Introduction to Big Data

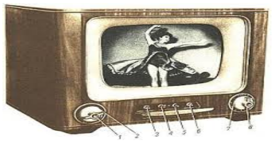
Daniel Hagimont
hagimont@enseeiht.fr

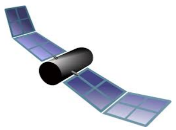
Context

- We generate more and more data
  - Individuals and companies
  - Kb → Mb → Gb → Tb → Pb → Eb → Zb → Yb → ???
- Few numbers
  - In 2013, Twitter generates 7 Tb per day and Facebook 10 Tb
  - The Square Kilometre Array radio telescope
    - Products 7 Pb of raw data per second, 50 Tb of analyzed data per day
  - Airbus generates 40 Tb for each plane test
  - Created digital data worldwide
    - 2010 : 1,2 Zb / 2011 : 1,8 Zb / 2012 : 2,8 Zb / 2020 : 40 Zb
    - 90 % of data were created in the last 2 years

Context

- Many data sources
  - Multiplication of computing devices and connected electronic equipments
  - Geolocation, e-commerce, social networks, logs, internet of things …
- Many data formats
  - Structured and unstructured data





Applications domains

- Scientific applications (biology, climate …)
- E-commerce (recommandation)
- Equipment supervision (e.g. energy)
- Predictive maintenance (e.g. airlines)
- Espionage

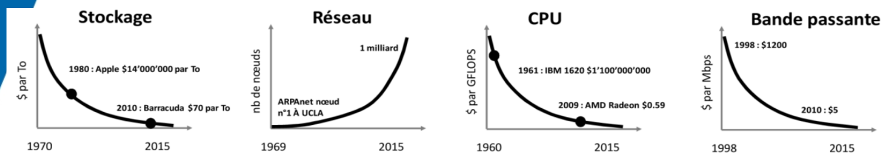
The NSA has built an infrastructure that allows it to intercept almost everything. With this capability, the vast majority of human communications are automatically ingested without targeting. E Snowden

New jobs

- Data Scientist
  - Geek/hacker : know how to develop, parameterize, deploy tools
  - HPC specialist : parallelism is key
  - IT specialist : know how to manage and transform data
  - Statistician : know how to use mathematics to classify, group and analyze information
  - Manager : know how to define objectives and identify the value of information

Computing infrastructures

- The reduced cost of infrastructures



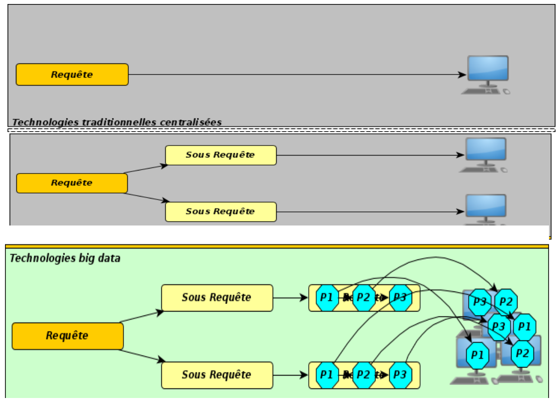
- Main actors (Google, Facebook, Yahoo, Amazon ...) developed frameworks for storing and processing data
- We generally consider that we enter the Big Data world when processing cannot be performed with a single computer

Definition of Big Data

- Definition
  - Rapid treatment of large data volumes, that could hardly be handled with traditional techniques and tools
- The three V of Big Data
  - Volume
  - Velocity
  - Variety
  - Two additional V
    - Veracity
    - Value

General approach

- Main principle : divide and conquer
  - Distribute IO and computing between several devices



Solutions

- Two main families of solutions
  - Processing in batch mode (e.g. Hadoop)
    - Data are initially stored in the cluster
    - Various requests are executed on these data
    - Data don't change / requests change
  - Processing in streaming mode (e.g. Storm)
    - Data are continuously arriving in streaming mode
    - Treatments are executed on the fly on these data
    - Data change / Requests don't change

} This lecture

The map-reduce principle

- We have to manage many stores around the world
  - A large document registers all the sales
    - For each sale : day – city – product - price
  - Objective : compute the total of sales per store
- The traditional method
  - A Hashtable memorizes the total for each store (<city, total>)
  - We iterate through all records
    - For each record, if we find the city in the Hashtable, we add the price

