

chiPSet Training School, September 2017, Novi Sad, Serbia



# Big Data Analytics with Apache



Apostolos N. Papadopoulos (Associate Prof.)  
([papadopo@csd.auth.gr](mailto:papadopo@csd.auth.gr), <http://datalab.csd.auth.gr/~apostol>)  
Data Science & Engineering Lab  
Department of Informatics, Aristotle University of Thessaloniki, GREECE

# Outline

What is Spark?

Basic features

Resilient Distributed Datasets (RDDs) and DataFrames

Existing libraries

Examples in Scala & Python

Further reading

# FUNDAMENTAL CONCEPTS

Let there  
be  
SPARK !

# What is Spark ?

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 **2.2.0 (July 11, 2017)**  
previous release **2.1.1 (March 2, 2017)**

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

<b>1.0.0</b>	- May 30, 2014	<b>1.4.1</b>	- July 15, 2015	<b>2.1.0</b>	- December 28, 2016
<b>1.0.1</b>	- July 11, 2014	<b>1.5.0</b>	- September 9, 2015	<b>2.1.1</b>	- March 2, 2017
<b>1.0.2</b>	- August 5, 2014	<b>1.5.1</b>	- October 2, 2015		
<b>1.1.0</b>	- September 11, 2014	<b>1.5.2</b>	- November 9, 2015		
<b>1.1.1</b>	- November 26, 2014	<b>1.6.0</b>	- January 4, 2016		
<b>1.2.0</b>	- December 18, 2014	<b>1.6.1</b>	- March 9, 2016		
<b>1.2.1</b>	- February 9, 2015	<b>1.6.2</b>	- June 25, 2016		
<b>1.3.0</b>	- March 13, 2015	<b>2.0.0</b>	- July 26, 2016		
<b>1.3.1</b>	- April 17, 2015	<b>2.0.1</b>	- October 3, 2016		
<b>1.4.0</b>	- June 11, 2015	<b>2.0.2</b>	- November 14, 2016		

# 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

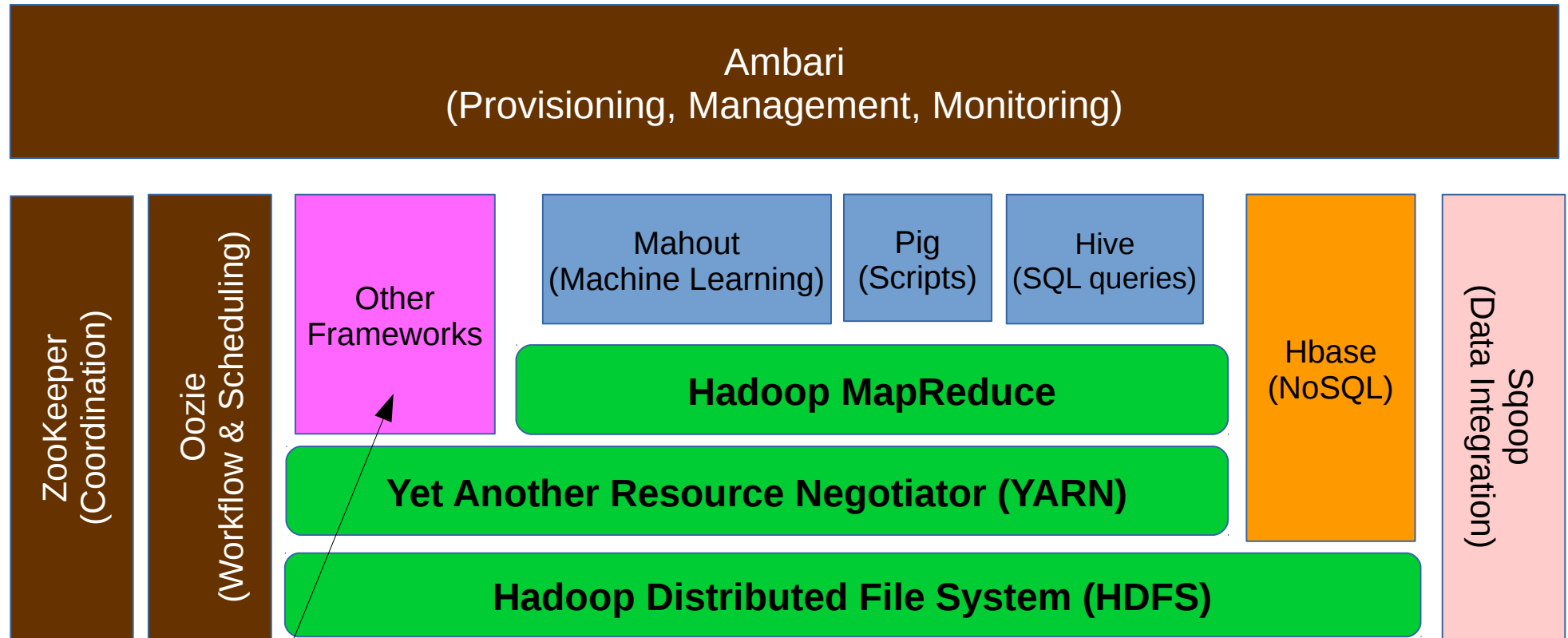
He also co-designed the MESOS cluster manager and he contributed to Hadoop fair scheduler.

