Introduction to Apache Spark

Fernando Cores

Index

- 3.1. What is Spark?
- 3.2. The Spark Programming model
- 3.3. Working with Resilient Distributed Datasets (RDDs)
- 3.4. Programming with Spark

How to access to Spark (I)

- We are going to use the Spark installation in the Pirgi big-data cluster (pirgi.udl.cat)
- You can work with Spark in two different ways:
  - Using Jupyter Notebooks through a web-brower in the following page:
    - http://pirgi.udl.cat:8000

    You have to login with your pirgi's account.
  - Using pyspark shell or launching yarn spark jobs:
    - You have to connect to the pirgi cluster using ssh:
      - ssh username@pirgi.udl.cat
    - and execute the pyspark shell:
      - pyspark --master yarn-client --driver-memory 1g --executor-memory 1g --executor-cores 1
    - or launch the spark yarn job:
      - spark-submit --master yarn --deploy-mode cluster --driver-memory 1g --executor-memory 512m --executor-cores 1 ./WordCount.py

How to access to Spark (Docker I)

- You can also use a Jupyter Spark image: jupyter/all-spark-notebook (5.26 GB)
  - Spark 2.0.2 with Hadoop 2.7, Jupyter Notebook 4.3.x, Python 3.x and Python 2.7.x environments
  - Available: https://github.com/jupyter/docker-stacks/tree/master/all-spark-notebook
- Pull the image:
  - docker pull jupyter/all-spark-notebook latest
- Runing the docker machine:
  - docker run -it --rm -p 8888:8888 –v<local_directory>:/home/jovyan/work -w /home/jovyan/work jupyter/all-spark-notebook
- Login on Web browser with the generated token:

  http://localhost:8888/?token=931d864750f56f8ec301e9ed4525a4665f8869a9eb718ecc

How to access to Spark (Docker II)

- You can also use a docker standalone Apache Spark (2.88 GB)
  - Hadoop 2.6.0 and Apache Spark v1.6.0 on Centos
  - Available: https://hub.docker.com/r/sequenceiq/spark/
- Pull the image:
  - docker pull sequenceiq/spark:1.6.0
- Running the docker machine:
  - docker run -it -p 8088:8088 -p 8042:8042 -h sandbox sequenceiq/spark:1.6.0 bash

What is Spark?

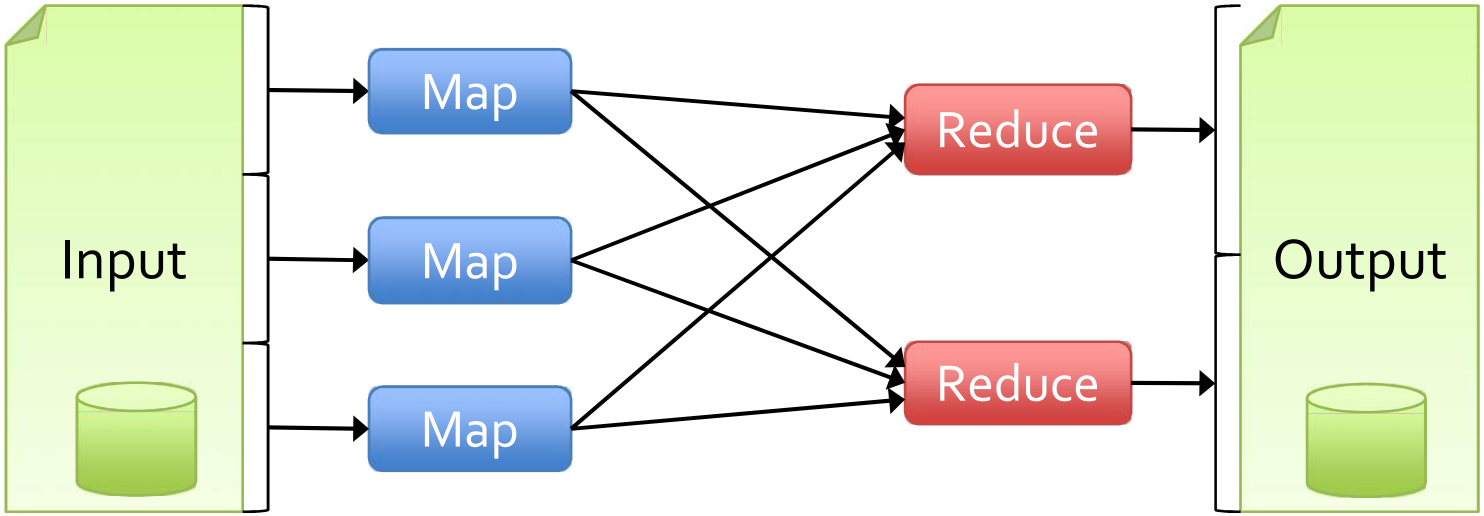
What is Spark?

- An Apache Foundation **open source** project; not a product
- Enables highly **iterative** analysis on **massive** volumes of data at scale
- An **in-memory computing** engine that works with **distributed** data; not a data store
- **Unified environment** for data scientists, developers and data engineers
- Radically simplifies the process of developing **intelligent apps** fuelled by data

Spark Motivation

Current popular programming models for clusters transform data flowing from stable storage to stable storage.