2012-01-01	London	Clothes	25.99
2012-01-01	Miami	Music	12.15
2012-01-02	NYC	Toys	3.10
2012-01-02	Miami	Clothes	50.00

⇩

London	25.99
Miami	12.15
NYC	3.10

⇨

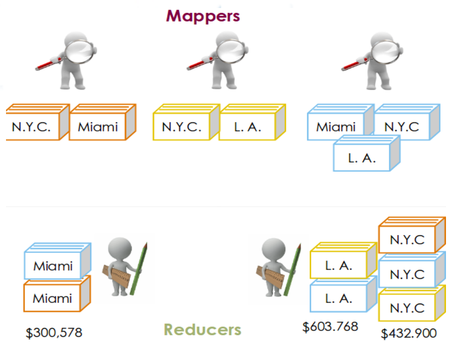
London	25.99
Miami	62.15
NYC	3.10

The map-reduce principle

- What happens if the document size is 1 Tb ?
  - I/O are slow
  - Memory saturation on the host
  - Treatment is too long
- Map-Reduce
  - Divide the document in several fragments
  - Several machines for computing on the fragments
  - Mappers : execute in parallel on the fragments
  - Reducers : aggregate the results from mappers

The map-reduce principle

- Mappers
  - Gather from a document fragment <city, price> pairs
  - Send them to reducers according to city
- Reducers
  - Each reducer is responsible for a set of city
  - Each reduce computes the total for each city

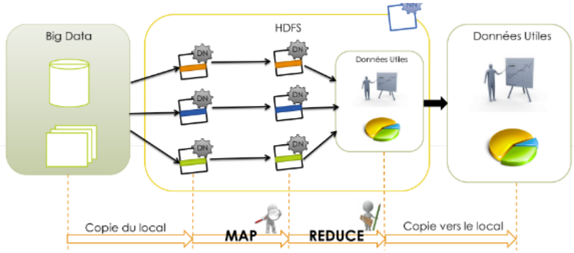


Hadoop

- Support the execution of Map-Reduce applications in a cluster
  - The cluster could group tens, hundreds or thousands of nodes
  - Each node provides storage and compute capacities
- Scalability
  - It should allow storage of very large volumes of data
  - It should allow parallel computing of such data
  - It should be possible to add nodes
- Fault tolerance
  - If a node crashes
    - ongoing computing should not fail (jobs are re-submitted)
    - Data should be still available (data is replicated)

Hadoop principles

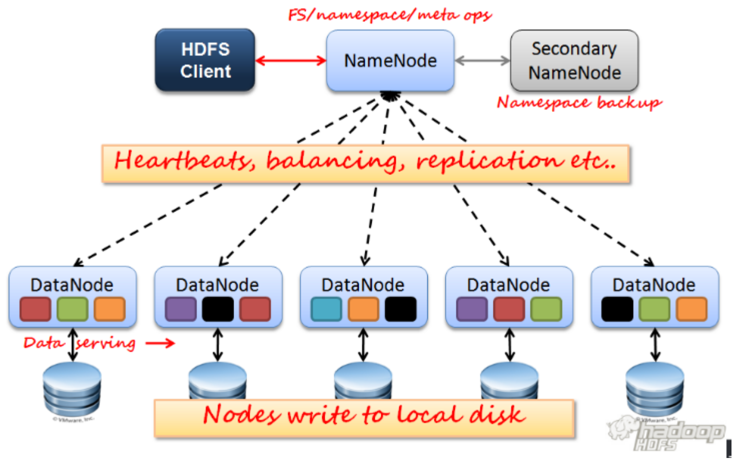
- Two main parts
  - Data storage : HDFS (Hadoop Distributed File System)
  - Data treatment : Map-Reduce
- Principle
  - Copy data to HDFS – data is divided and stored on a set of nodes
  - Treat data where they are stored (Map) and gather results (Reduce)
  - Copy results from HDFS



HDFS : Hadoop Distributed File System

- A new file system to read and write data in the cluster
- Files are divided in blocks between nodes
- Large block size (initially 64 Mb)
- Blocks are replicated in the cluster (3 times by default)
- Write-once-read-many : designed for one write / multiple reads
- HDFS relies on local file systems

