Large-Scale Data Mining and Machine Learning for Big Data Analytics

Apostolos N. Papadopoulos
Associate Professor
Data and Web Science Lab
Aristotle University of Thessalonki
http://datalab.csd.auth.gr

Outline

- What is Spark?

- Basic Features

- Resilient Distributed Datasets (RDDs) and DataFrames

- Existing Libraries

- Under the Hood

- Examples in Scala & Python

- Further Reading

FUNDAMENTAL CONCEPTS

What is Spark ?

Apache Spark is a **high-performance**, **general-purpose distributed computing system** that has become the most active Apache open source project, with more than 1,000 active contributors (*from High Performance Spark, O'Reilly, 2017*).

In brief, **Spark is a UNIFIED platform for cluster computing, enabling efficient big data management and analytics.**

It is an Apache Project and its current release is **2.4.4 (Sep 1, 2019)** **previous release 2.3.1** (**June 8, 2018**)

It is **one of the most active Apache projects**:

1.0.0 - May 30, 2014
1.0.1 - July 11, 2014
1.0.2 - August 5, 2014
1.1.0 - September 11, 2014
1.1.1 - November 26, 2014
1.2.0 - December 18, 2014
1.2.1 - February 9, 2014
1.3.0 - March 13, 2015
1.3.1 - April 17, 2015
1.4.0 - June 11, 2015
1.4.1 - July 15, 2015
1.5.0 - September 9, 2015
1.5.1 - October 2, 2015
1.5.2 - November 9, 2015
1.6.0 - January 4, 2016
1.6.1 - March 9, 2016
1.6.2 - June 25, 2016
2.0.0 - July 26, 2016
2.0.1 - October 3, 2016
2.0.2 - November 14, 2016
2.1.0 - December 28, 2016
2.1.1 - March 2, 2017
2.2.0 - Jully 11, 2017
2.2.1 - December 1, 2017
2.3.0 – February 28, 2018
2.3.2 – September 24, 2018
2.3.3 – February 15, 2019
2.4.0 – November, 2018
2.4.1 – March 31, 2019
2.4.3 – May 8, 2019

Who Invented Spark ?

Matei Zaharia

Born in Romania

University of Waterloo (B.Sc. Mathematics, Honors Computer Science)
Berkeley (Ph.D. cluster computing, big data)

Now: Assistant Professor @ CSAIL MIT

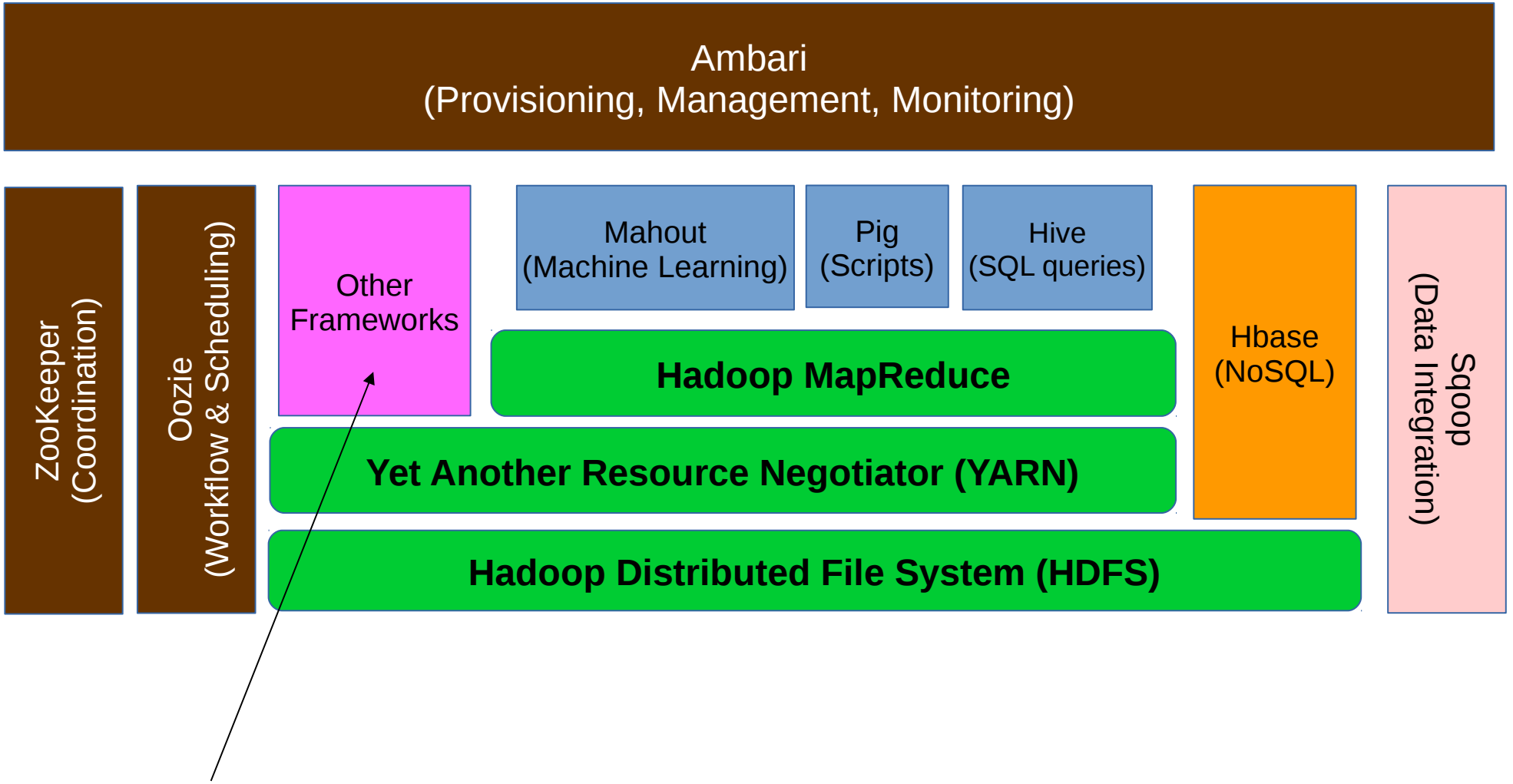
He also co-designed the MESOS cluster manager and he contributed to Hadoop fair scheduler.

Who Can Benefit from Spark ?

Spark is an excellent platform for:

- **Data Scientists**: Spark's collection of data-focused tools helps data scientists to go beyond problems that fit in a single machine
- **Engineers**: Application development in Spark is far more easy than other alternatives. Spark's unified approach eliminates the need to use many different special-purpose platforms for streaming, machine learning, and graph analytics.
- **Students**: The rich API provided by Spark makes it extremely easy to learn data analysis and program development in Java, Scala or Python.
- **Researchers**: New opportunities exist for designing distributed algorithms and testing their performance in clusters.

Spark in the Hadoop Ecosystem



Spark is somewhere here

Spark vs Hadoop MR: sorting 1PB

	Hadoop	Spark 100TB	Spark 1PB
Data Size	102.5 TB	100 TB	1000 TB
Elapsed Time	72 mins	23 mins	234 mins
# Nodes	2100	206	190
# Cores	50400	6592	6080
# Reducers	10,000	29,000	250,000
Rate	1.42 TB/min	4.27 TB/min	4.27 TB/min
Rate/node	0.67 GB/min	20.7 GB/min	22.5 GB/min

Source: Databricks

Spark Basics

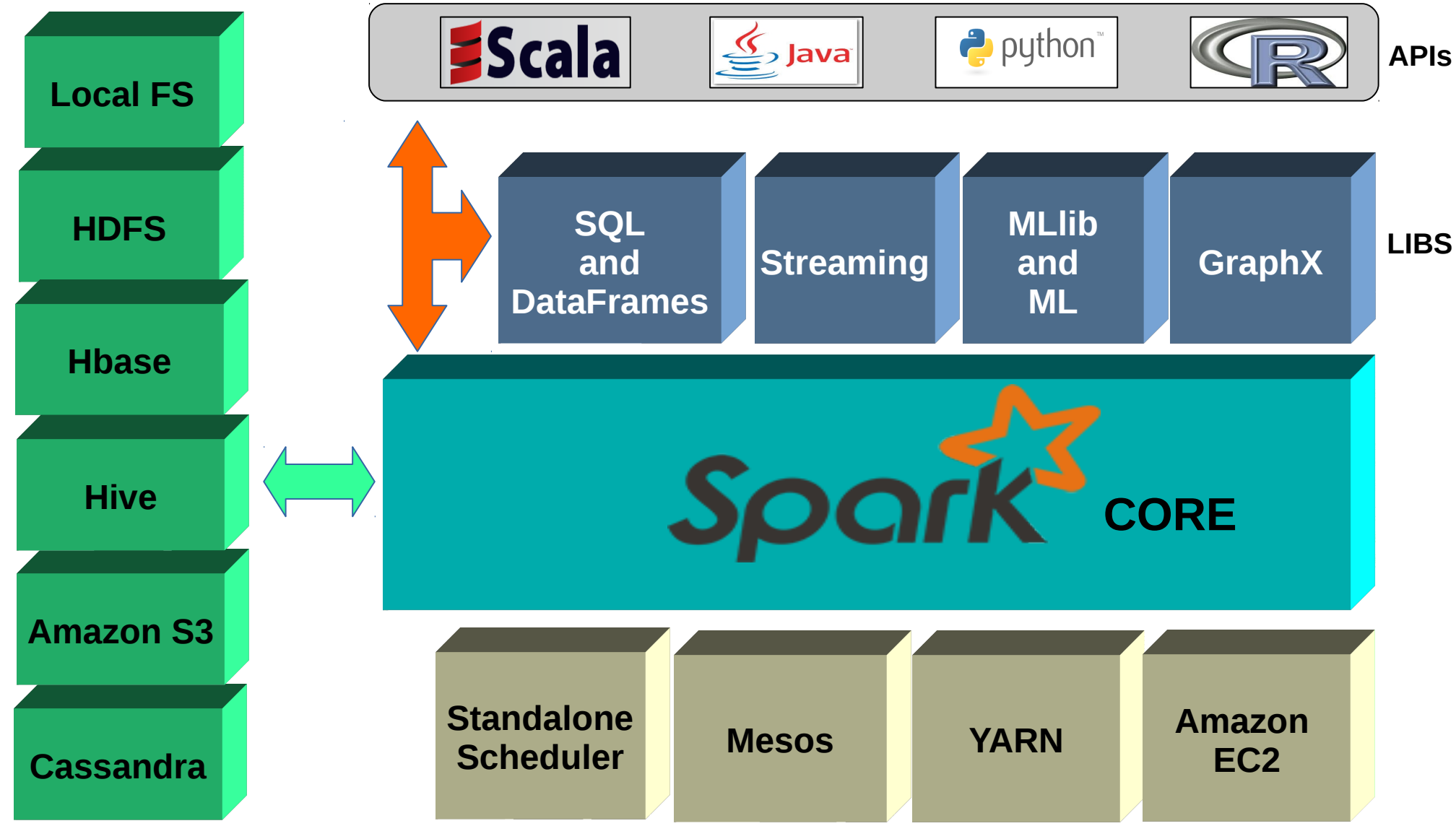
Spark is designed to be **fast** and **general purpose**.

The main functionality is implemented in Spark Core. Other components exist, that **integrate tightly** with Spark Core.

Benefits of tight integration:

- improvements in Core propagate to higher components
- it offers one unified environment

Spark Basics: architecture



Spark Basics: libraries

Currently the following libs exist and they are evolving really-really fast:

- SQL Lib
- Streaming Lib
- Machine Learning Lib (Mllib & ML)
- Graph Lib (GraphX)

We outline all of them but later we will cover some details about MLlib and GraphX

Spark SQL

Spark SQL is a library for querying structures datasets as well as distributed datasets.

Spark SQL allows relational queries expressed in **SQL**, **HiveQL**, or **Scala** to be executed using Spark.

Example:

```
hc = HiveContext(sc)
rows = hc.sql("select id, name, salary from emp")
rows.filter(lambda r: r.salary > 2000).collect()
```

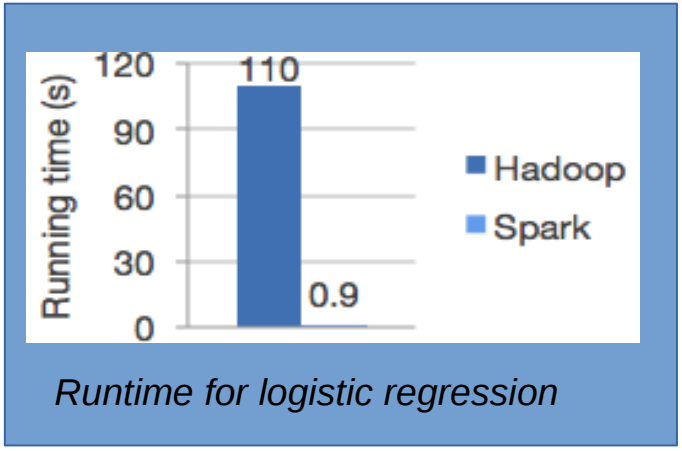
Spark MLlib & ML

MLlib is Spark's scalable machine learning library.

Two APIs: the RDD API and the DataFrame API.

