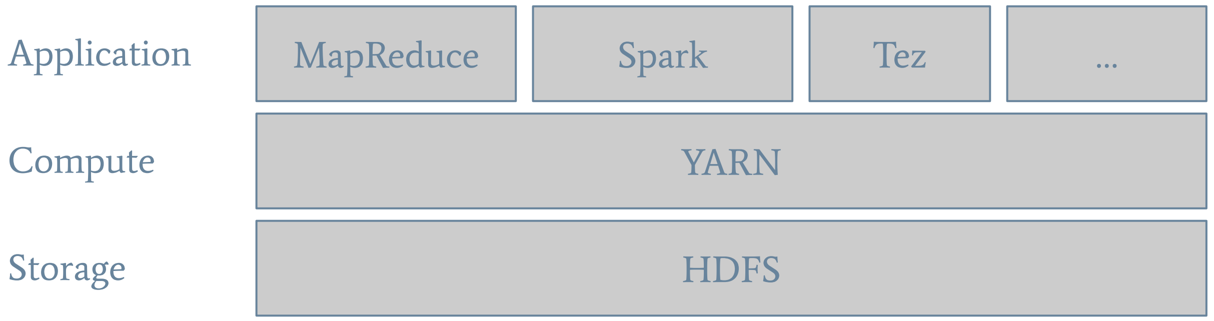
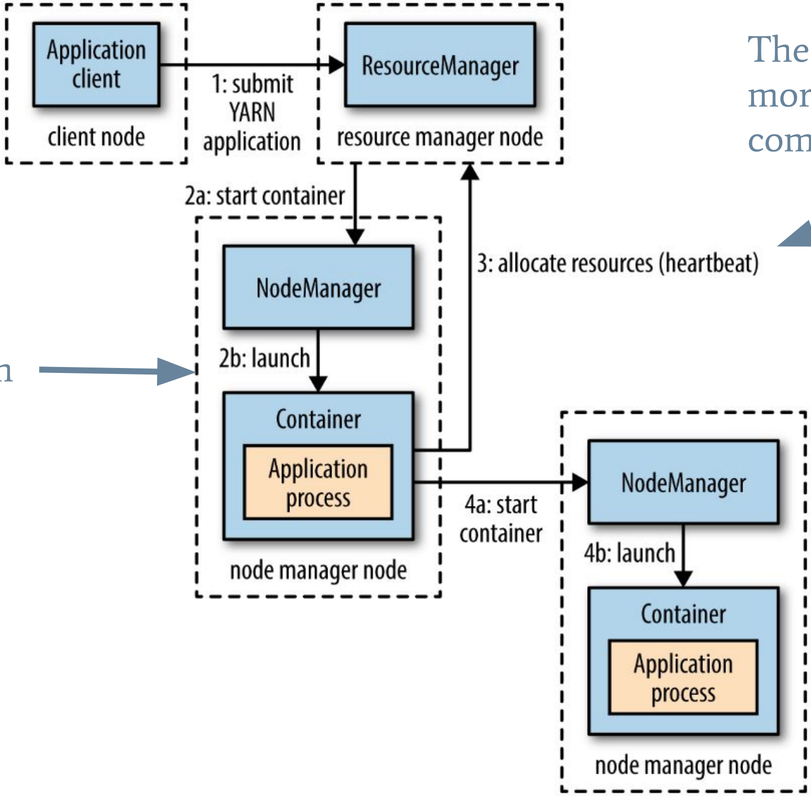
# Apache Spark

# YARN - Yet Another Resource Negotiator

- Last time, we mentioned that the MapReduce tasks were executed on the cluster?
  - Who is in charge of the resources allocation?
  - YARN!
- YARN
  - Manage the resources on the cluster
  - We can request resources with certain constraints (memory, specific machine, data location, ...)
  - In practice, we interact with higher level applications