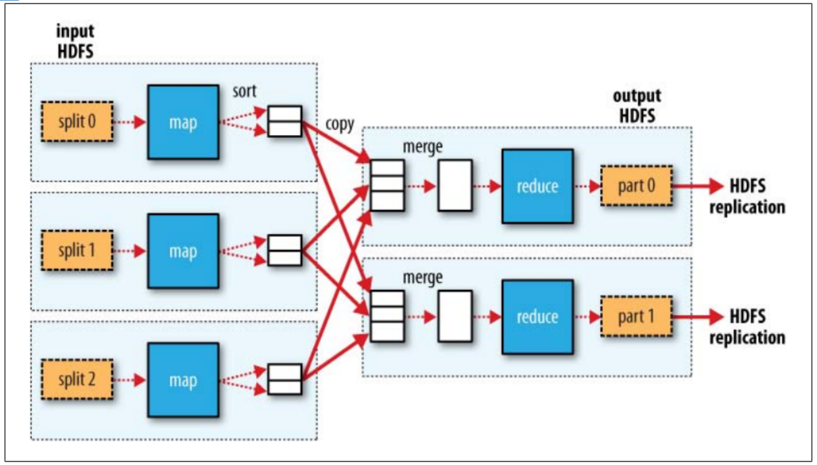
HDFS architecture



Programming with Hadoop

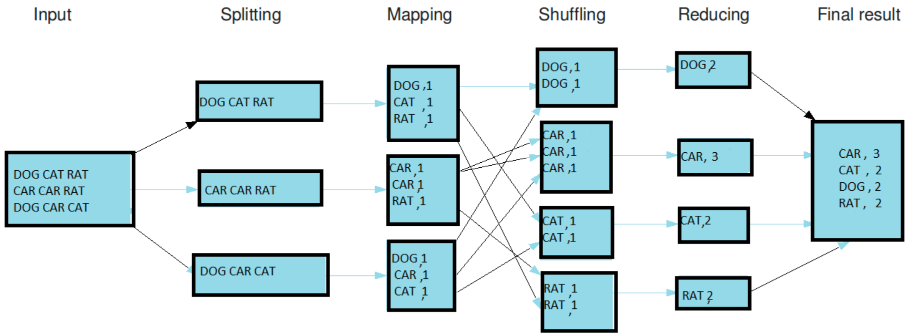
- Basic entity : key-value pair (KV)
- The map function
  - Input : KV
  - Output : {KV}
  - The map function receives successively a set of KV from the local block
- The reduce function
  - Input : K{V}
  - Output : {KV}
  - Each key received by a reduce is unique

Execution scheme



Example

The overall MapReduce word count process



Programming with Hadoop

- The WordCount application
  - Input : a large text file (or a set of text files)
    - Each line is read as a KV <line-number, line>
  - Output : number of occurrence of each word

Map

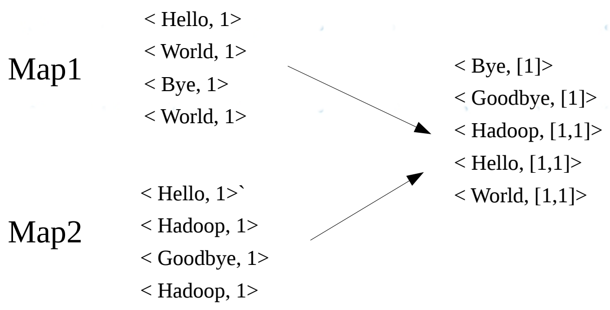
map(key, value) → List(key_i, $value_i$)

<1, Hello World Bye World> →
< Hello, 1>
< World, 1>
< Bye, 1>
< World, 1>

```
public static class TokenizerMapper
       extends Mapper<Object, Text, Text, IntWritable> {
    private final static IntWritable one = new IntWritable(1);
    private Text word = new Text();
    public void map(Object key, Text value, Context context
                    ) throws IOException, InterruptedException {
      StringTokenizer itr = new StringTokenizer(value.toString());
      while (itr.hasMoreTokens()) {
        word.set(itr.nextToken());
        context.write(word, one);
      }
    }
}
```


