach(println)

(ATL,152)

(ORD,145)

(DFW,143)

*//What are the most important airports according to PageRank?*

*// use pageRank*

val ranks = graph.pageRank(0.1).vertices

val impAirports = ranks.join(airports).sortBy(\_.\_2.\_1, false).map(\_.\_2.\_2)

impAirports.take(4)

*//res6: Array[String] = Array(ATL, ORD, DFW, DEN)*

graph.edges.filter { case ( Edge(org\_id, dest\_id,distance))=> distance > 1000}.take(3)

val sourceId: VertexId = 13024

*// 50 + distance / 20*

val gg = graph.mapEdges(e => 50.toDouble + e.attr.toDouble/20 )

val initialGraph = gg.mapVertices((id, \_) => if (id == sourceId) 0.0 else Double.PositiveInfinity)

val sssp = initialGraph.pregel(Double.PositiveInfinity)(

(id, dist, newDist) => math.min(dist, newDist), // Vertex Program

triplet => { // Send Message

if (triplet.srcAttr + triplet.attr < triplet.dstAttr) {

Iterator((triplet.dstId, triplet.srcAttr + triplet.attr))

} else {

Iterator.empty

}

},

(a,b) => math.min(a,b) // Merge Message

)

println(sssp.vertices.take(4).mkString("\n"))

(10208,277.79999999999995)

(10268,260.7)

(14828,261.65)

(14698,125.25)

println(sssp.edges.take(4).mkString("\n"))

Edge(10135,10397,84.6)

Edge(10135,13930,82.7)

Edge(10140,10397,113.45)

Edge(10140,10821,133.5)

sssp.edges.map{ case ( Edge(org\_id, dest\_id,price))=> ( (airportMap(org\_id), airportMap(dest\_id), price)) }.top(4)(Ordering.by(\_.\_3))

sssp.edges.map{ case ( Edge(org\_id, dest\_id,price))=> ( (airportMap(org\_id), airportMap(dest\_id), price)) }.takeOrdered(10)(Ordering.by(\_.\_3))

res21: Array[(String, String, Double)] = Array((WRG,PSG,51.55), (PSG,WRG,51.55), (CEC,ACV,52.8), (ACV,CEC,52.8), (ORD,MKE,53.35), (IMT,RHI,53.35), (MKE,ORD,53.35), (RHI,IMT,53.35), (STT,SJU,53.4), (SJU,STT,53.4))

sssp.vertices.collect.map(x => (airportMap(x.\_1), x.\_2)).sortWith(\_.\_2 < \_.\_2)

res21: Array[(String, Double)] = Array((LMT,0.0), (PDX,62.05), (SFO,65.75), (EUG,117.35)

## Spark- Execution Model

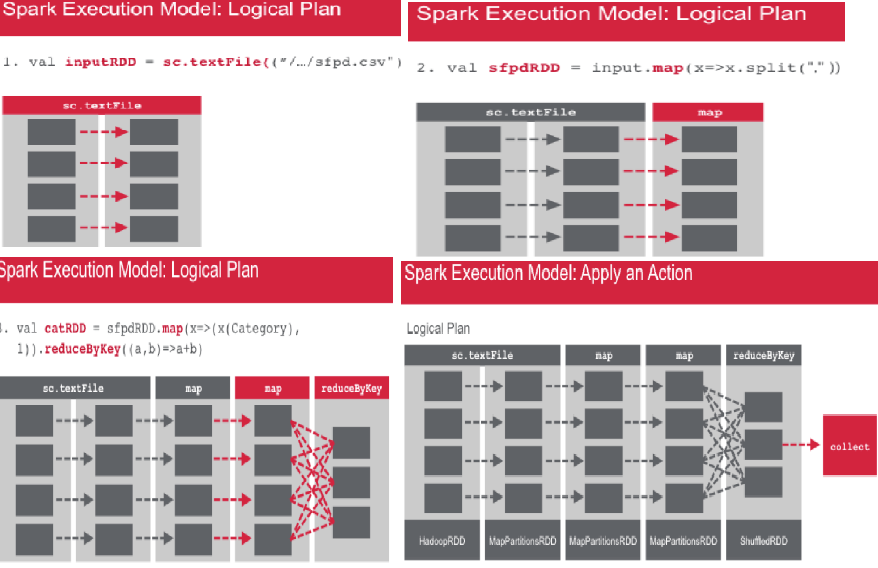
// /opt/mapr/spark/spark-<version>/bin/spark-shell

// val IncidntNum = 0;val Category = 1;val Descript = 2;val DayOfWeek = 3;val Date = 4;val Time = 5;val

// PdDistrict = 6;val Resolution = 7;val Address = 8;val X = 9;val Y = 10;val PdId = 11;

We are going to see how a user program translates into the units of physical execution. Let usfirst take a look at the logical plan.

### Logical Model



val inputRDD = sc.textFile("/pathtofile/sfpd.csv")

val sfpdRDD = input.map(\_.split(","))

val catRDD = sfpdRDD.map(col => (col(Category),1)).reduceByKey((x,y) => x+y)

val catRDD = sfpdRDD.map(col => (col(Category),1)).reduceByKey((x,y) => x+y).collect()

1. The first line creates an RDD called inputRDD from the sfpd.csv file.
2. The second line creates an RDD – sfpdRDD which splits the data in the input RDD based on the comma separator.
3. The third statement creates the catRDD by applying the map and reduceByKey transformations.No actions have been performed yet. Once Spark executes these lines, it defines a Directed

Acyclic Graph (DAG) of these RDD objects. Each RDD maintains a pointer to its parent(s) along

with the metadata about the type of relationship. RDDs use these pointer to trace its ancestors.

1. Now we add a collect action on the catRDD. The collect action triggers a computation. The

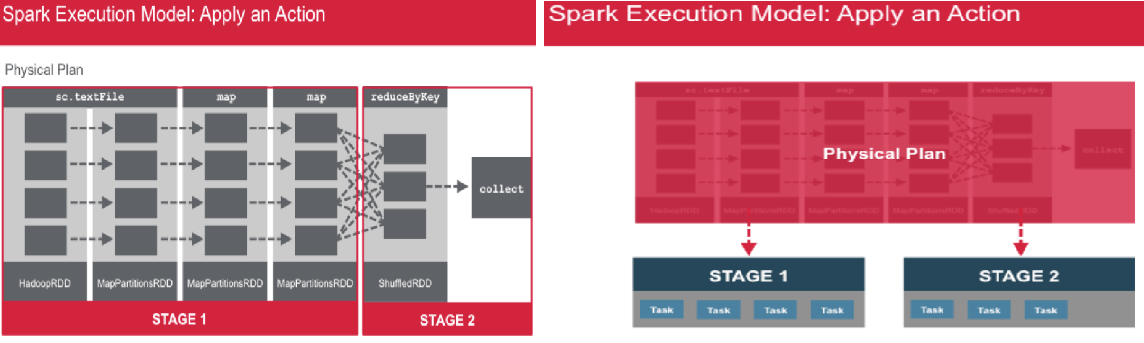
Spark scheduler creates a physical plan to compute the RDDs needed for the computation.

When the collect action is called, every partition of the RDD is materialized and transferred to

the driver program. The Spark scheduler then works backwards from the catRDD to create the

physical plan necessary to compute all ancestor RDDs.

### Physical Model



The scheduler usually outputs a computation stage for each RDD in the graph. However, when an RDD can be computed from its parent without movement of data, multiple RDDs arecollapsed into a single stage. The collapsing of RDDs into one stage is called **pipelining**.In the example, the map operations are not moving any data and hence the RDDs have beenpipelined into stage 1. Since the reduceByKey does a shuffle, it is in the next stage.

When an action is encountered, the DAG is translated into a Physical plan to compute theRDDs needed for performing the action. The Spark scheduler submits a job to compute all thenecessary RDDs. Each job is made up of one or more stages and each stage is composed oftasks. Stages are processed in order and individual tasks are scheduled and executed on thecluster.

• A task is a unit of work within a stage corresponding to one RDD partition.

• A stage is a group of tasks which perform the same computation in parallel.

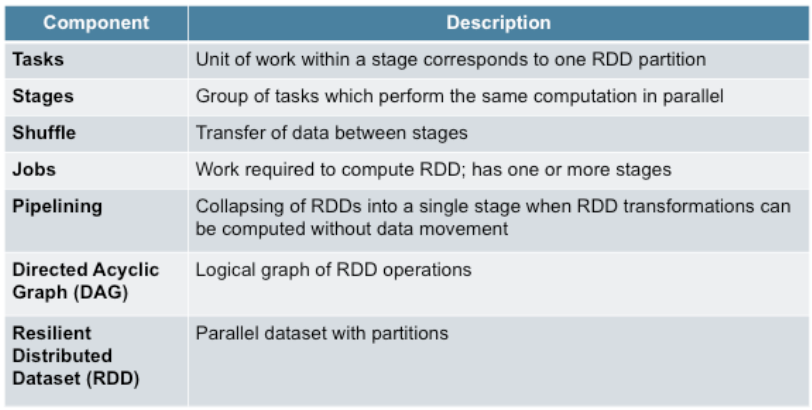
• A shuffle is the transferring of data between stages.

• A set of stages for a particular action is a job.

• When an RDD is computed from the parent without movement of data, the scheduler will pipeline or collapse RDDs into single stage

• DAG or Directed Acyclic Graph is the logical graph of RDD operations.

• RDD or Resilient Distributed Dataset is a parallel dataset with partitions.

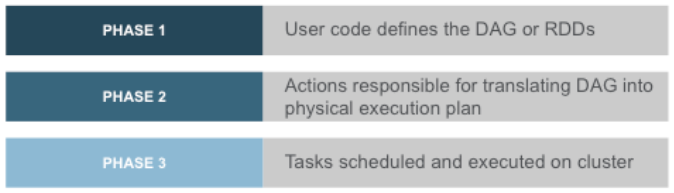


### Data Lineage Truncation

The situations in which the scheduler can truncate the lineage of an RDD graph are listed here.

1. Pipelining: When there is no movement of data from the parent RDD, the scheduler will pipeline the RDD graph collapsing multiple RDDs into a single stage.
2. When an RDD is persisted to cluster memory or disk, the Spark scheduler will truncate the lineage of the RDD graph. It will begin computations based on the persisted RDD.
3. Another situation when the Spark scheduler will truncate the lineage is if an RDD is already materialized due to an earlier shuffle, since shuffle outputs in Spark are written to disk. This optimization is built into Spark.

### Phases During Spark Execution



1. User code defines the DAG or RDDs

The user code defines RDDs and operations on RDDs. When you apply transformations on

RDDs, new RDDs are created that point to their parents resulting in a DAG.

2. Actions are responsible for translating the DAG into a physical execution plan

The RDD must be computed when an action is called on it. This results in computing theancestor(s). The scheduler will submit a job per action to compute all the required RDDs. Thisjob has one or more stages, which in turn is made up of tasks that operate in parallel onpartitions. A stage corresponds to one RDD unless the lineage is truncated due to pipelining.

3. Tasks are scheduled and executed on the cluster

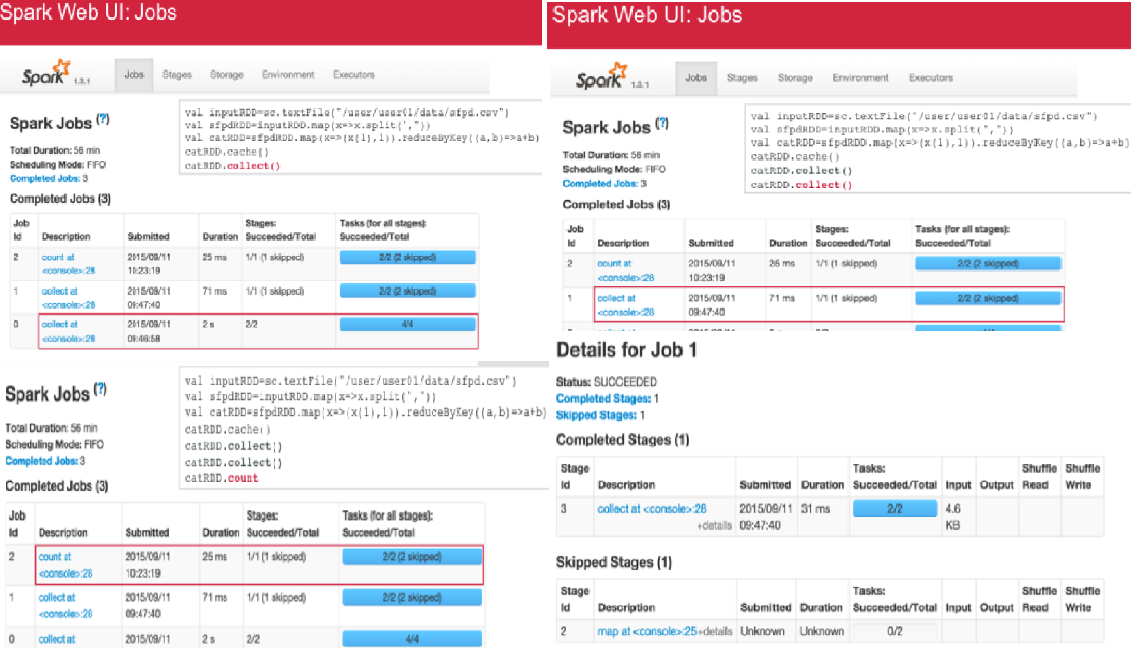
Stages are executed in order. The action is considered complete when the final stage in a job

completes.This sequence can occur many times when new RDDs are created.

