CentraleSupelec 2018-2019 MSC DSBA / DATA SCIENCES

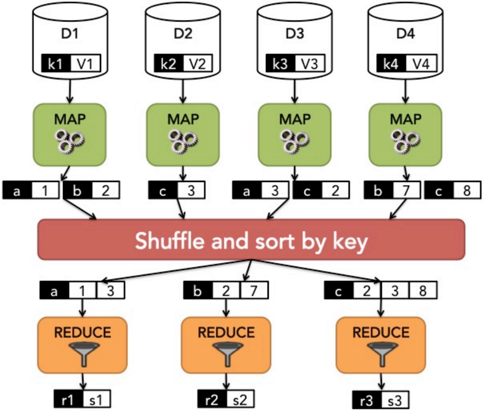
Big Data Algorithms, Techniques and Platforms

Distributed Computing with Spark

Hugues Talbot & Céline Hudelot, professors.

Recap : Processing Big Data with Hadoop MapReduce

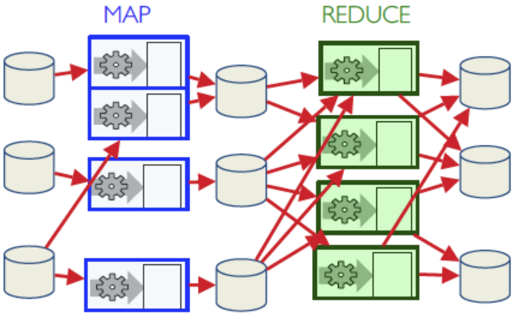
- A simple programming model for processing huge data sets in a distributed way.
- A framework that runs these programs on clusters of commodity servers, automatically handling the details of distributed computing



Processing Big Data with Hadoop MapReduce

Limitations

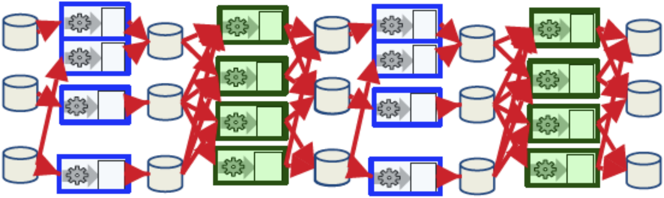
Efficiency : Each stage passes through the hard drives ! Frequent disk I/O

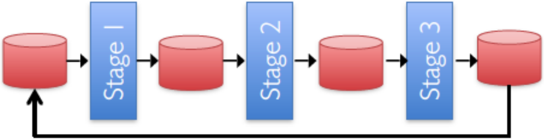


Processing Big Data with Hadoop MapReduce

Limitations for iterative jobs

Efficiency : Disk I/O for each repetition. The processing is slow when many small iterations.



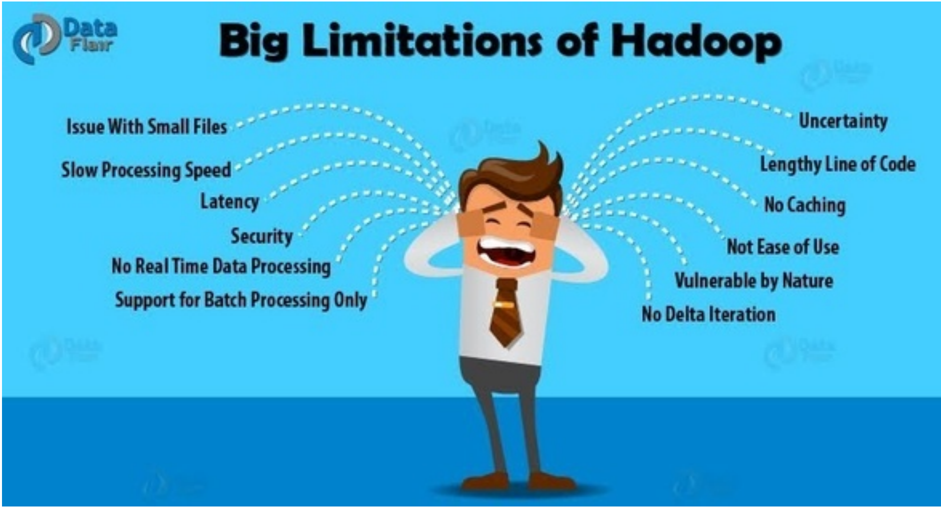


Processing Big Data with Hadoop MapReduce

Other limitations

- Efficiency
  - *Shuffle step* for each reduce step : high communication cost.
  - Limite exploitation of the main memory
- Real-time processing
  - Stream processing and random access is not possible.
  - Designed for batch processing
- Programming aspect
  - The MapReduce design pattern is very constrained and not so expressive.
  - Native support for JAVA only (c.f. your difficulties in using python)