Shuffle and sort

- Shuffle : group KVs whose K is identical
- Sort : sort by K
- Done by the framework



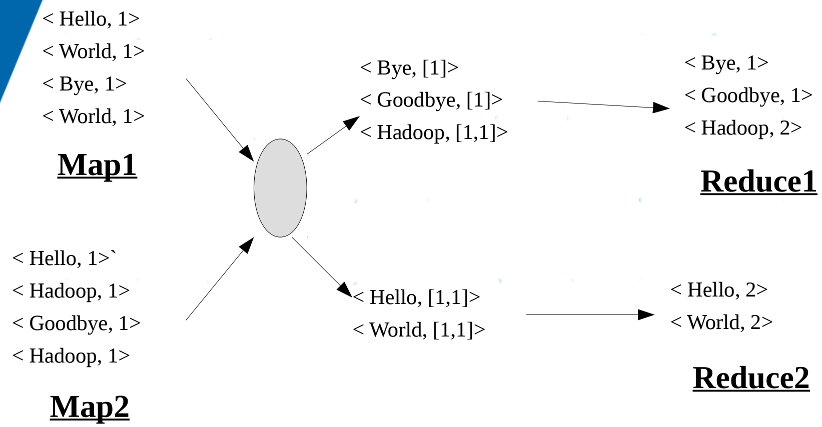
Reduce

reduce(key, List($value_i$)) → List(key_i, $value_i$)

< Bye, [1]>
< Goodbye, [1]>
< Hadoop, [1,1]>
< Hello, [1,1]>
< World, [1,1]>
→
< Bye, 1>
< Goodbye, 1>
< Hadoop, 2>
< Hello, 2>
< World, 2>

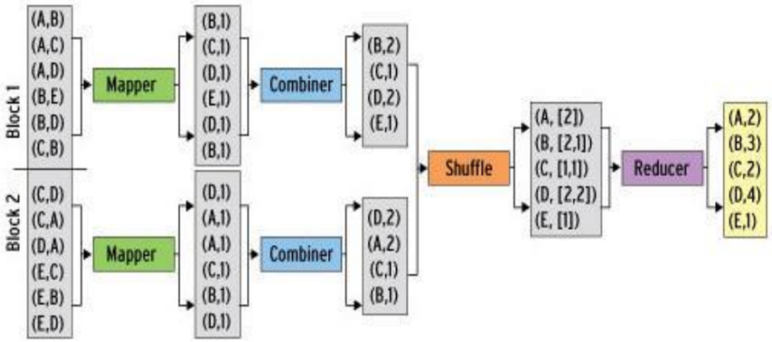
```
public static class IntSumReducer
       extends Reducer<Text,IntWritable,Text,IntWritable> {
    private IntWritable result = new IntWritable();

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

Several reduces



Combiner functions

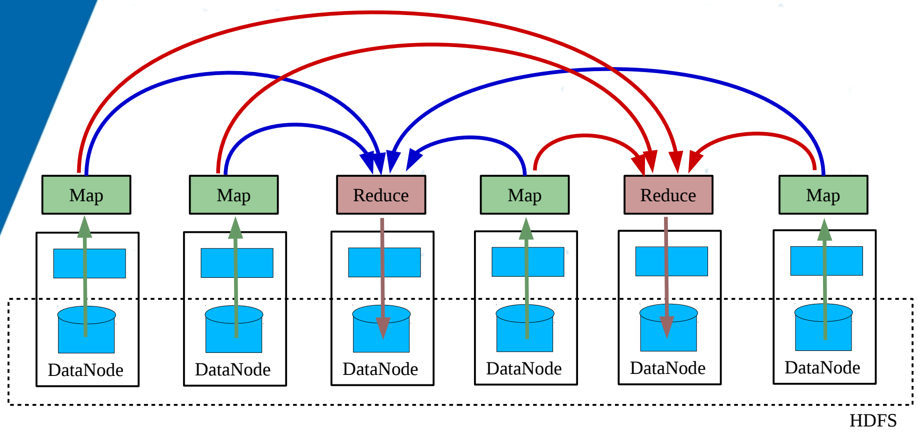
- Reduce data transfer between map and reduce
  - Executed at the ouput of map
  - Often the same function as reduce



Main program

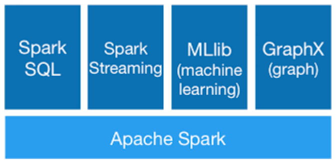
```
public class WordCount {
  public static void main(String[] args) throws Exception {
    Configuration conf = new Configuration();
    Job job = Job.getInstance(conf, "word count");
    job.setJarByClass(WordCount.class);
    job.setMapperClass(TokenizerMapper.class);
    job.setCombinerClass(IntSumReducer.class);
    job.setReducerClass(IntSumReducer.class);
    job.setOutputKeyClass(Text.class);
    job.setOutputValueClass(IntWritable.class);
    FileInputFormat.addInputPath(job, new Path(args[0]));
    FileOutputFormat.setOutputPath(job, new Path(args[1]));
    System.exit(job.waitForCompletion(true) ? 0 : 1);
 }
}
```


Execution in a cluster

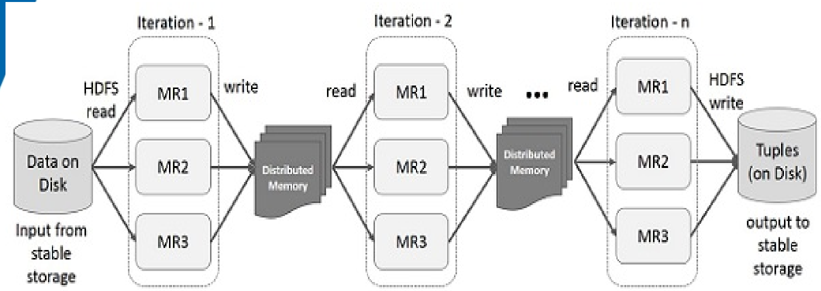


Sparks in few words

- Evolution from Hadoop
  - Speed: reducing read/write operations
    - Up to 10 times faster when running on disk
    - Up to 100 times faster when running in memory
  - Multiple-languages: Java, Scala or Python
  - Advance analytics: not only Map-Reduce
    - SQL
    - Streaming data
    - Machine Learning
    - Graph algorithms



Iterative scheme with Spark



- Keep data in memory as long as possible
- Store on disk only if memory is not sufficient
- Also don’t have to restart JVMs

Programming with Spark

- Initialization