### Spark MCS – UI

The Spark Web UI provides detailed information about the progress and performance for Sparkjobs. By default, this information is only available for the duration of the application. You canview the web UI after the event by setting spark.eventLog.enabled to true before starting theapplication.

In this example you see the jobs and stages based on the code sample shown.



* job 0 is the first job that was executed and corresponds to the first collect() action. Itconsists of 2 stages and each stage consists of 4 tasks.
* job 1 corresponds to the second collect() action. It consists of 1 stage which is made upof two tasks.
* job 2 corresponds to the count() action and is also consists of 1 stage containing twotasks.

Note that the first job took 2 seconds as compared to the second job which took 71 milliseconds. Because first collect computes all RDDs and then caches catRDD and therefore has twostages. The second **collect()** & the **count()** use the cached RDD and the scheduler truncates the lineage resulting in a skipped stage. This also results in job 1 (71 ms) being faster than job 0 (2 s).

Once you have identified the stage in which you are interested, you can click the link to drilldown to the Stage details page.Here we have summary metrics and aggregated metrics for completed tasks and metrics on alltasks.

On Storage (tab) page provides information about persisted RDDs. The RDD is persisted if youcalled persist() or cache() on the RDD followed by an action to compute on that RDD. This pagetells you which fraction of the RDD is cached and the quantity of data cached in various storagemedia.Scan this page to see if important datasets are fitting into memory.

On Environments (tab) page lists all the active properties of your Spark application environment.Use this page when you want to see which configuration flags are enabled.Note that only values specified throughspark-defaults.conf,SparkConf, or the command line willbe displayed here. For all other configuration properties, the default value is used.

On Executors (tab) page lists the active executors in the application. It also includes some metricsabout the processing and storage on each executor.Use this page to confirm that your application has the amount of resources that you wereexpecting.

You can use the Job details page and the stages tab to see which stages are running slow; to compare the metrics for a stage; look at each task.Look at the Executors tab to see if your application has the resources as expectedUse the Storage tab to see if the datasets are fitting into memory and what fraction is cached.

## Spark- PErformance Optimization

**Here are some ways to debug performance issues.**

* To detect shuffle problems, look at the Spark Web UI for any tasks that are significantly slow. A common source of performance problems in data-parallel systems that occurs when some small tasks take much longer than others is called **skew**. To see if skew is the problem, look at the Stages Details page and see if there are some tasks that are running significantly slower than others. Drill down to see if there is a small number of tasks that read or write more data than others.
* From the Stages Details page, you can also determine if tasks are running slow on certain nodes.
* From the Spark Web UI, you can also find those tasks that are spending too much time on reading, computing and/or writing.

In this case, look at the code to see if if you any expensive operations.

### Common Issues Leading To Slow Performance

* The level of parallelism
* The serialization format used during shuffle operations
* Managing memory to optimize your application

**1.Level Of Parallelism**

An RDD is divided into a set of partitions where each partition contains a subset of the data. Thescheduler will create a task for each partition. Each task requires a single core in the cluster.Spark by default will partition based on what it considers to be the best degree of parallelism.

* If there is too little parallelism – Spark might leave resources idle.
* If there is too much parallelism, overheads associated with each partition add up to becomesignificant.

Check Level of Partition

* You can do this through the Spark Web UI in the stages tab. Since a task in a stage maps to a single partition in the RDD, the total number of tasks will give you the number of partitions.
* You can also use rdd.partitions.size() to get the number of partitions in the RDD.

To tune the level of parallelism

* Specify the number of partitions, when you call operations that shuffle data, for example, reduceByKey.
* Redistribute the data in the RDD. This can be done by increasing or decreasing the number of partitions. You can use the repartition() method to specify the number of partitions or coalesce() to decrease the number of partitions.

**2. Serialization Format**

When a large amount of data is transferred over the network during shuffle operations, Sparkserializes objects into binary format. This can sometimes cause a bottleneck. By default Sparkuses the Java built-in serializer. However, it is often more efficient to use Kryo serialization.

**3. Managing Memory**

Memory can be used in different ways in Spark. Tuning Spark’s use of memory can helpoptimize your application.By default, Spark will use:

* 60% of space for RDD storage
* 20% for shuffle
* 20% for user programs

You can tune the memory usage by adjusting the memory regions used for RDD storage, shuffle and user programs.

1. Sometime it is better to use persist(MEMORY\_AND\_DISK). This will store the data on disk and load it into memory when needed. This cuts down expensive computations.
2. Sometime using MEMORY\_ONLY\_SER will cut down on garbage collection. Caching serialized objects may be slower than caching raw objects. However, it does decrease the time spent on garbage collection.

Note:

MEMORY\_ONLY\_SER: Store RDD as serialized Java objects (one byte array per partition). This is generally more space-efficient than deserialized objects, especially when using a fast serializer, but more CPU-intensive to read.

MEMORY\_AND\_DISK: Store RDD as deserialized Java objects in the JVM. If the RDD does not fit in memory, store the partitions that don't fit on disk, and read them from there when they're needed.

### Logging

* The Spark logging subsystem is based on log4j. The logging level or log output can be customized. An example of the log4j configuration properties is provided in the Spark conf directory which can be copied and suitably edited.
* The location of the Spark log files depends on the deployment mode.
* In the Spark Standalone mode, the log files are located in the work/ directory of the

Spark distribution on each worker.

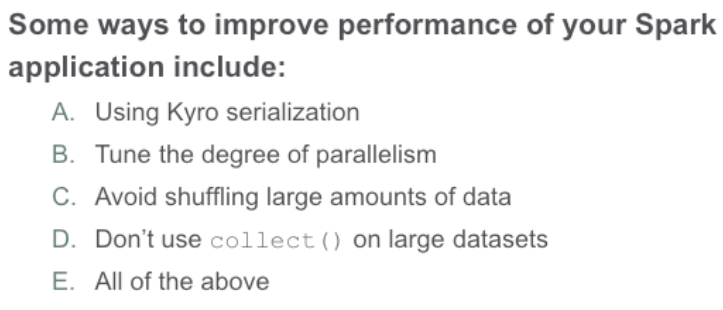
* In Mesos, the log files are in work/ directory of the Mesos slave and is accessible from

the Mesos master UI.

* To access the logs in YARN, use the YARN log collection tool.

### Best Practices

* If possible avoid having to shuffle large amounts of data. Use aggregateByKey when possible for aggregations. groupByKey on large dataset results in shuffling large amounts of data. If possible use reduceByKey. You can also use combineByKey or foldByKey.
* collect() action tries to copy every single element in the RDD to the single driver program. If you have a very large RDD, this can result in the driver crashing. The same problem occurs with countByKey, countByValue and collectAsMap.
* Filter out as much as you can to have smaller datasets
* If you have many idle tasks (~ 10k) , then coalesce
* If you are not using all the slots in the cluster, then repartition



## Spark- Code Snippet

### Data Load

**//load json file**

**val sfpdDF=sqlContext.jsonFile("/path to file/sfpd.json")**

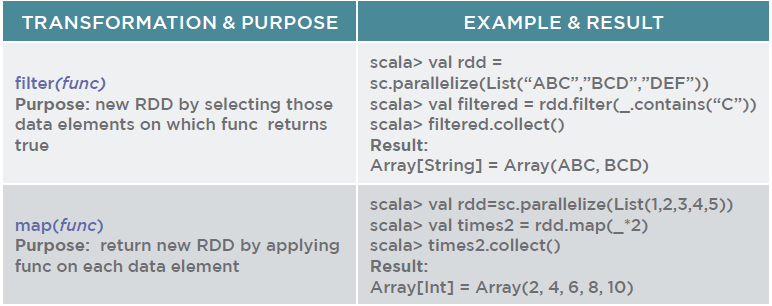
### filter

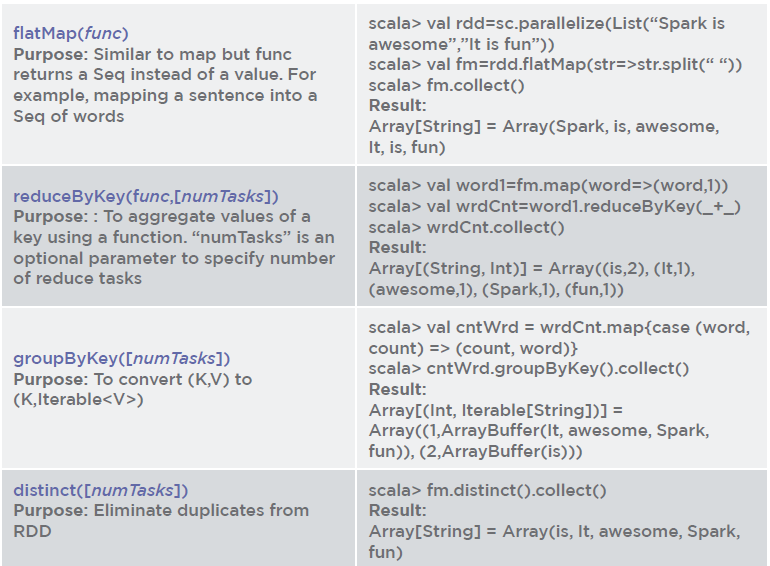
### MAP

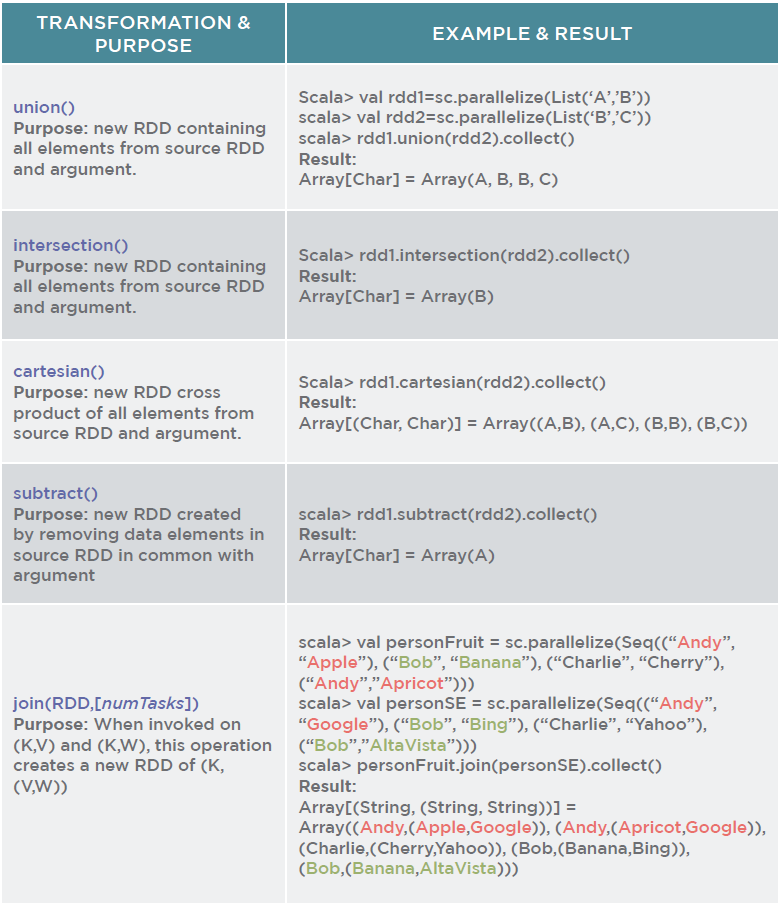
# Spark example for Transormation / Action

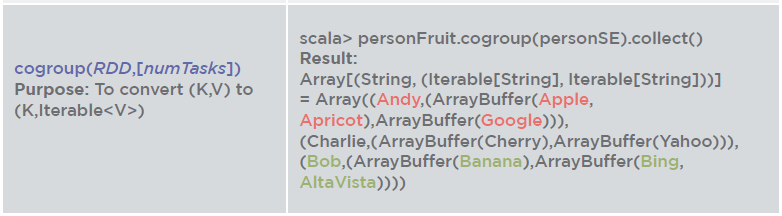
<https://dzone.com/storage/assets/3151-rd204-010d-spark_0.pdf>