## How YARN runs an application

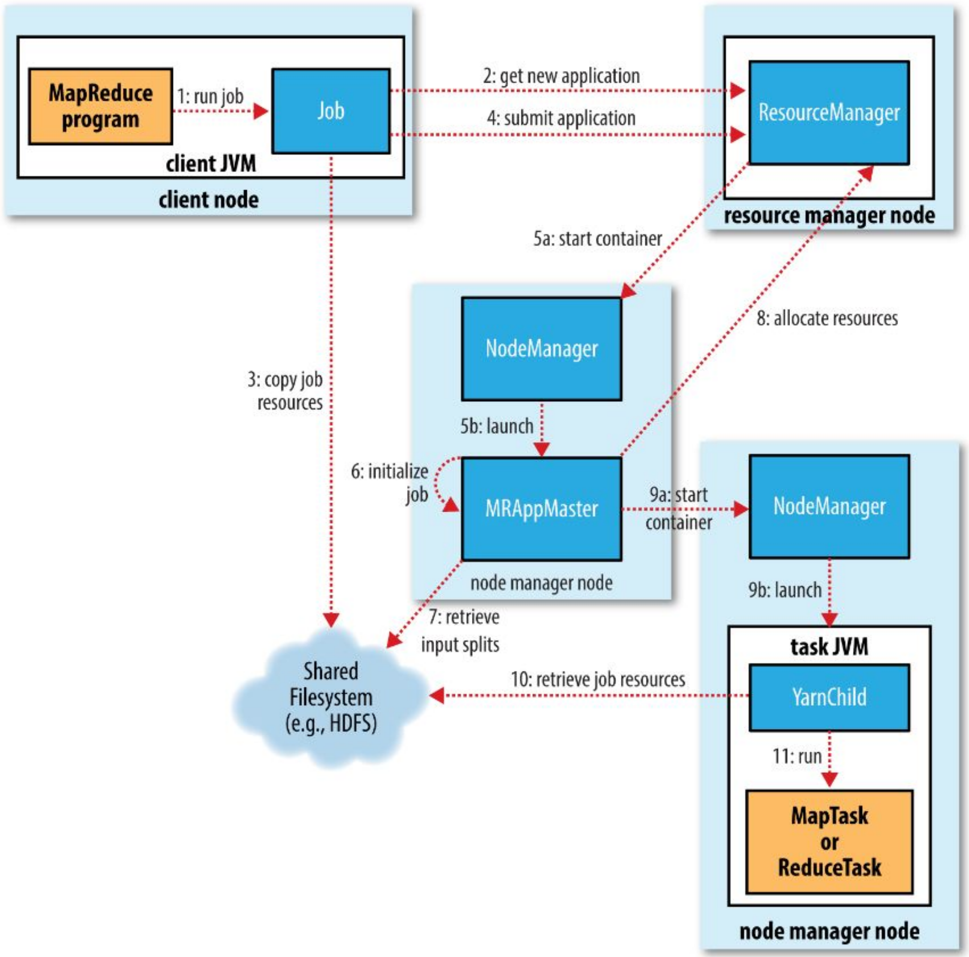


The main application can ask for more resources to make the computations

Main Application

