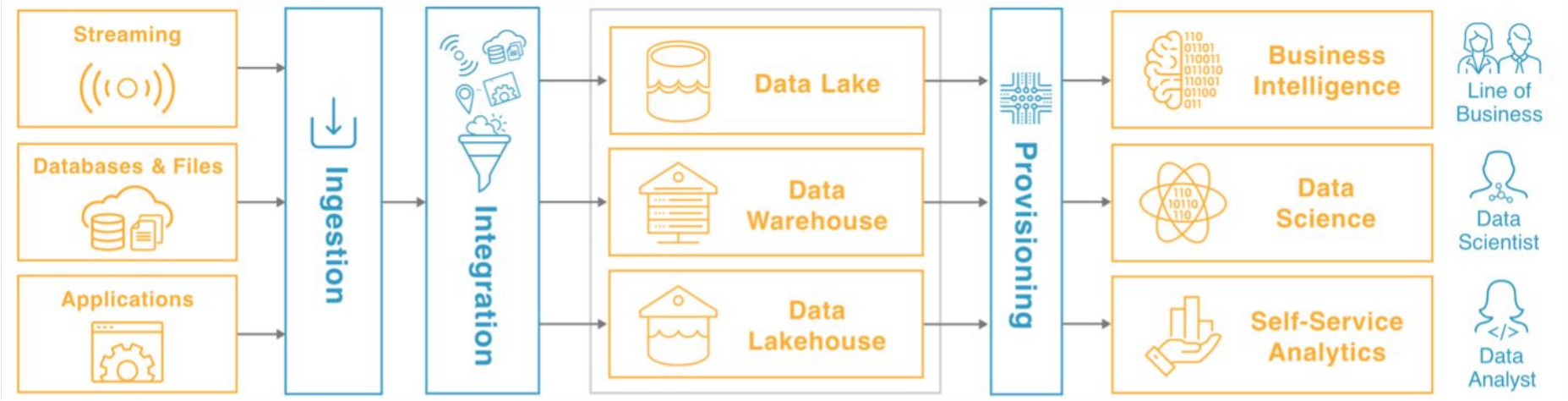
Spark Basics

Jorge Acosta Hernández
jorge.acosta@upm.es

With some slides from Jesús Montes

Nov. 2024


Streaming
Databases & Files
Applications
Ingestion
Integration
Data Lake
Data Warehouse
Data Lakehouse
Provisioning
Business Intelligence
Data Science
Self-Service Analytics
Line of Business
Data Scientist
Data Analyst

Spark

What is  ?

- Apache Spark™ is a **multi-language** engine for executing data engineering, data science, and machine learning on single-node machines or **clusters**
- Originally developed at the University of California, AMPLab. (2009)
- **Aimed to create a faster, more efficient and flexible data processing framework than Hadoop MapReduce**
- Donated to the Apache Foundation (2013)
- Most popular MapReduce successor
- Written in Scala, current stable version is 3.5.3 (released Sep 2024)

What is Apache Spark™ ?

- Distributed processing framework
- For processing batch data
- For processing stream data
- For OLAP
- Machine Learning
- Multiple Languages: Python, Scala, Java,R and SQL

