Scala allows interaction between distributed databases and empowers parallel data processing to reduce time.
The language is known for big data processing and containing large data into scalable volumes to make decisions.
Scala was introduced as a modification of Java. This means that it also supports Java’s API and libraries, which is beneficial for developers with a Java background.
Most of the database is in immutable format and Scala has higher-order functions to deal with this data just like in Python.
A vast majority of constructs in Scala simplify interactions with wrapper classes and container types.

Why should you use Scala?

Object-oriented programming

Everything in the language can be defined in objects and classes. Inheritance, Escalation, and Polymorphism are features of OOP in Scala. It’s useful for performing flexible composition operations by expanding classes as a substitute for multiple inheritances.

Functional programing language

Scala supports higher-order functions and comprises easy-to-remember syntax for defining functions. It provides nested functions, currying, and classes to group algebraic operations.

Statically typed Scala

Expressions defined in Scala verify if abstractions are consistently used across the program in a compile time. It ensures the following:

Generic classes
Placement of annotation
Upper and lower case details
OOP - Inner classes and abstract data
Polymorphic methods
Implicit conversions and parameters

Language extension

It’s said that construction of domain-specific applications are practically suitable with domain-specific language extensions. Any new language constructs can be added in Scala in the form of libraries. It’s even feasible without using meta-programming tools and methods like macros. Here’s how:

Implicit classes are used to add extension methods into existing classes.
User-extensibility of string interpolation via custom interpolators.

Integration with Java

Being an upgrade over Java, Scala works smoothly with Java Runtime Environment (JRE) without any intricacies. It’s highly compatible with the latest Java features like SAMs, lambdas, generics, and annotations.

Characteristics of Scala that have no substitute for Java, such as default parameters, are also compiled close to Java. The compilation methodologies (separate compiling, dynamic classes) is equivalent to Java with access to rich quality libraries.

Scala as a tool for data analytics operations

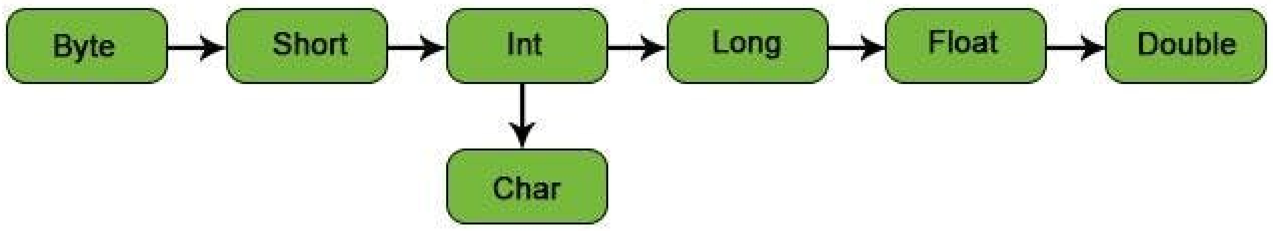
Data types

Type hierarchy
Any is the top type. It defines universal methods such as equals, hashCode, and toString. The subclasses of Any are AnyVal and AnyRef.

Anyval: It’s the class root of all value types that contains nine non-nullable value types: Double, Float, Long, Int, Short, Byte, Char, Unit, and Boolean.

AnyRef: It represents reference types declared as non-value types. AnyRef is the subclass of user-defined type. It defines java.lang.Object while using Scala in JRE.

Type casting


$$\begin{tikzcd}
\text{Byte} \arrow[r] & \text{Short} \arrow[r] & \text{Int} \arrow[r] \arrow[d] & \text{Long} \arrow[r] & \text{Float} \arrow[r] & \text{Double} \\
 & & \text{Char} & & &
\end{tikzcd}$$

Nothing and Null

Nothing is the bottom type and subset of all types of values. There is no value for type nothing. Non-termination, such as an exception thrown, program exit or an infinite loop, is the normal output in nothing type.

A null subtype is common for all reference types. Null gives only one value denoted literally as null. It’s initially meant for interoperability with other JVM languages and must be excluded in Scala programs.

Expressions

Expressions are sentences or statements that are generated through programs.
println is used to get the output of the expression.

Value
The word Val determines the result of the expression.
Variable
The type of a variable can be anything like the type of a value. It can also be named with expressions.
Blocks
All the expressions are stored inside blocks typing {}.

Functions and methods in Scala

Functions

Function consists of a number of statements that are used to perform a task. The following is the form of function declaration in Scala.

```
def functionName ([list of parameters]) : [return type]
```

Methods

Methods are almost the same as functions but there are some differences. The word def defines methods. def provides name, parameter list, return type, and body after inputting the data.

```
def add(x: Int, y: Int): Int = x + y
println(add(3, 2)) // 5
```

Main method

In Scala, the program begins with the main method. JVM requires the main method of a single parameter comprising input in the form of an array of strings. Here’s an example:

```
object Main { def main(args: Array[String]): Unit =
println(″Hello, Scala Learner!″) }
```

Classes and objects in Scala

Classes

Constructor parameters are written after the word class and are used to define the class.

```
class Greeter(prefix: String, suffix: String) { def greet(name:
String): Unit =
println(prefix + name + suffix) }
```

Class instance can be created with the word new.

```
val greeter = new Greeter("Hello, ", "!")
greeter.greet("Scala Learner") // Hello, Scala Learner!
```

Case classes

A special type of class in Scala is case class that has immutable objects in default. These objects are measured by values in contrast to classes where instances are measured by reference. Hence, it becomes more crucial for pattern matching.

The word case class is used for defining case classes.

```
case class Point(x: Int, y: Int)
```

Objects

The word object is used for defining objects.

```
object IdFactory
{ private var counter = 0 def create (): Int = { counter += 1 counter
} }
```

Packages and imports

Creating a package

You can modularize a program using packages. They're defined on top of the Scala file by stating the namespace of packages.

```
package users
class User
```

Imports

Imports features are used to access other package elements such as classes and functions. However, it isn't necessary to use import statements for accessing elements from the same package.

```
import users._  // import everything from the users package
import users.User  // import the class User
import users.{User, UserPreferences}  // Only imports selected
members
import users.{UserPreferences => UPrefs}  // import and rename for
convenience
```

Parallel collection in Scala

The purpose of implementing parallel collection is the same as sequential collections. The only difference lies in the way parallel collection is acquired. There are two methods:

1. Use the term as a conjunction in correct order.

```
import statement: import
scala.collection.parallel.immutable.ParVector
val pv = new ParVector[Int]
```

1. Convert from sequential collection.

```
val pv = Vector(1,2,3,4,5,6,7,8,9).par
```

Semantics

Even though the abstraction of parallel collection resembles conventional sequential collection, they are still different in terms of semantics. Their side effects and non-associative operators are not the same and lead to non-determinism.

The question of whether to learn Scala or an alternative language is entirely subjective and based on the career stage of a data scientist. It also depends on future job perspectives where Python seems more reliable. On a side note, Julia is another force that looks set to capture the data analytics market in the near future.