SPARK

→Apache Spark is a lightning-fast cluster computing designed for fast computation.

→It was built on top of Hadoop MapReduce and it extends the MapReduce model to efficiently use more types of computations which includes Interactive Queries and Stream Processing.

→Industries are using Hadoop extensively to analyze their data sets. The reason is that Hadoop framework is based on a simple programming model (MapReduce) and it enables a computing solution that is scalable, flexible, fault-tolerant and cost effective.

→Here, the main concern is to maintain speed in processing large datasets in terms of waiting time between queries and waiting time to run the program.

→Spark was introduced by Apache Software Foundation for speeding up the Hadoop computational computing software Process.

→Spark is not a modified version of Hadoop

→Spark uses Hadoop in two ways – one is **storage** and second is **processing**.

→Since Spark has its own cluster management computation, it uses Hadoop for storage purpose only.

→The main feature of Spark is its **in-memory cluster computing** that increases the processing speed of an application.

→Spark is designed to cover a wide range of workloads such as batch applications, iterative algorithms, interactive queries and streaming.

→Apart from supporting all these workload in a respective system, it reduces the management burden of maintaining separate tools.

Evolution of Apache Spark

→Spark is one of Hadoop's sub project developed in 2009 in UC Berkeley's AMPLab by **Matei Zaharia.**

→It was Open Sourced in 2010.
It was donated to Apache software foundation in 2013, and now Apache Spark has become a top level Apache project from Feb-2014.

Stable release: v2.3.0 / February 28, 2018;
Developer(s): Apache Software Foundation,
UC Berkeley AMPLab, Databricks
Operating system: Microsoft Windows, macOS, Linux
License: Apache License 2.0
Initial release: May 30, 2014;
Written in: Scala, Java, Python, R

Features of Apache Spark

Apache Spark has following features:

1) Speed:

→Spark helps to run an application in Hadoop cluster, up to 100 times faster in memory, and 10 times faster when running on disk.

→This is possible by reducing number of read/write operations to disk.

→It stores the intermediate processing data in memory.

2) Supports multiple languages

→Spark provides built-in APIs in Java, Scala, Python and R.

→Therefore, we can write applications in different languages.

→Spark comes up with 80 high-level operators for interactive Querying.

3) Advanced Analytics:

→Spark not only supports ‘Map’ and ‘reduce’.

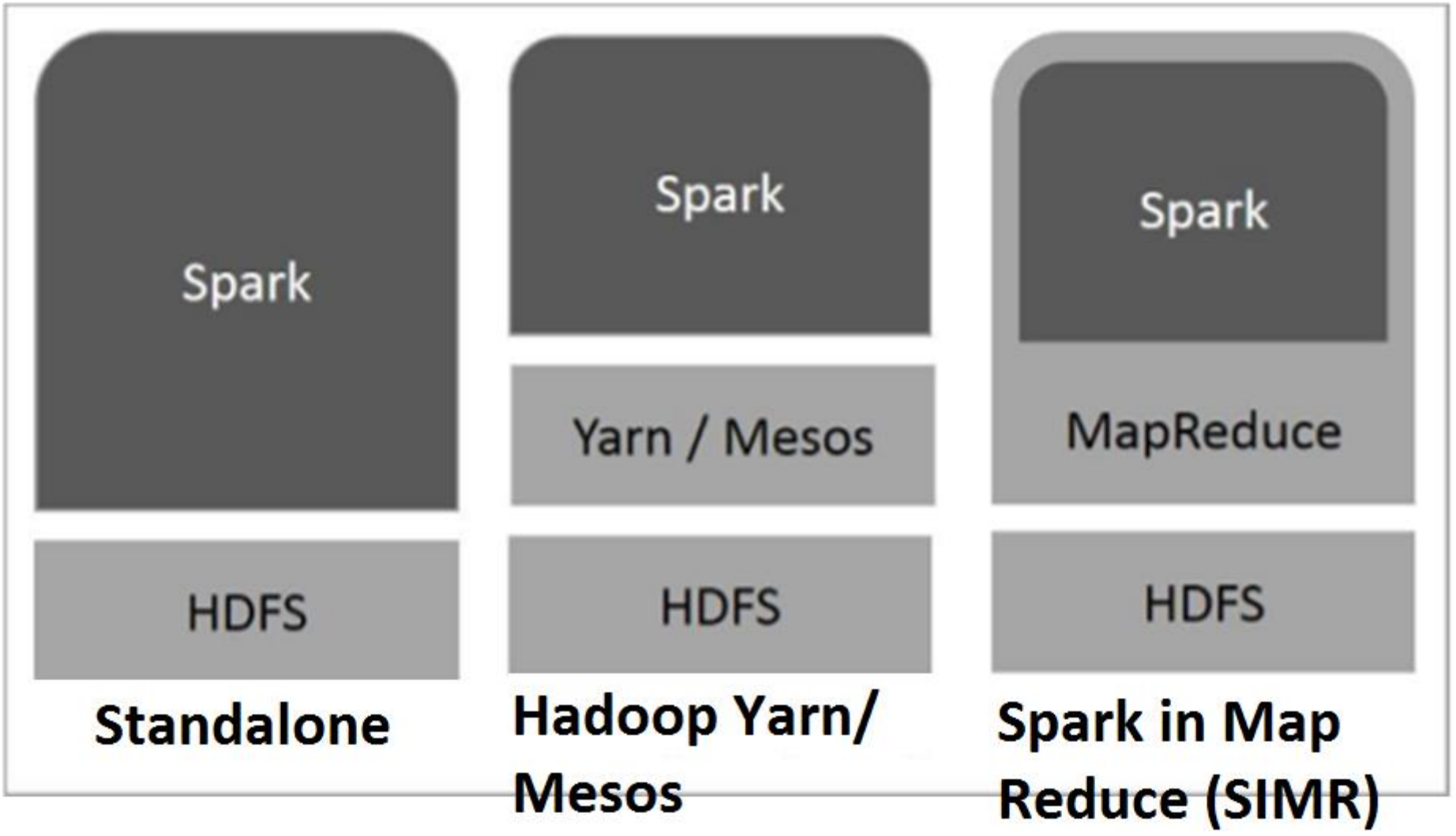
→It also supports SQL queries, Streaming data, Machine learning (ML), and Graph algorithms.

What is a Graph?

→A graph is made up of vertices(or nodes, or points) which are connected by edges(or arcs, or lines)

Spark Built on Hadoop

→The following diagram shows three ways of how Spark can be built with Hadoop components.



There are three ways of Spark deployment:

1) Standalone:

→Spark Standalone deployment means Spark occupies the place on top of HDFS(Hadoop Distributed File System) and space is allocated for HDFS, explicitly.

→Here, Spark and MapReduce will run side by side to cover all spark jobs on cluster.

2) Hadoop Yarn:

→Hadoop Yarn deployment means, simply, spark runs on Yarn Wthout any pre-installation or root access required.

→It helps to integrate Spark into Hadoop ecosystem or Hadoop stack.

→It allows other components to run on top of stack

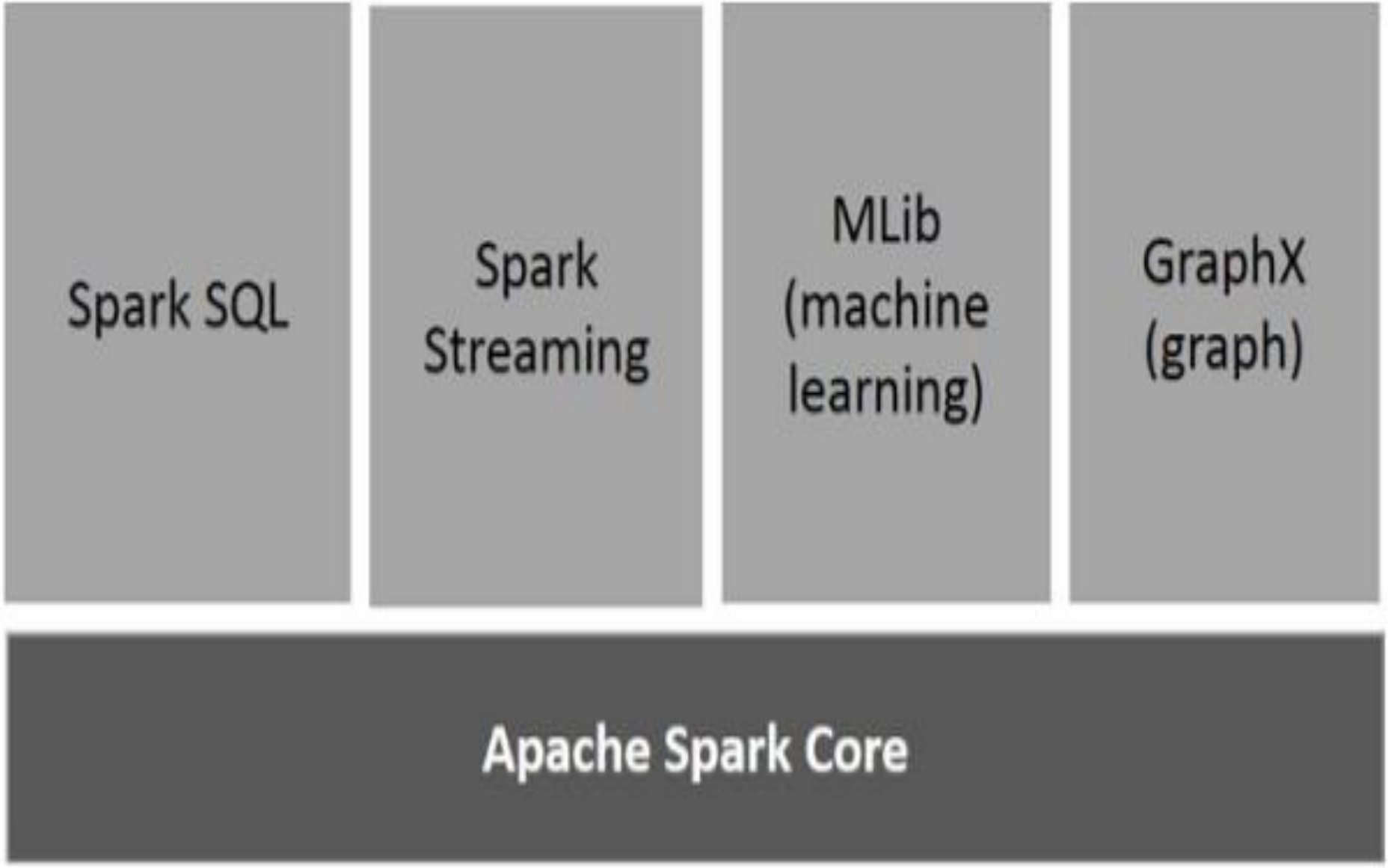
3) Spark in MapReduce (SIMR):

→Spark in MapReduce is used to launch spark job in addition to standalone deployment.

→With SIMR, user can start Spark and uses its shell without any administrative access.

Components of Spark

The following are the different components of Spark:



Apache Spark Core:

→Spark Core is the underlying general execution engine for spark platform that all other functionality is built upon.

→It provides In-Memory computing and referencing datasets in external storage systems.

Spark SQL:

→Spark SQL is a component on top of Spark Core that introduces a new data abstraction called SchemaRDD, which provides support for structured and semi-structured data.

Spark Streaming:

→Streaming data means “The data is flowing continuously”. Streaming data is also called as live data. To process streaming data “Spark Streaming” component will be used. Processing streaming data is called as “Streaming analytics.

→ “Spark Streaming” component uses “Spark Core” component.

MLlib (Machine Learning Library):

What is Machine Learning?

Machine learning is a field of computer science that uses statistical techniques to give computer systems the ability to "learn" with data, without being explicitly programmed.

→MLlib is a distributed machine learning framework above Spark because of the distributed memory-based Spark architecture.

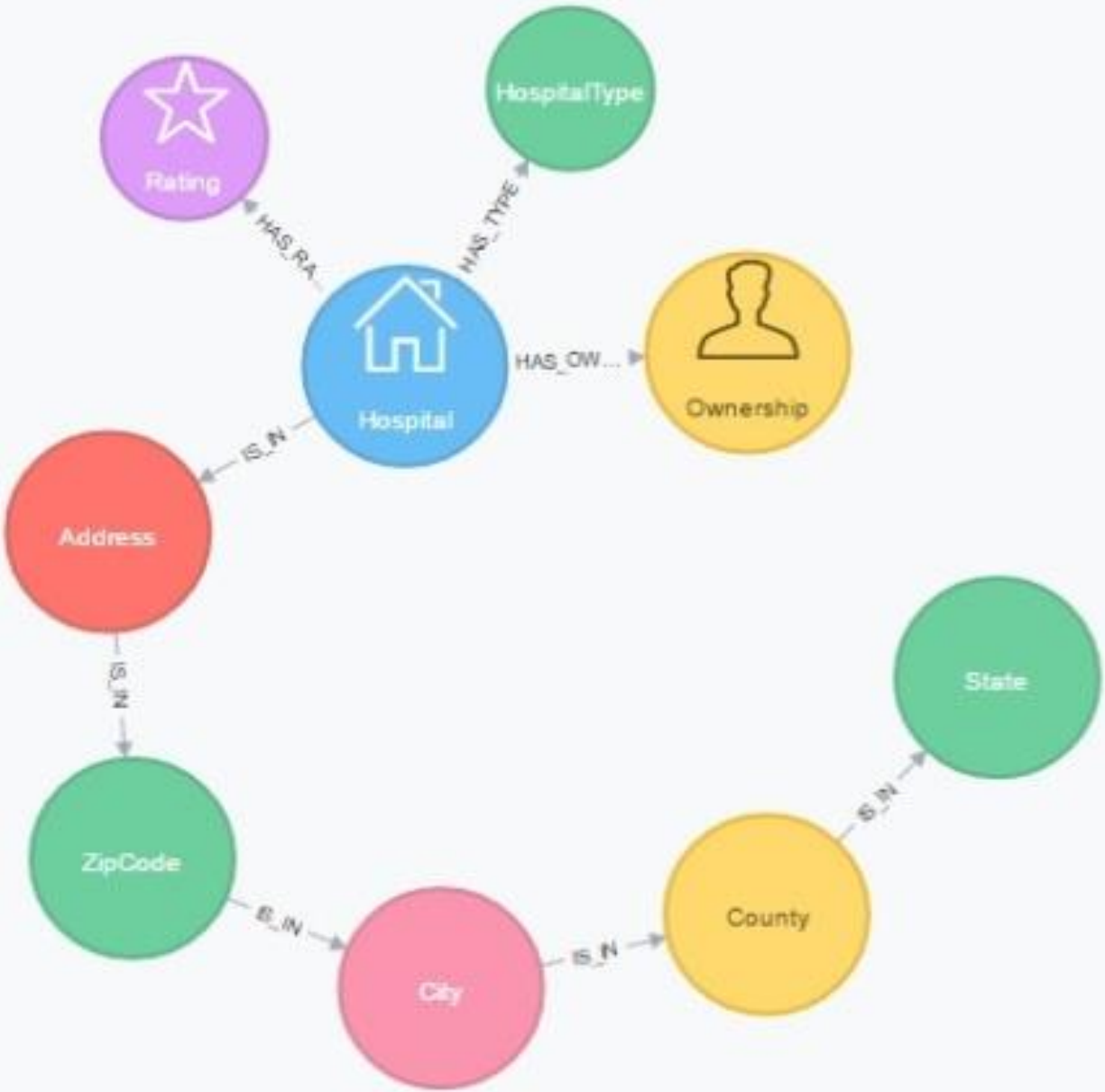
GraphX:

→GraphX is a distributed graph-processing framework on top of Spark.

What is a Graph?

→A graph is made up of vertices(or nodes, or points) which are connected by edges(or arcs, or lines)

→ The following image shows sample graph data.


Rating
HospitalType
HAS_RA...
HAS_TYPE
Hospital
HAS_OW...
Ownership
IS_IN
Address
IS_IN
State
ZipCode
IS_IN
IS_IN
County
IS_IN
City

Resilient Distributed Datasets (RDD)

→Resilient Distributed Datasets (RDD) is a fundamental data structure of Spark.

→It is an immutable distributed collection of objects.

Note: Resilience = Automatic regeneration of data/ Rollback to Original state.

→Each dataset in RDD is divided into logical partitions, which may be computed on different nodes of the cluster.

→RDDs can contain any type of Python, Java, or Scala objects, including user-defined classes.

→RDD is a fault-tolerant collection of elements that can be operated on in parallel.

What is “Fault Tolerance”?

Fault tolerance is the property that enables a system to continue operating properly in the event of the failure of (or one or more **faults** within) some of its components.

→There are two ways to create RDDs:

1) **Parallelizing** an existing collection in your driver program,
2) **Referencing a dataset** in an external storage system, such as a shared file system, HDFS, HBase, or any data source offering a Hadoop Input Format.

→Spark makes use of the concept of RDD to achieve faster and efficient MapReduce operations.

Data Sharing in MapReduce VS Data Sharing using Spark RDD

Data Sharing is Slow in MapReduce:

→Both **Iterative** and **Interactive** applications require faster data sharing across parallel jobs. Data sharing is slow in MapReduce due to **replication, serialization**, and **disk IO**.

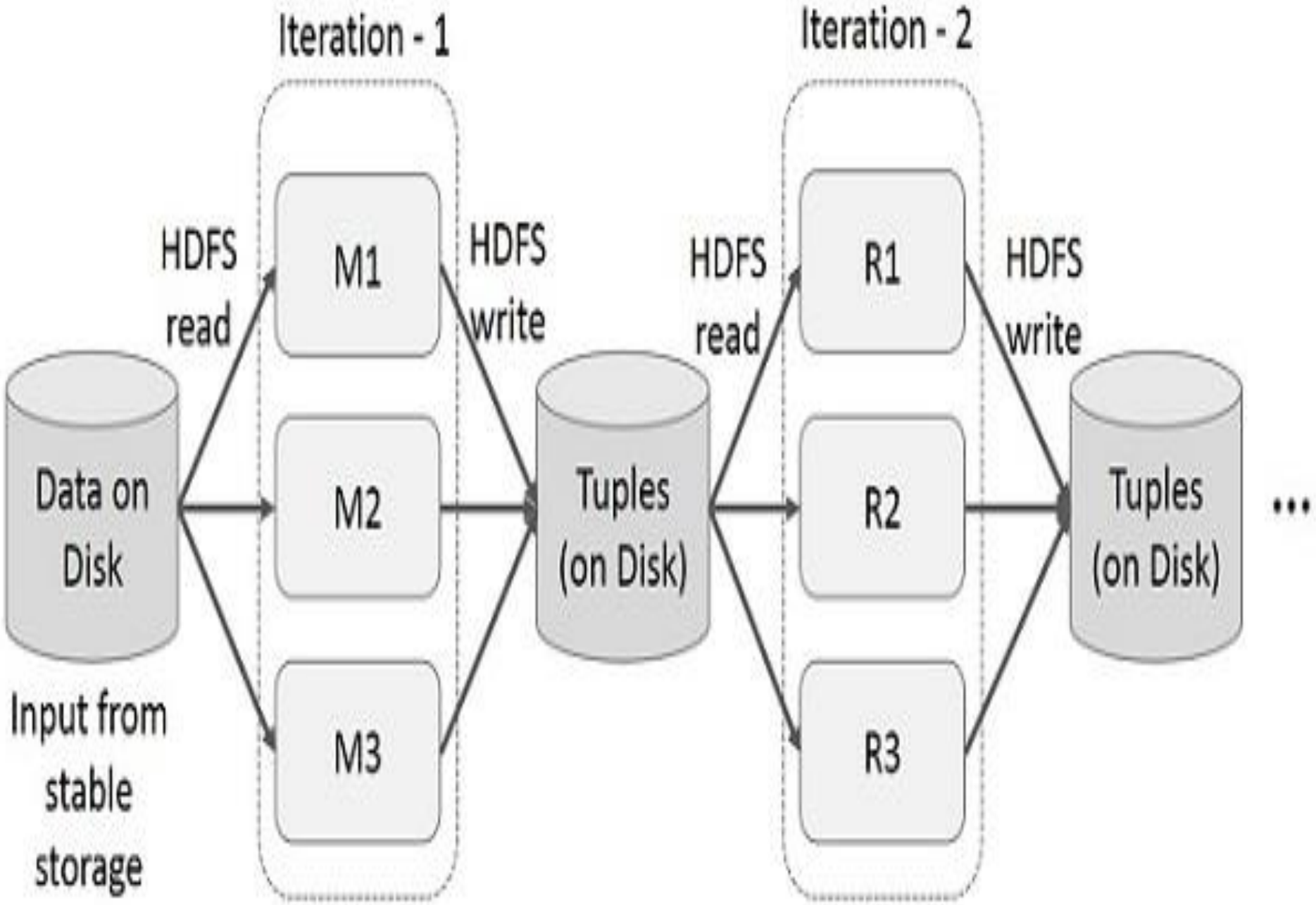
→Regarding storage system, most of the Hadoop applications, they spend more than 90% of the time doing HDFS read-write operations.

Iterative Operations on MapReduce

→Reuse intermediate results across multiple computations in multi-stage applications.

→The following illustration explains how the current framework works, while doing the iterative operations on MapReduce.

→This incurs substantial overheads due to data replication, disk I/O, and serialization, which makes the system slow.

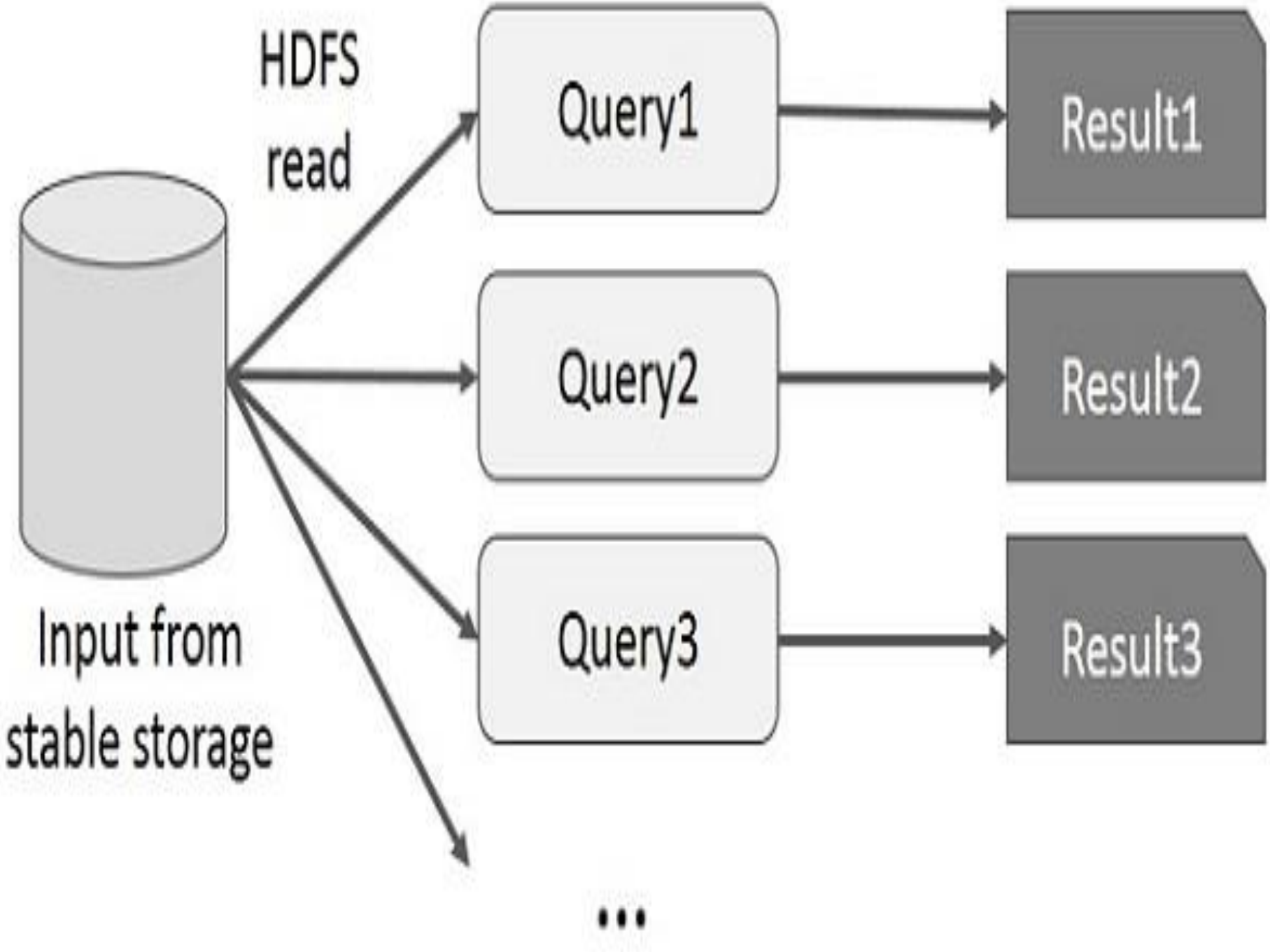

Iteration - 1
Iteration - 2
HDFS read
HDFS write
HDFS read
HDFS write
M1
M2
M3
R1
R2
R3
Data on Disk
Tuples (on Disk)
Tuples (on Disk)
...
Input from stable storage

Interactive Operations on MapReduce

→User runs ad-hoc queries on the same subset of data.

→Each query will do the disk I/O on the stable storage, which can dominate application execution time.

→The following illustration explains how the current framework works while doing the interactive queries on MapReduce.


HDFS
read
Query1
Result1
Query2
Result2
Query3
Result3
Input from
stable storage
...

Interactive Operations on MapReduce

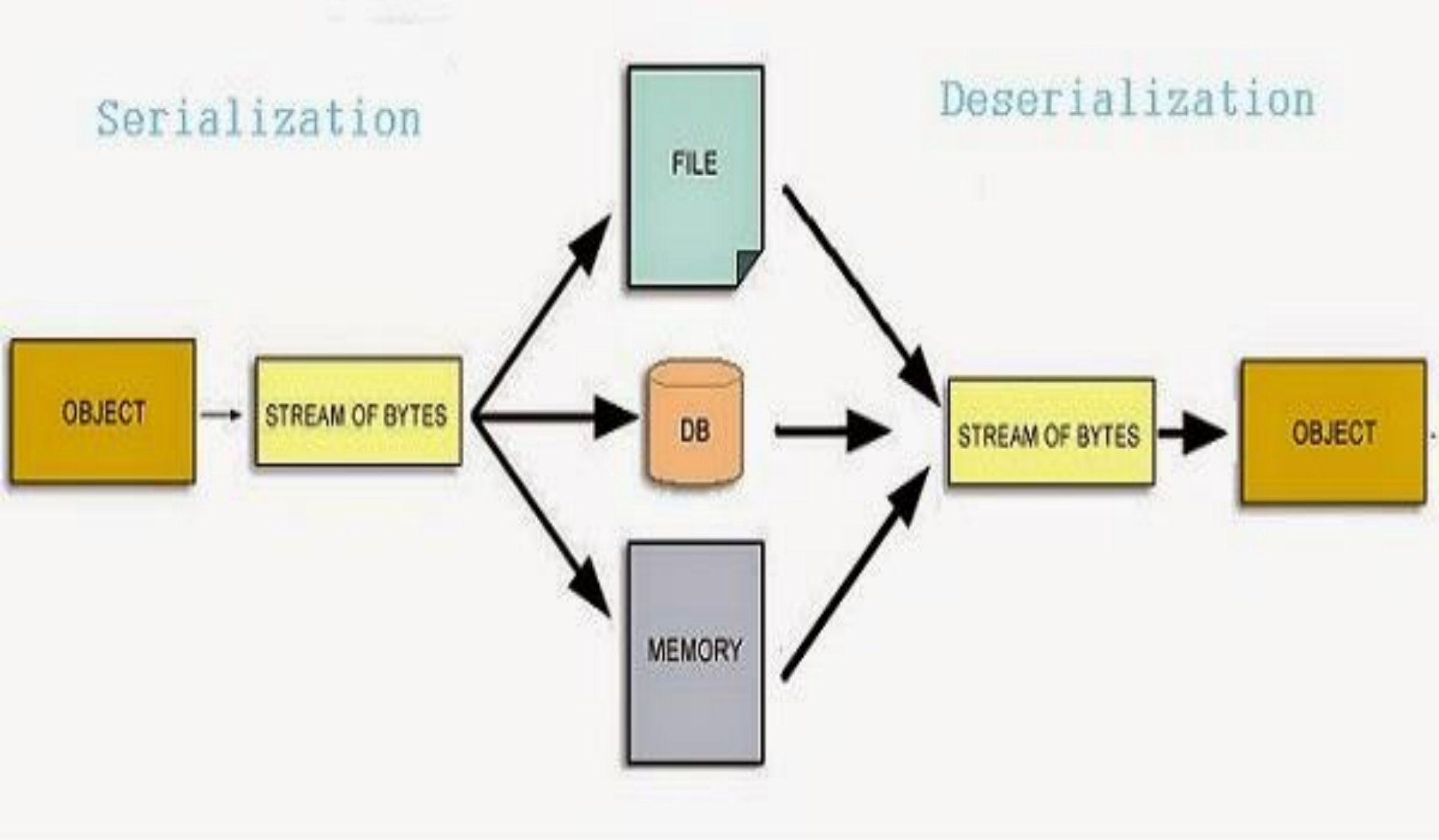
→User runs ad-hoc queries on the same subset of data.

→Each query will do the disk I/O on the stable storage, which can dominate application execution time.