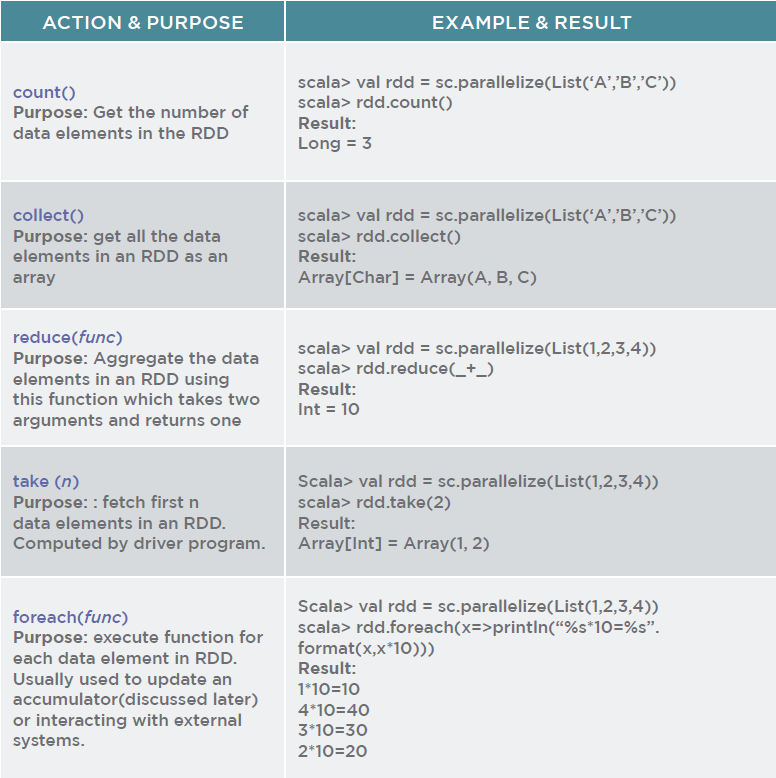
http://www.brunton-spall.co.uk/post/2011/12/02/map-map-and-flatmap-in-scala/

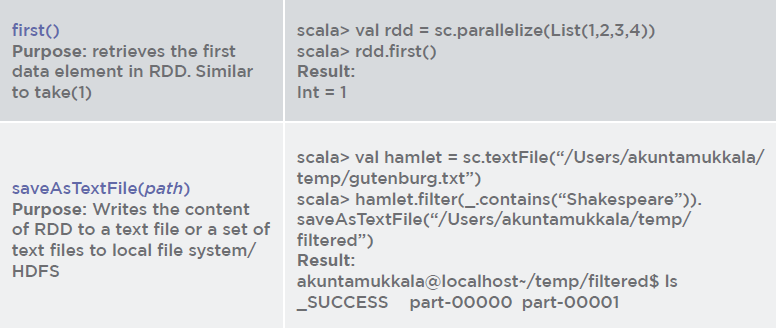


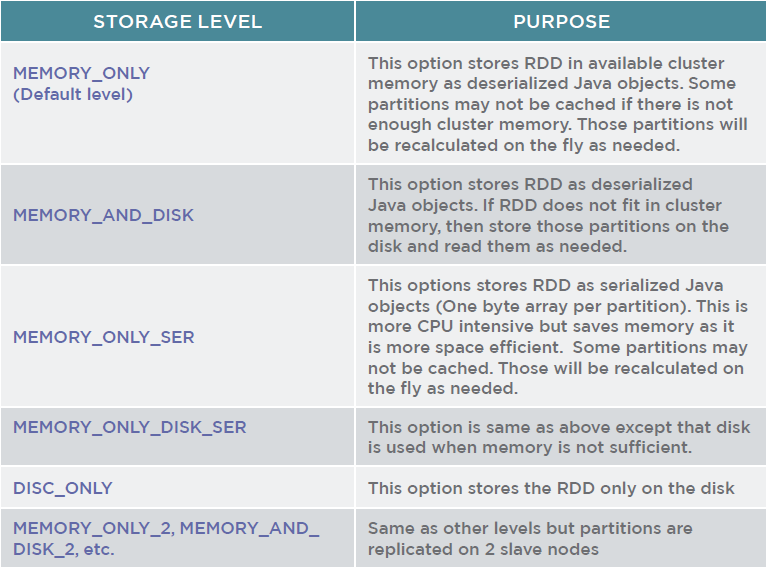












## Transformation Explained with Parallelize

### filter

*val rdd = sc.parallelize(List("ABC","BCD","DEF"))*

*val filtered = rdd.filter(\_.contains("C"))*

*filtered.collect()*

### map, flatMap,Map,mapValues

*val rdd = sc.parallelize(List(1,2,3,4,5))*

*val times2 = rdd.map(\_\*2)*

*times2.collect()*

In text analytics if you want to flatten out the entire book by words you can use flatmap.

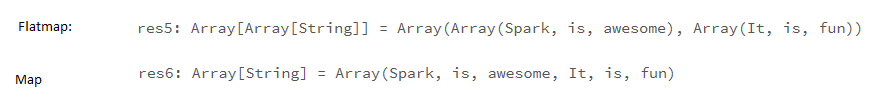
*val rdd = sc.parallelize(List("Spark is awesome","It is fun"))*

*val fm = rdd.flatMap(\_.split(" "))*

*val fm1 = rdd.map(\_.split(" "))*

*fm.collect()*

*fm1.collect()*

**

*val l = sc.parallelize(List(1,2,3,4,5))*

*def f(x: Int) = if(x > 2) Some(x) else None*

*val map1 = l.map(x => f(x))*

*val map2 = l.flatMap(x => f(x))*

*map1.collect*

*map2.collect*

*val t= (8,9)*

*t.\_1*

*t.\_2*

Map can be implemented a number of different ways, but regardless of how it is implemented it can be thought of as a sequence of Tuples, where a tuple is a pair of items, the key and the value.

*val m = Map(1 -> 2,2 -> 4,3 -> 6)*

*m.toList*

res27: List[(Int, Int)] = List((1,2), (2,4), (3,6))

Some times we probably don’t want to apply a function to the tuple, but to the value side of the tuple

*val m = Map(1 -> 2,2 -> 4,3 -> 6)*

*def f(x: Int) = if(x > 2) Some(x) else None*

*m.mapValues(v => v\*2)*

*m.mapValues(v => f(v))*

res68: scala.collection.immutable.Map[Int,Option[Int]] = Map(1 -> None, 2 -> Some(4), 3 -> Some(6))

But in my case I wanted to do something more like flat map in this case, I want a map to come out that misses out the key 1 because it’s value is None. flatMap doesn’t work on maps like mapValues, it get’s passed the tuple and if it returns a List single items you’ll get a list back, but if you return a tuple you’ll get a Map back.

*val m = Map(1 -> 22,2 -> 24,3 -> 26)*

*m.flatMap(e => List(e.\_2))*

res51: scala.collection.immutable.Iterable[Int] = List(22, 24, 26)

*m.flatMap(e => List(e))*# Returns Map

res50: scala.collection.immutable.Map[Int,Int] = Map(1 -> 22, 2 -> 24, 3 -> 26)

Ok so we are pretty close to using options with flatMap, we need to filter out our None’s, we can do returning a list with just e => f(e.\_2) and we’ll get the list of values without the None’s, but that isn’t really what I want. What I need to do is return an Option containing a tuple. So here’s our updated function:

*val m = Map(60 -> 22,2 -> 24,3 -> 26)*

*def h(k: Int, v: Int) = if (v > 22) Some( k -> v) else None*

*m.flatMap{case(k,v) => h(v,k)}*

res61: scala.collection.immutable.Map[Int,Int] = Map(22 -> 60)

*m.flatMap(e => h(e.\_1,e.\_2))*

res62: scala.collection.immutable.Map[Int,Int] = Map(2 -> 24, 3 -> 26)

Note that we switch to using curly braces, indicating a function block rather than parameters, and the function is a case statement. This means that the function block we pass to flatMap is a partialFunction that is only invoked for items that match the case statement, and in the case statement the unapply method on tuple is called to extract the contents of the tuple into the variables. This form of variable extraction is very common, and you’ll see it used a lot.

There is of course another way of writing that code that doesn’t use flatMap. Since what we are doing is removing all members of the map that don’t match a predicate, this is a use for the filter method:

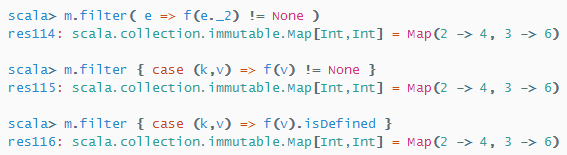
*val m = Map(1 -> 2,2 -> 4,3 -> 6)*

*def f(x: Int) = if(x > 2) Some(x) else None*

*m.filter(e => f(e.\_2) != None)*

*m.filter { case(k,v) => f(v) != None}*

*m.filter{ case (k,v) => f(v).isDefined}*



### reduceByKey

*val rdd = sc.parallelize(List("Spark is awesome","It is fun"))*

*val fm1 = rdd.flatMap(\_.split(" "))*

*val fm2 = rdd.map(\_.split(" "))*

*val word1 = fm1.map(word =>(word,1))*

*val wrdCnt1 = word1.reduceByKey((x,y) => x+y)*

*val word2 = fm2.map(word =>(word,1))*

*val wrdCnt2 = word2.reduceByKey((x,y) => x+y)*#throws error Cannot use map-side combining with array keys.

*wrdCnt1.collect()*

*# wrdCnt2.collect()*

*#Word2.collect()*

******

### groupByKey

*val rdd = sc.parallelize(List("Spark is awesome","It is fun"))*

*val fm1 = rdd.flatMap(\_.split(" "))*

*val fm2 = rdd.map(\_.split(" "))*

*val word1 = fm1.map(word =>(word,1))*

*val wrdCnt1 = word1.reduceByKey((x,y) => x+y)*

*val word2 = fm2.map(word =>(word,1))*

*val wrdCnt2 = word2.reduceByKey((x,y) => x+y)*# Error Cannot use map-side combining with array keys.

*val cntWrd1 = wrdCnt1.map{case(word,count) => (count,word)}*

*cntWrd1.reduceByKey((x,y) => x+y).collect()*

res69: Array[(Int, String)] = Array((1,ItawesomeSparkfun), (2,is))

*cntWrd1.groupByKey().collect()*

res70: Array[(Int, Iterable[String])] = Array((1,CompactBuffer(It, awesome, Spark, fun)), (2,CompactBuffer(is)))

### Set Operator

Paired RDD’s are not mandatory

*val rdd1 = sc.parallelize(List("A","B","C"))*

*val rdd2 = sc.parallelize(List("D","B","C"))*

*rdd1.union(rdd2).collect()*

*rdd1.intersection(rdd2).collect()*

*rdd1.subtract(rdd2).collect()*

*rdd1.cartesian(rdd2).collect()*

# Cartesian product produces Paied RDD output.

*val personFruit = sc.parallelize(Seq(("Andy","Apple"),("Bob","Banana"),("Charlie","Cherry"),("Andy","Apricot"),("kumar","Avacado")))*

*val rdd1 = sc.parallelize(List("A","B","C"))*

*val rdd2 = sc.parallelize(List("D","B","C"))*

*val rdd3 = rdd1.cartesian(rdd2)*

*rdd3.leftOuterJoin(personFruit).collect()*

res81: Array[(String, (String, Option[String]))] = Array((A,(D,None)), (A,(B,None)), (A,(C,None)), (B,(D,None)), (B,(B,None)), (B,(C,None)), (C,(D,None)), (C,(B,None)), (C,(C,None)))

#Set Operator for Paired RDD

*val personFruit = sc.parallelize(Seq(("Andy","Apple"),("Bob","Banana"),("Charlie","Cherry"),("Andy","Apricot"),("kumar","Avacado")))*

*val personSE = sc.parallelize(Seq(("Andy","Google"),("Bob","Bing"),("Charlie","Yahoo"),("Bob","Altavista"),("Andy","Apple")))*

*personFruit.intersection(personSE).collect()*

res78: Array[(String, String)] = Array((Andy,Apple))

### Join Operator

*val personFruit = sc.parallelize(Seq(("Andy","Apple"),("Bob","Banana"),("Charlie","Cherry"),("Andy","Apricot"),("kumar","Avacado")))*

*val personSE = sc.parallelize(Seq(("Andy","Google"),("Bob","Bing"),("Charlie","Yahoo"),("Bob","Altavista")))*

*personFruit.leftOuterJoin(personSE).collect()*

res82: Array[(String, (String, Option[String]))] = Array((Andy,(Apple,Some(Google))), (Andy,(Apricot,Some(Google))), (Charlie,(Cherry,Some(Yahoo))), (kumar,(Avacado,None)), (Bob,(Banana,Some(Bing))), (Bob,(Banana,Some(Altavista))))

*personSE.leftOuterJoin(personFruit).collect()*

*personFruit.join(personSE).collect()*

res83: Array[(String, (String, String))] = Array((Andy,(Apple,Google)), (Andy,(Apricot,Google)), (Charlie,(Cherry,Yahoo)), (Bob,(Banana,Bing)), (Bob,(Banana,Altavista)))

# Non Paired RDDS

*rdd1.leftOuterJoin(rdd2).collect()*#Error value leftOuterJoin is not a member of org.apache.spark.rdd.RDD[String] rdd1.leftOuterJoin(rdd2).collect()

### CoGroup Operator

For a key(Andy), values in rdd1 are grouped together(Apple, Apricot) and values of rdd2 are grouped together(Google), hence for a any Key you will have only one entry in “cogroup” however for the “join” you will have multiple entries for the each combinations of a key value in rdd1 and rdd2

*val personFruit = sc.parallelize(Seq(("Andy","Apple"),("Bob","Banana"),("Charlie","Cherry"),("Andy","Apricot"),("kumar","Avacado")))*

*val personSE = sc.parallelize(Seq(("Andy","Google"),("Bob","Bing"),("Charlie","Yahoo"),("Bob","Altavista")))*

*personFruit.cogroup(personSE).collect()*

res86: Array[(String, (Iterable[String], Iterable[String]))] = Array((Andy,(CompactBuffer(Apple, Apricot),CompactBuffer(Google))), (Charlie,(CompactBuffer(Cherry),CompactBuffer(Yahoo))), (kumar,(CompactBuffer(Avacado),CompactBuffer())), (Bob,(CompactBuffer(Banana),CompactBuffer(Bing, Altavista))))

## Spark SQL

Spark SQL provides a convenient way to run interactive queries over large data sets using Spark Engine, using a special type of RDD called SchemaRDD. SchemaRDDs can be created from existing RDDs or other external data formats such as Parquet files, JSON data or by running HQL on Hive. SchemaRDD is similar to a table in RDBMS. Once data are in SchemaRDD, the Spark engine will unify it with batch and streaming use cases. Spark SQL provides two types of contexts: SQLContext & HiveContext that extend SparkContext functionality.