From *Hadoop - The Definitive Guide*

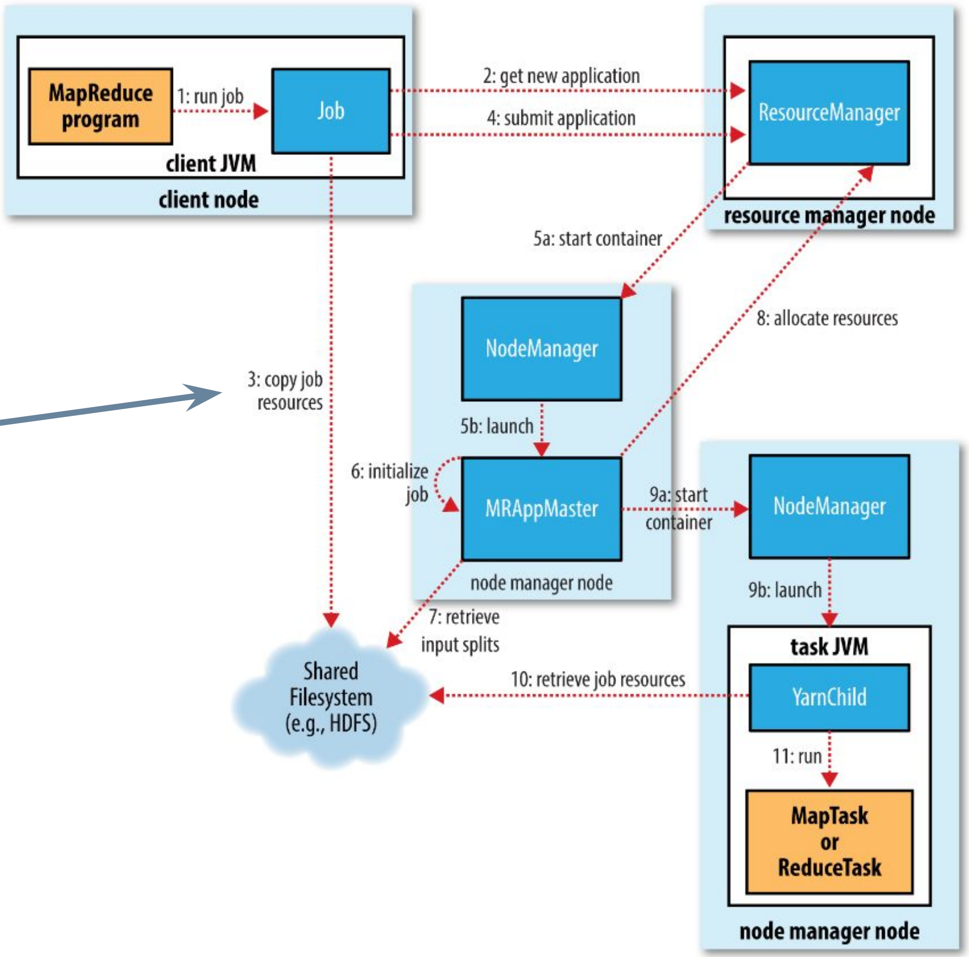
# MapReduce Example



From *Hadoop - The Definitive Guide*

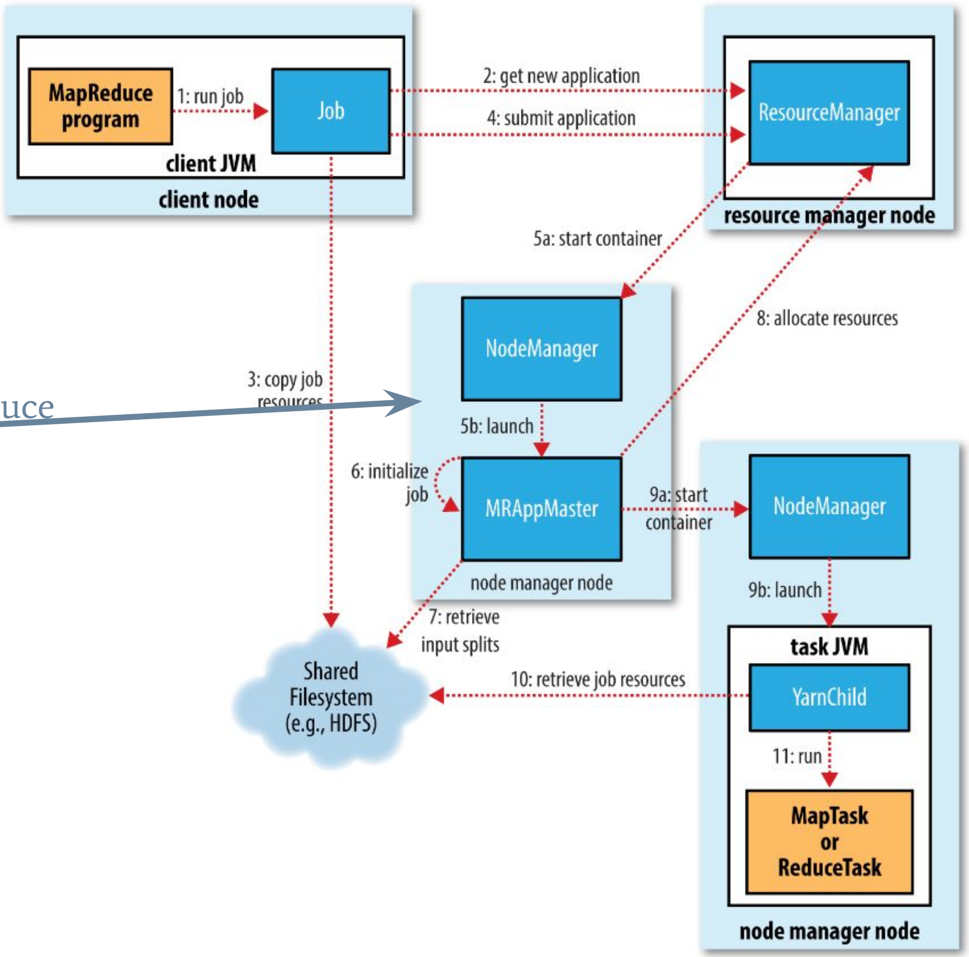