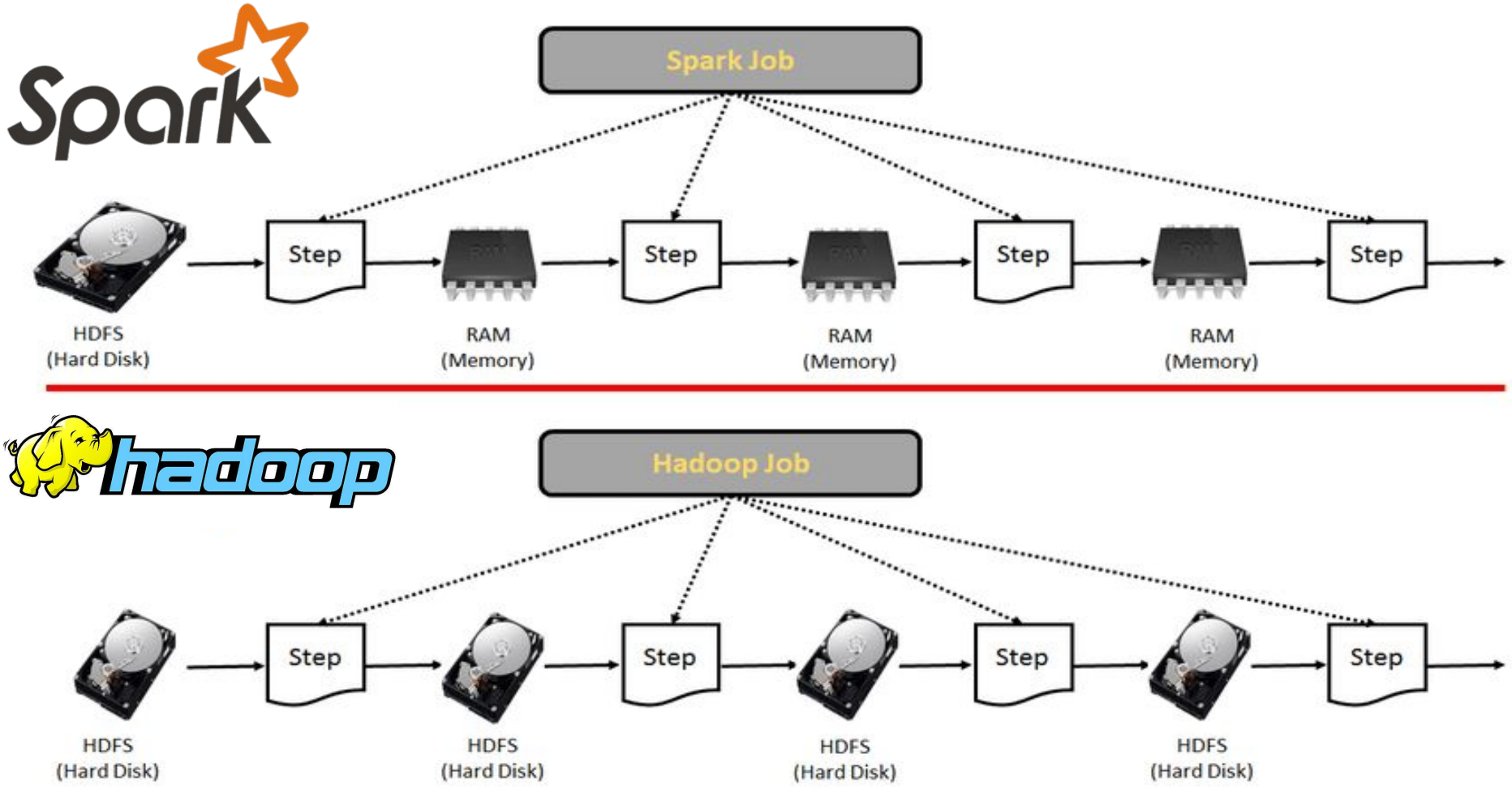
What isn't Apache Spark™ ?

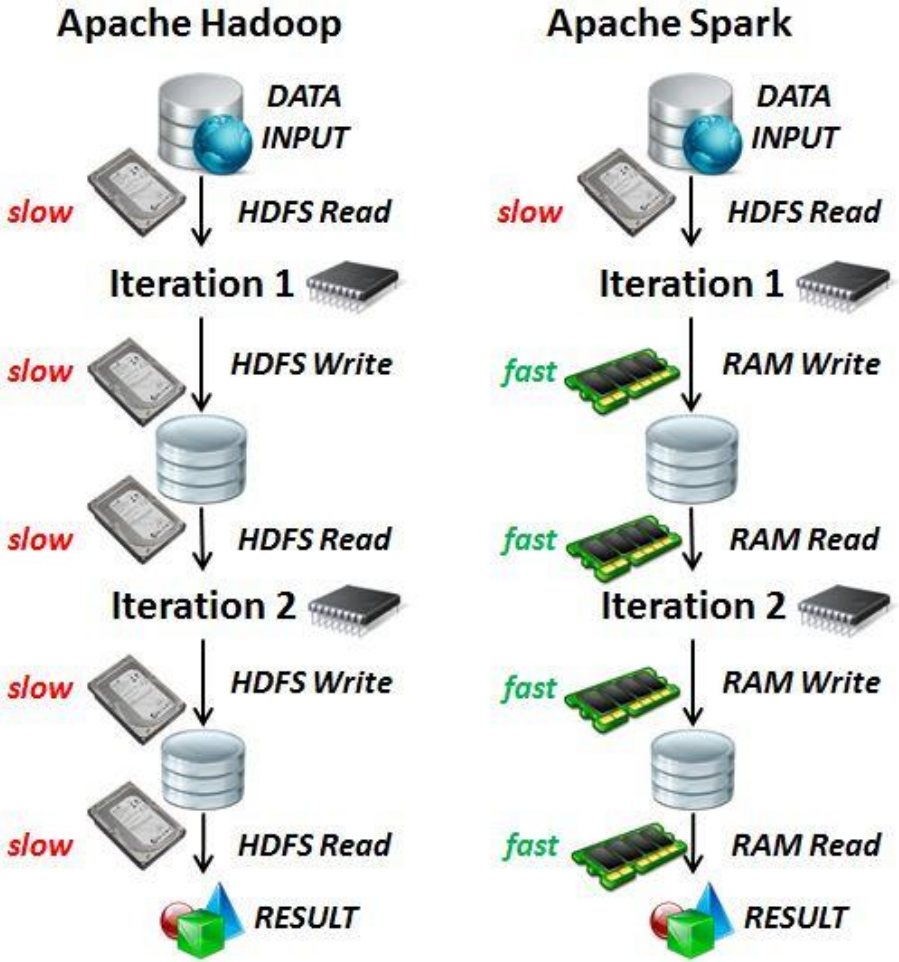
- Not a distributed database
- Not a distributed filesystem
- Not a database
- Doesn't have ACID properties
- There are some edge cases where it still cannot replace Hadoop :(