* SQLContext provides access to a simple SQL parser whereas
* HiveContext provides access to HiveQL parser. HiveContext enables enterprises to leverage their existing Hive infrastructure.

Let’s see a simple example using SQLContext.Say we have the following ‘|’ delimited file containing customer data:

John Smith|38|M|201 East Heading Way #2203,Irving, TX,75063

Liana Dole|22|F|1023 West Feeder Rd, Plano,TX,75093

Craig Wolf|34|M|75942 Border Trail,Fort Worth,TX,75108

John Ledger|28|M|203 Galaxy Way,Paris, TX,75461

Joe Graham|40|M|5023 Silicon Rd,London,TX,76854

Define Scala case class to represent each row:

*case class Customer(name:String,age:Int,gender:String,address:String)*

The following code snippet shows how to create SQLContext using SparkContext, read the input file, convert each line into a record in SchemaRDD and then query in simple SQL to find male consumers under the age of 30:

*val sparkConf = new SparkConf().setAppName(“Customers”)*

*val sc = new SparkContext(sparkConf)*

*val sqlContext = new SQLContext(sc)*

*val r = sc.textFile(“/Users/akuntamukkala/temp/customers.txt”)*

*val records = r.map(\_.split(‘|’))*

*val c = records.map(r=>Customer(r(0),r(1).trim.toInt,r(2),r(3)))*

*c.registerAsTable(“customers”)*

*sqlContext.sql(“select \* from customers where gender=’M’ and age < 30”).collect().foreach(println)*

Result:

[John Ledger,28,M,203 Galaxy Way,Paris, TX,75461]

## Spark Streaming

Spark Streaming provides a scalable, fault tolerant, efficient way of processing streaming data using Spark’s simple programming model. It converts streaming data into “micro” batches, which enable Spark’s batch programming model to be applied in Streaming use cases. This unified programming model makes it easy to combine batch and interactive data processing with streaming.

The core abstraction in Spark Streaming is Discretized Stream (DStream). DStream is a sequence of RDDs. Each RDD contains data received in a configurable interval of time. Figure 12 shows how Spark Streaming creates a DStream by converting incoming data into a sequence of RDDs. Each RDD contains streaming data received every 2 seconds as defined by interval length. This can be as small as ½ second, so latency for processing time can be under 1 second.

Spark Streaming also provides sophisticated window operators, which help with running efficient computation on a collection of RDDs in a rolling window of time. DStream exposes an API, which contains operators (transformations and output operators) that are applied on constituent RDDs.

Let’s try and understand this using a simple example given in Spark Streaming download. Say, you want to find the trending hash tags in your Twitter stream. Please refer to the following example to find the complete code snippet:

spark-1.0.1/examples/src/main/scala/org/apache/spark/examples/streaming/TwitterPopularTags.scala

*val sparkConf = new SparkConf().setAppName(“TwitterPopularTags”)*

*val ssc = new StreamingContext(sparkConf, Seconds(2))*

*val stream = TwitterUtils.createStream(ssc, None, filters)*

The above snippet is setting up Spark Streaming Context. Spark Streaming will create an RDD in DStream containing Tweets retrieved every two seconds.

*val hashTags = stream.flatMap(status => status.getText.split(“ “).filter(\_.startsWith(“#”)))*

The above snippet converts the Tweets into a sequence of words, then filters only those beginning with a #.

*val topCounts60 = hashTags.map((\_, 1)).reduceByKeyAndWindow(\_ + \_, Seconds(60)).map{case (topic, count) => (count, topic)}. transform(\_.sortByKey(false))*

The above snippet shows how to calculate a rolling aggregate of the number of times a hashtag was mentioned in a window of 60 seconds.

*topCounts60.foreachRDD(rdd => {*

*val topList = rdd.take(10)*

*println(“\nPopular topics in last 60 seconds (%s total):”.format(rdd.count()))*

*topList.foreach{case (count, tag) => println(“%s (%s tweets)”.format(tag, count))}*

*})*

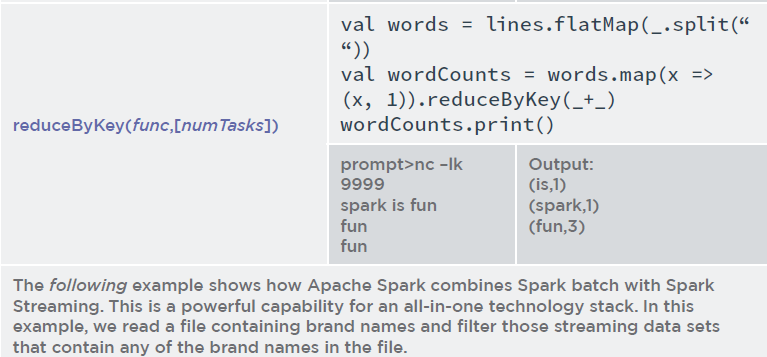
The above snippet shows how to extract the top ten trending Tweets and then print them out.

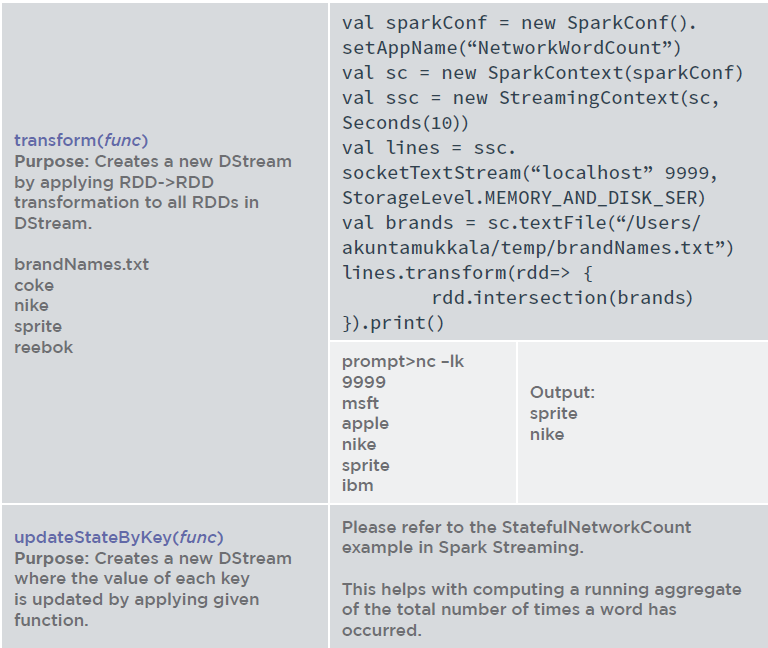
*ssc.start()*

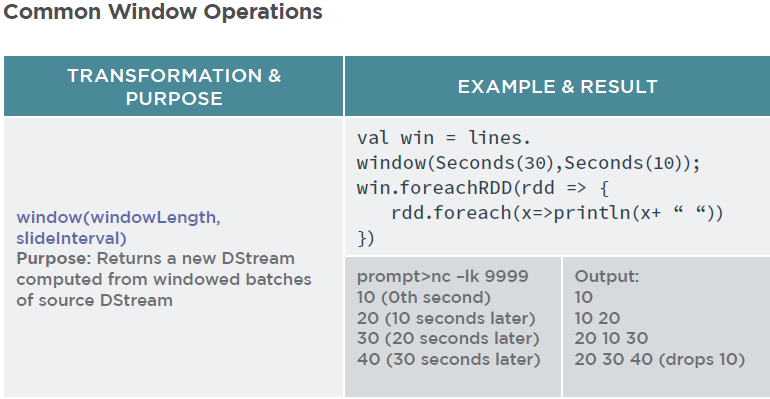
The above snippet instructs the Spark Streaming Context to start retrieving Tweets. Let’s look at a few popular operations. Assume that we are reading streaming text from a socket:

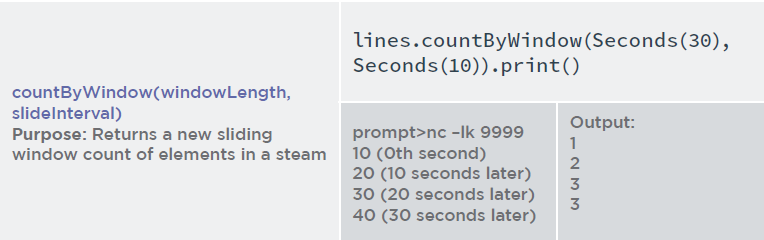
*val lines = ssc.socketTextStream(“localhost”, 9999, StorageLevel.MEMORY\_AND\_DISK\_SER)*











## Accumulator and Broadcast Variable

### Accumulator

Spark provides a very handy way to avoid mutable counters and counter synchronization issues by providing accumulators. The accumulators are initialized on a Spark context with a default value. These accumulators are available on Slave nodes, but Slave nodes can’t read them. Their only purpose is to fetch atomic updates and forward them to Master. Master is the only one that can read and compute the aggregate of all updates. For example, say we want to find the number of statements in a log file of log level ‘error’…

*akuntamukkala@localhost~/temp$ cat output.log*

*error*

*warning*

*info*

*trace*

*error*

*info*

*info*

*scala> val nErrors=sc.accumulator(0.0)*

*scala> val logs = sc.textFile(“/Users/akuntamukkala/temp/output.*

*log”)*

*scala> logs.filter(\_.contains(“error”)).foreach(x=>nErrors+=1)*

*scala> nErrors.value*

*Result: Int = 2*

### BroadCast

It is common to perform join operations on RDDs to consolidate data by a certain key. In such cases, it is quite possible to have large datasets sent around to slave nodes that host the partitions to be joined. This presents a huge performance bottleneck, as network I/O is 100 times slower than RAM access. In order to mitigate this issue, Spark provides broadcast variables, which, as the name suggests, are broadcasted to slave nodes. The RDD operations on the nodes can quickly access the broadcast variable value. For example, say we want to calculate the shipping cost of all line items in a file. We have a static look-up table that specifies cost per shipping type. This look-up table can be a broadcast variable.

*akuntamukkala@localhost~/temp$ cat packagesToShip.txt*

*ground*

*express*

*media*

*priority*

*priority*

*ground*

*express*

*media*

*scala> val map = sc.parallelize(Seq((“ground”,1),(“med”,2),*

*(“priority”,5),(“express”,10))).collect().toMap*

map: scala.collection.immutable.Map[String,Int] = Map(ground ->

1, media -> 2, priority -> 5, express -> 10)

*scala> val bcMailRates = sc.broadcast(map)*

In the above command, we create a broadcast variable, a map containing

cost by class of service.

*scala> val pts = sc.textFile(“/Users/akuntamukkala/temp/*

*packagesToShip.txt”)*

*scala> pts.map(shipType=>(shipType,1)).reduceByKey(\_+\_).map{case(shipType,nPackages)=>(shipType,nPackages\*bcMailRates.value(shipType))}.collect()*

In the above command we calculate shipping cost by looking up mailing rates from broadcast variable.

Array[(String, Int)] = Array((priority,10), (express,20), (media,4), (ground,2))

*scala> val shippingCost=sc.accumulator(0.0)*

*scala> pts.map(x=>(x,1)).reduceByKey(\_+\_).map{case(x,y)=>(x,y\*bcMailRates.value(x))}.foreach(v=>shippingCost+=v.\_2)*

*scala> shippingCost.value*

Result: Double = 36.0

Spark uses the Least Recently Used (LRU) algorithm to remove old, unused, cached RDDs to reclaim memory. It also provides a convenient unpersist() operation to force removal of cached/persisted RDDs.

## Certificate Preparation

### Standalone Application

Standalone Application

import org.apache.spark.sparkContext

import org.apache.spark.sparkContext.\_

import org.apache.spark.sparkConf

object KumarApplication {

def main (args :Array[String]){

val conf = new sparkConf().setAppName("KumarApplication")

val sc = new sparkContext(conf)

}

}

### Libraries

// SQLContext entry point for working with structured data

val sqlContext = new org.apache.spark.sql.SQLContext(sc)

// this is used to implicitly convert an RDD to a DataFrame.

import sqlContext.implicits.\_

// Import Spark SQL data types and Row.

import org.apache.spark.sql.\_

import org.apache.spark.util.StatCounter

import org.apache.spark.SparkConf

import org.apache.spark.streaming.{Seconds, StreamingContext}

import StreamingContext.\_

val ssc = new StreamingContext(sc, Seconds(2))

val textDStream = ssc.textFileStream("/user/user01/stream")

### Spark Basics

# number partitons

rdd.partitions.size

# type of partition

rdd.partitioner

### udf

def getyear(s:String):String = {

val year = s.substring(s.lastIndexOf('/')+1)

year

}

//2. register the function as a udf

sqlContext.udf.register("getyear",getyear \_)

//3. count inc by year

val incyearSQL=sqlContext.sql("SELECT getyear(date), count(incidentnum) AS countbyyear FROM sfpd GROUP BY getyear(date) ORDER BY countbyyear DESC")

incyearSQL.collect.foreach(println)

### Spark Streaming

import org.apache.spark.streaming.{Seconds, StreamingContext}

import StreamingContext.\_

val ssc = new StreamingContext(sc, Seconds(2))

val lines = ssc.textFileStream("/user/user01/stream")

**ssc.start()**

**ssc.awaitTermination()**

### Spark Basics

//1. How do you see the first element of the inputRDD?