# MapReduce Example

Resources = JAR, configuration, splits, …



From *Hadoop - The Definitive Guide*

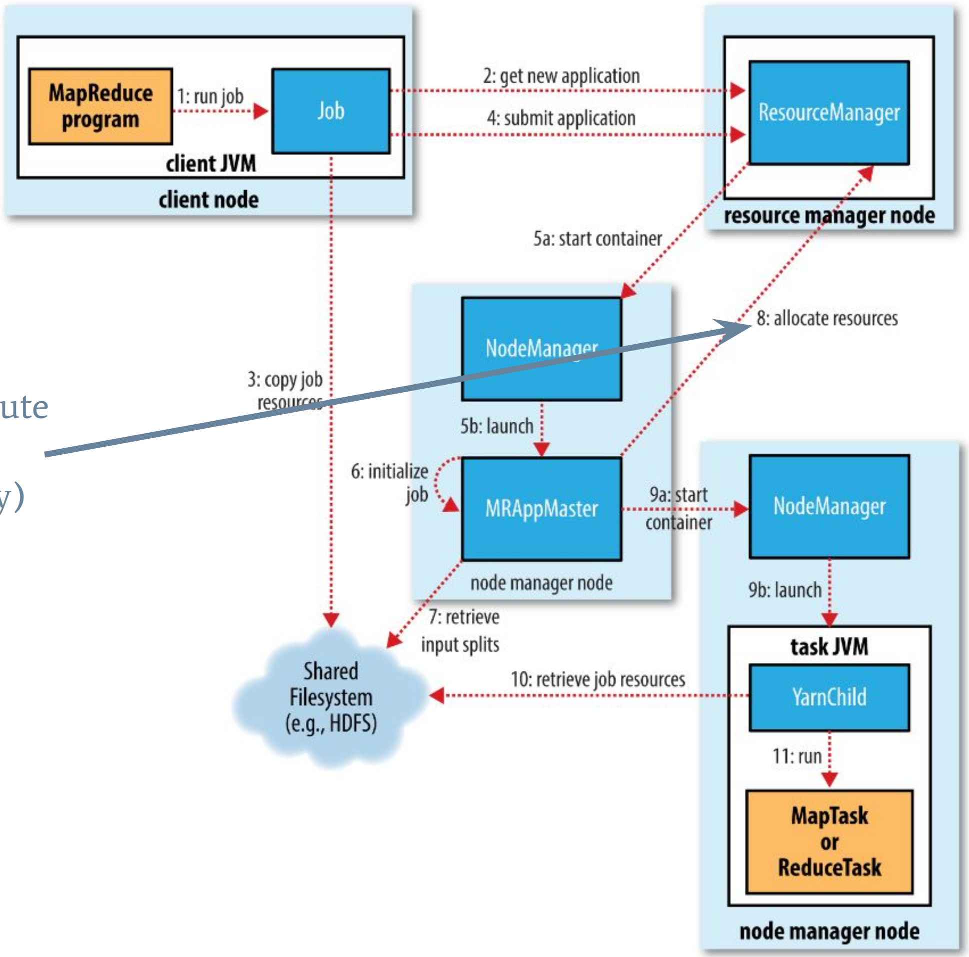
# MapReduce Example



MRAppMaster = MapReduce Manager