Processing Big Data with Hadoop MapReduce



In-Memory Processing

Main ideas

- Many datasets fit in memory (of a cluster).
- Memory is fast and avoid disk I/O.
- Idea : In-memory Computing - The data is kept in random access memory(RAM)

Plan

1 Introduction : Limitations of Hadoop

2 Spark
- RDD : Resilient Distributed Dataset
- Anatomy of a job
- Examples

3 Spark SQL and DataFrames

4 MLlib

Spark

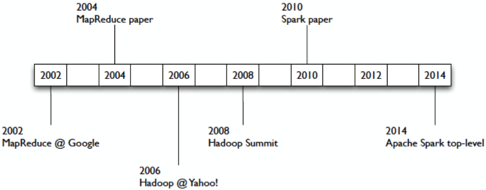
- **Spark** is another distributed computing framework.
- A cluster-computing platform that provides an API for distributed programming designed to be fast for interactive queries and iterative algorithms.
- A separate, fast, MapReduce engine.
- An unified computing engine and a set of libraries for parallel data processing on computer clusters
- In-memory data storage for very fast iterative queries.
- Handles batch, interactive, and real-time within a single framework.
- Use of **Resilient Distributed Dataset** : a data structure to distribute data and computations across the cluster

Spark

A bit of history

- **Spark** project started in 2009 at UC Berkeley AMPLab.
  - PhD thesis of Matei Zaharia [a]
- Open sourced in 2010.
- Today, the most popular project for big data analysis [b].

a. http://static.usenix.org/legacy/events/hotcloud10/tech/full_papers/Zaharia.pdf
b. https://tinyurl.com/y7nf5z5z



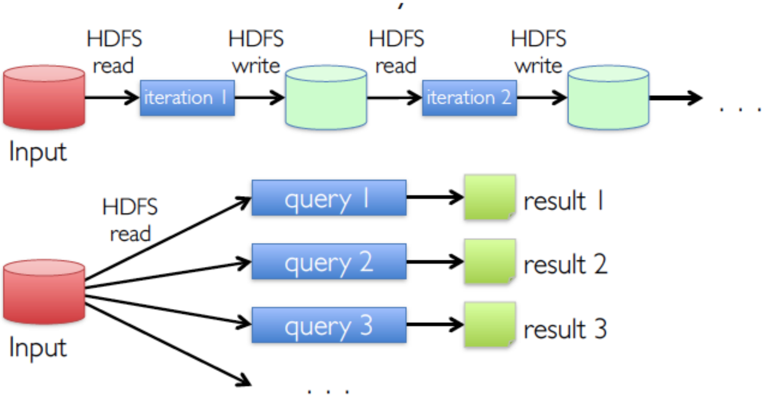
Spark

A bit of history

- Two funding papers :
  - *Spark : Cluster Computing with Working Sets* Matei Zaharia, Mosharaf Chowdhury, Michael J. Franklin, Scott Shenker, Ion Stoica USENIX HotCloud (2010) people.csail.mit.edu/matei/papers/2010/hotcloud_spark.pdf
  - *Resilient Distributed Datasets : A Fault-Tolerant Abstraction for In-Memory Cluster Computing*. Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J. Franklin, Scott Shenker, Ion Stoica.NSDI (2012) usenix.org/system/files/conference/nsdi12/nsdi12-final138.pdf

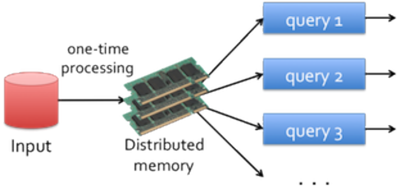
Spark : In-Memory **fast** data sharing

Data sharing in MapReduce

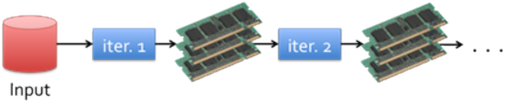


Spark : In-Memory **fast** data sharing

Replace disk with Memory



(a) Low-latency computations (queries)



(b) Iterative computations

10-100x faster than network and disk

Characteristics of Spark

- **Performance** : In-memory data processing.
- **Processing model** : Map Reduce and others models
- **Data Processing** : No need of external librairies
- **Database Management** : No need of external librairies
- **Programming aspects** : Java, **Scala** but also native python, R.