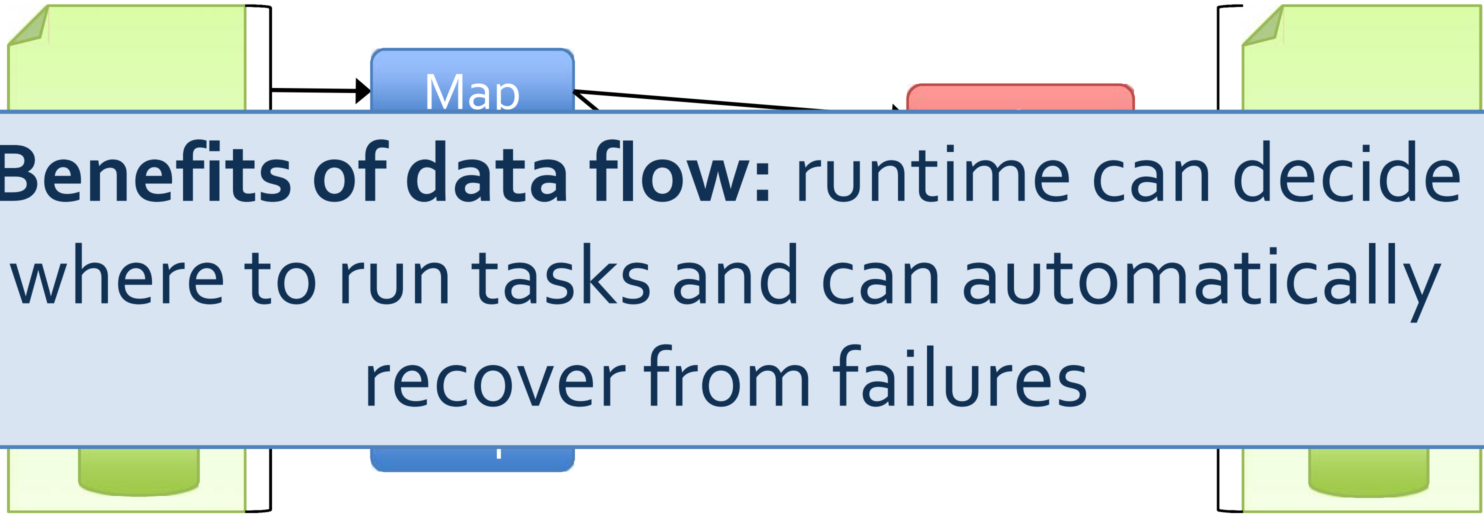
Example: MapReduce:



Spark Motivation

Current popular programming models for clusters transform data flowing from stable storage to stable storage.

Example: MapReduce:



Benefits of data flow: runtime can decide where to run tasks and can automatically recover from failures

Spark Motivation

- Acyclic data flow is a powerful abstraction, but is not efficient for applications that repeatedly reuse a *working set* of data:
  - **Iterative** algorithms (many in machine learning)
  - **Interactive** data mining tools (R, Excel, Python)
- Spark makes working sets a first-class concept to efficiently support these apps

Spark Goal

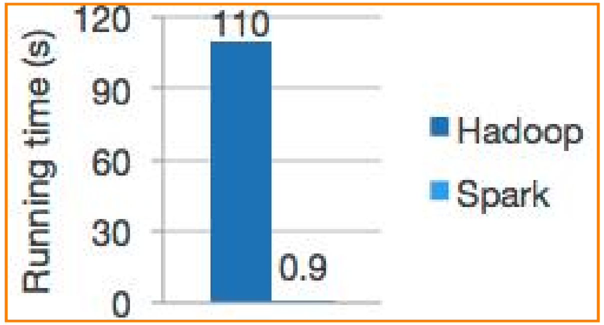
- Provide distributed memory abstractions for clusters to support apps with working sets
- Retain the attractive properties of MapReduce:
  - Fault tolerance (for crashes & stragglers)
  - Data locality
  - Scalability

Solution: augment data flow model with "resilient distributed datasets" (RDDs)

Apache Spark is...

Fast

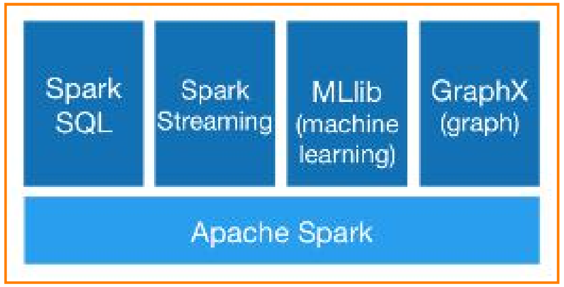
- Leverages aggressively cached in-memory distributed computing and JVM threads
- Faster than MapReduce for some workloads



Logistic regression in Hadoop and Spark

Ease of use (for programmers)

- Written in Scala, an object-oriented, functional programming language
- Scala, Python and Java APIs
- Scala and Python interactive shells
- Runs on Hadoop, Mesos, standalone or cloud