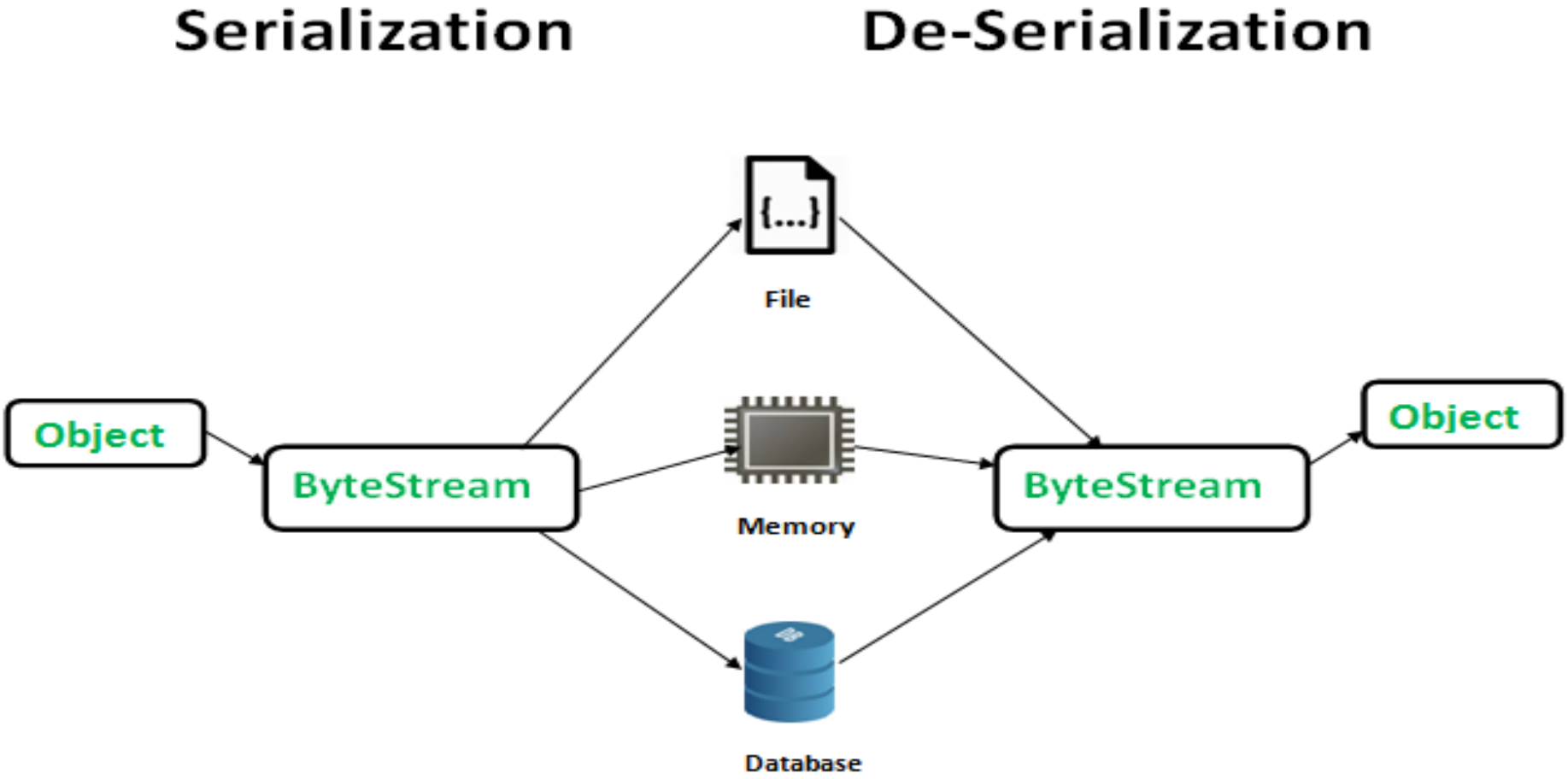
→The following illustration explains how the current framework works while doing the interactive queries on MapReduce.

Data Sharing using Spark RDD:

→Data sharing is slow in MapReduce due to **replication, serialization** and **disk IO**.



OR


Serialization
De-Serialization
{...}
File
Object
ByteStream
Memory
ByteStream
Object
Database

→Most of the Hadoop applications, they spend more than 90% of the time doing HDFS read-write operations.

→To overcome this problem, Apache Spark is invented.

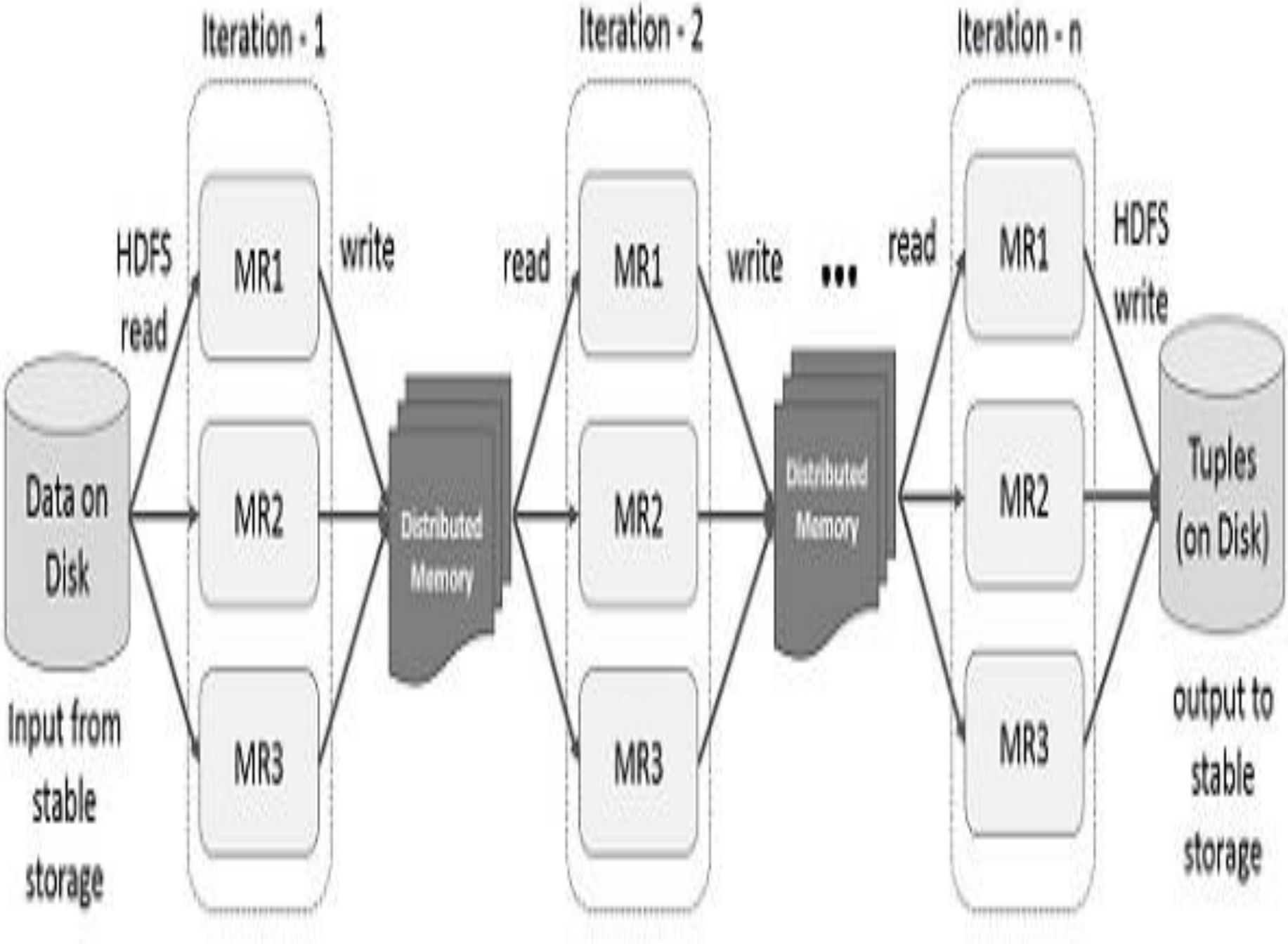
→The key idea of spark is **R**esilient **D**istributed **D**atasets (RDD).

→ RDD supports in-memory processing computation. So, with RDD Data sharing in memory is 10 to 100 times faster than network and Disk.

Iterative Operations on Spark RDD

→The illustration given below shows the iterative operations on Spark RDD. It will store intermediate results in a distributed memory instead of Stable storage (Disk) and make the system faster.

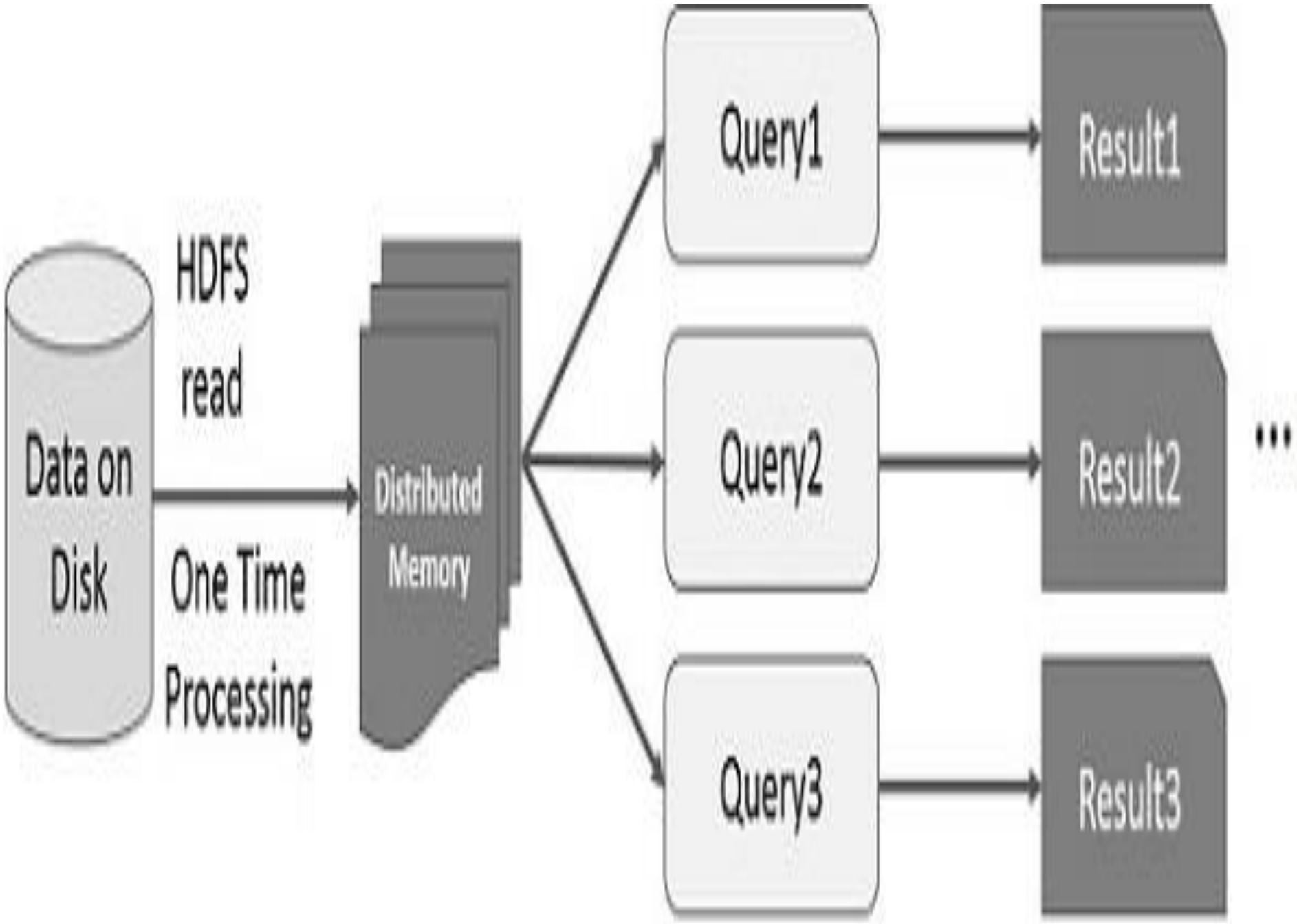
Note – If the Distributed memory (RAM) is not sufficient to store intermediate results (State of the JOB), then it will store those results on the disk.


Iteration - 1
Iteration - 2
Iteration - n
HDFS read
MR1
MR2
MR3
write
read
write
read
HDFS write
Data on Disk
Distributed Memory
Tuples (on Disk)
Input from stable storage
output to stable storage

Interactive Operations on Spark RDD

→This illustration shows interactive operations on Spark RDD.

→If different queries are run on the same set of data repeatedly, this particular data can be kept in memory for better execution times.


Data on Disk
HDFS read
One Time Processing
Distributed Memory
Query1
Query2
Query3
Result1
Result2
Result3
...

Verifying Scala version:

[cloudera@quickstart ~]$ scala -version

Scala code runner version 2.10.4

Verifying the Spark Installation

[cloudera@quickstart ~]$ spark-shell

Spark version 1.6.0

Apache Spark - Core Programming

→Spark Core is the base of the whole project. It provides distributed task dispatching, scheduling, and basic I/O functionalities.

→Spark uses a specialized fundamental data structure known as RDD (Resilient Distributed Datasets) that is a logical collection of data partitioned across machines.

→RDDs can be created in two ways:

1)one is by referencing datasets in external storage systems and

2)second is by applying transformations (e.g. map, filter, reducer, join) on existing RDDs.

Spark Shell

→Spark provides an interactive shell − a powerful tool to analyze data interactively.

→It is available in either Scala or Python language.

(((

Python software comes automatically in cloudera.

Spark + python = pyspark

[cloudera@quickstart ~]$ pyspark

>>> print “hello, python”

hello, python

>>>a=100

```
>>>b=200
>>>a+b
300
>>>
```

Press Ctrl d or type quit() to come out of pyspark.

)))

→Spark's primary abstraction is a distributed collection of items called a Resilient Distributed Dataset (RDD).

→RDDs can be created from Hadoop Input Formats (such as HDFS files) or by transforming other RDDs.

Open Spark Shell

The following command is used to open Spark shell.

$ spark-shell

Output:

Scala>

Create simple RDD

Note: First create “input.txt” as follows:

[cloudera@quickstart ~]$ cat > input.txt

Welcome to SPARK programming

Press ctrl d

Command to create a simple RDD from the text file.

scala> val inputfile = sc.textFile("file:///home/cloudera/input.txt")

Output:

inputfile: org.apache.spark.rdd.RDD[String] =
input.txt MapPartitionsRDD[1] at textFile at <console>:27

To print the number of lines of "inputfile"

scala> inputfile.count

Output:

res8: Long = 1

To Display the text/content of "inputfile"

scala> inputfile.collect.foreach(println)

Output:

Welcome to SPARK programming

→The Spark RDD API introduces **Transformations** and **Actions** to manipulate RDD.

→The Spark RDD API introduces **Transformations** and **Actions** to manipulate RDD.

RDD Transformations & Actions

→Two types of **Apache Spark** RDD operations are:

1)Transformations

2)Actions.

RDD Transformation:

→**Spark Transformation** is a function that produces new RDD from the existing RDDs.

→It takes RDD as input and produces one or more RDD as output.

→Each time it creates new RDD when we apply any transformation.

→Thus, the so input RDDs, cannot be changed since RDD are immutable in nature.

→Applying transformation built an **RDD lineage**, with the entire parent RDDs of the final RDD(s).

→RDD lineage, also known as **RDD operator graph** or **RDD dependency graph.** It is a logical execution plan i.e., it is Directed Acyclic Graph (**DAG**) of the entire parent RDDs of RDD

→**Transformations are lazy** in nature i.e., they get execute when we call an action.

→After the transformation, the resultant RDD is always different from its parent RDD. It can be smaller, bigger.

→**There are two types of transformations:**

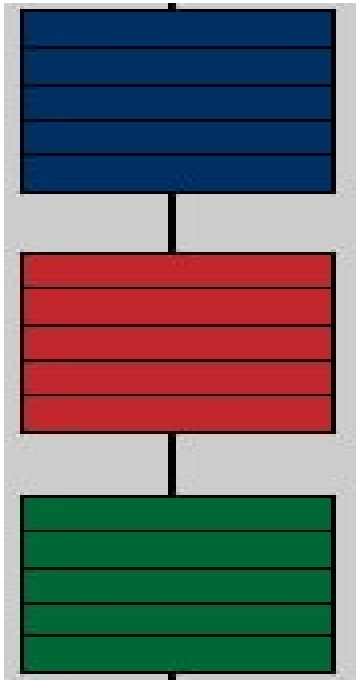
1) Narrow transformation

2) Wide transformation

1)Narrow transformation:

→In *Narrow transformation*, the records in single partition live in the single partition of parent RDD.

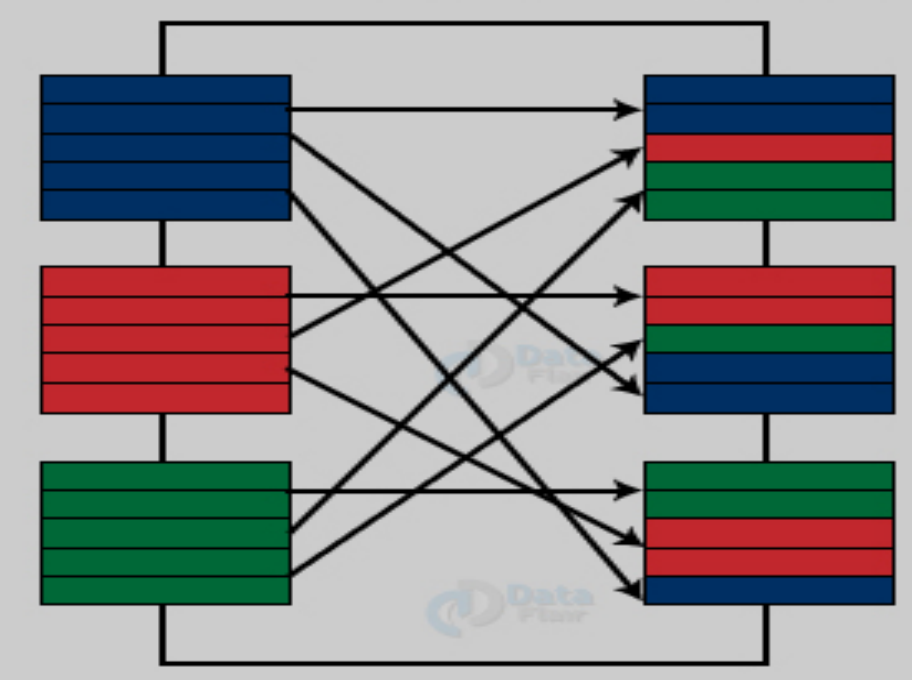
→*Narrow transformations* are the result of *map(), filter().*



2)Wide transformation:

→In wide transformation, the records in the single partition may live in many partitions of parent RDD.

→*Wide transformations* are the result of *groupByKey()* and *reduceByKey()*.



RDD Action

→**Transformations** **create RDDs** from each other, but when we want to work with the actual dataset, at that point action is performed.

→When the action is triggered after the result, new RDD is not formed like transformation.

→Thus, Actions are Spark RDD operations that give non-RDD values.

→The values of action are stored to drivers or to the external storage system.

→"RDD Action" brings laziness of RDD into motion.

→An action is one of the ways of sending data from *Executer* to the *driver.*

→Executors are agents that are responsible for executing a task. While the driver is a JVM process that coordinates workers and execution of the task.

1)RDD Transformation

1.1. map(func)
1.2. flatMap()
1.3. filter(func)
1.4. mapPartitions(func)
1.5. mapPartitionWithIndex()
1.6. union(dataset)
1.7. intersection(other-dataset)
1.8. distinct()
1.9. groupByKey()
1.10. reduceByKey(func, [numTasks])
1.11. sortByKey()
1.12. join()
1.13. coalesce()

2) RDD Action

2.1. count()
2.2. collect()
2.3. take(n)
2.4. top()
2.5. countByValue()
2.6. reduce()
2.7. fold()
2.8. aggregate()
2.9. foreach()

Spark Application is having:

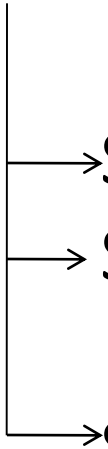
1) Mode
    1.1)Local Mode
    1.2) Batch Mode
2) Cluster
    2.1)YARN
    2.2)Standalone
    2.3)Mesos
3) Name: Spark Shell
4) Memory: 1GB
5) Cores: All available cores like dual core, Quadratic Core

Note: Dual core = 2 cores, Quadratic core = 4 cores, octal core = 8 cores etc...

Structure of a Spark Programming:

→Importing of API/Packages (Spark API) – default

1) Creation of Context(s)



→Spark Core →Spark Context

→ Spark sql → Spark Context, sql context (or) Hive context

→Spark Streaming → Spark Context, Streaming Context

* Spark Session

2) Read the data from external Source(s) and storing them into appropriate data storage objects like RDDs, Dataframes, Datasets ….

3) Performing the following Operations on data storage objects:

- 3.1) Transformations
- 3.2) Actions

4) Write the result to local storage system or external storage system

5) Stop the Spark Context (or simply context). When spark context is stopped then all other contexts also will be stopped.

The various contexts in Spark:

→The contexts are used to start spark application, to create data storage objects and to stop spark application

1)Spark Context:

This is used to start an application, to create RDDs, to stop the application

2)SQL Context:

This is used to create data frames and data sets.

3)Hive Context:

This is used to create data frames and data sets and also to access hive data into spark application

4) Spark Session:

→This is used to start an application, to create RDDs, data frames, and data sets with hive data accessible into spark application and also used to stop the application.

→This spark session context is introduced in spark 2.x onwards.

5)Streaming Context:

→ It is used to start streaming application and to create DStreams (Descretized Streams) and to stop streaming application.

Spark Component	Storage Object
Spark Core	**RDD**
Spark SQL	**Dataframe, Dataset**
Spark Streaming	**DStreams (Descretized Streams)**
Spark Mllib	**Vectors. The vectors are two types: 1) Sparse verctors, 2) Dense Vectors**
Spark GraphX	**Graph objects**

→On these data storage objects, the operations to be Performed are “Transformations and Actions”.

Note: Spark Core = Transformations and Actions

Spark Component	Hadoop component
Spark Core	Map Reduce(MR), Apache Pig
Spark SQL	Hive
Spark Streaming	Flume, Kafka
Spark Mllib	Mahout

→Spark application can read the data from:

1) LFS (Local File System)
2) HDFS (Hadoop Distributed File System)
3) NFS (Network File System)
4) Amazon S3 (s3 = Simple Storage Service)
5) RDBMS:
    - 5.1) Oracle
    - 5.2) MySQL
    - 5.3) Teradata
    - 5.4) Neteza
6) NOSQL Databases:
    - 6.1) Hbase
    - 6.2) Cassandra
    - 6.3) MongoDB

→ Spark Streaming applications read from streaming components like Flume, kafka, kinesis.

→Spark applications can be submitted in interactive mode(Local mode) or batch mode

→Local mode means our system is not participating in cluster.

→ Spark application can be executed in the following ways:

1) spark application in local mode
2) spark application in YARN cluster
3) Spark application in Standalone cluster
4) Spark application with python (or) Pyspark (or) Spark with Python API
5) Spark application in YARN cluster with python API

1)To start spark application in local mode:

[cloudera@quickstart ~]$ spark-shell --master local

(or) spark-shell

→By default Spark Context and hive context (i.e. sqlContext) are created.

→To see Spark Context:

scala> sc

Output:

res0: org.apache.spark.SparkContext = org.apache.spark.SparkContext@62f5765e

→To see sql context:

scala> sqlContext

Output:

res1: org.apache.spark.sql.SQLContext = org.apache.spark.sql.hive.HiveContext@253abda7

→To see spark application in browser:

Open browser and type localhost:4040

→To stop Spark (in Terminal) (or) to come out of **scala>**:

scala>:q

Note: after stopping spark at terminal then observe in browser by pressing F5 then “Unable to connect” message comes.

→For checking purpose, start spark application again and check in the browser by pressing F5 then spark application will be displayed.

2) To Start spark application in YARN cluster:

[cloudera@quickstart ~]$ spark-shell --master yarn

→Here also, By default Spark Context and hive context (i.e. sqlContext) are created.

→ To see spark context:

scala> sc

res0: org.apache.spark.SparkContext = org.apache.spark.SparkContext@219c8ce3

→To see sqlContext:

scala> sqlContext

res0: org.apache.spark.sql.SQLContext = org.apache.spark.sql.hive.HiveContext@6c8be54d

→To see applications in browser:
Open browser and type localhost:8088
→To quit from spark:
scala>:q

YARN cluster:

→YARN cluster is also called as a Hadoop cluster.
→YARN cluster is a master-slave architecture in which "Resource Manager" acts as a "Master process" and "Node Manager" acts as a "Slave process".

3) To connect to Stand alone cluster:

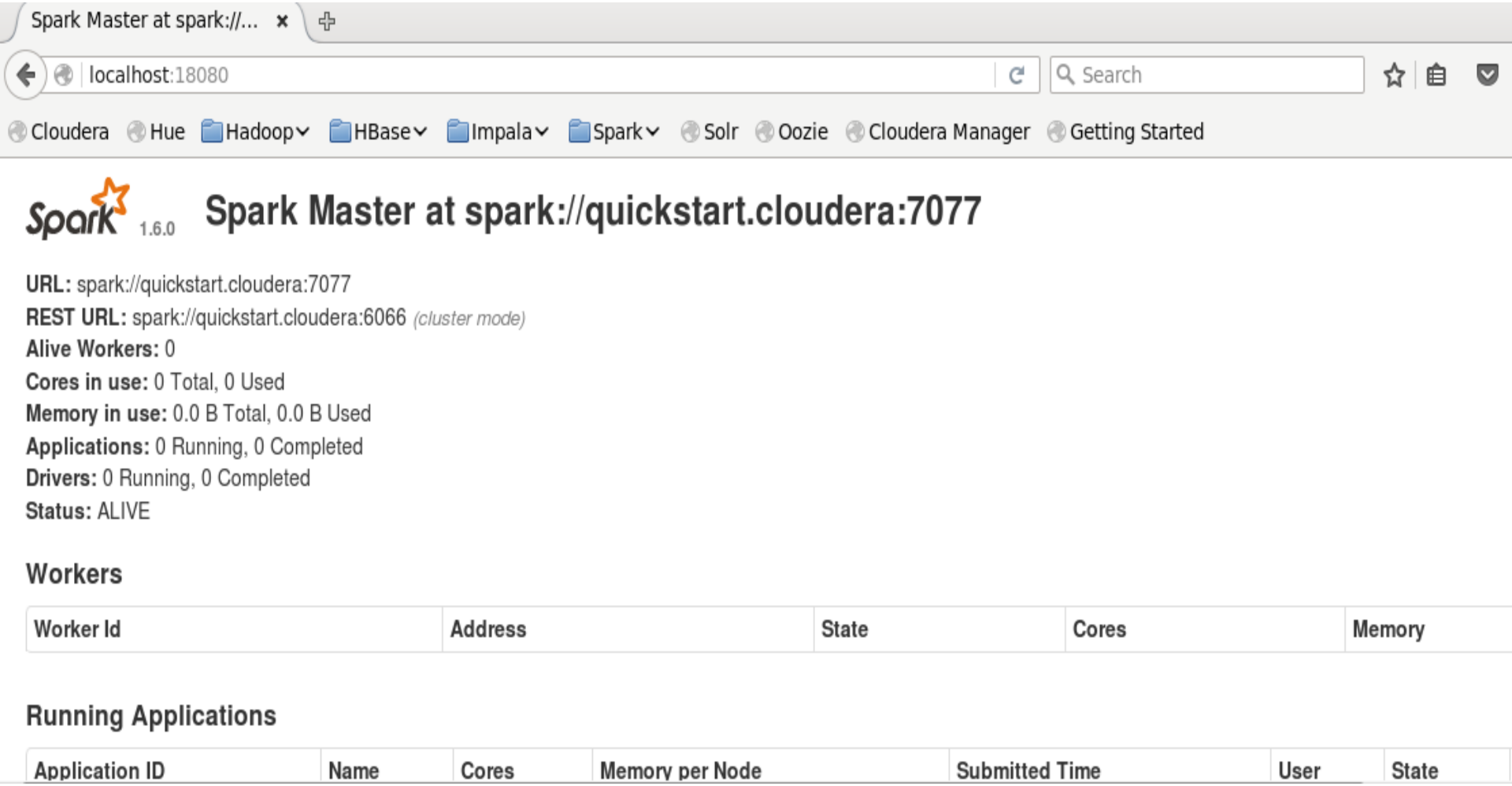
```
[cloudera@quickstart ~]$
[cloudera@quickstart ~]$ cd /usr/lib
[cloudera@quickstart lib]$ cd spark
[cloudera@quickstart spark]$ cd sbin
[cloudera@quickstart sbin]$ sudo ./start-master.sh
```

Output:

```
starting org.apache.spark.deploy.master.Master, logging to /var/log/spark/spark-root-org.apache.spark.deploy.master.Master-1-quickstart.cloudera.out
```

→We can also see spark master application in browser:

Open browser and type localhost:18080


Spark Master at spark://...
localhost:18080
Search
Cloudera
Hue
Hadoop
HBase
Impala
Spark
Solr
Oozie
Cloudera Manager
Getting Started
Spark 1.6.0
Spark Master at spark://quickstart.cloudera:7077
URL: spark://quickstart.cloudera:7077
REST URL: spark://quickstart.cloudera:6066 (cluster mode)
Alive Workers: 0
Cores in use: 0 Total, 0 Used
Memory in use: 0.0 B Total, 0.0 B Used
Applications: 0 Running, 0 Completed
Drivers: 0 Running, 0 Completed
Status: ALIVE
Workers
Worker Id
Address
State
Cores
Memory
Running Applications
Application ID
Name
Cores
Memory per Node
Submitted Time
User
State

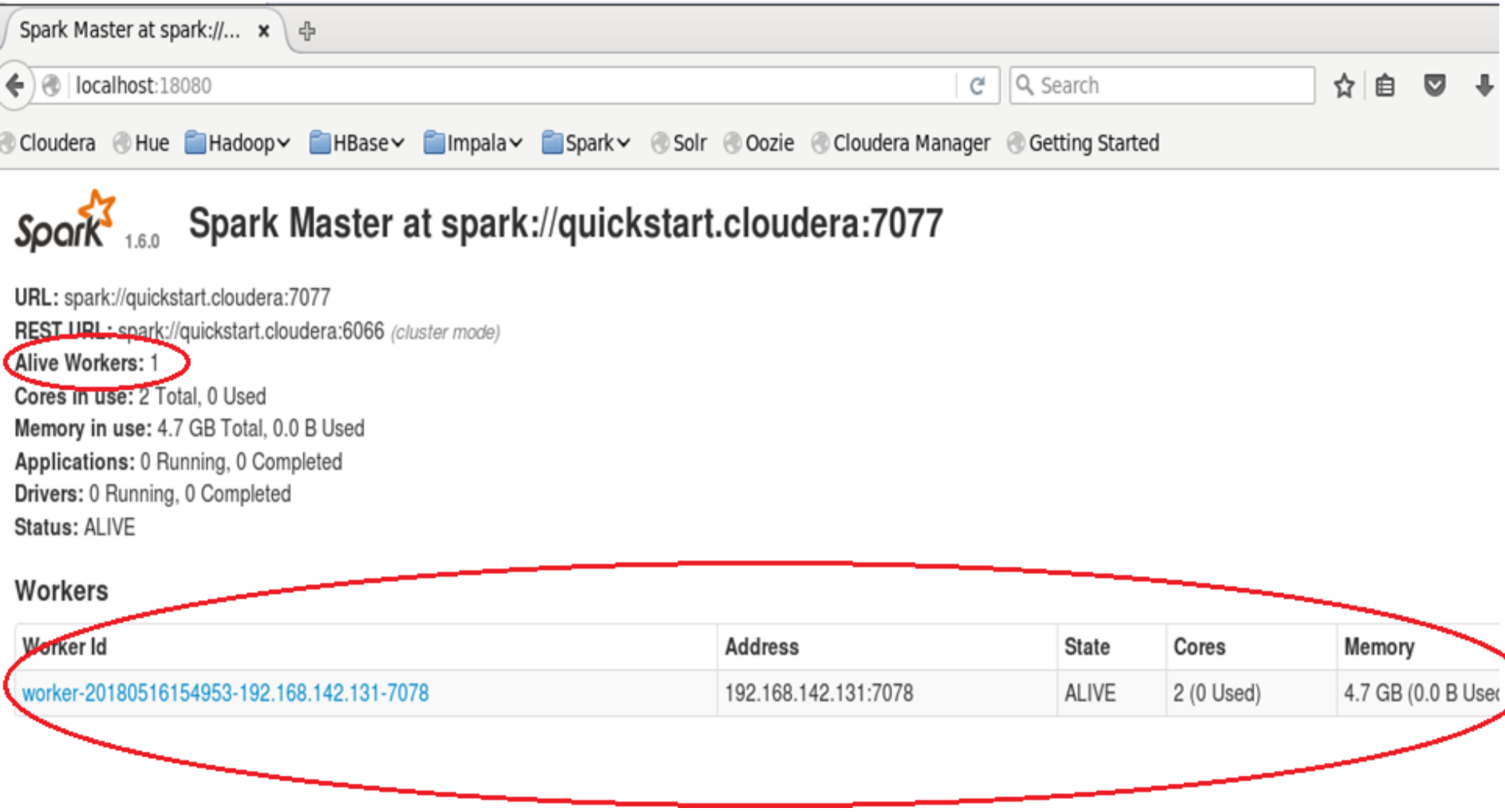
→To start slave process:

[cloudera@quickstart sbin]$ sudo ./start-slave.sh spark://quickstart.cloudera:7077

Output:

starting org.apache.spark.deploy.worker.Worker, logging to /var/log/spark/spark-root-org.apache.spark.deploy.worker.Worker-1-quickstart.cloudera.out

→To see in the browser whether slave process or worker process is started or not by pressing F5.