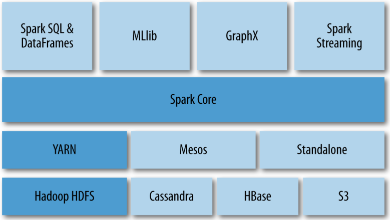
Characteristics of Spark

Hadoop	Spark
Java Virtual Machine (JVM)	
Write to disk (HDFS)	In-memory
Native data structures	Resilient Distributed Datasets (RDD)
Java (+ Hadoop streaming)	Java + Scala + Python + R
-	Python + Scala shell
Pluggable SQL (Hive)	Spark SQL (native)
Pluggable ML	Spark ML (native)

The Spark stack

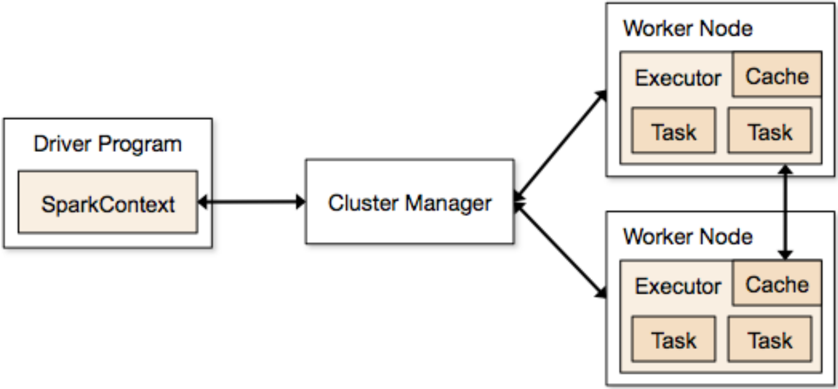
Spark focuses on computation rather than on data storage.

- Runs on a cluster with data warehousing and cluster management (e.g. Yarn, HDFS)
- **Spark Core Module** : basic and general functionalities, API.
- **Spark SQL** : SQL interface
- **Spark Streaming** : Stream processing.
- **MLlib** : Machine learning.
- **GraphX** : Parallel graph data processing.



The Spark execution model

- Spark **applications** are run as independent sets of processes, coordinated by a **Spark Context** in a **driver program**.
- Spark **application** = a driver program and executors on the cluster.
- Cluster manager : An external service for acquiring resources on the cluster



The Spark execution model

SparkContext

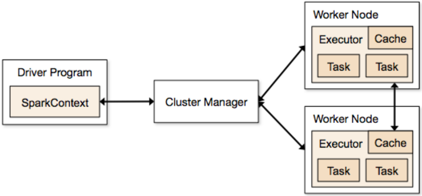
- First thing that a spark program does : create a SparkContext object that tell Spark how to access a cluster.

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

<SparkContext master=local[*] appName=pyspark-shell>
```

master	*description*
local	run Spark locally with one worker thread (no parallelism)
local[K]	run Spark locally with K worker threads (ideally set to # cores)
spark://HOST:PORT	connect to a Spark standalone cluster; PORT depends on config (7077 by default)
mesos://HOST:PORT	connect to a Mesos cluster; PORT depends on config (5050 by default)

The Spark execution model



The master :

1. Connects to a *cluster manager* (e.g. Yarn) which allocates resources across applications.
2. Acquires *executors* on cluster nodes (worker processes to run computations and store data).
3. Sends *app code* to the executors.
4. Sends *tasks* to the executors to run.

RDD : Resilient Distributed Dataset

- The key abstraction in Spark.
- Data structure used by Spark to distribute data and computations across the cluster.
- A fault-tolerant (automatically rebuilt by the operation history) collection of elements than can be operated in parallel.
- An **immutable distributed** collection of data.
- Can persist in memory, on disk, or both
- An object on which functions can be invoked.
- **Spark application** :
  - Data loading into one or more RDDs.
  - Computations by invoking functions on the RDDS

RDD : Resilient Distributed Dataset

Two types

- **Parallelized collections** : Take an existing in-memory collection and pass it to SparkContext parallelize method.

```
words = ["fish", "cats", "dogs"]
wordsRDD = sc.parallelize(words)
```

- **Hadoop datasets** : Read from local text file or from HDFS or other storage systems.