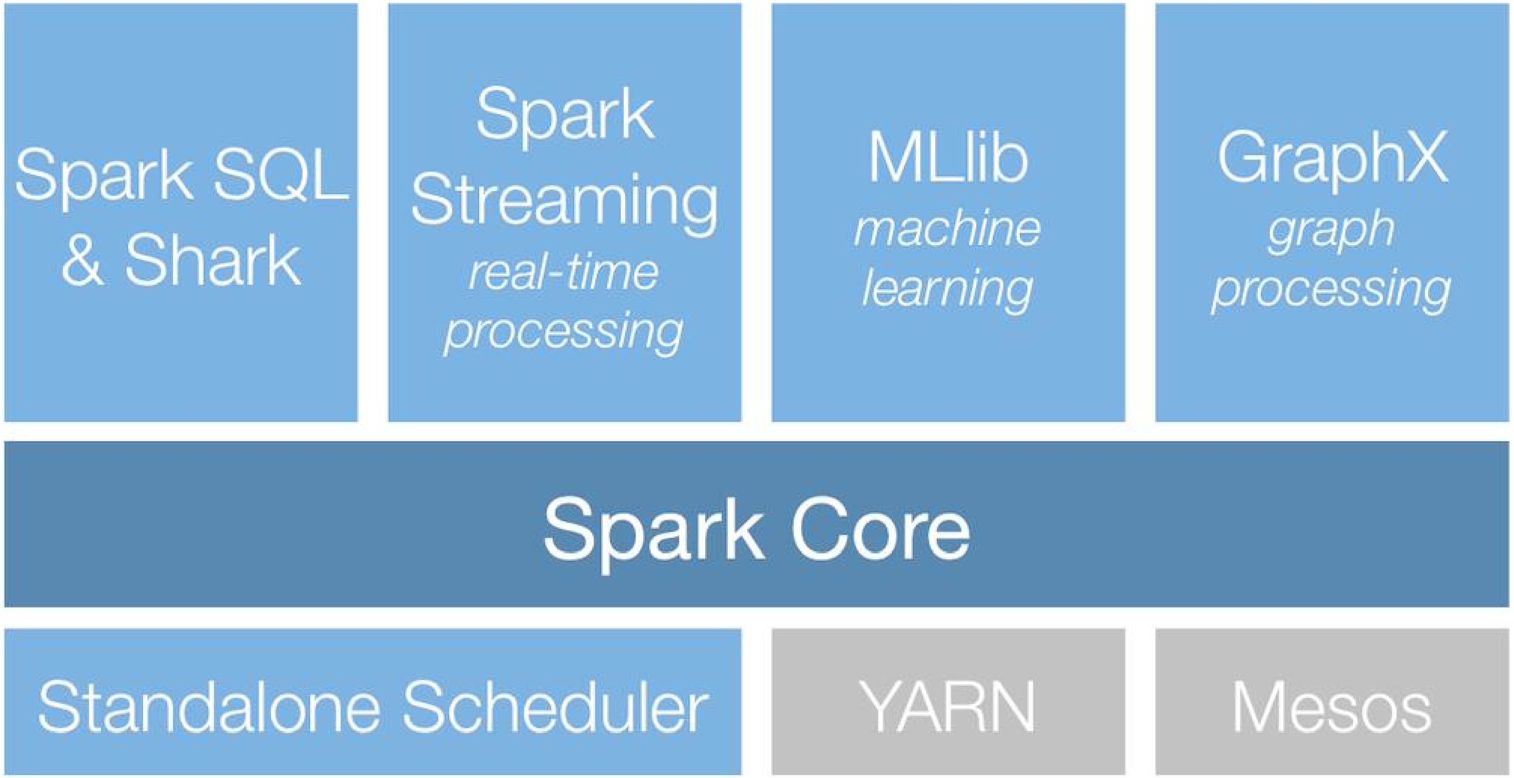


from http://spark.apache.org

General purpose

- Covers a wide range of workloads
- Provides SQL, streaming and complex analytics

Spark Stack

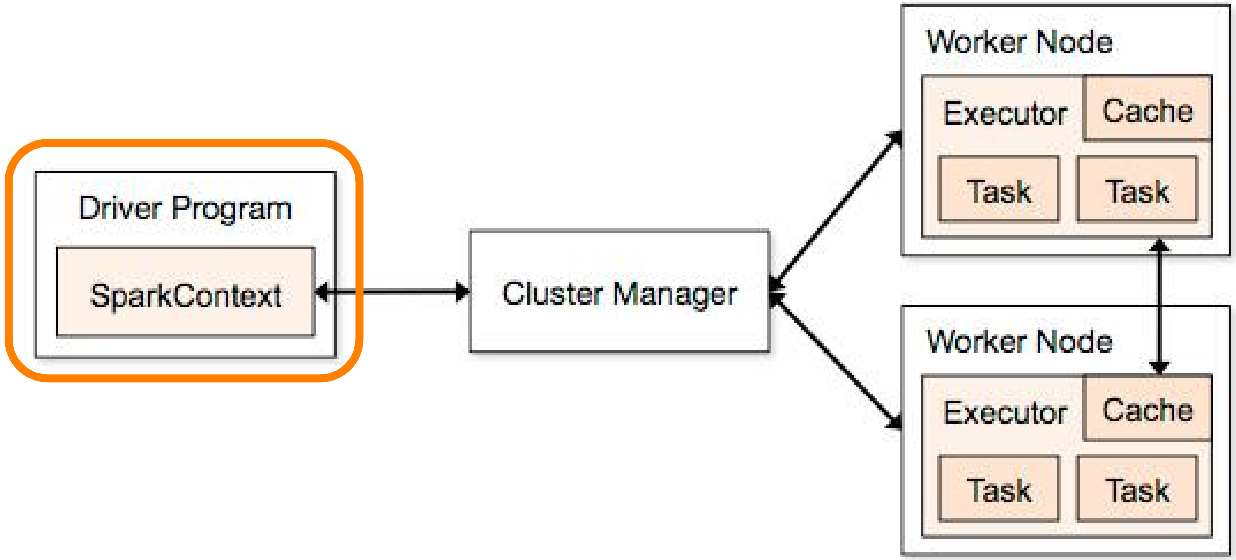

Spark SQL & Shark
Spark Streaming
real-time processing
MLlib
machine learning
GraphX
graph processing
Spark Core
Standalone Scheduler
YARN
Mesos

The Spark Programming model

Programming Model

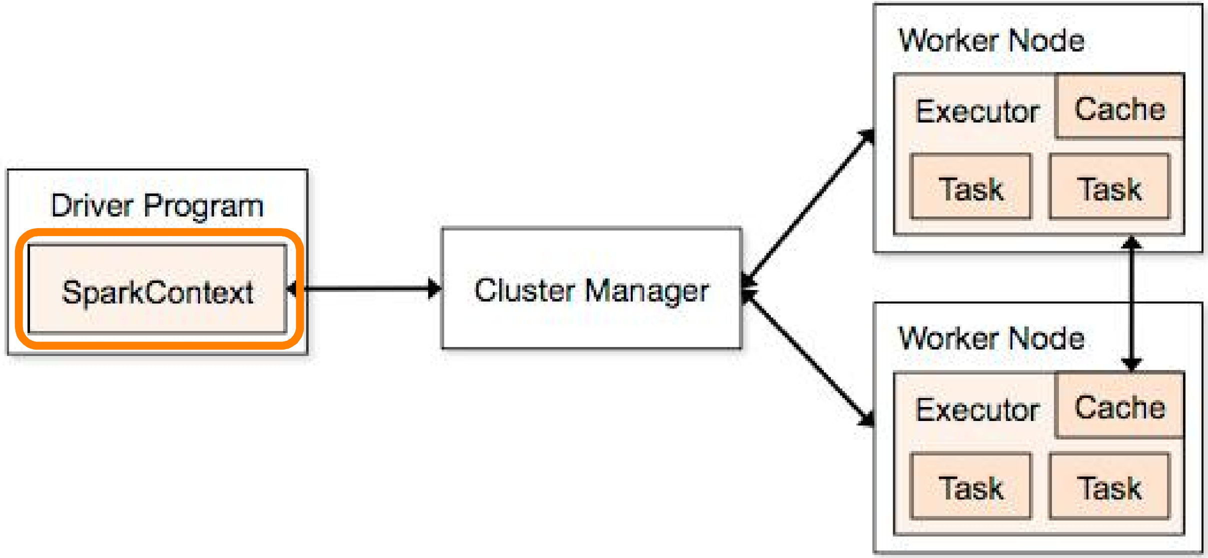
- Resilient distributed datasets (RDDs)
  - Immutable collections partitioned across cluster that can be rebuilt if a partition is lost
  - Created by transforming data in stable storage using data flow operators (map, filter, group-by, …)
  - Can be *cached* across parallel operations
- Parallel operations on RDDs
  - Reduce, collect, count, save, …
- Restricted shared variables
  - Accumulators, broadcast variables

Spark Application (I)



- A Spark application consists of a driver program that launches various parallel operations on a cluster.
  - The driver program contains application's main function and defines distributed datasets on the cluster, then applies operations to them.