Some supported algorithms:

- linear SVM and logistic regression
- classification and regression tree
- k-means clustering
- recommendation via alternating least squares
- **singular value decomposition (SVD)**
- linear regression with L1- and L2-regularization
- multinomial naive Bayes
- basic statistics
- feature transformations



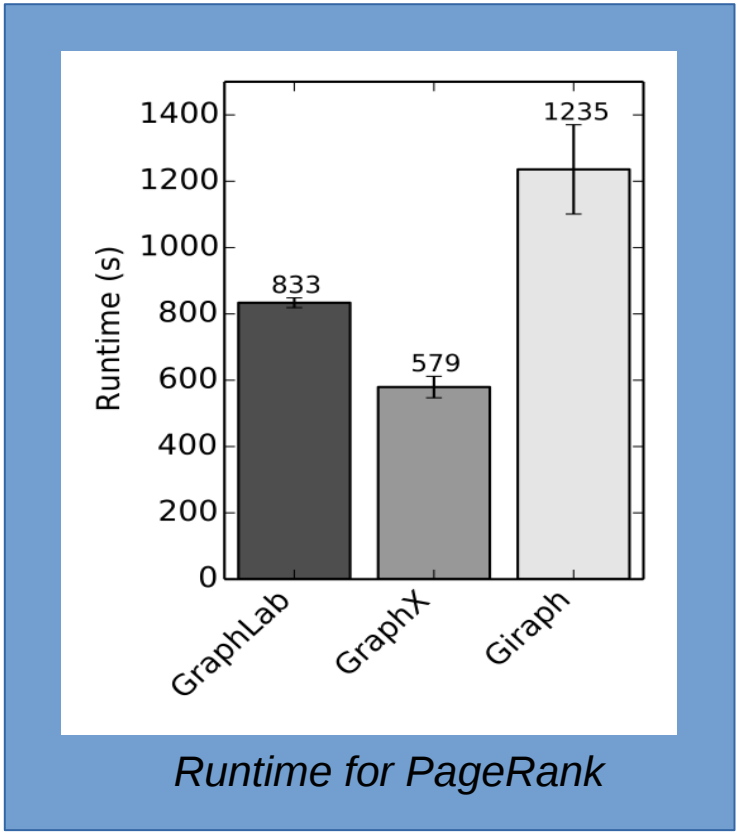
Runtime for logistic regression

Spark GraphX

GraphX provides an API for graph processing and graph-parallel algorithms on-top of Spark.

The current version supports:

- **PageRank**
- **Connected components**
- Label propagation
- SVD++
- Strongly connected components
- **Triangle counting**
- Core decomposition
- **...**



Runtime for PageRank

Spark Streaming

Spark Streaming is a library to ease the development of complex streaming applications.

Data can be inserted into Spark from different sources like **Kafka**, **Flume**, **Twitter**, **ZeroMQ**, **Kinesis** or **TCP sockets** can be processed using complex algorithms expressed with high-level functions like **map**, **reduce**, **join** and **window**.

RESILIENT

DISTRIBUTED

DATASETS

Resilient Distributed Datasets (RDDs)

Data manipulation in Spark is heavily based on RDDs. An RDD is an interface composed of:

- a set of partitions
- a list of dependencies
- a function to compute a partition given its parents
- a partitioner (optional)
- a set of preferred locations per partition (optional)

Simply stated: **an RDD is a distributed collections of items**. In particular: an RDD is a **read-only** (i.e., immutable) collection of items partitioned across a set of machines that can be rebuilt if a partition is destroyed.

Resilient Distributed Datasets (RDDs)

The RDD is the **most fundamental concept** in Spark since all work in Spark is expressed as:

- **creating** RDDs
- **transforming** existing RDDs
- **performing actions** on RDDs

Creating RDDs

Spark provides two ways to create an RDD:

- **loading** an already existing set of objects
- **parallelizing** a data collection in the driver

Creating RDDs

```
// define the spark context
val sc = new SparkContext(...)

// hdfsRDD is an RDD from an HDFS file
val hdfsRDD = sc.textFile("hdfs://...")

// localRDD is an RDD from a file in the local file system
val localRDD = sc.textFile("localfile.txt")

// define a List of strings
val myList = List("this", "is", "a", "list", "of", "strings")

// define an RDD by parallelizing the List
val listRDD = sc.parallelize(myList)
```

RDD Operations

There are **transformations** on RDDs that allow us to create new RDDs: `map, filter, groupBy, reduceByKey, partitionBy, sortByKey, join, etc`

Also, there are **actions** applied in the RDDs: `reduce, collect, take, count, saveAsTextFile, etc`

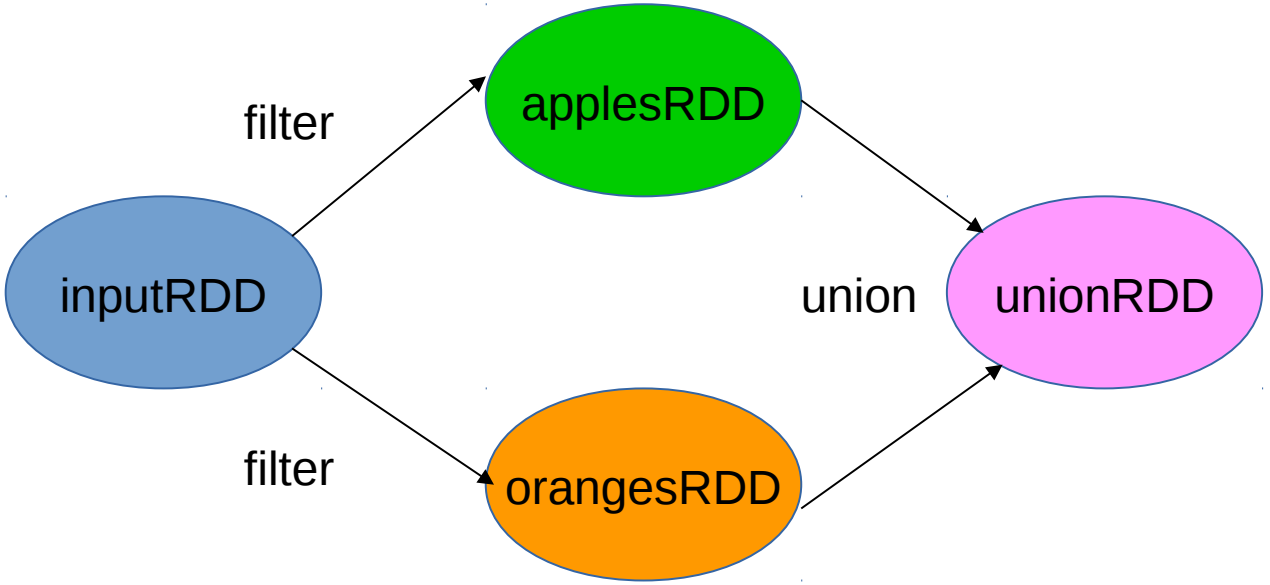
Note: computation takes place only in actions and not on transformations! (This is a form of **lazy evaluation**. More on this soon.)

RDD Operations: transformations

```
val inputRDD = sc.textFile("myfile.txt")

// lines containing the word “apple”
val applesRDD = inputRDD.filter(x => x.contains("apple"))

// lines containing the word “orange”
val orangesRDD = inputRDD.filter(x => x.contains("orange"))

// perform the union
val unionRDD = applesRDD.union(orangesRDD)
```

RDD Operations: transformations

Graphically speaking:



Spark maintains all necessary info to reconstruct an RDD based on the applied transformations. This is called the "lineage".

RDD Operations: actions

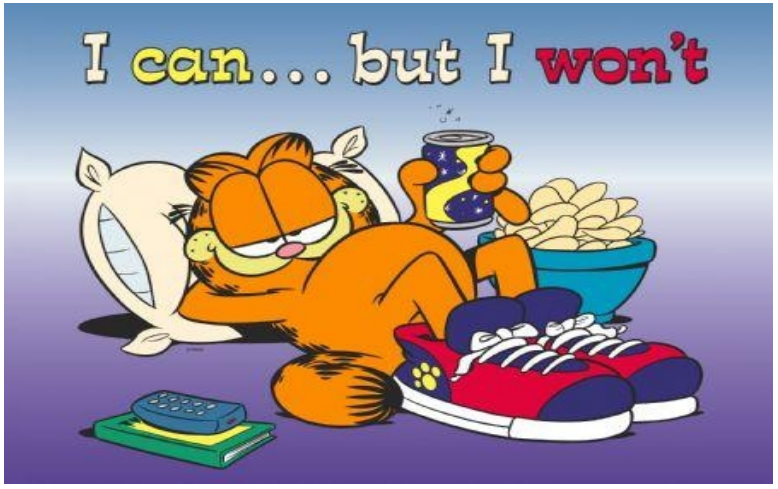
An action denotes that **finally something must be done**

We use the action `count()` to find the number of lines in unionRDD containing apples or oranges (or both) and then we print the 5 first lines using the action `take()`

```
val numLines = unionRDD.count()
unionRDD.take(5).foreach(println)
```

Lazy Evaluation

The benefits of being lazy



1. more optimization alternatives are possible if we see the **big picture**
2. we can avoid unnecessary computations

Ex:

Assume that from the unionRDD we need only the first 5 lines.

If we are eager, we need to compute the union of the two RDDs, materialize the result and then select the first 5 lines.

If we are lazy, there is no need to even compute the whole union of the two RDDs, since when we find the first 5 lines we may stop.

Lazy Evaluation

At any point we can **force the execution** of transformation by applying a simple action such as **`count()`**. This may be needed for debugging and testing.

Basic RDD Transformations

Assume that our RDD contains: `1,2,3,1`

map()	`rdd.map(x => x + 2)`	**{3,4,5,3}**
flatMap()	`rdd.flatMap(x => List(x-1,x,x+1))`	**{0,1,2,1,2,3,2,3,4,0,1,2}**
filter()	`rdd.filter(x => x>1)`	**{2,3}**
distinct()	`rdd.distinct()`	**{1,2,3}**
sample()	`rdd.sample(false,0.2)`	**non-predictable**

Two-RDD Transformations

These transformations require two RDDs

`union()`	`rdd.union(another)`
`intersection()`	`rdd.intersection(another)`
`subtract()`	`rdd.substract(another)`
`cartesian()`	`rdd.cartesian(another)`

Some Actions

collect()	rdd.collect()	**{1,2,3,1}**
count()	rdd.count()	**4**
countByValue()	rdd.countByValue()	**{(1,2),(2,1),(3,1)}**
take()	rdd.take(2)	**{1,2}**
top()	rdd.top(2)	**{3,2}**
reduce()	rdd.reduce((x,y) => x+y)	**7**
foreach()	rdd.foreach(func)	

Transformations for Key-Value RDDs

A **key-value RDD** contains pairs (*K*,*V*), where *K* is considered as the key and *V* is the value. This is a very important type of data and therefore, Spark supported specialized transformations for key-value RDDs, such as:

- `groupByKey`
- `aggregateByKey`
- `reduceByKey`
- `sortByKey`
- `join`

Transformations for Key-Value RDDs

`groupByKey([numPartitions]):`

When called on a dataset of (*K*, *V*) pairs, returns a dataset of (*K*, Iterable<*V*>) pairs.

`aggregateByKey(zeroValue)(seqOp, combOp, [numPartitions]):`

When called on a dataset of (*K*, *V*) pairs, returns a dataset of (*K*, *U*) pairs where the values for each key are aggregated using the given combine functions and a neutral "zero" value.

`reduceByKey(func, [numPartitions]):`

When called on a dataset of (*K*, *V*) pairs, returns a dataset of (*K*, *V*) pairs where the values for each key are aggregated using the given reduce function func, which must be of type (*V*,*V*) => *V*.

Transformations for Key-Value RDDs

`sortByKey([ascending],[numPartitions])`:

When called on a dataset of (*K*, *V*) pairs where *K* implements Ordered, returns a dataset of (*K*, *V*) pairs sorted by keys in ascending or descending order, as specified in the boolean ascending argument.

`join(otherDataset, [numPartitions])`:

When called on datasets of type (*K*, *V*) and (*K*, *W*), returns a dataset of (*K*, (*V*, *W*)) pairs with all pairs of elements for each key. Outer joins are supported through **`leftOuterJoin`**, **`rightOuterJoin`**, and **`fullOuterJoin`**.

