keep coding

BIG DATA PROCESSING

Fundamentos de Scala y Spark

Ivan Corbacho Fuerte

Índice

keep coding

Conceptos bases

keep coding

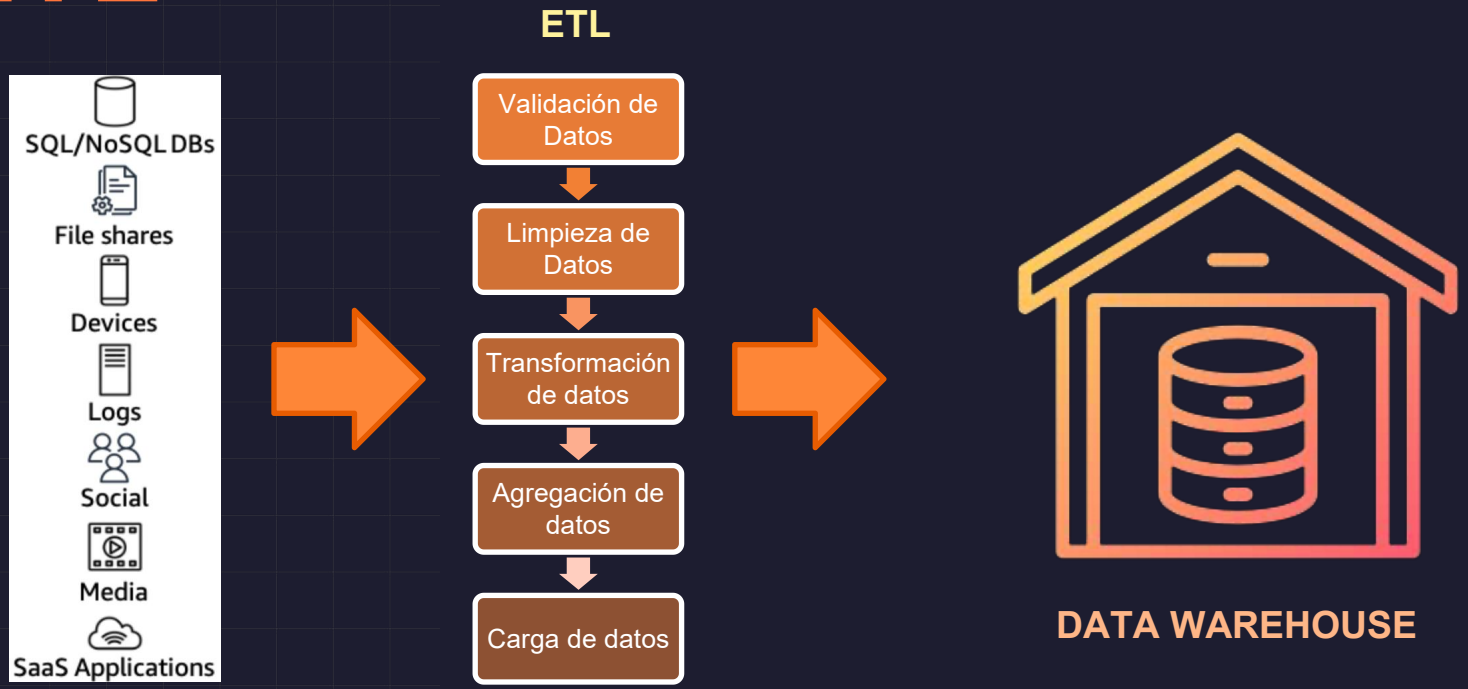
Scala

keep coding

Conceptos base

ETL - Build

Conceptos base
ETL



Conceptos Base
Build

```
def suma(a: Int, b: Int): Int = a + b
```

Build

```
public int suma(int a, int b) {
    return a + b;
}

// Bytecode correspondiente
0: iload_1
1: iload_2
2: iadd
3: ireturn
```

.class

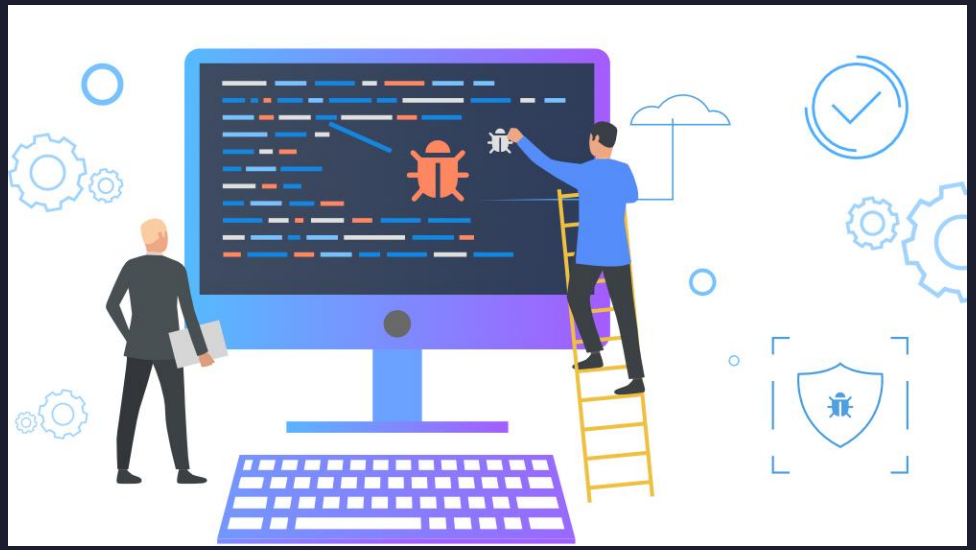
```
01010101 11001010 10101101...
```

keep coding

Conceptos Base
Scala?