VS

- Written in Java
- Native filesystem (HDFS)
- Memory in disc
- Complex
- For OLTP
- Doesn’t include ML libraries
- Scalable, discs are cheap

- Written in Scala
- Not native filesystem
- Memory in RAM
- Not so complex
- For OLAP
- Includes ML libraries
- Scalable, but RAMs aren’t cheap


Spark Job
Step
Step
Step
Step
HDFS
(Hard Disk)
RAM
(Memory)
RAM
(Memory)
RAM
(Memory)
Hadoop Job
Step
Step
Step
Step
HDFS
(Hard Disk)
HDFS
(Hard Disk)
HDFS
(Hard Disk)
HDFS
(Hard Disk)


Apache Hadoop
DATA INPUT
slow
HDFS Read
Iteration 1
slow
HDFS Write
slow
HDFS Read
Iteration 2
slow
HDFS Write
slow
HDFS Read
RESULT
Apache Spark
DATA INPUT
slow
HDFS Read
Iteration 1
fast
RAM Write
fast
RAM Read
Iteration 2
fast
RAM Write
fast
RAM Read
RESULT

execution modes

Spark can work in 2 modes:

- Local Mode (e.g. your own computer)
- Cluster Mode:
  - Standalone (Spark deploys its own computing cluster by a configuration defined by the user).
  - On top of cluster managers this cluster managers Yarn, Kubernetes and Mesos.

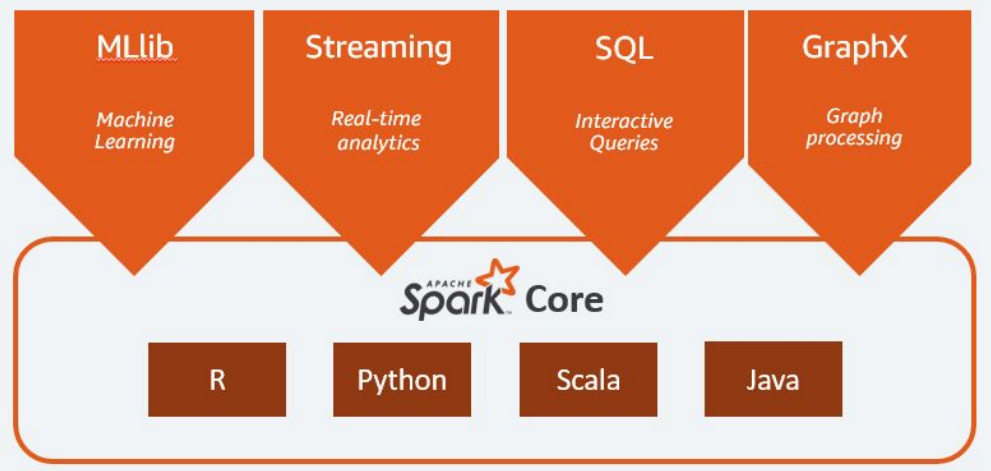

is flexible!

- Spark can read/write data from many local storage technologies:
  - Local filesystem
  - HDFS
  - Hive
  - Kudu
  - Cassandra
  - Delta Lake
  - HBase
  - MongoDB
  - ...

- It can also read/write data from cloud storage:
  - AWS S3
  - Azure Blob Storage
  - SnowFlake
  - Google Cloud Storage
  - OpenStack Swift
  - ...

Components

- **Spark Core**: The base component
- **Spark SQL**: High-level structured-data processing
- **MLlib**: Machine learning
- **GraphX**: Graph processing
- **Spark Streaming**: Stream processing
- **Spark Structured Streaming**: Better Stream processing



Spark Core

It contains the basic functionality of Spark

- Task scheduling (DAG)
- Memory management
- Fault recovery
- Storage-layer interaction
- Cluster resource manager interaction (YARN/Kubernetes/Mesos)
- *Resilient Distributed Datasets* (RDDs) API

The Spark Core makes possible to create and manage Spark Applications.