DIRECTED

ACYCLIC

GRAPHS

RDDs and DAGs

A set of RDDs is transformed to a Directed Acyclic Graph (DAG)

Input: RDDs and partitions to compute

Output: output from actions on those partitions

Roles:

> Build stages of tasks

> Submit them to lower level scheduler (e.g. YARN, Mesos, Standalone) as ready

> Lower level scheduler will schedule data based on locality

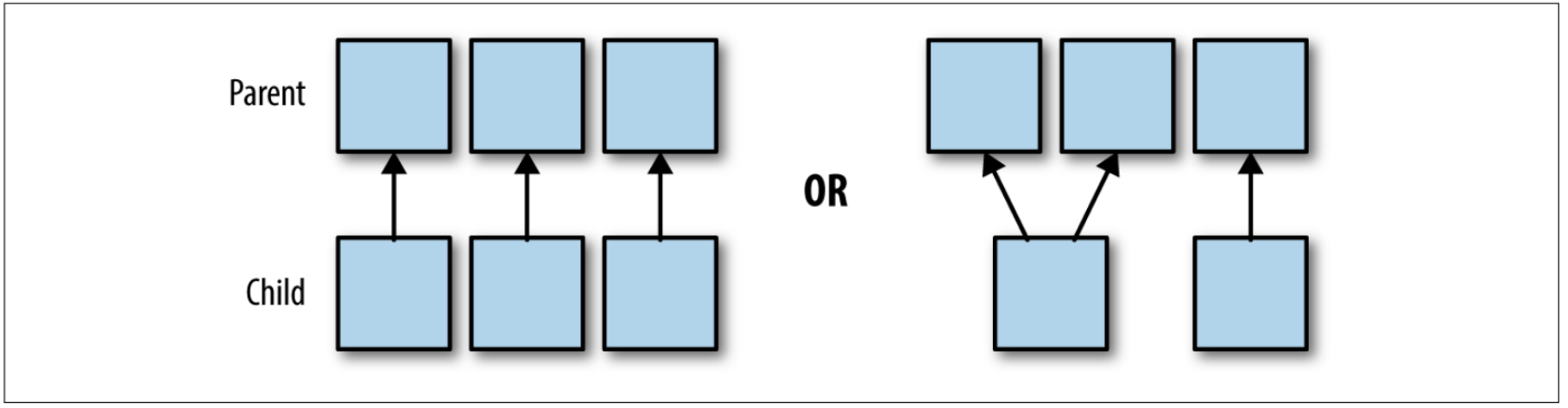
> Resubmit failed stages if outputs are lost

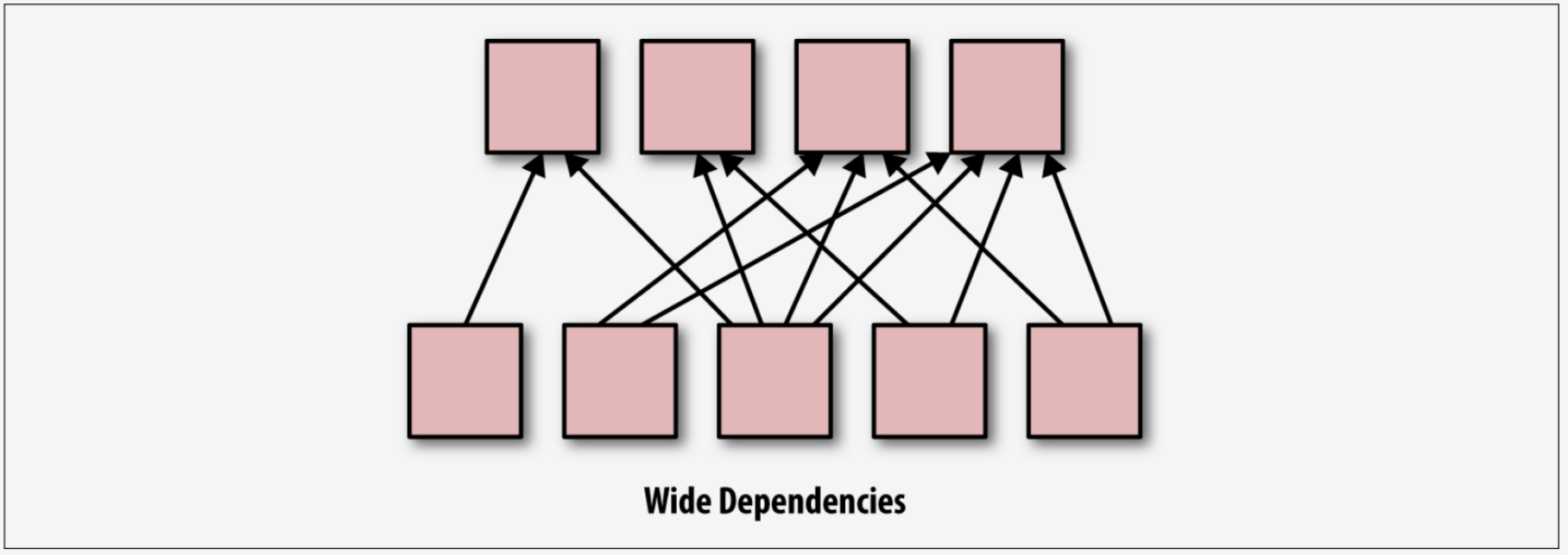
Narrow vs Wide Dependencies

Narrow transformations are those in which each partition in the child RDD has simple, finite dependencies on partitions in the parent RDD. Dependencies are only narrow if they can be determined at design time, irrespective of the values of the records in the parent partitions, and if each parent has at most one child partition. Specifically, partitions in narrow transformations can either depend on one parent (such as in the **map** operator), or a unique subset of the parent partitions that is known at design time (**coalesce**). Thus narrow transformations can be executed on an arbitrary subset of the data without any information about the other partitions.

In contrast, transformations with **wide dependencies** cannot be executed on arbitrary rows and instead require the data to be partitioned in a particular way, e.g., according to the value of their key. In sort, for example, records have to be partitioned so that keys in the same range are on the same partition. Transformations with wide dependencies include **sort** , **reduceByKey** , **groupByKey** , **join** , and anything that calls the **rePartition** function.

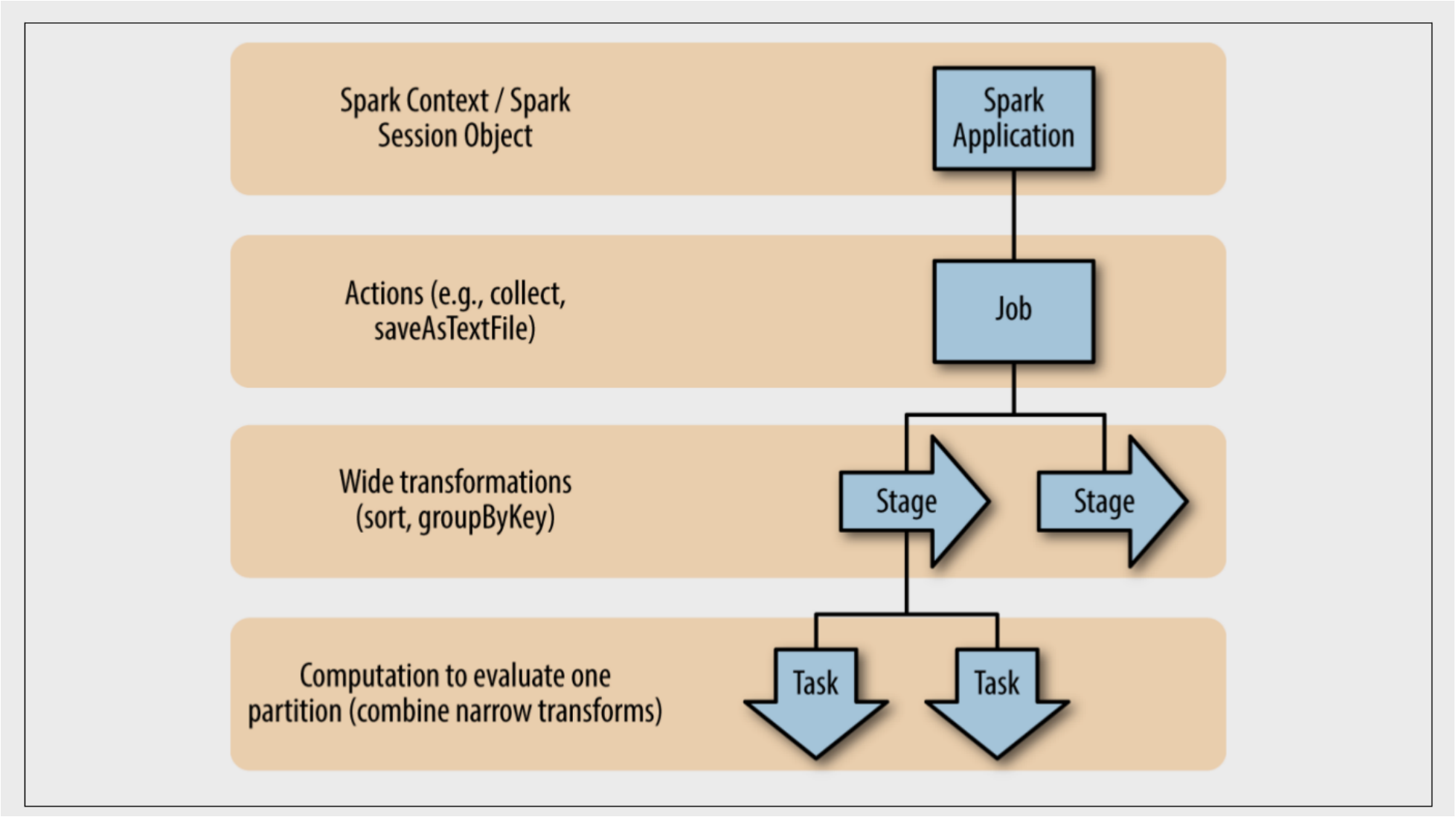
Narrow vs Wide Dependencies





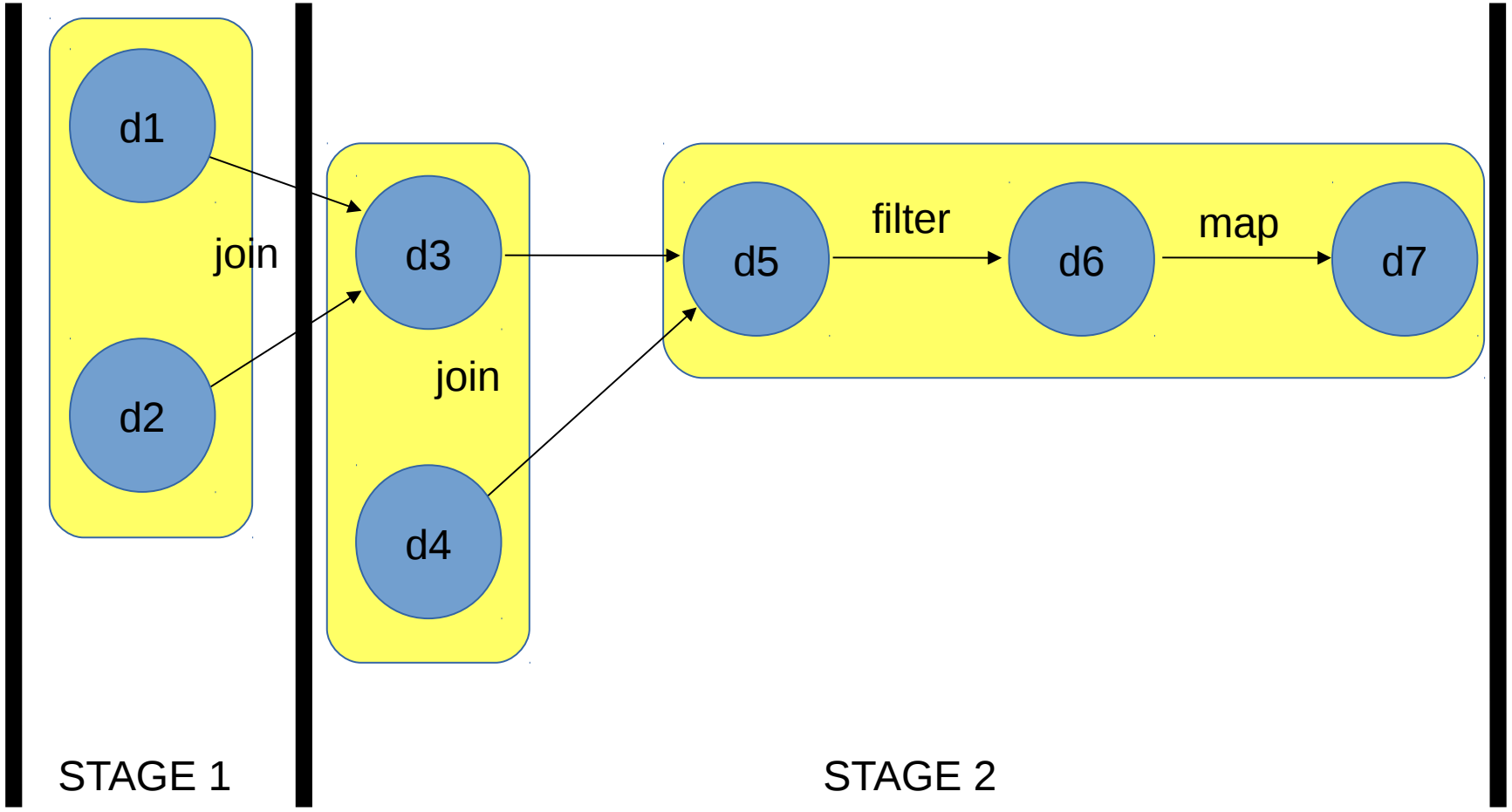
Source: High Performance Spark, O' Reilly, 2017.