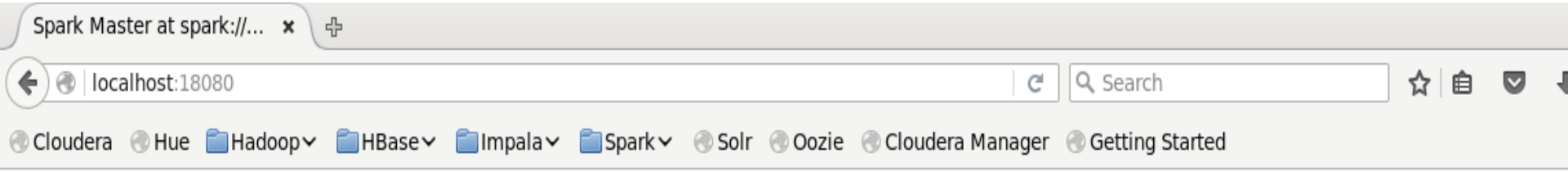
→To stop slave process or worker process:

[cloudera@quickstart sbin]$ sudo ./stop-slave.sh

Output:

stopping org.apache.spark.deploy.worker.Worker

→ To see whether slave process or worker process is stopped or not in the browser. For this press F5.



Spark Master at spark://quickstart.cloudera:7077

URL: spark://quickstart.cloudera:7077
REST URL: spark://quickstart.cloudera:6066 *(cluster mode)*
Alive Workers: 0
Cores in use: 0 Total, 0 Used
Memory in use: 0.0 B Total, 0.0 B Used
Applications: 0 Running, 0 Completed
Drivers: 0 Running, 0 Completed
Status: ALIVE

Workers

Worker Id	Address	State	Cores	Memory
worker-20180516154953-192.168.142.131-7078	192.168.142.131:7078	DEAD	2 (0 Used)	4.7 GB (0.0 B Us

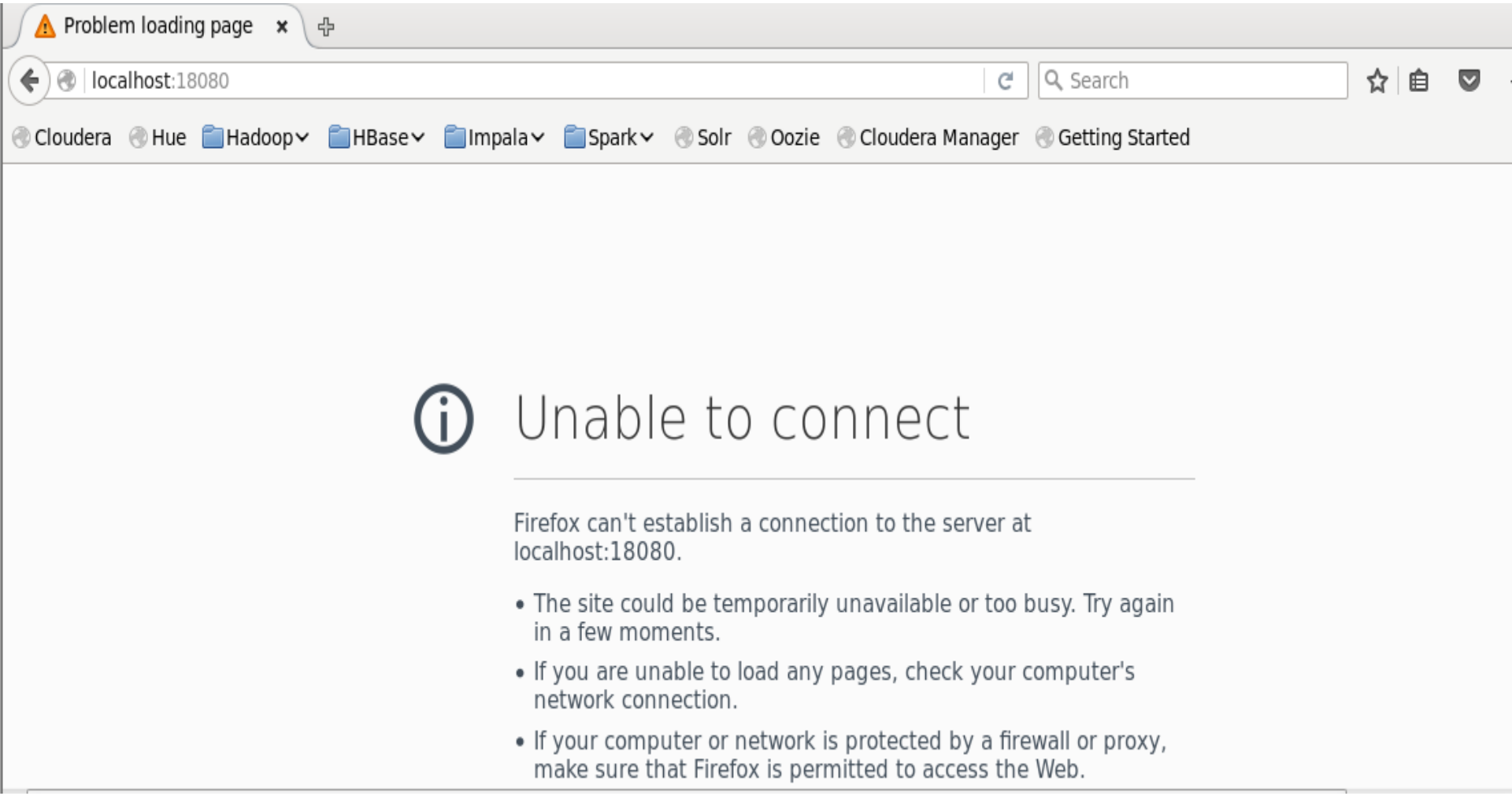
→To stop master process:

[cloudera@quickstart sbin]$ sudo ./stop-master.sh

Output:

stopping org.apache.spark.deploy.master.Master

→To see whether master process is stopped or not in the browser. For this press F5.


Problem loading page
localhost:18080
Search
Cloudera
Hue
Hadoop
HBase
Impala
Spark
Solr
Oozie
Cloudera Manager
Getting Started
Unable to connect
Firefox can't establish a connection to the server at localhost:18080.
• The site could be temporarily unavailable or too busy. Try again in a few moments.
• If you are unable to load any pages, check your computer's network connection.
• If your computer or network is protected by a firewall or proxy, make sure that Firefox is permitted to access the Web.

4) Pyspark (i.e. Spark with Python API)

→We can develop Spark application with Python language also.

To start "pyspark":

[cloudera@quickstart ~]$ pyspark

Note: The meaning is "python API in interactive mode (or) Spark application is launched in python API i.e. local mode

→ Spark Context is available

>>> sc

Output:

<pyspark.context.SparkContext object at 0x1ae9c50>

→ SQL context is available

>>> sqlContext

Output: <pyspark.sql.context.HiveContext object at 0x1b28290>

Sample python coding:

```
>>>a=100
>>>b=200
>>>a+b
300
```

To see Spark application in browser:

Type **localhost:4040**

Note: Press Ctrl d or quit() to come out of pyspark.

5) To Start spark application in YARN cluster with python API:

[cloudera@quickstart ~]$ pyspark --master yarn

>>> sc

<pyspark.context.SparkContext object at 0x154bc50>

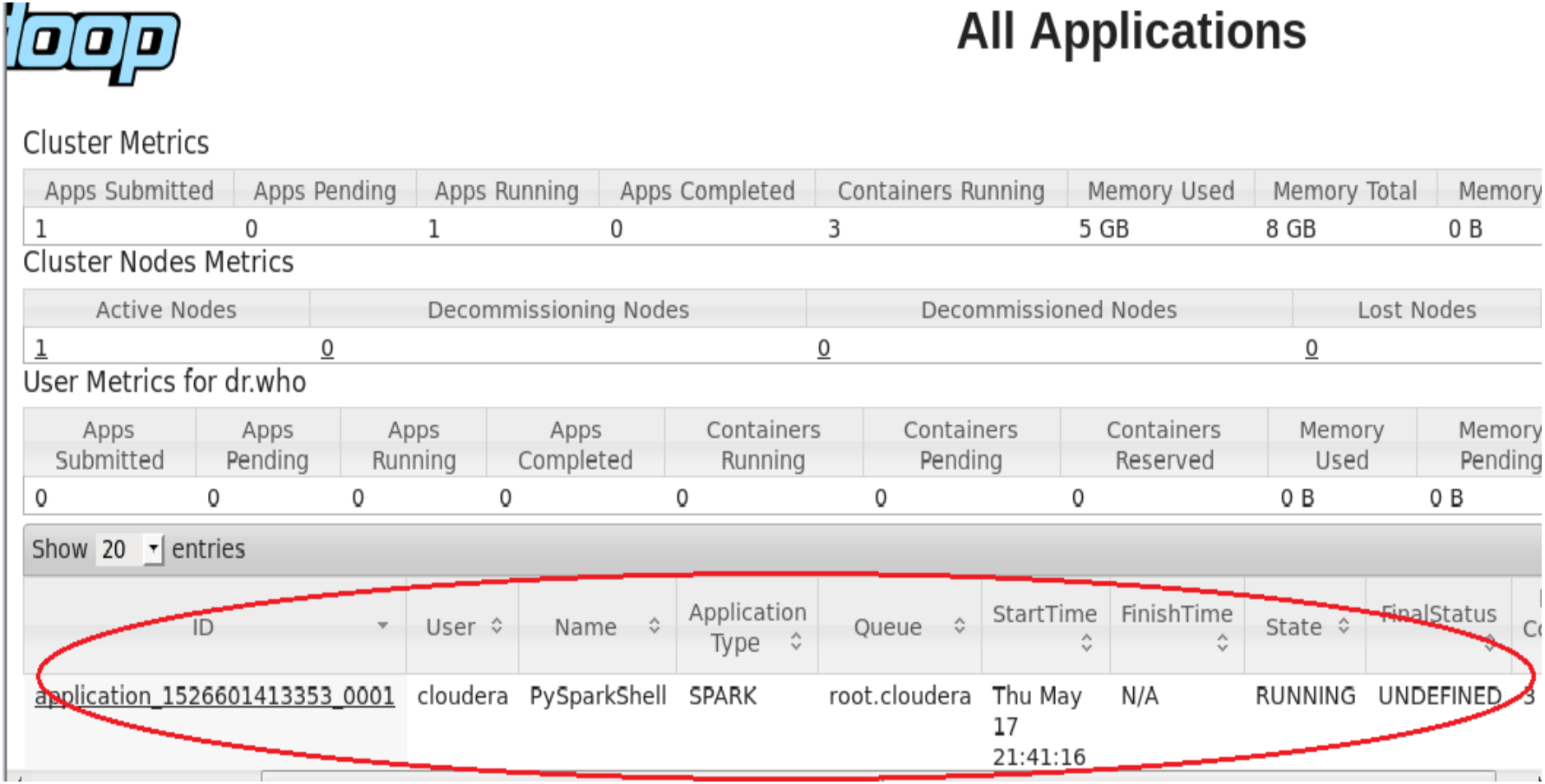
>>> sqlContext

<pyspark.sql.context.HiveContext object at 0x15ca350>

>>>

→We can also see in browser

Type **localhost:8088**



Note: Press Ctrl d or quit() at command prompt to come out of pyspark.

Word Count Program:

→Spark application performs word count operation.

→Double click on **Cloudera's Home** directory on Cloudera Desktop

→Right click and select **Create Document** option and select **Empty file** option and type **file1** and press enter key.

→Double click on **file1** and enter the following text:

Spark is powerful
Spark is super
spark is efficient
Spark is fast

→Select **save** option from file and select **Quit** option from File menu to come out of the editor.

→ Open Terminal:

[cloudera@quickstart ~]$ spark-shell

scala>val r1 = sc.textFile("file:///home/cloudera/file1")

Note: We need not type the path of file1, Simply goto Clodera's Home directory, right click on file1 select copy Option and paste after sc.textFile("

Output:

r1: org.apache.spark.rdd.RDD[String] = file:///home/cloudera/inputfile MapPartitionsRDD[1] at textFile at <console>:27

Verifying "r1":

scala> r1.collect

Output:

res0: Array[String] = Array(Spark is powerful, Spark is super, spark is efficient, Spark is fast)

scala> val r2 = r1.map(x=>x.split(" "))

Output:

r2: org.apache.spark.rdd.RDD[Array[String]] =
MapPartitionsRDD[2] at map at <console>:29

Verifying "r2":

scala> r2.collect

Output:

res1: Array[Array[String]] = Array(Array(Spark, is,
powerful), Array(Spark, is, super), Array(spark, is, efficient),
Array(Spark, is, fast))

scala> val r3 = r2.flatMap(x=>x)
r3: org.apache.spark.rdd.RDD[String] =
MapPartitionsRDD[3] at flatMap at <console>:31

Verifying "r3":
scala> r3.collect
res2: Array[String] = Array(Spark, is, powerful, Spark, is, super, spark, is, efficient, Spark, is, fast)

scala> val r4 = r3.map(x=>(x,1))

output

r4: org.apache.spark.rdd.RDD[(String, Int)] =
MapPartitionsRDD[4] at map at <console>:33

Verifying "r4":

scala> r4.collect

Output:

res3: Array[(String, Int)] = Array((Spark,1), (is,1), (powerful,1), (Spark,1), (is,1), (super,1), (spark,1), (is,1), (efficient,1), (Spark,1), (is,1), (fast,1))

scala> val r5 = r4.reduceByKey((x,y) => (x+y))

Output:

r5: org.apache.spark.rdd.RDD[(String, Int)] =
ShuffledRDD[5] at reduceByKey at <console>:35

Verifying "r5":

scala> r5.collect

Output:

res0: Array[(String, Int)] = Array((is,4), (fast,1),
(powerful,1), (Spark,3), (spark,1), (efficient,1), (super,1))

(or)

scala> r5.collect.foreach(println)

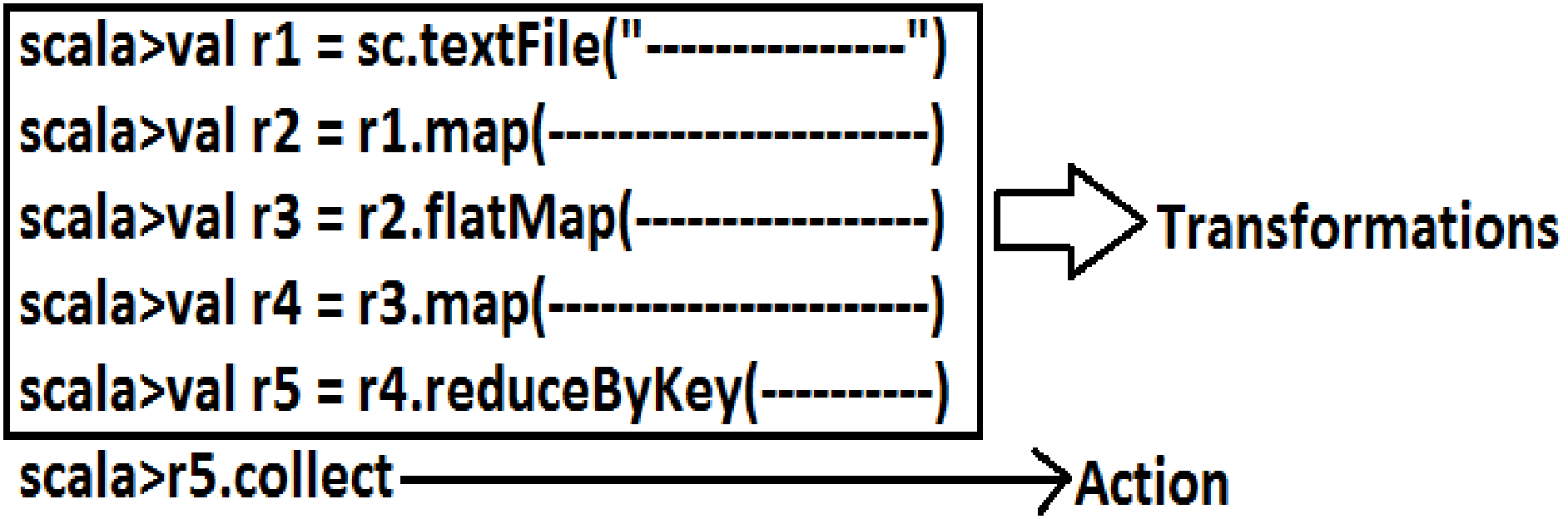
(is,4)
(fast,1)
(powerful,1)
(Spark,3)
(spark,1)
(efficient,1)
(super,1)

So overall steps for performing "Word Count" program:

```
scala> val r1 = sc.textFile("file:///home/cloudera/file1")
scala> val r2 = r1.map(x=>x.split(" "))
scala> val r3 = r2.flatMap(x=>x)
scala> val r4 = r3.map(x=>(x,1))
scala> val r5 = r4.reduceByKey((x,y) => (x+y))
scala> r5.collect.foreach(println)
(or)
scala> r5.collect.foreach(println(_))
```

Note: For Sorting purpose we can write the following:

```
scala> r5.sortBy(x => x._1).collect.foreach(println)
```



Lazy Evaluation:

→Transformations are evaluated Lazily. i.e Transformations are not executed/implemented/operated immediately.

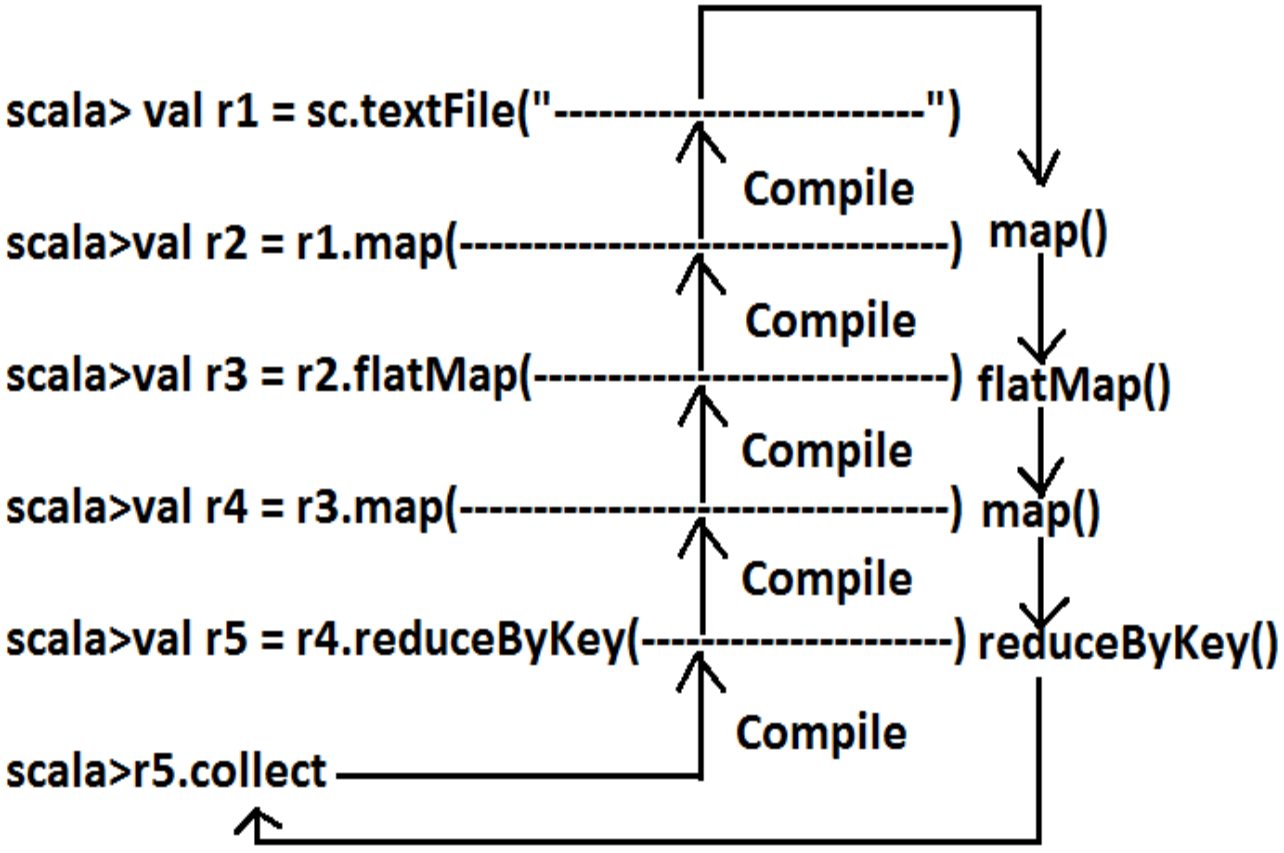
→ When actions are used then only transformations are executed.

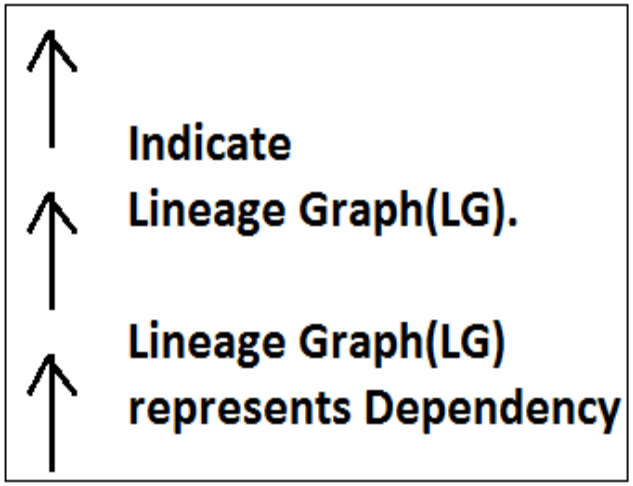
→When spark engine encounters a transformation, instead of executing it, it compiles and remembers the operation i.e. to be performed.

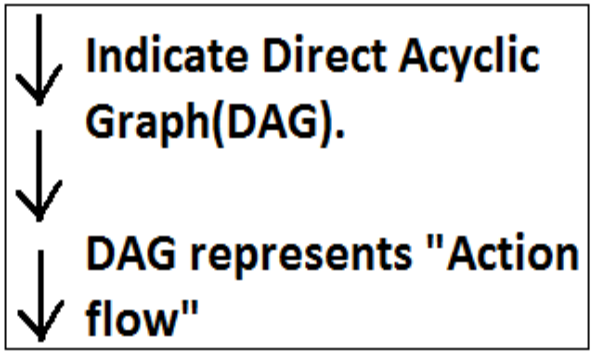
→When an action is encountered a path will be created, which defines the order of execution of the operation.

→This information is remembered in the form of **Lineage Graph**.

→The execution of the operations is represented in the form of a graph called as **Direct Acyclic Graph(DAG)**

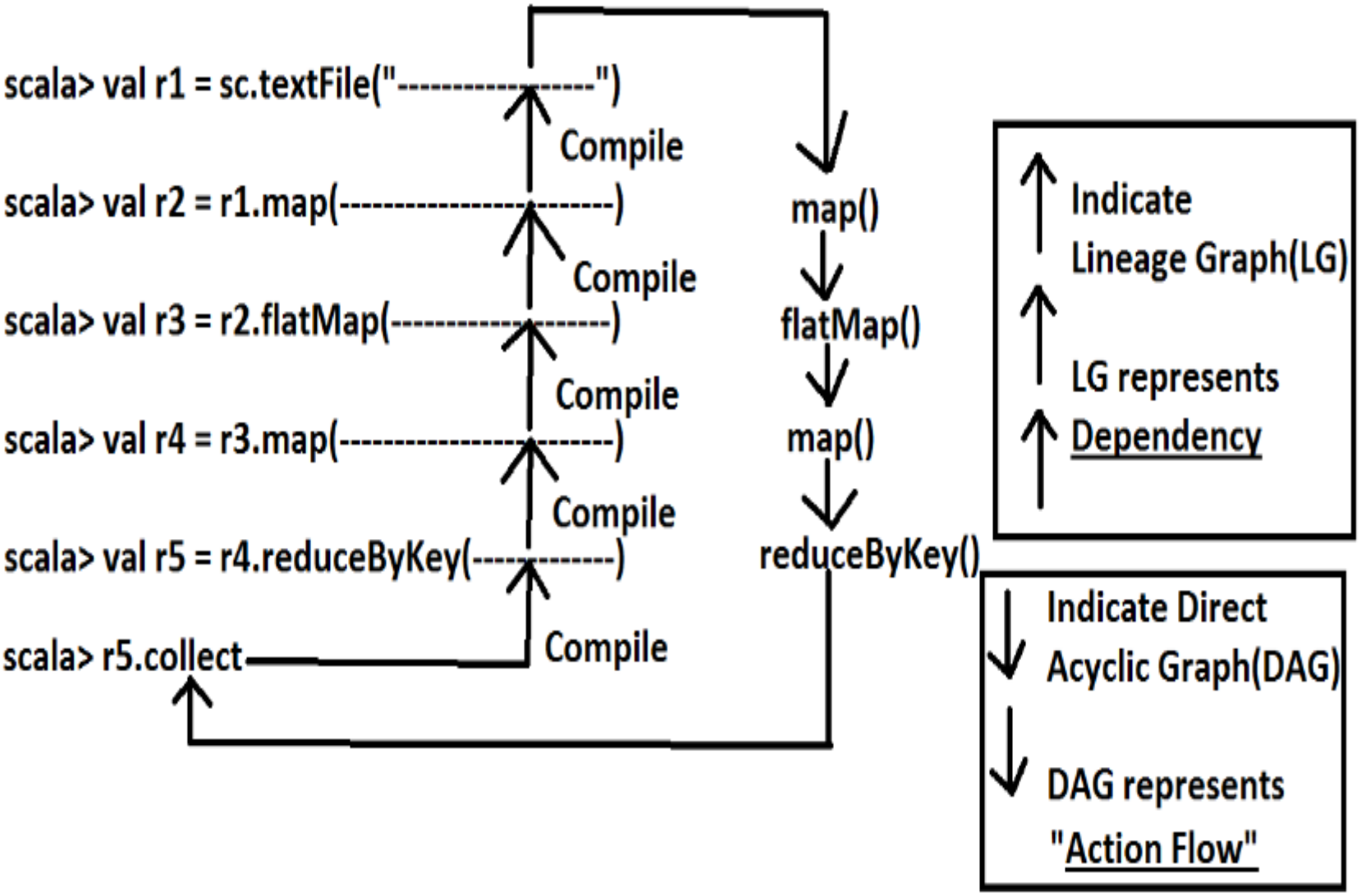


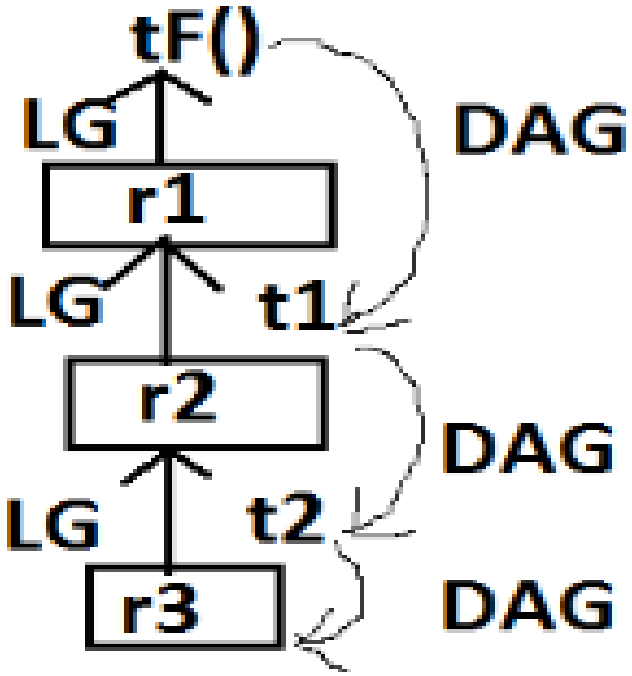




Lineage Graph (LG): It is a graph which contains the lists of operations and the order in which they must be performed.