- Basado en Java
- Lenguaje de programacion multiparadigma
- Conciso
- Elegante
- Tipado estatico (inferencia de tipo)
- Compila con la JVM

keep coding



Test

TDD

keep coding

Test
TDD

Principios Básicos

1. **Escribe una Prueba (Red)**
    1. Antes de escribir cualquier código nuevo, se escribe una prueba que define una pequeña mejora o nueva funcionalidad.
    2. Esta prueba inicialmente fallará porque la funcionalidad no está implementada todavía.
2. **Escribe el Código Mínimo Necesario (Green)**
    1. Se escribe el código más simple y directo que haga que la prueba pase.
    2. El objetivo es que la prueba pase lo más rápidamente posible.
3. **Refactoriza el Código (Refactor)**
    1. Se mejora el código escrito, asegurándose de que sigue pasando todas las pruebas.
    2. La refactorización se hace para mejorar la estructura del código, eliminar duplicación y optimizar el rendimiento.

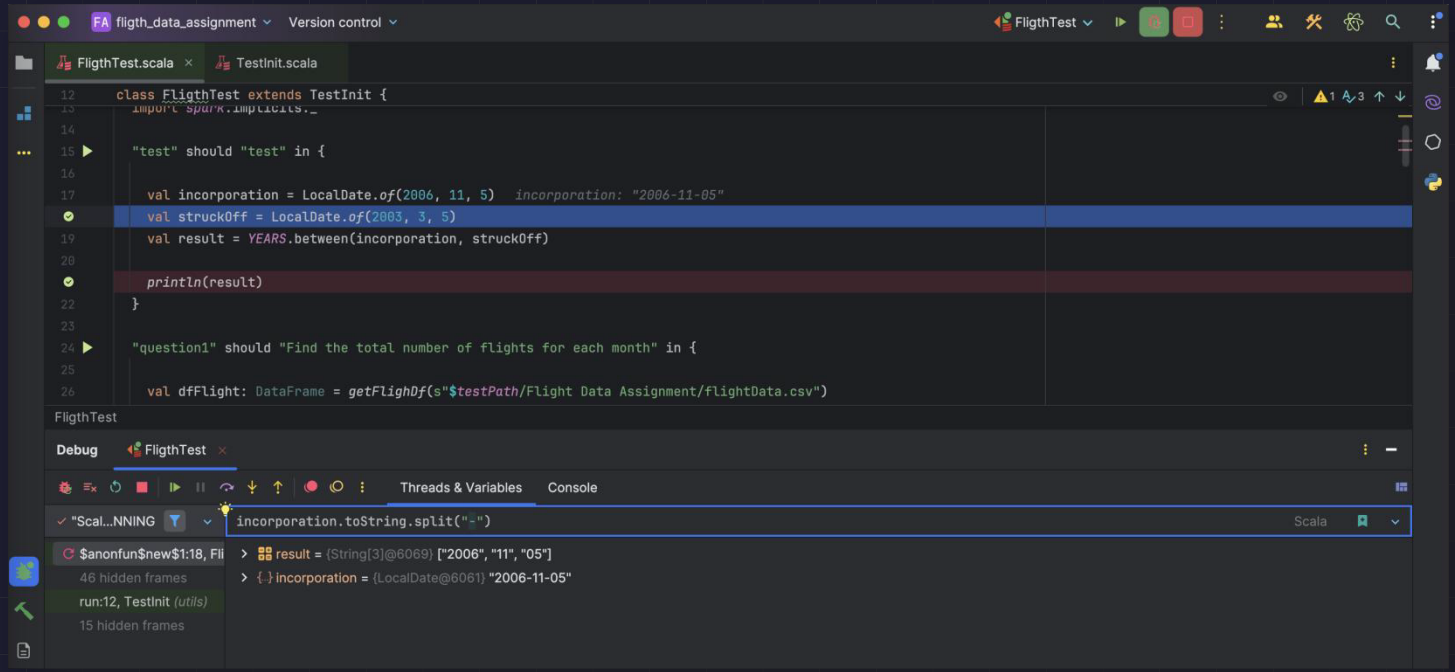
Test
TDD

Ventajas del TDD

- Mejor Diseño del Código
- Alta Cobertura de Pruebas
- Refactorización Segura
- Documentación Viva
- Retroalimentación Rápida

keep coding

Test
Debuging



keep coding

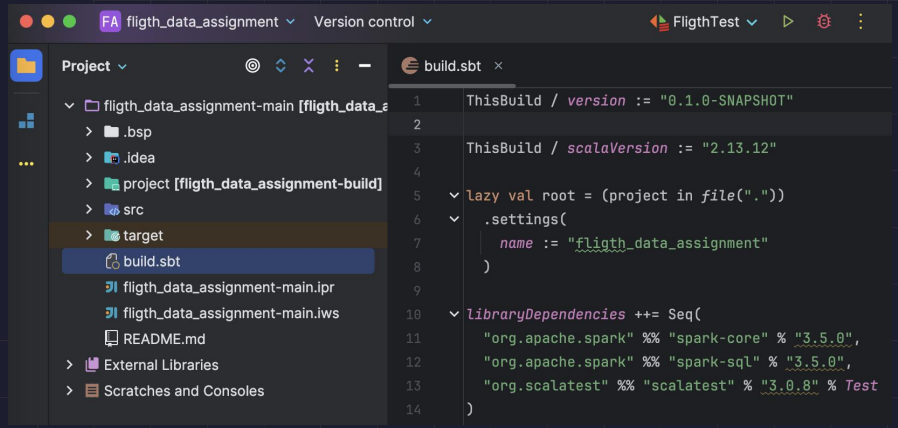
www.mvnrepository.com

Gestion de Dependencias

	Version	Scala	Vulnerabilities	Repository	Usages	Date
4.0.x	4.0.0-preview1	2.13		Central	23	Jun 03, 2024
3.5.x	3.5.1	2.13 2.12		Central	91	Feb 22, 2024
	3.5.0	2.13 2.12		Central	104	Sep 13, 2023
3.4.x	3.4.3	2.13 2.12		Central	31	Apr 18, 2024
	3.4.2	2.13 2.12		Central	74	Nov 30, 2023
	3.4.1	2.13 2.12		Central	95	Jun 23, 2023
	3.4.0	2.13 2.12		Central	91	Apr 13, 2023
3.3.x	3.3.4	2.13 2.12		Central	58	Dec 16, 2023
	3.3.3	2.13 2.12		Central	38	Aug 21, 2023
	3.3.2	2.13 2.12		Central	124	Feb 16, 2023
	3.3.1	2.13 2.12		Central	93	Oct 22, 2022