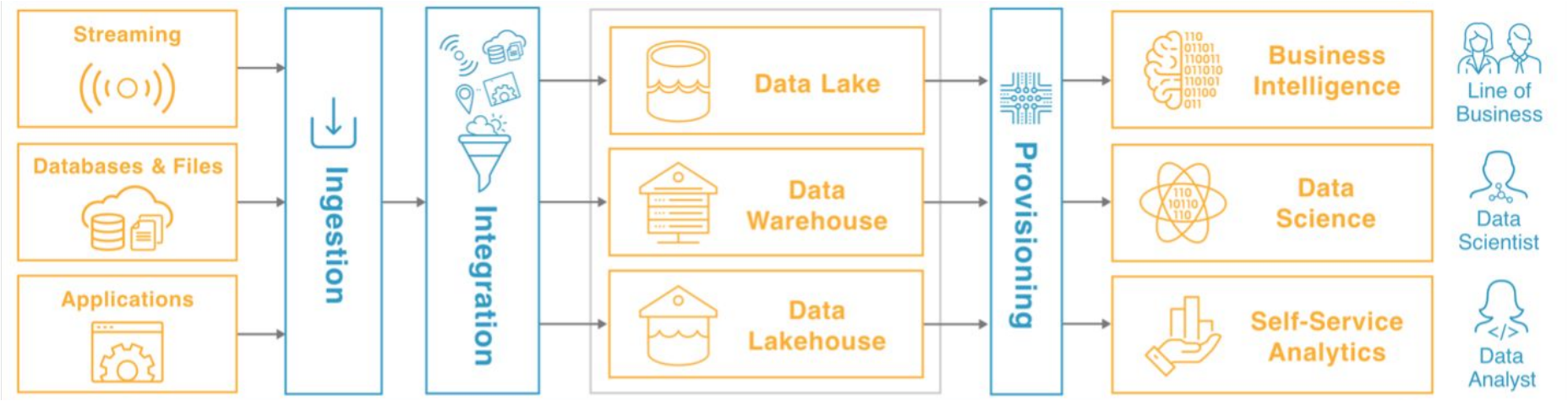
Summarizing

Spark's authors claim it is fast...

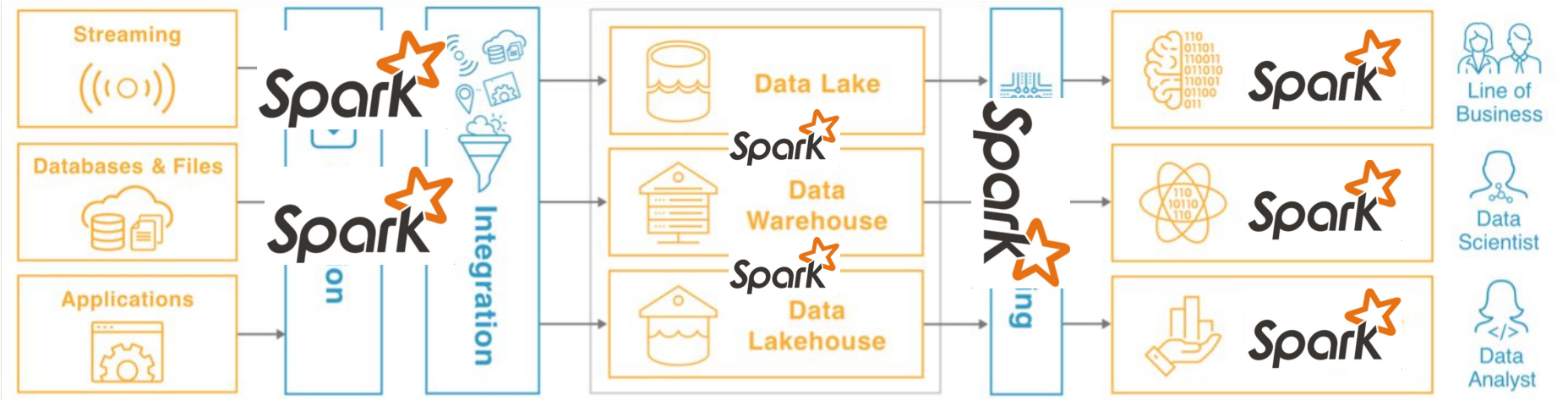
- Extends the MapReduce model
- Provides tools for OLAP
- Takes advantage of in-memory distributed computation

... and general purpose

- Different types of workloads: batch, interactive, iterative, streaming
- Multiple APIs: Scala, Java, Python, SQL and R
- Compatible with many cluster managers and cloud/local filesystems


Streaming
Databases & Files
Applications
Ingestion
Integration
Data Lake
Data Warehouse
Data Lakehouse
Provisioning
Business Intelligence
Data Science
Self-Service Analytics
Line of Business
Data Scientist
Data Analyst


Spark


Streaming
Databases & Files
Applications
Spark
Spark
on
Integration
Data Lake
Spark
Data
Warehouse
Spark
Data
Lakehouse
Spark
ng
Spark
Spark
Spark
Line of
Business
Data
Scientist
Data
Analyst

Data Variety

- Unstructured Data:
- Semi-Structured Data:
- Structured Data:

Data Variety

- Unstructured Data:
  - Images
  - Audio
  - Text files
  - Video
- Semi-Structured Data:
  - E-mail
  - JSON
  - CSV
  - Logs
- Structured Data:
  - SQL
  - CRM Data
  - Structured CSV
  - Transactions