RDD.first()

//2.What do you use to see the first 5 elements of the RDD?

RDD.take(5)

#groupByKey and reduceByKey

rdd.reduceByKey((x,y) => x+y)

rdd.groupByKey()

val sqlContext = new org.apache.spark.sql.SQLContext(sc)

import sqlContext.implicits.\_

//Defining the Auctions case class

case class Auctions(aucid:String, bid:Float,bidtime:Float,bidder:String,bidrate:Int,openbid:Float, price:Float,itemtype:String,dtl:Int)

//Loading the data into RDD with split

val inputRDD =sc.textFile("/user/user01/data/auctiondata.csv").map(\_.split(","))

// Mapping the inputRDD to the case class

val auctionsRDD = inputRDD.map(a=>Auctions(a(0),a(1).toFloat,a(2).toFloat,a(3),a(4).toInt, (5).toFloat,a(6).toFloat,a(7),a(8).toInt))

// converting auctionsRDD to a DataFrame

val auctionsDF = auctionsRDD.toDF()

//Registering the auctionsDF as a temporary table with the same name

auctionsDF.registerTempTable("auctionsTable")

//8. Check the data in the DataFrame

auctionsDF.show

//9. TO see the schema of the DataFrame

auctionsDF.printSchema

//7. Return the count of all auctions with final price greater than 200

auctionsDF.filter(auctionsDF("price")>200).count()

#filter

df.filter(df("col1") > 200)

scala> auctionsDF.select("itemtype").groupBy("itemtype").count().show()

+--------+-----+

|itemtype|count|

+--------+-----+

| xbox| 2784|

| palm| 5917|

| cartier| 1953|

+--------+-----+

scala> auctionsDF.select("itemtype").groupBy("itemtype").show()

<console>:36: error: value show is not a member of org.apache.spark.sql.GroupedData

auctionsDF.select("itemtype").groupBy("itemtype").show()

scala> auctionsDF.select("itemtype").groupBy("aucid").count().show()

#unable to resolve aucid

xboxes.describe("price").show

//8. Getting DF just for xboxes

val xboxes = sqlContext.sql("SELECT aucid,itemtype,bid,price,openbid FROM auctionsTable WHERE itemtype='xbox'")

auctionsDF.printSchema // no error

auctionsTable.PrintSchema // error auctionsTable not found

/1. Which five districts have the highest incidents?

val top5Dists = sfpdRDD.map(incident=>(incident(PdDistrict),1)).reduceByKey((x,y)=>x+y).map(x=>(x.\_2,x.\_1)).**sortByKey(false)**.take(5)

//4. What is the count of incidents by district?

val num\_inc\_dist = sfpdRDD.map(incident=>(incident(PdDistrict),1)).**countByKey()**

**//1. Top 5 Districts**

**val incByDist = sfpdDF.groupBy("pddistrict").count.sort($"count".desc).show(5)**

**val topByDistSQL = sqlContext.sql("SELECT pddistrict, count(incidentnum) AS inccount FROM sfpd GROUP BY pddistrict ORDER BY inccount DESC LIMIT 5")**

**//2. What are the top ten resolutions?**

**val top10Res = sfpdDF.groupBy("resolution").count.sort($"count".desc)**

**top10Res.show(10)**

**val top10ResSQL = sqlContext.sql("SELECT resolution, count(incidentnum) AS inccount FROM sfpd GROUP BY resolution ORDER BY inccount DESC LIMIT 10")**

**//3. Top 3 categories**

**val top3Cat = sfpdDF.groupBy("category").count.sort($"count".desc).show(3)**

**val top3CatSQL=sqlContext.sql("SELECT category, count(incidentnum) AS inccount FROM sfpd GROUP BY category ORDER BY inccount DESC LIMIT 3")**

**//4. Save the top 10 resolutions to a JSON file.**

**top10ResSQL.toJSON.saveAsTextFile("/user/user01/output")**

* Spark connects to the cluster Manager, Cluster Manager allocates resources across application.
* pyhton and spark "jars" are parsed through sparkContext and then send to executors
* Once connected spark accquires executors.

### **mapPartitions(func)**

Consider **mapPartitions** a tool for performance optimization if you have the horsepower.  It won’t do much for you when running examples on your local machine compared to running across a cluster.  It’s the same as **map**, but works with Spark RDD partitions.  Remember the first D in RDD is “Distributed” – Resilient Distributed Datasets.  Or, put another way, you could say it is distributed over partitions.

val rdd1 = sc.parallelize(1 to 12, 3)

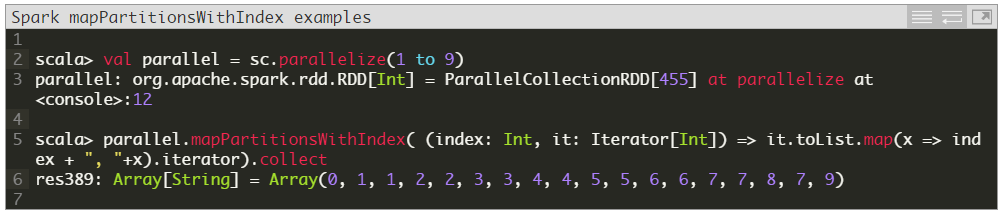
rdd1.mapPartitions(x => List(x.next).iterator).collect

val rdd1 = sc.parallelize(1 to 12)

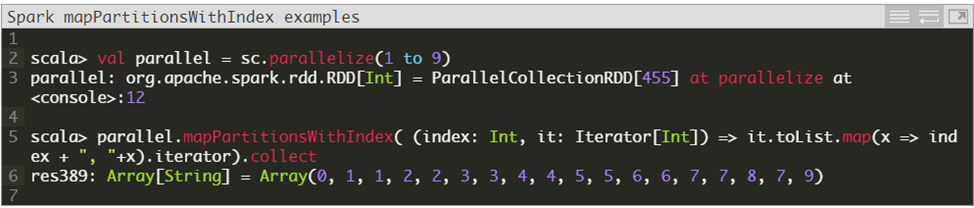
rdd1.mapPartitions(x => List(x.next).iterator).collect

### **mapPartitionsWithIndex(func)**

Similar to mapPartitions, but also provides a function with an Int value to indicate the index position of the partition.



When learning these APIs on an individual laptop or desktop, it might be helpful to show differences in capabilities and outputs.  For example, if we change the above example to use a parallelize’d list with 3 slices, our output changes significantly:



val rdd1 = sc.parallelize(1 to 12)

rdd1.mapPartitionsWithIndex((index: Int, it: Iterator[Int]) => it.toList.map(x => index + ", "+x).iterator).collect

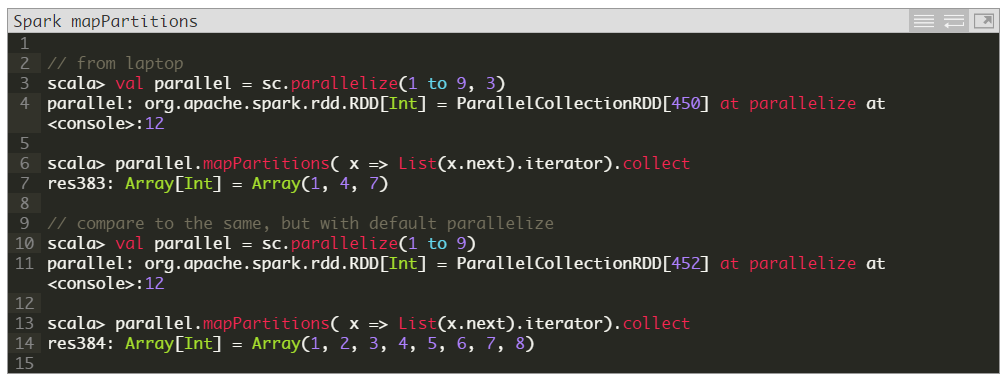
val rdd1 = sc.parallelize(1 to 12,3)

rdd1.mapPartitionsWithIndex((index: Int, it: Iterator[Int]) => it.toList.map(x => index + ", "+x).iterator).collect

**mapPartitionsWithIndex[U](f: (Int, Iterator[T]) ⇒ Iterator[U], preservesPartitioning: Boolean = false): RDD[U]**

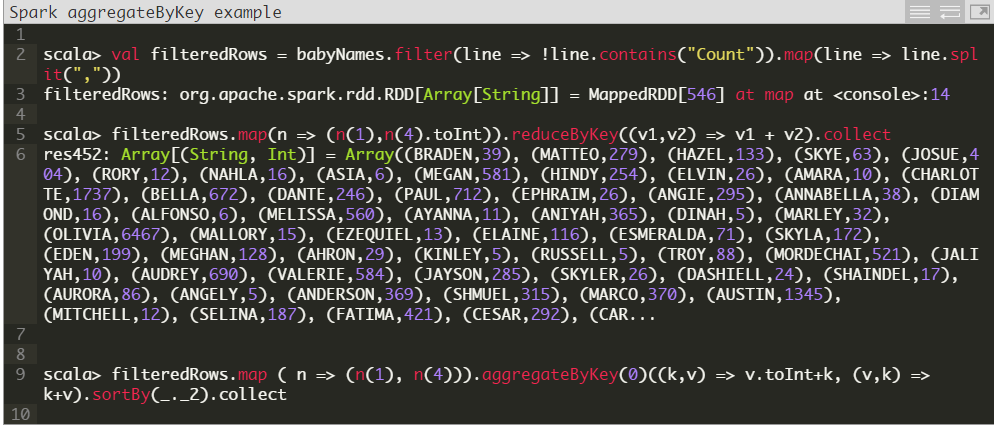
“Return a new RDD by applying a function to each partition of this RDD, while tracking the index of the original partition.

preservesPartitioning indicates whether the input function preserves the partitioner, which should be false unless this is a pair RDD and the input function doesn’t modify the keys.”



### aggregateByKey**(*zeroValue*)(*seqOp*, *combOp*, [*numTasks*])**

Ok, I admit, this one drives me a bit nuts.  Why wouldn’t we just use reduceByKey?  I don’t feel smart enough to know when to use aggregateByKey over reduceByKey.  For example, the same results may be produced:



val babyNamesCSV = sc.parallelize(List(("David", 6), ("Abby", 4), ("David", 5), ("Abby", 5)))

babyNamesCSV.reduceByKey((n,c) => n + c).collect

res1: Array[(String, Int)] = Array((Abby,9), (David,11))

babyNamesCSV.aggregateByKey(0)((k,v) => v.toInt+k, (v,k) => k+v).collect

babyNamesCSV.aggregateByKey(0)((accum, v) => accum + v, (v1, v2) => v1 + v2).collect

res1: Array[(String, Int)] = Array((Abby,9), (David,11))

=====

val pairs = sc.parallelize(Array(("a", 3), ("a", 1), ("b", 7), ("a", 5)))

val resReduce = pairs.reduceByKey(\_ + \_) //the same operation for everything

resReduce.collect

res3: Array[(String, Int)] = Array((b,7), (a,9))

//0 is initial value, \_+\_ inside partition, \_+\_ between partitions

val resAgg = pairs.aggregateByKey(0)(\_+\_,\_+\_) Now, imagine that you want the aggregation to be a Set of the values, that is a different

type that the values, that are integers (the sum of integers is also integers):

import scala.collection.mutable.HashSet

//the initial value is a void Set. Adding an element to a set is the first

//\_+\_ Join two sets is the \_++\_

val sets = pairs.aggregateByKey(new HashSet[Int])(\_+\_, \_++\_)

sets.collect

res5: Array[(String, scala.collection.mutable.HashSet[Int])] =Array((b,Set(7)), (a,Set(1, 5, 3)))resAgg.collect

res4: Array[(String, Int)] = Array((b,7), (a,9))

### CombineByKey

The combineByKey call is just such an optimization. When using combineByKey values are merged into one value at each partition then each partition value is merged into a single value. It’s worth noting that the type of the combined value does not have to match the type of the original value and often times it won’t be. The combineByKey function takes 3 functions as arguments:

1. A function that creates a combiner. In the aggregateByKey function the first argument was simply an initial zero value. In combineByKey we provide a function that will accept our current value as a parameter and return our new value that will be merged with addtional values.
2. The second function is a merging function that takes a value and merges/combines it into the previously collecte value(s).
3. The third function combines the merged values together. Basically this function takes the new values produced at the partition level and combines them until we end up with one singular value.