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



# Spark Basics

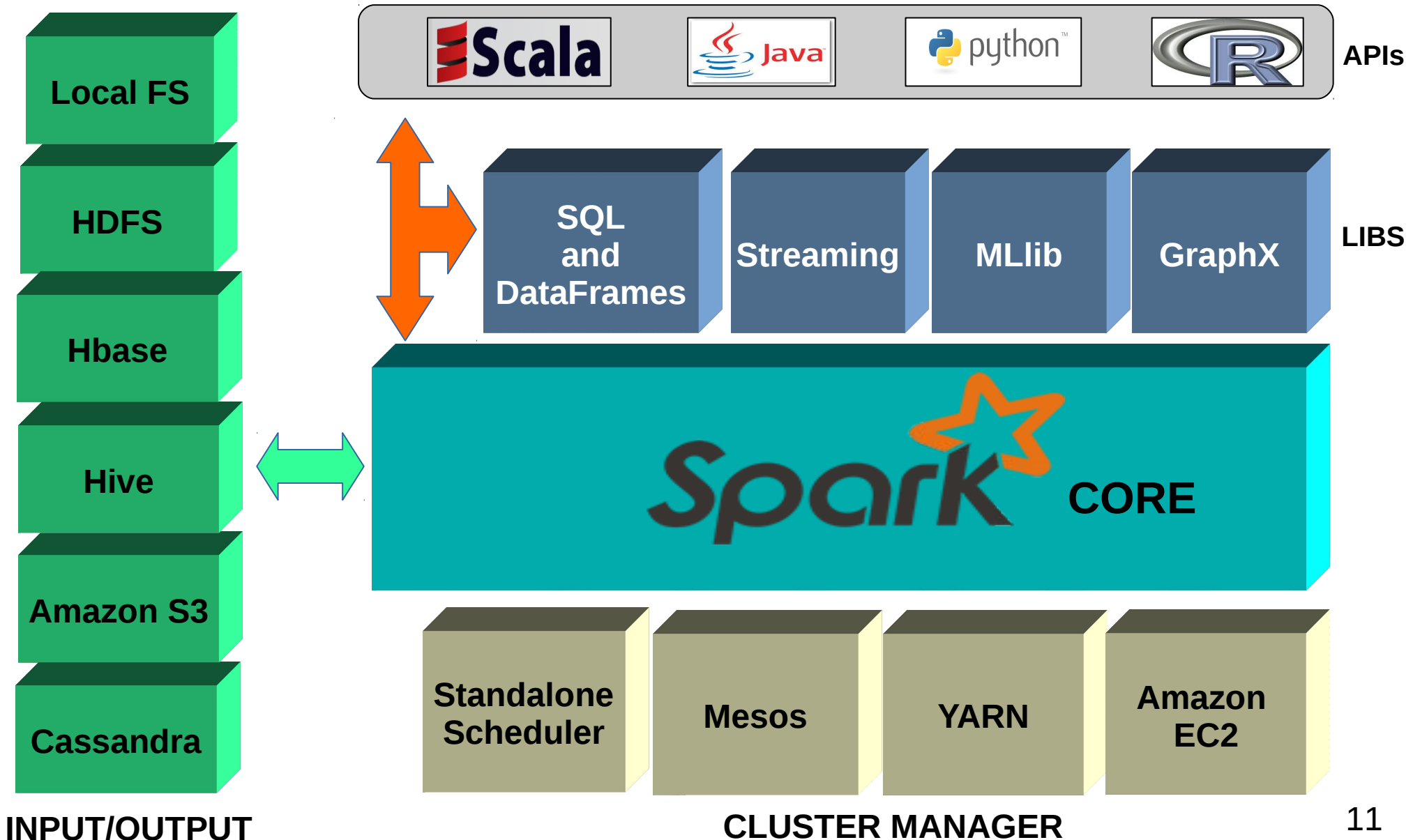
Spark is designed to be **fast** and **general purpose**.

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)
- 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 structured 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 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**.

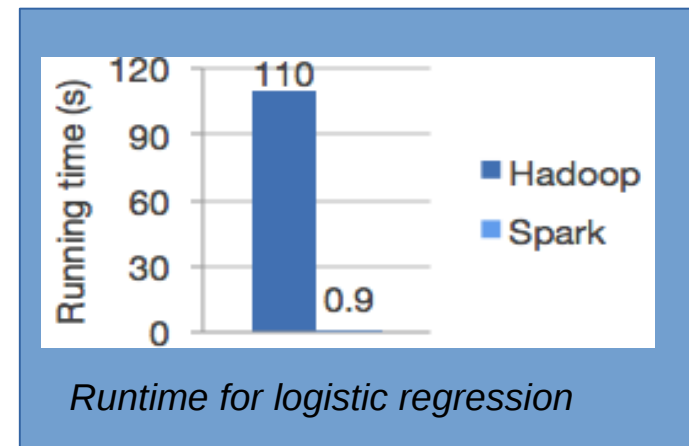
# Spark MLlib & ML

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

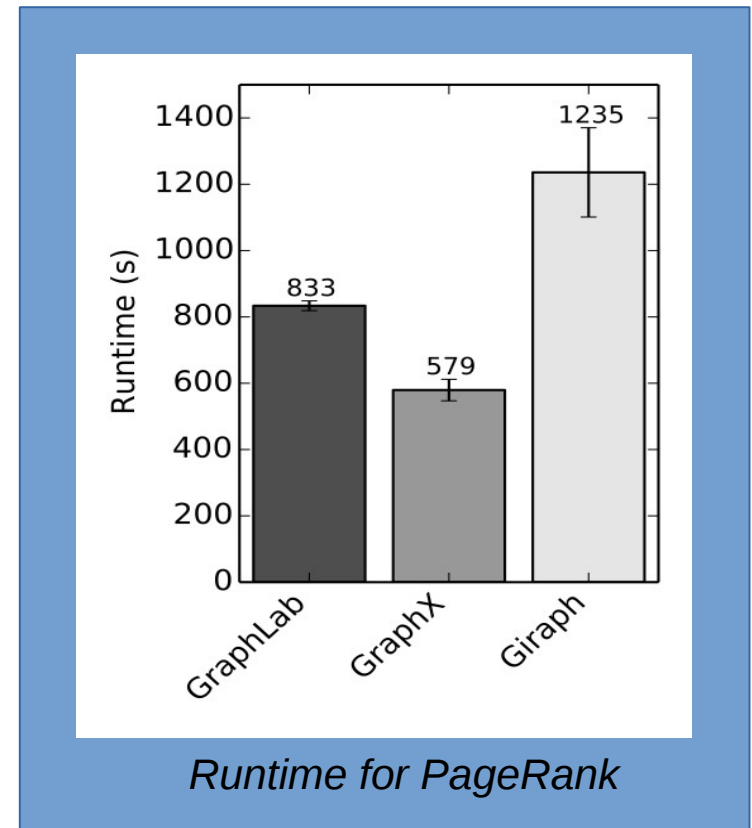


# Spark GraphX

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
- ...