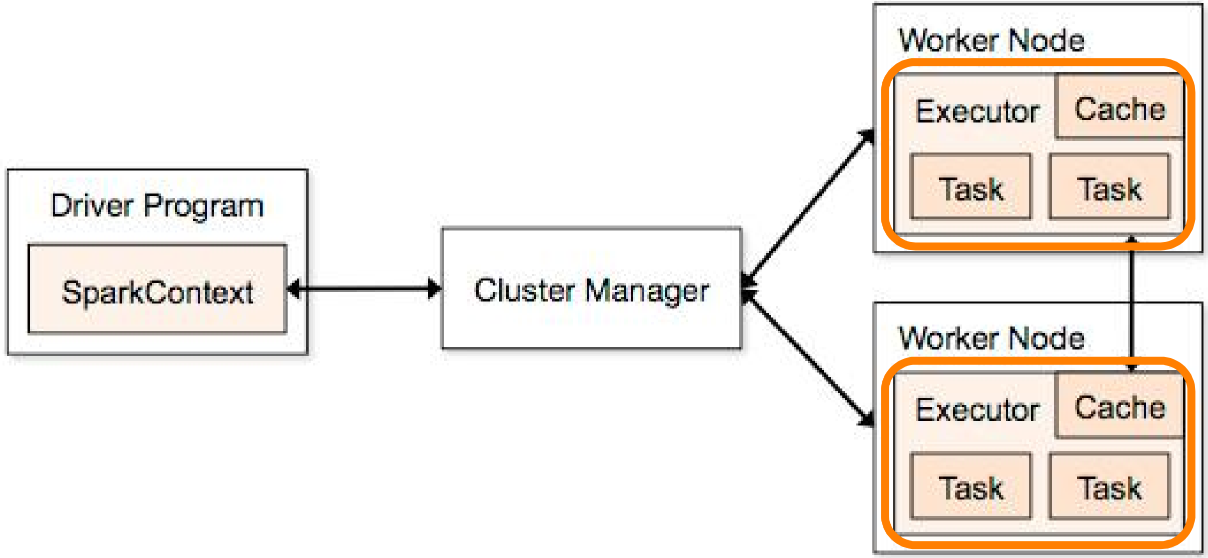
Spark Application (II)



- Driver programs access Spark through a **SparkContext** object.
  - Represents a connection to a computing cluster. It is used to create and manipulate distributed datasets and shared variables
  - SparkContext is initialized with an instance of a **SparkConf** object, which contains various Spark cluster-configuration settings.

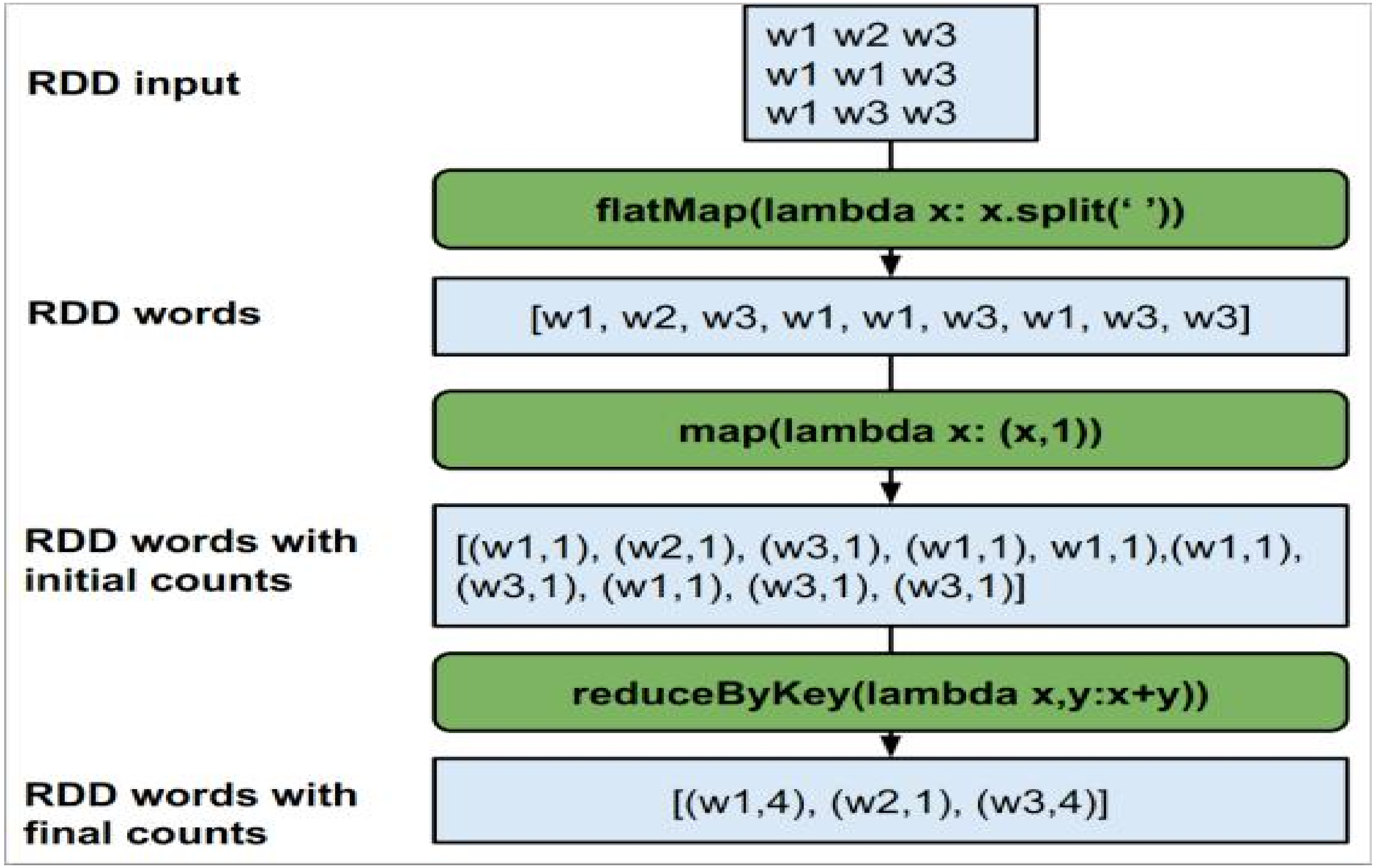
```
from pyspark import SparkConf, SparkContext
conf = SparkConf().setMaster("local").setAppName("My App")
sc = SparkContext(conf = conf)
```