Spark Application Tree

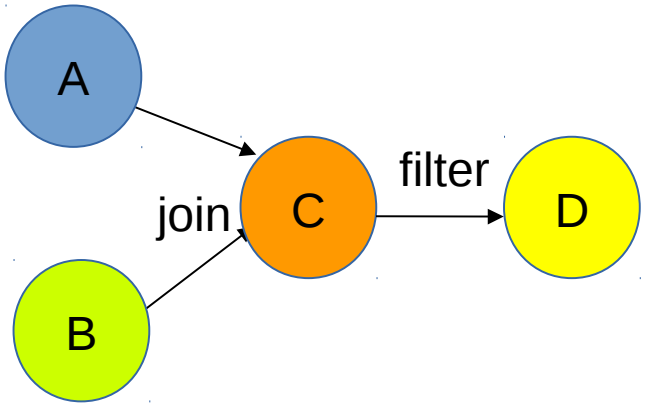


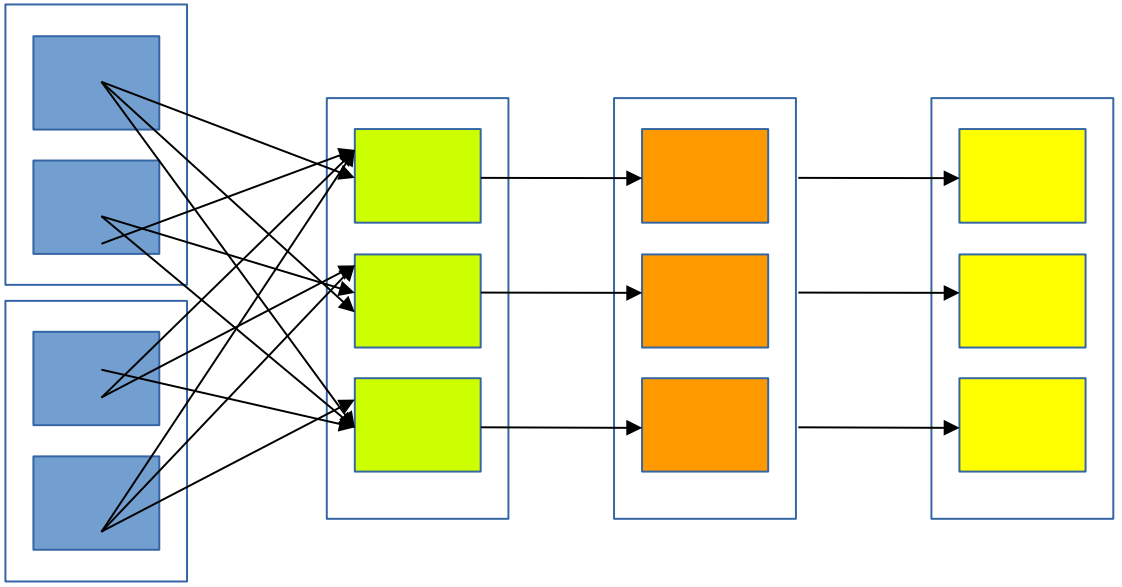
Source: High Performance Spark, O' Reilly, 2017.

DAG Scheduling



DAG Scheduling





RDD objects

DAG scheduler

```
A.join(B).filter(...).filter(...)
```

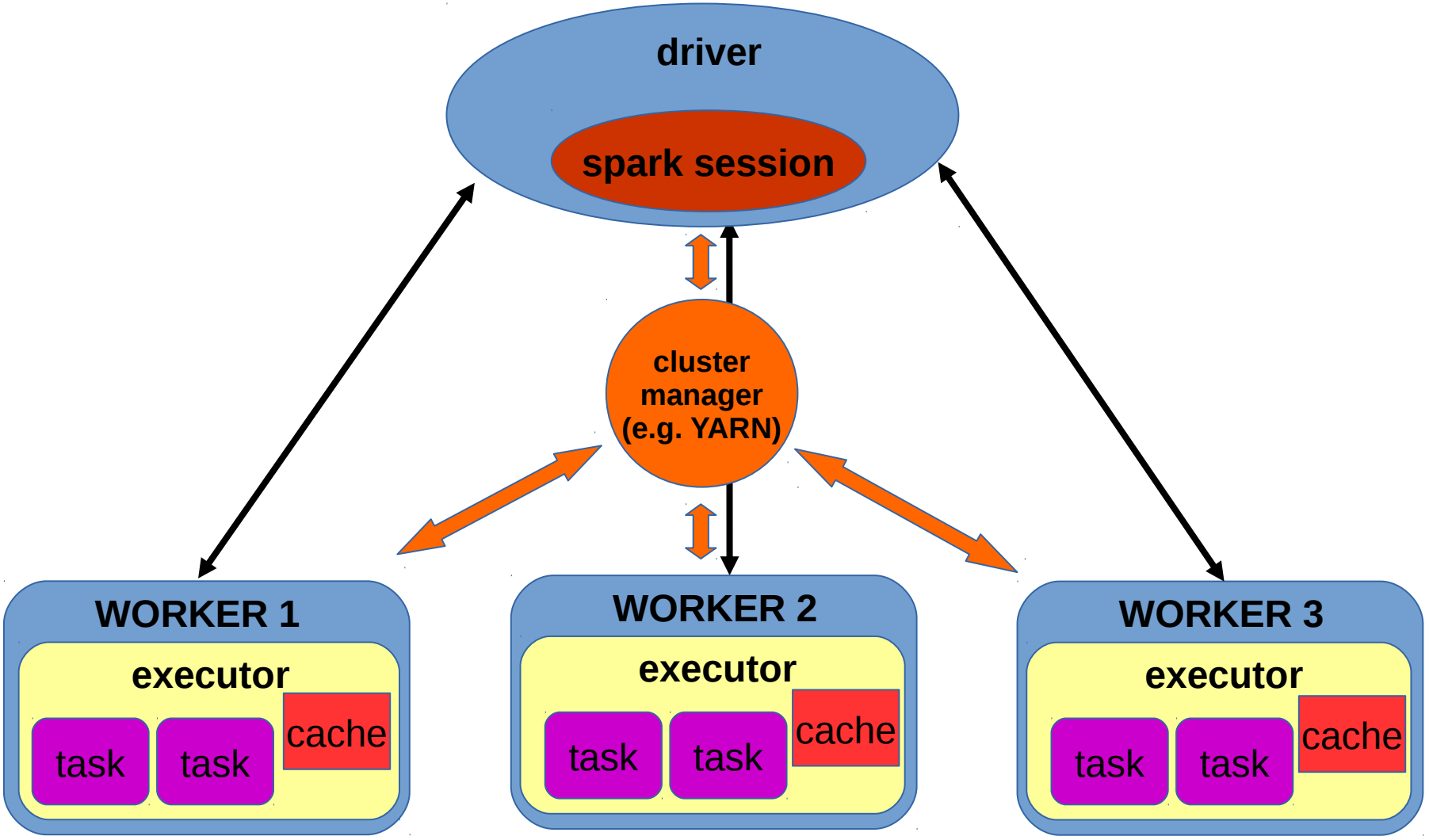
split graph into stages of tasks
submit each stage

Distributed Execution in Spark

Outline of the whole process:

1. The user submits a job with **`spark-submit`**.
2. **`spark-submit`** launches the driver program and invokes the **`main()`** method specified by the user.
3. The **driver program** contacts the **cluster manager** to ask for resources to launch **executors**.
4. The **cluster manager** launches **executors** on behalf of the **driver program**.
5. The **driver process** runs through the user application. Based on the RDD actions and transformations in the program, the **driver** sends work to **executors** in the form of **tasks**.
6. **Tasks** are run on **executor processes** to compute and save results.
7. If the **driver's `main()`** method exits or it calls **`SparkContext.stop()`** , it will terminate the **executors** and release resources from the **cluster manager**.

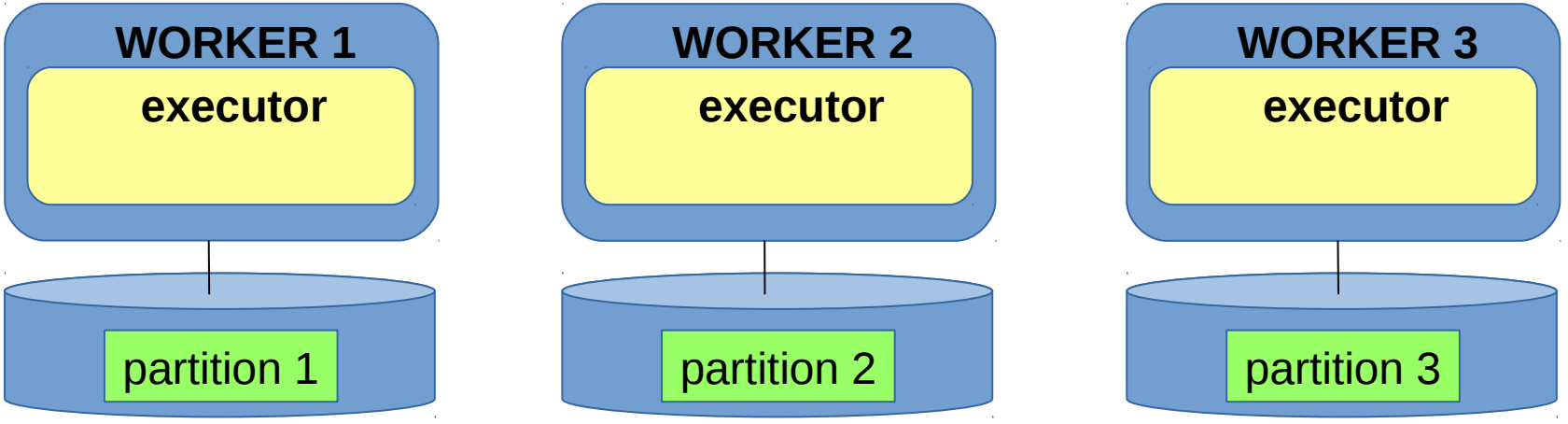
Under the Hood



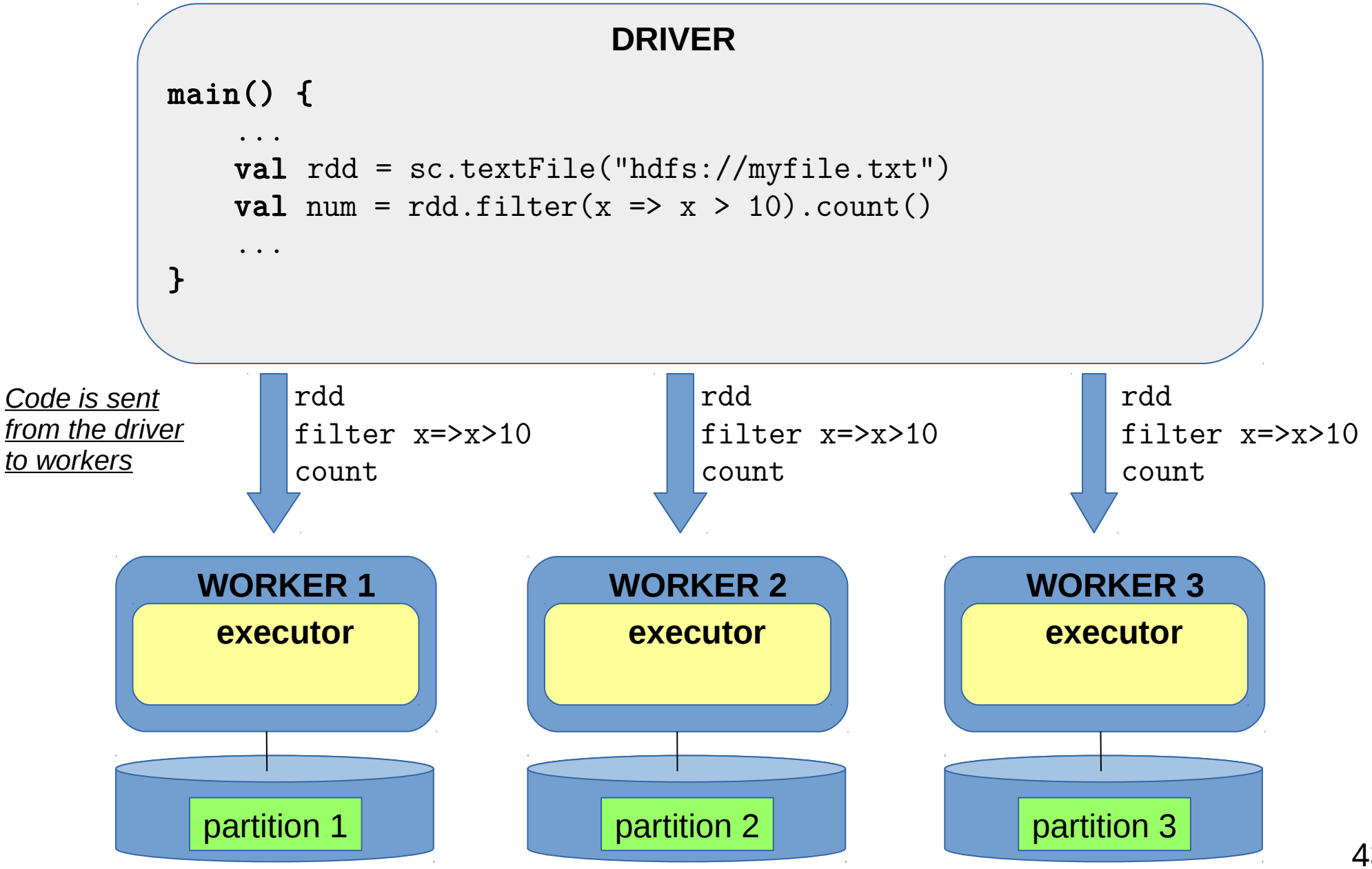
Under the Hood

DRIVER

```
main() {
    ...
    val rdd = sc.textFile("hdfs://myfile.txt")
    val num = rdd.filter(x => x > 10).count()
    ...
}
```



Under the Hood



Example Application

Given an RDD of integers, sum the occurrences of the same integer for any integer larger than 100 and then sort in ascending order.