From *Hadoop - The Definitive Guide*

# MapReduce Example

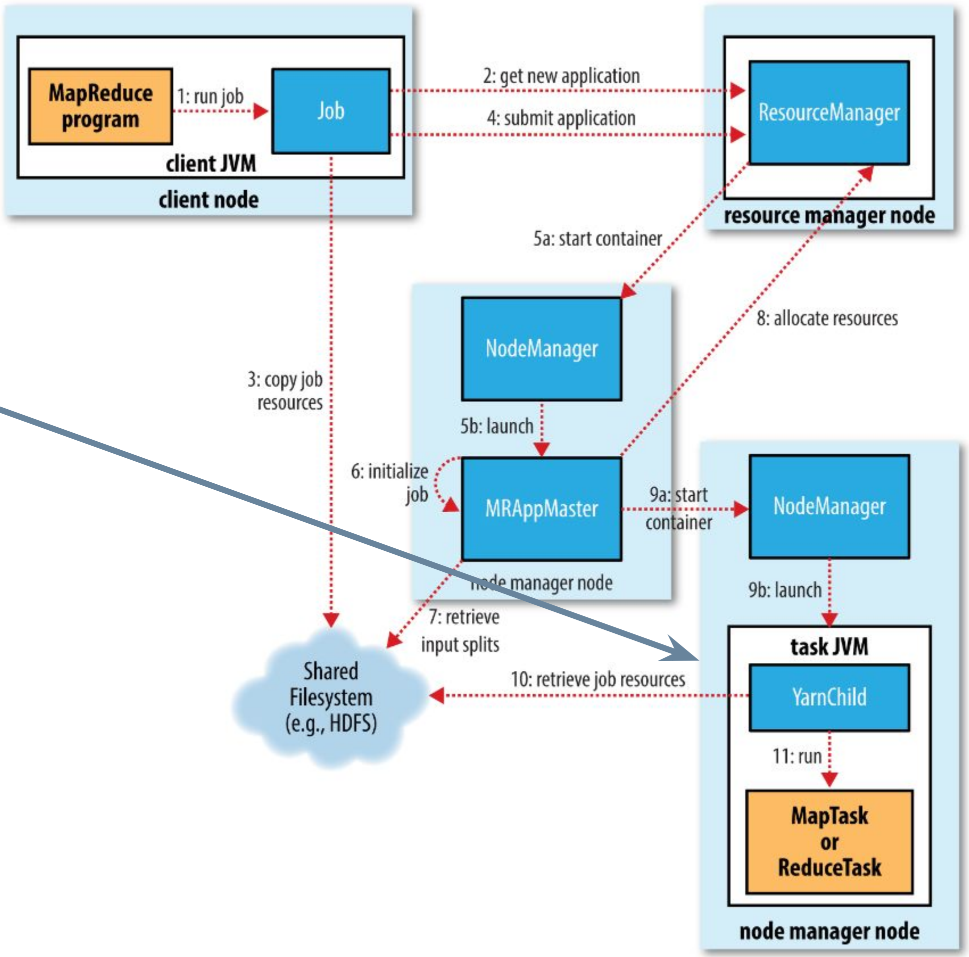


Asks for resources to execute map and reduce tasks (constraints = data locality)