Maven | Gradle | Gradle (Short) | Gradle (Kotlin) | SBT | Ivy | Grape | Leiningen | Buildr

```
<!-- https://mvnrepository.com/artifact/org.apache.spark/spark-core -->
<dependency>
    <groupId>org.apache.spark</groupId>
    <artifactId>spark-core_2.13</artifactId>
    <version>4.0.0-preview1</version>
</dependency>
```



keep coding

Scala

keep coding

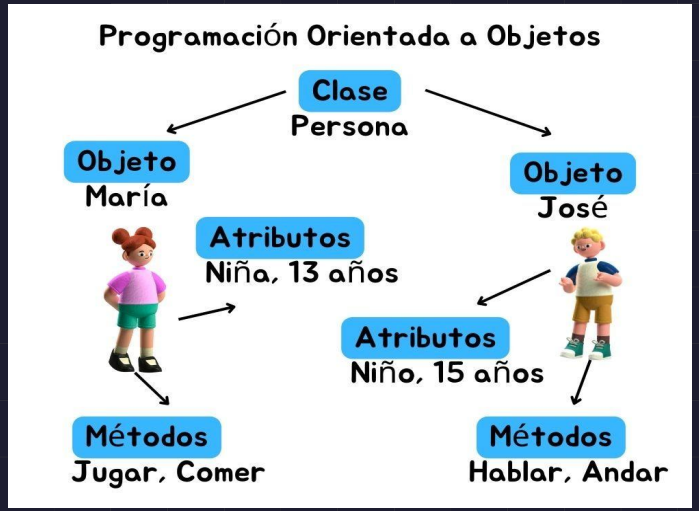
Scala

ETL - Build

keep coding

Scala - Clases y Objetos

- **Clase:**
  - Plantilla compuesto de atributos y métodos, que los objetos creados a partir de la clase tendrán.
- **Objeto:**
  - Instancia de una clase.
  - Un objeto contiene datos y métodos definidos por su clase.
- **Atributo:**
  - Caracterísiticas de un objeto.
- **Método:**
  - Conjunto de instrucciones que realizan una determinada tarea.



Scala - Tipos

- **Colección**
  - Scala proporciona una rica colección de tipos de datos para manejar conjuntos de elementos:

```
List: Una lista inmutable.
val list: List[Int] = List(1, 2, 3)

Array: Un arreglo mutable de tamaño fijo.
val array: Array[Int] = Array(1, 2, 3)

Set: Un conjunto inmutable de elementos únicos.
val set: Set[Int] = Set(1, 2, 3)

Map: Un mapa inmutable de pares clave-valor.
val map: Map[String, Int] = Map("a" -> 1, "b" -> 2)

Vector: Una secuencia inmutable optimizada para acceso aleatorio.
val vector: Vector[Int] = Vector(1, 2, 3)
```

Scala - Tipos

- **Genéricos**
  - Scala permite la definición de tipos genéricos para clases y métodos, lo que proporciona flexibilidad y reutilización de código.

```
class Box[T](value: T) { def get: T = value }
val intBox = new Box
val stringBox = new Box[String]("hello")

println(intBox.get)
println(stringBox.get)
```

Scala - Tipos

- **Función**
  - Las funciones en Scala son valores de primera clase y tienen sus propios tipos.

```
val add: (Int, Int) => Int = (a: Int, b: Int) => a + b

println(add(1,1))
```

Scala - Tipos

- **Opción (Option)**
  - El tipo Option se utiliza para manejar valores que pueden estar presentes o ausentes sin utilizar null.

```scala
val someValue: Option[Int] = Some(5)
val noValue: Option[Int] = None

println(someValue)
println(noValue)
println(someValue.get)
println(someValue.getOrElse("No"))
```

Scala - Tipos

- **Case Class**
  - Las case classes son clases especiales en Scala que se utilizan para la coincidencia de patrones y tienen características adicionales como la igualdad estructural y la descomposición automática.

```scala
case class Person(name: String, age: Int)
val person = Person("Alice", 25)
println(person.name)
```

keep coding

Scala - Tipos

- **Implícitos**
  - Scala tiene un sistema de implicits que permite la conversión automática entre tipos y la provisión automática de valores.

```
implicit val defaultInt: Int = 10
def addImplicit(a: Int)(implicit b: Int): Int = a + b
val result = addImplicit(5)
println(result)
```

Scala - Tipos

- **Existenciales**
  - Permiten que un tipo sea especificado solo en tiempo de ejecución.

```scala
val l = List(1,2,3)
val ls = List("1","2","3")
def printList(list: List[_]): Unit = list.foreach(println)

printList(l)
printList(ls)
```

Scala - POO

- **Encapsulamiento**
  - **Definición:** oculta los detalles internos de un objeto y expone lo necesario a través de métodos públicos.
  - **Objetivo:** Proteger los datos internos del objeto

```
class BankAccount(private var balance: Double) {
  def depositar(amount: Double): Unit = {
    if (amount > 0) balance += amount
  }

  def retirar(amount: Double): Unit = {
    if (amount > 0 && amount <= balance) balance -= amount
  }

  def getBalance: Double = balance
}

val account = new BankAccount(1000)
account.depositar(500)
account.retirar(200)
println(account.getBalance)
```