E.g., for the input: 1, 2, 1, 2, 3, 4, 8, 7, 6, 8, 6, 6

The output should be:

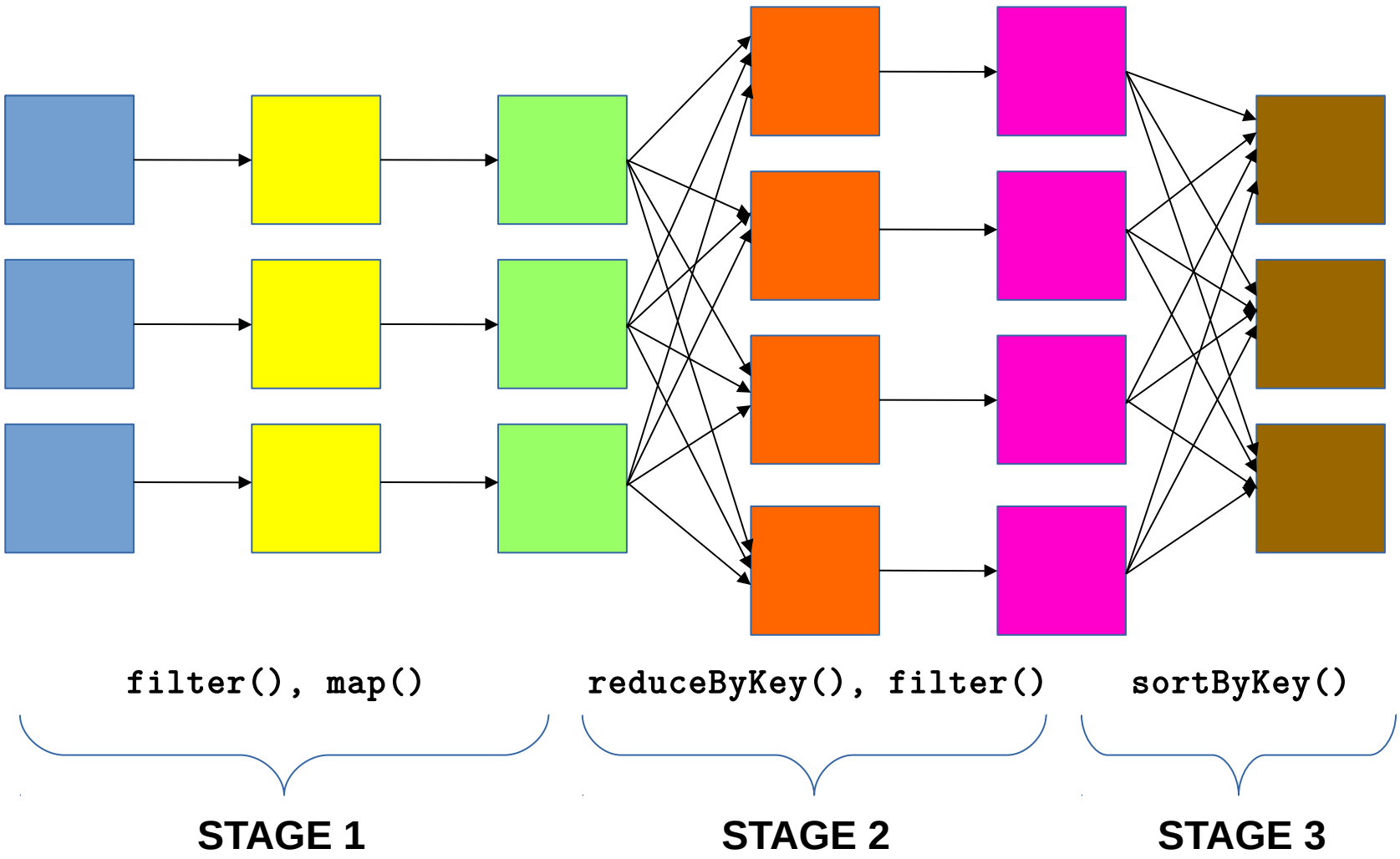
(1,2), (2,4), (3,3), (4,4), (6,18), (7,7), (8,16)

Example Application

```
val result = rdd // our initial RDD
.filter(x => x>100) // keep relevant integers
.map(x => (x,x)) // create key-value RDD
.reduceByKey(_+_) // reduce
.filter((x,y) => y > 1000) // filter on value
.sortByKey() // sort based on key
result.take(10).foreach(println) // print
```

CAN YOU IDENTIFY THE STAGES ?

Example Application



PERSISTENCE

Persistence

In many cases we want to use the same RDD multiple times **without recomputing it**.

Ex:

```
val result = rdd.map(x => x+1)
println(result.count())
println(result.collect().mkString(","))
```

We can ask Spark to keep (persist) the data.

Persistence

```
val result = rdd.map(x => x+1)
result.persist(StorageLevel.DISK_ONLY)
println(result.count())
println(result.collect().mkString(","))
```

Persistence levels:

`MEMORY_ONLY`

`MEMORY_ONLY_SER` (objects are serialized)

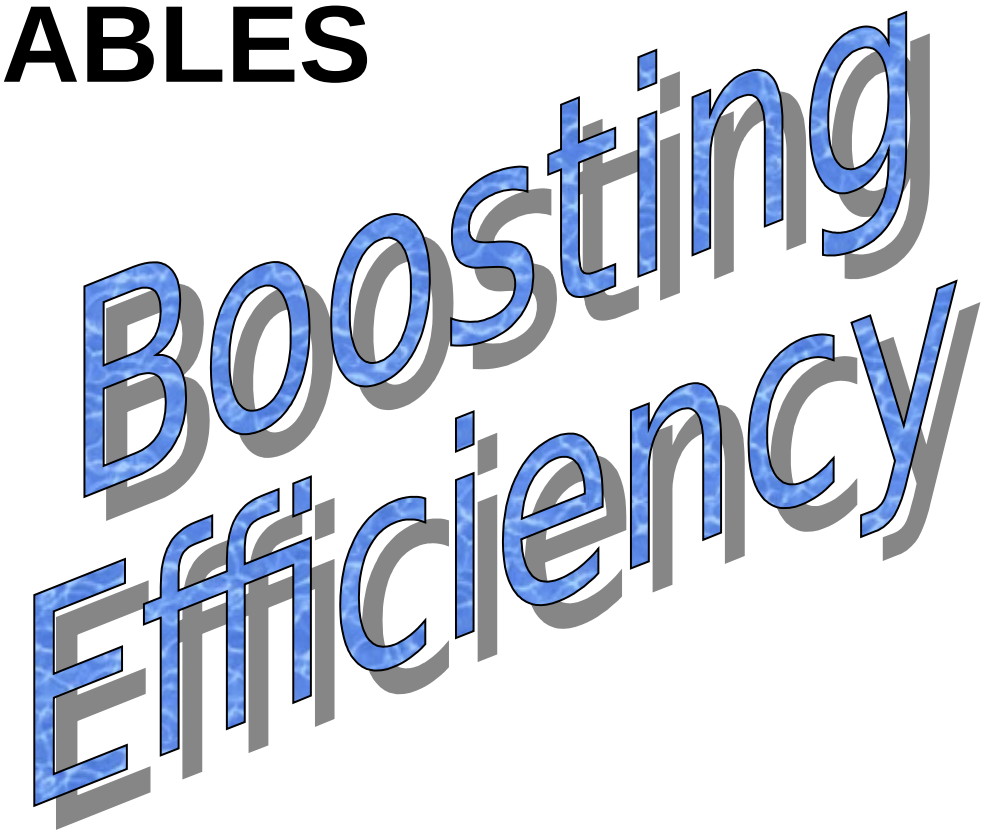
`MEMORY_AND_DISK`

`MEMORY_AND_DISK_SER` (objects are serialized)

`DISK_ONLY`

If we try to put to many things in RAM Spark starts flushing data to disk using a Least Recently Used policy.

BROADCAST VARIABLES AND ACCUMULATORS



Broadcast Variables

"Broadcast variables allow the programmer to keep a read-only variable cached on each machine **rather than shipping a copy of it with tasks**. They can be used, for example, to give every node a copy of a large input dataset in an efficient manner. Spark also attempts to distribute broadcast variables using **efficient broadcast algorithms** to **reduce communication cost**." (*source: Apache Spark website*)

- A piece of information is broadcasted to all executors.
- Broadcast variables are set by the driver and are read-only by the executors.
- Very useful feature to avoid sending large pieces of data again and again.

Broadcast Variables

```
val r = scala.util.Random
val vector = Array.fill(1000){r.nextInt(99)}

val rdd = Array.fill(1000000){r.nextInt(99)}.parallelize

val bvector = sc.broadcast(vector)
```

```
rdd.map(x => vector.contains(x))
rdd.map(x => bvector.value.contains(x))
```

Accumulators

The value of an accumulator can be modified by any executor (**shared variable**).

No special protection is required, Spark takes care of this.

Executors can only **write** to the accumulator. They cannot **read** the value of the accumulator.

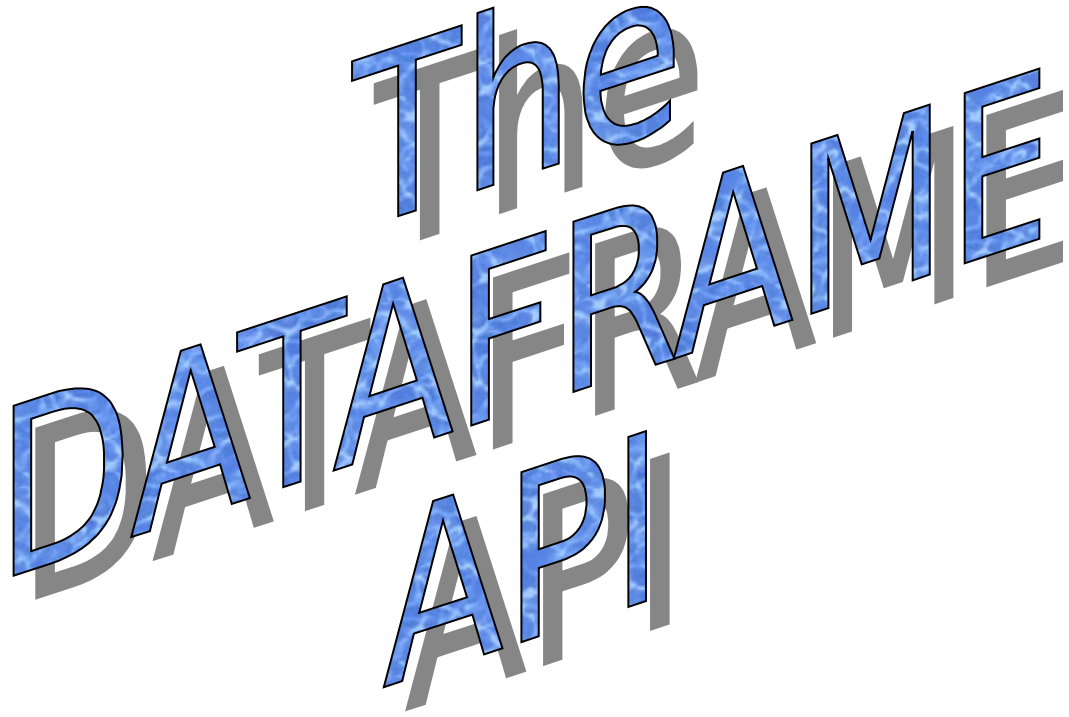
The value of an accumulator can be read by the driver only.

Accumulators

Example

```
// define an accumulator
val counter = sc.longAccumulator("counter")

// sum the contents of a list
sc.parallelize(1 to 9).foreach(x => counter.add(x))
```

DATAFRAMES



Why DataFrames ?

From Apache Spark documentation:

"A **DataFrame** is a dataset organized into named columns. It is conceptually equivalent to a table in a relational database or a data frame in R/Python, but with **richer optimizations under the hood**."

Important: DataFrames are structured!