# RESILIENT DISTRIBUTED DATASETS

The  
RDD  
API



# 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`, `groupByKey`, `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"
```

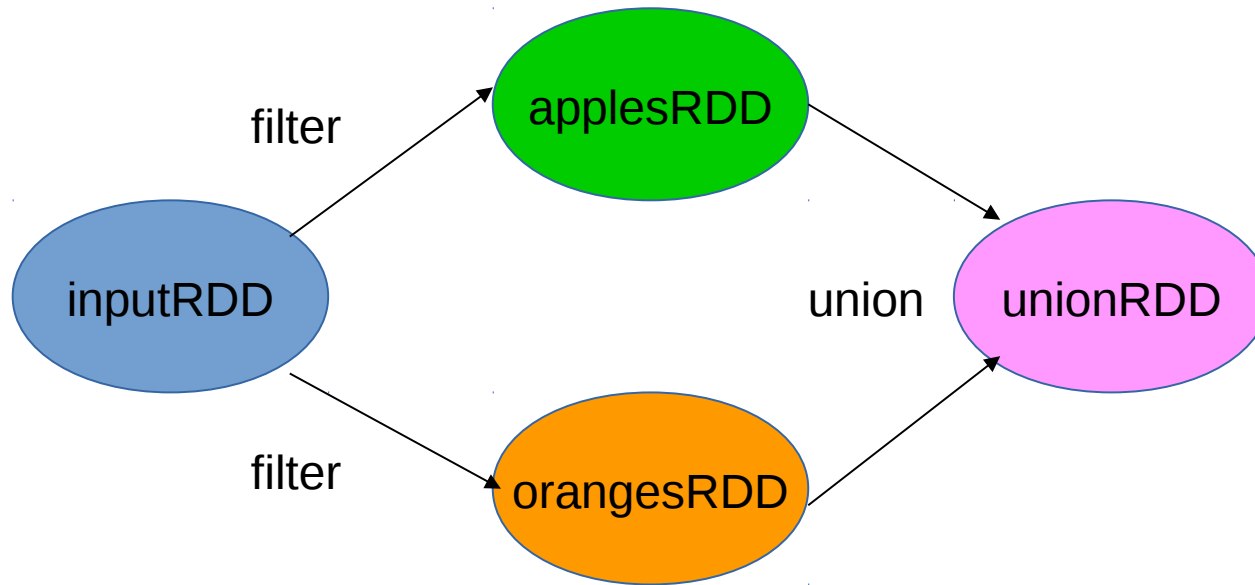
```
val orangesRDD = inputRDD.filter(x => x.contains("orange"))
```

```
// perform the union
```

```
val aoRDD = applesRDD.union(orangesRDD)
```

# RDD Operations: transformations

Graphically speaking:



# RDD Operations: actions

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

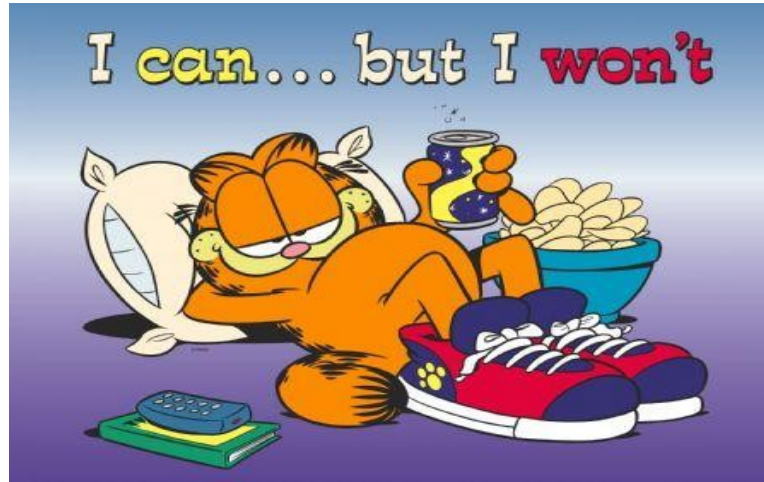
We use the action `count()` to find the number of lines in `unionRDD` containing apples or oranges (or both) and then we print the 5 first lines using the action `take()`

```
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 the list{1, 2, 3}.

<b>map()</b>	<code>rdd.map(x =&gt; x + 2)</code>	{3, 4, 5}
<b>flatMap()</b>	<code>rdd.flatMap(x =&gt; List(x-1, x, x+1))</code>	{0, 1, 2, 1, 2, 3, 2, 3, 4}
<b>filter()</b>	<code>rdd.filter(x =&gt; x&gt;1)</code>	{2, 3}
<b>distinct()</b>	<code>rdd.distinct()</code>	{1, 2, 3}
<b>sample()</b>	<code>rdd.sample(false, 0.2)</code>	non-predictable

# Two-RDD Transformations

These transformations require two RDDs

**union()**                      `rdd.union(another)`

**intersection()** `rdd.intersection(another)`

**subtract()**                `rdd.subtract(another)`

**cartesian()**            `rdd.cartesian(another)`

# Some Actions

<b>collect()</b>	<code>rdd.collect()</code>	<code>{1, 2, 3}</code>
<b>count()</b>	<code>rdd.count()</code>	<code>3</code>
<b>countByValue()</b>	<code>rdd.countByValue()</code>	<code>{(1, 1), (2, 1), (3, 1)}</code>
<b>take()</b>	<code>rdd.take(2)</code>	<code>{1, 2}</code>
<b>top()</b>	<code>rdd.top(2)</code>	<code>{3, 2}</code>
<b>reduce()</b>	<code>rdd.reduce((x, y) =&gt; x+y)</code>	<code>6</code>
<b>foreach()</b>	<code>rdd.foreach(func)</code>	

# **DIRECTED ACYCLIC GRAPHS**

Time  
to Run  
the Job

# RDDs and DAGs

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

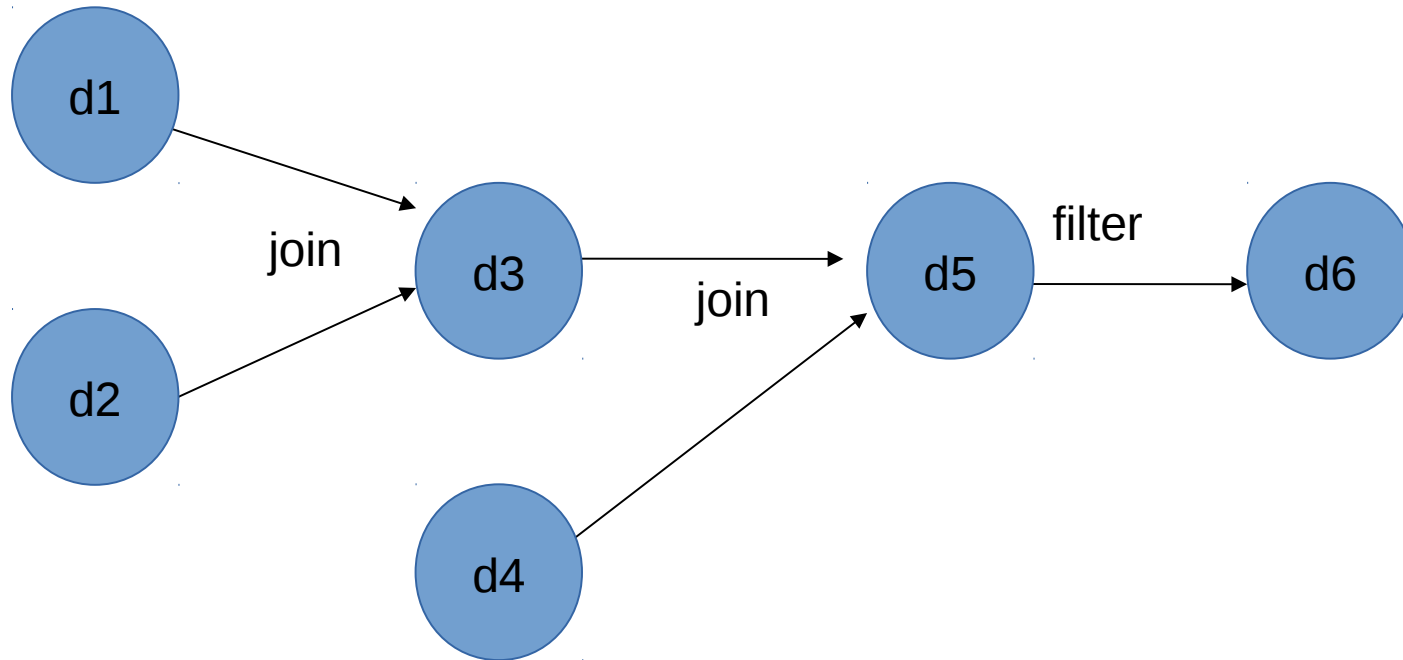
Input: RDD and partitions to compute

Output: output from actions on those partitions

Roles:

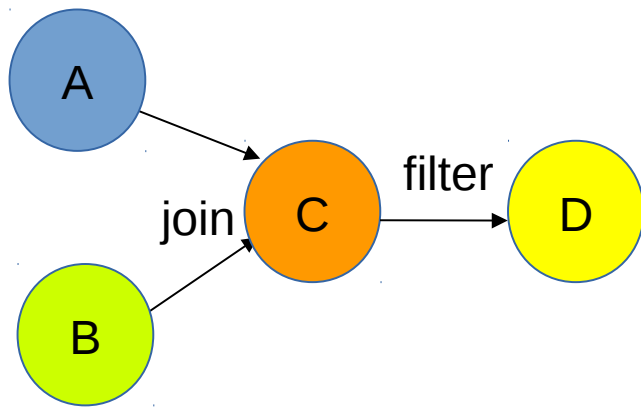
- > Build stages of tasks
- > Submit them to lower level scheduler (e.g. YARN, Mesos, Standalone) as ready
- > Lower level scheduler will schedule data based on locality
- > Resubmit failed stages if outputs are lost

# DAG Scheduling

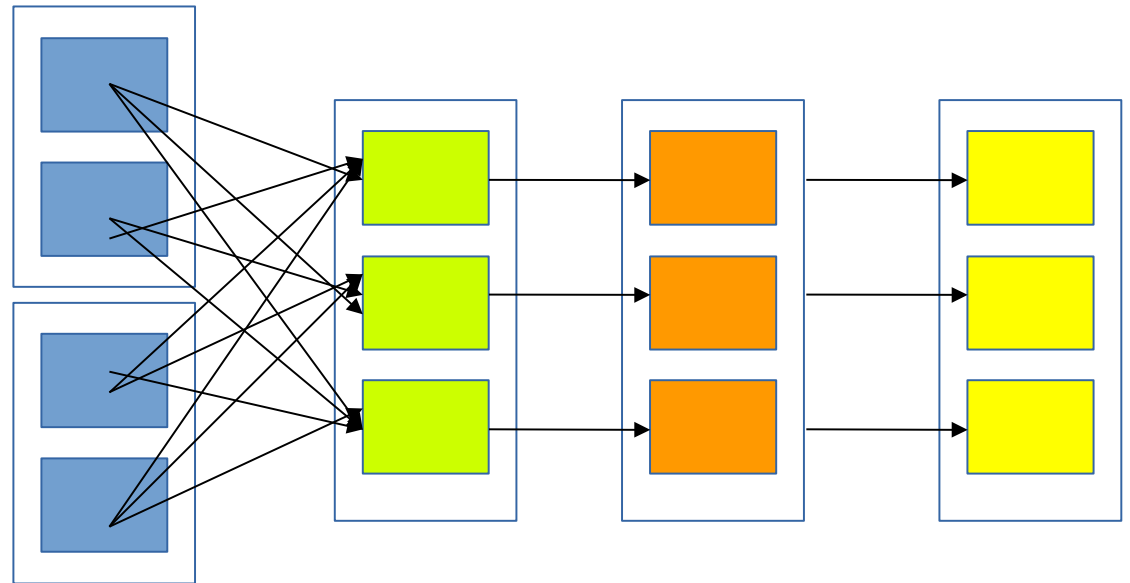




# DAG Scheduling



RDD objects



DAG scheduler

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

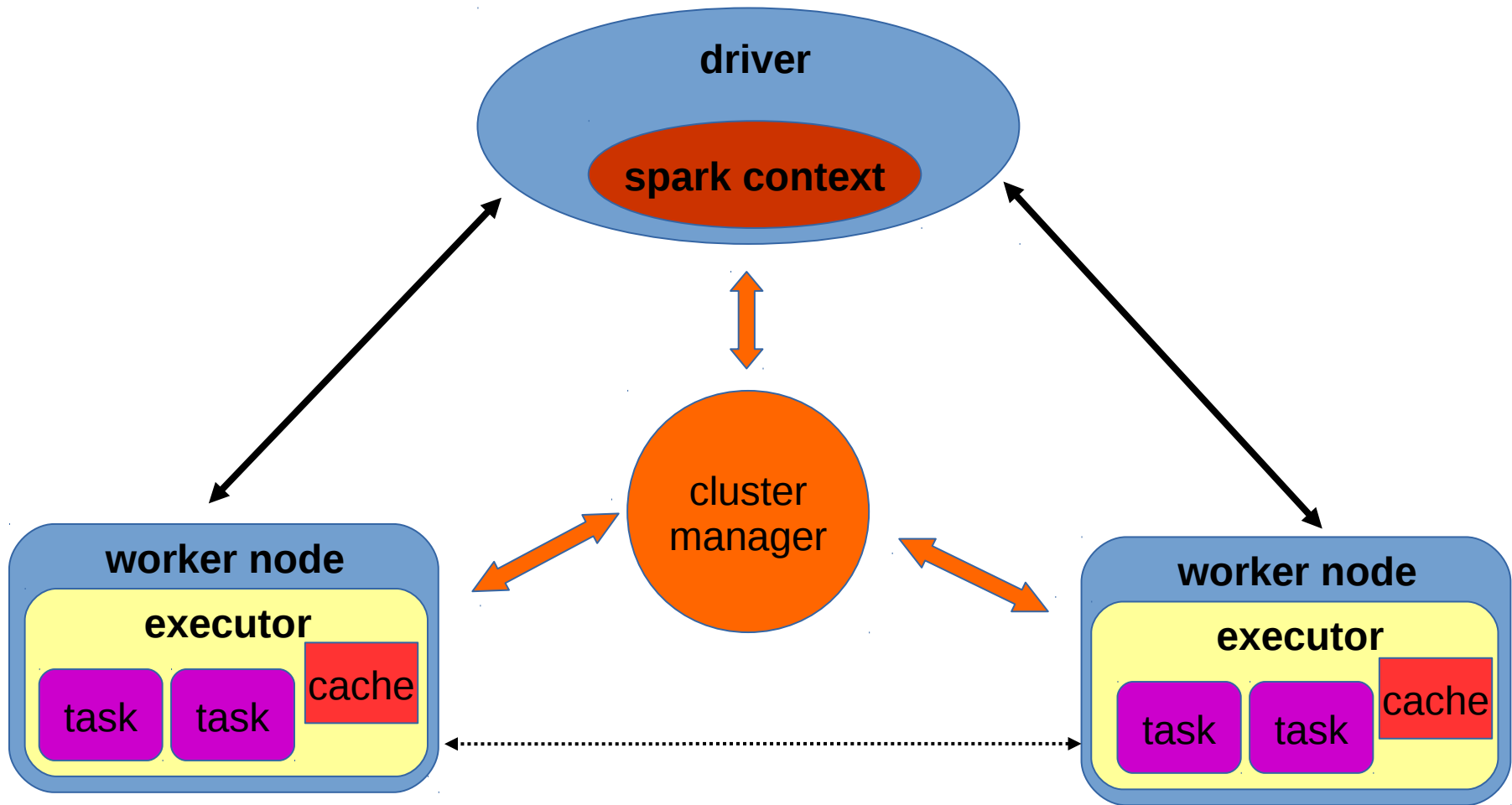
split graph into stages of tasks  
submit each stage