From *Hadoop - The Definitive Guide*

# MapReduce Example

Map or reduce task



From *Hadoop - The Definitive Guide*

## MapReduce Is Not The Only Solution For Our Computations…

- Can become hard to program
- Only batch processing, no streaming
- No interactive mode (a shell) for fast prototyping
- Does not leverage RAM. Most computations are saved on disk/HDFS
- No data modification
- Linear processing = we see the data once

## Introducing Spark!

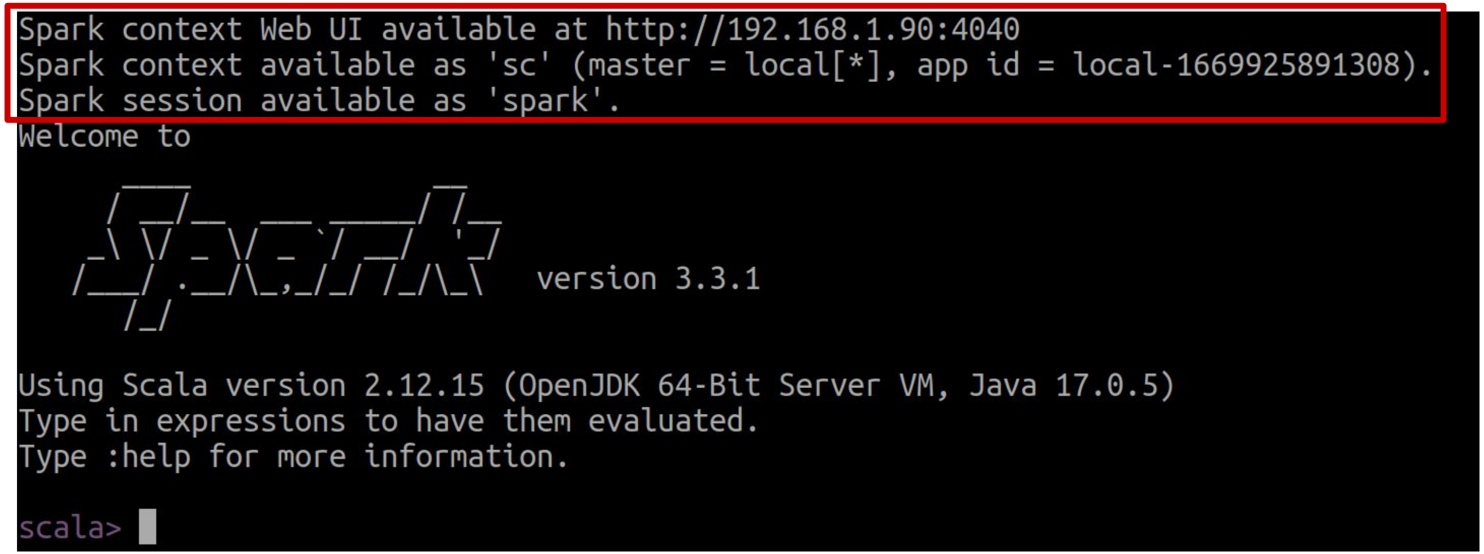
- Spark is a cluster computing framework
- Not based (only) on MapReduce
- Can run on YARN and use the HDFS
- Three languages: Scala (native language), Java, Python
- Comes with many additional modules

# Introducing Spark!

- Easy to program
- Streaming processing, batch processing, (near) real-time processing
- Interactive mode
- Smart usage of the RAM (10-100 times faster than MapReduce)
- Real-time data modification
- Iterative processing
- Nice tools for machine learning

Applications: customer segmentation, risk management, real-time fraud detection, big data analysis, ...