keep coding

Scala - POO

- **Abstracción**
  - **Definición:** Es el proceso de simplificar la complejidad mediante la ocultación de detalles innecesarios y la exposición de las características esenciales de un objeto.
  - **Objetivo:** Facilitar la gestión de sistemas complejos dividiéndolos en partes más manejables

```
abstract class Vehicle {
  def start(): Unit
  def stop(): Unit
}

class Car extends Vehicle {
  override def start(): Unit = println("Car is starting")

  override def stop(): Unit = println("Car is stopping")
}

class Bike extends Vehicle {
  override def start(): Unit = println("Bike is starting")

  override def stop(): Unit = println("Bike is stopping")
}

val myCar = new Car()
myCar.start()  // Output: Car is starting
myCar.stop()   // Output: Car is stopping

val myBike = new Bike()
myBike.start() // Output: Bike is starting
myBike.stop()  // Output: Bike is stopping
```

keep coding

Scala - POO

- **Herencia**
  - **Definición:** Es el mecanismo por el cual una clase (subclase) puede heredar propiedades y métodos de otra clase (superclase).
  - **Objetivo:** Promover la reutilización del código y establecer una jerarquía de clases que facilita la creación de nuevas funcionalidades basadas en clases existentes.

```
class Animal {
  def makeSound(): Unit = println("Some generic animal sound")
}

class Dog extends Animal {
  override def makeSound(): Unit = println("Woof!")
}

val myDog = new Dog()
myDog.makeSound()
```

Scala - POO

- **Polimorfismo**
  - **Definición:** Es la capacidad de un objeto para tomar múltiples formas. En términos de POO, permite que una sola interfaz sea utilizada para representar diferentes tipos de objetos.
  - **Objetivo:** Facilitar la flexibilidad y la interoperabilidad del código, permitiendo el uso de una única interfaz para diferentes implementaciones.

```
Definimos una clase base o superclase llamada Animal
abstract class Animal {
  def makeSound(): String
}
Definimos varias subclases que heredan de Animal
class Dog extends Animal {
  override def makeSound(): String = "Woof"
}

class Cat extends Animal {
  override def makeSound(): String = "Meow"
}

class Cow extends Animal {
  override def makeSound(): String = "Moo"
}

Podemos crear una lista de Animal que contenga diferentes tipos de animales
val animals: List[Animal] = List(new Dog(), new Cat(), new Cow())

/** Podemos iterar sobre la lista y llamar a 'makeSound' en cada uno */
animals.foreach(animal => println(animal.makeSound()))
```

Scala - PF

- **Funciones Puras**
  - **Definición:** Una función pura es aquella que, dada la misma entrada, siempre produce la misma salida y no tiene efectos secundarios observables.
  - **Ventajas:**
    - Facilitan el razonamiento sobre el código.
    - Hacen el código más predecible y fácil de testear.
    - Permiten la memoización (almacenamiento en caché de los resultados de funciones).

```
def square(x: Int): Int = x * x

println(square(5))  // Output: 25
println(square(5))  // Output: 25 (siempre produce el mismo resultado)
```

Scala - PF

- **Inmutabilidad**
  - **Definición:** Los datos no cambian una vez que han sido creados. En lugar de modificar estructuras de datos, se crean nuevas versiones con los cambios aplicados.
  - **Ventajas:**
    - Evita problemas de concurrencia.
    - Simplifica el seguimiento

```
val list = List(1, 2, 3)
val newList = list :+ 4  // Crea una nueva lista sin modificar la original

println(list)    // Output: List(1, 2, 3)
println(newList) // Output: List(1, 2, 3, 4)
```

Scala - PF

- **Funciones de Orden Superiror**
  - **Definición:** Son funciones que toman otras funciones como argumentos o devuelven funciones como resultado.
  - **Ventajas:**
    - Permiten la abstracción de patrones comunes.
    - Hacen el código más modular y reutilizable.

```
def applyTwice(f: Int => Int, x: Int): Int = f(f(x))

def increment(x: Int): Int = x + 1

println(applyTwice(increment, 5))  // Output: 7
```

Scala - PF

- **Composición de Funciones**
  - **Definición:** Es el proceso de combinar dos o más funciones para producir una nueva función.
  - **Ventajas:**
    - Facilita la creación de funciones complejas a partir de funciones simples.
    - Promueve la reutilización de código.

```
def addOne(x: Int): Int = x + 1
def double(x: Int): Int = x * 2
def addOneAndDouble(x: Int) = addOne(double(x))

println(addOneAndDouble(3))  // Output: 8
```

Scala - PF

- **Evaluación Perezosa (Lazy Evaluation)**
  - **Definición:** Las expresiones no se evalúan hasta que su valor sea realmente necesario.
  - **Ventajas:**
    - Permite la definición de estructuras de datos infinitas.
    - Puede mejorar el rendimiento al evitar cálculos innecesarios.

```
lazy val lazyVal = { println("Evaluating lazyVal") ; 42 }

println("Before accessing lazyVal")
println(lazyVal)  // Output: Evaluating lazyVal \n 42
println(lazyVal)  // Output: 42 (no se vuelve a evaluar)
```