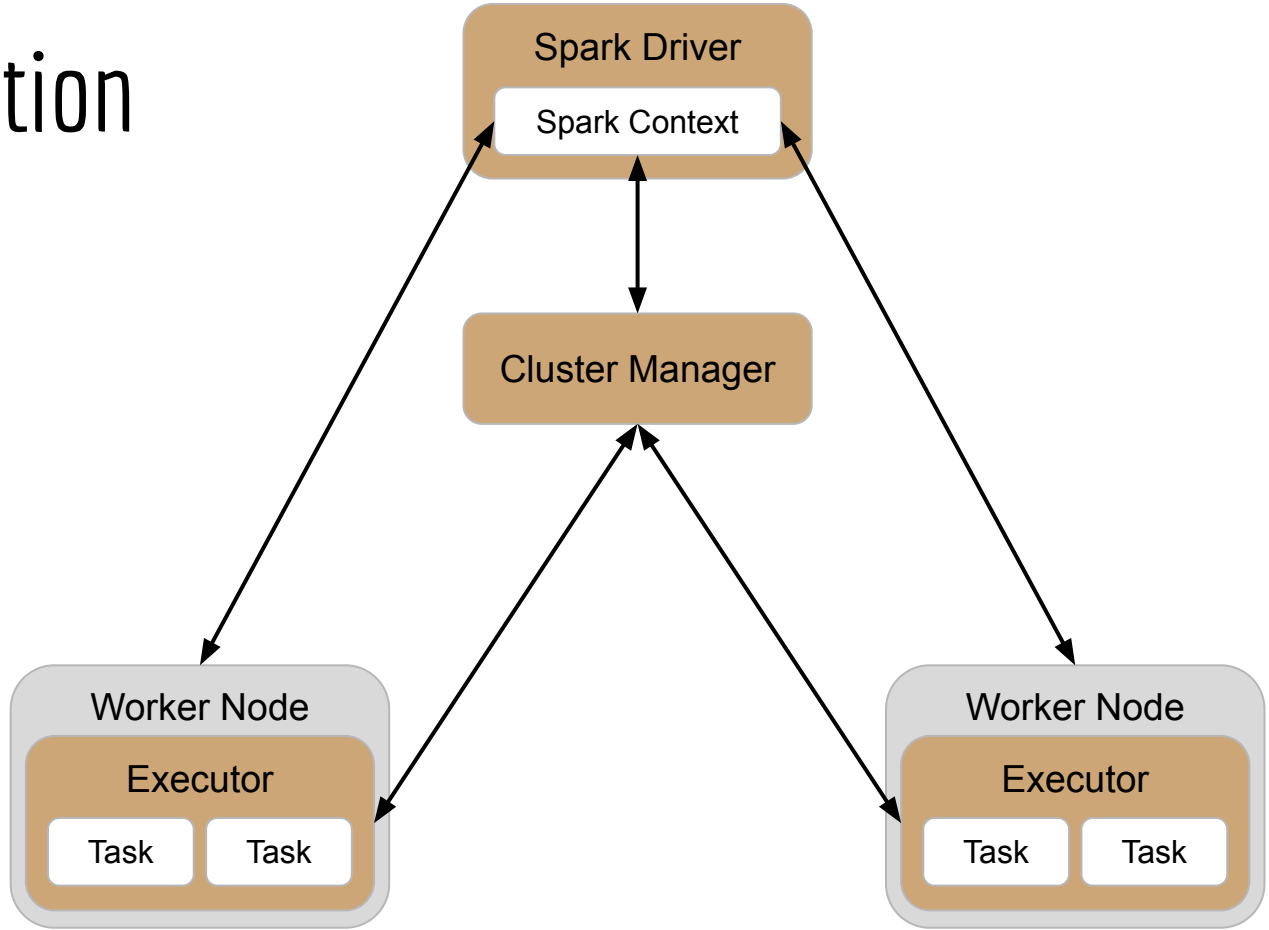
Data Variety

- Unstructured Data:
  - Images
  - Audio
  - Text files
  - Video
- Semi-Structured Data:
  - E-mail
  - JSON
  - CSV
  - Logs
- Structured Data:
  - SQL
  - CRM Data
  - Structured CSV
  - Transactions

RDDs

Dataframes

Spark Application



Spark Application

All Spark applications have the same components

- **Driver:** This is the central component that manages application execution. It contains the SparkContext, which configures and manages execution parameters and resource allocation.
- **Cluster manager**: Is the component in charge of organizing the application execution, either directly (standalone) or through an external application manager (YARN/Kubernetes/Mesos).
- **Executors**: They perform the data access and computation in the worker nodes.

Spark Context

A Spark Context represents a connection to a computing cluster:

- Master node URL
- Application name and other application parameters
- Driver resources (jar and/or py files)

Using a Spark Context, the application can create Spark objects, like RDDs, accumulators and broadcast variables.