Using DataFrames

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

val spark_session = SparkSession
  .builder()
  .appName("Spark SQL basic example")
  .config("spark.some.config.option", "some-value")
  .getOrCreate()

// For implicit conversions like converting RDDs to DataFrames
import spark.implicits._
```

Using DataFrames

```
// Create a DataFrame drom a json file
val df = spark_session.read.json("examples/src/main/resources/people.json")

// Create a DataFrame from a csv file
val df =  spark_session.read.option("header", "false").csv("reviews.csv")

// Show the first few lines of the DataFrame
df.show()

// Print the schema of the DataFrame
df.printSchema()
```

Using DataFrames

```
// Selection operation

val resdf = df.select($"name", $"age" + 1)


// SQL query passed as a string

val resdf = spark_session.sql("SELECT name, age FROM
  people WHERE age BETWEEN 13 AND 19")
```

Spark Simple Examples

Spark supports

- ✓ Java
- ✓ R
- ✓ **Python**
- ✓ **Scala**

We are going to use the **Scala API** in this lecture. We will play with **Spark Core** component and also run examples of **MLlib** and **GraphX** libraries that are very relevant to Graph Data Mining.

Also, some **Python** examples will be discussed.

Scala Examples

WordCount

```
import org.apache.spark.SparkContext._
import org.apache.spark.{SparkConf, SparkContext}

object WordCount {

  def main(args: Array[String]): Unit = {

    val sparkConf = new SparkConf().setMaster("local[2]").setAppName("WordCount")
    val sc = new SparkContext(sparkConf)  // create spark context

    val inputFile = "hdfs://myhdfs/leonardo.txt"
    val outputDir = "hdfs://myhdfs/output"
    val txtFile = sc.textFile(inputFile)

    txtFile.flatMap(line => line.split(" ")) // split each line based on spaces
      .map(word => (word,1)) // map each word into a word,1 pair
      .reduceByKey(_+_) // reduce
      .saveAsTextFile(outputDir) // save the output

    sc.stop()
  }
}
```

WordCount in Hadoop

```
import java.io.IOException;
import java.util.*;

import org.apache.hadoop.fs.Path;
import org.apache.hadoop.conf.*;
import org.apache.hadoop.io.*;
import org.apache.hadoop.mapreduce.*;
import org.apache.hadoop.mapreduce.lib.input.FileInputFormat;
import org.apache.hadoop.mapreduce.lib.input.TextInputFormat;
import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;
import org.apache.hadoop.mapreduce.lib.output.TextOutputFormat;

public class WordCount {

 public static class Map extends Mapper<LongWritable, Text, Text,
 IntWritable> {
    private final static IntWritable one = new IntWritable(1);
    private Text word = new Text();

   public void map(LongWritable key, Text value, Context context)
 throws IOException, InterruptedException {
        String line = value.toString();
        StringTokenizer tokenizer = new StringTokenizer(line);
        while (tokenizer.hasMoreTokens()) {
            word.set(tokenizer.nextToken());
            context.write(word, one);
        }
    }
 }

 public static class Reduce extends Reducer<Text, IntWritable, Text,
 IntWritable> {

    public void reduce(Text key, Iterable<IntWritable> values, Context
 context)
      throws IOException, InterruptedException {
        int sum = 0;
        for (IntWritable val : values) {
            sum += val.get();
        }
        context.write(key, new IntWritable(sum));
    }
 }
```

```
    public static void main(String[] args) throws Exception {
        Configuration conf = new Configuration();
        Job job = new Job(conf, "wordcount");
        job.setOutputKeyClass(Text.class);
        job.setOutputValueClass(IntWritable.class);
        job.setMapperClass(Map.class);
        job.setReducerClass(Reduce.class);
        job.setInputFormatClass(TextInputFormat.class);
        job.setOutputFormatClass(TextOutputFormat.class);

        FileInputFormat.addInputPath(job, new Path(args[0]));
        FileOutputFormat.setOutputPath(job, new Path(args[1]));


        job.waitForCompletion(true);

    }

}
```

PageRank

```
object PageRank {

  def main(args: Array[String]) {
    val iters = 10  // number of iterations for pagerank computation
    val currentDir = System.getProperty("user.dir")  // get the current directory
    val inputFile = "file://" + currentDir + "/webgraph.txt"
    val outputDir = "file://" + currentDir + "/output"

    val sparkConf = new SparkConf().setAppName("PageRank")
    val sc = new SparkContext(sparkConf)
    val lines = sc.textFile(inputFile, 1)

    val links = lines.map { s => val parts = s.split("\\s+")(parts(0), parts(1))}.distinct().groupByKey().cache()
    var ranks = links.mapValues(v => 1.0)
    for (i <- 1 to iters) {
      println("Iteration: " + i)
      val contribs = links.join(ranks).values.flatMap{ case (urls, rank) => val size = urls.size urls.map(url =>
 (url, rank / size)) }

      ranks = contribs.reduceByKey(_ + _).mapValues(0.15 + 0.85 * _)
    }

    val output = ranks.collect()
    output.foreach(tup => println(tup._1 + " has rank: " + tup._2 + "."))

    sc.stop()
  }
}
```

More on MLlib

MLlib provides some additional data types common in Machine Learning

Vector (a math vector, either sparse or dense)

LabeledPoint (useful in classification and regression)

Rating (useful in recommendation algorithms)

Several **Models** (used in training algorithms)

SVD with MLlib

```
import org.apache.spark.mllib.linalg.Matrix

import org.apache.spark.mllib.linalg.distributed.RowMatrix

import
 org.apache.spark.mllib.linalg.SingularValueDecomposition


val mat: RowMatrix = ...


// Compute the top 20 singular values and corresponding singular vectors.

val svd: SingularValueDecomposition[RowMatrix, Matrix] =
 mat.computeSVD(20, computeU = true)

val U: RowMatrix = svd.U // The U factor is a RowMatrix.

val s: Vector = svd.s // The singular values are stored in a local dense vector.

val V: Matrix = svd.V // The V factor is a local dense matrix.
```

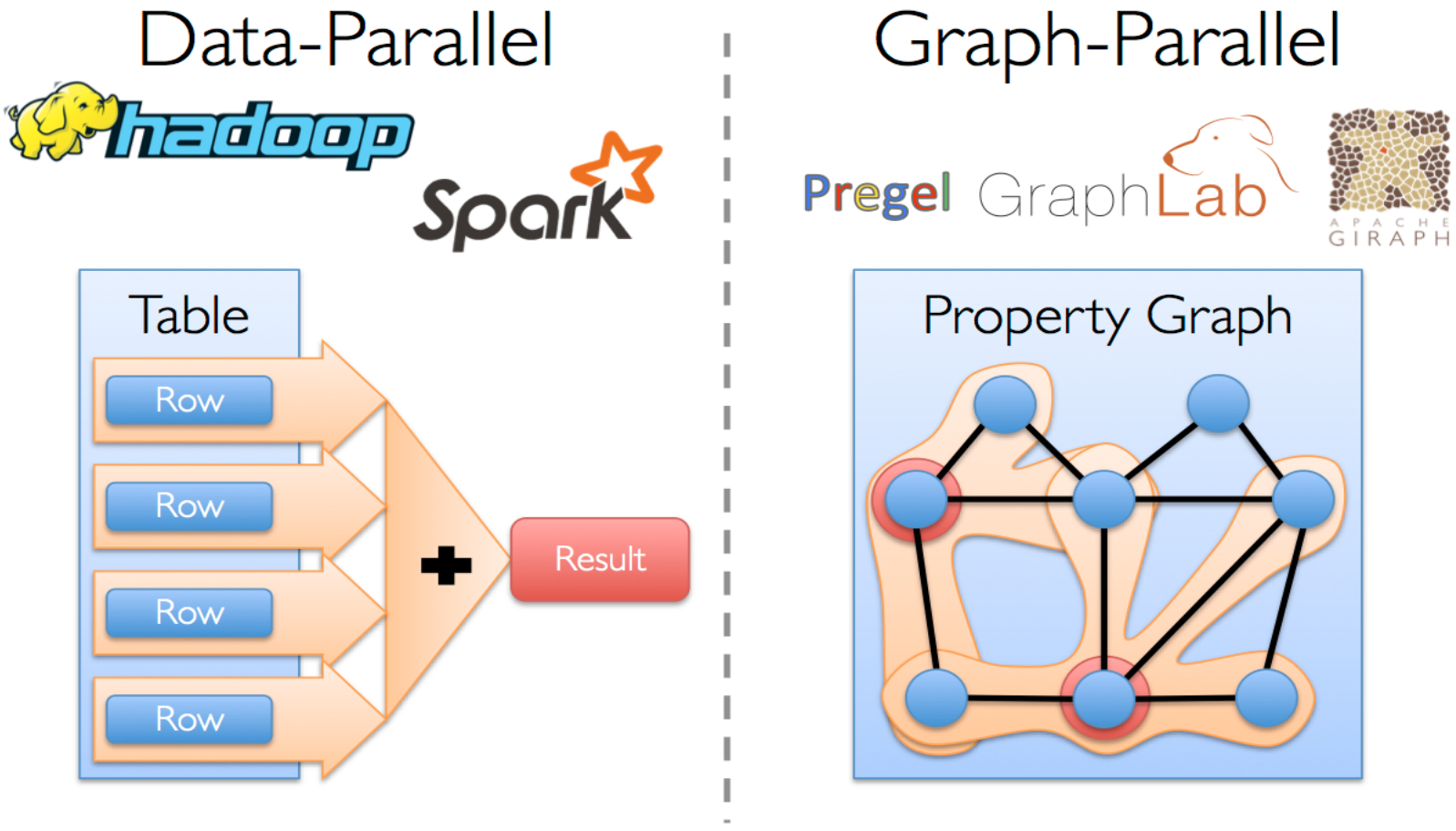
More on GraphX

The basic concept in GraphX is the property graph

The **property graph** is a directed multigraph with user defined objects attached to each vertex and edge.

GraphX optimizes the representation of vertex and edge types when they are plain old data-types (e.g., int) reducing in memory footprint by storing them in **specialized arrays**.

More on GraphX



"While graph-parallel systems are optimized for iterative diffusion algorithms like PageRank they are not well suited to more basic tasks like constructing the graph, modifying its structure, or expressing computation that spans multiple graphs"

Source: http://ampcamp.berkeley.edu

More on GraphX

This means that for some tasks Spark may not show the best performance in comparison to other ***dedicated*** graph processing systems.

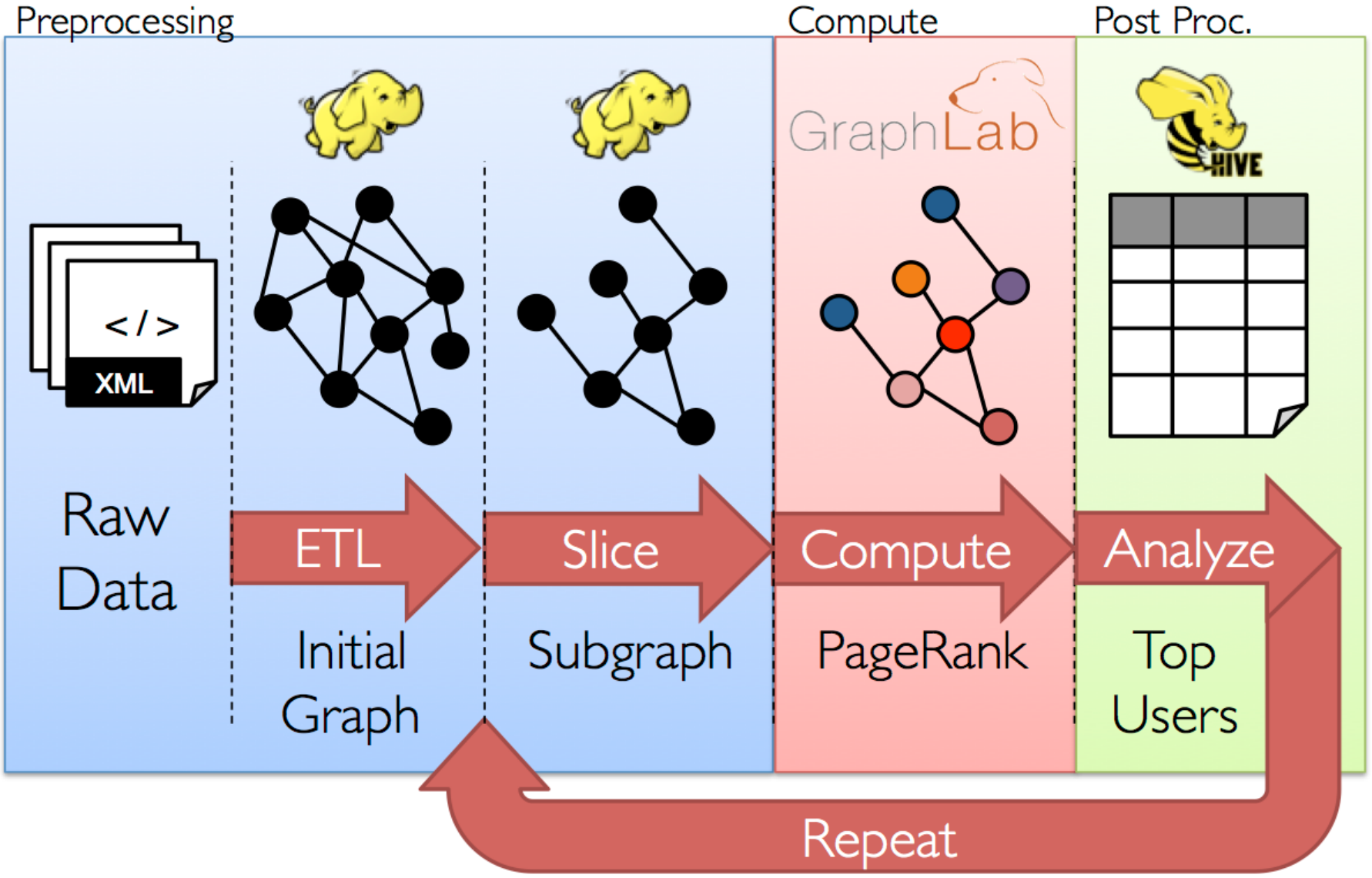
Ex:

PageRank on Live-Journal network (available @snap)

GraphLab is 60 times faster than Hadoop

GraphLab is **16 times** faster than Spark

More on GraphX



Source: http://spark.apache.org

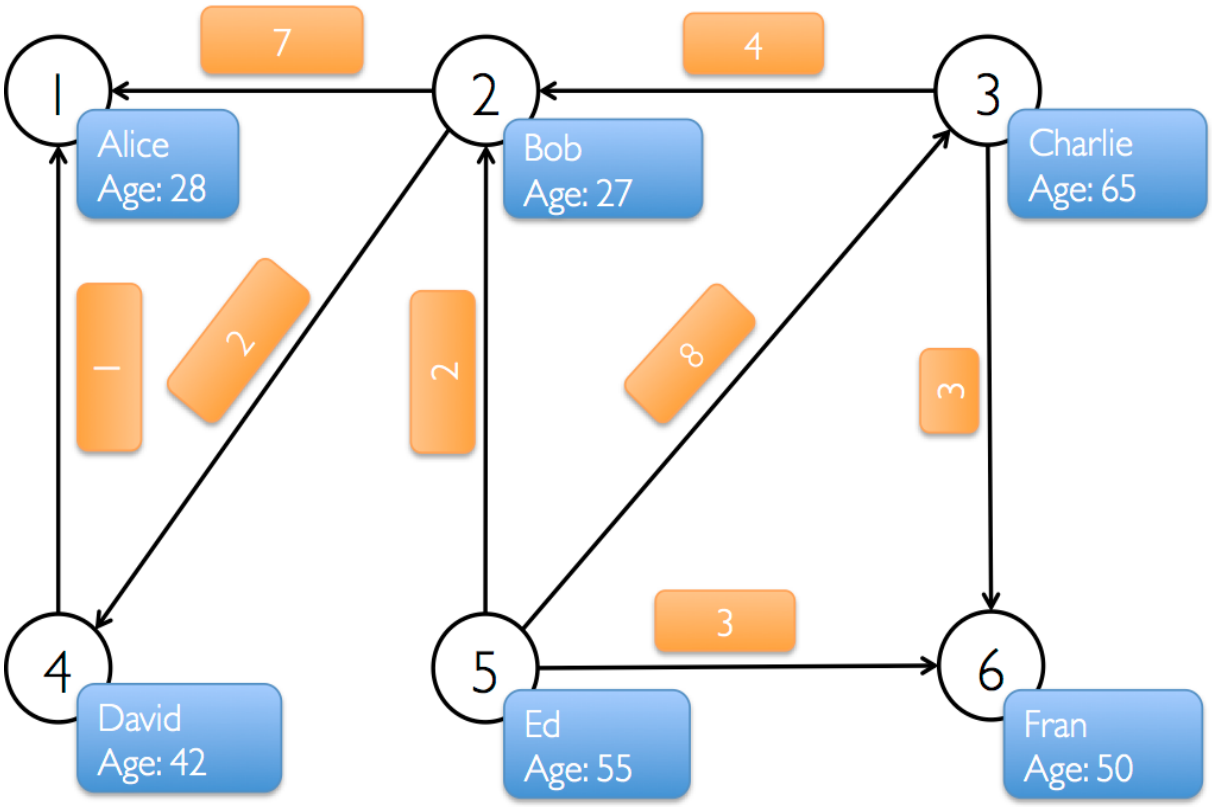
More on GraphX

To use GraphX we need to import