For our example lets take a look at calculating an average score. Calculating an average is a litte trickier compared to doing a count for the simple fact that counting is [associative](https://en.wikipedia.org/wiki/Associative_property) and [commutative](https://en.wikipedia.org/wiki/Commutative_property), we just sum all values for each partiton and sum the partition values. But with averages, it’s not that simple, an average of averages is not the same as taking an average across all numbers. But we can collect the total number scores and total score per partition then divide the total overall score by the number of scores. Here’s our example:

//type alias for tuples, increases readablity

type ScoreCollector = (Int, Double)

type PersonScores = (String, (Int, Double))

val initialScores = Array(("Fred", 88.0), ("Fred", 95.0), ("Fred", 91.0), ("Wilma", 93.0), ("Wilma", 95.0), ("Wilma", 98.0))

val wilmaAndFredScores = sc.parallelize(initialScores).cache()

val createScoreCombiner = (score: Double) => (1, score)

val scoreCombiner = (collector: ScoreCollector, score: Double) => {

val (numberScores, totalScore) = collector

(numberScores + 1, totalScore + score)

}

val scoreMerger = (collector1: ScoreCollector, collector2: ScoreCollector) => {

val (numScores1, totalScore1) = collector1

val (numScores2, totalScore2) = collector2

(numScores1 + numScores2, totalScore1 + totalScore2)

}

val scores = wilmaAndFredScores.combineByKey(createScoreCombiner, scoreCombiner, scoreMerger)

val averagingFunction = (personScore: PersonScores) => {

val (name, (numberScores, totalScore)) = personScore

(name, totalScore / numberScores)

}

val averageScores = scores.collectAsMap().map(averagingFunction)

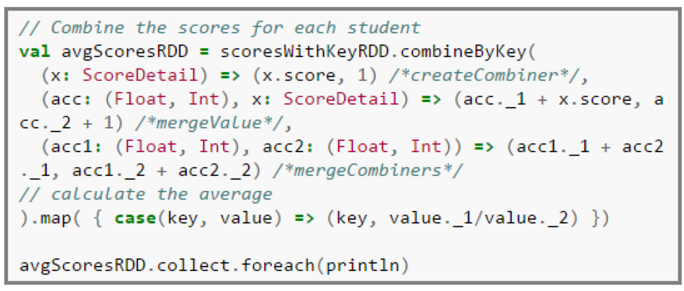
println("Average Scores using CombingByKey")

averageScores.foreach((ps) => {

val(name,average) = ps

println(name+ "'s average score : " + average)

})



Let’s describe what’s going on in the example above:

1. The createScoreCombiner takes a double value and returns a tuple of (Int, Double)
2. The scoreCombiner function takes a ScoreCollector which is a [type alias](http://www.scala-lang.org/files/archive/spec/2.11/04-basic-declarations-and-definitions.html#type-declarations-and-type-aliases) for a tuple of (Int,Double). We alias the values of the tuple to numberScores and totalScore (sacraficing a one-liner for readablility). We increment the number of scores by one and add the current score to the total scores received so far.
3. The scoreMerger function takes two ScoreCollectors adds the total number of scores and the total scores together returned in a new tuple.
4. We then call the combineByKey function passing our previously defined functions.
5. We take the resulting RDD, scores, and call the collectAsMap function to get our results in the form of (name,(numberScores,totalScore)).
6. To get our final result we call the map function on the scores RDD passing in the averagingFunction which simply calculates the average score and returns a tuple of (name,averageScore)

#### Final Results

After running our spark job, the results look like this:

# Spark Microsoft

Spark 2.1 X is used as a part of Microsoft Azure, Analytics HD Insight for Microsoft Certification in Spark.

## Data Exploration

Microsoft Spark Insight

Lab1: Data exploration

## Exploring Data with DataFrames and Spark SQL

##In this exercise, you will explore data using the Spark DataFrames API and Spark SQL.

## Load Data Using an Explicit Schema

##To explore data, you must load it into a programmatic data object such as a DataFrame. If the ##structure of the data is known ahead of time, you can explicitly specify the schema for the DataFrame.

In this exercise, you will work with data that records details of flights.

###########

*import org.apache.spark.sql.Encoders*

*case class flight(DayofMonth:Int, DayOfWeek:Int, Carrier:String, OriginAirportID:Int, DestAirportID:Int, DepDelay:Int, ArrDelay:Int)*

*val flightSchema = Encoders.product[flight].schema*

*val flights = spark.read.schema(flightSchema).option("header", "true").csv("wasb:///data/raw-flight-data.csv")*

*flights.show()*

+----------+---------+-------+---------------+-------------+--------+--------+

|DayofMonth|DayOfWeek|Carrier|OriginAirportID|DestAirportID|DepDelay|ArrDelay|

+----------+---------+-------+---------------+-------------+--------+--------+

| 19| 5| DL| 11433| 13303| -3| 1|

| 19| 5| DL| 14869| 12478| 0| -8|

| 19| 5| DL| 14057| 14869| -4| -15|

| 19| 5| DL| 15016| 11433| 28| 24|

| 19| 5| DL| 11193| 12892| -6| -11|

| 19| 5| DL| 10397| 15016| -1| -19|

## Infer a Data Schema

## If the structure of the data source is unknown, you can have Spark auomatically infer the schema.

## In this case, you will load data about airports without knowing the schema.

*val airports = spark.read.option("inferSchema","true").option("header","true").csv("wasb:///data/airports.csv")*

*airports.show()*

airport\_id| city|state| name|

+----------+-----------+-----+--------------------+

| 10165|Adak Island| AK| Adak|

| 10299| Anchorage| AK|Ted Stevens Ancho...|

| 10304| Aniak| AK| Aniak Airport|

| 10754| Barrow| AK|Wiley Post/Will R...|

| 10551| Bethel| AK| Bethel Airport|

| 10926| Cordova| AK|Merle K Mudhole S...

## Use DataFrame Methods

## Spark DataFrames provide functions that you can use to extract and manipulate data. For example, ## you can use the \*\*select\*\* function to return a new DataFrame containing columns selected from an ## existing DataFrame.

*val cities = airports.select("city", "name")*

*cities.show()*

*val flightsByOrigin = flights.join(airports, $"OriginAirportID" === $"airport\_id").groupBy("city").count()*

*flightsByOrigin.show()*

+-----------------+-----+

| city|count|

+-----------------+-----+

| Phoenix| 403|

| Omaha| 539|

| Raleigh/Durham| 834|

| Anchorage| 40|

*flights.count()*

res14: Long = 50515

*flights.describe().show()*

+-------+------------------+------------------+-----------------+------------------+-------

|summary| DayofMonth| DayOfWeek| OriginAirportID| DestAirportID|

+-------+------------------+------------------+-----------------+------------------+-------

| count| 50515| 50515| 50515| 50515|

| mean|19.067762050875977| 3.909848559833713|12317.80336533703|12321.356824705534|11.6522

| stddev| 8.01198651960034|2.0553928360789504|1466.944371084807|1466.2639555532598|41.8210

| min| 1| 1| 10140| 10140|

| max| 30| 7| 15376| 15376|

## Determine the Presence of Duplicates

## The data you have to work with won't always be perfect - often you'll want to \*clean\* the data; for ## example to detect and remove duplicates that might affect your model. You can use the ## ## \*\*dropDuplicates\*\* function to create a new DataFrame with the duplicates removed, enabling you to ## determine how many rows are duplicates of other rows.

*flights.count() - flights.dropDuplicates().count()*

res18: Long = 356

## Identify Missing Values

As well as determing if duplicates exist in your data, you should detect missing values, and either remove rows containing missing data or replace the missing values with a suitable relacement. The \*\*na.drop\*\* function creates a DataFrame with any rows containing missing data removed - you can specify a subset of columns, and whether the row should be removed in \*any\* or \*all\* values are missing. You can then use this new DataFrame to determine how many rows contain missing values.

flights.count() - flights.dropDuplicates().na.drop("any", Array("ArrDelay", "DepDelay")).count()

res19: Long = 1036

### Clean the Data

Now that you've identified that there are duplicates and missing values, you can clean the data by removing the duplicates and replacing the missing values. The \*\*na.fill\*\* function replaces missing values with a specified replacement value. In this case, you'll remove all duplicate rows and replace missing \*\*ArrDelay\*\* and \*\*DepDelay\*\* values with \*\*0\*\*.

*val data=flights.dropDuplicates().na.fill(0, Array("ArrDelay", "DepDelay"))*

*data.count()*

res20: Long = 50159

### Check Summary Statistics

After cleaning the data, you should re-check the statistics - removing rows and changing values may affect the distribution of the data, which in turn could affect any predictive models you might create.

*data.describe().show()*

### Explore Relationships in the Data

Predictive modeling is largely based on statistical relationships between fields in the data. To design a good model, you need to understand how the data points relate to one another and identify any apparent correlation. The \*\*stat.corr\*\* function calculates a correlation value between -1 and 1, indicating the strength of correlation between two fields. A strong positive correlation (near 1) indicates that high values for one column are often found with high values for the other, which a string negative correlation (near -1) indicates that \*low\* values for one column are often found with \*high\* values for the other. A correlation near 0 indicates little apparent relationship between the fields.

*data.stat.corr("DepDelay", "ArrDelay")*

res23: Double = 0.9443232901549696

### Use Spark SQL

In addition to using the DataFrame API directly to query data, you can persist DataFrames as table and use Spark SQL to query them using the SQL language. SQL is often more intuitive to use when querying tabular data structures.

*data.createOrReplaceTempView("flightData")*

*spark.sql("SELECT DayOfWeek, AVG(ArrDelay) AS AvgDelay FROM flightData GROUP BY DayOfWeek ORDER BY DayOfWeek").show()*

## Use the Inline SQL \*Magic\*

Jupyter Notebooks support \*magics\*, which enable you to include inline code and functionality. For example, the \*\*%%sql\*\* magic enables you to write SQL queries and visualize the results directly in the notebook. Run the following query, and view the table of results that is returned.

*%%sql*

*SELECT DepDelay, ArrDelay FROM flightData*

Change the \*\*Table\*\* visualization of results above to a \*\*Scatter\*\* visualization to see the relationship between the \*\*DepDelay\*\* and \*\*ArrDelay\*\* values more clearly (use the \*\*-\*\* function to plot the actual values) - visualizations like this make it easier to show relationships as apparent \*structure\* in the data. For example, the positive correlation between \*\*DepDelay\*\* and \*\*ArrDelay\*\* seems to be a linear relationsip, creaing a diagonal line of plotted points.

### Query Multiple Tables

You can create and query multiple temporary tables. Run the cells below to create a temporary table from the \*\*airports\*\* DataFrame, and then use an inline query to query it together with the flights data.

*airports.createOrReplaceTempView("airportData")*

*%%sql*

*SELECT a.name, AVG(f.ArrDelay) AS*

*FROM flightData AS f JOIN airportData AS a*

*ON f.DestAirportID = a.airport\_id*

*GROUP BY a.name*

*ORDER BY AVG(f.ArrDelay)*

##########################################################################################################################################################################

## Classification

Microsoft Spark Insight

Lab2.1: Classification

## Creating a Classification Model, In this exercise, you will implement a classification model that uses features of a flight to predict whether or not the flight will be delayed.

### Import Spark SQL and Spark ML Libraries First, import the libraries you will need:

*import org.apache.spark.sql.types.\_*

*import org.apache.spark.sql.functions.\_*

*import org.apache.spark.ml.classification.LogisticRegression*

*import org.apache.spark.ml.feature.VectorAssembler*

*val csv = spark.read.option("inferSchema","true").option("header", "true").csv("wasb:///data/flights.csv")*

*csv.show()*

### Prepare the Data

Most modeling begins with exhaustive exploration and preparation of the data. In this example, the data has been cleaned for you. You will simply select a subset of columns to use as \*features\* and create a Boolean \*label\* field named \*\*Late\*\* with the value \*\*1\*\* for flights that arrived 15 minutes or more after the scheduled arrival time, or \*\*0\*\* if the flight was early or on-time.

(Note that in a real scenario, you would perform additional tasks such as handling missing or duplicated data, scaling numeric columns, and using a process called \*feature engineering\* to create new features for your model).

*val data = csv.select($"DayofMonth", $"DayOfWeek", $"OriginAirportID", $"DestAirportID", $"DepDelay", ($"ArrDelay" > 15).cast("Int").alias("Late"))*

*data.show()*

### Split the Data

It is common practice when building supervised machine learning models to split the source data, using some of it to train the model and reserving some to test the trained model. In this exercise, you will use 70% of the data for training, and reserve 30% for testing.

*val splits = data.randomSplit(Array(0.7, 0.3))*

*val train = splits(0)*

*val test = splits(1)*

*val train\_rows = train.count()*

*val test\_rows = test.count()*

*println("Training Rows: " + train\_rows + " Testing Rows: " + test\_rows)*

### Prepare the Training Data

To train the classification model, you need a training data set that includes a vector of numeric features, and a label column. In this exercise, you will use the \*\*VectorAssembler\*\* class to transform the feature columns into a vector, and then rename the \*\*Late\*\* column to \*\*label\*\*.

*val assembler = new VectorAssembler().setInputCols(Array("DayofMonth", "DayOfWeek", "OriginAirportID", "DestAirportID", "DepDelay")).setOutputCol("features")*

*val training = assembler.transform(train).select($"features", $"Late".alias("label"))*

*training.show()*

### Train a Classification Model

Next, you need to train a classification model using the training data. To do this, create an instance of the classification algorithm you want to use and use its \*\*fit\*\* method to train a model based on the training DataFrame. In this exercise, you will use a \*Logistic Regression\* classification algorithm - though you can use the same technique for any of the classification algorithms supported in the spark.ml API.