Spark Application (III)



+ To run their operations, driver programs typically manage a number of nodes called **executors**.
  - They perform individual tasks and returns the results to Driver
  - Provide memory storage for datasets

Example: WordCount in Spark (I)



Example: WordCount in Spark (II)

Python

```python
from pyspark import SparkContext
sc = SparkContext(appName="WordCount Spark App")
text_file = sc.textFile("hdfs://...")
counts = text_file.flatMap(lambda line: line.split(" ")) \
             .map(lambda word: (word, 1)) \
             .reduceByKey(lambda a, b: a + b)
counts.saveAsTextFile("hdfs://...")
counts.count()
```

Scala

```scala
val sc = new SparkContext("local[2]", "WordCount Spark App")
val textFile = sc.textFile("hdfs://...")
val counts = textFile.flatMap(line => line.split(" "))
                .map(word => (word, 1))
                .reduceByKey(_ + _)
counts.saveAsTextFile("hdfs://...")
counts.count()
```

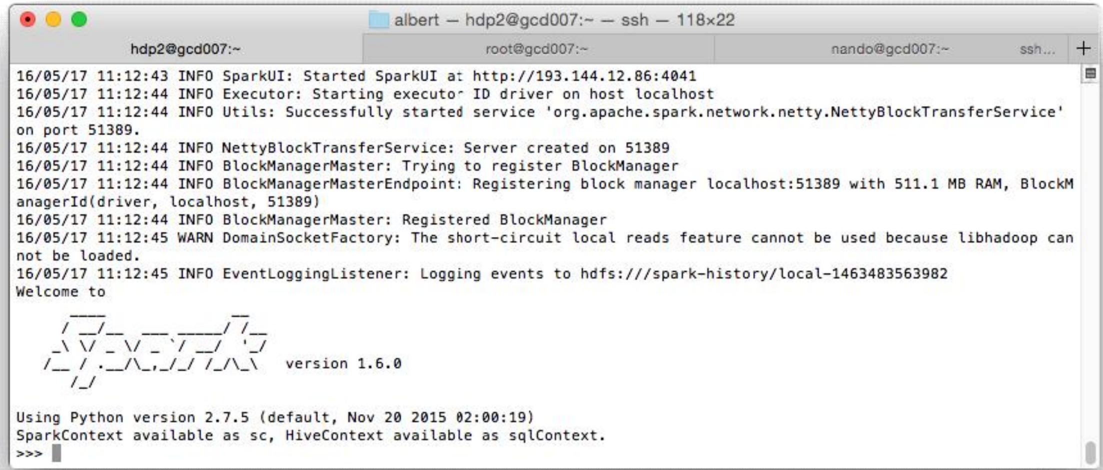
Example: WordCount in Spark (III)

Java

```
public class JavaApp {
    public static void main(String[] args) {
        JavaSparkContext sc = new JavaSparkContext("local[2]", "WordCount Spark App");
                JavaRDD<String> textFile = sc.textFile("hdfs://...");
                JavaRDD<String> words = textFile.flatMap(new FlatMapFunction<String, String>() {
                  public Iterable<String> call(String s) { return Arrays.asList(s.split(" ")); }
                });
                JavaPairRDD<String, Integer> pairs = words.mapToPair(new PairFunction<String, String, Integer>() {
                  public Tuple2<String, Integer> call(String s) { return new Tuple2<String, Integer>(s, 1); }
                });
                JavaPairRDD<String, Integer> counts = pairs.reduceByKey(new Function2<Integer, Integer, Integer>() {
                  public Integer call(Integer a, Integer b) { return a + b; }
                });
                counts.saveAsTextFile("hdfs://...");
```

Spark's Shells

- The Spark shell provides an easy and convenient way to quickly prototype certain operations without having to develop a full program, packaging it and then deploying it.
- Spark supports writing programs interactively using either the Scala or Python REPL (interactive shell).
  - Scala shell: ./bin/spark-shell
  - Phyton shell: ./bin/pyspark