# Spark Interactive Mode



```
Spark context Web UI available at http://192.168.1.90:4040
Spark context available as 'sc' (master = local[*], app id = local-1669925891308).
Spark session available as 'spark'.
Welcome to
      ____              __
     / __/__  ___ _____/ /__
    _\ \/ _ \/ _ `/ __/  '_/
   /___/ .__/\_,_/_/ /_/\_\   version 3.3.1
      /_/

Using Scala version 2.12.15 (OpenJDK 64-Bit Server VM, Java 17.0.5)
Type in expressions to have them evaluated.
Type :help for more information.

scala>
```

## Spark Data Structure: RDD

- RDD = Resilient Distributed Datasets
- Core data structure in Spark
- RDD = immutable + partitioned data + distributed
- Low-level data - Not used a lot in practice
  - Used when we need optimizations
  - We need to reimplement many things
- More formally, an RDD is composed of:
  - A list of partitions
  - A function to split the data (how to split the input data)
  - Dependencies on other RDDs
  - (Optional) A partitioner for key-value RDD (which split goes in which partition)
  - (Optional) Preferences for computations

## Spark Data Structure: RDD - Creation

- From local variable

```
scala> val x = List(1, 2, 3, 4)
x: List[Int] = List(1, 2, 3, 4)
scala> sc.parallelize(x, 3)
res3: org.apache.spark.rdd.RDD[ Int] = ParallelCollectionRDD[11]…
```

- From a text file

```
scala> val textFile = sc.textFile("peter_pan.txt")
textFile: org.apache.spark.rdd.RDD[String] = …
```

- From another Spark structure, using `.rdd`

## Spark Data Structure: RDD - Access

- An RDD is potentially a large file distributed on several machine. Try to limit the explicit reads that could be expensive.
- Get all the data: `myRDD.collect()`
- Get the size: `myRDD.count()`
- Get the first element: `myRDD.first`
- Get the `k` first elements: `myRDD.take(k)`
- Save an RDD: `myRDD.saveAsTextFile(path)`

## Spark Data Structure: DataFrame

- Distributed table-like collection with rows, and named and typed columns
- The type-checking happens at *runtime*
- Looks like Pandas dataframes

## Spark Data Structure: DataFrame - Creation

- From a raw text file
  - `val textFile = spark.read.textFile(myFile)`
- From a JSON file
  - `val df = spark.read.json(myFile)`
  - For multiple lines: `spark.read.option("multiLine", true).json(myFile)`
- From a CSV
  - `val df = spark.read.csv(myFile)`

Then, we can access the schema with:

`df.printSchema()` or `df.schema`

## Spark Datatypes

Spark comes with a set of predefined types. You can find them in:

- `org.apache.spark.sql.types._`
- E.g.: `IntegerType`, `StringType`, `ArrayType`, `StructType`, `StructField`

We can force the schema:

```
import org.apache.spark.sql.types.{StructField, StructType, LongType, StringType}
val mySchema = StructType(Array(
    StructField("Product", StringType, true),
    StructField("Price", LongType, false)
))
val df = spark.read.schema(mySchema).json(myFile)
```

## Spark Data Structure: DataFrame - Manual Creation

It is possible to create a DataFrame by hand

```
import org.apache.spark.sql.Row
import org.apache.spark.sql.types.{StructField, StructType, StringType, LongType}
val mySchema = new StructType(Array(
    new StructField("FirstName", StringType, true),
    new StructField("LastName", StringType, true),
    new StructField("BirthYear", LongType, false)))
val myRows = Seq(Row("Bob", null, 1945L))
val myRDD = sc.parallelize(myRows)
val myDf = spark.createDataFrame(myRDD, mySchema)
```

## Spark Data Structure: DataFrame - Access

- Most methods seen for RDD (except `saveAsTextFile`) exist for DataFrame
- We can always transform a DataFrame in an RDD: `df.rdd`
- `df.show(k)` prints the k first lines as a table
- Get a description of the dataframe: `df.describe().show()`
- Functions similar to SQL:
  - SELECT => `selectExpr:df.selectExpr("FirstName as name")`
  - WHERE => `where: df.where("FirstName = 'Bob'")`
  - `join, groupBy, orderBy, union, distinct, limit, …`
- Accessing a field in a Row:
  - Field given the index: `myRow.get(idx)`
  - Field given the name: `myRow.getAs[Type]("fieldName")`
    - `myRow.getAs[String]("FirstName")`

