Big and Fast Data

2.5B TB of data produced daily globally (2.5 EB-exabyte)

21.5B devices with internet access

by 2025: 463 EB daily, 75B devices

Main Vs:

-> **Volume** - large amounts of data

90% of all data created in the last 2 years

every human 1.7MB/s (2020)

306.4B emails, 500M Tweets daily

-> **Variety** - different forms of data from different sources

structured data - format is known (SQL tables)

semi-structured data (JSON, XML)

unstructured data (text, audio, video)

-> **Velocity** - quickly changing content

data center log files

sensor reporting

stock markets

Other Vs:

-> Value

-> Validity - sound, clean

-> Veracity - trustworthiness

-> Volatility - time-relevance

-> Visibility

-> Virality

Processing - **ETL** cycle

Extract - raw/semi-structured to structured data

Transform - convert units, join data sources, cleanup

Load - data into another system for further processing

big data engineering - building pipelines
big data analytics - discovering patterns

batch processing - all data in data store, program processes the whole dataset
stream processing - processing as data arrives to the system

Approaches to distributed data processing operations:

-> **data-parallelism** - divide the data, apply the same algorithm
-> **task-parallelism** - divide the problem, run on cluster of machines

Desired **properties**:

-> robustness and fault-tolerance
-> low latency reads and updates
-> scalability
-> generalisation
-> extensibility
-> ad hoc queries
-> minimal maintenance
-> debuggability

Large scale processing on **distributed**, **commodity** computers, enabled by advanced **software** using **elastic** resource allocation.
Big Data is **software**-driven industry.

Problems:

-> **modelling** - outcome influencing factors
-> **information retrieval** - search engines
-> **collaborative filtering** - similar user based recommendations
-> **outlier detection** - discovering outstanding transactions

Programming Languages for Big Data

Scala - data intensive systems

Python - data analytics tasks

Java - assembly of big data systems

R - serious data analytics with great plotting tools

Scala - combination of functional programming and object orientation, compiled

Python - combination of imperative programming and object orientation, interpreted

HelloWorld

Scala:

```
object Hello extends App {
    println("Hello, world")
    for (i <- 1 to 10) {
    System.out.println("Hello")
    }
}
```

blocks denoted by {}

insensitive to space/tab

Python:

```
for i in range(1, 10):
    print("Hello, world")
```

blocks denoted by tab (or double space)

Declarations

Scala:

```
val a: Int = 5
val b = 5
b = 6 // re-assignment to val

// Type of foo is inferred
val foo = new ImportantClass(...)

var a = "Foo"
a = "Bar"
a = 4 // type mismatch
```

single-assignment

multiple assignment

type inference

Python:

```
a : int = 5
a = "Foo"

a = ImportantClass(...)
```

Declaring functions

Scala:

```
def max(x: Int, y: Int): Int =
  if (x >= y) x else y
```

statically typed
typed expressions
return type

Python:

```
def max(x : int, y : int) -> int:
  if x >= y:
    return x
  else:
    return y
```

dynamically typed
statements
type-optional

Higher order functions

Scala:

```
def bigger(x: Int, y: Int,
  f: (Int,Int) => Boolean) =
  f(x, y)

bigger (1, 2, (x, y) => (x < y))
bigger (1, 2, (x, y) => (x > y))
// Compile error
bigger (1, 2, x => x)
```

true
false
→ 2 - parameter function

Python:

```
def bigger(x, y, f):
  return f(x, y)

bigger(1,2, lambda x,y: x > y)
bigger(1,2, lambda x,y: x < y)
# Runtime error
bigger(1,2, lambda x: x)
```

false
true
→ 2 parameter function

Declaring Classes

Scala:

```scala
class Foo(val x: Int,
          var y: Double = 0.0)

// Type of a is inferred
val a = new Foo(1, 4.0)
println(a.x) //x is read-only
println(a.y) //y is read-write
a.y = 10.0
println(a.y) //y is read-write
a.y = "Foo"   // Type mismatch, y is double
```

read-only
read-write
default value
default constructor automatically created
1
4.0
10.0
double

Python:

```python
class Foo():
    def __init__(self, x, y):
        self.x = x
        self.y = y
a = Foo(3,2)
print a.x
a.x = "foo"
print a.x
```

constructor
attributes - read-write by default
3
no type enforcement
foo

Object-Oriented Programming

Scala:

```scala
class Foo(val x: Int,
          var y: Double = 0.0)

class Bar(x: Int, y: Int, z: Int)
  extends Foo(x, y)

trait Printable {
  val s: String
  def asString() : String
}

class Baz(x: Int, y: Double, private val z: Int)
  extends Foo(x, y) with Printable {
  override val s: String = s
  override def asString(): String = ???
}
```

inheritance
interface (include attributes)

Python:

```
class Foo():
  def __init__(self, x, y):
    self.x = x
    self.y = y

class Bar(Foo):
  def __init__(self, x, y, z):
    Foo.__init__(self, x, y)
    self.z = z
```

→ inheritance

→ super - parent constructor

Data Classes

Scala:

```
case class Address(street: String,
  number: Int)
case class Person(name: String,
  address: Address)
val p = Person("G", Address("a", 2))
```

→ attributes are read-only (val)

→ default object comparison method

Python:

```
from dataclasses import dataclass

@dataclass
class Address:
    street: str
    number: int
@dataclass
class Person:
    name: str
    addr: Address

p = Person("G", Address("a", 2))
```

→ no constructor needed

→ read-write fields

default equals method

Pattern matching in Scala:

```
// Code for demo only, won't compile
value match {
  // Match on a value, like if
  case 1 => "One"
  // Match on the contents of a list
  case x :: xs => "The remaining contents are " + xs
  // Match on a case class, extract values
  case Email(addr, title, _) => s"New email: $title..."
  // Match on the type
  case xs : List[_] => "This is a list"
  // With a pattern guard
  case xs : List[Int] if xs.head == 5 => "This is a list of integers"
  case _ => "This is the default case"
}
```