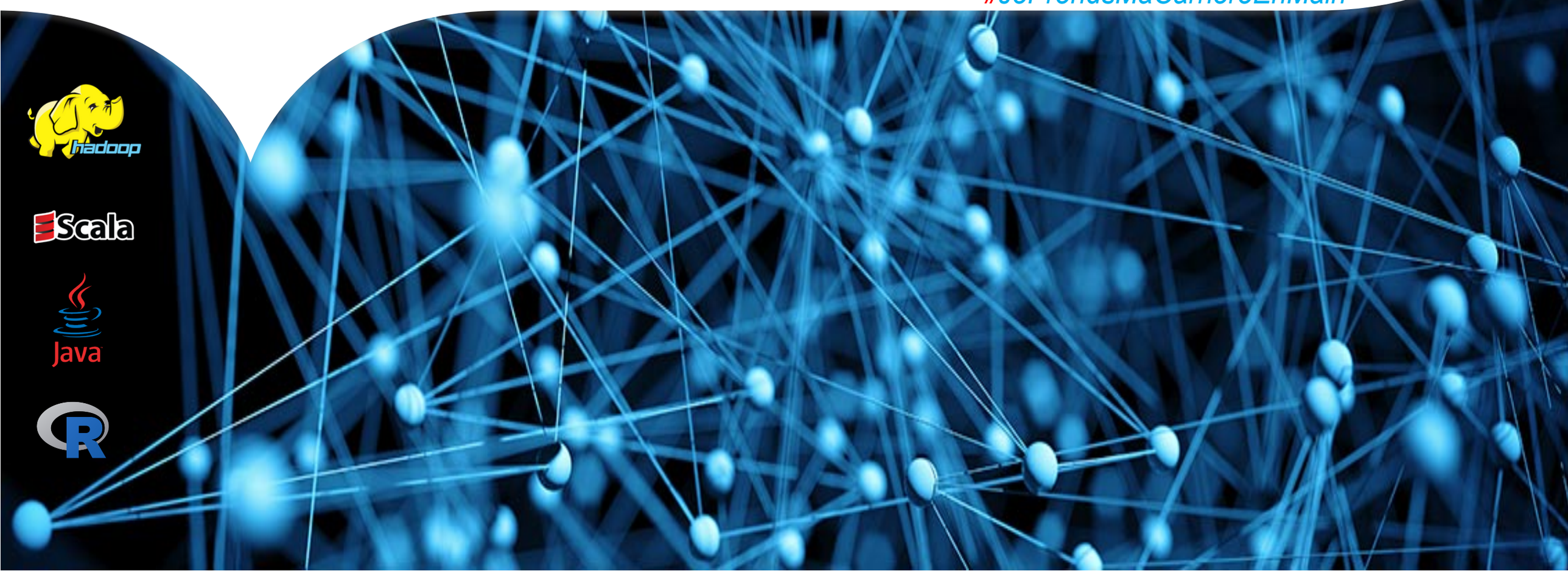
Introduction to Spark

Farhang Habibi

#JePrendsMaCarrièreEnMain



FORMATION-BIGDATA.COM

www.formation-bigdata.com

Contents

What is Spark

- Spark is a big data processing engine

- Can be run either in local or on a cluster :
  - Apache Hadoop YARN, Apache Mesos, Spark Standlone
  - Downloading Apache Spark
  - Installing pyspark with Anaconda in your local machine

- Interactive programming (Read/Evaluate/Print/Loop)
  - The Spark instance is previously defined
  - Using your command line on local or gateway machine :
    cml> pyspak - - local (or - - yarn)
  - Using notebooks : Zeppelin, Jupyter, Databricks

- Application programming
  - Programming in a text editor
  - You build the Spark instance
  - cml> spark-submit - - local (or - - yarn) yourCode.py

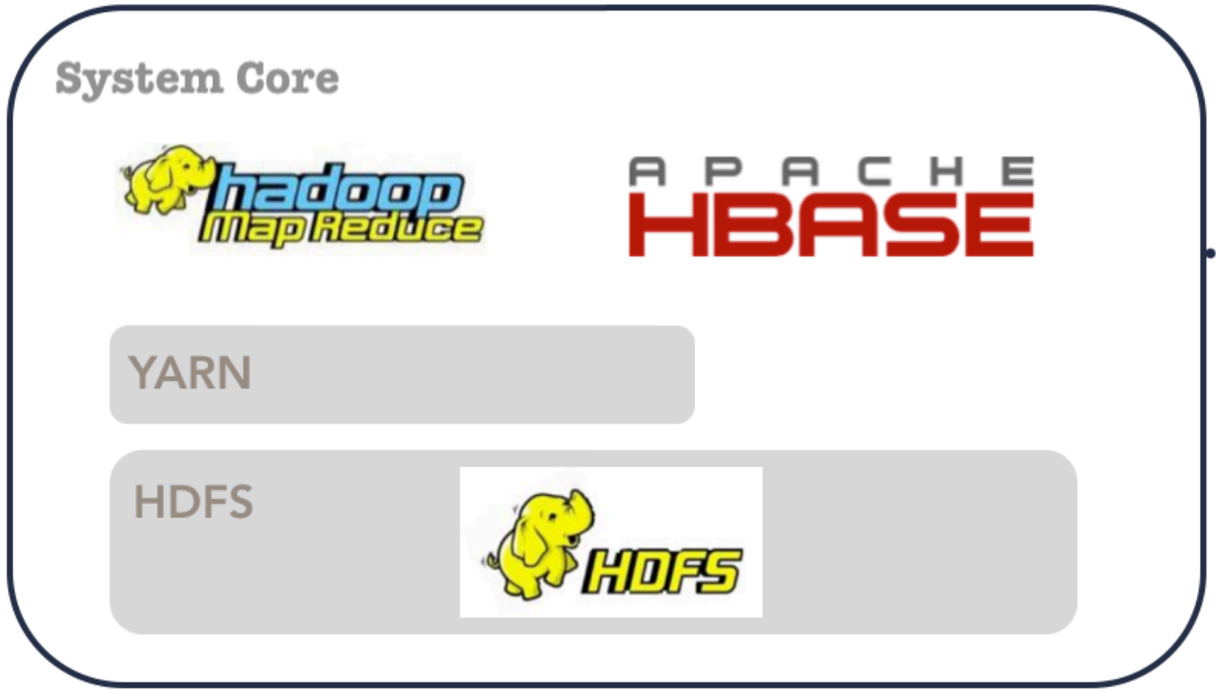
- Spark is initially written in Scala and is available in Python, R, Java etc.
  - We focus on Spark in Python (PySpark)

Contents

Hadoop cluster

It is a solution for big data storage and processing problem

- A cluster with a master node and several slave (worker) nodes.
- Data are replicated, partitioned and distributed across the cluster nodes.
- Master node contains the meta data of all files and frequently check the nodes state.



Core of Hadoop system

HDFS : file system to store the files.
YARN : resource manager
MapReduce : data processing engin
HBase : NoSQL database

HDFS

HDFS : Hadoop Distributed File System

- Distributed file system where the cluster is based on.
- Main storage system for Hadoop applications
- Distributes the files in different cluster nodes (machines)
- Abstraction layer for the cluster storage.