# 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



**PERSISTENCE**

stay  
where  
you are !

# 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 too many things in RAM Spark starts flushing data to disk using a Least Recently Used policy.

# **BROADCAST VARIABLES AND ACCUMULATORS**

Increasing  
Efficiency

# 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.

**DATAFRAMES**

The  
DATAFRAME  
API

# 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 from 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.sql("SELECT name, age FROM people  
WHERE age BETWEEN 13 AND 19")
```

# Spark Simple Examples

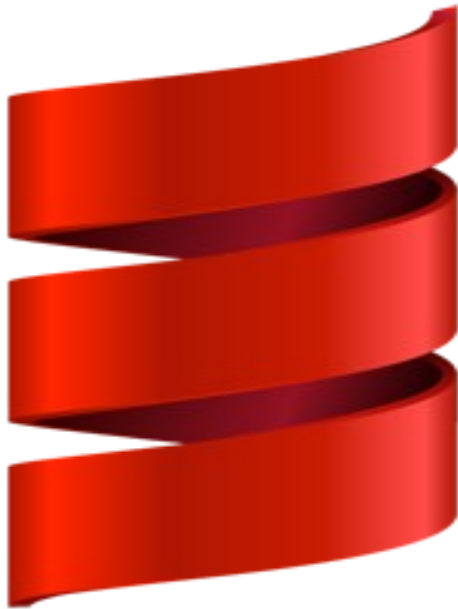
Spark supports

- ✓ 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

# Hello Spark

things we  
must  
import



```
import org.apache.spark.SparkContext
import org.apache.spark.SparkContext._
import org.apache.spark.SparkConf