Direct Acyclic Graph (DAG): It represents the flow of execution of RDDs


scala> val r1 = sc.textFile("----------")
Compile
scala> val r2 = r1.map(----------)
map()
Compile
scala> val r3 = r2.flatMap(----------)
flatMap()
Compile
scala> val r4 = r3.map(----------)
map()
Compile
scala> val r5 = r4.reduceByKey(----------)
reduceByKey()
Compile
scala> r5.collect
Indicate
Lineage Graph(LG)
LG represents
Dependency
Indicate Direct
Acyclic Graph(DAG)
DAG represents
"Action Flow"


tF()
LG
DAG
r1
LG
t1
r2
DAG
LG
t2
r3
DAG

DAG Visualization:

→To See Direct Acyclic Graph (DAG) do the following steps:

Note: go to cloudera's Home directory on cloudera desktop, right click select Create Document, next select Empty file, type file name as file1 and press enter key, Open file1 and type the following text:

apple banana cherry
banana apple cherry
cherry apple banana

Save file1 and come out of editor and open the terminal.

→Start spark application in local mode:

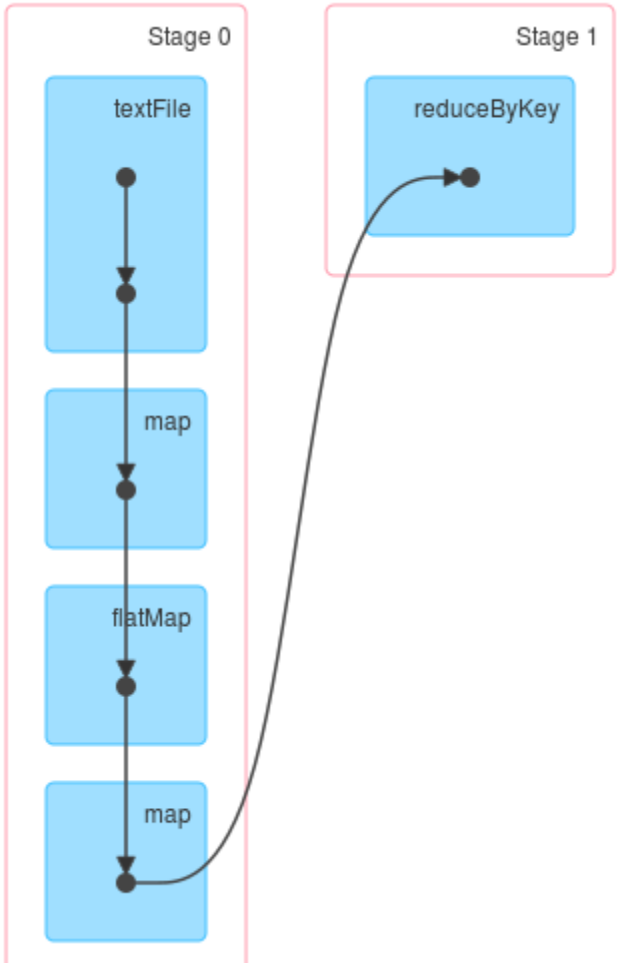
```
[cloudera@quickstart ~]$ spark-shell
```

→Do wordcount program

```
scala> val r1 = sc.textFile("file:///home/cloudera/file1")
scala> val r2 = r1.map(x=>x.split(" "))
scala> val r3 = r2.flatMap(x=>x)
scala> val r4 = r3.map(x=>(x,1))
scala> val r5 = r4.reduceByKey((x,y) => (x+y))
scala> r5.collect
```

→ Open browser and type: localhost:4040

→ The following DAG appears:


Stage 0
textFile
map
flatMap
map
Stage 1
reduceByKey

→For Every action, one DAG will be created. To prove this, do the following steps:

Note: Make sure that file1 is existed in Cloudera's Home directory.

→Start spark application in local mode:

[cloudera@quickstart ~]$ spark-shell

→Do wordcount program

scala> val r1 = sc.textFile("file:///home/cloudera/file1")

scala> val r2 = r1.map(x=>x.split(" "))

scala> val r3 = r2.flatMap(x=>x)

scala> r3.collect

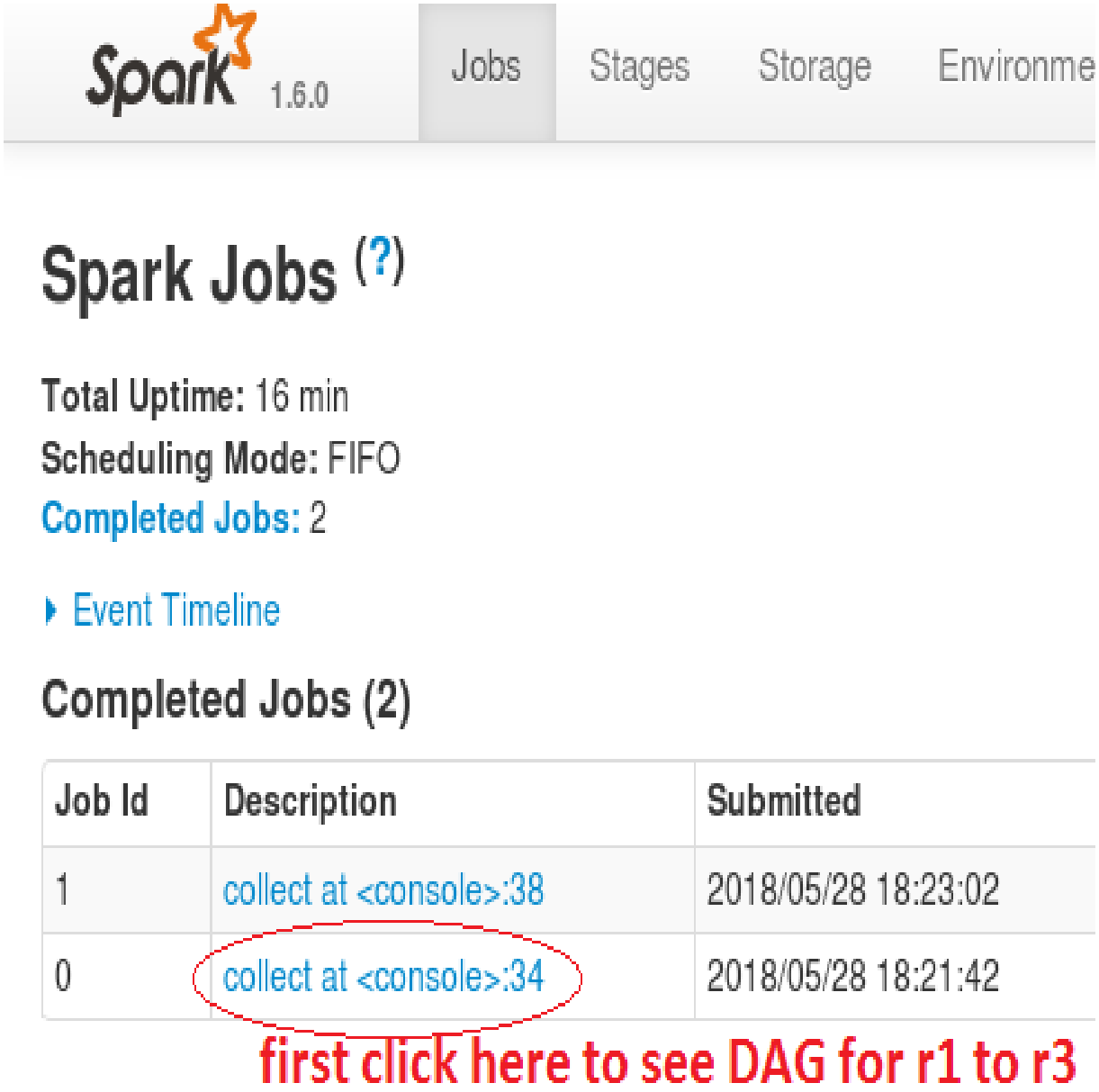
scala> val r4 = r3.map(x=>(x,1))

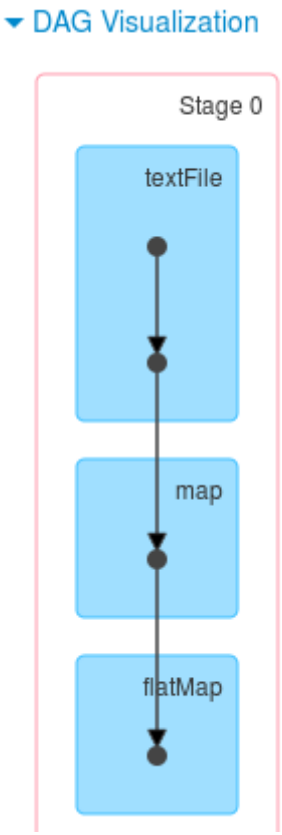
scala> val r5 = r4.reduceByKey((x,y) => (x+y))

scala> r5.collect

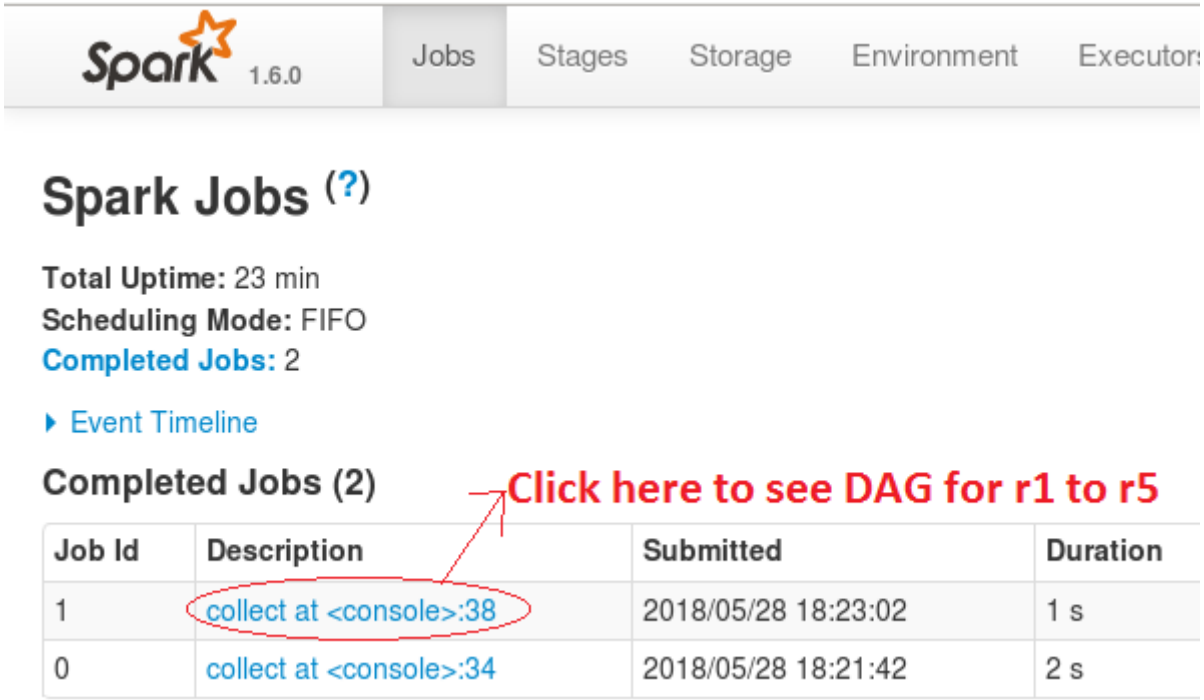
Note: one DAG from r1 to r3 and another DAG from r1 to r5 are created. Let us see this in the browser.

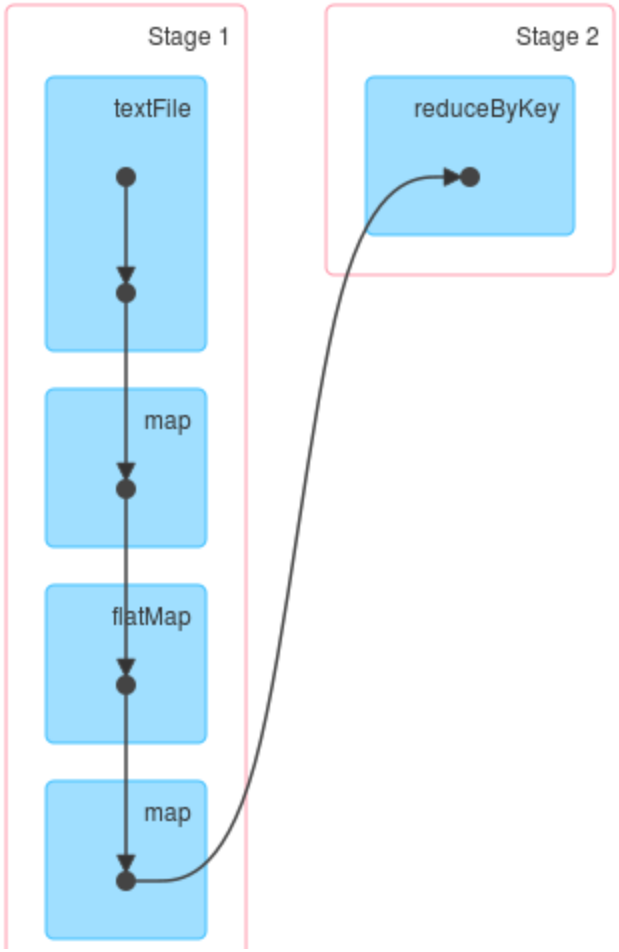
→ Open browser and type: localhost:4040


Spark 1.6.0
Jobs
Stages
Storage
Environme
Spark Jobs (?)
Total Uptime: 16 min
Scheduling Mode: FIFO
Completed Jobs: 2
Event Timeline
Completed Jobs (2)
Job Id
Description
Submitted
1
collect at <console>:38
2018/05/28 18:23:02
0
collect at <console>:34
2018/05/28 18:21:42
first click here to see DAG for r1 to r3


DAG Visualization
Stage 0
textFile
map
flatMap

→Click on back button in browser and click on another link as follows:




Stage 1
textFile
map
flatMap
map
Stage 2
reduceByKey

Persistence:

(Persist = Store/backup)

(((Eg: History feature of Browser. So, History is persisted. So, the user can use the previous history)))

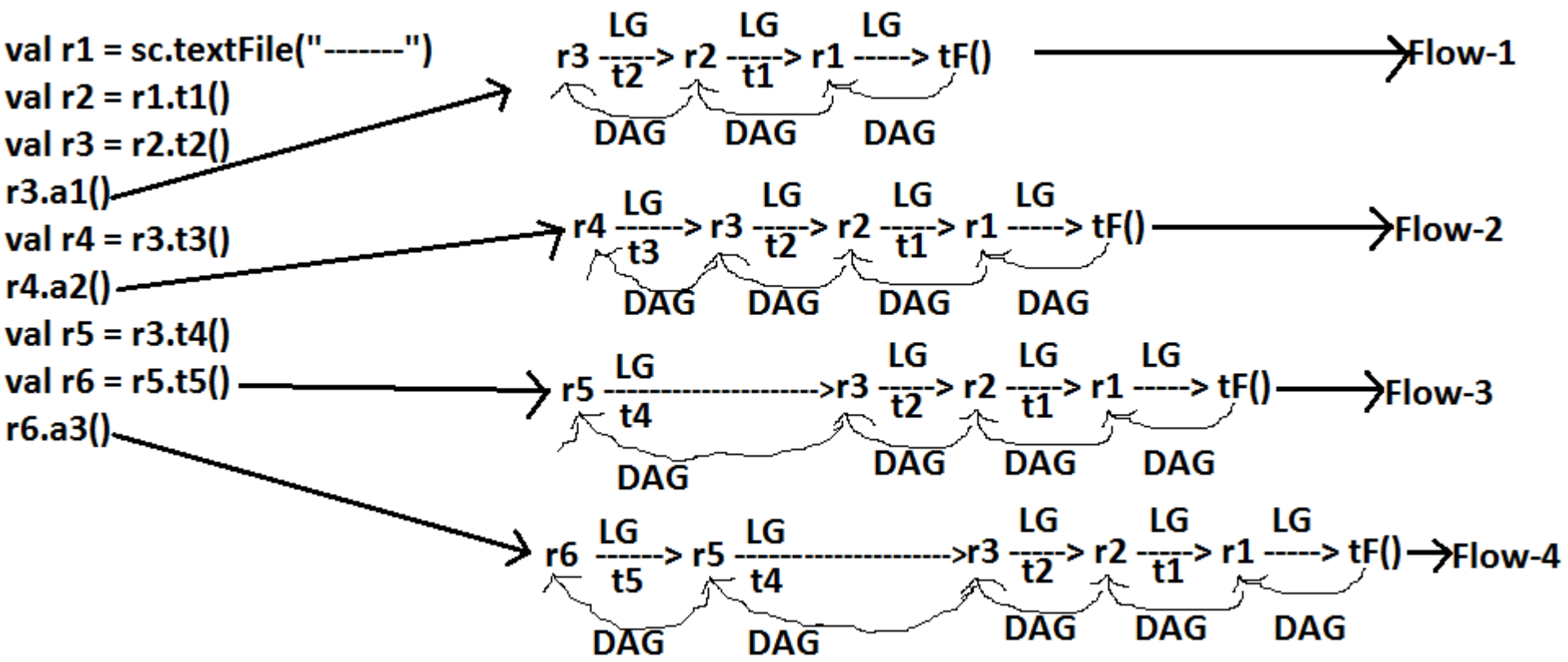
→The general behaviour of spark execution is once the data set is completed it will be deleted from the cluster (by garbage collector).

→In order to improve the performance of the applications we can make some data sets to be in cluster even after their computation (or) even after their completion.

→So, Persistence is for improving performance. So, Common RDDs will be persisted with configuration.

→We can configure persistence of common RDDs and for complex operations/transformations to improve performance. i.e If we persist common RDDs & Common Transformations then we can reuse them so that performance is improved.

→Consider the following example:



Note: t1() - t5() are transformations
a1() - a3() are actions, r1 - r6: RDDs (Resilient Distributed Datasets)
tF() - textFile(), LG - Lineage Graph, DAG - Direct Acyclic Graph

→In the given flows “r3” is the RDD which is used in multiple flows.

→“r3” is the RDD which has the same hierarchy in multiple flows. Therefore if we make “r3” data to be available in the cluster then the flows f2, f3, and f4 will start the execution directly from “r3” itself instead of starting from the beginning. So, “r3” RDD should be persisted. The following the procedure to persist “r3” RDD.

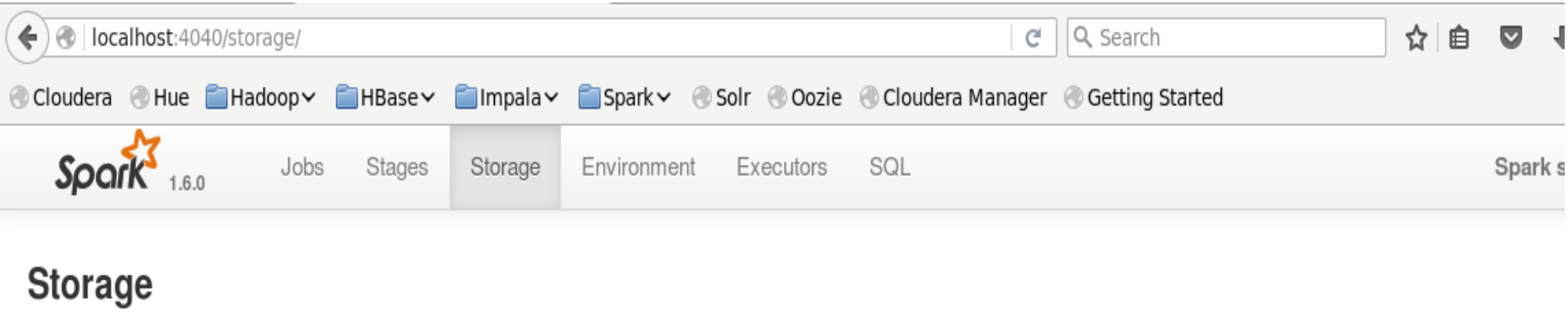
→ Let us persist “r3” RDD in word count program as follows:

Implementation of Persistence:

→start spark application in local mode

[cloudera@quickstart ~]$ spark-shell

Note: After starting spark shell, Go to browser, type localhost:4040 and go to “storage” tab. Check this “storage” tab. Nothing is pesisted.



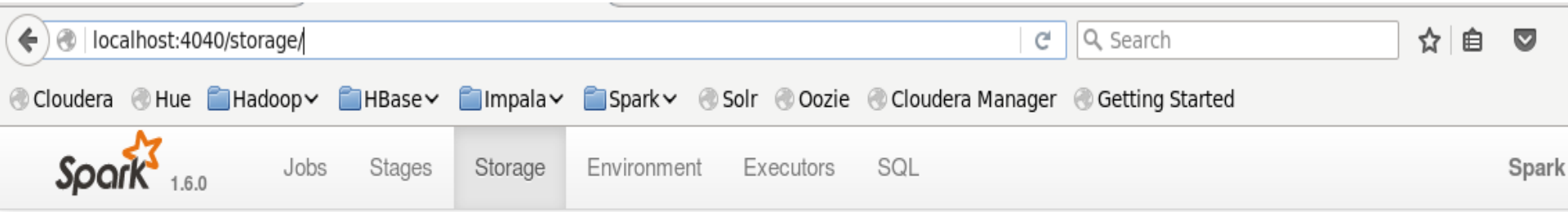
```
scala> val r1 = sc.textFile("file:///home/cloudera/file1")
scala> val r2 = r1.map(x=>x.split(" "))
scala> val r3 = r2.flatMap(x=>x)
scala> r3.persist()    (or)  r3.cache()
```

Note: “r3” RDD is persisted.

```
scala> val r4 = r3.map(x=>(x,1))
scala> val r5 = r4.reduceByKey((x,y) => (x+y))
scala> r5.collect
```

Note: persist() is also called as “Transformation”

Note: Now go to “Storage” tab in browser press F5 to refresh and check the persistence of “r3” RDD.



Storage

RDDs

RDD Name	Storage Level	Cached Partitions	Fraction Cached	Size in Memory	Size in ExternalBlockStore
MapPartitionsRDD	Memory Deserialized 1x Replicated	2	100%	576.0 B	0.0 B

→Data can be pesisted:

1) In memory/disk
2) In Serialized/Deserialized
3) With Replica/Without Replica

Note: Serialized = Compressed. It is like zipping concept in windows. “Serialized” process saves the memory. Ex: 128 MB of RDD data is serialized as 64MB.

→Serialized object is automatically de-serialized.

Note: In spark, maximum of two replicas only can be created.

→ Before making persistence, first perform “Word Count” procedure as follows:

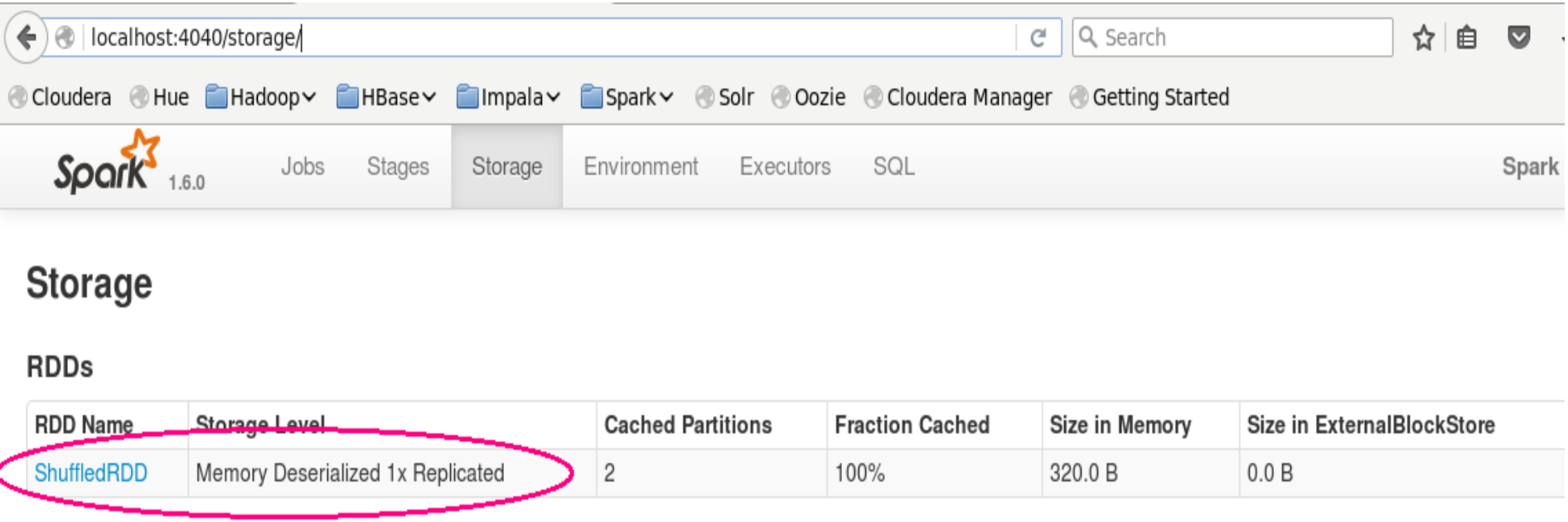
scala> val r1 = sc.textFile("file:///home/cloudera/file1")
scala> val r2 = r1.map(x=>x.split(" "))
scala> val r3 = r2.flatMap(x=>x)
scala> val r4 = r3.map(x=>(x,1))
scala> val r5 = r4.reduceByKey((x,y) => (x+y))
→ Now, let us persist “r5” RDD
→To make use of all options of persistence:
scala> import org.apache.spark.storage._
Note: “_” is like * in java
→Options of persistence: Memory, deserialized and without replica
scala> r5.persist(StorageLevel.MEMORY_ONLY)
Note: These options(Memory, deserialized, without replica) are default even if we use apply simply persist() function without parameters.

Note: Nothing appeared in “Storage” tab of browser.

→ “persistence” is happened after applying “collect” action only

Scala> r5.collect

→ Go to browser, type localhost:4040 and go to “Storage” tab and see that “r5” RDD is persisted.



→To unpersist the above property:

scala> r5.unpersist()

Note: go to browser then go to “Storage”tab press refresh button and see that “r5” is unpersisted (disappeared from the “storage”.)

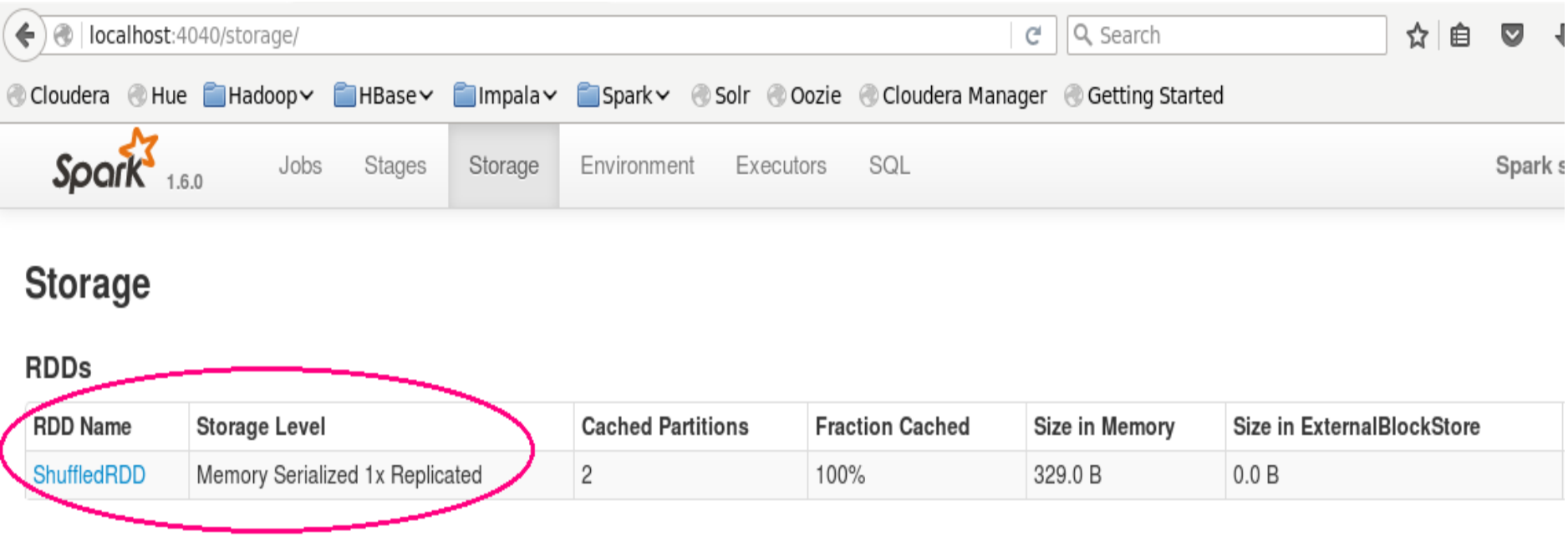
→Options of persistence: Memory, Serialized & without replica

scala> r5.persist(StorageLevel.MEMORY_ONLY_SER)

Note: SER = Serialized

scala>r5.collect

Note: go to browser, press refresh button in “storage” tab.



→To unpersist the above property:

scala>r5.unpersist()

Note: In browser, go to storage tab, press refresh button and observe that “r5” RDD is unpersisted.