```
import org.apache.spark._
import org.apache.spark.graphx._
import org.apache.spark.rdd.RDD
```

More on GraphX

```
val vertexArray = Array(
  (1L, ("Alice", 28)),
  (2L, ("Bob", 27)),
  (3L, ("Charlie", 65)),
  (4L, ("David", 42)),
  (5L, ("Ed", 55)),
  (6L, ("Fran", 50))
  )
val edgeArray = Array(
  Edge(2L, 1L, 7),
  Edge(2L, 4L, 2),
  Edge(3L, 2L, 4),
  Edge(3L, 6L, 3),
  Edge(4L, 1L, 1),
  Edge(5L, 2L, 2),
  Edge(5L, 3L, 8),
  Edge(5L, 6L, 3)
  )
```



Source: http://ampcamp.berkeley.edu

More on GraphX

Parallelizing nodes and edges

```
val vertexRDD: RDD[(Long, (String, Int))] =
 sc.parallelize(vertexArray)
val edgeRDD: RDD[Edge[Int]] =
 sc.parallelize(edgeArray)
```

Now we have **`vertexRDD`** for the nodes and **`edgeRDD`** for the edges.

More on GraphX

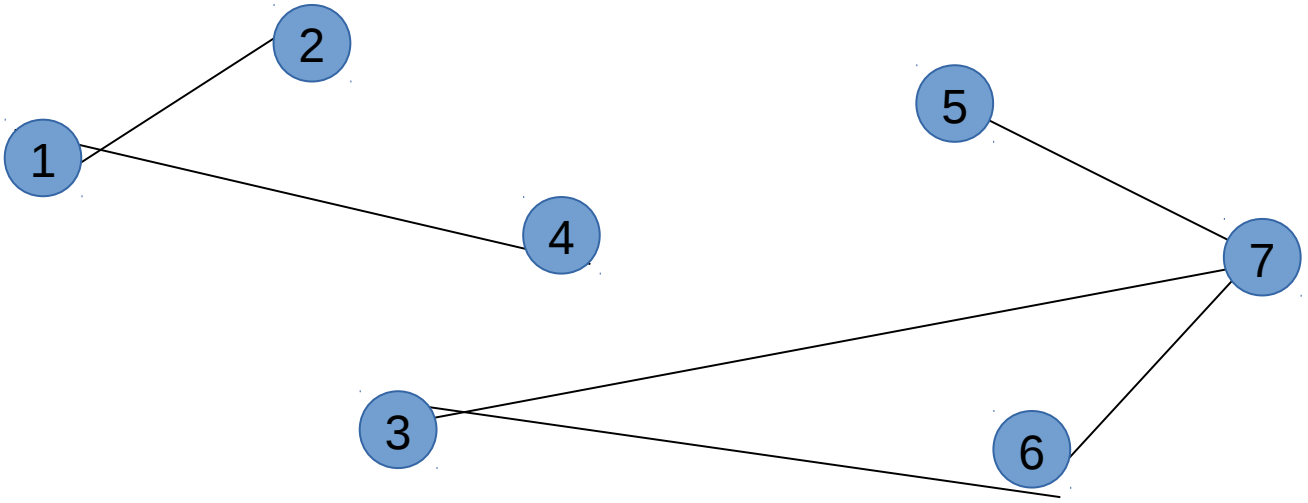
Last step: define the graph object

```
val graph: Graph[(String, Int), Int]
 = Graph(vertexRDD, edgeRDD)
```

PageRank with GraphX

```
object PageRank {
  def main(args: Array[String]): Unit = {
    val conf = new SparkConf().setAppName("PageRank App")
    val sc = new SparkContext(conf)
    val currentDir = System.getProperty("user.dir")
    val edgeFile = "file://" + currentDir + "/followers.txt"

    val graph = GraphLoader.edgeListFile(sc, edgeFile)

    // run pagerank
    val ranks = graph.pageRank(0.0001).vertices

    println(ranks.collect().mkString("\n")) // print result
  }
}
```

Connected Components



This graph has two connected components:

cc1 = {1, 2, 4}

cc2 = {3, 5, 6, 7}

Output:
(1,1) (2,1) (4,1)
(3,3) (5,3) (6,3) (7,3)

Connected Components

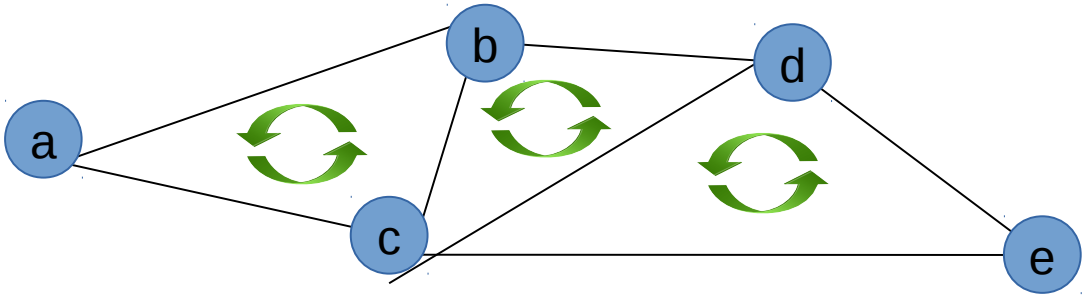
```
object ConnectedComponents {

  def main(args: Array[String]): Unit = {
    val conf = new SparkConf().setAppName("ConnectedComponents
 App")
    val sc = new SparkContext(conf)

    val currentDir = System.getProperty("user.dir")
    val edgeFile = "file://" + currentDir + "/graph.txt"
    val graph = GraphLoader.edgeListFile(sc, edgeFile)

    // find the connected components
    val cc = graph.connectedComponents().vertices

    println(cc.collect().mkString("\n"))  // print the result
  }
}
```

Counting Triangles

Triangles are very important in Network Analysis:

- dense subgraph mining (communities, trusses)
- triangular connectivity
- network measurements (e.g. clustering coefficient)

Example



Counting Triangles

```
object TriangleCounting {
  def main(args: Array[String]): Unit = {
    val conf = new SparkConf().setAppName("TriangleCounting App")
    val sc = new SparkContext(conf)

    val currentDir = System.getProperty("user.dir")
    val edgeFile = "file://" + currentDir + "/enron.txt"

    val graph = GraphLoader
                .edgeListFile(sc, edgeFile,true)
                .partitionBy(PartitionStrategy.RandomVertexCut)

    // Find number of triangles for each vertex
    val triCounts = graph.triangleCount().vertices

    println(triCounts.collect().mkString("\n"))
  }
}
```

Spark SQL: planets

We have a **JSON** file (**`planets.json`**) containing information about the planets of our solar system

```
{"name":"Mercury","sundist":"57910",  "radius":"2440"}
{"name":"Venus",  "sundist":"108200", "radius":"6052"}
{"name":"Earth",  "sundist":"149600", "radius":"6378"}
{"name":"Mars",   "sundist":"227940", "radius":"3397"}
{"name":"Jupiter","sundist":"778330", "radius":"71492"}
{"name":"Saturn", "sundist":"1429400","radius":"60268"}
{"name":"Uranus", "sundist":"2870990","radius":"25559"}
{"name":"Neptune","sundist":"4504300","radius":"24766"}
{"name":"Pluto",  "sundist":"5913520","radius":"1150"}
```

Spark SQL: planets

The JSON schema looks like this:

```
root
 |-- name: string (nullable = true)
 |-- radius: string (nullable = true)
 |-- sundist: string (nullable =
 true)
```

Spark SQL: planets

We need to do the following:

1. extract the schema from **`planets.json`**
2. load the data
3. execute a SQL query

Spark SQL: planets

```
object Planets {
  def main(args: Array[String]) {

    // Create spark configuration and spark context
    val conf = new SparkConf().setAppName("Planets App")
    val sc = new SparkContext(conf)
    val sqlContext = new org.apache.spark.sql.SQLContext(sc)

    val currentDir = System.getProperty("user.dir")  // get the current directory
    val inputFile = "file://" + currentDir + "/planets.json"

    val planets = sqlContext.jsonFile(inputFile)

    planets.printSchema()
    planets.registerTempTable("planets")

    val smallPlanets = sqlContext.sql("SELECT name,sundist,radius FROM planets WHERE radius < 10000")

    smallPlanets.foreach(println)

    sc.stop()