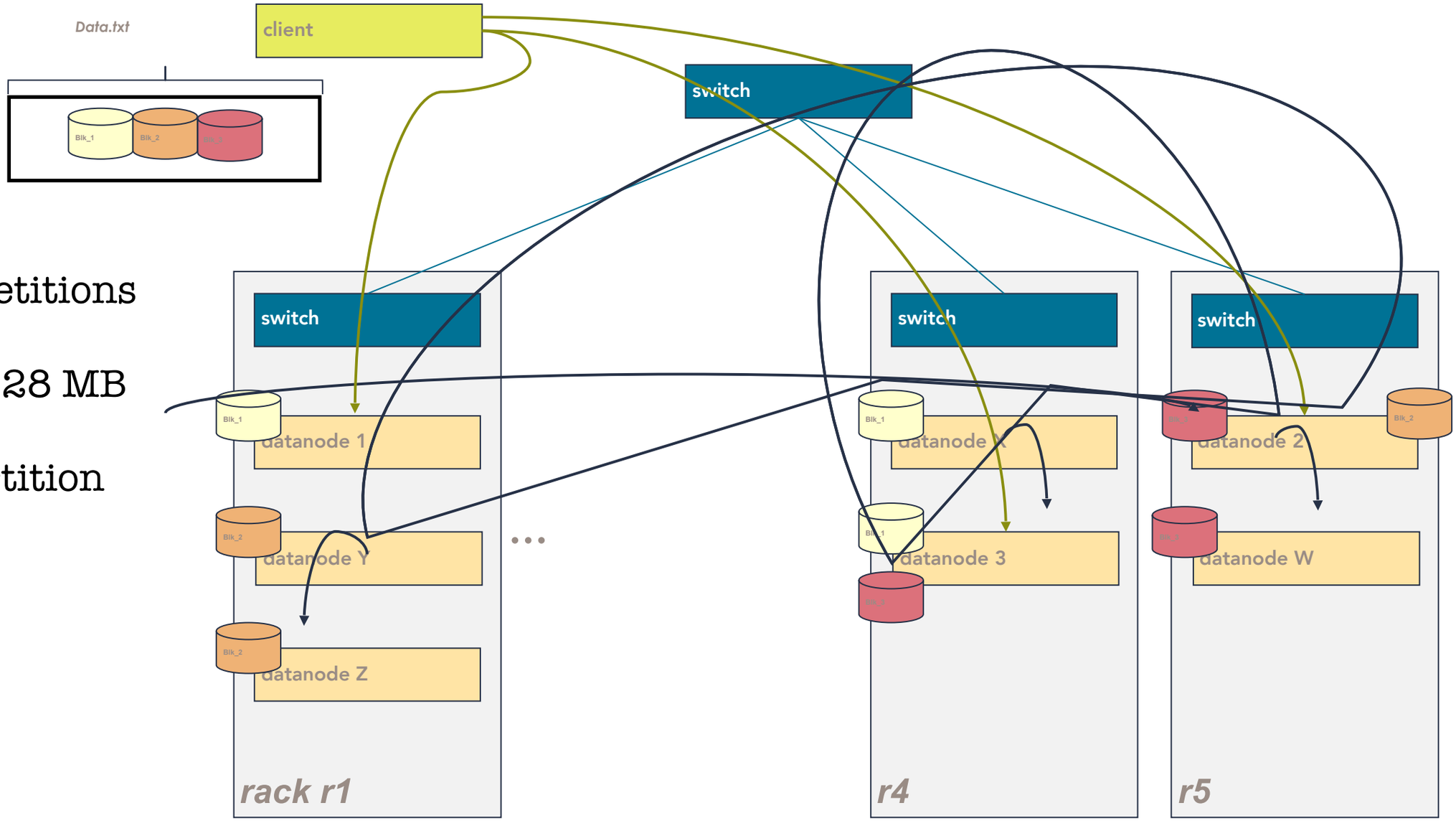
HDFS

➢File 1TB = 3TB storage

Split the file to petitions

Partition size = 128 MB

3 replica per partition

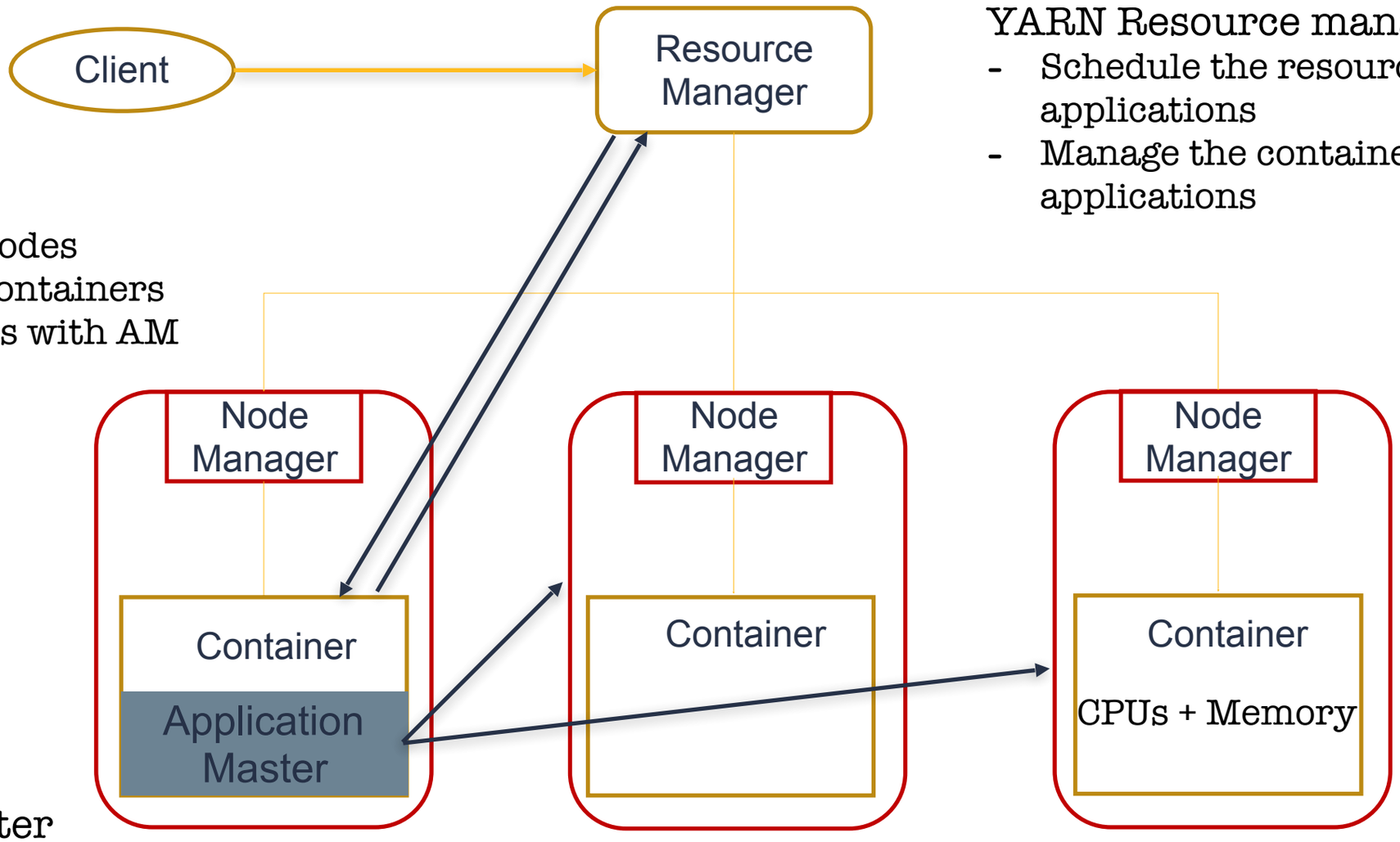


YARN

YARN : Yet Another Resource Negotiator

- It manages the computing resources of a cluster to optimise data processing
- YARN ResourceManager (one per cluster) receives the job application and manage the available cluster resources.
- YARN NodeManager is executed in all worker nodes to launch and monitor the containers (memory + CPUs)
- YARN ApplicationMaster (one per application) located in one of the worker nodes and monitors the worker performances. Asks for more resources from the ResourceManager if it is needed.

YARN



YARN Resource manager
- Schedule the resource usage for applications
- Manage the containers for different applications

Node manager
- Installed on nodes
- Monitor the containers
- Communicates with AM

Application master
- One per application
- located in a container
- monitor the executing node performance
- asks for more resources from RM

MapReduce

Highly parallel distributed data processing engine for big data.

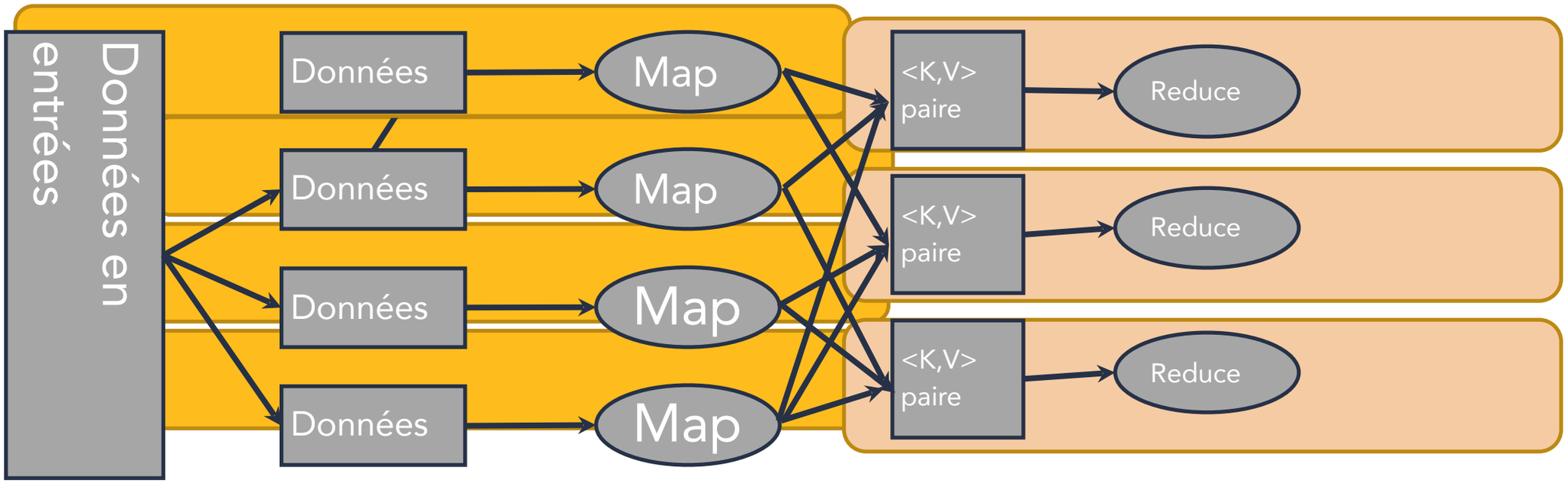
The programs based on MR are automatically parallelised and executed on different nodes of a cluster

MapReduce applies two different operations on data :

< MAP : read the data from the HDFS and transforms them to key-value pairs

< REDUCE : applies (mathematical) operations on all values with same key produced by MAP and save the results on the HDFS.

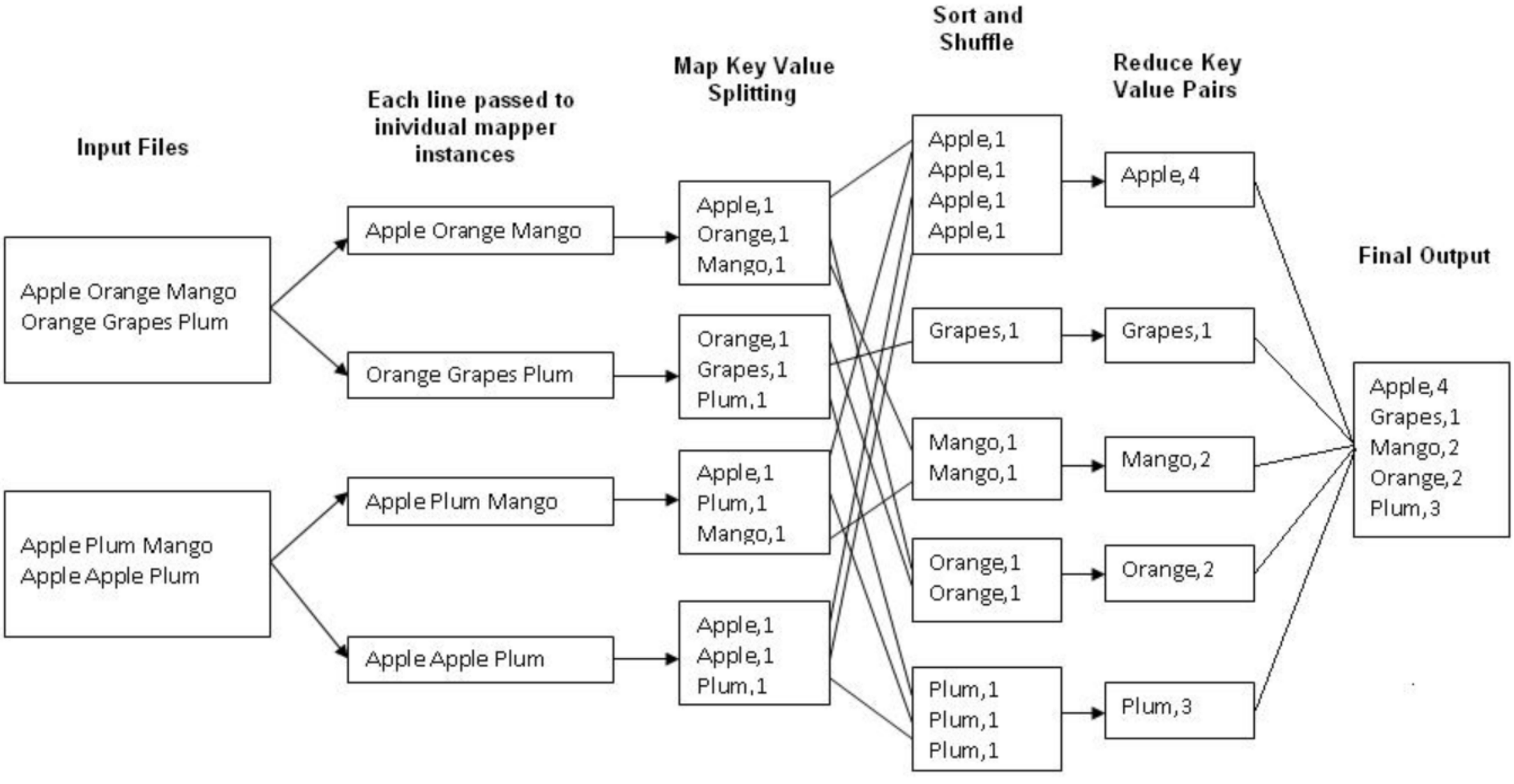
MapReduce



Entrée HDFS

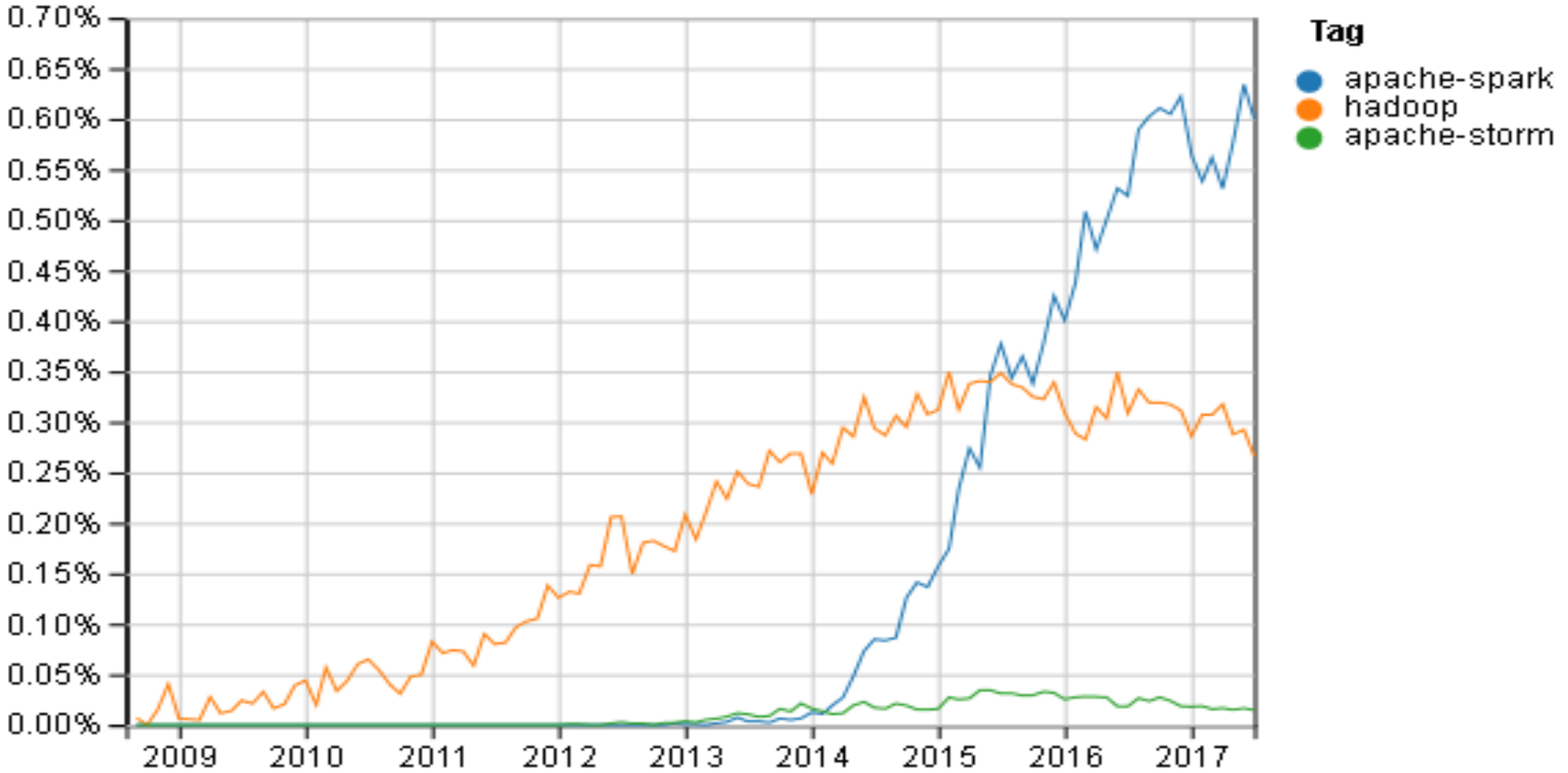
Sortie HDFS

MapReduce word-count example



Contents

Spark emergence



- StackOverflow trend : people get more interested in Spark compared to other Bigdata technologies