object HelloSpark {
  def main(args: Array[String]): Unit = {
    println("Hello, Spark!")
  }
}
```

# 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") // config
    val sc = new SparkContext(sparkConf) // create spark context

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

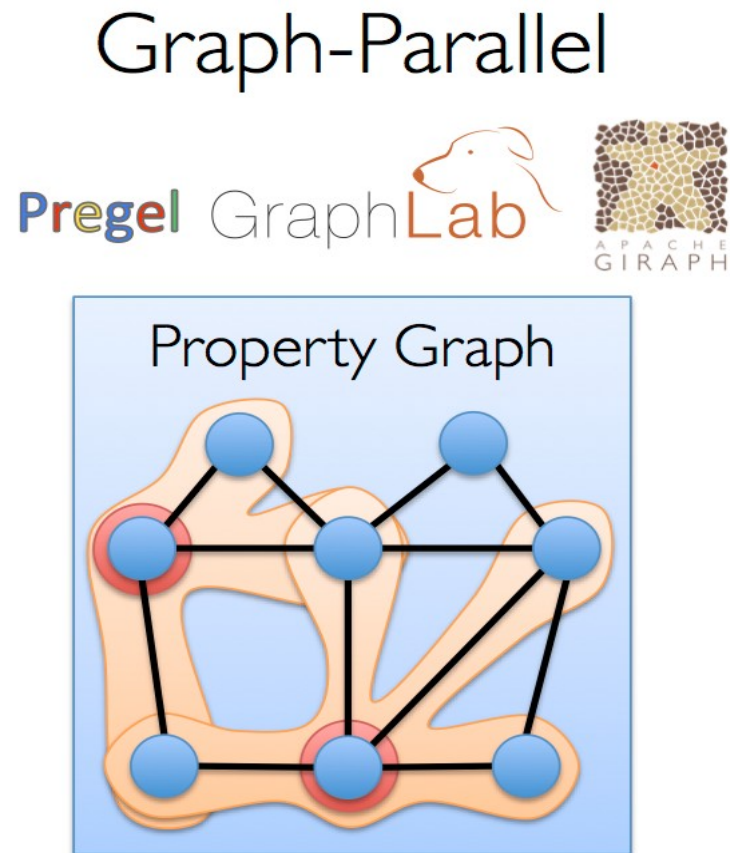
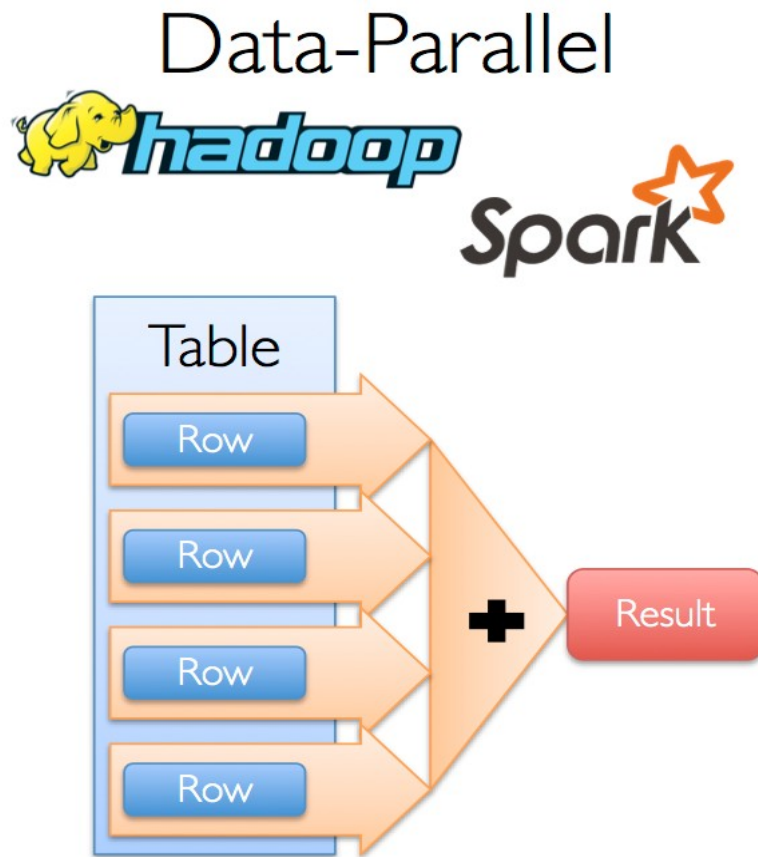
The basic concept in GraphX is the property graph

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”*

# More on GraphX

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

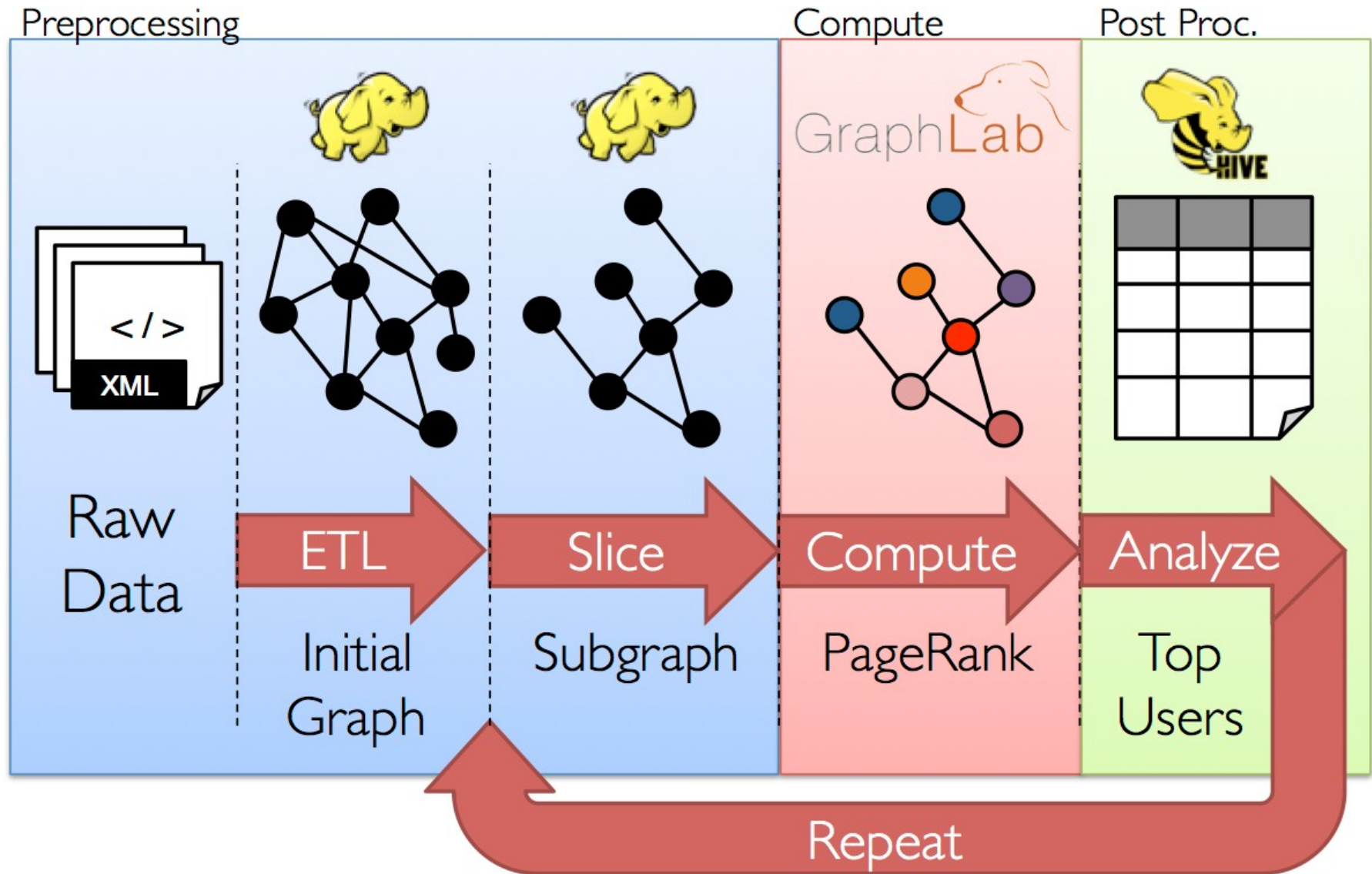
Ex:

PageRank on Live-Journal network (available @snap)

GraphLab is 60 times faster than Hadoop

GraphLab is **16 times** faster than Spark

# More on GraphX



# More on GraphX

To use GraphX we need to import

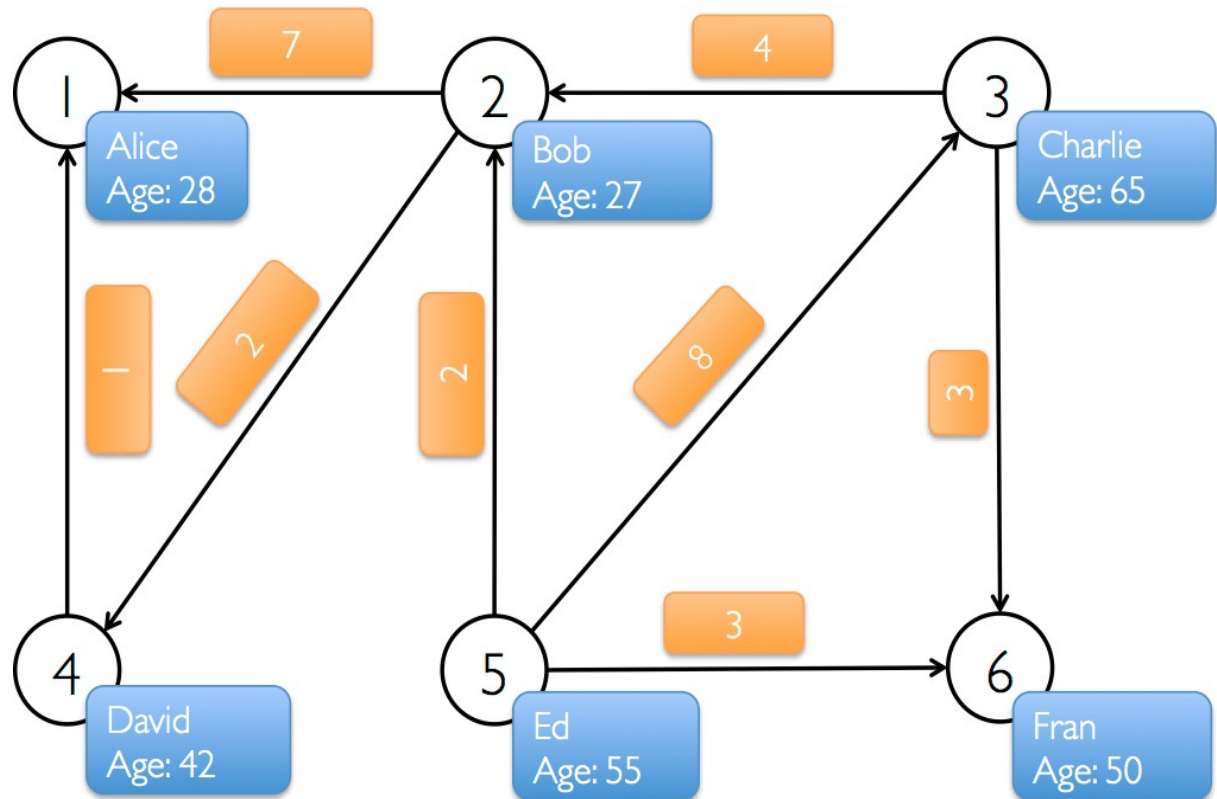
```
import org.apache.spark._
```