  }
}
```

Spark SQL: sales

Transaction_Date	Product	Price	Payment_Type	Name	City	State	Country	Account_Created	Last_Login	Latitude	Longitude
1/2/09 6:17	Product1	1200	Mastercard	carolina	Basildon	England	United Kingdom	1/2/09 6:00	1/2/09 6:08	51.5	-1.1166667
1/2/09 4:53	Product1	1200	Visa	Betina	Parkville	MO	United States	1/2/09 4:42	1/2/09 7:49	39.195	-94.68194
1/2/09 13:08	Product1	1200	Mastercard	Federica e Andrea	Astoria	OR	United States	1/1/09 16:21	1/3/09 12:32	46.18806	-123.83
1/3/09 14:44	Product1	1200	Visa	Gouya	Echuca	Victoria	Australia	9/25/05 21:13	1/3/09 14:22	-36.1333333	144.75
1/4/09 12:56	Product2	3600	Visa	Gerd W	Cahaba Heights	AL	United States	11/15/08 15:47	1/4/09 12:45	33.52056	-86.8025
1/4/09 13:19	Product1	1200	Visa	LAURENCE	Mickleton	NJ	United States	9/24/08 15:19	1/4/09 13:04	39.79	-75.23806
1/4/09 20:11	Product1	1200	Mastercard	Fleur	Peoria	IL	United States	1/3/09 9:38	1/4/09 19:45	40.69361	-89.58889
1/2/09 20:09	Product1	1200	Mastercard	adam	Martin	TN	United States	1/2/09 17:43	1/4/09 20:01	36.34333	-88.85028
1/4/09 13:17	Product1	1200	Mastercard	Renee Elisabeth	Tel Aviv	Tel Aviv	Israel	1/4/09 13:03	1/4/09 22:10	32.0666667	34.7666667
1/4/09 14:11	Product1	1200	Visa	Aidan	Chatou	Ile-de-France	France	6/3/08 4:22	1/5/09 1:17	48.8833333	2.15
1/5/09 2:42	Product1	1200	Diners	Stacy	New York	NY	United States	1/5/09 2:23	1/5/09 4:59	40.71417	-74.00639
1/5/09 5:39	Product1	1200	Amex	Heidi	Eindhoven	Noord-Brabant	Netherlands	1/5/09 4:55	1/5/09 8:15	51.45	5.4666667
1/2/09 9:16	Product1	1200	Mastercard	Sean	Shavano Park	TX	United States	1/2/09 8:32	1/5/09 9:05	29.42389	-98.49333
1/5/09 10:08	Product1	1200	Visa	Georgia	Eagle	ID	United States	11/11/08 15:53	1/5/09 10:05	43.69556	-116.35306
1/2/09 14:18	Product1	1200	Visa	Richard	Riverside	NJ	United States	12/9/08 12:07	1/5/09 11:01	40.03222	-74.95778
1/4/09 1:05	Product1	1200	Diners	Leanne	Julianstown	Meath	Ireland	1/4/09 0:00	1/5/09 13:36	53.6772222	-6.3191667
1/5/09 11:37	Product1	1200	Visa	Janet	Ottawa	Ontario	Canada	1/5/09 9:35	1/5/09 19:24	45.4166667	-75.7
1/6/09 5:02	Product1	1200	Diners	barbara	Hyderabad	Andhra Pradesh	India	1/6/09 2:41	1/6/09 7:52	17.3833333	78.4666667
1/6/09 7:45	Product2	3600	Visa	Sabine	London	England	United Kingdom	1/6/09 7:00	1/6/09 9:17	51.52721	0.14559
1/2/09 7:35	Product1	1200	Diners	Hani	Salt Lake City	UT	United States	12/30/08 5:44	1/6/09 10:52	40.76083	-111.89028
1/6/09 12:56	Product1	1200	Visa	Jeremy	Manchester	England	United Kingdom	1/6/09 10:58	1/6/09 13:31	53.5	-2.2166667
1/1/09 11:05	Product1	1200	Diners	Janis	Ballynora	Cork	Ireland	12/10/07 12:37	1/7/09 1:52	51.8630556	-8.58
1/5/09 4:10	Product1	1200	Mastercard	Nicola	Roodepoort	Gauteng	South Africa	1/5/09 2:33	1/7/09 5:13	-26.1666667	27.8666667
1/6/09 7:18	Product1	1200	Visa	asuman	Chula Vista	CA	United States	1/6/09 7:07	1/7/09 7:08	32.64	-117.08333
1/2/09 1:11	Product1	1200	Mastercard	Lena	Kuopio	Ita-Suomen Laani	Finland	12/31/08 2:48	1/7/09 10:20	62.9	27.6833333
1/1/09 2:24	Product1	1200	Visa	Lisa	Sugar Land	TX	United States	1/1/09 1:56	1/7/09 10:52	29.61944	-95.63472
1/7/09 8:08	Product1	1200	Diners	Bryan Kerrene	New York	NY	United States	1/7/09 7:39	1/7/09 12:38	40.71417	-74.00639
1/2/09 2:57	Product1	1200	Visa	chris	London	England	United Kingdom	1/3/08 7:23	1/7/09 13:14	51.52721	0.14559
1/1/09 20:21	Product1	1200	Visa	Maxine	Morton	IL	United States	10/24/08 6:48	1/7/09 20:49	40.61278	-89.45917
1/8/09 0:42	Product1	1200	Visa	Family	Los Gatos	CA	United States	1/8/09 0:28	1/8/09 3:39	37.22667	-121.97361
1/8/09 3:56	Product1	1200	Mastercard	Katherine	New York	NY	United States	1/8/09 3:33	1/8/09 6:19	40.71417	-74.00639
1/8/09 3:16	Product1	1200	Mastercard	Linda	Miami	FL	United States	1/8/09 3:06	1/8/09 6:34	25.77389	-80.19389
1/8/09 1:59	Product1	1200	Mastercard	SYLVIA	Vesenaz	Geneve	Switzerland	11/28/07 11:56	1/8/09 7:20	46.2333333	6.2
1/3/09 9:03	Product1	1200	Diners	Sheila	Brooklyn	NY	United States	1/3/09 8:47	1/8/09 10:38	40.65	-73.95

Spark SQL: sales

```
val basicDF =  ss.read.option("header", "true").csv(“sales.csv”)
basicDF.printSchema()
basicDF.show()
```

Spark SQL: sales

```
// Define some helpful User-Defined Functions (UDFs)
 val udf_toDouble = udf[Double, String]( _.toDouble)
 val udf_toInt = udf[Int, String]( _.toInt)

 // Convert columns to the appropriate data type.
 val modifiedDF = basicDF
      .withColumn("Transaction_Date", to_timestamp($"Transaction_Date", "mm/dd/yy HH:mm"))
      .withColumn("Account_Created", to_timestamp($"Account_Created", "mm/dd/yy"))
      .withColumn("Last_Login", to_timestamp($"Last_Login", "mm/dd/yy"))
      .withColumn("Latitude", udf_toDouble($"Latitude"))
      .withColumn("Longitude", udf_toDouble($"Longitude"))
      .withColumn("City", rtrim($"City"))

modifiedDF.printSchema()
modifiedDF.show()
```

Spark SQL: sales

```
// Show only LAT-LON of sales.
modifiedDF.select($"Latitude",  $"Longitude").show()

// Show number of sales per country and also average price of sold products per country (use group by).

modifiedDF.groupBy($"Country").agg(count("Transaction_Date"),avg("Price")).show()

// Show number of sales per Country only for VISA payments (use filter).
modifiedDF.filter($"Payment_Type" === "Visa").groupBy($"Country").agg(count("Transaction_Date")).show()
```

Spark SQL: sales

Another way to write an SQL query

```
// First, register the DF as a temporary view with the name "sales".
 modifiedDF.createOrReplaceTempView("sales")

// Now, you can use the sqlContext to run SQL queries, since "sales" is like a dtabase table.
// Get all sales for France.
modifiedDF.sqlContext.sql("SELECT * FROM sales WHERE Country='France'").show()

// Get all pairs of sales using Transaction_Date as the key, to detect pairs of sales that correspond to the
// same price. This is equivalent to a self-join on column "Price".
modifiedDF
      .sqlContext
      .sql("SELECT r.Transaction_Date, s.Transaction_Date FROM sales as r, sales as s WHERE r.Price =
   s.Price")
      .show()
```

Python Examples

WordCount in Python

```
from __future__ import print_function
import sys
from operator import add
from pyspark import SparkContext

if __name__ == "__main__":
    if len(sys.argv) != 2:
        print("Usage: wordcount <file>", file=sys.stderr)
        exit(-1)
    sc = SparkContext(appName="PythonWordCount")
    lines = sc.textFile(sys.argv[1], 10)
    counts = lines.flatMap(lambda x: x.split(' ')) \
                  .map(lambda x: (x, 1)) \
                  .reduceByKey(add)
    output = counts.collect()
    for (word, count) in output:
        print("%s: %i" % (word, count))

    sc.stop()
```

kMeans in Python

```
from __future__ import print_function
import sys
import numpy as np
from pyspark import SparkContext
from pyspark.mllib.clustering import KMeans

def parseVector(line):
    return np.array([float(x) for x in line.split(' ')])

if __name__ == "__main__":
    if len(sys.argv) != 3:
        print("Usage: kmeans <file> <k>", file=sys.stderr)
        exit(-1)
    sc = SparkContext(appName="KMeans")
    lines = sc.textFile(sys.argv[1], 10)
    data = lines.map(parseVector)
    k = int(sys.argv[2])
    model = KMeans.train(data, k)
    print("Final centers: " + str(model.clusterCenters))
    print("Total Cost: " + str(model.computeCost(data)))
    sc.stop()
```

Some Spark Users

Resources

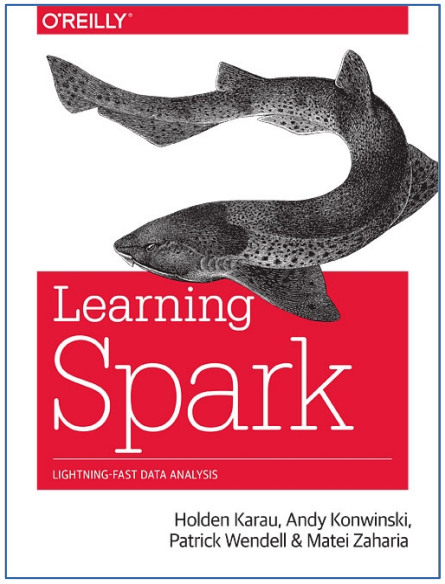
The best way to begin learning Spark is to study the material at Spark's official website

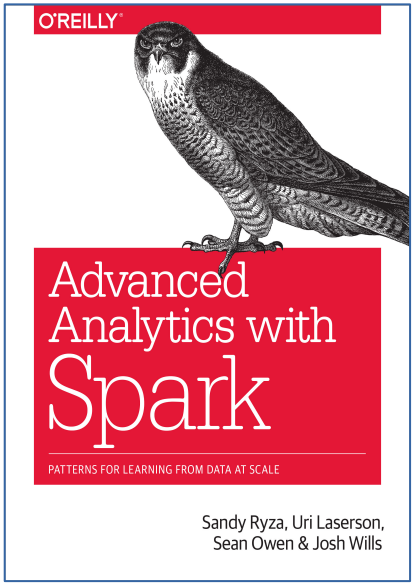
https://spark.apache.org

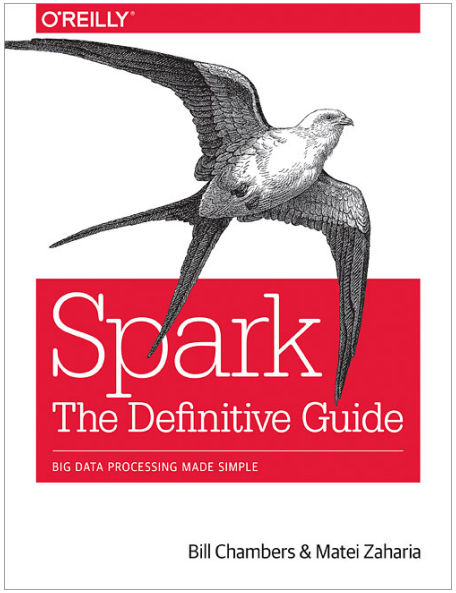
From this website you have access to **Spark Summits and other events** which contain useful video lectures for all Spark components.

Spark Books

Books to learn Spark







Useful MOOCs

Coursera (www.coursera.org):

Introduction to Big Data

Big Data Analysis with Scala and Spark

Data Manipulation at Scale, Systems and Algorithms

edX (www.edx.org):

Introduction to Apache Spark

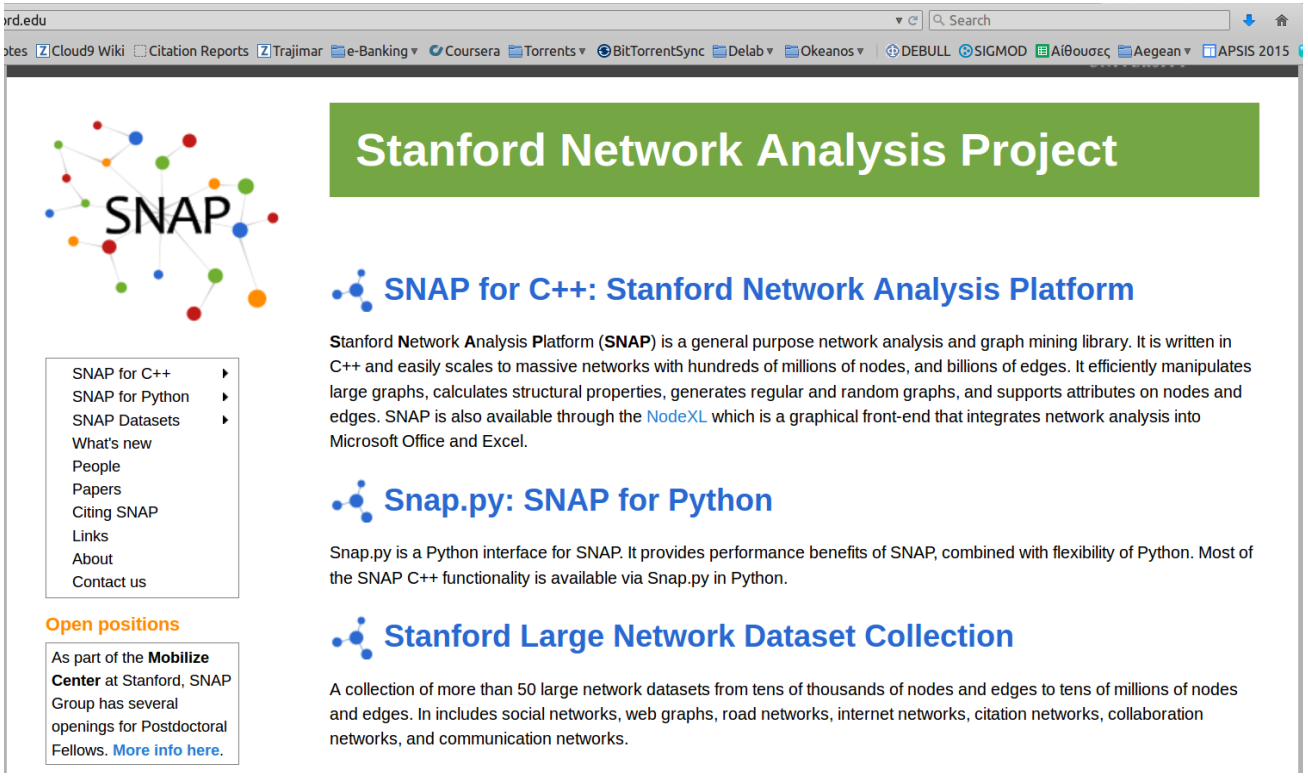
Distributed Machine Learning with Apache Spark

Big Data Analysis with Apache Spark

Dataset Download

Where to find more graph data ?

Take a look at
http://snap.stanford.edu



Thank you

Questions ?