Practice: Execute WordCount in the pyspark shell

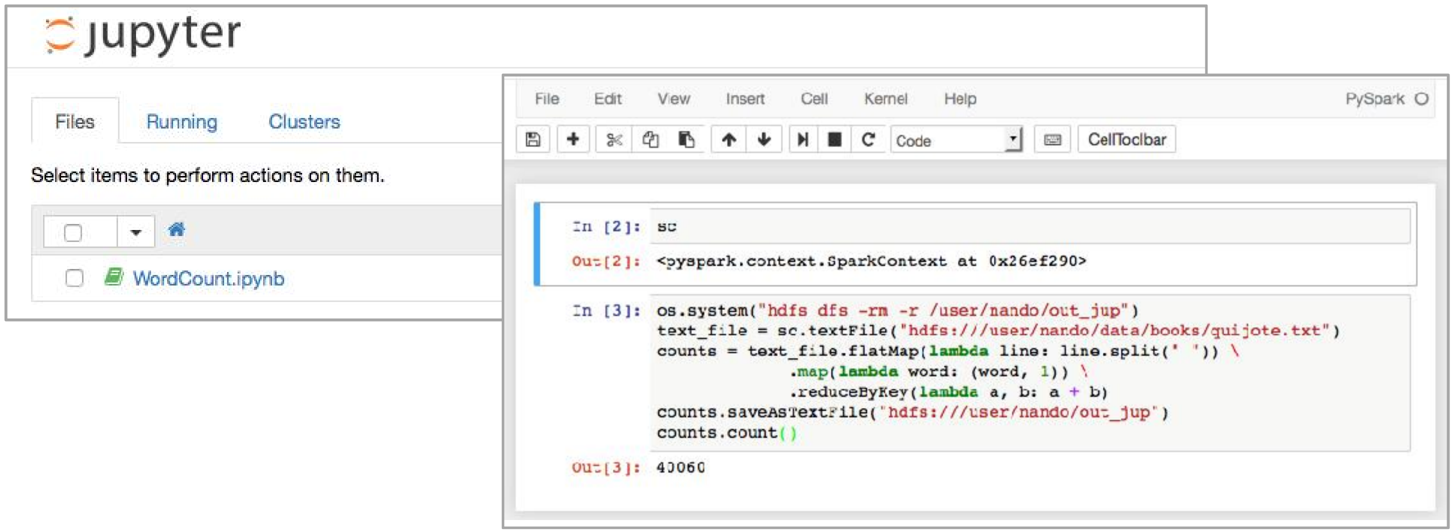
- Execute the WordCount spark application in the pyspark shell
  - Open the shell
    - /bin/pyspark
  - Use the books directory as input:
    - hdfs:///shared/nando/data/books/
  - Copy and execute each line of the code

```
os.system("hdfs dfs -rm -r /shared/nando/output/spark/wc")
text_file = sc.textFile("hdfs:///shared/nando/data/books/")
counts = text_file.flatMap(lambda line: line.split(" ")) \
            .map(lambda word: (word, 1)) \
            .reduceByKey(lambda a, b: a + b)
counts.saveAsTextFile("hdfs:///shared/nando/output/spark/wc")
counts.count()
```

  - Check the results:
    - hdfs dfs –cat /user/<your_user>/<out_dir>/p*

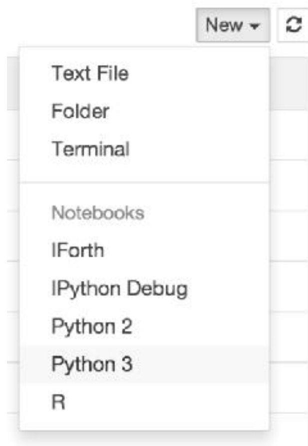
Jupyter Notebook

- **Jupyter** notebook is an interactive Python shell which lets you interact with your data one step at a time and also perform simple visualizations
  - Supports tab auto-completion on class names, functions, methods, variables.
  - Offers more explicit and colour-highlighted error messages than the command line python shell.
  - Provides integration with basic UNIX shell allowing to run simple shell commands.



Jupyter Notebook Basics (I)

- The notebook dashboard serves as a home page for the notebook. Its main purpose is to display the notebooks and files in the current directory.
  - The top of the notebook list displays clickable contents of the current directory.
    - By clicking on these element or on sub-directories in the notebook list, you can navigate your file system.
    - It also allows the creation of new directories, upload files or run notebooks.
    - To shutdown, delete, duplicate, or rename a notebook check the checkbox next to it and an array of controls will appear at the top of the notebook list
  - To create a new notebook, click on the "New" button at the top of the list and select a kernel from the dropdown.



Jupyter Notebook Basics (II)

- The notebook user interface (UI) allows you to run code and author notebook documents interactively.
  - The notebook UI has the following main areas: Menu, Toolbar and Notebook area and cells
  - The notebook has a modal user interface. This means that the keyboard does different things depending on which mode the Notebook is in.
    - There are two modes: **edit mode** and **command mode**.
  - There are two main types of cells: Code and Markdown cells
    - Code cells allows to write and execute spark code
    - Markdown cells allows to write markup text, Markdown, that it is a superset of HTML

```
1. Practice: Execute WordCount in the Notebook
    * Input Data: Datasets/Books
    * Result: Results/Out_WordCount
```

1. Practice: Execute WordCount in the Notebook
   - Input Data: Datasets/Books
   - Result: Results/Out_WordCount

In [2]:
```
import pyspark
sc = pyspark.SparkContext('local[*]')
```

Jupyter Notebook Basics (III)

- The Jupyter Notebook is an interactive environment for writing and running code. The notebook is associated with the IPython kernel, therefore runs Python code.
  - Code cells allow you to enter and run code
    - Alt-Enter runs the current cell and inserts a new one below.
    - Ctrl-Enter run the current cell and enters command mode.

```
In [2]: a = 10

In [3]: print(a)

        10
```

  - Code is run in a separate process called the Kernel. The Kernel can be interrupted or restarted.

Practice: Execute WordCount in the Notebook (cluster)

+ Execute the WordCount spark application in the notebook
  - Open the web navigator with the following url

    http://pirgi.udl.cat:8000

  - Login with your cluster account.
  - Create a new pyspark notebook
    - Create the Spark Context:

      ```
      import pyspark
      sc = pyspark.SparkContext('local[*]')
      ```

    - Copy the word count code, using the linux or hdfs file system for the input and output files.
    - Execute (Ctrl+Enter or Menu Cell → Run Cells)

Standalone Spark Applications

- The *spark-submit* script in Spark's bin directory is used to launch applications on a cluster.
  - It can use all of Spark's supported cluster managers (stadalone, apache Mesos, Hadoop YARN) through a uniform interface.
- Spark-submit syntax:

```
./bin/spark-submit  --class <main-class> --master <master-url> \
  --deploy-mode <deploy-mode>  --conf <key>=<value> \
  ... # other options
  <application-jar>  [application-arguments]
```

- Bundling Your Application's Dependencies:
  - If your code depends on other projects, you will need to package them alongside your application in order to distribute the code to a Spark cluster (create an assembly jar).
  - For Python, you can use the --py-files argument of spark-submit to add .py, .zip or .egg files to be distributed with your application.