Spark emergence

Same trend in google searches

Spark components

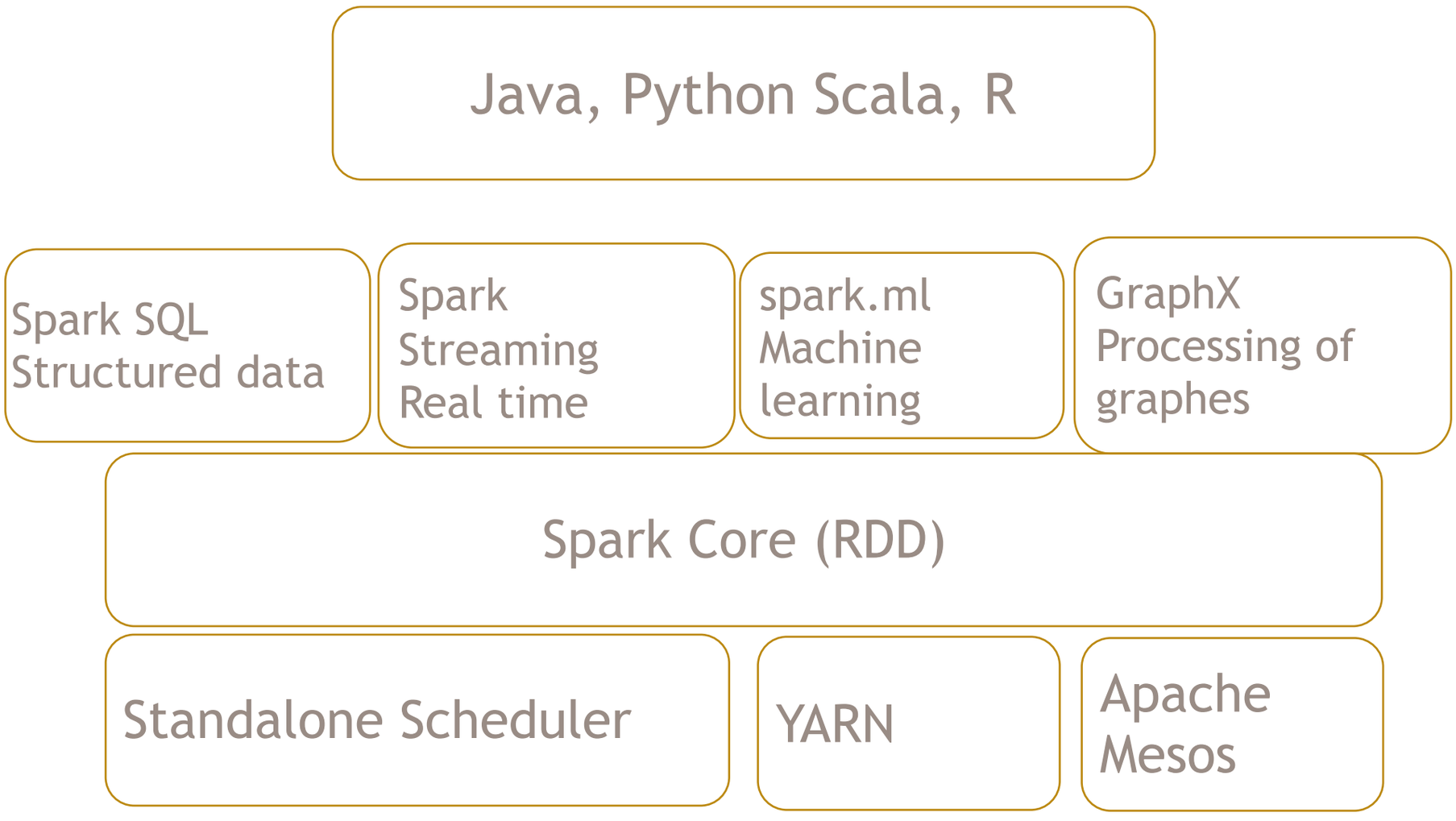
Spark is an open source framework for distributed (parallel) computing of big quantity of data.

Spark has several advantages compared to other big data technologies such as MapReduce.

Spark framework has unified different tools for different big data processing necessities :

- Spark SQL to process the structured data.
- Spark ML to apply different Machine Learning estimators.
- Spark Streaming for real time data processing.
- GraphX for graphe processing.

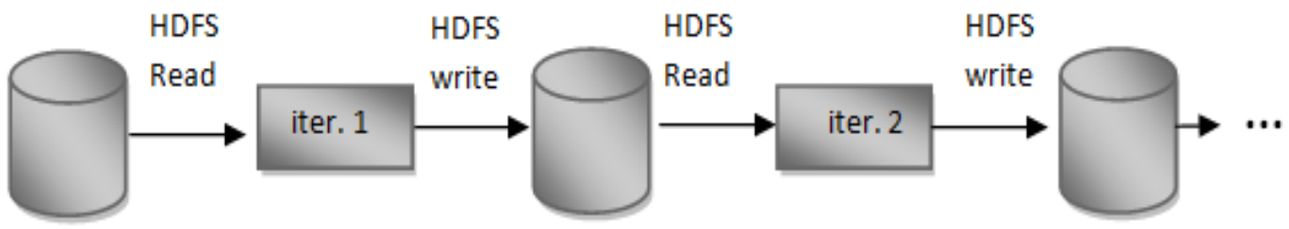
Spark components



Contents

Spark vs MapReduce

- MapReduce has quite simplified the big data analysis by using large cluster of machines susceptible to fail.
- However users need to make more complicated applications, like iterative operations (e.g. for ML) or the interactive operations (e.g. launching several queries on the same data sample).
- In iterative process : MapReduce makes a job for each iteration and shares the data between the jobs by writing the data on the HDFS.

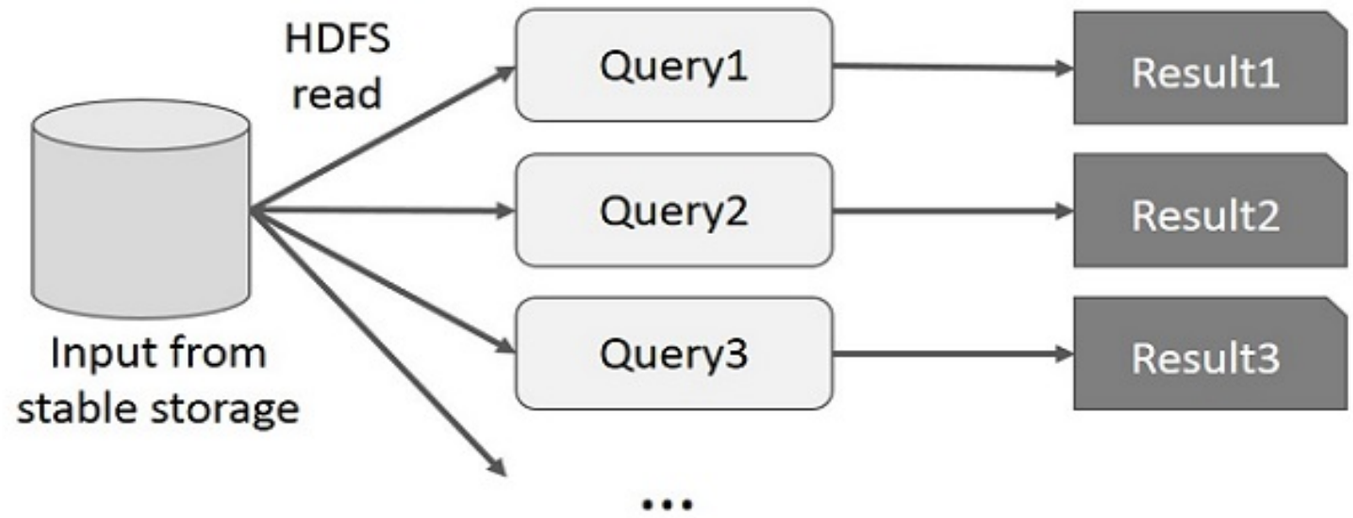


Iterative processing with MapReduce

- Both iterative and interactive applications need to share the data more rapidly entre the parallel jobs.

Spark vs MapReduce

➢In interactive operations, several queries launch on the same dataset : each query should reread the data which is time consuming.

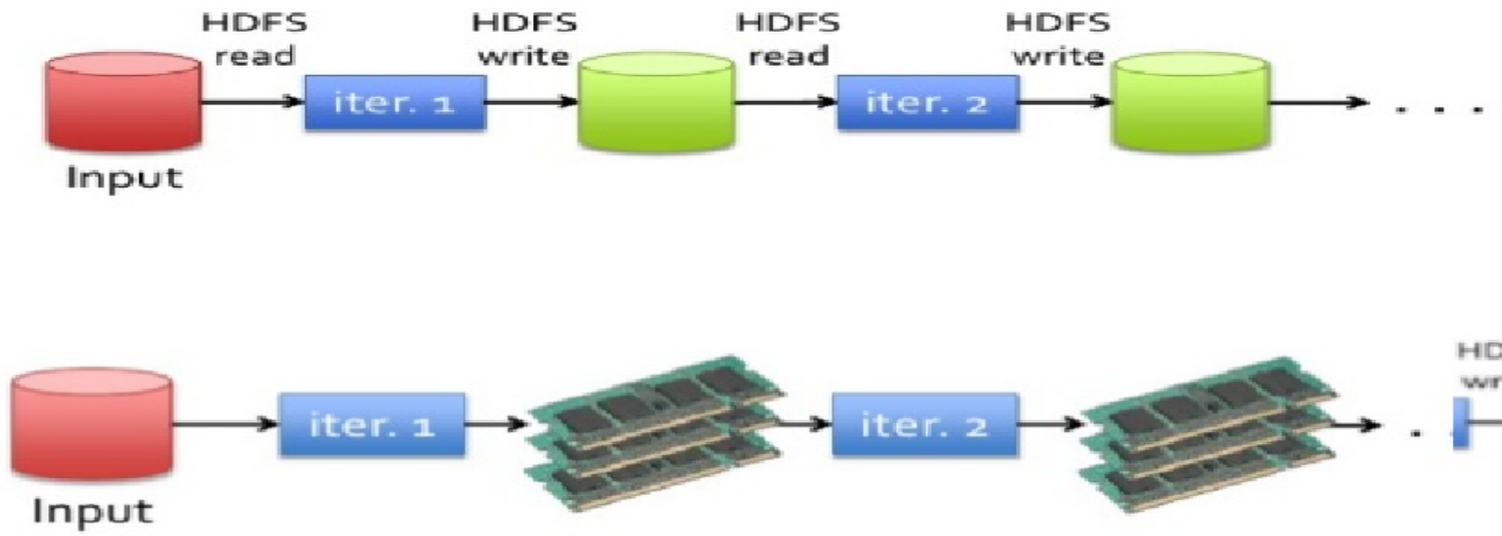


➢Data sharing is slow in MapReduce due to the replication, serialisation, and disc I/O.

➢More than 90% of the time for the Hadoop applications execution is consumed by HDFS read-write operations.

Spark vs MapReduce

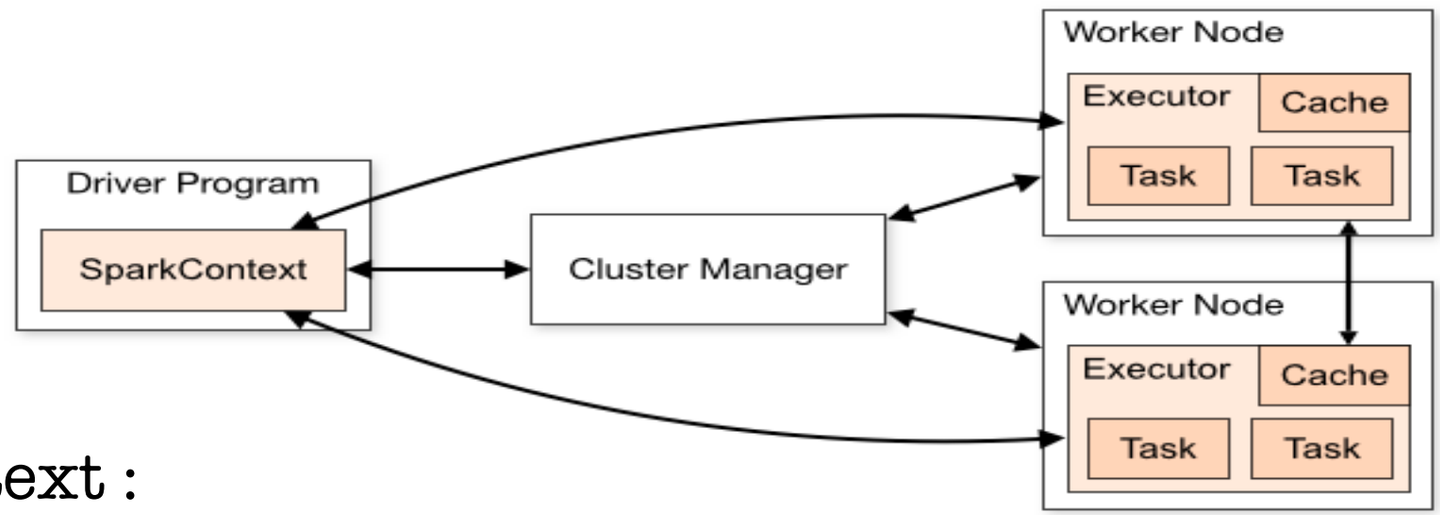
Spark : data processing based in the shared memory (RAM) :
the intermediate results are saved in the memory. This makes Spark 10 times faster than Hadoop.