```
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

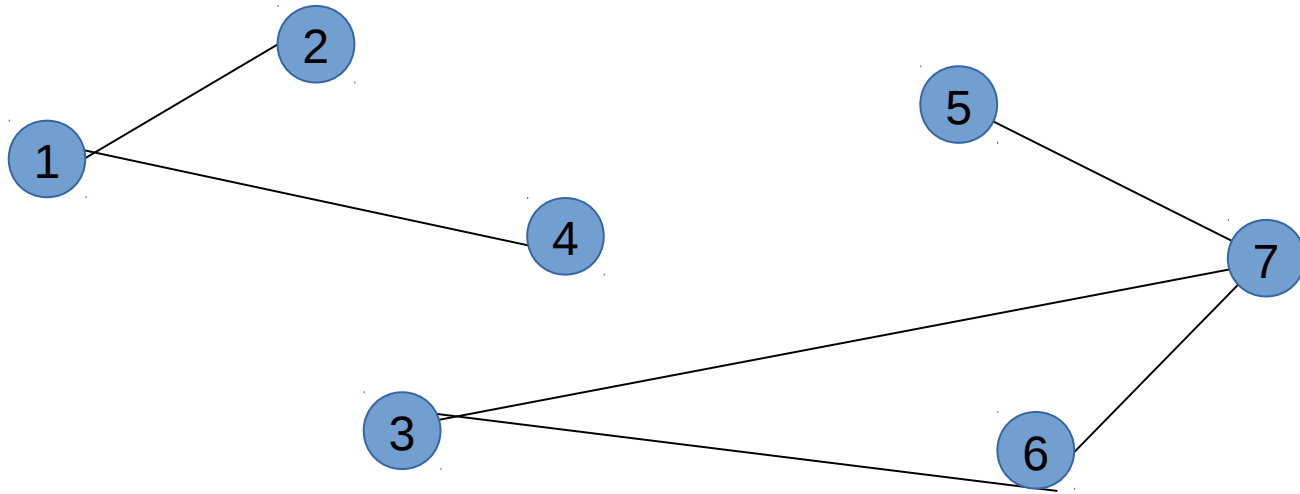
```
val graph: Graph[(String, Int), Int]  
    = Graph(vertexRDD, edgeRDD)
```

# 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

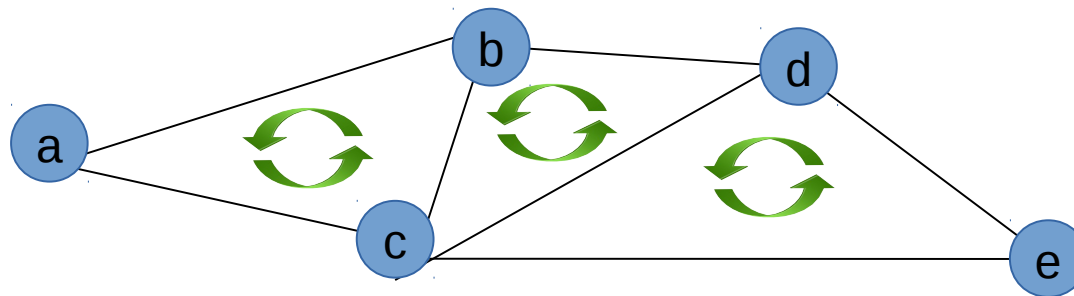
```
object ConnectedComponents {  
  
  def main(args: Array[String]): Unit = {  
    val conf = new SparkConf().setAppName("ConnectedComponents  
App")  
    val sc = new SparkContext(conf)  
  
    val currentDir = System.getProperty("user.dir")  
    val edgeFile = "file://" + currentDir + "/graph.txt"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

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 Example

The JSON schema looks like this:

```
root
```

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

# Spark SQL Example

We need to do the following:

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

# Spark SQL Example

```
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()  
  