A Spark Context is usually represented by an object of the *SparkContext* class, and it can be configured using the *SparkConf* class.

Resilient Distributed Dataset (RDD)

An RDD is a distributed collection of items. Is a fundamental data structure in Spark.

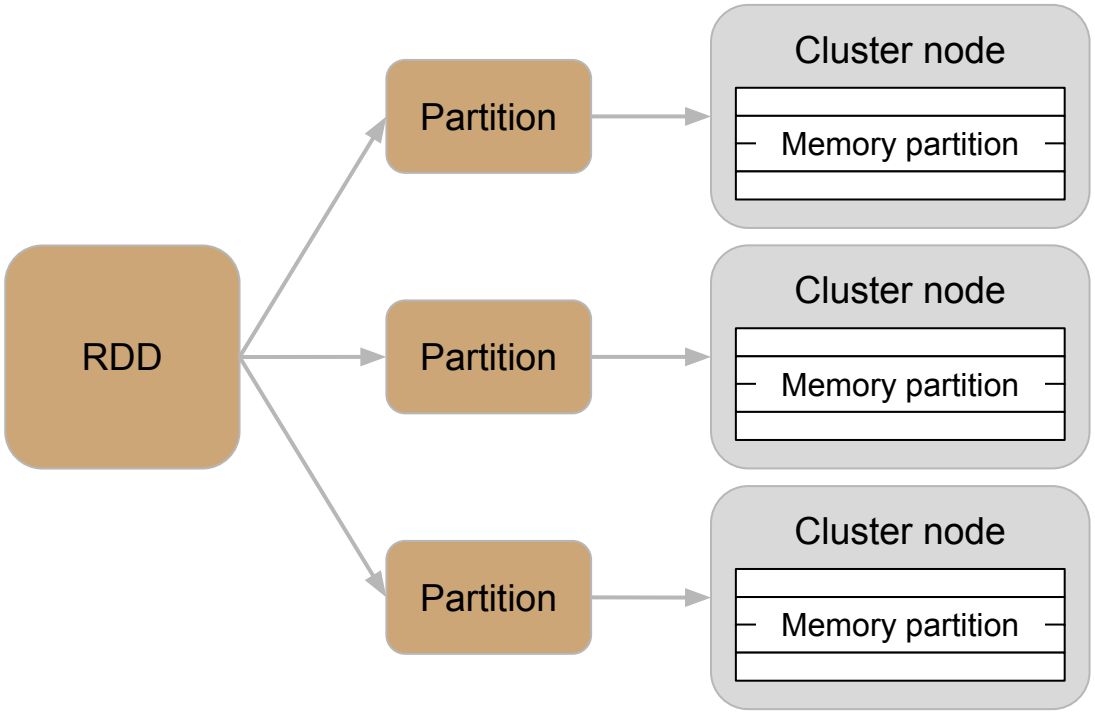
- RDDs can be created...
  - ... from local application objects (parallelize),
  - ... from local/cloud InputFormats (such as HDFS files)
  - ... or by transforming other RDDs.
- RDDs have...
  - ...transformations, which return pointers to new RDDs...
  - ... actions, which return values

- RDDs...
  - ... are Immutable,
  - ... are Lazy-evaluated,
  - ... are Fault-tolerant
  - ... and provide Persistency and Partitioning Control.