*val lr = new LogisticRegression().setLabelCol("label").setFeaturesCol("features").setMaxIter(10).setRegParam(0.3)*

*val model = lr.fit(training)*

*println ("Model trained!")*

### Prepare the Testing Data

Now that you have a trained model, you can test it using the testing data you reserved previously. First, you need to prepare the testing data in the same way as you did the training data by transforming the feature columns into a vector. This time you'll rename the \*\*Late\*\* column to \*\*trueLabel\*\*.

*val testing = assembler.transform(test).select($"features", $"Late".alias("trueLabel"))*

*testing.show()*

### Test the Model

Now you're ready to use the \*\*transform\*\* method of the model to generate some predictions. You can use this approach to predict delay status for flights where the label is unknown; but in this case you are using the test data which includes a known true label value, so you can compare the predicted status to the actual status.

*val prediction = model.transform(testing)*

*val predicted = prediction.select("features", "prediction", "probability", "trueLabel")*

*predicted.show(100)*

Looking at the result, the \*\*prediction\*\* column contains the predicted value for the label, and the \*\*trueLabel\*\* column contains the actual known value from the testing data. It looks like there are a mix of correct and incorrect predictions - later in this course you'll learn how to measure the accuracy of a model.

## PipeLine

Microsoft Spark Insight

Lab2: Pipe Line

## Creating a Pipeline

In this exercise, you will implement a pipeline that includes multiple stages of \*transformers\* and \*estimators\* to prepare features and train a classification model. The resulting trained \*PipelineModel\* can then be used as a transformer to predict whether or not a flight will be late.

### Import Spark SQL and Spark ML Libraries First, import the libraries you will need:

*import org.apache.spark.sql.types.\_*

*import org.apache.spark.sql.functions.\_*

*import org.apache.spark.ml.Pipeline*

*import org.apache.spark.ml.feature.{VectorAssembler, StringIndexer, VectorIndexer, MinMaxScaler}*

*import org.apache.spark.ml.classification.DecisionTreeClassifier*

### Load Source Data

The data for this exercise is provided as a CSV file containing details of flights. The data includes specific characteristics (or \*features\*) for each flight, as well as a column indicating how many minutes late or early the flight arrived.You will load this data into a DataFrame and display it.

*val csv = spark.read.option("inferSchema","true").option("header", "true").csv("wasb:///data/flights.csv")*

*csv.show()*

### Prepare the Data

Most modeling begins with exhaustive exploration and preparation of the data. In this example, the data has been cleaned for you. You will simply select a subset of columns to use as \*features\* and create a Boolean \*label\* field named \*\*label\*\* with the value \*\*1\*\* for flights that arrived 15 minutes or more after the scheduled arrival time, or \*\*0\*\* if the flight was early or on-time.

*val data = csv.select($"DayofMonth", $"DayOfWeek", $"Carrier", $"OriginAirportID", $"DestAirportID", $"DepDelay", ($"ArrDelay" > 15).cast("Double").alias("label"))*

*data.show()*

### Split the Data

It is common practice when building supervised machine learning models to split the source data, using some of it to train the model and reserving some to test the trained model. In this exercise, you will use 70% of the data for training, and reserve 30% for testing. In the testing data, the \*\*label\*\* column is renamed to \*\*trueLabel\*\* so you can use it later to compare predicted labels with known actual values.

*val splits = data.randomSplit(Array(0.7, 0.3))*

*val train = splits(0)*

*val test = splits(1).withColumnRenamed("label", "trueLabel")*

*val train\_rows = train.count()*

*val test\_rows = test.count()*

*println("Training Rows: " + train\_rows + " Testing Rows: " + test\_rows)*

### Define the Pipeline

A predictive model often requires multiple stages of feature preparation. For example, it is common when using some algorithms to distingish between continuous features (which have a calculable numeric value) and categorical features (which are numeric representations of discrete categories). It is also common to \*normalize\* continuous numeric features to use a common scale (for example, by scaling all numbers to a proportinal decimal value between 0 and 1).

A pipeline consists of a a series of \*transformer\* and \*estimator\* stages that typically prepare a DataFrame for

modeling and then train a predictive model. In this case, you will create a pipeline with seven stages:

- A \*\*StringIndexer\*\* estimator that converts string values to indexes for categorical features

- A \*\*VectorAssembler\*\* that combines categorical features into a single vector

- A \*\*VectorIndexer\*\* that creates indexes for a vector of categorical features

- A \*\*VectorAssembler\*\* that creates a vector of continuous numeric features

- A \*\*MinMaxScaler\*\* that normalizes continuous numeric features

- A \*\*VectorAssembler\*\* that creates a vector of categorical and continuous features

- A \*\*DecisionTreeClassifier\*\* that trains a classification model.

*val strIdx = new StringIndexer().setInputCol("Carrier").setOutputCol("CarrierIdx")*

*val catVect = new VectorAssembler().setInputCols(Array("CarrierIdx", "DayofMonth", "DayOfWeek", "OriginAirportID", "DestAirportID")).setOutputCol("catFeatures")*

*val catIdx = new VectorIndexer().setInputCol(catVect.getOutputCol).setOutputCol("idxCatFeatures")*

*val numVect = new VectorAssembler().setInputCols(Array("DepDelay")).setOutputCol("numFeatures")*

*val minMax = new MinMaxScaler().setInputCol(numVect.getOutputCol).setOutputCol("normFeatures")*

*val featVect = new VectorAssembler().setInputCols(Array("idxCatFeatures", "normFeatures")).setOutputCol("features")*

*val dt = new DecisionTreeClassifier().setLabelCol("label").setFeaturesCol("features")*

*val pipeline = new Pipeline().setStages(Array(strIdx, catVect, catIdx, numVect, minMax, featVect, dt))*

### Run the Pipeline as an Estimator

The pipeline itself is an estimator, and so it has a \*\*fit\*\* method that you can call to run the pipeline on a specified DataFrame. In this case, you will run the pipeline on the training data to train a model.

*val model = pipeline.fit(train)*

*println("Pipeline complete!")*

### Test the Pipeline Model

The model produced by the pipeline is a transformer that will apply all of the stages in the pipeline to a specified DataFrame and apply the trained model to generate predictions. In this case, you will transform the \*\*test\*\* DataFrame using the pipeline to generate label predictions.

*val prediction = model.transform(test)*

*val predicted = prediction.select("features", "prediction", "label")*

*predicted.show(100, truncate=false*)

The resulting DataFrame is produced by applying all of the transformations in the pipline to the test data. The \*\*prediction\*\* column contains the predicted value for the label, and the \*\*trueLabel\*\* column contains the actual known value from the testing data.

## Classification Evaluation

## Evaluating a Classification Model

In this exercise, you will create a pipeline for a classification model, and then apply commonly used metrics to evaluate the resulting classifier.

### Prepare the Data

First, import the libraries you will need and prepare the training and test data:

// Import Spark SQL and Spark ML libraries

*import org.apache.spark.sql.types.\_*

*import org.apache.spark.sql.functions.\_*

*import org.apache.spark.ml.Pipeline*

*import org.apache.spark.ml.feature.VectorAssembler*

*import org.apache.spark.ml.classification.LogisticRegression*

// Load the source data

*val csv = spark.read.option("inferSchema","true").option("header", "true").csv("wasb:///data/flights.csv")*

// Select features and label

*val data = csv.select($"DayofMonth", $"DayOfWeek", $"OriginAirportID", $"DestAirportID", $"DepDelay", ($"ArrDelay" > 15).cast("Int").alias("label"))*

// Split the data

*val splits = data.randomSplit(Array(0.7, 0.3))*

*val train = splits(0)*

*val test = splits(1).withColumnRenamed("label", "trueLabel")*

### Test the Model

*Now you're ready to apply the model to the test data.*

*val prediction = model.transform(test)*

*val predicted = prediction.select("features", "prediction", "trueLabel")*

*predicted.show(100, truncate=false)*

*### Compute Confusion Matrix Metrics*

*Classifiers are typically evaluated by creating a \*confusion matrix\*, which indicates the number of:*

*- True Positives*

*- True Negatives*

*- False Positives*

*- False Negatives*

*From these core measures, other evaluation metrics such as \*precision\* and \*recall\* can be calculated.*

*val tp = predicted.filter("prediction == 1 AND truelabel == 1").count().toFloat*

*val fp = predicted.filter("prediction == 1 AND truelabel == 0").count().toFloat*

*val tn = predicted.filter("prediction == 0 AND truelabel == 0").count().toFloat*

*val fn = predicted.filter("prediction == 0 AND truelabel == 1").count().toFloat*

*val metrics = spark.createDataFrame(Seq(*

*("TP", tp),*

*("FP", fp),*

*("TN", tn),*

*("FN", fn),*

*("Precision", tp / (tp + fp)),*

*("Recall", tp / (tp + fn)))).toDF("metric", "value")*

*metrics.show()*

### View the Raw Prediction and Probability

The prediction is based on a raw prediction score that describes a labelled point in a logistic function. This raw prediction is then converted to a predicted label of 0 or 1 based on a probability vector that indicates the confidence for each possible label value (in this case, 0 and 1). The value with the highest confidence is selected as the prediction.

*prediction.select("rawPrediction", "probability", "prediction", "trueLabel").show(100, truncate=false)*

Note that the results include rows where the probability for 0 (the first value in the \*\*probability\*\* vector) is only slightly higher than the probability for 1 (the second value in the \*\*probability\*\* vector). The default \*discrimination threshold\* (the boundary that decides whether a probability is predicted as a 1 or a 0) is set to 0.5; so the prediction with the highest probability is always used, no matter how close to the threshold.

### Review the Area Under ROC

Another way to assess the performance of a classification model is to measure the area under a ROC curve for the model. the spark.ml library includes a \*\*BinaryClassificationEvaluator\*\* class that you can use to compute this. The ROC curve shows the True Positive and False Positive rates plotted for varying thresholds.

*import org.apache.spark.ml.evaluation.BinaryClassificationEvaluator*

*val evaluator = new BinaryClassificationEvaluator().setLabelCol("trueLabel").setRawPredictionCol("rawPrediction").setMetricName("areaUnderROC")*

*val auc = evaluator.evaluate(prediction)*

*println("AUC = " + (auc))*

### Change the Discrimination Threshold

The AUC score seems to indicate a reasonably good model, but the performance metrics seem to indicate that it predicts a high number of False Negative labels (i.e. it predicts 0 when the true label is 1), leading to a low Recall. You can affect the way a model performs by changing its parameters. For example, as noted previously, the default discrimination threshold is set to 0.5 - so if there are a lot of False Positives, you may want to consider raising this; or conversely, you may want to address a large number of False Negatives by lowering the threshold.

// Redefine the pipeline

*val lr2 = new LogisticRegression().setLabelCol("label").setFeaturesCol("features").setThreshold(0.35).setMaxIter(10).setRegParam(0.3)*

*val pipeline2 = new Pipeline().setStages(Array(assembler, lr2))*

// Retrain the model

*val model2 = pipeline2.fit(train)*

// Retest the model

*val newPrediction = model2.transform(test)*

*newPrediction.select("rawPrediction", "probability", "prediction", "trueLabel").show(100, truncate=false)*

Note that some of the \*\*rawPrediction\*\* and \*\*probability\*\* values that were previously predicted as 0 are now predicted as 1

// Recalculate confusion matrix

*val tp2 = newPrediction.filter("prediction == 1 AND truelabel == 1").count().toFloat*

*val fp2 = newPrediction.filter("prediction == 1 AND truelabel == 0").count().toFloat*

*val tn2 = newPrediction.filter("prediction == 0 AND truelabel == 0").count().toFloat*

*val fn2 = newPrediction.filter("prediction == 0 AND truelabel == 1").count().toFloat*

*val metrics2 = spark.createDataFrame(Seq(*

*("TP", tp2),*

*("FP", fp2),*

*("TN", tn2),*

*("FN", fn2),*

*("Precision", tp2 / (tp2 + fp2)),*

*("Recall", tp2 / (tp2 + fn2)))).toDF("metric", "value")*

*metrics2.show()*

### Tunning Model Parameters

## Tuning Model Parameters

In this exercise, you will optimise the parameters for a classification model.

### Prepare the DataFirst, import the libraries you will need and prepare the training and test data:

// Import Spark SQL and Spark ML libraries

*import org.apache.spark.sql.types.\_*

*import org.apache.spark.sql.functions.\_*

*import org.apache.spark.ml.Pipeline*

*import org.apache.spark.ml.feature.VectorAssembler*

*import org.apache.spark.ml.classification.LogisticRegression*

*import org.apache.spark.ml.evaluation.BinaryClassificationEvaluator*

*import org.apache.spark.ml.tuning.{ParamGridBuilder, TrainValidationSplit}*

// Load the source data

*val csv = spark.read.option("inferSchema","true").option("header", "true").csv("wasb:///example/flightsv1.csv")*

// Select features and label

*val data = csv.select($"DayofMonth", $"DayOfWeek", $"OriginAirportID", $"DestAirportID", $"DepDelay", ($"ArrDelay" > 15).cast("Int").alias("label"))*

// Split the data

*val splits = data.randomSplit(Array(0.7, 0.3))*

*val train = splits(0)*

*val test = splits(1).withColumnRenamed("label", "trueLabel")*

// Define the pipeline

*val assembler = new VectorAssembler().setInputCols(Array("DayofMonth", "DayOfWeek", "OriginAirportID", "DestAirportID", "DepDelay")).setOutputCol("features")*

*val lr = new LogisticRegression().setLabelCol("label").setFeaturesCol("features")*

*val pipeline = new Pipeline().setStages(Array(assembler, lr))*

### Tune Parameters

You can tune parameters to find the best model for your data. A simple way to do this is to use \*\*TrainValidationSplit\*\* to evaluate each combination of parameters defined in a \*\*ParameterGrid\*\* against a subset of the training data in order to find the best performing parameters.