```
>>> linesRDD = sc.textFile("./ProjectWeek2.md")
>>> print(linesRDD)
./ProjectWeek2.md MapPartitionsRDD[5] at textFile at
    NativeMethodAccessorImpl.java:0
>>> RDD = sc.textFile("hdfs:/share/data.txt")
```

RDD : Operations

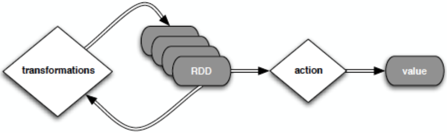
Two types of operations on RDD :

- Transformations : Take in one or more RDDs and return a new RDD.
  - *e.g. map, filter, intersection, groupbykey,...*
  - **lazy execution** :does not execute immediately, the execution will not start until an action is triggered. Spark maintains the record of which operation is being called (Through **DAG**).
- actions : Take in an RDD and return a value
  - *e.g. collect, count, reduce*
  - Trigger the execution of transformations.

A spark application

Create RDDs, apply transformations and actions.

RDD : Operations



RDD operations

Transformations	Actions
map, distinct, filter, reduceByKey, sortByKey, join...	reduce, collect, count, first, take...
Arguments: 1 or more RDD	
Returns: RDD	Returns: not an RDD
Lazy evaluation	Immediate evaluation
Sometimes shuffle	Shuffle necessary

RDD : Transformations

transformation	*description*
`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 0 or more output items (so *func* should return a Seq rather than a single item)
`sample(withReplacement, fraction, seed)`	sample a fraction *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
`distinct([numTasks]))`	return a new dataset that contains the distinct elements of the source dataset

RDD : Transformations

transformation	*description*
groupByKey([*numTasks*])	when called on a dataset of (K, V) pairs, returns a dataset of (K, Seq[V]) pairs
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
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
cogroup(*otherDataset*, [*numTasks*])	when called on datasets of type (K, V) and (K, W), returns a dataset of (K, Seq[V], Seq[W]) tuples – 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)

RDD : Transformations

Filter

```
r1 = sc.parallelize([1, 2, 3, 4])
f = r1.filter(lambda x : x > 3)
>>> print(f)
PythonRDD[7] at RDD at PythonRDD.scala:48
>>> print(f.collect())
[4]
```

Map

```
square = r1.map(lambda x : x*x)
>>> print(square)
PythonRDD[8] at RDD at PythonRDD.scala:48
>>> print(square.collect())
[1, 4, 9, 16]
```

RDD : Transformations

Lambda functions are not mandatory.

Map

```
def square(x):
    return x*x

squares=r1.map(square)
print(squares.collect())

[1, 4, 9, 16]
```

RDD : Transformations

Map

```
r1 = sc.parallelize([1, 2, 3, 4])
squares = r1.map(lambda x : [x, x*x])
print(squares.collect())

[[1, 1], [2, 4], [3, 9], [4, 16]]
```

flatMap

```
squares = r1.flatMap(lambda x : [x, x*x])
print(squares.collect())

[1, 1, 2, 4, 3, 9, 4, 16]
```

RDD : Set transformations

Set transformations

```
RDD1 = sc.parallelize([1,2,3,4])
RDD2 = sc.parallelize([6,5,4,3])

print (RDD1.union(RDD2).collect())
>>> [1, 2, 3, 4, 6, 5, 4, 3]
print (RDD1.union(RDD2).distinct().collect())
>>> [1, 2, 3, 4, 5, 6]
print (RDD1.intersection(RDD2).collect())
>>> [3, 4]
print(RDD1.subtract(RDD2).collect())
>>> [1,2]
```

RDD of key/value pairs

RDDs of the type RDD[(K,V)]

Transformations

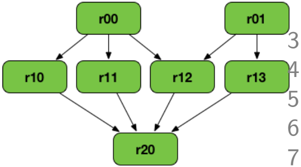
```
rdd=sc.parallelize(["foo", "bar", "baz", "baz"])
pairRDD = rdd.map(lambda word : (word, 1))
>>> [('foo', 1), ('bar', 1), ('baz', 1), ('baz', 1)]

occurrences = pairRDD.reduceByKey(lambda x, y : x + y)
print(occurrences.collect())
>>> [('foo', 1), ('bar', 1), ('baz', 2)]