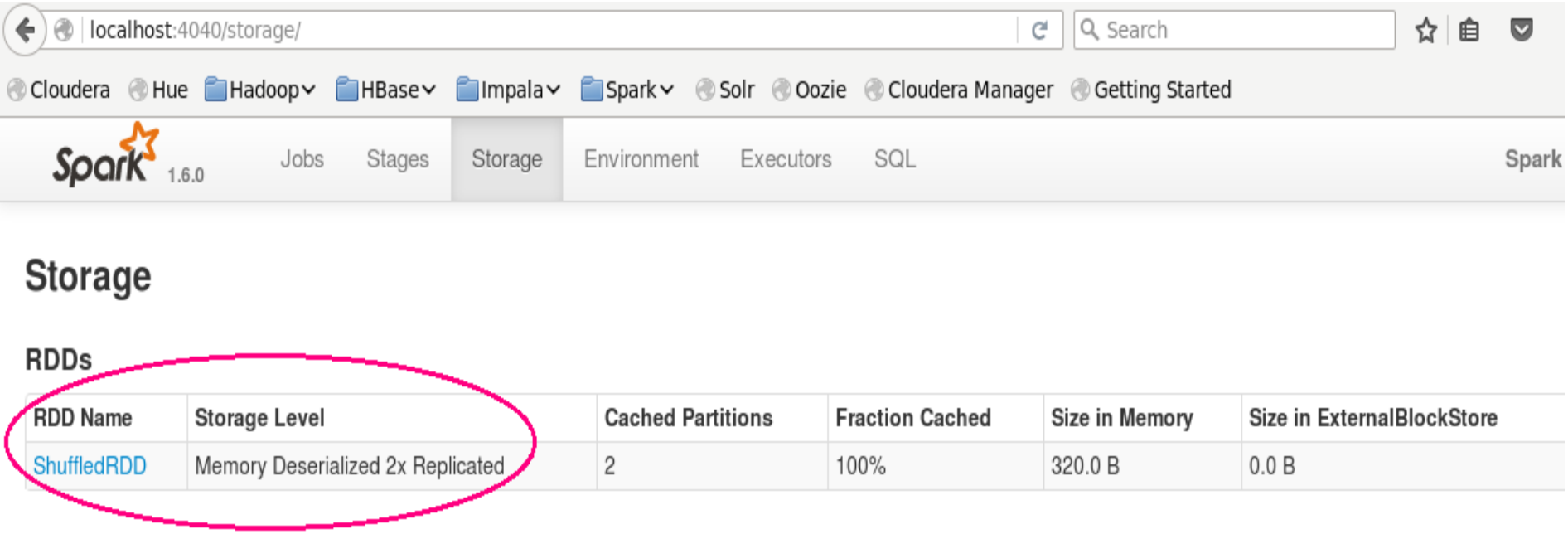
→Options of persistence: Memory, deserialized & 2 replicas

scala> r5.persist(StorageLevel.MEMORY_ONLY_2)

Note: “2” means maximum two replicas only

scala>r5.collect

Note: go to browser, press refresh button in “storage” tab.



→To unpersist the above property:

scala>r5.unpersist()

Note: In browser, go to storage tab, press refresh button and observe that “r5” RDD is unpersisted.

→ Options of Persistence: Disk, Serialized, Without Replica

Note: When we specify Disk then it is always Serialized.

scala>r5.persist(StorageLevel.DISK_ONLY)

scala>r5.collect

Note: Observe in storage tab of browser

scala>r5.unpersist()

→Options of persistence: Disk, serialized, 2 replicas

scala>r5.persist(StorageLevel.DISK_ONLY_2)

scala>r5.collect

Note: Observe in storage tab of browser

scala>r5.unpersist()

→Options of persistence: Memory, Disk, De-serialized, without replica

Note: If space is available in memory then RDD is persisted in memory, if no space is available in memory then RDD is persisted in disk.

scala>r5. persist(StorageLevel.MEMORY_AND_DISK)

scala>r5.collect()

→ Observe the storage tab in the browser for persisted RDD

scala>r5.unpersist()

→Observe the storage tab in the browser for un-persisted RDD.

→Options of persistence: Memory, Disk, Serialized, without replica

Note: If space is available in memory then RDD is persisted in memory, if no space is available in memory then RDD is persisted in disk.

scala>r5.persist(StorageLevel.MEMORY_AND_DISK_SER)
scala>r5.collect()
→Observe the storage tab in the browser for persisted RDD.
scala>r5.unpersist()
→Observe storage tab in browser for un-persisted RDD.
→Options of persistence: Memory, Disk, De-serialized, 2 replicas
scala>r5.persist(StorageLevel.MEMORY_AND_DISK_2)
scala>r5.collect

→To know the spark application:

scala> sc.appName

Output:

res0: String = Spark shell

→ To know spark version:

scala> sc.version

output

Output:

res1: String = 1.6.0

→To know the mode in which spark application is running and in which cluster it is running:

scala> sc.master

res11: String = local[*] → i.e. Local mode

→Start spark application in YARN cluster:

Open another terminal:

[cloudera@quickstart ~]$ spark-shell --master yarn

scala> sc.master

res0: String = yarn-client

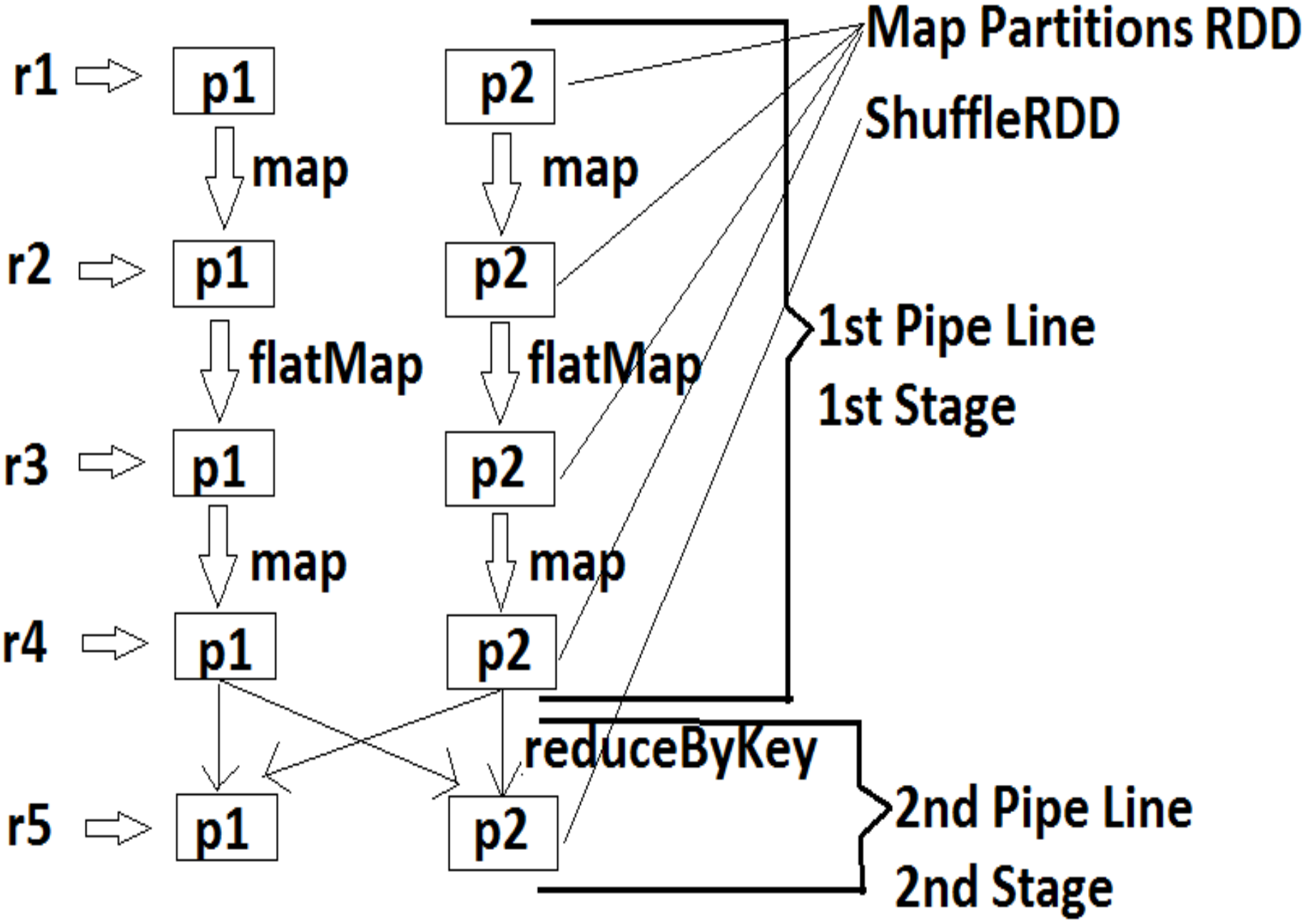
<u>Stages & Tasks</u> (v.v.v.Imp topic)

→There are two types of transformations (Operations).

→One type of transformations perform the operations on the partitions & stores the result in the same location. These partitions do not move the data from one partition to another partition. (So it is called as narrow partition)

→The another type of transformations will require the data in the partitions to be re-distributed. This re-distribution of data among partitions are called as Shuffling (This transformation is also called as Wide Transformation)

The Number of Partitions: 2 in Word Count Example



→Spark Engine executes the operations in terms of stages with one stage at a time.
→Every shuffle operation will create a **new stage,** and sequence of all the operations which do not involve shuffling will form a pipeline and which will fall under one stage.
→When a new stage is created then the number of partitions may increase or decrease or may remain same.
→Each partition will be executed with an executor (i.e. Program to execute partition is called as executor)

→A partition under execution/processing is called task.

→ So, Number of partitions = Number of executors = number of tasks

Note: By default the number of partitions will be created based on number of cores. For example, dual core processors are then 2 partitions will be created.

To know the number of partitions:

Note: By default, two partitions will be created for every text RDD i.e. RDD which contains a text file.

→To know the number of partitions, first create a file in LFS as follows:

```
[cloudera@quickstart ~]$ cat > file1
apple banana cherry
banana apple cherry
cherry apple banana
```

Note: Press **ctrl d** to save and come out of **cat** command.

→Start spark shell:

```
[cloudera@quickstart ~]$ spark-shell
```

scala> val r1 =
sc.textFile("file:///home/cloudera/file1")

Output:

r1: org.apache.spark.rdd.RDD[String] =
file:///home/cloudera/file1 MapPartitionsRDD[1] at
textFile at <console>:27

→ To know the number of partitions:

scala> r1.getNumPartitions

Output:

res0: Int = 2 → i.e. 2 partitions

→Among two partitions, to know how many lines of text is existed in which partition:

```
scala> r1.mapPartitions(x =>
Array(x.size).iterator).collect
```

Output:

res1: Array[Int] = Array(2, 1) → i.e. 2 lines of text in first partition and one line of text in in second partition.

Example-2:

→In the previous example a text file contain three lines and now let us create four lines of text as follows:

```
[cloudera@quickstart ~]$ cat > file1
Spark is super
Spark is powerful
Spark is efficient
Spark is fast
```

Note: Press **ctrl d** to save and come out of **cat** command.

→Start spark shell:

```
[cloudera@quickstart ~]$ spark-shell
```

scala> val r1 =
sc.textFile("file:///home/cloudera/file1")
(or)
Sc.textFile("file:///home/cloudera/file1",2)
((Note:Here, 2 means two partitions. So, minimum two partitions will be created whether we mention it or not.))

Output:

r1: org.apache.spark.rdd.RDD[String] =
file:///home/cloudera/file1 MapPartitionsRDD[1] at textFile at <console>:27

→ To know the number of partitions:

scala> r1.getNumPartitions

Output:

res0: Int = 2 → i.e. 2 partitions

→Among two partitions, to know how many lines of text is existed in which partition:

```
scala> r1.mapPartitions(x =>
Array(x.size).iterator).collect
```

Output:

res1: Array[Int] = Array(3, 1) → i.e. 3 lines of text in first partition and one line of text in in second partition.

Example-3: Three partitions

[cloudera@quickstart ~]$ cat > file1
Spark is super
Spark is powerful
Spark is efficient
Spark is fast
Note: Press **ctrl d** to save and come out of **cat** command.
→Start spark shell:
[cloudera@quickstart ~]$ spark-shell

scala> val r1 =
sc.textFile("file:///home/cloudera/file1", 3)
Note: 3 means three partitions.
Output:
r1: org.apache.spark.rdd.RDD[String] =
file:///home/cloudera/file1 MapPartitionsRDD[1] at
textFile at <console>:27
→ To know the number of partitions:
scala> r1.getNumPartitions
Output:
res0: Int = 3 → i.e. 3 partitions

→Among two partitions, to know how many lines of text is existed in which partition:

```
scala> r1.mapPartitions(x =>
Array(x.size).iterator).collect
```

Output:

res1: Array[Int] = Array(2, 1, 1) → i.e. 2 lines of text in first partition and one line of text in second partition and another one line of text in third partition.

Applying filter() transformation on Array collection:

```
scala> val x =
Array("spark","hadoop","mapreduce","spark core")
```

Output:

```
x: Array[String] = Array(spark, hadoop, mapreduce,
spark core)
```

```
scala> x.filter(a => a.contains("spark"))
```

Output:

```
res0: Array[String] = Array(spark, spark core)
```

Applying map() transformation on Array collection:

Note: map() transformation is used to apply same operation on each record or on each element.

```
scala> val y = Array(1,2,3,4)
```

Output:

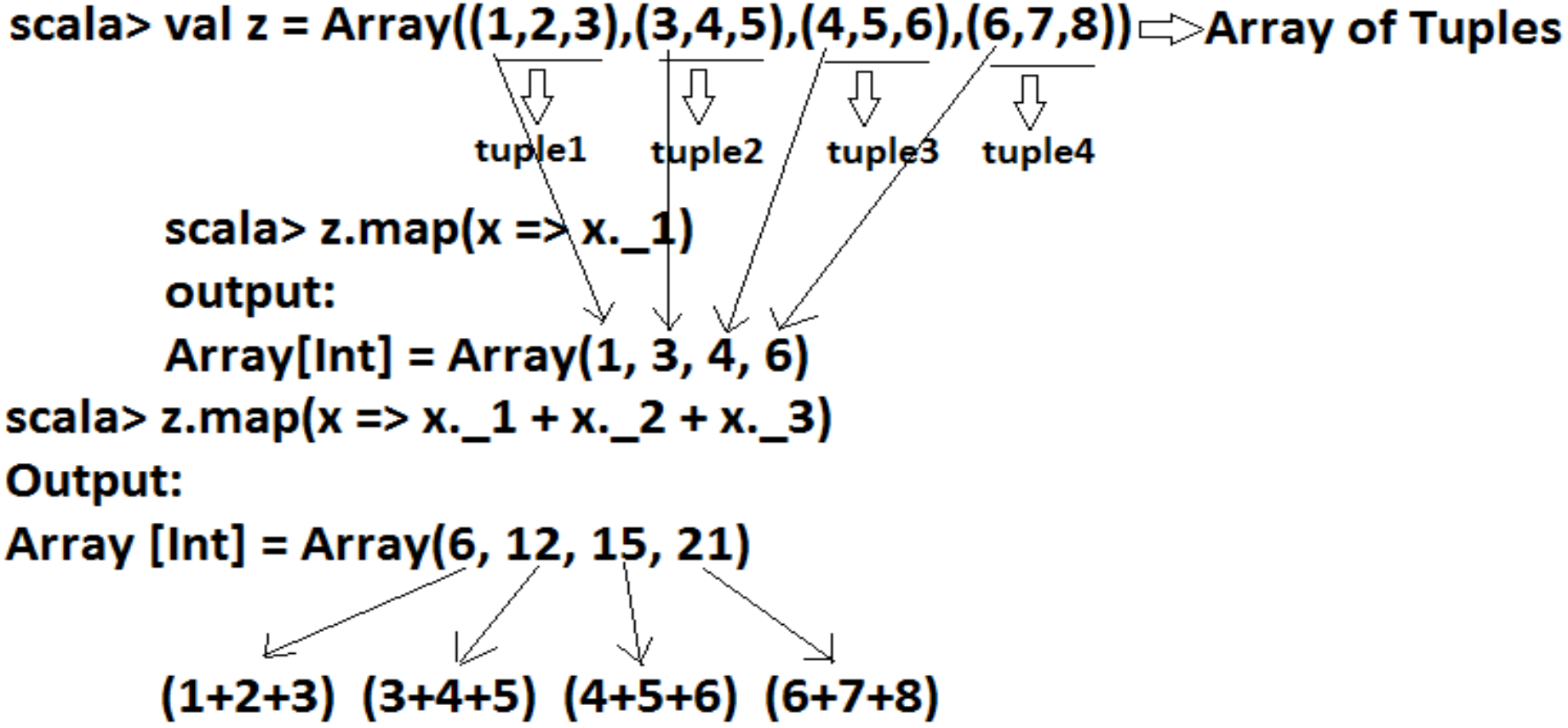
```
y: Array[Int] = Array(1, 2, 3, 4)
```

```
scala> y.map(a => a+10)
```

Output:

```
res1: Array[Int] = Array(11, 12, 13, 14)
```

To understand the next operation this explanation is required:



Same operations are performed as follows:

Applying map() transformation on Array of Tuple:

```
scala> val z = Array((1,2,3),(3,4,5),(4,5,6),(6,7,8))
```

Output:

```
z: Array[(Int, Int, Int)] = Array((1,2,3), (3,4,5), (4,5,6), (6,7,8))
```

```
scala> z.map(x => x._1)
```

Output:

```
res0: Array[Int] = Array(1, 3, 4, 6)
```

```
scala> z.map(x =>x._1 + x._2 + x._3)
```

Output:

```
res1: Array[Int] = Array(6, 12, 15, 21)
```

Applying map() transformation on Array of Arrays

```
scala> val a =
Array(Array(1,2,3),Array(3,4,5),Array(4,5,6),Array(6,7,8))
```

Output:

```
a: Array[Array[Int]] = Array(Array(1, 2, 3), Array(3, 4, 5), Array(4, 5, 6), Array(6, 7, 8))
```

```
scala> a.map(x =>(x(0)+x(1)+x(2)))
```

Output:

```
res2: Array[Int] = Array(6, 12, 15, 21)
```

The following diagram explains this:

```
scala> val a = Array(Array(1,2,3),Array(3,4,5),Array(4,5,6),Array(6,7,8))
```

Output:

Array of Arrays

```
a: Array[Array[Int]] = Array(Array(1, 2, 3), Array(3, 4, 5), Array(4, 5, 6), Array(6, 7, 8))
```

```
scala> a.map(x => (x(0) + x(1) + x(2)))
```

1+2+3=6, Similarly 3+4+5=12 4+5+6=15 6+7+8=21

Output:

```
res2: Array[Int] = Array(6, 12, 15, 21)
```

Applying map() transformation on Array of Lists:

```
scala> val b =
Array(List(1,2,3),List(3,4,5),List(4,5,6),List(6,7,8))
```

Output:

```
b: Array[List[Int]] = Array(List(1, 2, 3), List(3, 4, 5),
List(4, 5, 6), List(6, 7, 8))

scala> b.map(x=>(x(0)+x(1)+x(2)))
```

Output:

```
res4: Array[Int] = Array(6, 12, 15, 21)
```

Applying flatMap() transformation on Array of Arrays:

Note: flatMap() puts all arrays elements into one array.

```
scala> val a =
Array(Array(1,2,3),Array(3,4,5),Array(4,5,6),Array(6,7,8))
```

Output:

```
a: Array[Array[Int]] = Array(Array(1, 2, 3), Array(3, 4, 5), Array(4, 5, 6), Array(6, 7, 8))
```

```
scala> a.flatMap(x=>x)
```

Output:

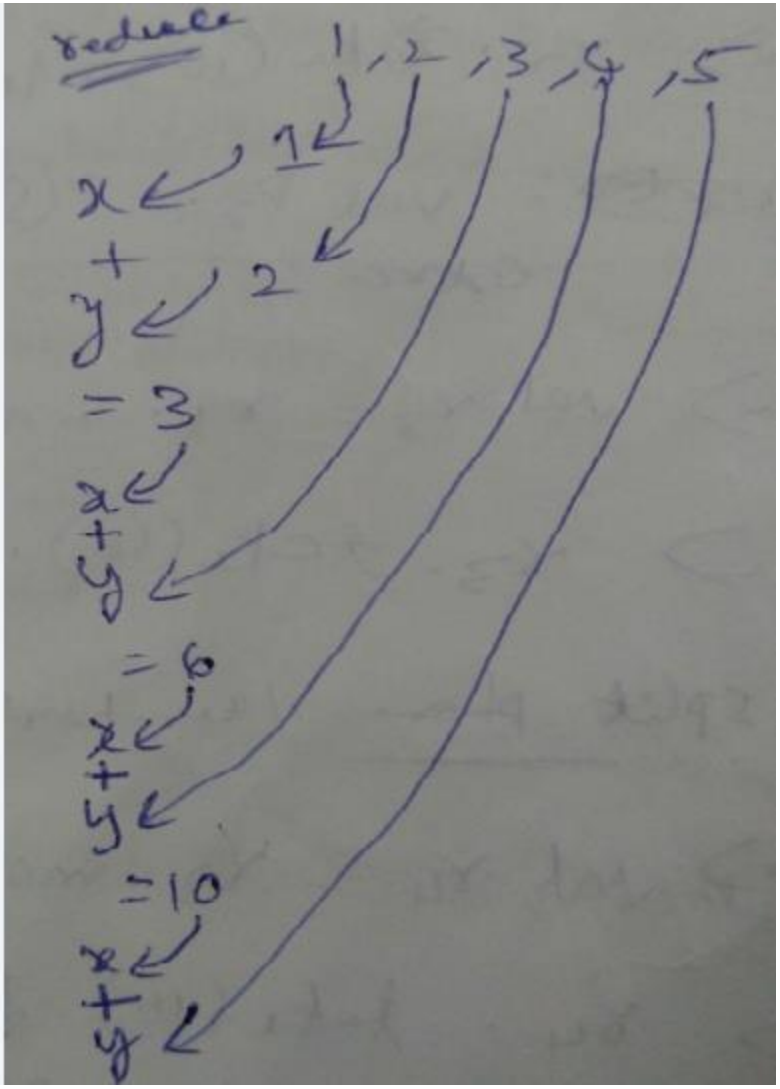
```
res5: Array[Int] = Array(1, 2, 3, 3, 4, 5, 4, 5, 6, 6, 7, 8)
```

Applying reduce() action on Array collection i.e. performing aggregation operation on Array Collection:

```
scala> val data = Array(1,2,3,4,5)
data: Array[Int] = Array(1, 2, 3, 4,5)

scala> data.reduce((x,y) => (x+y))
res0: Int = 15
```

Note: The above **reduce()** method implementation is shown as follows:


reduce
1, 2, 3, 4, 5
x
+
y
= 3
x+y
= 6
x+y
= 10
x+y

Aggregation operations on RDDs

→When aggregation operation is performed on RDD, the operation is first performed on partitions and then on the results of partitions.

→ The following diagram explains this meaning:

Partition-1

1 2 3

1+2 = 3

3+3 = 6

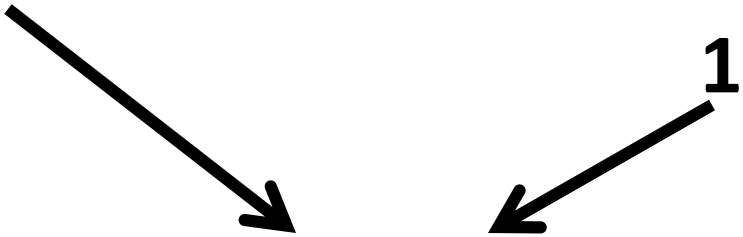
Partition-2

4 5 6 7

4+5 = 9

9+6 = 15

15+7 = 22



6, 22

6 + 22 = 28

Aggregation operations on RDDs

1) reduce()
2) fold()
3) aggregate()

Note: The above methods are also called as RDD actions.

1) reduce():

Note: reduce() is RDD action
This method is used to apply on RDD to perform arithmetic operations over the elements of RDD and we can obtain the addition of elements, subtraction of elements etc. So the elements are reduced to single element as result.

scala> val r1=sc.makeRDD(Array(1,2,3,4,5,6,7))
(or)
scala> val r1=sc.parallelize(Array(1,2,3,4,5,6,7))
Output:
r1: org.apache.spark.rdd.RDD[Int] = ParallelCollectionRDD[0] at makeRDD at <console>:27
Note: ParallelCollectionRDD is an RDD of a collection of elements with numSlices partitions and optional locationPrefs. ParallelCollectionRDD is the result of SparkContext.parallelize and SparkContext.makeRDD methods. So that ParallelCollectionRDD is suitable for parallel processing (or) ParallerCollectionRDD is adapted to be suitable for running on a parallel processing system.

To display the number of partitions of ‘r1’ RDD

scala> r1.getNumPartitions

Output:

res0: Int = 2

To display the number of elements in each partition:

scala>

r1.mapPartitions(x=>Array(x.size).iterator).collect

Output:

res1: Array[Int] = Array(3, 4)

Note: first three elements(1,2,3) exist in Partition1 and remaining four elements (4,5,6,7) exist in Partition2.

scala> val reduce_agg = r1.reduce((x,y) => (x+y))

reduce_agg: Int = 28

→The following is the explanation:

Partition-1

1 2 3

1+2 = 3

3+3 = 6

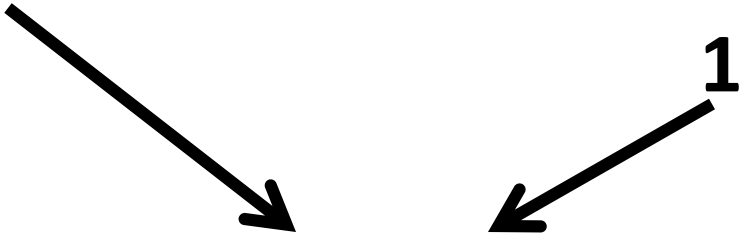
Partition-2

4 5 6 7

4+5 = 9

9+6 = 15

15+7 = 22



6, 22

6 + 22 = 28

fold() aggregation method:

Note: fold() is RDD action.

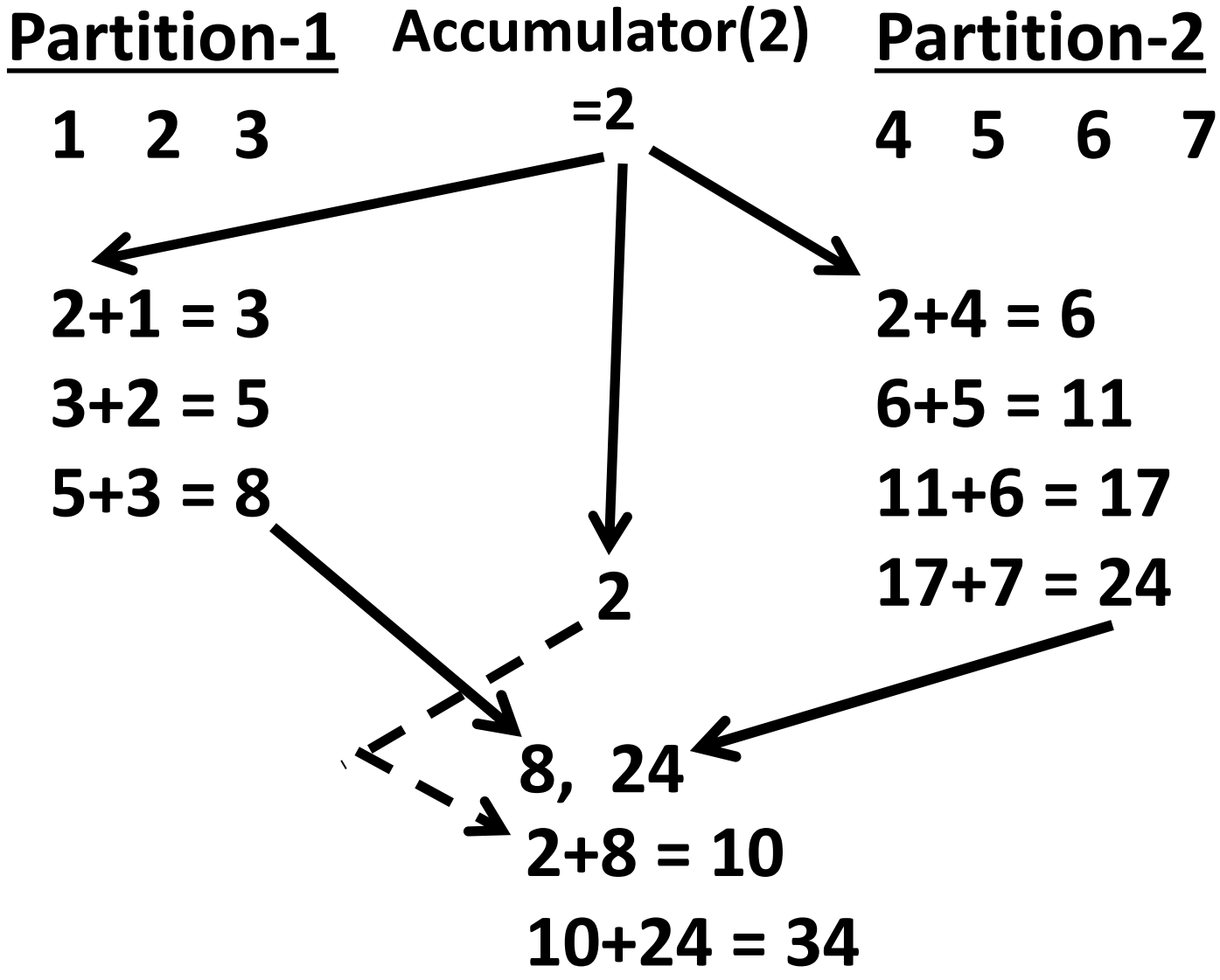
→fold() method initializes a given value at the time of performing the desired operation(Ex: addition operation, Subtraction operation etc.) over the elements of partitions of RDD (Resilient Distributed Data Set). The same given value is accumulated to final result also.

→So, fold() method is also called as Accumulator method.

Task: Accumulate or initialize value 2 to each partition at the time of performing addition operation over the elements of partitions.