```
SparkConf conf = new SparkConf().setAppName("WordCount");
JavaSparkContext sc = new JavaSparkContext(conf);
```

- Spark relies on Resilient Distributed Datasets (RDD)
  - Datasets that are partitioned on nodes
  - Can be operated in parallel

Programming with Spark

- RDD created from a Java object

```
List<Integer> data = Arrays.asList(1, 2, 3, 4, 5);
JavaRDD<Integer> rdd = sc.parallelize(data);
```

- RDD created from an external storage (file)

```
JavaRDD<String> rdd = sc.textFile("data.txt");
```


Programming with Spark

- Driver program: the main program
- Two types of operation on RDD
  - Transformations: create a new RDD from an existing one
    - e.g. map() passes each RDD element through a given function
  - Actions: compute a value from a existing RDD
    - e.g. reduce() aggregates all RDD elements using a given function and computes a single value
- Transformations are lazily computed when needed to perform an action (optimization)
- By default, transformations are cached in memory, but they can be recomputed if they don't fit in memory

Programming with Spark

- Example with lambda expressions
  - map(): apply a function to each element of a RDD
  - reduce(): apply a function to aggregate all values from a RDD
    - Function must be associative and commutative for parallelism

```
JavaRDD<String> lines = sc.textFile("data.txt");
JavaRDD<Integer> lineLengths = lines.map(s -> s.length());
int totalLength = lineLengths.reduce((a, b) -> a + b);
lineLengths.persist(StorageLevel.MEMORY_ONLY());
```

- Or with Java functions

```
JavaRDD<String> lines = sc.textFile("data.txt");
JavaRDD<Integer> lineLengths = lines.map(new Function<String, Integer>() {
        public Integer call(String s) { return s.length(); }
});
int totalLength = lineLengths.reduce(new Function2<Integer, Integer, Integer>() {
        public Integer call(Integer a, Integer b) { return a + b; }
});
```


Programming with Spark

- Execution of operations (transformations/actions) is distributed
  - Variables in the driver program are serialized and copied on remote hosts (they are not global variables)