  }  
}
```





# 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

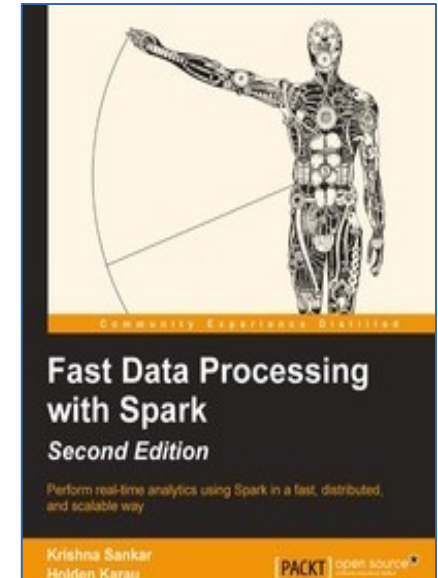
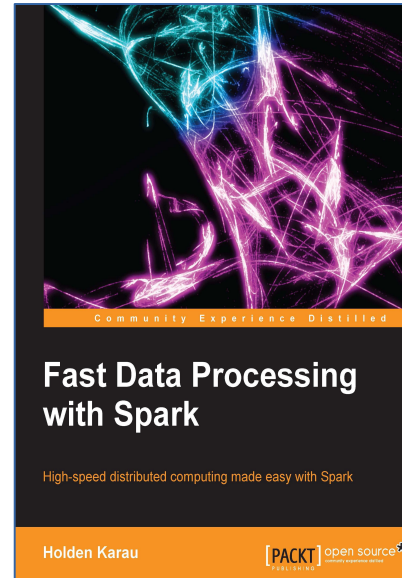
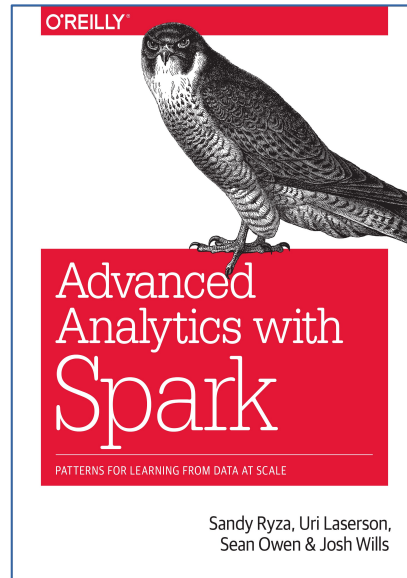
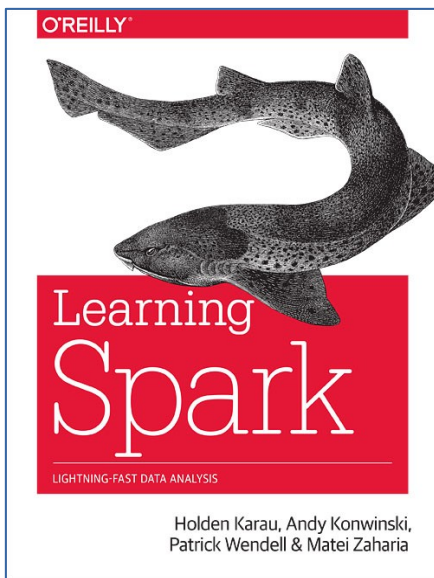
The best way to begin learning Spark is to study the material at Spark's official website

<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.

# Books

## Books to learn Spark



# Useful MOOCs

## **Coursera ([www.coursera.org](http://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](http://www.edx.org)):**

Introduction to Apache Spark

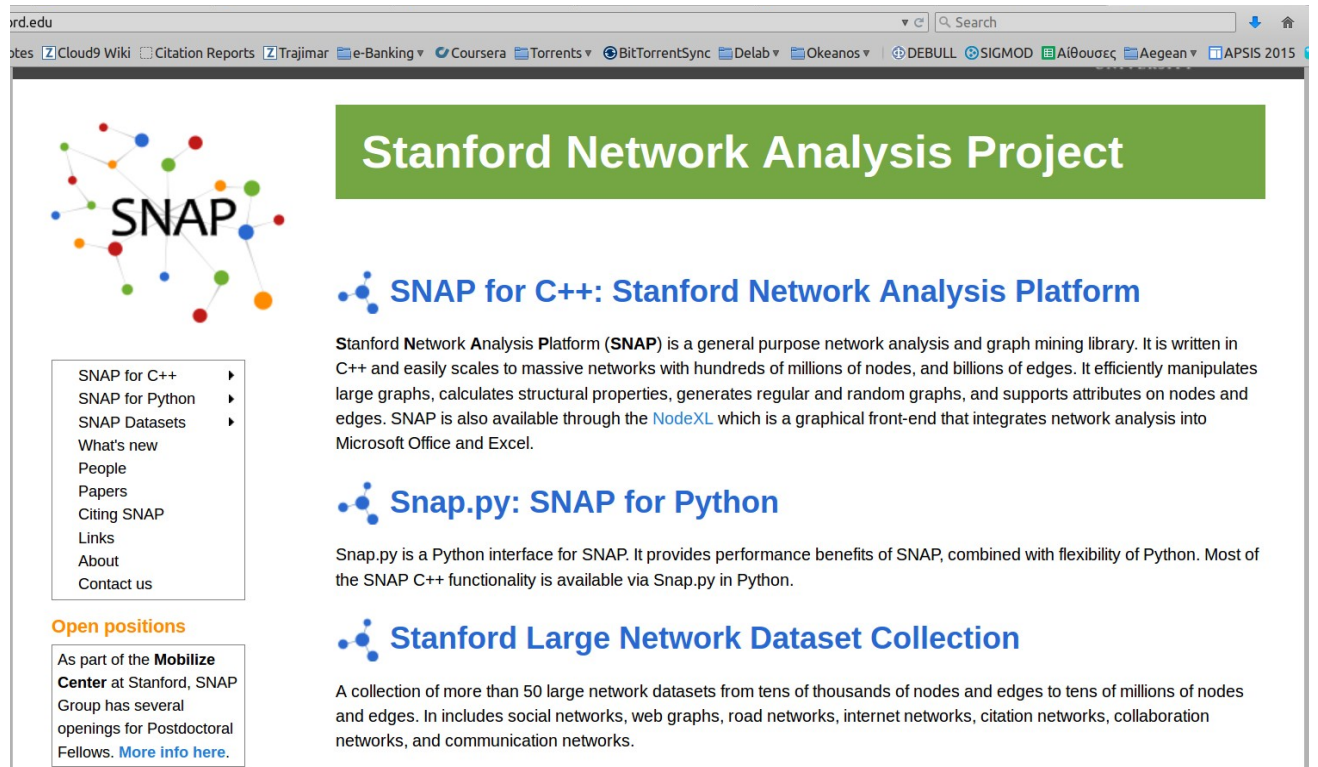
Distributed Machine Learning with Apache Spark

Big Data Analysis with Apache Spark

# Dataset Download

Where to find more graph data ?

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



rd.edu

Cloud9 Wiki Citation Reports Trajimar e-Banking Coursera Torrents BitTorrentSync Delab Okeanos DEBULL SIGMOD ΑΙΘΟΥΣΑ Aegean APSIS 2015

**Stanford Network Analysis Project**

**SNAP for C++: Stanford Network Analysis Platform**

Stanford Network Analysis Platform (SNAP) is a general purpose network analysis and graph mining library. It is written in C++ and easily scales to massive networks with hundreds of millions of nodes, and billions of edges. It efficiently manipulates large graphs, calculates structural properties, generates regular and random graphs, and supports attributes on nodes and edges. SNAP is also available through the [NodeXL](#) which is a graphical front-end that integrates network analysis into Microsoft Office and Excel.

**Snap.py: SNAP for Python**

Snap.py is a Python interface for SNAP. It provides performance benefits of SNAP, combined with flexibility of Python. Most of the SNAP C++ functionality is available via Snap.py in Python.

**Stanford Large Network Dataset Collection**

A collection of more than 50 large network datasets from tens of thousands of nodes and edges to tens of millions of nodes and edges. It includes social networks, web graphs, road networks, internet networks, citation networks, collaboration networks, and communication networks.

SNAP

- SNAP for C++
- SNAP for Python
- SNAP Datasets
- What's new
- People
- Papers
- Citing SNAP
- Links
- About
- Contact us

**Open positions**

As part of the **Mobilize Center** at Stanford, SNAP Group has several openings for Postdoctoral Fellows. [More info here.](#)



# Thank you

Questions ?