Interactive processing : Spark enables the users to execute real time processing on the cluster without putting the codes in a java, scala ou python package.

Contents

Spark architecture



SparkContext :

➔ A Spark application is an instance of SparkContext class.

➔ It is used to create the RDDs, the necessary codes and objects for Spark application(in Zeppelin/Databricks/spark-shell “sc” represents the instance of SparkContext).

Spark Driver :

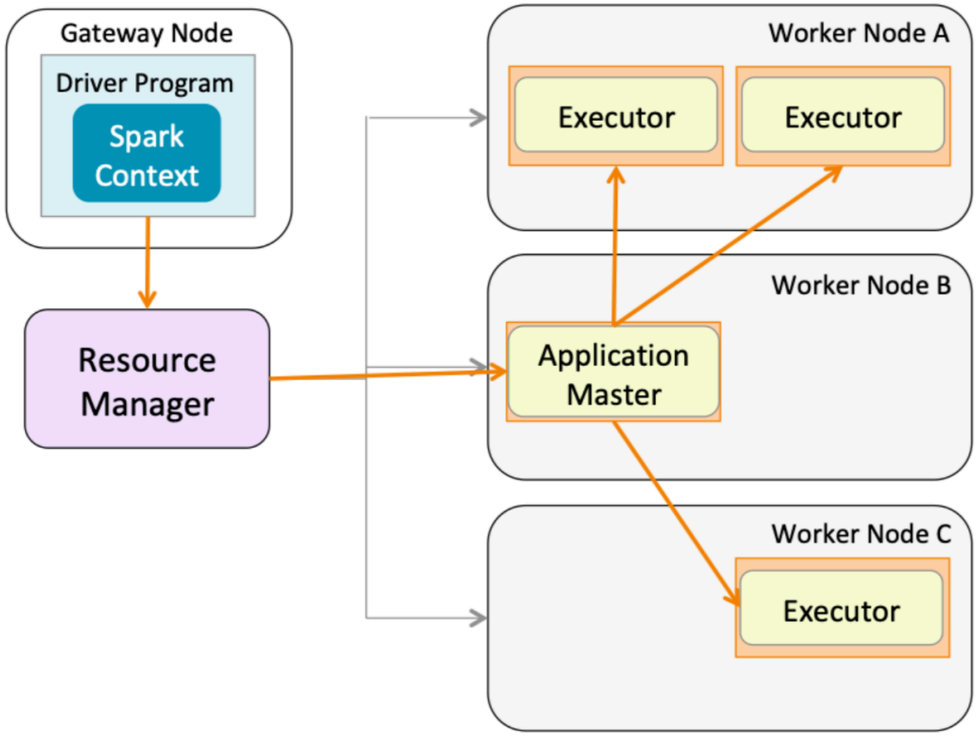
➔ Driver manages the Spark execution and creates the SparkContext instance.

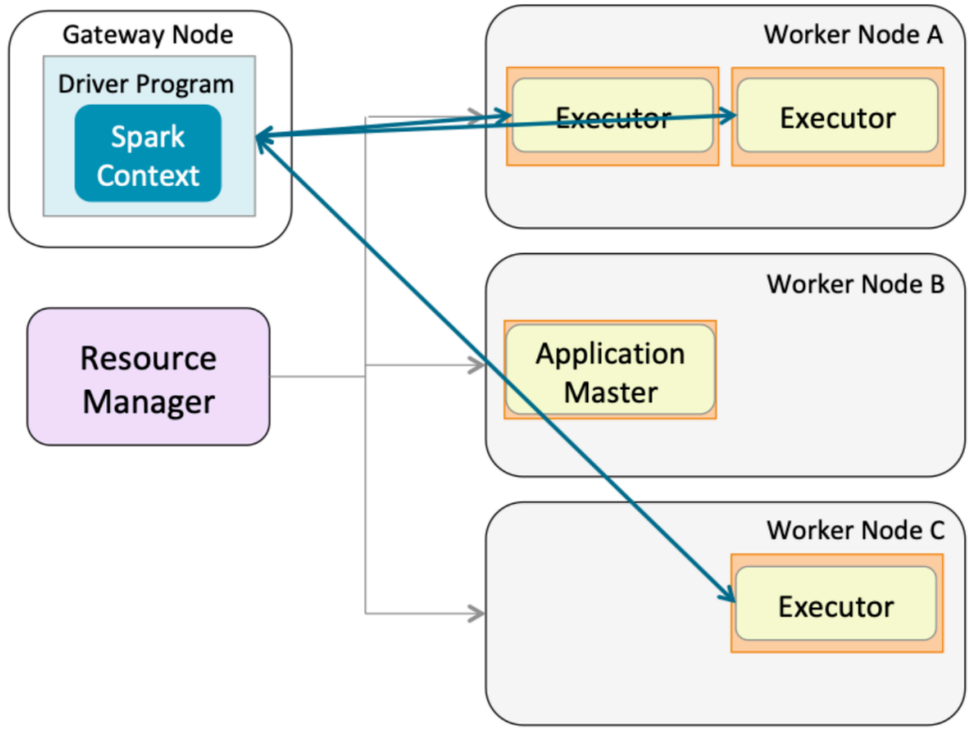
➔ It writes and prints the logs / Monitoring.

Spark executor

➔ Executes the application on local or on cluster nodes.

Spark architecture





YARN Resource manager

- Manage the containers for different applications
- Schedule the resource usage for applications

Application master

- One per application
- located in a container
- monitor the executors performance
- asks for more resources from RM

CLOUDERA
Educational Services

Contents

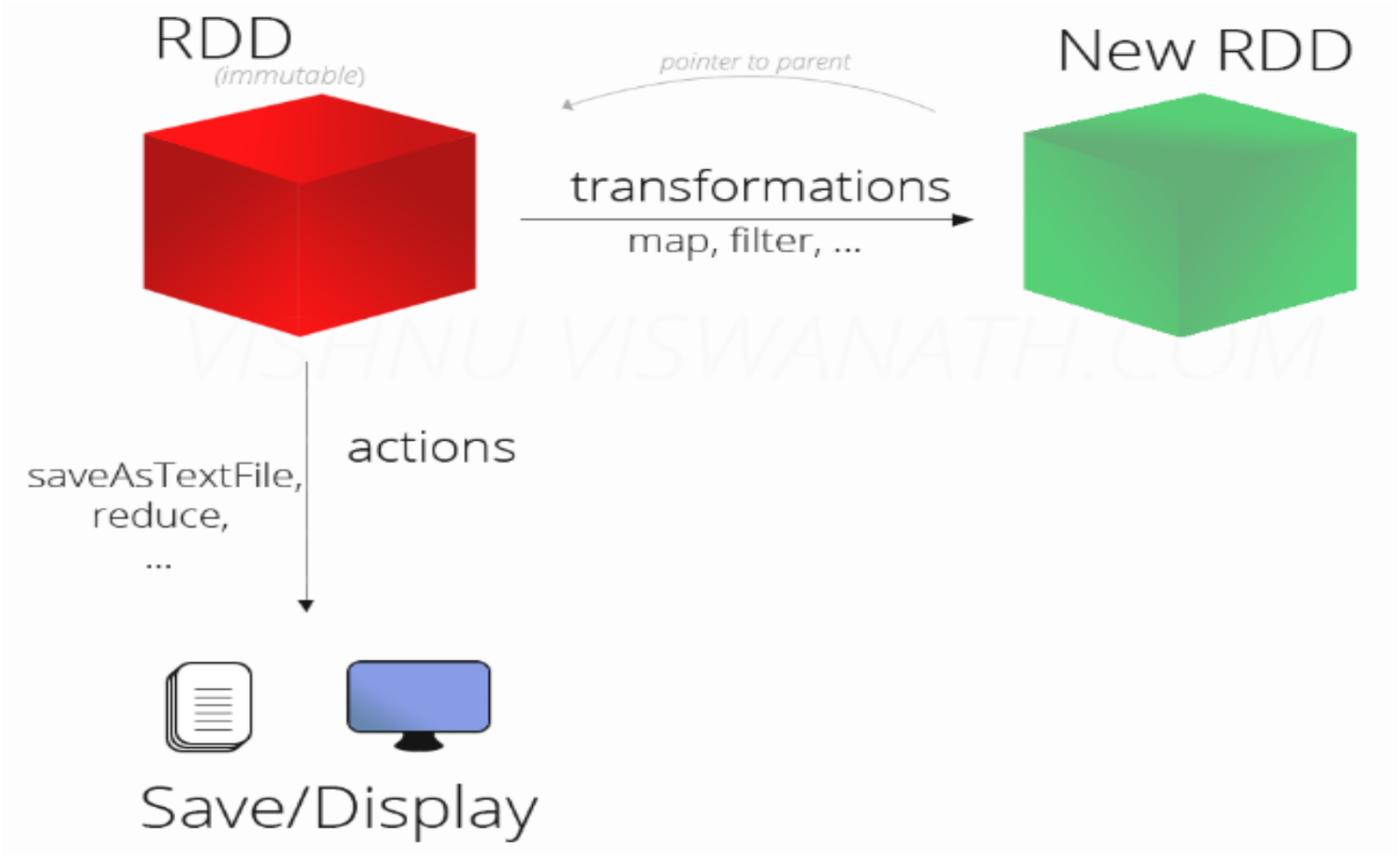
Resilient Distributed Dataset (RDD)

- The core of Spark engine is based on RDD concept.
- RDD is an in-memory object.
- Can be either produced from external sources or by programming (Dataset).
- It is built by smaller data partitions (Distributed)
  - data partitions reside in different executors across the cluster.
- If a partition is lost it will be automatically reproduced (Resilient)
  - RDDs are fault-tolerant. If an executor fails, application master allocate another one.
- RDD is an un-structured dataset.
  - No schema
- RDD can be transformed by using anonymous Lambda functions.

RDD

➢RDD is am immutable collection:

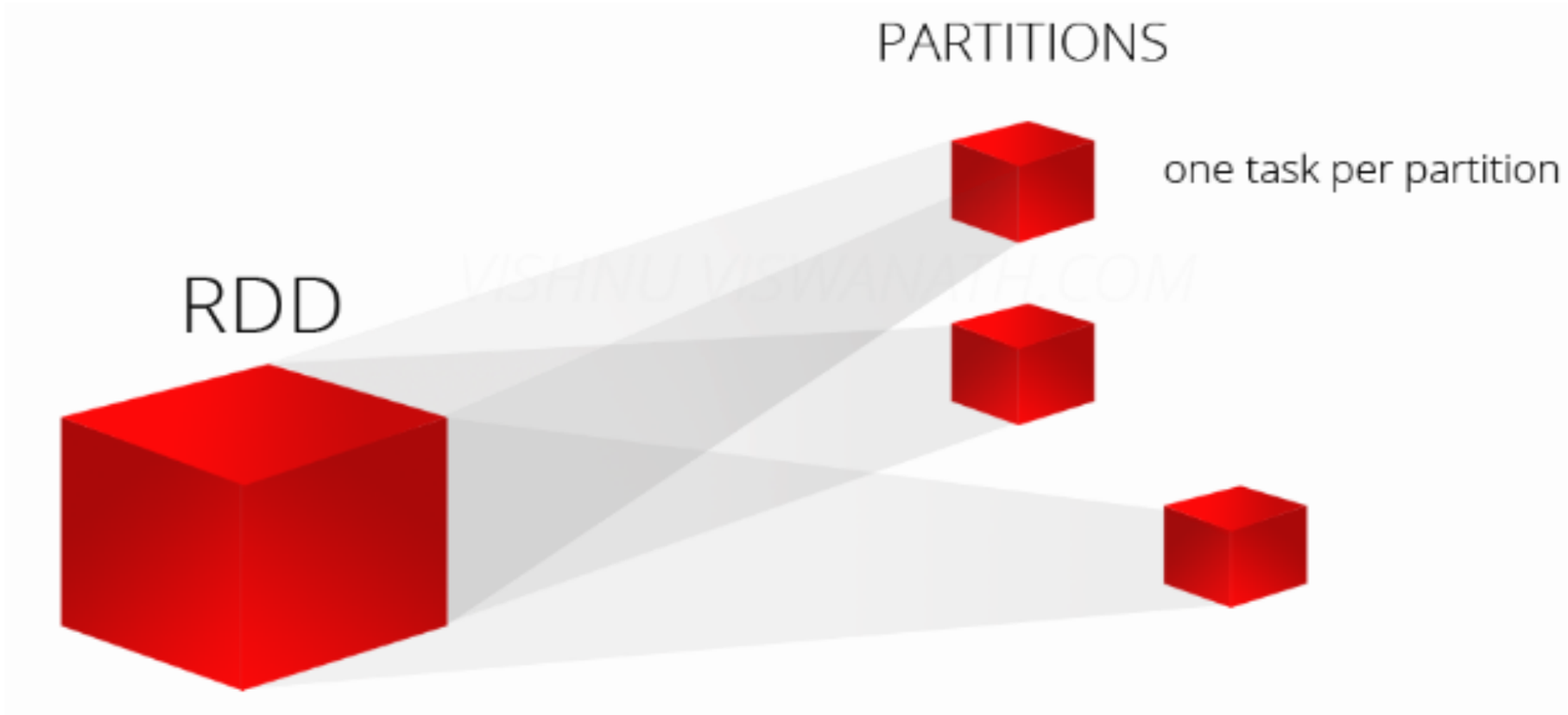
each operation creates a new RDD.