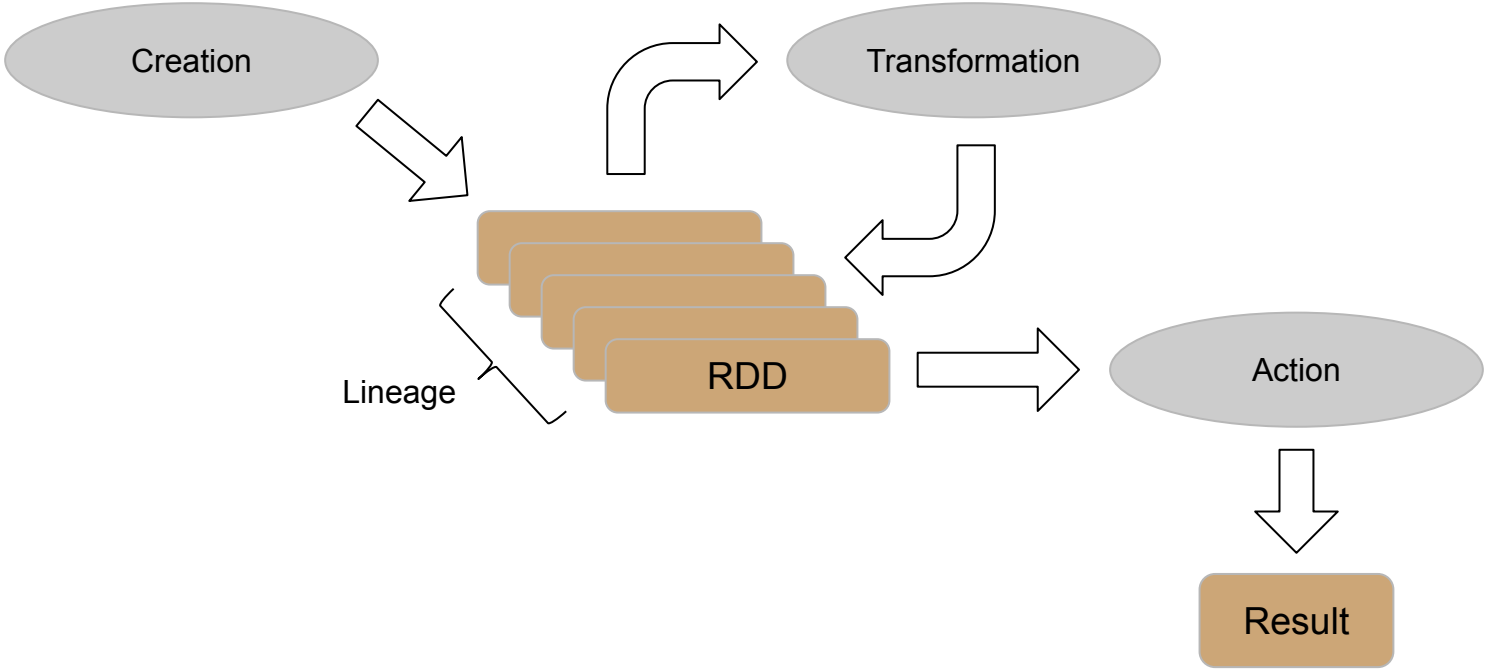
In principle, an RDD can contain any type of object. **In practice, only objects that can be serialized can be used.** Otherwise the RDD could not be distributed throughout the cluster.

Resilient Distributed Dataset (RDD)

- RDDs are managed by *RDD* objects.
- An RDD object contains references to *Partition* objects.
- Each *Partition* object corresponds to a subset of data.
- Partitions are assigned to nodes in the cluster.
- By default, each partition will be stored in memory (RAM).



Transformations and Actions



Transformations

- RDDs are immutable, so the only way to transform its content is by generating a new RDD.
- Transformations are not executed when declared. Instead, they are registered by the Spark Driver and organized into an execution plan.
- Examples of transformations: ?

Transformations

- RDDs are immutable, so the only way to transform its content is by generating a new RDD.
- Transformations are not executed when declared. Instead, they are registered by the Spark Driver and organized into an execution plan.
- Examples of transformations: filter the values of an RDD, group values by some criteria.

Actions

- Actions compute a result based on the contents of an RDD. The result can be returned to the Driver or stored (in a local file, HDFS, etc.).
- When an action is requested, the Driver performs the execution plan, computing the target RDD contents and applying the indicated action.
- Examples of actions: ?

Actions

- Actions compute a result based on the contents of an RDD. The result can be returned to the Driver or stored (in a local file, HDFS, etc.).
- When an action is requested, the Driver performs the execution plan, computing the target RDD contents and applying the indicated action.
- Examples of actions: output the RDD contents (totally or partially), count the elements in an RDD.

Spark APIs

- Scala
  - The most complete one (**after all, Spark is written in Scala**)
  - The default language of the Spark Shell
- Java and Python
  - Almost as complete as the Scala API
  - Some minor limitations/difficulties, but perfectly functional
- R
  - Only available at a DataFrame level (no direct access to RDDs)
  - Limited, but functional for basic/medium-complexity operations
  - Greatly improved in Spark 3.x
- SQL
  - Execute SQL queries

The Spark Shell

Spark can be used interactively through the Spark Shell

- Is a typical REPL (read-eval-print loop) application.
- Is launched with the *spark-shell* script located inside the *bin* directory of the Spark installation.
- The default Spark Shell is, in fact, a standard Scala interpreter with several tweaks (pre-loaded libraries, etc.).
- It contains an Spark Context already initialized (in the *sc* object).
- Spark also provides shell versions for Python (*pyspark*), R (*sparkR*) and SQL (*spark-sql*).

Let's try PySpark

1. First, start the PySpark Shell
   `# pyspark`
   This will start the Python interpreter and initialize a Spark Context, it will be preloaded in a variable called **sc**. After a short while, you will see a python prompt (>>>)
2. Type the example code on the right in the shell. Make sure to run one line at a time, and check the results of each one of them.

```
nums = list(range(1,51))
oneRDD = sc.parallelize(nums)
oneRDD.count()

otherRDD = oneRDD.map(lambda x: x *
2)
result = otherRDD.collect()
for v in result:
    print(v)

sum = oneRDD.reduce(lambda x,y: x+y)
print(sum)
```

Creating/Loading/Saving RDDs

- Parallelize an existing collection

```
myRDD1 = sc.parallelize(['a', 'b', 'c'])
```

- Load from file

```
myRDD2 = sc.textFile("file:///tmp/textfile.txt")
myRDD3 = sc.textFile("hdfs:///tmp/book/textfile.txt")
```

Only in `local` mode

Only if HDFS is available

- Save to file

```
oneRDD.saveAsTextFile("file:///tmp/oneRDD")
otherRDD.saveAsObjectFile("hdfs:///tmp/otherRDD")
```

Only in `local` mode

Only if HDFS is available

- Different sources, destinations and data formats are supported.