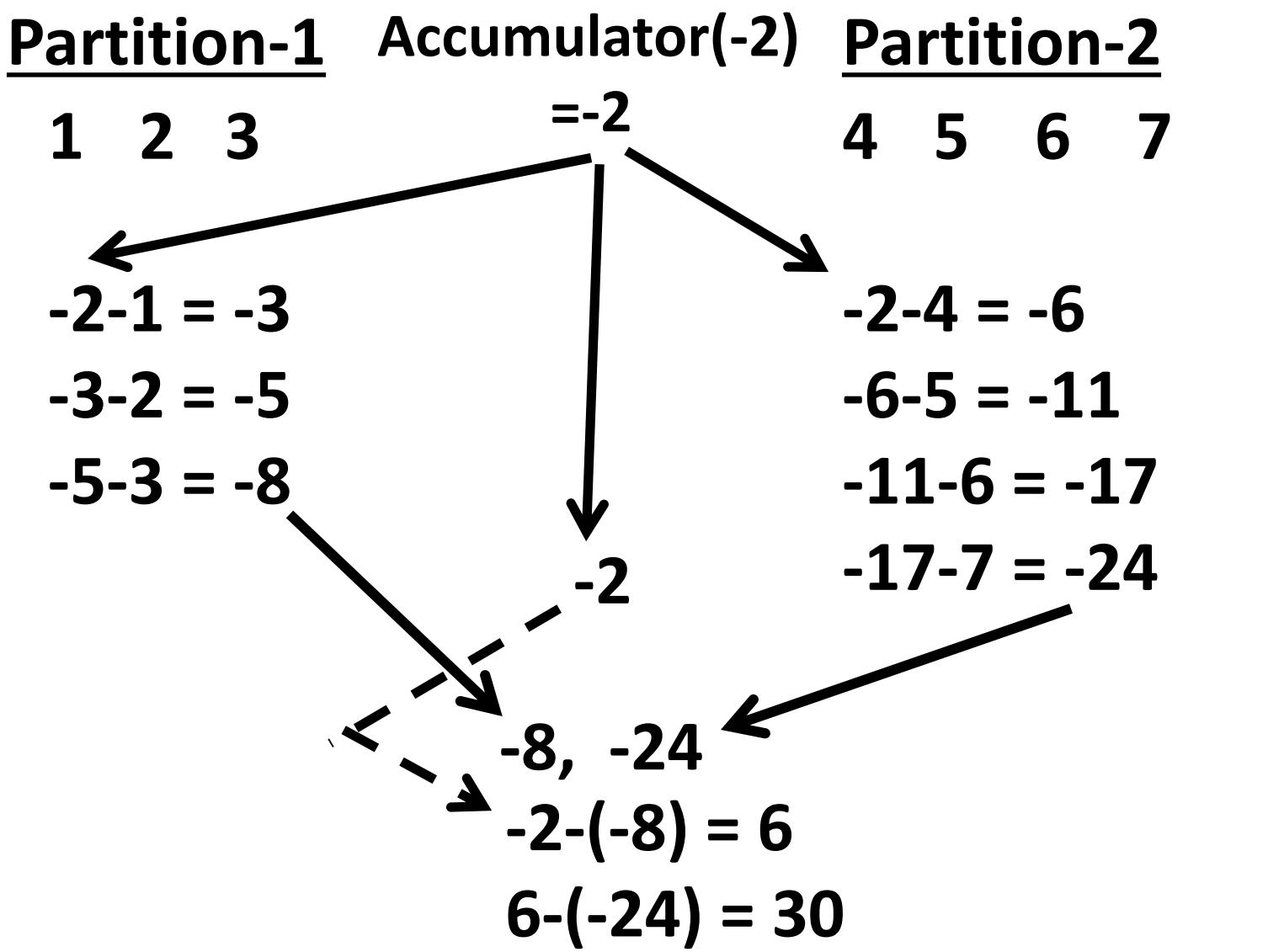
scala> val r1=sc.makeRDD(Array(1,2,3,4,5,6,7))

scala> val fold_aggregation=r1.fold(2)((x,y)=>(x+y))
(or)
scala> val fold_aggregation=r1.fold(2)(_+_)
Output:
fold_aggregation: Int = 34


Partition-1
Accumulator(2)
=2
Partition-2
1 2 3
4 5 6 7
2+1 = 3
3+2 = 5
5+3 = 8
2+4 = 6
6+5 = 11
11+6 = 17
17+7 = 24
2
8, 24
2+8 = 10
10+24 = 34

Simiralry,

scala> val fold_aggregation=r1.fold(-2)((x,y)=>(x-y))
(or)
scala> val fold_aggregation=r1.fold(-2)(_-_)
Output:
fold_aggregation: Int = 30



Explanation: -2-(-8) => -2+8=6 6-(-24) => 6+24=30

aggregate() method:

Note: aggregate() is RDD action.

→fold() method allows only one aggregation on the partitions and on the result of partitions.

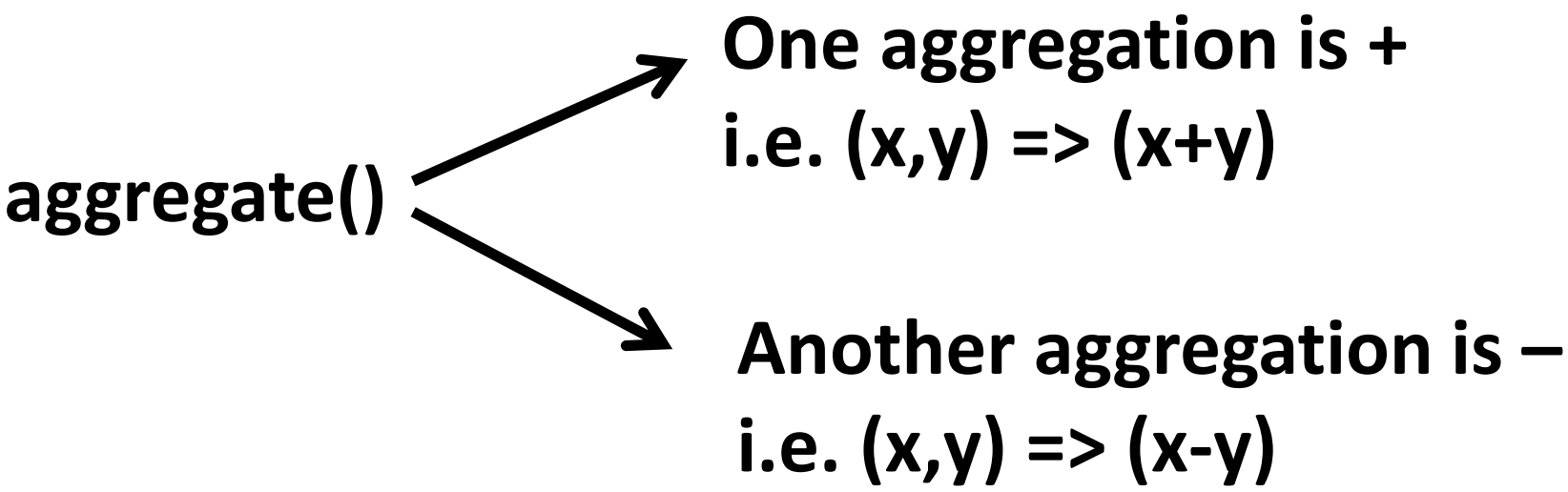
→aggregate() method allows to perform an aggregation on the partitions and another aggregation on the results of the partitions.

scala> val r1=sc.makeRDD(Array(1,2,3,4,5,6,7))

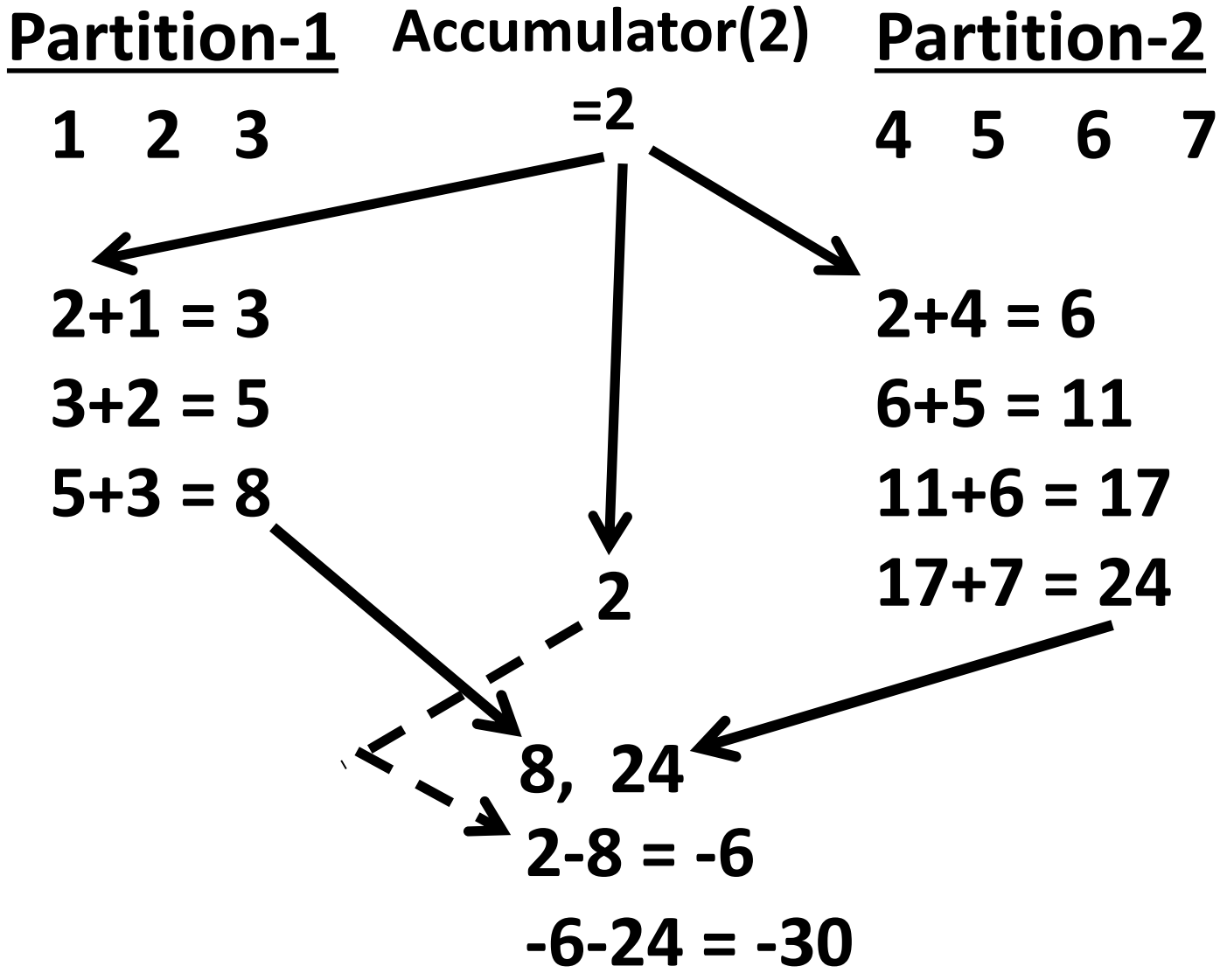
scala> val aggregate=r1.aggregate(2)((x,y)=>(x+y), (x,y)=>(x-y))

Output:

aggregate: Int = -30



See the following diagram also for the explanation:


Partition-1
Accumulator(2)
=2
Partition-2
1 2 3
4 5 6 7
2+1 = 3
3+2 = 5
5+3 = 8
2+4 = 6
6+5 = 11
11+6 = 17
17+7 = 24
2
8, 24
2-8 = -6
-6-24 = -30

Use cases for aggregations:

```
scala> val
data=sc.makeRDD(Array(("Programmer",50000),("Admin",60000),("Architect",70000),("Developer",55000),("TechnicalHead",75000),("TeamLeader",65000)))
```

Output:

```
data: org.apache.spark.rdd.RDD[(String, Int)] = ParallelCollectionRDD[1] at makeRDD at <console>:27
```

Requirement: Highest Salary

```
scala> val highest_salary=data.reduce((x,y)=>if(x._2>y._2)x else y)
```

Output:

```
highest_salary: (String, Int) = (TechnicalHead,75000)
```

Requirement: Lowest Salary

```
scala> val lowest_salary=data.reduce((x,y)=>if(x._2<y._2)x else y)
```

Output:

```
lowest_salary: (String, Int) = (Programmer,50000)
```

Requirement: Less than 55000 salary

scala> val salary=data.fold("salary",55000)((x,y)=>if(x._2<y._2) x else y)

Output:

salary: (String, Int) = (Programmer,50000)

Note:

Here, salary is just a string.

y=55000

→The RDDs which are created by makeRDD() are also called as “Pair RDD”.

→**Pair RDD:** The RDD in which the data in the form of pairs

→**Example of pair RDD:**

scala> val x = sc.makeRDD(Array((“Spark”,1), (“hadoop”,2)))

→Here, (“Spark”,1) is one pairRDD and (“hadoop”,2) is another pairRDD.

→By default Spark will consider first element of pair as **key** and second element of pair as **value**.

Aggregation operation on PairRDD with groupByKey() method:

Note: groupByKey() is a wide transformation.

→groupByKey() method groups the values based on key.

```
scala> val r1 = sc.makeRDD(Array(("a",10),("b",20),("c",30),("d",40),("a",50),("b",60),("c",70),("d",80)))
```

Output:

```
r1: org.apache.spark.rdd.RDD[(String, Int)] = ParallelCollectionRDD[0] at makeRDD at <console>:27
```

scala> val groupbykeyresult = r1.groupByKey()

Output:

groupbykey: org.apache.spark.rdd.RDD[(String, Iterable[Int])] = ShuffledRDD[1] at groupByKey at <console>:29

scala> groupbykeyresult.collect.foreach(println)

Output:

(d,CompactBuffer(40, 80))
(b,CompactBuffer(20, 60))
(a,CompactBuffer(10, 50))
(c,CompactBuffer(30, 70))

Finding the maximum value in every pairRDD

```
scala> val max =
groupbykeyresult.map(x=>(x._1,x._2.max))
```

Output:

```
max: org.apache.spark.rdd.RDD[(String, Int)] =
MapPartitionsRDD[2] at map at <console>:31
scala> val min =
groupbykeyresult.map(x=>(x._1,x._2.min))
```

Finding the minimum value in every pairRDD:

```
scala> min.collect.foreach(println)
(d,40)
(b,20)
(a,10)
(c,30)
```

To display in sorted order:

```
scala> min.sortBy(x => x._1).collect.foreach(println)
```

Output:

```
(a,10)
(b,20)
(c,30)
(d,40)
```

Sum of the elements of pairRDD:

```
scala> val sum =
groupbykeyresult.map(x=>(x._1,x._2.sum))
```

Output:

```
sum: org.apache.spark.rdd.RDD[(String, Int)] =
MapPartitionsRDD[2] at map at <console>:31

scala> sum.collect.foreach(println)
```

Output:

```
(d,120)
(b,80)
(a,60)
(c,100)
```

(or)

```
scala> r1.groupByKey().map(x =>
(x._1,x._2.sum)).collect.foreach(println)
```

Outuput:

```
(d,120)
(b,80)
(a,60)
(c,100)
```

Here, r1 is:

```
scala> val r1 =
sc.makeRDD(Array(("a",10),("b",20),("c",30),("d",40),(
"a",50),("b",60),("c",70),("d",80)))
```

Finding the average of values in pairRDD:

```
scala> val average =
groupbykeyresult.map(x=>(x._1,x._2.sum/x._2.size))

scala> average.collect.foreach(println)
```

Output:

(d,60) → 80+40/2 = 60
(b,40) → 20+60/2 = 40
(a,30) → 10+50/2 = 30
(c,50) → 30+70/2 = 50

Creating the Required number of partitions of RDD:

→First create RDD, for example pair RDD

```
scala> val r1 =
sc.makeRDD(Array(("a",10),("b",20),("c",30),("d",40),(
"a",50),("b",60),("c",70),("d",80)))
```

Output:

```
r1: org.apache.spark.rdd.RDD[(String, Int)] =
ParallelCollectionRDD[11] at makeRDD at
<console>:27
```

→Next, test the number of partitions created by default:

```
scala> r1.getNumPartitions
```

Output:

```
res2: Int = 2
```

→Next, display the number of elements in each partition:

```
scala> r1.mapPartitions(x =>
Array(x.size).iterator).collect
```

Output:

```
res3: Array[Int] = Array(4, 4)
```

→Next, create your desired number of partitions:

scala> val r2 = r1.repartition(3)

Output:

r2: org.apache.spark.rdd.RDD[(String, Int)] = MapPartitionsRDD[9] at repartition at <console>:29

Note: 3 partitions are created for **r1** RDD.

→Next, displaying the number of partitions of RDD:

scala> r2.getNumPartitions

Output:

res4: Int = 3

→Next, to display the number of elements of each partition of **r1** RDD:

```
scala> r2.mapPartitions(x =>
Array(x.size).iterator).collect
```

Output:

```
res5: Array[Int] = Array(2, 4, 2)
```

Coalesce() method:

→It is a transformation method (or) RDD transformation.

→This function decreases the partitions without involving shuffling process.

→But repartition() method also decreases or increases the partitions with shuffling operation or shuffling process.

→Shuffling is a costly operation.

→If output files to be placed to HDFS without extra process called shuffling then use **coalesce()** function.

Steps to use coalesce() method:

Step-1: Create an RDD with (key,value) pairs or Pair RDD or parallelCollection RDD

scala> val r1 = sc.makeRDD(Array(("a",10),("b",20),("c",30),("d",40),("a",50),("b",60),("c",70),("d",80)))

Output:

r1: org.apache.spark.rdd.RDD[(String, Int)] = ParallelCollectionRDD[0] at makeRDD at <console>:27

Step-2: Display the number of partitions in RDD

```
scala> r1.getNumPartitions
```

Output:

```
res0: Int = 2
```

Step-3: Display the number of elements in each partition of RDD:

```
scala> r1.mapPartitions(x =>
Array(x.size).iterator).collect
```

Output:

res3: Array[Int] = Array(4, 4) → 4 elements in first partition and remaining 4 elements in second partition.

Step-4: Applying coalesce() method to decrease two partitions into one partition:

scala> val r2 = r1.coalesce(1)

Output:

r2: org.apache.spark.rdd.RDD[(String, Int)] = CoalescedRDD[4] at coalesce at <console>:29

Step-5: After applying **coalesce() method,** displaying the partitions of RDD:

scala> r2.getNumPartitions

Output:

res4: Int = 1 → two partitions became only one partition.

Step-6: Displaying the number of elements in partition:

scala> r2.mapPartitions(x =>
Array(x.size).iterator).collect

Output:

res9: Array[Int] = Array(8)

Note: Observe in browser also. Type localhost:4040 in address bar of browser and press enter key.
Next click on **Stages** tab and observe under shuffle read and shuffle write columns and they are empty because shuffling did not happen. So, coalesce() function did not do shuffling.

→Do the same steps by using repartition() method as follows then we observe that "shuffling" happens:

Step-1: Create an RDD with (key,value) pairs or Pair RDD or parallelCollection RDD

scala> val r1 = sc.makeRDD(Array(("a",10),("b",20),("c",30),("d",40),("a",50),("b",60),("c",70),("d",80)))

Output:

r1: org.apache.spark.rdd.RDD[(String, Int)] = ParallelCollectionRDD[0] at makeRDD at <console>:27

Step-2: Display the number of partitions in RDD

```
scala> r1.getNumPartitions
```

Output:

```
res0: Int = 2
```

Step-3: Display the number of elements in each partition of RDD:

```
scala> r1.mapPartitions(x =>
Array(x.size).iterator).collect
```

Output:

res3: Array[Int] = Array(4, 4) → 4 elements in first partition and remaining 4 elements in second partition.

Step-4: Applying repartition() method to decrease two partitions into one partition:

scala> val r2 = r1.repartition(1)

Output:

r2: org.apache.spark.rdd.RDD[(String, Int)] = CoalescedRDD[4] at coalesce at <console>:29

Step-5: After applying **repartition() method,** displaying the partitions of RDD:

scala> r2.getNumPartitions

Output:

res4: Int = 1 → two partitions became only one partition.

Step-6: Displaying the number of elements in partition:

scala> r2.mapPartitions(x =>
Array(x.size).iterator).collect

Output:

res9: Array[Int] = Array(8)

Note: Observe in browser also. Type localhost:4040 in address bar of browser and press enter key.
Next click on **Stages** tab and observe under shuffle read and shuffle write columns and they are not empty because shuffling happened. So, repartition() function performs shuffling.

Aggregation operation on PairRDD with reduceByKey() method:

Note: reduceByKey() is a wide transformation.

→reduceByKey() method performs aggregation operations like addition, subtraction, multiplication etc..

```
scala> val r1 = sc.makeRDD(Array(("a",10),("b",20),("c",30),("d",40),("a",50),("b",60),("c",70),("d",80)))
```

Output:

```
r1: org.apache.spark.rdd.RDD[(String, Int)] = ParallelCollectionRDD[0] at makeRDD at <console>:27
```

→Apply reduceByKey() method for performing addition operation on the elements of r1 RDD:

```
scala> val reducebykeyresult = r1.reduceByKey((x,y) => (x+y))
```

Output:

```
reducebykeyresult: org.apache.spark.rdd.RDD[(String, Int)] = ShuffledRDD[12] at reduceByKey at <console>:29
```

→Next, display the data of “reducebykeyresult” RDD:

scala> reducebykeyresult.collect.foreach(println)

Output:

(d,120)

(b,80)

(a,60)

(c,100)

(or)

In a single statement, we can perform as follows:

scala> r1.reduceByKey((x,y) => (x+y)).collect.foreach(println)

(or)

scala> r1.reduceByKey(_+_).collect.foreach(println)

(d,120)

(b,80)

(a,60)

(c,100)

Aggregation operation on PairRDD with foldByKey() method:

→foldByKey() method applies an accumulator to all the elements of partitions and performs specified operation (for example: accumulator is multiplied with every element of partition (or) accumulator is added to every element of partition etc..)

→**Example:**

```
scala> val r1 = sc.makeRDD(Array(("a",10),("b",20),("c",30),("d",40),("a",50),("b",60),("c",70),("d",80)))
```

Output:

```
r1: org.apache.spark.rdd.RDD[(String, Int)] = ParallelCollectionRDD[0] at makeRDD at <console>:27
```

To display the number of partitions:

```
scala> r1.getNumPartitions
```

Output:

```
res2: Int = 2
```

To display the number of elements of each partition:

```
scala>
r1.mapPartitions(x=>Array(x.size).iterator).collect
```

Output:

```
res0: Array[Int] = Array(4, 4)
```

To apply foldByKey() method:

```
scala> val foldbykeyresult = r1.foldByKey(2)((x,y) =>
(x*y))
```

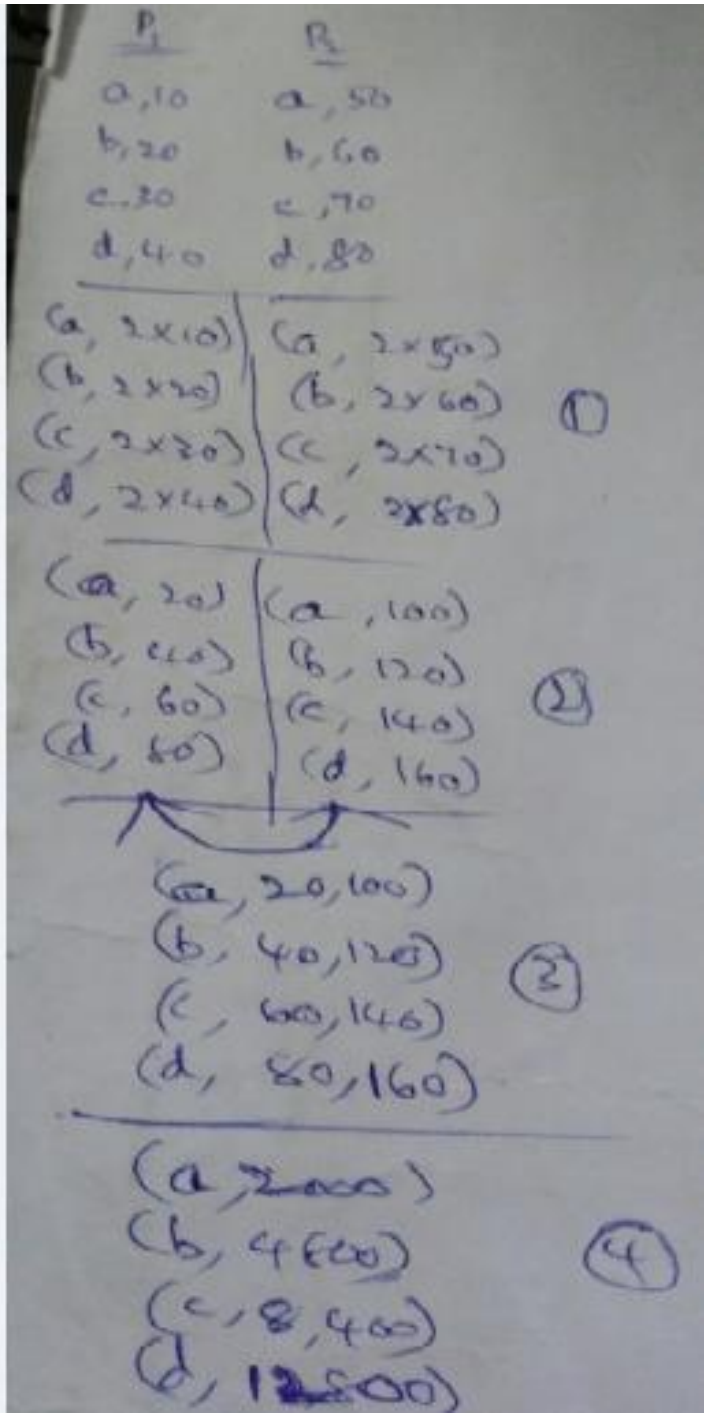
Output:

```
foldbykeyresult: org.apache.spark.rdd.RDD[(String,
Int)] = ShuffledRDD[2] at foldByKey at <console>:29
```

To display result of foldByKey() method:

```
scala> foldbykeyresult.collect.foreach(println)
(d,12800)
(b,4800)
(a,2000)
(c,8400)
```

The following diagram explains the above result.


P1
P2
a,10
b,20
c,30
d,40
a,50
b,60
c,70
d,80
(a, 2x10) (a, 2x50)
(b, 2x20) (b, 2x60)
(c, 2x30) (c, 2x70)
(d, 2x40) (d, 2x80)
1
(a, 20) (a, 100)
(b, 40) (b, 120)
(c, 60) (c, 140)
(d, 80) (d, 160)
2
(a, 20, 100)
(b, 40, 120)
(c, 60, 140)
(d, 80, 160)
3
(a, 2000)
(b, 4800)
(c, 8, 400)
(d, 12800)
4

Aggregation operation on PairRDD with aggregateByKey() method:

→aggregateByKey() method applies an accumulator to all the elements of partitions and performs specified one operation (for example: accumulator is multiplied with every element of partition (or) accumulator is added to every element of partition etc..) and another specified operation on the result of partition results.