RDD

➢ RDD is a partitioned collection of objects :

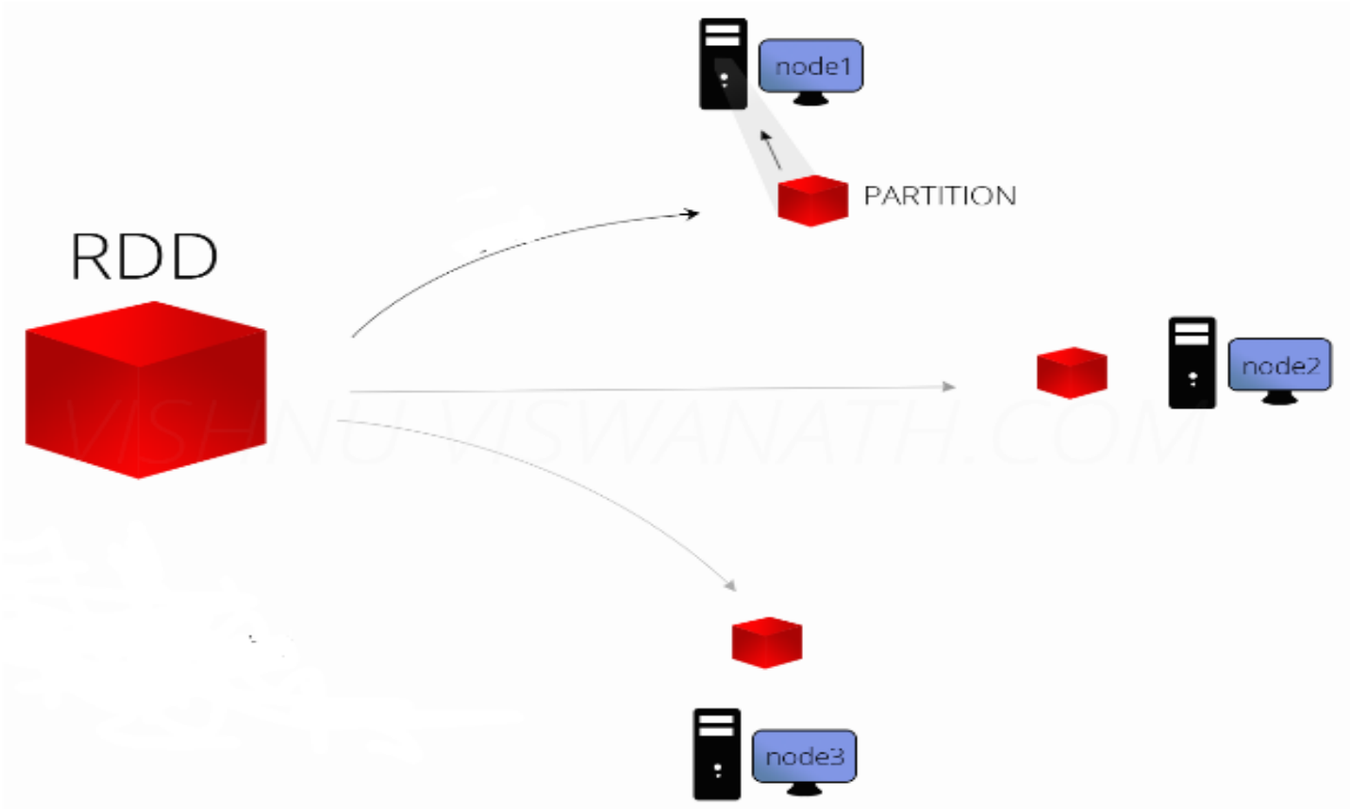
more partitions more parallelism.



RDD

➢RDD is fault-tolerant :

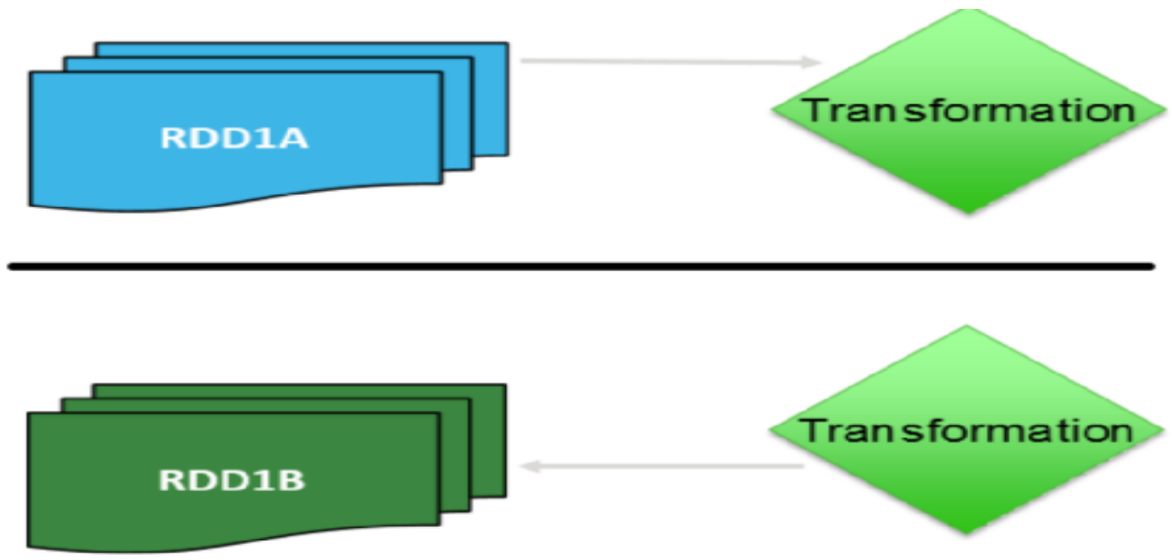
if an RDD partition fails, it will be recreated on another node.



Contents

RDD operations

➔ Transformations : make a new RDD.



➔ Actions : computes a new value.

RDD operations

- RDDs accept different element types :
  - Integer, string, boolean, character
  - List, tuple, dictionary, collections with nested-structure

- Pair RDDs contain key-value elements in tuple form.

- RDDs can be created from different sources :
  - External files with different formats : text, csv, json, etc
  - From another RDD
  - Created programmatically
  - By converting dataframes or datasets (latter only for Scala)
  - From a database : JBDC, Casandra, HBase

- RDDs are created by applying SparkContext methods.
  - In interactive programming “sc” is the default instance of SparkContext

RDD creation

- Exemples :


```
// Crating an RDD from a python collection :
data = [1, 2, 3, 4, 5]
distData = sc.parallelize(data)

// Crating an RDD from a file :
distFile = sc.textFile("data.txt")
```

RDD creation

RDD example

File: descriptif.txt

Le fichier texte suivant va être transformé en RDD.
Le fichier est transformé en RDD par la function textFile.
Chaque ligne de ce fichier est un élément de l'RDD.

```
descriptif_RDD =
sc.textFile("descriptif.txt")
```

RDD: descriptif_RDD

Le fichier texte suivant va être transformé en RDD.
Le fichier est transformé en RDD par la function textFile.
Chaque ligne de ce fichier est un élément de l'RDD.

RDD creation

RDD example

RDD: descriptif_RDD

Le fichier texte suivant va être transformé en RDD.
Le fichier est transformé en RDD par la function textFile.
Chaque ligne de ce fichier est un élément de l'RDD.

descriptif_RDD.count()

Valeur = 3

RDD creation

RDD example (map)

RDD: descriptif_RDD

Le fichier texte suivant va être transformé en RDD.
Le fichier est transformé en RDD par la function textFile.
Chaque ligne de ce fichier est un élément de l'RDD.

descriptif_RDD.map(lambda line : line.toUpperCase())

LE FICHIER TEXTE SUIVANT VA ÊTRE TRANSFORMÉ EN RDD.
LE FICHIER EST TRANSFORMÉ EN RDD PAR LA FUNCTION TEXTFILE.
CHAQUE LIGNE DE CE FICHIER EST UN ÉLÉMENT DE L'RDD.

RDD transformations

Transformation	Signification
map(func)	Retourne un nouveau RDD obtenu en appliquant la fonction func à chaque item du RDD de départ.
filter(func)	Retourne un nouveau RDD obtenu en sélectionnant les items de la source pour lesquels la fonction func retourne “vrai”.
flatMap(func)	Similaire à map mais chaque item du RDD source peut être transformé en 0 ou plusieurs items ; retourne une séquence (Seq) plutôt qu’un seul item.
union(otherDataset)	Retourne un RDD qui est l’union des items du RDD source et du RDD argument (otherDataset).
distinct()	Retourne un RDD qui est obtenu du RDD source en éliminant les doublons des items.
groupBy	Retourne un RDD qui est regroupé par un élément d’un RDD source.

Liste des actions et transformations : http://spark.apache.org/docs/2.1.1/programming-guide.html

RDD actions

Action	Signification
reduce(func)	Agréger les éléments du RDD en utilisant la fonction func (qui prend 2 arguments et retourne 1 résultat).
collect()	Retourner tous les items du RDD comme un tableau au programme driver. A utiliser seulement si le RDD a un volume faible (par ex., après des opérations de type filter très sélectives).
count()	Retourner le nombre d'items du le RDD.
take(n)	Retourner un tableau avec les n premiers items du RDD.
first()	Retourner le premier item du RDD (similaire à take(1)).
countByKey()	Pour RDD de type (clé, valeur), retourne l'ensemble de paires (clé, Int) avec le nombre de valeurs pour chaque clé.
saveAsTextFile(path)	Écrit les items du RDD dans un fichier texte dans un répertoire du système de fichiers local, HDFS ou autre fichier supporté par Hadoop. Spark appelle toString pour convertir chaque item en une ligne de texte dans le fichier.

RDD operations example

```
// Creating an RDD from a collection (transformation) :
data = [1, 2, 3, 4, 5]
distData = sc.parallelize(data)

// Addin 1 to each element (transformation) :
rdd1 = distData.map(lambda e : e + 1)
( [2,3,4,5,6] )

// Filtering the element > 3 (transformation) :
rdd2 = rdd1.filter(lambda e : e > 3)
( [4,5,6] )

// Summing over all elements  (action) :
somme = rdd2.reduce(lambda e1, e2 : e1 + e2)
( 15 )
// Number of elements in an RDD (action) :
distData.count()
( 5 )
// Printing all elements of an RDD (action) :
distData.collect()
( [1, 2, 3, 4, 5] )
```

Spark : A lazy performer

- The very first created RDD is the parent RDD.
- Other RDDs transformed from the parent are child RDDs.
- A series of transformations ended by an action is called a query.
- The transformed RDDs are not created before you ask an action.
  - Spark makes an execution plan without running it (lazy performer)
- When an action is requested, the execution plan is run.
  - Then the RDDs are erased from the memory.
- If another query is executed, Spark rebuilds the RDDs from the source.

Contents

Running Spark jobs

➢ Two possibilities to run spark application (Python):

- Interactive : Running pyspark shell command or a notebook (Zeppelin/Jupyter/Databricks)

```
(base) PySpark |play> pyspark
Python 3.7.6 (default, Jan  8 2020, 13:42:34)
[Clang 4.0.1 (tags/RELEASE_401/final)] :: Anaconda, Inc. on darwin
Type "help", "copyright", "credits" or "license" for more information.
2020-06-12 14:52:53 WARN  NativeCodeLoader:62 - Unable to load native-hadoop l
ibrary for your platform... using builtin-java classes where applicable
Setting default log level to "WARN".
To adjust logging level use sc.setLogLevel(newLevel). For SparkR, use setLogLe
vel(newLevel).
2020-06-12 14:52:55 WARN  Utils:66 - Service 'SparkUI' could not bind on port
4040. Attempting port 4041.
2020-06-12 14:52:55 WARN  Utils:66 - Service 'SparkUI' could not bind on port
4041. Attempting port 4042.
Welcome to
      ____              __
     / __/__  ___ _____/ /__
    _\ \/ _ \/ _ `/ __/  '_/
   /__ / .__/\_,_/_/ /_/\_\   version 2.4.0
      /_/

Using Python version 3.7.6 (default, Jan  8 2020 13:42:34)
SparkSession available as 'spark'.
>>> sc.parallelize( ["Hello", "world"] ).collect()
['Hello', 'world']
>>>
```

- Application programming : Write the whole code in a .py file and execute it via spark-submit command.