```
int counter = 0;
JavaRDD<Integer> rdd = sc.parallelize(data);

// Wrong: Don't do this!!
rdd.foreach(x -> counter += x);
println("Counter value: " + counter);
```

- Should use special Accumulator/Broadcast variables

Programming with Spark

- Many operations rely on key-value pairs
  - Example (count the lines)
    - mapToPairs(): each element of the RDD produces a pair
    - reduceByKey(): apply a function to aggregate values for each key

```
JavaRDD<String> lines = sc.textFile("data.txt");
JavaPairRDD<String, Integer> pairs = lines.mapToPair(s -> new Tuple2(s, 1));
JavaPairRDD<String, Integer> counts = pairs.reduceByKey((a, b) -> a + b);
```


WordCount example

```
JavaRDD<String> data = sc.textFile(inputFile).flatMap(s ->
                                        Arrays.asList(s.split(" ")).iterator());

JavaPairRDD<String, Integer> counts =
                        data.mapToPair(w -> new Tuple2<String, Integer>(w,1)).
                                        reduceByKey((c1,c2) -> c1 + c2);
```

Many APIs

groupByKey([numTasks])	When called on a dataset of (K, V) pairs, returns a dataset of (K, Iterable<V>) pairs. Note: If you are grouping in order to perform an aggregation (such as a sum or average) over each key, using reduceByKey or aggregateByKey will yield much better performance. Note: By default, the level of parallelism in the output depends on the number of partitions of the parent RDD. You can pass an optional numTasks argument to set a different number of tasks.
reduceByKey(func, [numTasks])	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. Like in groupByKey, the number of reduce tasks is configurable through an optional second argument.
aggregateByKey(zeroValue)(seqOp, combOp, [numTasks])	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. Like in groupByKey, the number of reduce tasks is configurable through an optional second argument.
sortByKey([ascending], [numTasks])	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, [numTasks])	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.
cogroup(otherDataset, [numTasks])	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).
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. This is more efficient than calling repartition and then sorting within each partition because it can push the sorting down into the shuffle machinery.

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.
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. Spark will call toString on each element to convert it to a line of text in the file.

saveAsSequenceFile(path)	Write the elements of the dataset as a Hadoop SequenceFile in a given path in the local filesystem, HDFS or any other Hadoop-supported file system. This is available on RDDs of key-value pairs that implement Hadoop's Writable interface. In Scala, it is also available on types that are implicitly convertible to Writable (Spark includes conversions for basic types like Int, Double, String, etc).
saveAsObjectFile(path)	Write the elements of the dataset in a simple format using Java serialization, which can then be loaded using SparkContext.objectFile().
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. Note: modifying variables other than Accumulators outside of the foreach() may result in undefined behavior. See Understanding closures for more details.

Using Spark

- Install Spark
  - tar xzf spark-2.2.0-bin-hadoop2.7.tgz
  - Define environment variables
    - export SPARK_HOME=<path>/spark-2.2.0-bin-hadoop2.7
    - export PATH=$PATH:$SPARK_HOME/bin:$SPARK_HOME/sbin
- Development with eclipse
  - Create a Java Project
  - Add jars in the build path
    - $SPARK_HOME/jars/spark-core_2.11-2.2.0.jar
    - $SPARK_HOME/jars/scala-library-2.11.8.jar
    - $SPARK_HOME/jars/hadoop-common-2.7.3.jar
    - Could include all jars, but not very clean
  - Your application should be packaged in a jar
- Launch the application
  - spark-submit --class <classname> --master <url-master> target/<jarfile>
    - Centralized: <url-master> = local or local[n]
    - Cluster: <url-master> = url to access the cluster's master

Cluster mode

- Starting the master
  - start-master.sh
  - You can check its state and see its URL at http://master:8080
- Starting slaves
  - start-slave.sh -c 1 <url master>
  - // -c 1 to use only one core
- Files
  - If not running on top of HDFS, you have to replicate your files on the slaves
  - Else you program should refer to the input file in HDFS with a URL

Conclusion

- Spark is just the beginning
- You should have a look at
  - Spark streaming
  - Spark SQL
  - ML Lib
  - GraphX
  - ...