occurrences = occurrences.sortBy(lambda x: x[1],
    ascending=False)
print(occurrences.collect())
>>> [('baz', 2), ('foo', 1), ('bar', 1)]
```

RDD Lineage : Logical Execution Plan

RDD Lineage (RDD operator graph or RDD dependency graph) : graph of all the parent RDDs of a RDD.



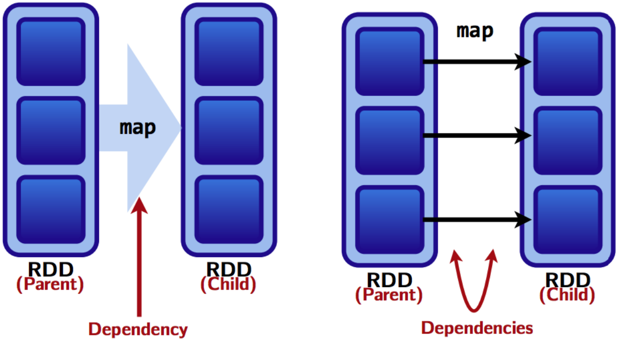

```
r00=sc.parallelize([0,1,2,3,4,5,6,7,8,9])
r01=sc.parallelize(list(range(0,91,10)))
r10=r00.cartesian(r01)
r11=r00.map(lambda x: (x,x))
r12=r00.zip(r01)
r13=r01.keyBy(lambda x: x/20)
r20 = r11.union(r12).union(r13).intersection(r10)
```

Lazy evaluation : A RDD lineage graph is hence a graph of what transformations need to be executed after an action has been called. To get RDD Lineage Graph in Spark, we can use the toDebugString() method.

How RDDs are represented ?

RDDs are made up of 4 parts :

- Partitions : Atomic pieces of the dataset.
- Dependencies : relationships between its RDD and its partition with the RDD(s) it was derived from.
- function : for computing the dataset from its parent RDDs.
- Metadata : related to its partioning scheme and data placement.



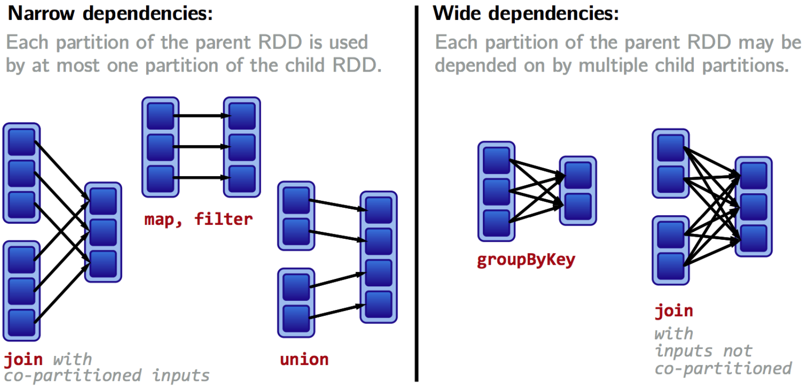
RDD Dependencies and Shuffle

A shuffle can occur when the resulting RDD depends on other elements from the same RDD or another RDD (**data must move across network**).

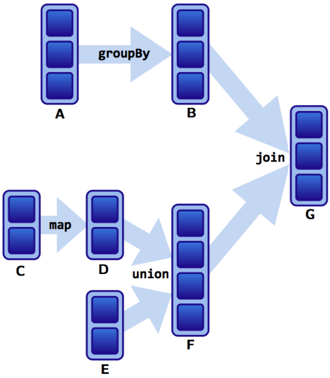
Two kinds of dependencies

- Narrow dependencies : Each partition of the parent RDD is used by at most one partition of the child RDD.
  No shuffle necessary. Optimizations like pipelining possible
- Wide dependencies : Each partition of the parent RDD may be used by multiple child partitions
  Shuffle necessary for all or some data over the network.

RDD Dependencies and Shuffle



RDD Dependencies : example

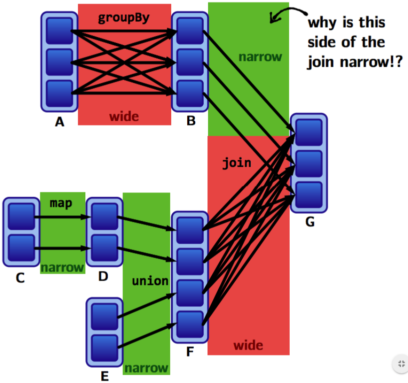


RDD Dependencies : example

Let's visualize an example program and its dependencies.

Wide transformations:
`groupBy, join`

Narrow transformations:
`map, union, join`



RDD : Actions

Actions return final result of RDD computations.

action	*description*
`reduce(func)`	aggregate the elements of the dataset using a function *func* (which takes two arguments and returns one), and should also 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 – 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 – currently not executed in parallel, instead the driver program computes all the elements
`takeSample(withReplacement, fraction, seed)`	return an array with a random sample of *num* elements of the dataset, with or without replacement, using the given random number generator seed

RDD : Actions

action	*description*
`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. Only available on RDDs of key-value pairs that either implement Hadoop's `Writable` interface or are implicitly convertible to `Writable` (Spark includes conversions for basic types like `Int`, `Double`, `String`, etc).
`countByKey()`	only available on RDDs of type `(K, V)`. Returns a `Map` of `(K, Int)` pairs with the count of each key
`foreach(`*`func`*`)`	run a function *func* on each element of the dataset – usually done for side effects such as updating an accumulator variable or interacting with external storage systems

RDD : Actions

Reduce