→**Example:**

```
scala> val r1 =
sc.makeRDD(Array(("a",10),("b",20),("c",30),("d",40),(
"a",50),("b",60),("c",70),("d",80)))
```

Output:

```
r1: org.apache.spark.rdd.RDD[(String, Int)] =
ParallelCollectionRDD[0] at makeRDD at <console>:27
```

To apply aggregateByKey() method:

```
scala> val aggregatebykeyresult =
r1.aggregateByKey(2)((x,y) => (x+y), (x,y) => (x-y))
```

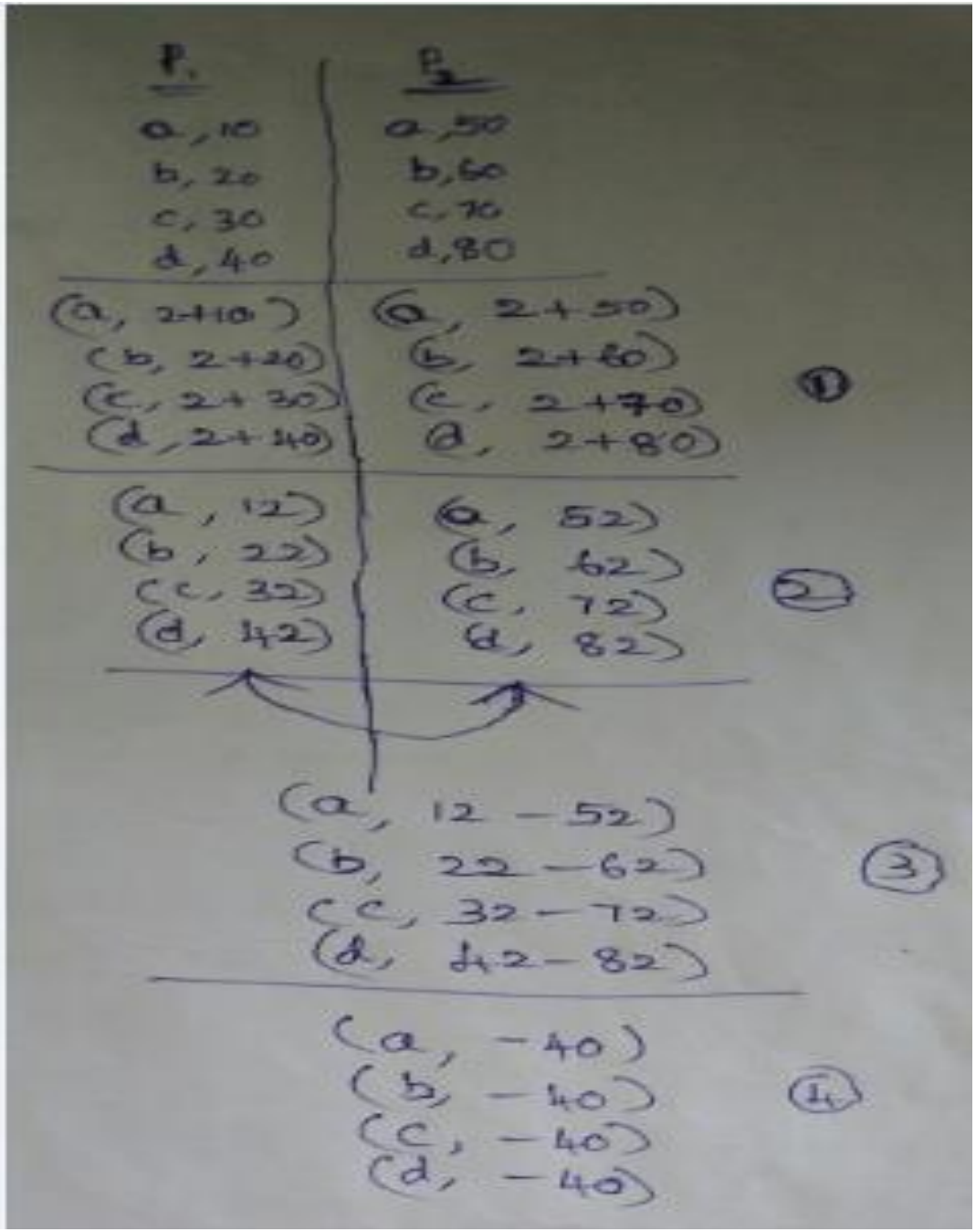
Output:

```
foldbykeyresult: org.apache.spark.rdd.RDD[(String,
Int)] = ShuffledRDD[2] at foldByKey at <console>:29
```

To display result of aggregageByKey() method:

```
scala> aggregatebykeyresult.collect.foreach(println)
(d,-40)
(b,-40)
(a,-40)
(c,-40)
```

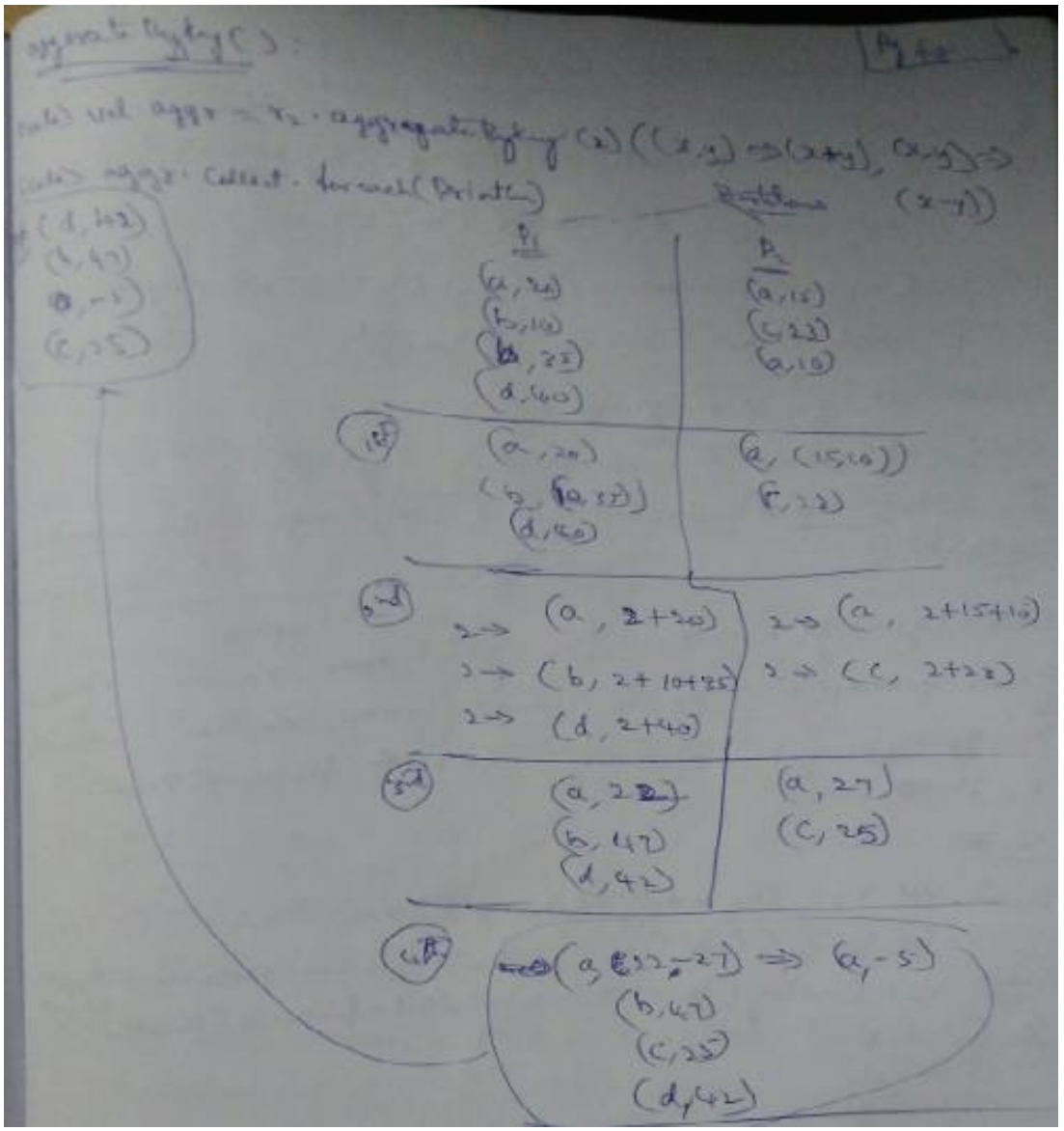
The following diagram explains the above result.


P1
a, 10
b, 20
c, 30
d, 40
P2
a, 50
b, 60
c, 70
d, 80
(a, 2+10) (a, 2+50)
(b, 2+20) (b, 2+60)
(c, 2+30) (c, 2+70)
(d, 2+40) (d, 2+80)
1
(a, 12) (a, 52)
(b, 22) (b, 62)
(c, 32) (c, 72)
(d, 42) (d, 82)
2
(a, 12 - 52)
(b, 22 - 62)
(c, 32 - 72)
(d, 42 - 82)
3
(a, -40)
(b, -40)
(c, -40)
(d, -40)
4

To display in a sorted order:

```
scala> aggregatebykeyresult.sortBy(x =>
x._1).collect.foreach(println)
(a,-40)
(b,-40)
(c,-40)
(d,-40)
```

Example-2: (for your reference)



Example: for sorting

```
scala> val
rdd=sc.makeRDD(Array("spark","scala","hadoop","ma
preduce","hive"))
```

Output:

```
 rdd: org.apache.spark.rdd.RDD[String] =
ParallelCollectionRDD[16] at makeRDD at
<console>:27
```

Applying Sorting:

```
scala> rdd.sortBy(x=>x).collect.foreach(println)
```

Output:hadoop
hive
mapreduce
scala
spark

Displaying tuples in sorted order based on specified key:

```
scala> val rdd=sc.makeRDD(Array((100,"spark"),(200,"scala"),(300,"hadoop"),(400,"hive")))
```

Output:

```
rdd: org.apache.spark.rdd.RDD[(Int, String)] = ParallelCollectionRDD[22] at makeRDD at <console>:27
```

Displaying “rdd”

scala> rdd.collect.foreach(println)

Output:

(100,spark)
(200,scala)
(300,hadoop)
(400,hive)

Displaying in sorted order based on 2nd element:

```
scala> rdd.sortBy(x=>x._2).collect.foreach(println)
```

Output:

```
(300,hadoop)
(400,hive)
(200,scala)
(100,spark)
```

Another Example for sorting:

scala> val
rdd=sc.makeRDD(Array((1,2,3),(2,1,3),(3,1,2)))

Displaying in sorted order based on 3rd element in tuples:

scala> rdd.sortBy(x=>x._3).collect.foreach(println)

Output:

(3,1,2)
(1,2,3)
(2,1,3)

Making desired element of tuple as a key/converting a tuple into (key,value) pairs:

```
scala> val r1=sc.makeRDD(Array((1,2,"spark"),(3,4,"scala"),(5,6,"hadoop")))
```

Output:

```
r1: org.apache.spark.rdd.RDD[(Int, Int, String)] = ParallelCollectionRDD[7] at makeRDD at <console>:27
```

Making 3rd element of tuple as a key (or) converting a tuple into (key,value) pairs:

```
scala> val r2=r1.keyBy(x=>x._3)
```

Output:

```
r2: org.apache.spark.rdd.RDD[(String, (Int, Int, String))] = MapPartitionsRDD[8] at keyBy at <console>:29
```

Displaying (key,value) pairs of "r2" RDD:

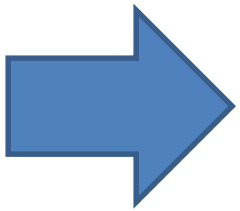
scala> r2.collect.foreach(println)

Output:

(spark,(1,2,spark))
(scala,(3,4,scala))
(hadoop,(5,6,hadoop))

Note: (spark, (1,2,spark))

Key **value**



Pair RDDs

Aggregation operation on PairRDD with cogroup() method:

→ Multiple Pair RDDs can be combined using **cogroup**() method.

Creating "r1" Pair RDD:

```
scala> val r1=sc.makeRDD(Array(("m",50000),("f",52000),("m",60000),("f",55000)))
```

Output:

```
r1: org.apache.spark.rdd.RDD[(String, Int)] = ParallelCollectionRDD[9] at makeRDD at <console>:27
```

Creating “r2” Pair RDD:

```
scala> val
r2=sc.makeRDD(Array(("m",65000),("f",60000),("m",7
0000),("f",75000)))
```

Output:

```
r2: org.apache.spark.rdd.RDD[(String, Int)] =
ParallelCollectionRDD[10] at makeRDD at
<console>:27
```

Applying “cogroup()” method to combine “r1” and “r2” Pair RDDs:

scala> val cogroupRDD=r1.cogroup(r2)

Output:

cogroupRDD: org.apache.spark.rdd.RDD[(String, (Iterable[Int], Iterable[Int]))] = MapPartitionsRDD[1] at cogroup at <console>:31

Note: “r1” and “r2” pair RDDs are combined and stored in “cogroupRDD”

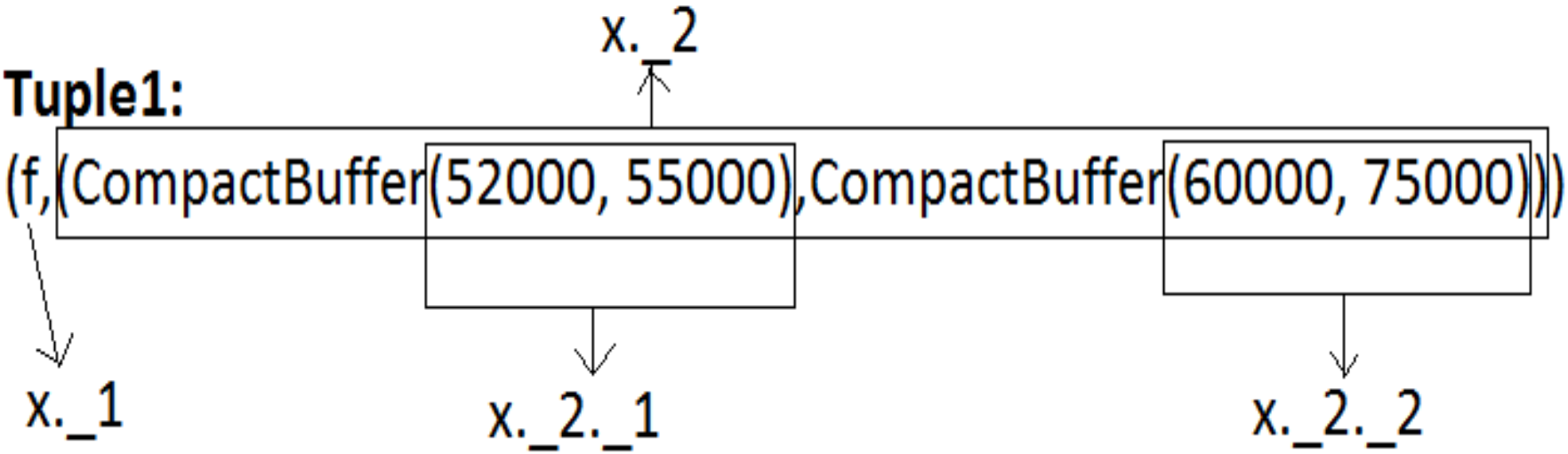
Displaying "cogroupRDD":

scala> cogroupRDD.collect.foreach(println)

Output:

(f,(CompactBuffer(52000, 55000),CompactBuffer(60000, 75000)))
(m,(CompactBuffer(50000, 60000),CompactBuffer(65000, 70000)))

Note:



Tuple2:

(m,(CompactBuffer(50000, 60000),CompactBuffer(65000, 70000)))

Displaying maximum value:

```
scala>
cogroupRDD.map(x=>(x._1,x._2._1.max,x._2._2.max))
.collect.foreach(println)
```

Output:

```
(f,55000,75000)
(m,60000,70000)
```

Loading entire folder of files into one RDD at a time:

→Loading entire folder into one RDD at a time. This folder contains multiple files.

Step-1: create folder namely “FilesFolder” in “clouder’sHome” folder 

(It is existed on desktop of Cloudera)

Step-2: Open “FilesFolder” directory and create some two files. i.e. In “FilesFolder” directory → right click → select “Create Document” option → select “Empty File” option → type file name as “file1” and press enter key → open “file1” and type some content and come out of “file1”. Similarly, create “file2” also.

Step-3: Come to “spark shell” and create one RDD to load all files of “FilesFolder” directory as follows:

```
scala> val r1=sc.textFile("file:///home/cloudera/FilesFolder")
```

Output:

```
r1: org.apache.spark.rdd.RDD[String] = file:///home/cloudera/FilesFolder MapPartitionsRDD[1] at textFile at <console>:27
```

Step-4: To display the data of “r1” RDD:

scala> r1.collect.foreach(println)

Output:

This is file2. → Content of “file1”

This is file1. → Content of “file2”

To load entire folder of files into RDD at a time with (Key, Value) pairs, where "key" is filename(with entire file path) and "Value" is the content of the file:

→ Above two steps are same.

Step-3: Come to "spark shell" and create one RDD to load all files of "FilesFolder" directory as follows:

scala> val r1=sc.wholeTextFiles("file:///home/cloudera/FilesFolder")

Output:

r1: org.apache.spark.rdd.RDD[(String, String)] = file:///home/cloudera/FilesFolder MapPartitionsRDD[1] at wholeTextFiles at <console>:27

Display the file names i.e. “key” part of “r1” RDD:

scala> r1.map(x=>x._1).collect.foreach(println)

Output:

file:/home/cloudera/FilesFolder/file2

file:/home/cloudera/FilesFolder/file1

Display the content of the files, i.e. value part of “r1” RDD:

scala> r1.map(x=>x._2).collect.foreach(println)

Output:

This is file2. → Content of “file2”

This is file1. → Content of “file1”

Zip() method:

→This method makes first RDD elements as **keys** and the second RDD elements as **values.**

(or)

→It is used to map the corresponding elements from two RDDs.

Step-1: Create an RDD with Array of integer elements

```
scala> val r1 = sc.makeRDD(Array(1,2,3))
```

Output:

```
r1: org.apache.spark.rdd.RDD[Int] =
ParallelCollectionRDD[0] at makeRDD at <console>:27
```

Step-2: Create an RDD with Array of Strings:

```
scala> val r2 =
sc.makeRDD(Array("spark","scala","hadoop"))
```

Output:

```
r2: org.apache.spark.rdd.RDD[String] =
ParallelCollectionRDD[1] at makeRDD at <console>:27
```

Step-3: Apply zip() method on two RDDs:

```
scala> val ziprdd = r1.zip(r2)
```

Output:

```
ziprdd: org.apache.spark.rdd.RDD[(Int, String)] =
ZippedPartitionsRDD2[2] at zip at <console>:31
```

Step-4: Display the (key,value) pairs of "ziprdd":

scala> ziprdd.collect

Output:

res0: Array[(Int, String)] = Array((1,spark), (2,scala), (3,hadoop))

(or)

scala> ziprdd.collect.foreach(println)

Output:

(1,spark)

(2,scala)

(3,hadoop)

zipWithIndex() method:

This method is used to create an index to each element of RDD.

Step-1: Create an RDD with Array of string elements

```
scala> val r1 = sc.makeRDD(Array("spark","scala","hadoop"))
```

Output:

```
r1: org.apache.spark.rdd.RDD[String] = ParallelCollectionRDD[3] at makeRDD at <console>:27
```

Step-2: Apply "zipWithIndex()" method on RDD

scala> r1.zipWithIndex.collect

Output:

res2: Array[(String, Long)] = Array((spark,0), (scala,1), (hadoop,2))

(or)

scala> r1.zipWithIndex.collect.foreach(println)

Output:

(spark,0)
(scala,1)
(hadoop,2)

Finding “maximum” element from above “r1” RDD:

scala> r1.max

Output:

res4: String = spark

Finding ”minimum” element from above ”r1” RDD:

scala> r1.min

Output:

res5: String = hadoop

Replacing one delimiter with desired delimiter:

In the data of RDD, programmer can replace the existed delimiter with desired delimiter.

For example: In RDD, if the loaded data contains space as a delimiter then programmer can replace it with comma.

Step-1: Create a file with the words and each word is separated with space delimiter as follows:

Open 

Cloudera's Home directory which is existed on Cloudera Desktop.

Next, right-click and select "Create Document" option → select "Empty file" option → type "file1" as a file name → open "file1" → type the matter as follows:

Spark scala hadoop mapreduce

→Next, Save “file1” and exit from this “file1”

Step-2: open Spark shell as follows:

[cloudera@quickstart ~]$ spark-shell

Step-3: Load “file1” into “r1” RDD:

scala> val r1 = sc.textFile("file:///home/cloudera/file1")

Output:

r1: org.apache.spark.rdd.RDD[String] = file:///home/cloudera/file1 MapPartitionsRDD[7] at textFile at <console>:27

Step-4: Apply "replace()" method to replace space delimiter as comma delimiter and store into "r2" RDD:

```
scala> val r2 = r1.map(x => x.replace(" ",","))
```

Output:

```
r2: org.apache.spark.rdd.RDD[String] =
MapPartitionsRDD[8] at map at <console>:29
```

Step-5: Display Comma Seperated values from "r2" RDD:

```
scala> r2.collect.foreach(println)
```

Output:

```
spark,scala,hadoop,mapreduce
```

Example-2: Replace space with tab

scala> val r1 = sc.textFile("file:///home/cloudera/file1")

Output: r1: org.apache.spark.rdd.RDD[String] = file:///home/cloudera/file1 MapPartitionsRDD[10] at textFile at <console>:27

scala> val r2 = r1.map(x => x.replace(" ","\t"))

Output: r2: org.apache.spark.rdd.RDD[String] = MapPartitionsRDD[11] at map at <console>:29

scala> r2.collect.foreach(println)

Output: spark scala hadoop mapreduce

Example-3: Replace space with next line(i.e. \n)

scala> val r1 = sc.textFile("file:///home/cloudera/file1")

Output: r1: org.apache.spark.rdd.RDD[String] = file:///home/cloudera/file1 MapPartitionsRDD[13] at textFile at <console>:27

scala> val r2 = r1.map(x => x.replace(" ","\n"))

Output:r2: org.apache.spark.rdd.RDD[String] = MapPartitionsRDD[14] at map at <console>:29

scala> r2.collect.foreach(println)

spark
scala
hadoop
mapreduce

Example-3: Replace space with hyphen (i.e. -)

scala> val r1 = sc.textFile("file:///home/cloudera/file1")

Output: r1: org.apache.spark.rdd.RDD[String] = file:///home/cloudera/file1 MapPartitionsRDD[13] at textFile at <console>:27

scala> val r2 = r1.map(x => x.replace(" ",“-"))

Output:r2: org.apache.spark.rdd.RDD[String] = MapPartitionsRDD[14] at map at <console>:29

scala> r2.collect.foreach(println)

spark-scala-hadoop-mapreduce

Spark SQL

→**Spark SQL** is a **Spark** module for structured data processing.

→Spark SQL integrates relational processing with Spark's functional programming. It provides support for various data sources and makes it possible to weave SQL queries with code transformations thus resulting in a very powerful tool.

→**Spark SQL** is a **Spark** module for structured data processing.

→Spark SQL uses “Data Frames and Data Sets” as data storage objects.

→ “Data Frames and Data Sets” are for structured and semi-structured data

→The contexts to create “Data Frames and Data Sets” are: 1) sqlcontext 2) hivecontext 3) sparksession

Features Of Spark SQL

The following are the features of Spark SQL:

1) Integration With Spark

→Spark SQL queries are integrated with Spark programs.

→Spark SQL allows us to query structured data inside Spark programs, using SQL or a DataFrame API which can be used in Java, Scala, Python and R.

2)Uniform Data Access

DataFrames and SQL support a common way to access a variety of data sources, like Hive, Avro, Parquet, ORC(Optimized Row Columnar), JSON, and JDBC. This joins the data across these sources. This is very helpful to accommodate all the existing users into Spark SQL.

3)Hive Compatibility

Spark SQL runs unmodified Hive queries on current data. It rewrites the Hive front-end and meta store, allowing full compatibility with current Hive data, queries, and UDFs.

4) Standard Connectivity

Connection is through JDBC or ODBC. JDBC and ODBC are the industry norms for connectivity for business intelligence tools.

5) Performance And Scalability

→Spark SQL incorporates a cost-based optimizer, code generation and columnar storage to make queries agile alongside computing thousands of nodes using the Spark engine, which provides full mid-query fault tolerance.

→The interfaces provided by Spark SQL provide Spark with more information about the structure of both the data and the computation being performed.

→Internally, Spark SQL uses this extra information to perform extra optimization. Spark SQL can directly read from multiple sources (files, HDFS, JSON/Parquet files, existing RDDs, Hive, etc.). It ensures fast execution of existing Hive queries.

Spark SQL Libraries

Spark SQL has the following four libraries which are used to interact with relational and procedural processing:

1) Data Source API (Application Programming Interface):

This is a universal API for loading and storing structured data.

→ It has built in support for Hive, Avro, JSON, JDBC, Parquet, etc.

→Supports third party integration through Spark packages

→Support for smart sources.

2) DataFrame API:

→A DataFrame is a distributed collection of data organized into named column. It is equivalent to a relational table in SQL used for storing data into tables.

→It is a Data Abstraction and Domain Specific Language (DSL) applicable on structure and semi structured data.

→DataFrame API is distributed collection of data in the form of named column and row.

→It is lazily evaluated like Apache Spark

→Transformations and can be accessed through SQL Context and Hive Context.

→It processes the data in the size of Kilobytes to Petabytes on a single-node cluster to multi-node clusters.

→Supports different data formats (Avro, CSV, Elastic Search and Cassandra) and storage systems (HDFS, HIVE Tables, MySQL, etc.).

→Can be easily integrated with all Big Data tools and frameworks via Spark-Core.

→Provides API for Python, Java, Scala, and R Programming.

3) SQL Interpreter And Optimizer:

→SQL Interpreter and Optimizer is based on functional programming constructed in Scala.
It is the newest and most technically evolved component of SparkSQL.
→It provides a general framework for transforming trees, which is used to perform analysis/evaluation, optimization, planning, and run time code spawning.
→This supports cost based optimization (run time and resource utilization is termed as cost) and rule based optimization, making queries run much faster than their RDD (Resilient Distributed Dataset) counterparts.

e.g. Catalyst is a modular library which is made as a rule based system. Each rule in framework focuses on the distinct optimization.

4) SQL Service:

SQL Service is the entry point for working along structured data in Spark. It allows the creation of DataFrame objects as well as the execution of SQL queries.

Simple Build Tool (SBT):

sbt is an open-source build tool for Scala and Java projects, similar to Java's Maven and Ant.

Steps for installing SBT:

→In google, type sbt download → click on sbt – Download link → come down in the page → sbt 0.13.16 (.zip) (.tgz) (.msi) (or) sbt 1.0.0 (.zip) (.tgz) (.msi)

Here, click on .tgz link

→ In Cloudera'sHome directory → create INSTALL directory → paste sbt 0.13.16.tgz file into INSTALL directory → right click on sbt 0.13.16.tgz file and select "Extract Here" option → sbt folder will be created.

→ **Set the path in "bashrc" file as follows:**

[cloudera@quickstart ~]$ sudo gedit ~/.bashrc

export SBT_HOME=/home/cloudera/INSTALL/sbt

export PATH=$SBT_HOME/bin:$PATH

Note: save the file and come out of "bashrc" file

→**Update the changes done to "bashrc" file as follows:**

[cloudera@quickstart ~]$ source ~/.bashrc

→ **Create "SparksqlProject" directory**

[cloudera@quickstart ~]$ mkdir sparksqlproject

[cloudera@quickstart ~]$ cd sparksqlproject

→**Start sbt:**

[cloudera@quickstart sparksqlproject]$ sbt

> →This is sbt prompt

→Now, Check libraries available in SBT:

> libraryDependencies

[info] * org.scala-lang:scala-library:2.10.6

→Now, add “spark core” library to SBT as follows:

> set libraryDependencies += "org.apache.spark" % "spark-core_2.10" % "1.6.0"

Note: Here, 2.10 is scala version and 1.6.0 is spark version.

→ Now add “spark sql” libraries to SBT as follows:

> set libraryDependencies += "org.apache.spark" % "spark-sql_2.10" % "1.6.0"

→Now, save the changes with **session save** command:

> session save

→ Now, load the above libraries:

> reload

→Now, check all the libraries:

> libraryDependencies

Output:

[info] * org.scala-lang:scala-library:2.10.6

[info] * org.apache.spark:spark-core_2.10:1.6.0

[info] * org.apache.spark:spark-sql:1.6.0-cdh 5.12.0

Note: Next when we give **console** command then the above specified libraries are downloaded from **maven** repository.

→Now, type **console** command to start **scala>** prompt

> console

Note: Now, downloading starts from maven repository (Internet is required). So, this downloading takes few minutes for the first time only.

→Now, Scala prompt comes as follows:

scala>

(((

Note: If we want to come out of scala> prompt then give **:q** and then SBT prompt come, if we want to come out of SBT prompt then give **exit** command

)))

(((Skip this slide:

Extra Reference only:

→If any **major:minor** Exception come during execution then do the following steps:

Open google → type: jdk-8u151-linux-x64.tar → After downloading, copy this file "jdk-8u151-linux-x64.tar" into INSTALL folder of Cloudera's Home folder

(available on cloudera desktop)

→ Right click on "jdk-8u151-linux-x64.tar" i.e. "jdk-8u151-linux-x64.tar.gz" → select "extract here" option → open **bashrc** file as follows:

(((Skip this slide: This slide is continuation to the above slide

[cloudera@quickstart ~]$ gedit ~/.bashrc

Note: Remove the following lines:

export JAVA_HOME=/usr/java/jdk1.7.0_67-cloudera

export PATH=$JAVA_HOME/bin:$PATH

Note: Type the following lines

export JAVA_HOME=/home/cloudera/INSTALL/jdk1.8.0_151

export PATH=$JAVA_HOME/bin:$PATH

Note: Save the file and exit from "bashrc" file and update the changes as follows:

[cloudera@quickstart ~]$ source ~/.bashrc

(((Skip this slide: This slide is continuation to the above slide:

So when you work with hadoop, JDK 1.7 is required and when you work with sparkSQL then JDK 1.8 is required.

)))

(((Extra Reference:

Interviewer may ask how to get **libraryDependencies += "org.apache.spark" % "spark-sql_2.10" % "1.6.0"** .

To get this reference do the following steps:

(Question also may be: how to use **maven** repository)

Note: Internet connection is required.

On Windows operating system, in Google chrome browser, type maven repository → Click on "Maven Repository: Search/Browse/Explore" link → type spark-sql in the search text box and click on "search" button → "1. **Spark Project SQL**" link → come down of the page, under 1.6.x, again under 1.6.0, click on 2.10 link (Note: 1.6.0 is spark version number, 2.10 is scala language version number) → click on SBT tab →

copy libraryDependencies += "org.apache.spark" %% "spark-sql" % "1.6.0" to console where sbt is running.
i.e. as follows:
> set libraryDependencies += "org.apache.spark" %% "spark-sql" % "1.6.0”

And Similarly, how to get **libraryDependencies += "org.apache.spark" %% "spark-core" % "1.6.0"**
On Windows operating system, in Google chrome browser, type maven repository → Click on “Maven Repository: Search/Browse/Explore” link →

type spark-core in the search text box and click on "search" button → "1. **Spark Project Core**" link → come down of the page, under 1.6.x, again under 1.6.0, click on 2.10 link (Note: 1.6.0 is spark version number, 2.10 is scala language version number) → click on SBT tab → copy libraryDependencies += "org.apache.spark" %% "spark-core" % "1.6.0" to console where sbt is running. i.e. as follows:

> set libraryDependencies += "org.apache.spark" %% "spark-core" % "1.6.0"

)))

To load “Text” data into Data storage object i.e. “Data Frame”: (i.e. load Text data from LFS to Data Frame object)

Step-1: Create “sparksqlProject” directory as follows:

[cloudera@quickstart ~]$ mkdir sparksqlproject

Step-2: Go to “sparksqlproject” directory as follows:

[cloudera@quickstart ~]$ cd sparksqlproject

[cloudera@quickstart sparksqlproject]$

Step-3: Start “SBT (Simple Build Tool)” as follows:

[cloudera@quickstart sparksqlproject]$ sbt

Step-4: add “spark core library” as follows:

> set libraryDependencies += "org.apache.spark" % "spark-core_2.10" % "1.6.0"

Step-5: Add “spark sql library” as follows:

> set libraryDependencies += "org.apache.spark" % "spark-sql_2.10" % "1.6.0"

Step-6: give “session save” command as follows:

> session save

Step-7: give “reload” command as follows:

> reload

Step-8: give “console” command to start “scala>” prompt or scala interpreter as follows:

> console

Step-9: import the following package:

scala> import org.apache.spark._

Output: import org.apache.spark._

Step-10: create “conf” object as follows:

scala> val conf = new SparkConf().setAppName("SparkSQLApplication").setMaster("local[2]")

Note: Here “2” means the number of cores in processor.

Step-11: create “sc” object i.e. spark context object as follows:

scala> val sc = new SparkContext(conf)

Note: process goes on….