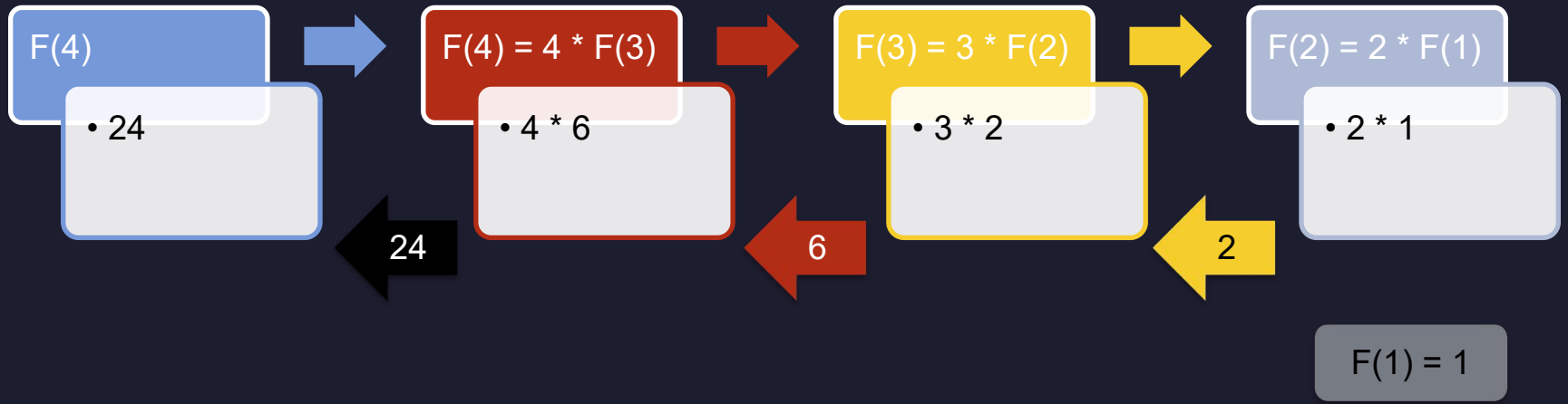
Scala - PF

- **Recursividad**
  - **Definición:** Es la técnica de definir una función en términos de sí misma.
  - **Ventajas:**
    - Puede ser una alternativa a las estructuras de control iterativas.
    - Facilita el trabajo con estructuras de datos recursivas (como listas y árboles).

```
def sumList(lst: List[Int]): Int = lst match {
  case Nil => 0
  case head :: tail => head + sumList(tail)
}
println(sumList(List(1, 2, 3, 4)))  // Output
```

Scala - PF

F(X) = X * F(X-1) ; F(1) = 1 ; X > 0



keep coding

Spark

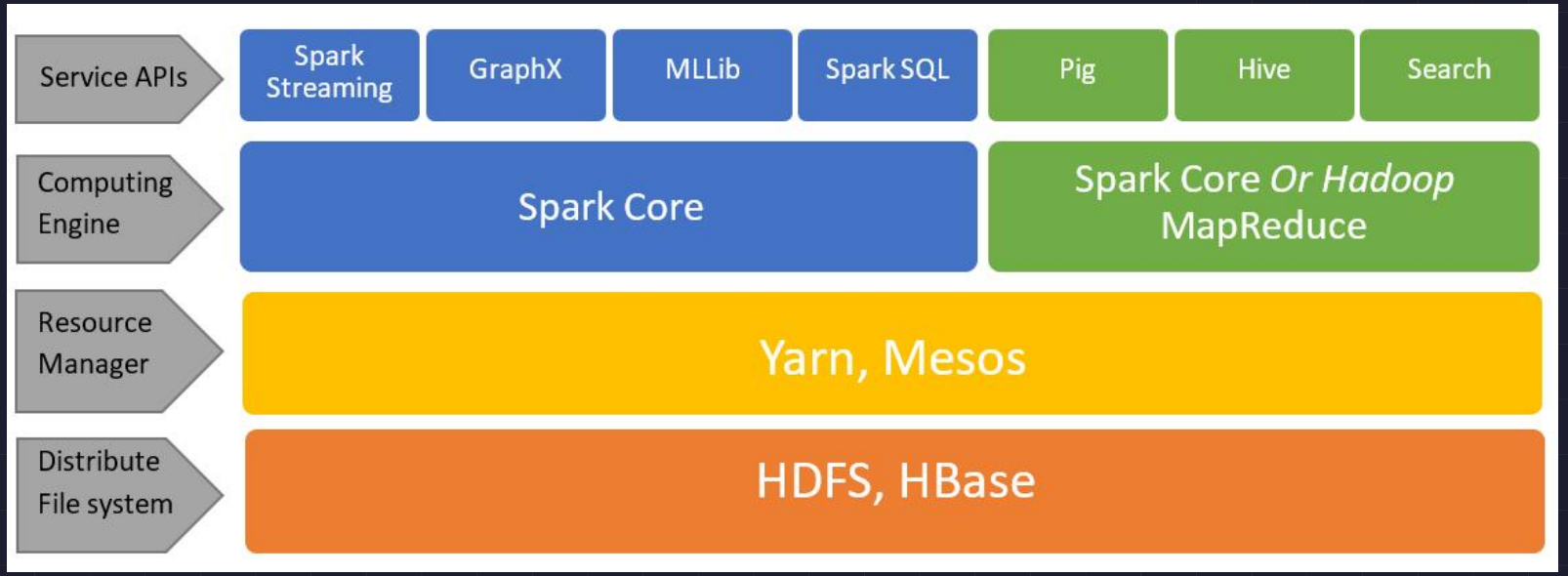
Spark

Apache Spark es un framework de procesamiento de datos de código abierto que se utiliza para procesar grandes volúmenes de datos de manera rápida y eficiente.

- **Velocidad:** Spark trabaja principalmente en memoria.
- **Facilidad de uso:** Spark proporciona APIs fáciles de usar para trabajar con grandes conjuntos de datos en Python, Java, Scala y R.
- **Módulos integrados:** Spark incluye una serie de bibliotecas integradas que facilitan el desarrollo:
  - **Spark SQL:** Para trabajar con datos estructurados y realizar consultas SQL.
  - **Spark Streaming:** Para el procesamiento de datos en tiempo real.
  - **MLlib:** Para machine learning.
  - **GraphX:** Para el procesamiento de gráficos.
- **Resiliencia y tolerancia a fallos:** Spark utiliza una estructura de datos llamada **RDD** (Resilient Distributed Datasets.
- **Escalabilidad**: Spark puede escalar horizontalmente a miles de nodos en un clúster, lo que permite procesar grandes volúmenes de datos de manera eficiente.