```
r1 = sc.parallelize([1, 2, 3, 4])
somme = r1.reduce(lambda x, y : x + y)
print(somme)
>>> 10
```

Count

```
r1 = sc.parallelize([1, 2, 3, 4])
print(r1.count())
>>> 4
print(r1.countByValue())
>>> defaultdict(<class 'int'>, {1: 1, 2: 1, 3: 1, 4: 1})
```

RDD : Actions on Pairs RDD

countByKey

```
rdd=sc.parallelize(["foo", "bar", "baz", "baz"])
pairRDD = rdd.map(lambda word : (word, 1))
print(pairRDD.countByKey())
>>> defaultdict(<class 'int'>, {'foo': 1, 'bar': 1, 'baz'
    : 2})
```

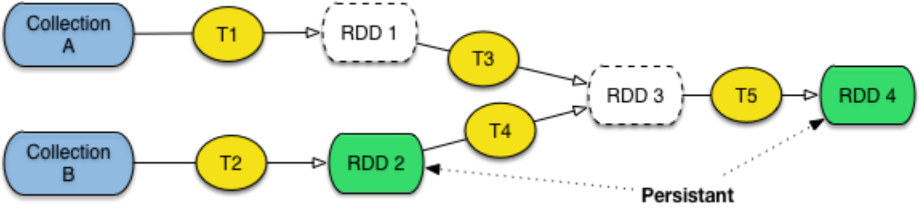
Count

```
r1 = sc.parallelize([1, 2, 3, 4])
print(r1.count())
>>> 4
print(r1.countByValue())
>>> defaultdict(<class 'int'>, {1: 1, 2: 1, 3: 1, 4: 1})
```

RDD : Persistence

- By default, each transformed RDD is recomputed each time you run an action to it.
- You can specify the RDD to be cached in memory or on disk
- `persist()` : specify the storage level to persist a RDD.
- `cache()` : default storage level : `MEMORY_ONLY`

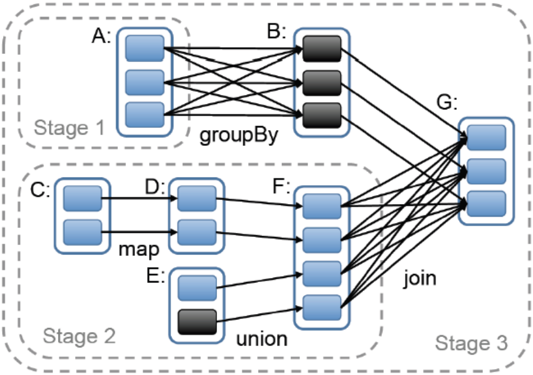
RDD : Persistence



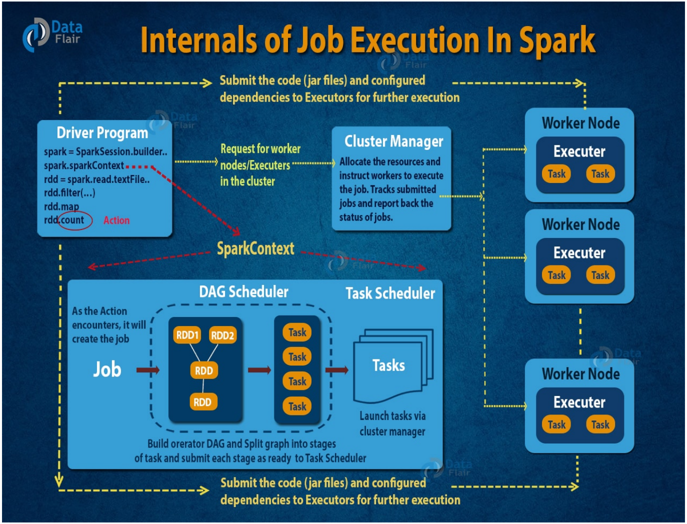
Job scheduling

To execute an action on a RDD :

- Scheduler decides the stages (set of tasks that can be runned in parallel) from the RDD lineage graph.
- Each stage contains as many pipelined transformations with **narrow dependencies**.



Anatomy of a job



Example : wordcount in Spark

Wordcount