RDD transformations

- Most RDD transformations are based on passing functions as parameters of RDD method calls.
- The result of these method calls is a new RDD object, containing the results of the requested transformation.
- Scala's powerful anonymous **function syntax** makes very easy to write concise code for Spark (**lambda functions in python**).
- Most part of RDD transformations are based on basic principles of **functional programming** and the MapReduce model.

RDD transformations

The most basic RDD transformation is *map*:

```
map[U](f: (T) ⇒ U): RDD[U]
```

It operates over an RDD of objects of class T. Its only argument is a function that takes an argument of class T and produces an output object of class U. The result is a new RDD of objects of class U, obtained by applying the function to each element of the input RDD.

Example:

```
myRDD = sc.parallelize([1,2,3])
myMappedRDD = myRDD.map( lambda x : 2 * x )
result = myMappedRDD.collect()
```

RDD transformations

- `map[U](f: (T) ⇒ U): RDD[U]`
- `flatMap[U](f: (T) ⇒ TraversableOnce[U]): RDD[U]`
  Similar to a regular *map*, but the parameter function can return zero or more elements.

RDD transformations

- `filter(f: (T) ⇒ Boolean): RDD[T]`
  Return a new RDD containing only the elements that satisfy a predicate.
- `distinct(): RDD[T]`
  Return a new RDD containing the distinct elements in this RDD.
- `sample(Replacement: Boolean, fraction: Double): RDD[T]`
  Returns a sampled subset of this RDD.

RDD transformations

- `sortBy[K](f: (T) ⇒ K, ascending: Boolean = true): RDD[T]`
  Return this RDD sorted by the given key function.
- `union(other: RDD[T]): RDD[T]`
  Return the union of this RDD and another one.
- `intersection(other: RDD[T]): RDD[T]`
  Return the intersection of this RDD and another one.
- `subtract(other: RDD[T]): RDD[T]`
  Return an RDD with the elements from this that are not in other.
- For more visit: spark-core-docs

Pair RDDs

- In Python/Scala, tuples are part of the language basic syntax, and RDDs of tuples can be created in Spark.
  - `RDD = [(K1,V1),(K1,V2),(K2,V1),(K2,V3)]`
  - `Keys = [K1,K2]`
  - `Values = [V1,V2,V3]`
- The Spark API provides a specific set of transformations and actions for working with key-value RDDs.
- Example:

```
myKVArray = [(1, 'a'), (1, 'b'), (2, 'a'), (2, 'c')]
myPairRDD = sc.parallelize(myKVArray)
```

Pair RDD transformations

- `keys: RDD[K]`
- `values: RDD[V]`
- `groupByKey(numPartitions: Int): RDD[(K, Iterable[V])]`
  Group the values for each key in the RDD into a single sequence.
- `reduceByKey(func: (V, V) ⇒ V): RDD[(K, V)]`
  Merge the values for each key using an associative and commutative reduce function.
- `join[W](other: RDD[(K, W)]): RDD[(K, (V, W))]`
  Return an RDD containing all pairs of elements with matching keys.
- For more visit: spark-core-docs

RDD actions

Basic RDD actions:

- `take(num: Int): Array[T]`
  Take the first num elements of the RDD.
- `collect(): Array[T]`
  Return an array that contains all of the elements in this RDD.
- `count(): Long`
  Return the number of elements in the RDD.
- `reduce(f: (T, T) ⇒ T): T`
  Reduces the elements of this RDD using the specified commutative and associative binary operator.
- `foreach(f: (T) ⇒ Unit): Unit`
  Applies a function f to all elements of this RDD.
- For more visit: spark-core-docs

Pair RDD actions:

- `collectAsMap(): Map[K, V]`
  Return the key-value pairs in this RDD to the master as a Map.
- `lookup(key: K): Seq[V]`
  Return the list of values in the RDD for key key.
- For more visit: spark-core-docs

Word Count with Spark RDDs

Start PySpark and type the following code:

```
bookRDD = sc.textFile("file:///tmp/books/some_book.txt")
wordsRDD = bookRDD.flatMap(lambda x: x.split(' ')) \
                  .filter(lambda x: x != "")

    wordPairRDD = wordsRDD.map(lambda x: (x,1))
countedWordsRDD = wordPairRDD.reduceByKey(lambda x,y : x+y)
countedWordsRDD = countedWordsRDD \
                            .sortBy(lambda x: x[1], ascending=False)
countedWordsRDD.saveAsTextFile("file:///tmp/some_book_wc_output")
```