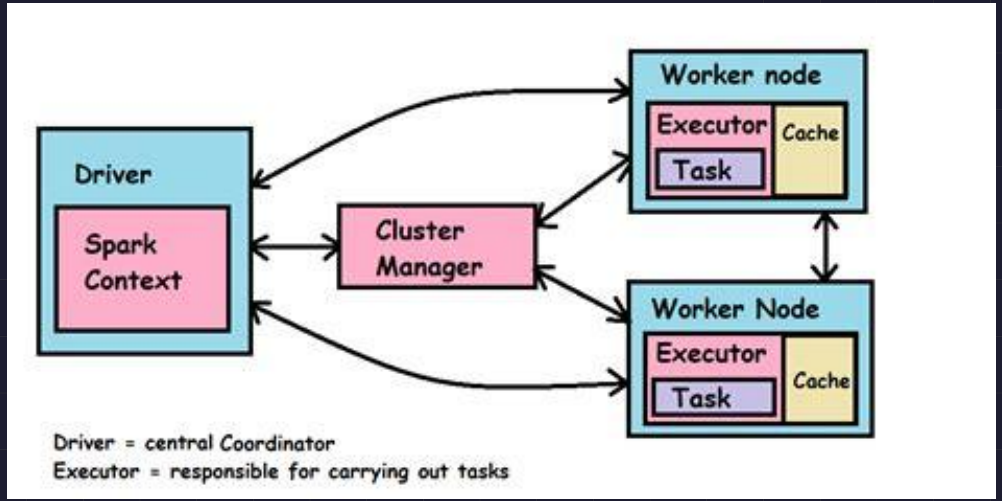
keep coding

Spark- Diagrama



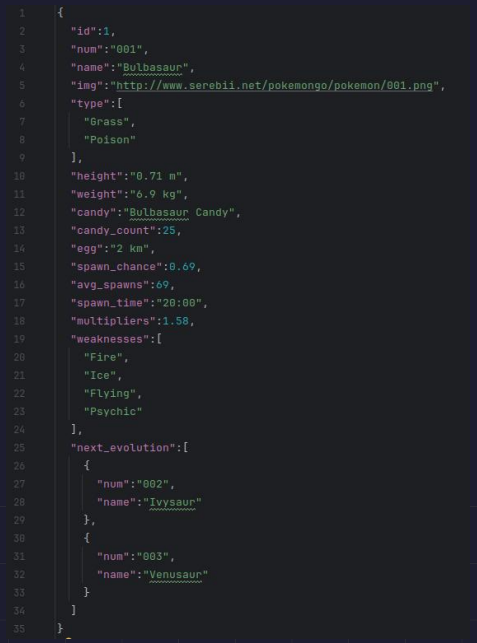
Spark- SparkContext

- Es lo primero que se crea en un programa.
- Da acceso al Cluster Manager
- En la Spark-Shell, por defecto se crea uno que se controla con la variable “sc”



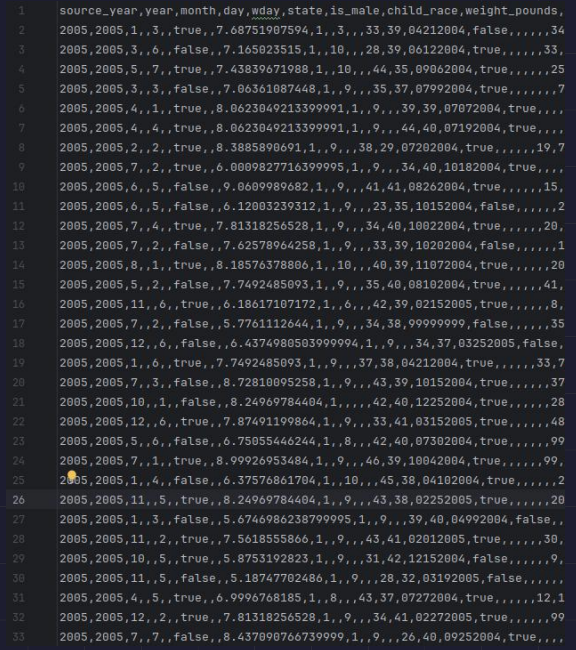
Ficheros- Json/Csv

JSON



```
{
  "id":1,
  "num":"001",
  "name":"Bulbasaur",
  "img":"http://www.serebii.net/pokemongo/pokemon/001.png",
  "type":[
    "Grass",
    "Poison"
  ],
  "height":"0.71 m",
  "weight":"6.9 kg",
  "candy":"Bulbasaur Candy",
  "candy_count":25,
  "egg":"2 km",
  "spawn_chance":0.69,
  "avg_spawns":69,
  "spawn_time":"20:00",
  "multipliers":1.58,
  "weaknesses":[
    "Fire",
    "Ice",
    "Flying",
    "Psychic"
  ],
  "next_evolution":[
    {
      "num":"002",
      "name":"Ivysaur"
    },
    {
      "num":"003",
      "name":"Venusaur"
    }
  ]
}
```