```
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://...")
```

Running a script

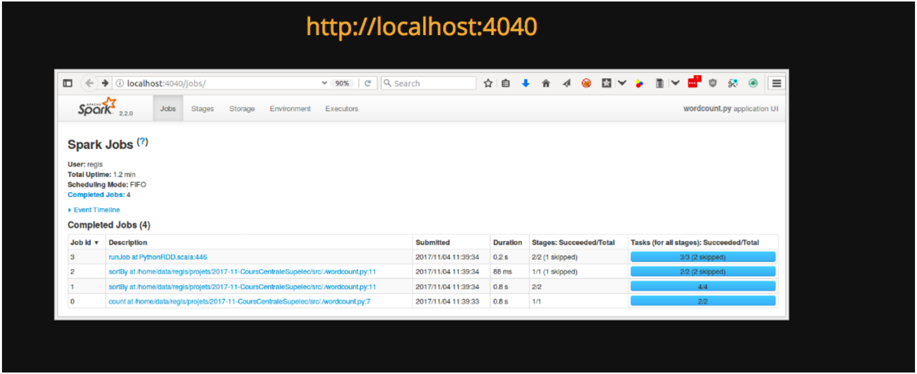
wordcount.py

```
from pyspark import SparkContext
sc = SparkContext()
rdd = sc.textFile("iliad.mb.txt")
result = rdd.flatMap(lambda sentence: sentence.split())\
    .map(lambda word: (word, 1))\
    .reduceByKey(lambda v1, v2: v1 + v2)\
    .sortBy(lambda wc: -wc[1])\
    .take(10)
print(result)
```

Running the script

```
# with all the cores
${SPARK_HOME}/bin/spark-submit ./wordcount.py ./text.txt
# with a single core
${SPARK_HOME}spark-submit --master local[1] ./wordcount.
    py ./text.txt
```

Debugging with Spark UI



Plan

1. Introduction : Limitations of Hadoop
2. Spark
   - RDD : Resilient Distributed Dataset
   - Anatomy of a job
   - Examples
3. Spark SQL and DataFrames
4. MLlib

Spark SQL and DataFrames

Spark SQL : a programming module for structured data processing :

- provides a programming abstraction called DataFrame.
- acts as distributed SQL query engine

What is a dataframe ?

A distributed collection of rows under named columns (same as RDBMS) that have as characteristics :

- **Immutable** : We can create DataFrame once but can not change it.
- **Lazy Evaluations** : A task is not executed until an action is performed
- **Distributed** : DataFrame are distributed in nature.

Spark DataFrames

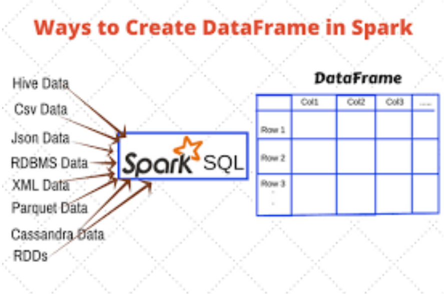
Why ?

- Processing large collection of structured or semi-structured data.
- Allows higher level languages like SQL.
- Optimize execution plan on queries on these data.

Creating a DataFrame

Different ways :

- Using and loading data from different data format.
- From existing RDDs.
- By programmatically specifying schema.



Creating a DataFrame

Creating DataFrame from RDD

```
from pyspark.sql import SQLContext,Row
sqlContext = SQLContext(sc)
l = [('John',25),('Jacques',22),('Sally',20),('Emma',26)]
rdd = sc.parallelize(l)
people = rdd.map(lambda x: Row(name=x[0], age=int(x[1])))
schemaPeople = sqlContext.createDataFrame(people)
schemaPeople.printSchema()
>>>
root
 |-- age: long (nullable = true)
 |-- name: string (nullable = true)
```

Creating a DataFrame

Creating DataFrame from a csv file