## Spark Data Structure: DataFrame - Modifications

- Always a copy!
- Add a column: `withColumn`
  - `myDf.withColumn("age", expr("2022 - BirthYear")).show()`
  - `expr` = Expression = Set of transformations
- Rename a column: `withColumnRename`
- Remove column: `drop`

## Spark Data Structure: Dataset

- Same as DataFrame, except the type checking happens at *compile time*
- Only in Scala and Java
- Slower than DataFrames

## Spark High-Order Functions

- When we apply a transformation function on an RDD/DataFrame/Dataset, the evaluation is lazy
  - The computation happens when we ask for the results (e.g., through collect or saveAsTextFile)
- We find similar functions to what we saw in Scala:
  - filter, map, fold, foreach, groupBy, map, reduce, zip, ...
  - flatMap = map + flatten
- If we have a collection of key/values (RDD[KeyType, ValueType]), we have additional functions, the aggregation functions:
  - countByKey, foldByKey, groupByKey, reduceByKey
  - Groups by key and then apply the function (fold, reduce, count, ...) on each group
  - We cannot modify the key
- With DataFrames, you can only do a `groupByKey` or use `.rdd` to get an RDD and all the aggregation functions

# MapReduce Equivalent in Spark

```
val input: RDD[(K1, V1)] = …
input.flatMap(mapper)  // Mapper (we need to merge the resulting lists)
      .groupByKey().sortByKey()  // Shuffle
      .map(reducer)              // Reduce
```

Or

```
input.flatMap(mapper)
      .reduceByKey(reducer)   // Here, the reducer cannot change the key
      .sortByKey()
```

## MapReduce Equivalent in Spark - Word Count Example

```
textFile.zipWithIndex.map(x => (x._2,x._1))
    .flatMap(_._2.split(" ").map(x => (x, 1)))
    .groupByKey().sortByKey()
    .map(x => (x._1, x._2.reduce(_ + _)))
```

Or

```
textFile.zipWithIndex.map(x => (x._2, x._1))
    .flatMap(_._2.split(" ").map(x => (x, 1)))
    .reduceByKey(_ + _)
    .sortByKey()
```

## Persistence

- It is possible to cache an RDD/DataFrame/Dataset to make future access faster
  - *myVar.cache()*
  - The variable is cached during the first run, and can only be accessed by the job that cached it
- Caching makes future executions faster
- Useful for interactive exploration and iterative algorithms.

## Shared Variables

Spark provides two mechanisms to share variables among tasks:

- *Broadcast variables*: a constant serialized and sent to all executors

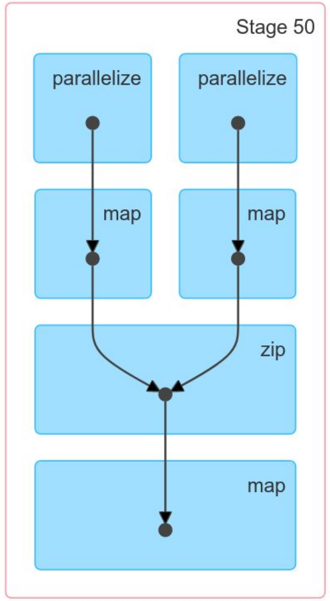
```
val lookup = sc.broadcast(Map(1 -> "a",
                    2 -> "e", 3 -> "i", 4 -> "o", 5 -> "u"))
val result = sc.parallelize(Array(2, 1, 3)).map(lookup.value(_))
result.collect()  // Array(e, a, i)
```

- *Accumulator*: Shared variable that can only be modified by a associative and commutative operation. `LongAccumulator` is defined by default, but we can create custom accumulators with `AccumulatorV2`

```
val acc = sc.longAccumulator
val result = sc.parallelize(Array(1, 2, 3))
          .map(i => {acc.add(i); i }).reduce(_ + _)
acc.value
```

## Spark Computation Model

- A computation is represented as a Directed Acyclic Graph (DAG) is Scala
- The DAG can be accessed through the WebUI



## Let’s go to the lab!