CSV



```
source_year,year,month,day,wday,state,is_male,child_race,weight_pounds,
2005,2005,1,,3,,true,,7.68751907594,1,,3,,,33,39,04212004,false,,,,,34
2005,2005,3,,6,,false,,7.165023515,1,,10,,,28,39,06122004,true,,,,,33,
2005,2005,5,,7,,true,,7.43839671988,1,,10,,,44,35,09062004,true,,,,,25
2005,2005,3,,3,,false,,7.06361087448,1,,9,,,35,37,07992004,true,,,,,,7
2005,2005,4,,1,,true,,8.0623049213399991,1,,9,,,39,39,07072004,true,,,,
2005,2005,4,,4,,true,,8.0623049213399991,1,,9,,,44,40,07192004,true,,,,
2005,2005,2,,2,,true,,8.3885890691,1,,9,,,38,29,07202004,true,,,,,,19,7
2005,2005,7,,2,,true,,6.0009827716399995,1,,9,,,34,40,10182004,true,,,,
2005,2005,6,,5,,false,,9.0609989682,1,,9,,,41,41,08262004,true,,,,,,15,
2005,2005,6,,5,,false,,6.12003239312,1,,9,,,23,35,10152004,false,,,,,,2
2005,2005,7,,4,,true,,7.81318256528,1,,9,,,34,40,10022004,true,,,,,,20,
2005,2005,7,,2,,false,,7.62578964258,1,,9,,,33,39,10202004,false,,,,,,1
2005,2005,8,,1,,true,,8.18576378806,1,,10,,,40,39,11072004,true,,,,,,20
2005,2005,5,,2,,false,,7.7492485093,1,,9,,,35,40,08102004,true,,,,,,41,
2005,2005,11,,6,,true,,6.18617107172,1,,6,,,42,39,02152005,true,,,,,,8,
2005,2005,7,,2,,false,,5.7761112644,1,,9,,,34,38,99999999,false,,,,,,35
2005,2005,12,,6,,false,,6.4374980503999994,1,,9,,,34,37,03252005,false,
2005,2005,1,,6,,true,,7.7492485093,1,,9,,,37,38,04212004,true,,,,,,33,7
2005,2005,7,,3,,false,,8.72810095258,1,,9,,,43,39,10152004,true,,,,,,37
2005,2005,10,,1,,false,,8.24969784404,1,,,,,42,40,12252004,true,,,,,,28
2005,2005,12,,6,,true,,7.87491199864,1,,9,,,33,41,03152005,true,,,,,,48
2005,2005,5,,6,,false,,6.75055446244,1,,8,,,42,40,07302004,true,,,,,,99
2005,2005,7,,1,,true,,8.99926953484,1,,9,,,46,39,10042004,true,,,,,,99,
2005,2005,1,,4,,false,,6.37576861704,1,,10,,,45,38,04102004,true,,,,,,2
2005,2005,11,,5,,true,,8.24969784404,1,,9,,,43,38,02252005,true,,,,,,20
2005,2005,1,,3,,false,,5.6746986238799995,1,,9,,,39,40,04992004,false,,
2005,2005,11,,2,,true,,7.5618555866,1,,9,,,43,41,02012005,true,,,,,,30,
2005,2005,10,,5,,true,,5.8753192823,1,,9,,,31,42,12152004,false,,,,,,9,
2005,2005,11,,5,,false,,5.18747702486,1,,9,,,28,32,03192005,false,,,,,,
2005,2005,4,,5,,true,,6.9996768185,1,,8,,,43,37,07272004,true,,,,,,12,1
2005,2005,12,,2,,true,,7.81318256528,1,,9,,,34,41,02272005,true,,,,,,99
2005,2005,7,,7,,false,,8.437090766739999,1,,9,,,26,40,09252004,true,,,,
```

Ficheros - Avro vs Parquet

order_id	Country	Date
1	Spain	12/12/2024
2	Portugal	13/12/2024
3	Italy	14/12/2024

ROW 1	1|Spain|12/12/2024
ROW 2	2|Portugal|13/12/2024
ROW 3	3|Italy|14/12/2024

order_id	1|2|3
Country	Spain|Portugal|Italy
Date	12/12/2024|13/12/2024|14/12/2024

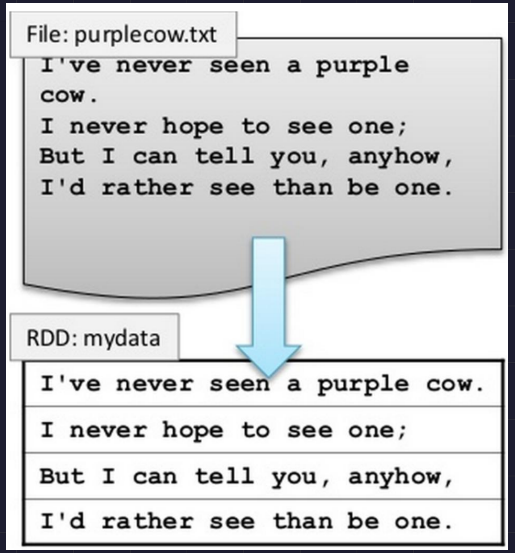
Ficheros - Resumen

- **Avro:** Almacena datos en filas. Bueno para escritura intensiva y transmisión de datos. Incluye esquema con los datos.
- **Parquet:** Almacena datos en columnas. Ideal para lectura intensiva y análisis. Alta compresión y rendimiento en consultas.
- **JSON:** Formato jerárquico y flexible, adecuado para datos complejos y APIs. Fácil de leer y escribir, pero más pesado.
- **CSV:** Formato tabular y simple, ideal para datos estructurados y hojas de cálculo. Ligero, pero no soporta datos complejos.

Spark - RDD

- RDD es la unidad básica de Computación en Spark
- Conceptualmente similar a una Lista tradicional
- RDD contiene cualquier tipo de elementos
  - Primitivos(int,float,etc)
  - Secuencias (tuplas, listas, dicts, etc.)
  - Objetosserializables
- Un RDD se crea a partir de una fuente de datos o a partir de otro RDD

Transformaciones RDD -> RDD'	Acciones RDD -> Result
Map	count
flatMap	reduce
filter	collect
groupBy	…
join	



keep coding

Spark - Variables Accumulators

- Los accumulators son variables que sirve de contadores, ya que se pueden smar informacion en operaciones asociativas
- Sirven para implementar contadores y acumuladores distribuidos entre todo el cluster de Spark
- Se soportan de manera nativa acumuladores de numeros y de colecciones mutables
- Se pueden extender para que se soporten mas tipos
- Los acumuladores no se pueden leer desde las tareas, solo desde la aplicacion del driver.

```
my_accumulator = sc.accumulator(0)
my_accumulator.add(1)
print my_accumulator.value
```

keep coding

Spark - Variables Accumulators

- Permite guardar un valor de solo lectura cacheado en todas las maquihna implicadas en el procesamiento.
- Estas variables estaran disponible en todas las tareas que se lancen.
- Utili para distribuir un gran dataset que se va a utilizer muchas veces. (Tamaño pequeño).
- Reduce la comunicacion entre sistemas diferentes.

```
val myVariable= ... // p.e diccionario muy pesado my_VariableBroad = sc.broadcast(myVariable)
def myF(s, d) = d[s]
def myF_vbleB(s) = myVariableBroad.value[s]
val myRDD2 = myRDD.map(myF(d))
val myRDD3 = myRDD.map(myF_vbleB) // más eficiente
```