Example: Standalone Spark Applications

```
# Run a Python application on a Spark standalone cluster
./bin/spark-submit --master spark://207.184.161.138:7077 \
  examples/src/main/python/pi.py 1000

# Run on a YARN cluster
./bin/spark-submit --class org.apache.spark.examples.SparkPi \
  --master yarn --deploy-mode cluster \  # can be client for client mode
  --executor-memory 20G --num-executors 50 \
  /path/to/examples.jar 1000

# Run on our YARN cluster, with resource limitations.
export SPARK_HOME=/usr/hdp/2.3.4.0-3485/spark
export PYTHONPATH=$SPARK_HOME/python/:$PYTHONPATH
spark-submit --class org.apache.spark.examples.SparkPi --master yarn \
   --deploy-mode cluster --driver-memory 1g --executor-memory 512m \
   --executor-cores 1 $SPARK_HOME/lib/spark-examples*.jar 10
```

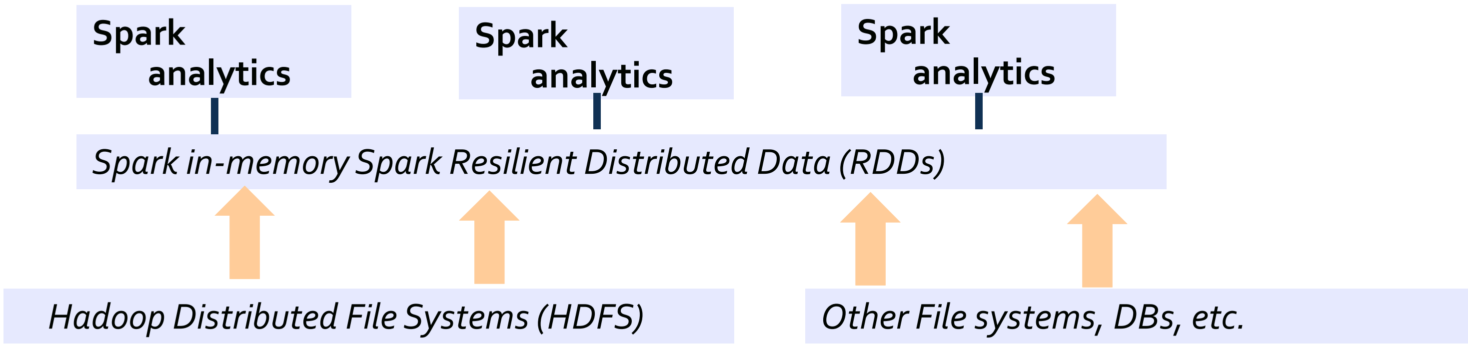
Practice: Execute WordCount in the standalone mode

- Execute the WordCount spark application in the standalone mode
  - Implement your spark+python program (wordcount example)
  - Copy the python file to the hadoop cluster
  - Submit the spark job:
    - spark-submit --master yarn --deploy-mode cluster ./WordCount.py
  - Check the results:
    - hdfs dfs –cat <out_dir>/p*t
    - See the output logs in http://pirgi.udl.cat:8088/proxy/application_1464002881097_0061/
- Monitor your job
  - http://pirgi.udl.cat:18081/
- On Error, consult the logs:
  - yarn logs -applicationId application_1456747411295_0025

Resilient Distributed Datasets (RDDs)

Resilient Distributed Datasets (RDDs)

- Spark's basic unit of data
- Immutable, fault tolerant collection of records that can be distributed and operated on in parallel across a cluster
- Fault tolerance
  - If data in memory is lost it will be recreated from lineage
- Caching, persistence (memory, spilling, disk) and check-pointing
- Many database or file type can be supported



Resilient Distributed Datasets (RDDs)

- An RDD is physically distributed across the cluster, but manipulated as one logical entity:
  - Each RDD is split into multiple **partitions**, which may be computed on different nodes of the cluster.
  - Spark will “distribute” any required processing to all partitions where the RDD exists and perform necessary redistributions and aggregations as well.
  - Partitioning can be based on a key in each record (using hash or range partitioning)
- Example: Consider a distributed RDD “Names” made of names

	Partition 1	Partition 2	Partition 3
Names	Michael Jacques Dirk	Cindy Dan Susan	Dirk Frank Jacques

Benefits of RDD Model

- Consistency is easy due to immutability
- Inexpensive fault tolerance (log lineage rather than replicating/checkpointing data)
- Locality-aware scheduling of tasks on partitions
- Despite being restricted, model seems applicable to a broad variety of applications

Creating RDDs

- There are two ways to create RDDs:
  - Parallelizing an existing collection in your driver program.

```
data = [1, 2, 3, 4, 5]
distData = sc.parallelize(data)
```

  - Referencing a dataset in an external storage system, such as a shared filesystem, HDFS, HBase, or any data source offering a Hadoop InputFormat.

```
distFile = sc.textFile("data.txt")
```

- Once created, the distributed dataset can be operated on in parallel.

```
distData = distData.reduce(lambda a,b, a+b)
```

- The number of partitions can be set manually by passing it as a second parameter to parallelize:

```
distData = sc.parallelize(data,10)
```

  - Spark will run one task for each partition of the cluster.
  - Spark tries to set the number of partitions automatically based on your cluster.

External Datasets

- Spark can create distributed datasets from any storage source supported by Hadoop:
  - Local filesystem, HDFS, Cassandra, HBase, Amazon S3, etc.
- Using a path on the local filesystem, the file must also be accessible at the same path on worker nodes.
- Support running on directories, compressed files, and wildcards
- Apart from text files, Spark's supports several other data formats:
  - ***SparkContext.wholeTextFiles(***path, minPart=None, use_unicode=True***)*** lets read a directory containing multiple small text files, and returns each of them as (filename, content) pairs.
  - ***RDD.saveAsPickleFile***(path, batchSize=10) and ***SparkContext.pickleFile(***name, minPartitions=None) support saving an RDD as a SequenceFile of serialized pickled Python objects.
  - SequenceFile and Hadoop Input/Output Formats

RDD Operations

- RDDs support two types of operations:
  - **Transformations:** which create a new dataset from an existing one.
    - ***map***: is a transformation that passes each dataset element through a function and returns a new RDD representing the results
  - **Actions:** which return a value to the driver program after running a computation on the dataset.
    - ***reduce***: is an action that aggregates all the elements of the RDD using some function and returns the final result to the driver program
- Transformations in Spark are ***lazy***, they do not compute their results right away.
  - The transformations are only computed when an action requires a result to be returned to the driver program.
- Transformed RDD may be recomputed each time you run an action on it
  - A RDD may also *persist* in memory using the ***persist*** (or ***cache***) method, to keep the elements around on the cluster to improve the acess time.