```
df = sqlContext.read.format('com.databricks.spark.csv').
    options(header='true', inferschema='true', sep=';').
    load('arbres.csv')
df.printSchema()

root
 |-- GEOPOINT: string (nullable = true)
 |-- ARRONDISSEMENT: integer (nullable = true)
 |-- GENRE: string (nullable = true)
 |-- ESPECE: string (nullable = true)
 |-- FAMILLE: string (nullable = true)
 |-- ANNEE PLANTATION: integer (nullable = true)
 |-- HAUTEUR: double (nullable = true)
 |-- CIRCONFERENCE: double (nullable = true)
 |-- ADRESSE: string (nullable = true)
 |-- NOM COMMUN: string (nullable = true)
 |-- VARIETE: string (nullable = true)
 |-- OBJECTID: integer (nullable = true)
```

A lot of useful dataframe manipulations

Show first n observation

```
>>> df.head(2)

[Row(GEOPOINT='(48.857140829, 2.29533455314)',
    ARRONDISSEMENT=7, GENRE='Maclura', ESPECE='pomifera',
    FAMILLE='Moraceae', ANNEE PLANTATION=1935, HAUTEUR
    =13.0, CIRCONFERENCE=None, ADRESSE='Quai Branly,
    avenue de La Motte-Piquet, avenue de la Bourdonnais,
    avenue de Suffren', NOM COMMUN='Oranger des Osages',
    VARIETE=None, OBJECTID=6, NOM_EV='Parc du Champs de
    Mars'),
 Row(GEOPOINT='(48.8685686134, 2.31331809304)',
    ARRONDISSEMENT=8, GENRE='Calocedrus', ESPECE='
    decurrens', FAMILLE='Cupressaceae', ANNEE PLANTATION
    =1854, HAUTEUR=20.0, CIRCONFERENCE=195.0, ADRESSE='
    Cours-la-Reine, avenue Franklin-D.-Roosevelt, avenue
    Matignon, avenue Gabriel', NOM COMMUN='Cèdre à encens'
    , VARIETE=None, OBJECTID=11, NOM_EV='Jardin des Champs
     Elysées')]
```

A lot of useful dataframe manipulations

Count the number of rows

```
>>> df.count()
97
```

Getting the columns

```
df.columns

['GEOPOINT',
 'ARRONDISSEMENT',
 'GENRE',
 'ESPECE',
 'FAMILLE',
 'ANNEE PLANTATION',
 'HAUTEUR',
 'CIRCONFERENCE',
 'ADRESSE',
 'NOM COMMUN',
```

A lot of useful dataframe manipulations

Summary statistics

```
df.describe().show()

+-------+--------------------+--------------------+-------+--

|summary|            GEOPOINT|    ARRONDISSEMENT|  GENRE|
         ESPECE|        FAMILLE|  ANNEE PLANTATION|
      HAUTEUR|      CIRCONFERENCE|                ADRESSE|
      NOM COMMUN|  VARIETE|            OBJECTID|
     NOM_EV|
+-------+--------------------+--------------------+-------+--

|   count|                  97|                 97|     97|
                97|                  97|                77|
             96|                  92|                     86|
                97|         11|                 97|
           97|
|    mean|                null|13.608247422680412|   null|
              null|                null|1869.8831168831168|
```

A lot of useful dataframe manipulations

Data selection

Selecting columns

```
df.select("ARRONDISSEMENT","GENRE").show(4)
```

Number of distinct objects

```
df.select('GENRE').distinct().count()
```

Drop the all rows with null value

```
df.select('GENRE').distinct().count()
```

Fill the null values with a constant

```
df.fillna(-1)
```

And many others :
http ://spark.apache.org/docs/2.2.0/api/python/pyspark.sql.html

Apply SQL Queries on DataFrame

```
df.registerTempTable('arbres_table')
sqlContext.sql('select ARRONDISSEMENT from arbres_table')
    .show(5)


+-------------+
|ARRONDISSEMENT|
+-------------+
|            7|
|            8|
|            9|
|           12|
|           12|
+-------------+
only showing top 5 rows
```

Plan

1 Introduction : Limitations of Hadoop

2 Spark
- RDD : Resilient Distributed Dataset
- Anatomy of a job
- Examples

3 Spark SQL and DataFrames

4 MLlib

MLlib

- The Spark **Machine Learning API**.
- Supports different kind of algorithms :
  - **mllib.classification** : various methods for binary classification, multiclass classification and regression analysis.
  - **mllib.clustering** : unsupervised learning.
  - **mllib.fpm** : Frequent pattern matching.
  - **mllib.linalg** : utilities for linear algebra.
  - **mllib.recommendation** : collaborative filtering.
  - **mllib.regression**.