Application Programming

SparkContext creation :

```
# spark libraries
from pyspark import SparkContext, SparkConf

if __name__== '__main__':

logging.info('--------------- manipSimpleRDD.py ---------------')
# configuring the spark application
conf = SparkConf().setAppName('simpleRDDs').setMaster('local[*]')
# constructing the spark context
sc = SparkContext(conf = conf)
```

Executing a Spark application in local from command-line:

```
spark-submit manipSimpleRDD.py
```

Running an application on cluster

Executing a Spark application in a cluster

```
spark-submit --master yarn --driver-memory 3G --num-executors 15 --executor-memory 6G --executor-cores 2
```

Deployment mode : client (default)

The driver create the SparkContext instance on the gateway/edge

```
spark-submit --master yarn --deploy-mode client yourCode.py
```

Deployment mode : cluster

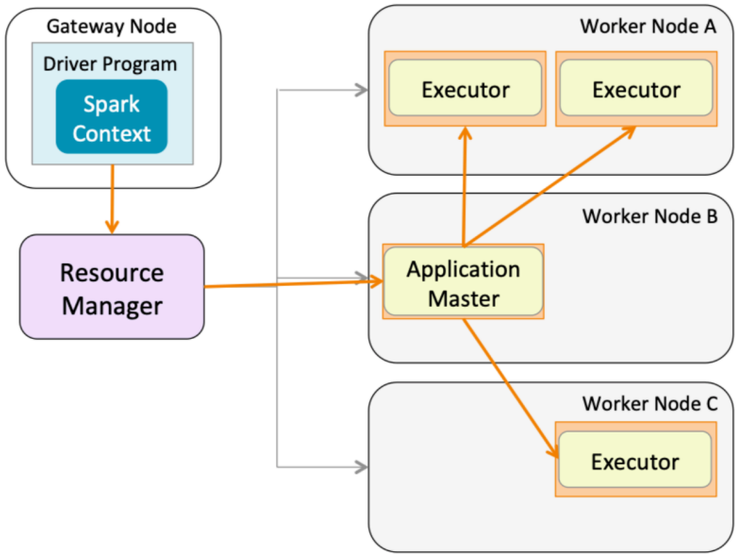
The driver runs on the application master node (in cluster)
good for long time computations

```
spark-submit --master yarn --deploy-mode cluster yourCode.py
```

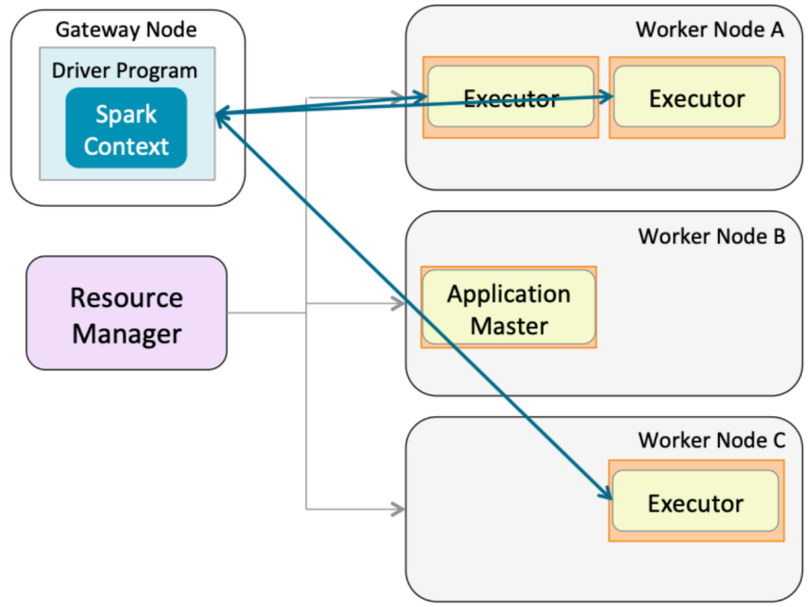
To see more options

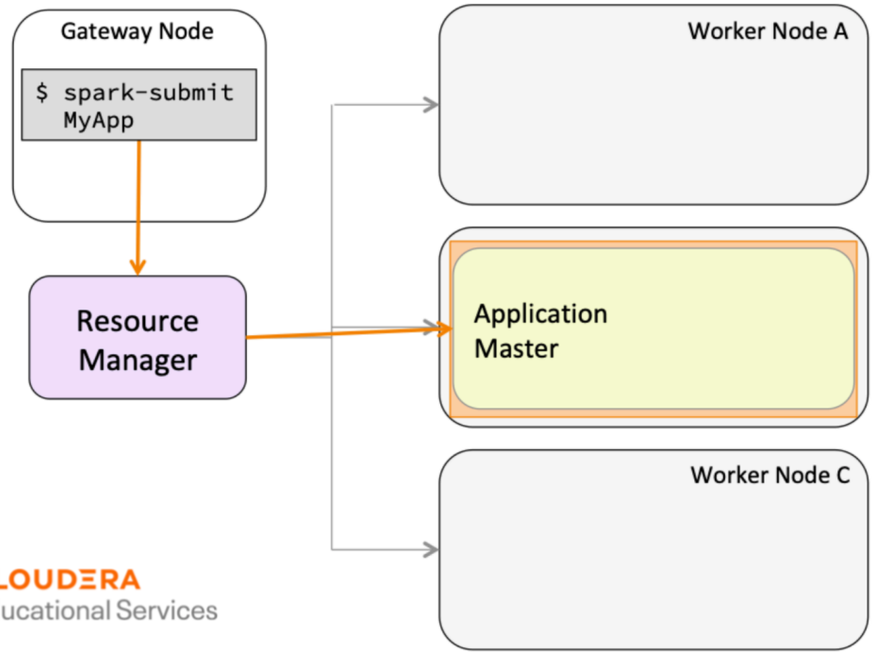
```
spark-submit --help
```

Deployment mode comparison

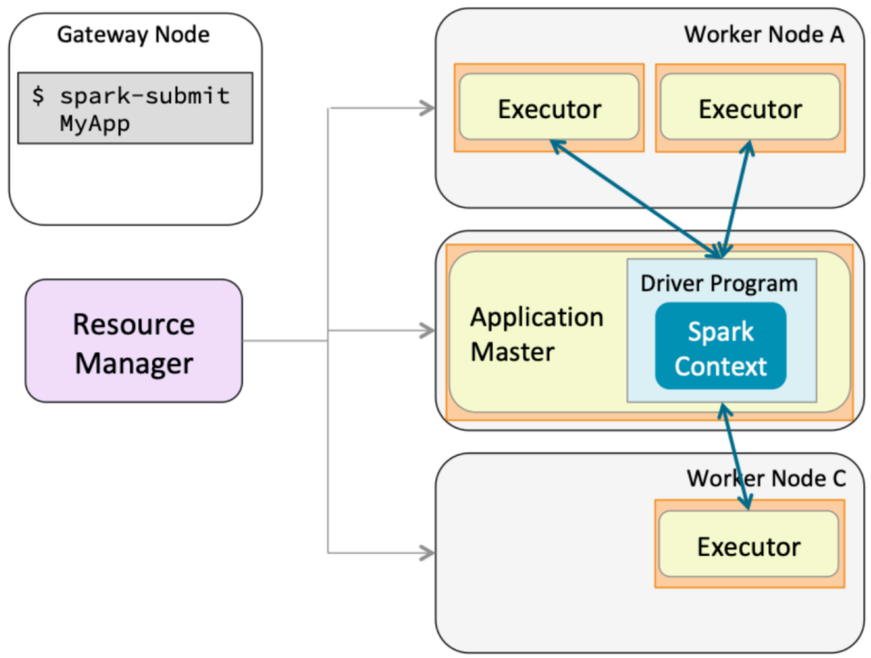


Client mode

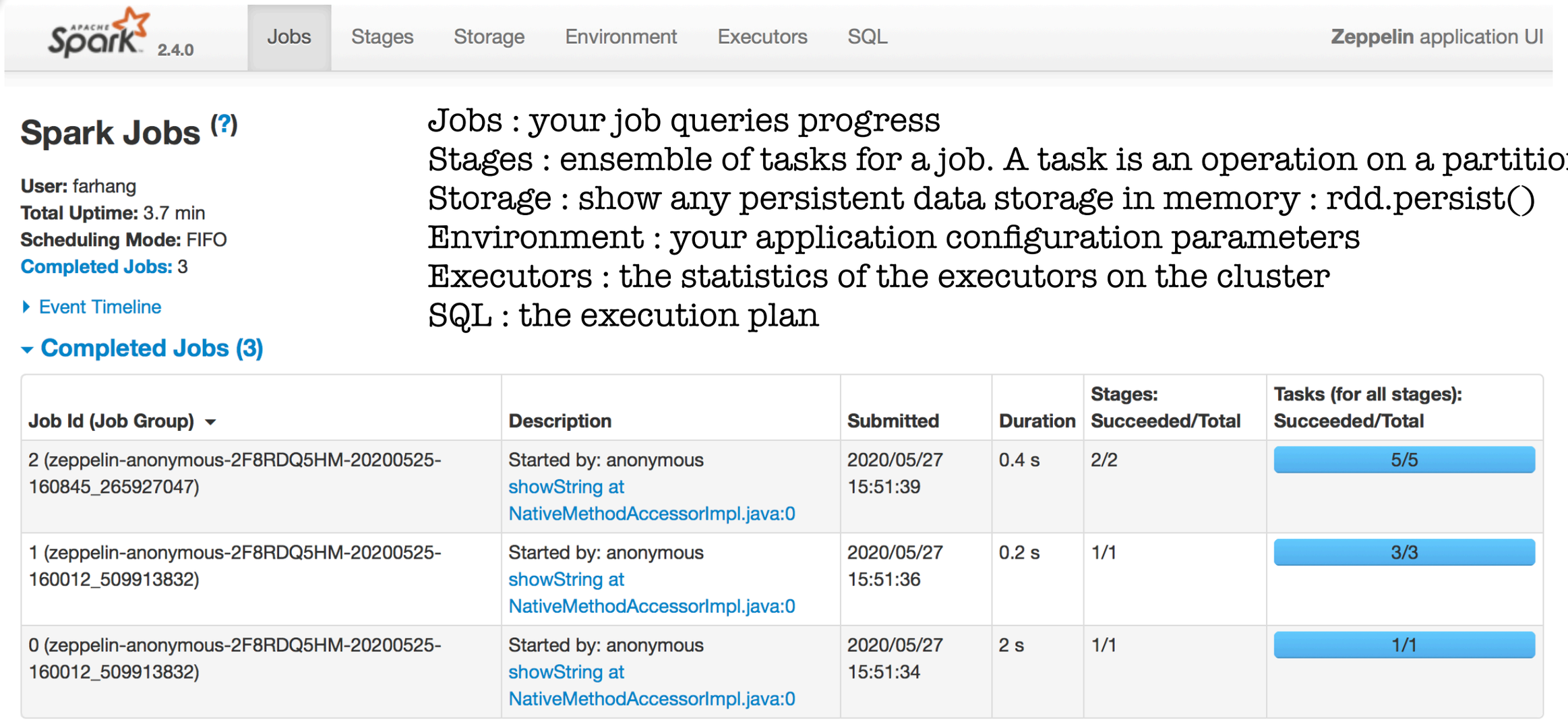




Cluster mode



Spark web UI



Jobs : your job queries progress
Stages : ensemble of tasks for a job. A task is an operation on a partition
Storage : show any persistent data storage in memory : rdd.persist()
Environment : your application configuration parameters
Executors : the statistics of the executors on the cluster
SQL : the execution plan

Spark UI is by default on the gate 4040 of either you localhost or the gateway url For cluster deploy-mode use YARN UI to access Spark UI

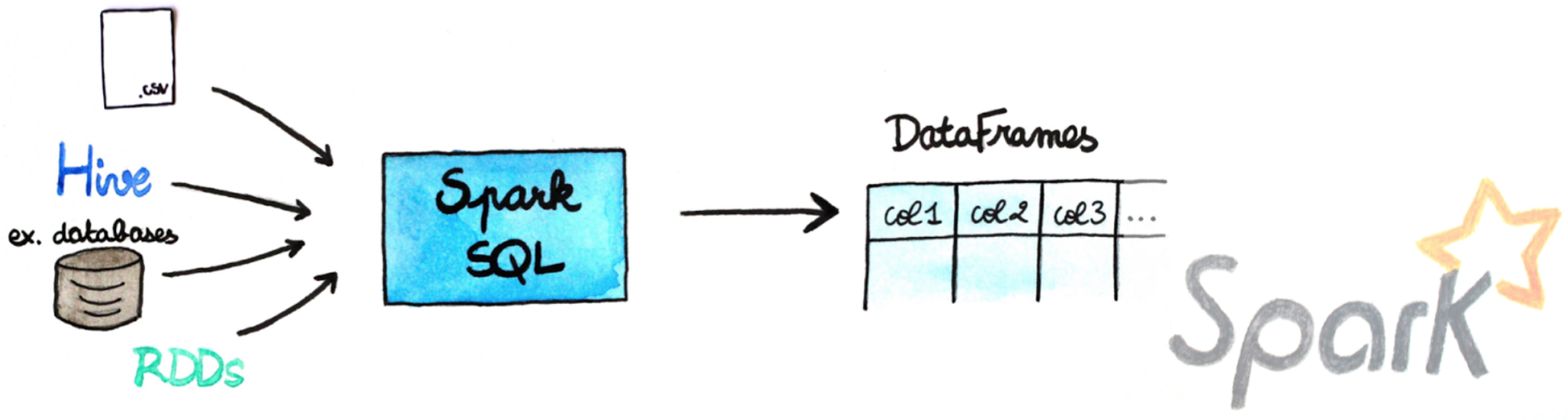
Contents

Spark SQL : Dataframes

- Spark SQL, a component of Apache Spark framework, is used to process the structured data by executing SQL-like queries.