*val paramGrid = new ParamGridBuilder().addGrid(lr.regParam, Array(0.3, 0.1, 0.01)).addGrid(lr.maxIter, Array(10, 5)).addGrid(lr.threshold, Array(0.35, 0.3)).build()*

*val tvs = new TrainValidationSplit().setEstimator(pipeline).setEvaluator(new BinaryClassificationEvaluator).setEstimatorParamMaps(paramGrid).setTrainRatio(0.8)*

*val model = tvs.fit(train)*

*val prediction = model.transform(test)*

*val predicted = prediction.select("features", "prediction", "probability", "trueLabel")*

*predicted.show(100)*

+--------------------+----------+--------------------+---------+

| features|prediction| probability|trueLabel|

+--------------------+----------+--------------------+---------+

|[1.0,1.0,10140.0,...| 0.0|[0.85225827636082...| 0|

|[1.0,1.0,10397.0,...| 0.0|[0.89724735541319...| 0|

|[1.0,1.0,10397.0,...| 1.0|[0.44691548182353...| 0|

|[1.0,1.0,10397.0,...| 0.0|[0.90340571607893...| 0|

|[1.0,1.0,10397.0,...| 0.0|[0.89731717100094...| 0|

|[1.0,1.0,10397.0,...| 0.0|[0.89148261659105...| 0|

|[1.0,1.0,10397.0,...| 0.0|[0.91320089084569...| 0|

### Compute Confusion Matrix Metrics

Now you can examine the confusion matrix metrics to judge the performance of the model.

val tp = predicted.filter("prediction == 1 AND truelabel == 1").count().toFloat

val fp = predicted.filter("prediction == 1 AND truelabel == 0").count().toFloat

val tn = predicted.filter("prediction == 0 AND truelabel == 0").count().toFloat

val fn = predicted.filter("prediction == 0 AND truelabel == 1").count().toFloat

val metrics = spark.createDataFrame(Seq(

("TP", tp),

("FP", fp),

("TN", tn),

("FN", fn),

("Precision", tp / (tp + fp)),

("Recall", tp / (tp + fn)))).toDF("metric", "value")

metrics.show()

+---------+---------+

| metric| value|

+---------+---------+

| TP| 2211.0|

| FP| 165.0|

| TN| 11796.0|

| FN| 919.0|

|Precision|0.9305556|

| Recall|0.7063898|

+---------+---------+

### Review the Area Under ROC

You can also assess the accuracy of the model by reviewing the area under ROC metric.

*val evaluator = new BinaryClassificationEvaluator().setLabelCol("trueLabel").setRawPredictionCol("prediction").setMetricName("areaUnderROC")*

*val aur = evaluator.evaluate(prediction)*

*println("AUR = " + (aur))*

AUR = 0.8462974715749509

## Regression Evaluation

## Evaluating a Regression Model

In this exercise, you will create a pipeline for a linear regression model, and then test and evaluate the model.

### Prepare the DataFirst, import the libraries you will need and prepare the training and test data:

// Import Spark SQL and Spark ML libraries

*import org.apache.spark.sql.types.\_*

*import org.apache.spark.sql.functions.\_*

*import org.apache.spark.ml.Pipeline*

*import org.apache.spark.ml.feature.VectorAssembler*

*import org.apache.spark.ml.regression.LinearRegression*

// Load the source data

*val csv = spark.read.option("inferSchema","true").option("header", "true").csv("wasb:///example/flightsv1.csv")*

// Select features and label

*val data = csv.select($"DayofMonth", $"DayOfWeek", $"OriginAirportID", $"DestAirportID", $"DepDelay", $"ArrDelay".alias("label"))*

// Split the data

*val splits = data.randomSplit(Array(0.7, 0.3))*

*val train = splits(0)*

*val test = splits(1).withColumnRenamed("label", "trueLabel")*

*val train\_rows = train.count()*

*val test\_rows = test.count()*

*println("Training Rows: " + train\_rows + " Testing Rows: " + test\_rows)*

// Define the pipeline

*val assembler = new VectorAssembler().setInputCols(Array("DayofMonth", "DayOfWeek", "OriginAirportID", "DestAirportID", "DepDelay")).setOutputCol("features")*

*val lr = new LinearRegression().setLabelCol("label").setFeaturesCol("features").setMaxIter(10).setRegParam(0.3)*

*val pipeline = new Pipeline().setStages(Array(assembler, lr))*

// Train the model

*val model = pipeline.fit(train)*

*val prediction = model.transform(test)*

*val predicted = prediction.select("features", "prediction", "trueLabel")*

*predicted.show()*

+--------------------+-------------------+---------+

| features| prediction|trueLabel|

+--------------------+-------------------+---------+

|[1.0,1.0,10140.0,...| 832.4344949702855| 812|

|[1.0,1.0,10397.0,...| 0.3644299829216007| 3|

|[1.0,1.0,10397.0,...| 32.227535937969265| 25|

|[1.0,1.0,10397.0,...| -5.664671790912316| -11|

|[1.0,1.0,10397.0,...| -6.668722840208762| 4|

|[1.0,1.0,10397.0,...| -4.759559025702929| -23|

### Examine the Predicted and Actual Values

You can plot the predicted values against the actual values to see how accurately the model has predicted. In a perfect model, the resulting scatter plot should form a perfect diagonal line with each predicted value being identical to the actual value - in practice, some variance is to be expected.

Run the cells below to create a temporary table from the \*\*predicted\*\* DataFrame and then retrieve the predicted and actual label values using SQL. You can then display the results as a scatter plot, specifying \*\*-\*\* as the function to show the unaggregated values.

*predicted.createOrReplaceTempView("regressionPredictions")*

*%%sql*

*SELECT trueLabel, prediction FROM regressionPredictions*

| **trueLabel** | **prediction** |
| --- | --- |
| 812 | 832.434495 |
| 3 | 0.364430 |
| 25 | 32.227536 |
| -11 | -5.664672 |

### Retrieve the Root Mean Square Error (RMSE)

There are a number of metrics used to measure the variance between predicted and actual values. Of these, the root mean square error (RMSE) is a commonly used value that is measured in the same units as the predicted and actual values - so in this case, the RMSE indicates the average number of minutes between predicted and actual flight delay values. You can use the \*\*RegressionEvaluator\*\* class to retrieve the RMSE.

*import org.apache.spark.ml.evaluation.RegressionEvaluator*

*val evaluator = new RegressionEvaluator().setLabelCol("trueLabel").setPredictionCol("prediction").setMetricName("rmse")*

*val rmse = evaluator.evaluate(prediction)*

*println("Root Mean Square Error (RMSE): " + (rmse))*

Root Mean Square Error (RMSE): 13.763243535775056

### Linear Regression example 2

import org.apache.spark.sql.functions.\_

import org.apache.spark.sql.Row

import org.apache.spark.sql.types.\_

import org.apache.spark.ml.regression.LinearRegression

import org.apache.spark.ml.feature.VectorAssembler

val csv = spark.read.option("inferSchema","true").option("header", "true").csv("wasb:///data/flights.csv")

csv.show()

val data = csv.select($"DayofMonth", $"DayOfWeek", $"OriginAirportID", $"DestAirportID", $"DepDelay", $"ArrDelay")

data.show()

val splits = data.randomSplit(Array(0.7, 0.3))

val train = splits(0)

val test = splits(1)

val train\_rows = train.count()

val test\_rows = test.count()

println("Training Rows: " + train\_rows + " Testing Rows: " + test\_rows)

val assembler = new VectorAssembler().setInputCols(Array("DayofMonth", "DayOfWeek", "OriginAirportID", "DestAirportID", "DepDelay")).setOutputCol("features")

val training = assembler.transform(train).select($"features", $"ArrDelay".cast("Int").alias("label"))

training.show()

val lr = new LinearRegression().setLabelCol("label").setFeaturesCol("features").setMaxIter(10).setRegParam(0.3)

val model = lr.fit(training)

println("Model Trained!")

val testing = assembler.transform(test).select($"features", $"ArrDelay".cast("Int").alias("trueLabel"))

testing.show()

val prediction = model.transform(testing)

val predicted = prediction.select("features", "prediction", "trueLabel")

predicted.show()

### Cross Validation for Regression

cross validation

// Import Spark SQL and Spark ML libraries

*import org.apache.spark.sql.types.\_*

*import org.apache.spark.sql.functions.\_*

*import org.apache.spark.ml.Pipeline*

*import org.apache.spark.ml.feature.VectorAssembler*

*import org.apache.spark.ml.regression.LinearRegression*

*import org.apache.spark.ml.tuning.{ParamGridBuilder, CrossValidator}*

*import org.apache.spark.ml.evaluation.RegressionEvaluator*

// Load the source data

*val csv = spark.read.option("inferSchema","true").option("header", "true").csv("wasb:///example/flightsv1.csv")*

// Select features and label

*val data = csv.select($"DayofMonth", $"DayOfWeek", $"OriginAirportID", $"DestAirportID", $"DepDelay", $"ArrDelay".alias("label"))*

// Split the data

*val splits = data.randomSplit(Array(0.7, 0.3))*

*val train = splits(0)*

*val test = splits(1).withColumnRenamed("label", "trueLabel")*

// Define the pipeline

*val assembler = new VectorAssembler().setInputCols(Array("DayofMonth", "DayOfWeek", "OriginAirportID", "DestAirportID", "DepDelay")).setOutputCol("features")*

*val lr = new LinearRegression().setLabelCol("label").setFeaturesCol("features")*

*val pipeline = new Pipeline().setStages(Array(assembler, lr))*

### Tune Parameters

You can tune parameters to find the best model for your data. To do this you can use the \*\*CrossValidator\*\* class to evaluate each combination of parameters defined in a \*\*ParameterGrid\*\* against multiple \*folds\* of the data split into training and validation datasets, in order to find the best performing parameters. Note that this can take a long time to run because every parameter combination is tried multiple times.

*val paramGrid = new ParamGridBuilder().addGrid(lr.regParam, Array(0.3, 0.01)).addGrid(lr.maxIter, Array(10, 5)).build()*

*val cv = new CrossValidator().setEstimator(pipeline).setEvaluator(new RegressionEvaluator).setEstimatorParamMaps(paramGrid).setNumFolds(2)*

*val model = cv.fit(train)*

*val prediction = model.transform(test)*

*val predicted = prediction.select("features", "prediction", "trueLabel")*

*predicted.show()*

+--------------------+-------------------+---------+

| features| prediction|trueLabel|

+--------------------+-------------------+---------+

|[1.0,1.0,10140.0,...|-7.2232267788877955| -16|

|[1.0,1.0,10397.0,...| 67.24202320873606| 47|

|[1.0,1.0,10397.0,...|-7.1569995407819595| -12|

|[1.0,1.0,10397.0,...| 1.8689596518846434| -19|

|[1.0,1.0,10397.0,...| 70.62264296110631| 74|

|[1.0,1.0,10693.0,...| 4.293382321272619| 2|

|[1.0,1.0,10693.0,...| -5.80780089050698| -1|

### Examine the Predicted and Actual Values

You can plot the predicted values against the actual values to see how accurately the model has predicted. In a perfect model, the resulting scatter plot should form a perfect diagonal line with each predicted value being identical to the actual value - in practice, some variance is to be expected.

Run the cells below to create a temporary table from the \*\*predicted\*\* DataFrame and then retrieve the predicted and actual label values using SQL. You can then display the results as a scatter plot, specifying \*\*-\*\* as the function to show the unaggregated values.

*predicted.createOrReplaceTempView("regressionPredictions")*

*%%sql*

*SELECT trueLabel, prediction FROM regressionPredictions*

| **trueLabel** | **prediction** |
| --- | --- |
| -16 | -7.223227 |
| 47 | 67.242023 |
| -12 | -7.157000 |
| -19 | 1.868960 |

### Retrieve the Root Mean Square Error (RMSE)

There are a number of metrics used to measure the variance between predicted and actual values. Of these, the root mean square error (RMSE) is a commonly used value that is measured in the same units as the prediced and actual values - so in this case, the RMSE indicates the average number of minutes between predicted and actual flight delay values. You can use the \*\*RegressionEvaluator\*\* class to retrieve the RMSE.

*val evaluator = new RegressionEvaluator().setLabelCol("trueLabel").setPredictionCol("prediction").setMetricName("rmse")*

*val rmse = evaluator.evaluate(prediction)*

*println("Root Mean Square Error (RMSE): " + (rmse))*

Root Mean Square Error (RMSE): 14.224652486105716