Tipos

RDD

- Original (2010)
- Colección de objetos JVM
- Operaciones funcionales (map, filter, reduce...)
- Type-safe

ob_Silvia: Persona
ob_Pepe: Persona
ob_Mariia: Persona

DATAFRAME

- Evolución del SchemaRDD
- Colección de objetos Row
- Operaciones basadas en expresiones
- Más eficiente por optimización
- No type-safe

[Silvia: string, 33: int]: Row
[Pepe: string, 22: int]: Row
[Maria: string, 19: int]: Row

DATASET

- Aparece en v1.6, oficial en 2.0 (2016)
- Internamente colección de Row, externamente objetos JVM
- Mezcla de RDD y DataFrame
- Eficiente por optimización
- Type-safe

ob_Silvia: Persona
ob_Pepe: Persona
ob_Mariia: Persona

Spark - SQL

```
// Spark SQL
df.registerTempTable("table1")
val output = sqlContext.sql("SELECT * FROM table1 WHERE col1='xxxx'")
```

Resumen

keep coding

Inicializando SparkSession

```
import org.apache.spark.sql.SparkSession

val spark = SparkSession.builder()
    .appName("MyApp")
    .config("spark.some.config.option", "some-value")
    .getOrCreate()
```

Crear DataFrames

RDDs

```
import org.apache.spark.sql.Row
import org.apache.spark.rdd.RDD

val sc = spark.sparkContext
val lines = sc.textFile("people.txt")
val parts = lines.map(_.split(","))
val people: RDD[Row] = parts.map(p => Row(p(0), p(1).toInt))
val peopledf = spark.createDataFrame(people)
```

Data Sources

```
val df = spark.read.json( )
df.show()

"customer.json"
val df2 = spark.read.format("json").load("people.json")
val df3 = spark.read.parquet("users.parquet")
val df4 = spark.read.text("people.txt")
```

Filtro

```
df.filter(col("age") > 24).show
df.filter("age > 24").show
```

Eliminacion de duplicados

```
val df = df.dropDuplicates()
```

Inicializando SparkSession

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

// Select
 df.select("firstName").show() // Show all entries in firstName column
 df.select("firstName", "lastName").show()
 df.select("firstName", "age",
     explode(col("phoneNumber")).alias("contactInfo"))
.select("contactInfo.type", "firstName","age")
.show()

df.select(col("firstName"), col("age") + 1).show()

df.select(col("age") > 24).show() // Show all entries where age > 24

// When
 df.select(col("firstName"), when(col("age") > 30, 1).otherwise(0)).show()

df.filter(col("firstName").isin("Jane", "Boris")).collect()

// Like
df.select(col("firstName"), col("lastName").like("Smith")).show()

// Startswith - Endswith
df.select(col("firstName"),col("lastName").startsWith("Sm")).show()

df.select(col("lastName").endsWith("th")).show()

// Substring
df.select(col("firstName").substr(1, 3).alias("name")).collect()

// Between
df.select(col("age").between(22, 24)).show()
```

Add, Update & Remove Column

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

// Agregar columnas
val df = dataDf.withColumn("city", col("address.city"))
        .withColumn("postalCode", col("address.postalCode"))
        .withColumn("state", col("address.state"))
        .withColumn("streetAddress", col("address.streetAddress"))
        .withColumn("telePhoneNumber",
explode(col("phoneNumber.number")))
        .withColumn("telePhoneType", explode(col("phoneNumber.type")))

// Renombrar columna
val dfRenamed = df.withColumnRenamed("telePhoneNumber","phoneNumber")

// Eliminar columnas
val dfFinal = dfRenamed.drop("address", "phoneNumber")
```

Missing & Replacing Values

```
// Reemplazar valores nulos con 50
df.na.fill(50).show()

// Devolver un nuevo DataFrame omitiendo filas con valores nulos
df.na.drop().show()

// Devolver un nuevo DataFrame reemplazando un valor con otro
df.na.replace(10, 20).show()
```

Agrupar, Ordenar y Sql

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

df.groupBy("age") .count()

// Ordenar por edad en orden descendente
peopledf.sort(col("age").desc).collect()

// Ordenar por edad en orden descendente (otra forma)
df.sort(col("age").desc).collect()

// Ordenar por edad en orden descendente y ciudad en orden
ascendente
df.orderBy(col("age").desc, col("city").asc).show()

// **Reparticionamiento**

// Crear un DataFrame con 10 particiones
df.repartition(10).rdd.getNumPartitions()

// Reducir el número de particiones a 1
df.coalesce(1).rdd.getNumPartitions()

// **Ejecutar consultas programáticamente**

// Registrar DataFrames como vistas
peopledf.createGlobalTempView("people")
df.createTempView("customer")
df.createOrReplaceTempView("customer")

// **Consultar vistas**
val df5 = spark.sql("SELECT * FROM customer")
df5.show()
val peopledf2 = spark.sql("SELECT * FROM
global_temp.people").show()
```

Inspeccionar Datos

```
df.dtypes // Devuelve los nombres de las columnas y sus tipos de datos
df.show() // Muestra el contenido del DataFrame
df.head() // Devuelve las primeras n filas (por defecto 1)
df.first() // Devuelve la primera fila
df.take(2) // Devuelve las primeras 2 filas
df.schema // Devuelve el esquema del DataFrame
df.describe().show() // Calcula estadísticas resumidas
df.columns // Devuelve las columnas del DataFrame
df.count() // Cuenta el número de filas en el DataFrame
df.distinct().count() // Cuenta el número de filas distintas en el
DataFrame
df.printSchema() // Imprime el esquema del DataFrame
df.explain() // Imprime los planes lógico y físico de ejecución
```

Escritura

```
// Guarda los datos en formato Parquet
df.select("firstName", "city")
  .write
  .mode("overwrite")
  .parquet("nameAndCity.parquet")

 // Guarda los datos en formato JSON
df.select("firstName", "age")
  .write
  .format("json")
  .save("namesAndAges.json")
```

Detener Spark

```
spark.stop()
```

www.keepcoding.io cursos@keepcoding.io (+34) 916 33 1779