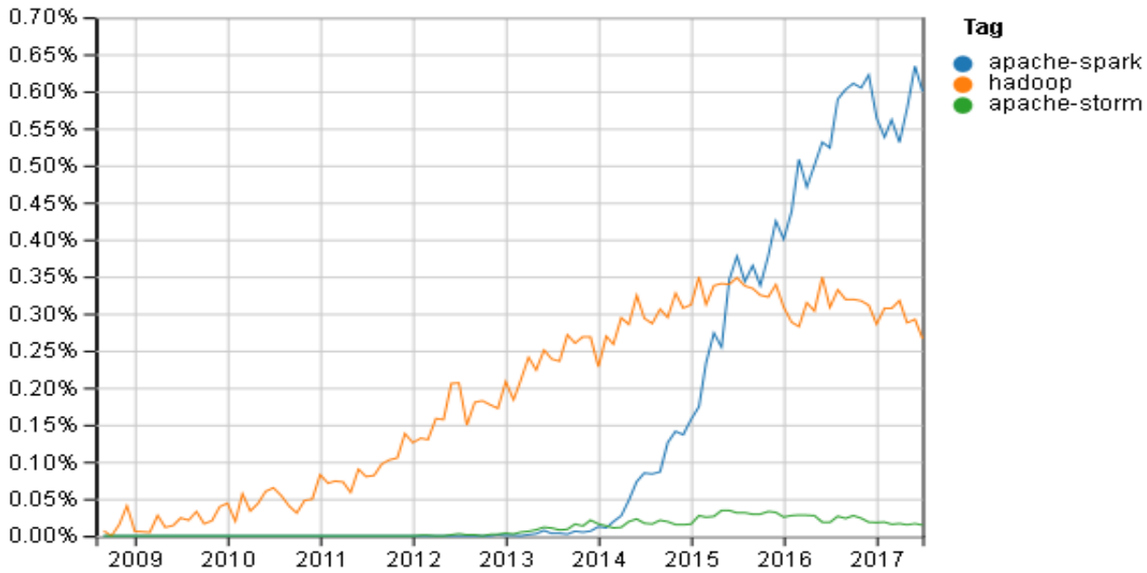
Spark SQL constructs the Dataframes from different data sources such as : Parquet or JSON files, RDD, Hive Table, etc.

The Dataframes are the RDDs represented logically in rows



Spark SQL : Dataframes

- Dataframes appeared after releasing Spark 2.0.0 (2014)



- To create dataframes an instance of SparkSeisson should be created
  - SparkSeisson is included in pyspark.sql library

- The default SparkSeisson instance in REPLs (e.g. notebooks) is “spark”

Contents

Dataframe operations

- Text, csv files
- Binary files like Parquet, ORC, Avro
  - Parquet format compacts a file up to 40% of the initial volume
- Hive tables
- From clouds : Amazon S3, Microsoft ADLS

Dataframe operations

- Read a csv file :
  spark.read.format("csv").option("header", "True").load("/pathTo/data.csv")
  convenient form : spark.read.csv("/pathTo/data.csv", header=True)

- Read a parquet file :
  spark.read.format("parquet").load("/pathTo/data.parquet")
  convenient form : spark.read.parquet("/pathTo/data.parquet")

- You can simultaneously read all the same format files in a folder :
  spark.read.json("/pathTo/*.json")

- Read a hive table :
  spark.read.table("tableName")
  - Hive warehouse path should be known for the Spark
  - Hive support should be activated in Spark config xml file.

Dataframe operations

- Write function for dataframes save them either in a file or in a table
- Different file formats : csv, json, parquet etc.
- Different write mode :
  overwrite, error (default), append
  - Spark is a distributed processing engine and save the DF partitions in a folder.
- Example save a dataframe (df) as csv file :
  df.write.mode(“overwrite”).csv(“/pathTo/myDF.csv”, sep=“\t”, header=True)
- Example save df in as a hive table :
  df.write.mode(“append”).option(“path","/pathTo/myData").saveAsTable("myTable")

Dataframe operations

- Dataframes contain structured data and have schema
  - Schema shows the column names and types
  - Determining the schema is an eagerly (not lazy) process for Spark
  - Schema are immutable. You should transforme a dataframe for a new schema

- Schema are either detected automatically by spark or defined programatically

- Example (automatic schema detection) :

```
df = spark.read.csv("/pathTo/data,csv", header=True, inferSchema=True)
```

- Example (defining a schema) :

```
from pyspark.sql.types import StructType, StructField, StingType, IntegerType

myStruct = [  StructField( "id", IntegerType ), StructField( "name", StringType ) ]
mySchema = StructType( myStruct )

df = spark.read.csv("/pathTo/data,csv", header=True, schema=mySchema)
```

- df.printSchema() shows the schema of the dataframe df.

Dataframe operations

- Column selection by “select” method :
  df.select(“column1”, “column2”, …)

- You can apply any built-in function on the column while selecting :
  from pyspark.sql.functions import col, lower

  df.select(lower(df[“column1”]).alias(“c_1”), “column2”, …)
  or
  df.select(lower(df.column1).alias(“c_1”), “column2”, …)
  or
  df.select(lower(col(“column1”)).alias(“c_1”), “column2”, …)

  - By applying “alias” method you can rename a column

- You have already seen different column access possibilities

Dataframe operations : UDF

```
from pyspark.sql.functions import udf

@udf("string")
def convertMonth(x) :
    return x.strftime('%B')

outputDF = inputDF.withColumn("month",convertMonth("appointmentTime"))
```

- It is possible to apply **U**ser **D**efined **F**unctions to create a new column from one or more existing columns.
- The inputs for UDF should be pyspark.sql.Column type
- The output is a single column with a type in pyspark.sql.types
- You can use any Python packages (NumPy, Pandas, sk-learn, etc) in your UDF.

Dataframe operations : Built-in functions

- Before start writing the UDF make sure that the function you are looking for does not already exist.
- There are many native functions for Spark which are well-optimised for Spark applications.
- You can review them in Apache Spark documentation page at : https://spark.apache.org/docs/latest/api/python/pyspark.sql.html

Dataframe queries

```
data = [ ["Marc", 60], ["Lea", 45],
         ["Lise", 60], ["Sylvain", 45],
         ["Victor", 7], ["Olivier", 45] ]
cols = ["name", "age" ]
d_f = spark.createDataFrame( data, cols )
d_f.show()
```

```
+-------+---+
|   name|age|
+-------+---+
|   Marc| 60|
|    Lea| 45|
|   Lise| 60|
|Sylvain| 45|
| Victor|  7|
|Olivier| 45|
+-------+---+
```

```
d_f.filter( "age < 60" ).orderBy("age", ascending=False).show()
```

```
+-------+---+
|   name|age|
+-------+---+
|    Lea| 45|
|Sylvain| 45|
|Olivier| 45|
| Victor|  7|
+-------+---+
```

```
d_f.groupby( "age" ).count().alias("count").show()
```

```
+---+-----+
|age|count|
+---+-----+
|  7|    1|
| 60|    2|
| 45|    3|
+---+-----+
```

Dataframe queries

```
// Create DateFrame from json file
df = spark.read.json("examples/src/main/resources/
people.json")

// Show the content of the DataFrame
df.show()

// Print the schema in a tree format
df.printSchema()
```

```
// Select only the "name" column
df.select( “name" ).show()

// Select everybody, but increment the age by 1
df.select( df[“name”], df.age+ 1 ).show()

// Select people older than 21
df.filter(df['age'] > 21).show()

// Count people by age
df.groupBy( “age” ).count().show()
```

SparkSeission instance creation

In Spark 1.6 to import SQL functions to Spark an instance of SQLContext should be created from already existing SparkContext :

```
from pyspark.sql import SQLContext
sqlContext = SQLContext(sc)
```

In Spark 2 and later releases the frames operations are part of SparkSession instance which is made independently :

```
from pyspark.sql import SparkSession
spark = SparkSession.builder.appName("appName").getOrCreate()
```

After constructing the SparkSession instance the SparkContext will be automatically created on it and is accessible as spark.SparkContext

Contents

Stage and tasks

- A series of processing applied on the partition is a Spark task.
  -You can visualise the task progress in Spark UI in storage tab.
- A stage is the group of tasks (with same operations) applied on different partitions.
- The ensemble of stages for a query is job.

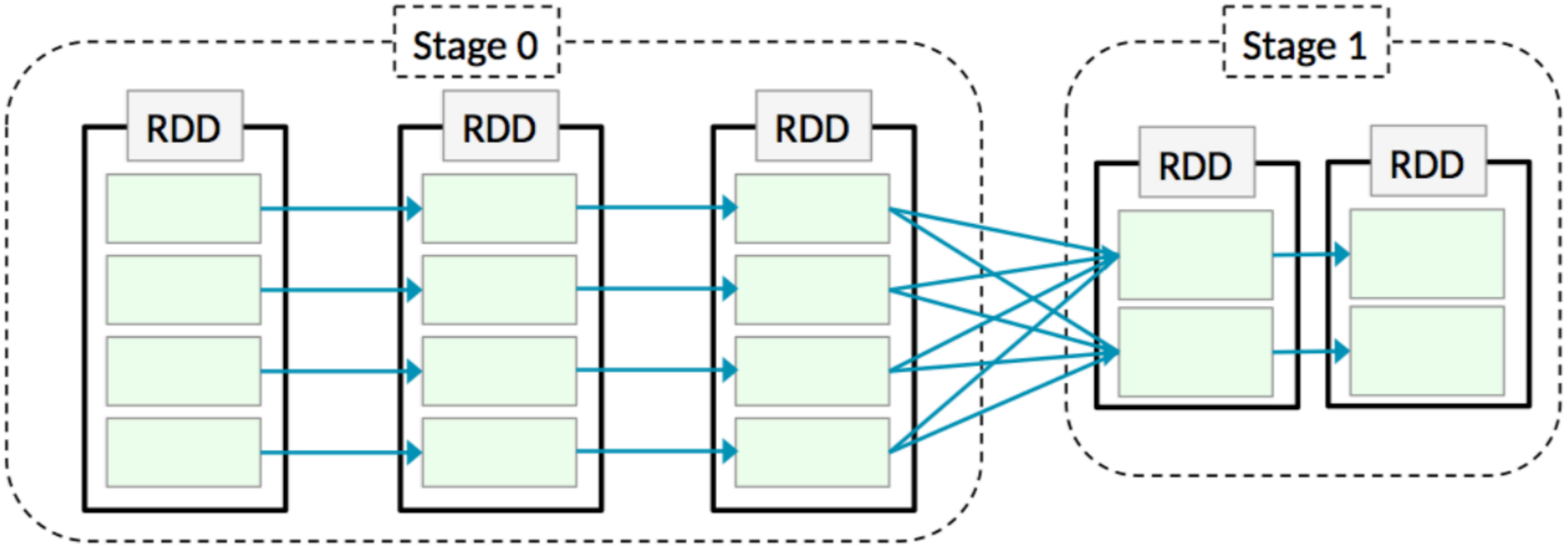
- E.g. this query job contains 2 stages :
  - A stage of 8 tasks (filtering and selection on 8 partitions).
  - One-task stage to “show” data (only display 20 rows of a partition.)

```
outputDF = inputDF.filter( col( "date" ) > lit( "2020-02-29" ) )\
                  .select( "country", "recovered" )
outputDF.show()
```

Stage Id	Pool Name	Description	Submitted	Duration	Tasks: Succeeded/Total
20	5656840774354885068	outputDF = inputDF.filter(col("date") > lit(... showString at NativeMethodAccessorImpl.java:0 +details	2020/06/18 09:59:06	69 ms	1/1
19	5656840774354885068	outputDF = inputDF.filter(col("date") > lit(... showString at NativeMethodAccessorImpl.java:0 +details	2020/06/18 09:58:53	13 s	8/8

Stage and tasks

The operations with shuffling, repartition the data and make a new stage
(e.g. join, groupby, etc)



CLOUDERA
Educational Services

Contents

Data persistence

- Keeping a Dataframe/RDD in memory

- When ?
  - Lot's of resources and shuffling were applied to make a Dataframe/RDD
  - Iterative process where a data sample is referred for several times
  - The reproduction does n't worth for probable failures.