Example: Transformations & Actions

```
text_file = sc.textFile("hdfs:///user/nando/data/books/quijote.txt”)

# Calculate the length for each file line with map transformation

lineLengths = text_file.map(lambda s: len(s))

# Set lineLengths RDD to be keep in memory (persist)

print lineLengths.persist().is_cached

# Calculate the file length using the reduce action

totalLength = lineLengths.reduce(lambda a, b: a + b)

print "Total file lenght: "+format(totalLength)
```

RDD Operations

Transformations (define a new RDD)	**Parallel operations/actions** (return a result to driver)
map	reduce
filter	collect
sample	count
union	save
groupByKey	lookupKey
reduceByKey	…
join	
cache	
…	

Programming with Spark

Passing Functions to Spark

- Spark's API relies heavily on passing functions in the driver program to run on the cluster.

- There are three recommended ways to do this:
  - **Lambda expressions**, for simple functions that can be written as an expression.

    ```
    field = self.field
    rdd.map(lambda s: field + s)
    ```

  - **Local defs** inside the function calling into Spark, for longer code.

    ```
    if __name__ == "__main__":
        def myFunc(s):
            words = s.split(" ")
            return len(words)

        sc = SparkContext(...)
        sc.textFile("file.txt").map(myFunc)
    ```

  - **Top-level functions** in a module.

Working with Key-Value Pairs

- A few special operations are only available on RDDs of key-value pairs.
  - The most common ones are distributed “shuffle” operations, such as grouping or aggregating the elements by a key.
- In Python, these operations work on RDDs containing built-in Python tuples such as (1, 2).
  - Simply create such tuples and then call your desired operation.
- Example:

```
lines = sc.textFile("hdfs:///user/nando/data/books/quijote.txt")
pairs = lines.flatMap(lambda line:line.split(" ")).map(lambda w: (w, 1))
counts = pairs.reduceByKey(lambda a, b: a + b)
counts.sortByKey()
counts.collect()
```

Transformations (I)

- ***map(func)***

  Return a new distributed dataset formed by passing each element of the source through a function *func*.

- **filter**(*func*)

  Return a new dataset formed by selecting those elements of the source on which func returns true.

- **flatMap**(*func*)

  Similar to map, but each input item can be mapped to cero or more output items (so func should return a Seq rather than a single item).

- **mapPartitions**(*func*)

  Similar to map, but runs separately on each partition (block) of the RDD, so func must be of type Iterator<T> => Iterator<U> when running on an RDD of type T.

Transformations (II)

- **mapPartitionsWithIndex**(*func*)

  Similar to mapPartitions, but also provides func with an integer value representing the index of the partition, so func must be of type (Int, Iterator<T>) => Iterator<U> when running on an RDD of type T.

- **sample**(*withReplacement*, *fraction*, *seed*)

  Sample a fraction of the data, with or without replacement, using a given random number generator seed.

- **union**(*otherDataset*)

  Return a new dataset that contains the union of the elements in the source dataset and the argument.

- **intersection**(*otherDataset*)

  Return a new RDD that contains the intersection of elements in the source dataset and the argument.

Transformations (III)

- **distinct**([*numPartitions=none*])

  Return a new dataset that contains the distinct elements of the source dataset.

- **groupByKey**([*numPartitions*])

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

- **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.

- **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. Allows an aggregated value type that is different than the input value type, while avoiding unnecessary allocations.

Transformations (IV)

- **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.

- **cogroup**(*otherDataset*, [NumPartitions])

  When called on datasets of type (K, V) and (K, W), returns a dataset of (K, (Iterable<V>, Iterable<W>)) tuples. This operation is also called groupWith.

- **cartesian**(*otherDataset*)

  When called on datasets of types T and U, returns a dataset of (T, U) pairs (all pairs of elements).

Transformations (IV)

- **pipe**(*command*, *[envVars]*)

  Pipe each partition of the RDD through a shell command, e.g. a Perl or bash script. RDD elements are written to the process's stdin and lines output to its stdout are returned as an RDD of strings.

- **coalesce**(*numPartitions*)

  Decrease the number of partitions in the RDD to numPartitions. Useful for running operations more efficiently after filtering down a large dataset.

- **repartition**(*numPartitions*)

  Reshuffle the data in the RDD randomly to create either more or fewer partitions and balance it across them. This always shuffles all data over the network.

- **repartitionAndSortWithinPartitions**(*partitioner*)

  Repartition the RDD according to the given partitioner and, within each resulting partition, sort records by their keys.

Actions (I)

- **reduce**(*func*)

  Aggregate the elements of the dataset using a function *func* (which takes two arguments and returns one). The function should be commutative and associative so that it can be computed correctly in parallel.

- **collect**()

  Return all the elements of the dataset as an array at the driver program. This is usually useful after a filter or other operation that returns a sufficiently small subset of the data.

- **count**()

  Return the number of elements in the dataset.

- **first**()

  Return the first element of the dataset (similar to take(1)).

- **take**(*n*)

  Return an array with the first n elements of the dataset.

Actions (II)

- **takeSample**(*withReplacement*, *num*, [*seed*])

  Return an array with a random sample of *num* elements of the dataset, with or without replacement, optionally pre-specifying a random number generator seed.

- **takeOrdered**(*n*, *[ordering]*)

  Return the first n elements of the RDD using either their natural order or a custom comparator.

- **saveAsTextFile**(*path*)

  Write the elements of the dataset as a text file (or set of text files) in a given directory in the local filesystem, HDFS or any other Hadoop-supported file system.

- **countByKey**()

  Only available on RDDs of type (K, V). Returns a hashmap of (K, Int) pairs with the count of each key.

- **foreach**(*func*)

  Run a function *func* on each element of the dataset. This is usually done for side effects such as updating an Accumulator or interacting with external storage systems.

Questions???