- How ?
  - "persist" method by giving the storage level
  - "cache" method (only in-memory persistence)

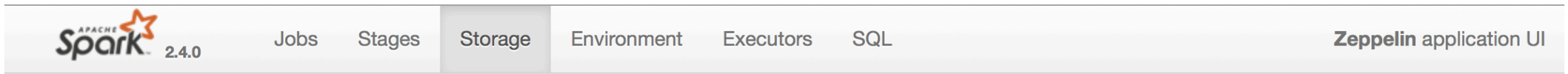
- Storage levels
  - MEMORY-ONLY, DISK-ONLY, MEMORY-AND-DISK
  - Tables are always memory-only persistent
  - Defaults : memory-only for RDDs and memory-and-disk for dataframes

- Python serialises the in-memory data

- Replication
  - MEMORY-ONLY_2, DISK-ONLY_2, MEMORY-AND-DISK_2

Data persistence

```
from pyspark import StorageLevel
df_c = cov.join( cpop, on="country" ).persist( StorageLevel.MEMORY_ONLY )
df_c.show()
```



Storage

RDDs

ID	RDD Name	Storage Level	Cached Partitions	Fraction Cached	Size in Memory	Size on Disk
502	*(2) Project [country#457, date#458, confirmed#459, deaths#460, recovered#461, population#426] +- *(2) BroadcastHashJoin [country#457], [country#435], Inner, BuildRight :- *(2) Project [country#457, date#458, confirmed#459, deaths#460, recovered#461] : +- *(2) Filter isnotnull(country#457) : +- *(2) FileScan csv [country#457,date#458,confirmed#459,deaths#460,recovered#461] Batched: false, Format: CSV, Location: InMemoryFileIndex[file:/Users/farhang/Teachings/PySpark/covid.csv], PartitionFilters: [], PushedFilters: [IsNotNull(country)], ReadSchema: struct<country:string,date:timestamp,confirmed:int,deaths:int,recovered:int> +- BroadcastExchange HashedRelationBroadcastMode(List(input[1, string, true])) +- *(1) Project [population#426, CASE WHEN (country#425 = United States) THEN US ELSE country#425 END AS country#435] +- *(1) Filter isnotnull(CASE WHEN (country#425 = United States) THEN US ELSE country#425 END) +- *(1) FileScan csv [country#425,population#426] Batc...	Memory Serialized 1x Replicated	1	100%	4.4 MB	0.0 B

Contents

Catalyst optimiser

Catalyst optimiser contains a library that consists of set of rules to handle the relational query processing.

```
from pyspark.sql.functions import lit, col

outputDF = inputDF.groupby( "country", "date" ).count()\
                  .filter( col( "date" ) > lit( "2020-02-29" ) )\
                  .orderBy( "count", ascending=False )

outputDF.explain()
```

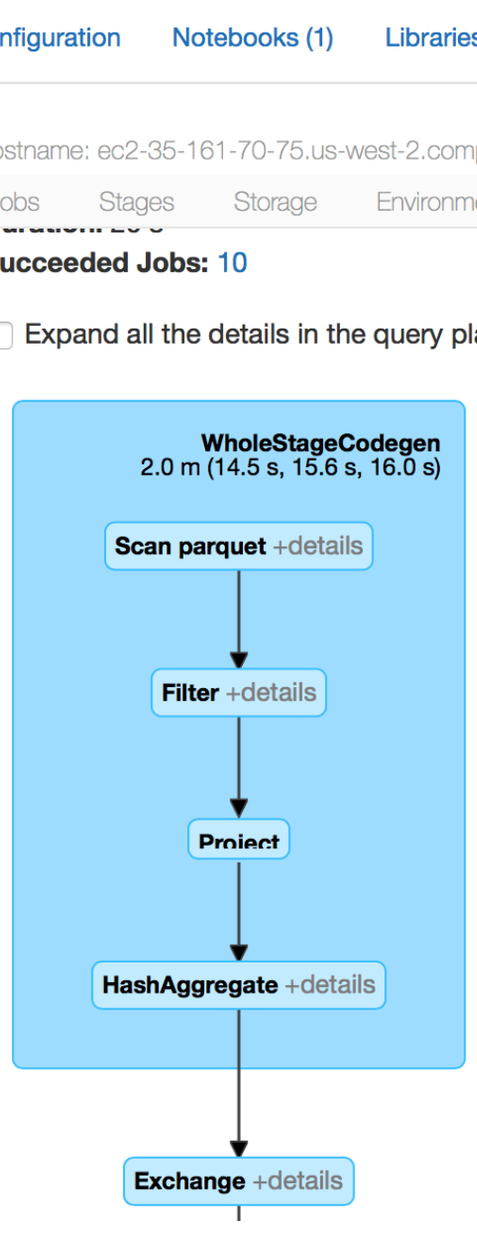
```
== Physical Plan ==
Sort [count#286L DESC NULLS LAST], true, 0
+- Exchange rangepartitioning(count#286L DESC NULLS LAST, 200), [id=#628]
   +- *(2) HashAggregate(keys=[country#220, date#221], functions=[finalmerge_count(merge count#294L) AS count(1)#285L])
      +- Exchange hashpartitioning(country#220, date#221, 200), [id=#624]
         +- *(1) HashAggregate(keys=[country#220, date#221], functions=[partial_count(1) AS count#294L])
            +- *(1) Project [country#220, date#221]
               +- *(1) Filter (isnotnull(date#221) && (cast(date#221 as string) > 2020-02-29))
                  +- *(1) FileScan parquet [country#220,date#221] Batched: true, DataFilters: [isnotnull(date#221), (cast(date#221 as string) > 202
0-02-29)], Format: Parquet, Location: InMemoryFileIndex[dbfs:/FileStore/tables/covid.parquet], PartitionFilters: [], PushedFilters: [IsNotNull(date
)], ReadSchema: struct<country:string,date:timestamp>
```

Catalyst optimiser

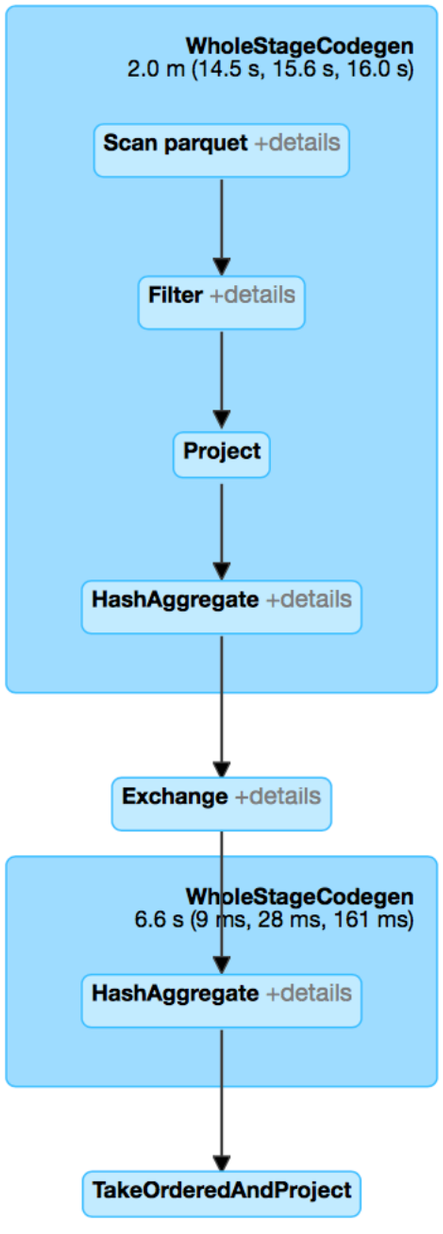
```
outputDF.explain( True )
```

```
== Parsed Logical Plan ==
'Sort ['count DESC NULLS LAST], true
+- Filter (cast(date#221 as string) > 2020-02-29)
   +- Aggregate [country#220, date#221], [country#220, date#221, count(1) AS count#320L]
      +- Relation[country#220,date#221,confirmed#222L,deaths#223L,recovered#224L] parquet

== Analyzed Logical Plan ==
country: string, date: timestamp, count: bigint
Sort [count#320L DESC NULLS LAST], true
+- Filter (cast(date#221 as string) > 2020-02-29)
   +- Aggregate [country#220, date#221], [country#220, date#221, count(1) AS count#320L]
      +- Relation[country#220,date#221,confirmed#222L,deaths#223L,recovered#224L] parquet

== Optimized Logical Plan ==
Sort [count#320L DESC NULLS LAST], true
+- Aggregate [country#220, date#221], [country#220, date#221, count(1) AS count#320L]
   +- Project [country#220, date#221]
      +- Filter (isnotnull(date#221) && (cast(date#221 as string) > 2020-02-29))
         +- Relation[country#220,date#221,confirmed#222L,deaths#223L,recovered#224L] parquet

== Physical Plan ==
```

Catalyst optimiser



In Spark UI SQL tab
you can see the optimised
execution plan both
graphically and in text format.



Contents

Local installation

- Java 8 should be pre-installed

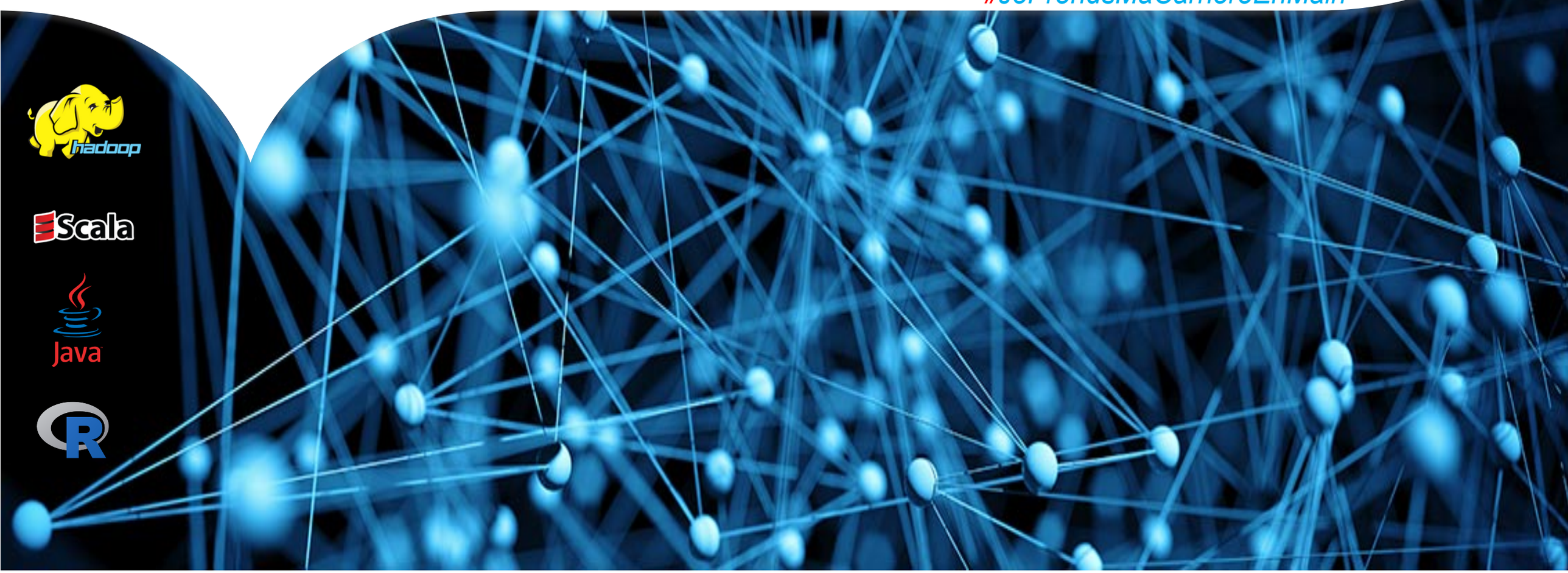
- pyspark from the shell
  - Download Apache Spark, unzip in your desired path
  - In bashrc or bash_profile set the following environmental parameters :
    export SPARK_MAJOR_VERSION=2
    export SPARK_HOME= <path to the download Apache Spark folder>
    export PYSPARK_PYTHON= < path to your Python repository>
  - In your command-line type : pyspark

- Appache Zeppelin
  - Apache Spark should be installed and environmental parameters be set
  - Download Apache Zeppelin unzip in your desired path
  - Go to <Zeppelin directory>/bin and run : zeppelin-daemon.sh start
  - Run Zeppeln on localhost:8080
  - For Zeppelin 0.8 you may need to download Apache Hadoop to set HADOOP_CONF_DIR in <Zeppelin directory>/bin/zeppelin-env.sh

- Jupyter
  - conda install pyspark
  - In bashrc/bash_profile set the SPARK_HOME to your downloaded spark version e.g. :
    SPARK_HOME=/opt/anaconda3/pkgs/pyspark-2.4.5-py_0/site-packages/pyspark/
  - Create your SparkSession instance in Jupyter notebook before start programming

Thank you

#JePrendsMaCarrièreEnMain



FORMATION-BIGDATA.COM

www.formation-bigdata.com