Step-12: import the following package:

scala> import org.apache.spark.sql._

Output: import org.apache.spark.sql._

Step-13: Create “SQL context” object as follows:

scala> val sqlc = new SQLContext(sc)

Output: sqlc: org.apache.spark.sql.SQLContext = org.apache.spark.sql.SQLContext@1c7997e6

Step-14: Open another terminal and create the text file as follows:

[cloudera@quickstart ~]$ cat > file1

Spark is a powerful framework to process:

Structured Data

Semi-Structured Data

and Un-Structured Data

Note: Press **ctrl d** to save and come out of **cat** command.

[cloudera@quickstart ~]$

Step-15: Again come back to the terminal where **scala>** prompt is running and give the following command to load the text data into “Data Frame” storage object:

```
scala> val df_text = sqlc.read.format("text").load("file:///home/cloudera/file1")
```

Output: 18/10/15 05:21:40 INFO TextRelation: Listing file:/home/cloudera/file1 on driver

```
df_text: org.apache.spark.sql.DataFrame = [value: string]
```

(or)

scala> val df_text = sqlc.read.text("file:///home/cloudera/file1")

Output: 18/10/15 05:23:50 INFO TextRelation: Listing file:/home/cloudera/file1 on driver

df_text: org.apache.spark.sql.DataFrame = [value: string]

Step-16: Display “df_text” storage object as follows:

scala> df_text.show **(or)** df_text.show(30) →

Meaning is displaying first 30 records

Note: process goes on…

Output:

```
+--------------------+
|               value|
+--------------------+
|Spark is a powerf...|
|     Structured Data|
|Semi-Structured Data|
|and Un-Structured...|
+--------------------+
```

To load "Text" data into Data storage object i.e. "Data Frame": (i.e. load Text data from HDFS to Data Frame object)

((((

Note: "Steps 1 to 14" need not be done again because they are already done in the above section. So, start directly from **Step 15.**

If you want to from the beginning then do the following:

To remove the existed "sparksqlproject" directory in a single step first come out of scala> prompt, SBT prompt, and come out of "sparksqlproject directory and then give the following command:

[cloudera@quickstart ~]$ rm -rf sparksqlproject **))))**

Steps from the beginning:

Step-1: Create “sparksqlProject” directory as follows:

[cloudera@quickstart ~]$ mkdir sparksqlproject

Step-2: Go to “sparksqlproject” directory as follows:

[cloudera@quickstart ~]$ cd sparksqlproject

[cloudera@quickstart sparksqlproject]$

Step-3: Start “SBT (Simple Build Tool)” as follows:

[cloudera@quickstart sparksqlproject]$ sbt

Step-4: add “spark core library” as follows:

> set libraryDependencies += "org.apache.spark" % "spark-core_2.10" % "1.6.0"

Step-5: Add “spark sql library” as follows:

> set libraryDependencies += "org.apache.spark" % "spark-sql_2.10" % "1.6.0"

Step-6: give “session save” command as follows:

> session save

Step-7: give “reload” command as follows:

> reload

Step-8: give “console” command to start “scala>” prompt or scala interpreter as follows:

> console

Step-9: import the following package:

scala> import org.apache.spark._

Output: import org.apache.spark._

Step-10: create “conf” object as follows:

scala> val conf = new SparkConf().setAppName("SparkSQLApplication").setMaster("local[2]")

Note: Here “2” means the number of cores in processor.

Step-11: create “sc” object i.e. spark context object as follows:

scala> val sc = new SparkContext(conf)

Note: process goes on….

Step-12: import the following package:

scala> import org.apache.spark.sql._

Output: import org.apache.spark.sql._

Step-13: Create “SQL context” object as follows:

scala> val sqlc = new SQLContext(sc)

Output: sqlc: org.apache.spark.sql.SQLContext = org.apache.spark.sql.SQLContext@1c7997e6

Step-14: Open another terminal and create the text file as follows:

[cloudera@quickstart ~]$ cat > file1

Spark is a powerful framework to process:

Structured Data

Semi-Structured Data

and Un-Structured Data

Note: Press **ctrl d** to save and come out of **cat** command.

[cloudera@quickstart ~]$

→Place this “file1” into HDFS as follows:

```
[cloudera@quickstart ~]$ hdfs dfs -put file1 /user/cloudera
```

Step-15: Again come back to the terminal where **scala>** prompt is running and give the following command to load the text data into “Data Frame” storage object:

```
scala> val df_text = sqlc.read.format("text").load
("hdfs://quickstart.cloudera:8020/user/cloudera/file1
")
```

Output: 18/10/15 05:21:40 INFO TextRelation: Listing hdfs://quickstart.cloudera:8020/user/cloudera/file1 on driver

```
df_text: org.apache.spark.sql.DataFrame = [value:
string]
```

(or)

scala> val df_text = sqlc.read.text("hdfs://quickstart.cloudera:8020/user/cloudera/file1")

Output: 18/10/15 05:23:50 INFO TextRelation: Listing hdfs://quickstart.cloudera:8020/user/cloudera/file1 on driver

df_text: org.apache.spark.sql.DataFrame = [value: string]

Step-16: Display “df_text” storage object as follows:

scala> df_text.show

Note: process goes on…

Output:

```
+--------------------+
|               value|
+--------------------+
|Spark is a powerf...|
|     Structured Data|
|Semi-Structured Data|
|and Un-Structured...|
+--------------------+
```

To load CSV (Comma Separated Values) data into "Data Frame" object:

(((Note: To remove the existed "sparksqlproject" directory in a single step first come out of scala> prompt, SBT prompt, and come out of "sparksqlproject directory and then give the following command:

[cloudera@quickstart ~]$ rm -rf sparksqlproject **)))**

Step-1: Create "sparksqlProject" directory as follows:

[cloudera@quickstart ~]$ mkdir sparksqlproject

Step-2: Go to "sparksqlproject" directory as follows:

[cloudera@quickstart ~]$ cd sparksqlproject

[cloudera@quickstart sparksqlproject]$

Step-3: Start "SBT (Simple Build Tool)" as follows:

[cloudera@quickstart sparksqlproject]$ sbt

Step-4: add “spark core library” as follows:

> set libraryDependencies += "org.apache.spark" % "spark-core_2.10" % "1.6.0"

Step-5: Add “spark sql library” as follows:

> set libraryDependencies += "org.apache.spark" % "spark-sql_2.10" % "1.6.0"

Step-6: Add “spark **csv (comma separated values)**” library as follows:

> set libraryDependencies += "com.databricks" %% "spark-csv" % "1.5.0"

Step-7: give “session save” command as follows:

> session save

Step-8: give “reload” command as follows:

> reload

Step-9: give “console” command to start **scala>** prompt:

> console

Step-10: import the following package:

scala> import org.apache.spark._

Output: import org.apache.spark._

Step-11: Create “conf” object as follows:

scala> val conf = new SparkConf().setAppName("SparkSQLApplication").setMaster("local[2]")

Output: conf: org.apache.spark.SparkConf = org.apache.spark.SparkConf@305af60

Step-12: Create “Spark Context” object as follows:

scala> val sc = new SparkContext(conf)

Note: Process goes on…

Step-13: import the following package:

scala> import org.apache.spark.sql._

Output: import org.apache.spark.sql._

Step-14: Create “SQL Context” object as follows:

scala> val sqlc = new SQLContext(sc)

Output: sqlc: org.apache.spark.sql.SQLContext = org.apache.spark.sql.SQLContext@551a8dac

Step-15: Open another terminal and create **csv** (comma separated values) as follows:

[cloudera@quickstart ~]$ cat > employees

1001,Raju,IT,70000

1002,Ravi,IT,80000

1003,Rani,HR,75000

1004,Roja,HR,75000

Note: press **ctrl d** to save and come out of **cat** command.

[cloudera@quickstart ~]$

Note: come out of this terminal by giving **exit** command

Step-16: Come back to the terminal where **scala>** prompt is running and load this **csv** data into “Data Frame” storage object as follows:

```
scala> val df_csv = sqlc.read.format("csv").load("file:///home/cloudera/employees")
```

Note: process goes on….

Step-17: Display “df_csv” object as follows:

```
scala> df_csv.show
```

Output:

```
+----+----+---+-----+
|  C0|  C1| C2|   C3|
+----+----+---+-----+
|1001|Raju| IT|70000|
|1002|Ravi| IT|80000|
|1003|Rani| HR|75000|
|1004|Roja| HR|75000|
+----+----+---+-----+
```

To load CSV (Comma Separated Values) data into "Data Frame" object: (with column headings)

(((Note: To remove the existed "sparksqlproject" directory in a single step first come out of scala> prompt, SBT prompt, and come out of "sparksqlproject directory and then give the following command:

[cloudera@quickstart ~]$ rm -rf sparksqlproject **)))**

Step-1: Create "sparksqlProject" directory as follows:

[cloudera@quickstart ~]$ mkdir sparksqlproject

Step-2: Go to "sparksqlproject" directory as follows:

[cloudera@quickstart ~]$ cd sparksqlproject

[cloudera@quickstart sparksqlproject]$

Step-3: Start "SBT (Simple Build Tool)" as follows:

[cloudera@quickstart sparksqlproject]$ sbt

Step-4: add “spark core library” as follows:

> set libraryDependencies += "org.apache.spark" % "spark-core_2.10" % "1.6.0"

Step-5: Add “spark sql library” as follows:

> set libraryDependencies += "org.apache.spark" % "spark-sql_2.10" % "1.6.0"

Step-6: Add “spark **csv (comma separated values)**” library as follows:

> set libraryDependencies += "com.databricks" %% "spark-csv" % "1.5.0"

Step-7: give “session save” command as follows:

> session save

Step-8: give “reload” command as follows:

> reload

Step-9: give “console” command to start **scala>** prompt:

> console

Step-10: import the following package:

scala> import org.apache.spark._

Output: import org.apache.spark._

Step-11: Create “conf” object as follows:

scala> val conf = new SparkConf().setAppName("SparkSQLApplication").setMaster("local[2]")

Output: conf: org.apache.spark.SparkConf = org.apache.spark.SparkConf@305af60

Step-12: Create “Spark Context” object as follows:

scala> val sc = new SparkContext(conf)

Note: Process goes on…

Step-13: import the following package:

scala> import org.apache.spark.sql._

Output: import org.apache.spark.sql._

Step-14: Create “SQL Context” object as follows:

scala> val sqlc = new SQLContext(sc)

Output: sqlc: org.apache.spark.sql.SQLContext = org.apache.spark.sql.SQLContext@551a8dac

Step-15: Open another terminal and create **csv** (comma separated values) as follows:

[cloudera@quickstart ~]$ cat > employees

ID,NAME,DEPT,SALARY

1001,Raju,IT,70000

1002,Ravi,IT,80000

1003,Rani,HR,75000

1004,Roja,HR,70000

Note: press **ctrl d** to save and come out of **cat** command.

[cloudera@quickstart ~]$

Note: come out of this terminal by giving **exit** command

Step-16: Come back to the terminal where **scala>** prompt is running and load this **csv** data into “Data Frame” storage object as follows:

```
scala> val df_csv = sqlc.read.format("csv").option("header","true").load("file:///home/cloudera/employees")
```

Note: process goes on….

Step-17: Display “df_csv” object as follows:

```
scala> df_csv.show
```

Output:

```
+----+----+----+------+
|  ID|NAME|DEPT|SALARY|
+----+----+----+------+
|1001|Raju|  IT| 70000|
|1002|Ravi|  IT| 80000|
|1003|Rani|  HR| 75000|
|1004|Roja|  HR| 70000|
+----+----+----+------+
```

→ To display the schema of “Data Frame” then give the following command:

scala> df_csv.printSchema

Output:

```
root
 |-- ID: string (nullable = true)
 |-- NAME: string (nullable = true)
 |-- DEPT: string (nullable = true)
 |-- SALARY: string (nullable = true)
```

Note: By default all values are taken as **string** data type.

Note: If the data types should be according to the data then use **option("inferSchema","true")** as follows:

```
scala> val df_csv =
sqlc.read.format("csv").option("header","true").optio
n("inferSchema","true").load("file:///home/cloudera/
employees")
```

→ **Now, see the schema:**

```
scala> df_csv.printSchema
```

Output:

```
root
 |-- ID: integer (nullable = true)
 |-- NAME: string (nullable = true)
 |-- DEPT: string (nullable = true)
 |-- SALARY: integer (nullable = true)
```

Note: If you observe **ID, SALARY** columns are **string** type previously but now they are integers.

For example, CSV data is as follows:

1001,Raju,IT,70000

1002,Ravi,IT,80000

1003,Rani,HR,75000

1004,Roja,HR,70000

Note: Create the above CSV data in LFS as follows:

Now, we want to give the column headings directly in data frame then do the following steps:

scala> val sqlc = new SQLContext(sc) → up to this step every this same as above. (i.e. up to 14th step), **note that if you have already done up to 14th step then directly do the following 15th step.**

[cloudera@quickstart ~]$ cat > employees
1001,Raju,IT,70000
1002,Ravi,IT,80000
1003,Rani,HR,75000
1004,Roja,HR,70000
Press **Ctrl d** to save and come out of **cat** command.
Now, we want to give the column headings directly in data frame then do the following steps:
scala> val sqlc = new SQLContext(sc) → up to this step every this same as above. (i.e. up to 14th step), **note that if you have already done up to 14th step then directly do the following 15th step.**

(((Skip this slide. This is Extra Reference only:

→If "sparksqlproject" directory is deleted then do all 14 steps.

→If the system is restarted then do the following steps:

```
[cloudera@quickstart ~]$ cd sparksqlproject
[cloudera@quickstart sparksqlproject]$ sbt
> console
scala> import org.apache.spark._
scala> val conf = new
SparkConf().setAppName("SparkSQLApplication").setMaster("local[2]")
scala> val sc = new SparkContext(conf)
scala> import org.apache.spark.sql._
scala> val sqlc = new SQLContext(sc)
```

))))

Step-15: import the following package

```
scala> import org.apache.spark.sql.types._
import org.apache.spark.sql.types._
```

Step-16: Set the Field names and their data types

```
scala> val fields =
StructType(StructField("ID",IntegerType) ::
StructField("Name",StringType) ::
StructField("DeptName",StringType) ::
StructField("Salary",IntegerType) :: Nil)
```

Output:

```
fields: org.apache.spark.sql.types.StructType =
StructType(StructField(ID,IntegerType,true),
StructField(Name,StringType,true),
StructField(DeptName,StringType,true),
StructField(Salary,IntegerType,true))
```

Step-17: Create a data frame as follows

```
scala> val df_csv =
sqlc.read.format("csv").schema(fields).load("file:///ho
me/cloudera/employees")
```

Output: df_csv: org.apache.spark.sql.DataFrame =
[ID: int, Name: string, DeptName: string, Salary: int]

Step-18: Print Schema of “df_csv”:

```
scala> df_csv.printSchema
```

Output:

```
root
 |-- ID: integer (nullable = true)
 |-- Name: string (nullable = true)
 |-- DeptName: string (nullable = true)
 |-- Salary: integer (nullable = true)
```

Step-19: Display the data of “df_csv”

```
scala> df_csv.show
```

Output:

```
+----+-------+--------+------+
|  ID|   Name|DeptName|Salary|
+----+-------+--------+------+
|1001|   Raju|      IT| 70000|
|1002|   Ravi|      IT| 80000|
|1003|   Rani|    HR| 75000|
|1004|   Roja|    HR| 70000|
+----+-------+--------+------+
```

(((Skip this slide. This slide is for Extra Knowledge only.

How to see what libraries are existed in "sparksqlproject" directory?

Answer:

[cloudera@quickstart ~]$ cd sparksqlproject

[cloudera@quickstart sparksqlproject]$ ls

Output: build.sbt project target

Note: build.sbt is a file, project and target are the directories. These are created automatically when we created "sparksqlproject" directory. Note that "sparksqlproject" directory is our directory i.e. user-defined directory.

(((Skip this slide: This slide is continuation to the above slide.

[cloudera@quickstart sparksqlproject]$ gedit build.sbt

Note: The following libraries will be displayed

libraryDependencies += "org.apache.spark" % "spark-core_2.10" % "1.6.0"

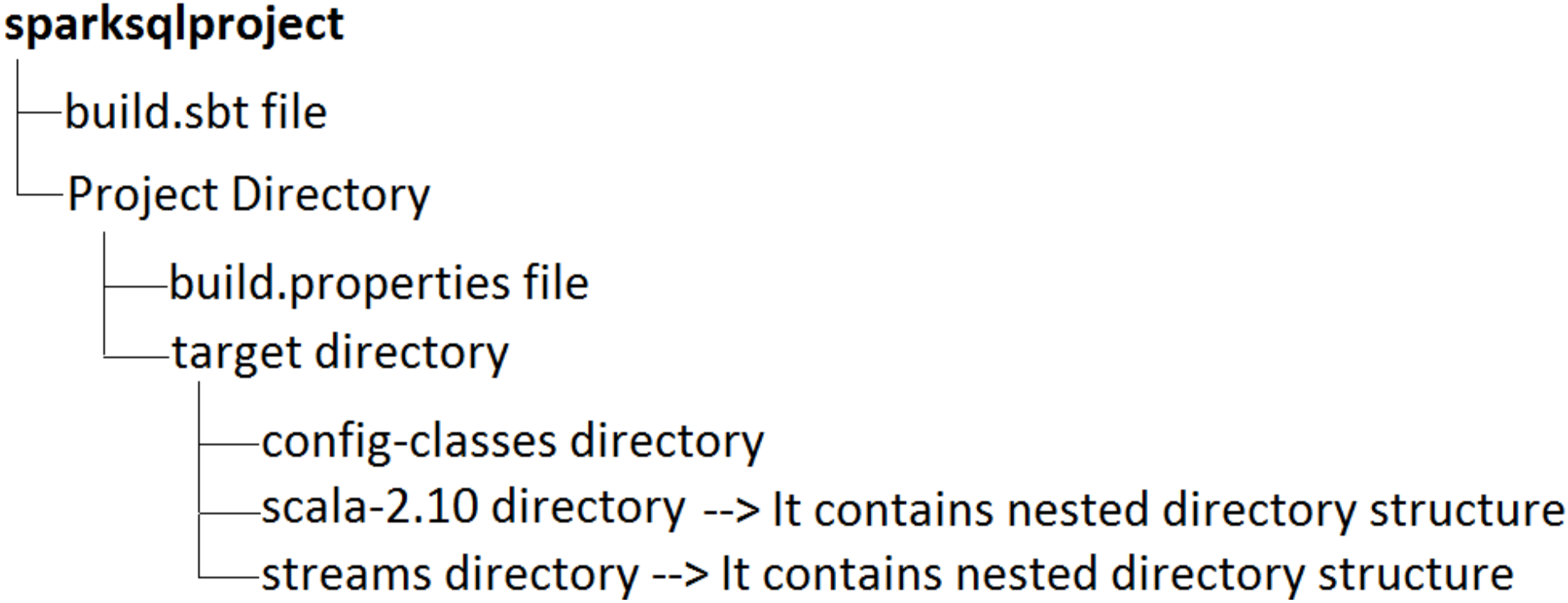
libraryDependencies += "org.apache.spark" % "spark-sql_2.10" % "1.6.0"

libraryDependencies += "com.databricks" %% "spark-csv" % "1.5.0"

(((Skip this slide: This slide is continuation to the above slide.

What is the complete directory structure of "sparksqlproject" directory?

Answer:



(((Skip this slide. It is continuation to the above slide.
In the above directory structure "build.properties" file contains what?

Answer: sbt.version=0.13.16

)))

To load JSON (Java Script Object Notation) data into "Data Frame" object

Note: If already "sparksqlproject" directory is existed then do the following steps:

(((

Note: "CSV(Comma Separated Values)" library is not required for JSON data but if already "CSV" library is existed then it does not create any problem. So, we need not remove it.

)))

```
[cloudera@quickstart ~]$ cd sparksqlproject
[cloudera@quickstart sparksqlproject]$ sbt
> console
scala> import org.apache.spark._
scala> val conf = new
SparkConf().setAppName("SparkSQLApplication").set
Master("local[2]")
scala> val sc = new SparkContext(conf)
scala> import org.apache.spark.sql._
scala> val sqlc = new SQLContext(sc)
```

Next step: Create JSON data as follows:

Open another terminal:

[cloudera@quickstart ~]$ cat > jsondata

{"ID":1001,"NAME":"Raju", "SALARY":70000}

{"ID":1002,"NAME":"Ravi", "SALARY":80000}

Note: Press **ctrl d** to save and come out of **cat** command.

Come back to scala> prompt window and give the following command:

scala> val df_json = sqlc.read.format("json").load("file:///home/cloudera/jsondata")

Note: Process goes on….

scala> df_json.show

Output:

```
+----+----+----+------+
ID|NAME|SALARY|
+----+----+----+------+
1001|Raju| 70000|
1002|Ravi| 80000|
+----+----+----+------+
```

Now, print schema of "df_json":

scala> df_json.printSchema

Output:

root

|-- ID: long (nullable = true)

|-- NAME: string (nullable = true)

|-- SALARY: long (nullable = true)

Note: By default **ID** column is taken as **long** type because it contains **integer** data.

How to put common lines of code in a file and execute that file?

Answer:

Once **Scala>** prompt is started then the following are the **common lines** to start the spark application:

```
scala> import org.apache.spark._
scala> val conf = new
SparkConf().setAppName("SparkSQLApplication").set
Master("local[2]")
scala> val sc = new SparkContext(conf)
scala> import org.apache.spark.sql._
scala> val sqlc = new SQLContext(sc)
```

So, the above lines of code can be placed in a file and execute that file as follows:

Do the following steps:

```
[cloudera@quickstart ~]$ cd sparksqlproject
[cloudera@quickstart sparksqlproject]$ sbt
> console
scala>
```

Open another terminal and create sparkstart file:

```
[cloudera@quickstart ~]$ cat > sparkstart
import org.apache.spark._
val conf = new
SparkConf().setAppName("SparkSQLApplication").set
Master("local[2]")
val sc = new SparkContext(conf)
import org.apache.spark.sql._
val sqlc = new SQLContext(sc)
```

Press **ctrl d** to save and come out of **cat** command.

Come back to scala> prompt and load "sparkstart" file as follows:

scala> :load /home/cloudera/sparkstart

Execute the remaining steps as follows:

scala> val df_json = sqlc.read.format("json").load("file:///home/cloudera/jsondata")

scala> df_json.show

Output:

```
+----+----+----+------+
ID|NAME|SALARY|
+----+----+----+------+
1001|Raju| 70000|
1002|Ravi| 80000|
+----+----+----+------+
```

Spark SQL provides two languages to work with data:

1) Domain Specific Language: which can be used directly on data frames and data sets

2) Structured Query Language (SQL): which can be used only on tables. For this data frame or data set must be converted into table.

Note: We can create either temporary or permanent table.

Note: We can create permanent tables in Hive Context only. So, we can create temporary tables in sql context

Steps to use "Domain Specific Language" and "Structured Query Language":

Step-1: Go to "sparksqlproject" directory:

[cloudera@quickstart ~]$ cd sparksqlproject

Step-2: Start SBT

[cloudera@quickstart sparksqlproject]$ sbt

Step-3: Start "Console" i.e. Scala interpreter or scala> prompt

> console

Step-4: Import “spark” package

```
scala> import org.apache.spark._
```

Step-5: Create Spark Configuration with spark application name and with Master

```
scala> val conf = new SparkConf().setAppName("SparkSQLApplication").setMaster("local[2]")
```

Step-6: Create “Spark Context”

```
scala> val sc = new SparkContext(conf)
```

Step-7: Import “SQL” package:

```
scala> import org.apache.spark.sql._
```

Step-8: Create “SQL Context”

```
scala> val sqlc = new SQLContext(sc)
```

Step-9: Create “CSV” data by opening another terminal

[cloudera@quickstart ~]$ cat > employees

ID,NAME,AGE,SALARY

1001,Raju,24,70000

1002,Ravi,23,65000

1003,Rani,22,60000

1004,Roja,25,75000

Press **Ctrl d** to save and come out of **cat** command.

Step-10: Come back to “scala>” prompt and load the employees data into “Data Frame”

```
scala> val df_csv = sqlc.read.format("csv").option("header","true").option("inferSchema","true").load("file:///home/cloudera/employees")
```

Now, execute “Domain Specific Language”:

```
scala> df_csv.show
```

(or)

```
scala> df_csv.select("*").show
```

Output:

```
+----+----+---+------+
|  ID|NAME|AGE|SALARY|
+----+----+---+------+
|1001|Raju| 24| 70000|
|1002|Ravi| 23| 65000|
|1003|Rani| 22| 60000|
|1004|Roja| 25| 75000|
+----+----+---+------+
```

Now, execute “Domain Specific Language” by displaying first two rows only:

```
scala> df_csv.show(2)
+----+----+---+------+
|  ID|NAME|AGE|SALARY|
+----+----+---+------+
|1001|Raju| 24| 70000|
|1002|Ravi| 23| 65000|
+----+----+---+------+
```

Now, Create a temporary table for the purpose of executing “Structured Query Language”

```
scala> df_csv.registerTempTable("employees")
```

Now execute “Structured Query Language”

```
scala> sqlc.sql("select * from employees").show
```

Output:

```
+----+----+---+------+
|  ID|NAME|AGE|SALARY|
+----+----+---+------+
|1001|Raju| 24| 70000|
|1002|Ravi| 23| 65000|
|1003|Rani| 22| 60000|
|1004|Roja| 25| 75000|
+----+----+---+------+
```

Now execute “Structured Query Language” by displaying first 2 records only

```
scala> sqlc.sql("select * from employees").show(2)
```

(or)

```
scala> sqlc.sql("select * from employees limit 2").show
```

Output:

```
+----+----+---+------+
|  ID|NAME|AGE|SALARY|
+----+----+---+------+
|1001|Raju| 24| 70000|
|1002|Ravi| 23| 65000|
+----+----+---+------+
```

Now execute “Structured Query Language” by describing the employees table

scala> sqlc.sql("describe employees").show

Output:

col_name	data_type	comment
ID	int	
NAME	string	
AGE	int	
SALARY	int	

Now execute “Domain Specific Language” by describing the “Data Frame”

scala> df_csv.printSchema

Output:

```
root
 |-- ID: integer (nullable = true)
 |-- NAME: string (nullable = true)
 |-- AGE: integer (nullable = true)
 |-- SALARY: integer (nullable = true)
```

Now execute “Structured Query Language” by displaying employees table data with “ID” and “NAME” columns with first two records

```
scala> sqlc.sql("select ID,NAME from employees limit 2").show
```

Output:

```
+----+----+
|  ID|NAME|
+----+----+
|1001|Raju|
|1002|Ravi|
+----+----+
```

Now execute “Domain Specific Language” by displaying employees table data with “ID” and “NAME” columns with first two records

```
scala> df_csv.select("ID","NAME").show(2)
```

Output:

```
+----+----+
|  ID|NAME|
+----+----+
|1001|Raju|
|1002|Ravi|
+----+----+
```

Now execute “Domain Specific Language” by displaying employees table data with “ID” and “NAME” columns with first two records through “col” function

Note: For this import “functions” package as follows:

```
scala> import org.apache.spark.sql.functions._
```

Now execute as follows:

```
scala> df_csv.select(col("ID"),col("NAME")).show(2)
```

Output:

```
+----+----+
|  ID|NAME|
+----+----+
|1001|Raju|
|1002|Ravi|
+----+----+
```

Now execute “Domain Specific Language” by displaying employees table data with “ID” and “NAME” columns with first two records through “expr” function

Note: For this import “functions” package as follows:

scala> import org.apache.spark.sql.functions._

Now execute as follows:

```
scala>
df_csv.select(expr("ID"),expr("NAME")).show(2)
```

Output:

```
+----+----+
|  ID|NAME|
+----+----+
|1001|Raju|
|1002|Ravi|
+----+----+
```

Operations on “Data Frames”

→ Creating a “Data Frame” from another “Data Frame”

Note: already “df_csv” data frame is available previously(see slide no:290). Now, let us create a new data frame from “df_csv”

scala> val df1 = df_csv.select("ID","NAME","AGE","SALARY")

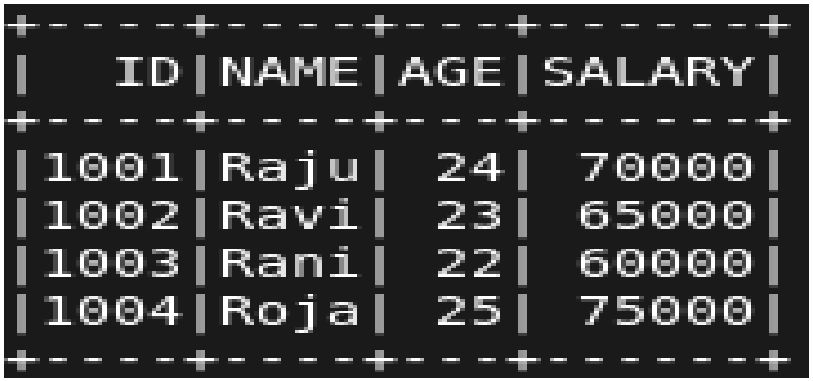
Output:

df1: org.apache.spark.sql.DataFrame = [ID: int, NAME: string, AGE: int, SALARY: int]

→ Now display the data of “df1”

scala> df1.show

Output:



ID	NAME	AGE	SALARY
1001	Raju	24	70000
1002	Ravi	23	65000
1003	Rani	22	60000
1004	Roja	25	75000

Adding a new column to “Data Frame” with values:

→ Let us add “COUNTRY” and its value as “India” to the “Data Frame”

```
scala> val df2 =
df1.withColumn("COUNTRY",lit("India"))
```

Note:See “df1” in the above slide

Now, display the data of “df2”

```
scala> df2.show
```

```
+----+----+---+------+-------+
|  ID|NAME|AGE|SALARY|COUNTRY|
+----+----+---+------+-------+
|1001|Raju| 24| 70000|  India|
|1002|Ravi| 23| 65000|  India|
|1003|Rani| 22| 60000|  India|
|1004|Roja| 25| 75000|  India|
+----+----+---+------+-------+
```

Adding a new column to “Data Frame” with values:

→ Task: Let us add “COUNTRY” and its value as “India” to the “Data Frame”

```
scala> val df2 =
df1.withColumn("COUNTRY",lit("India"))
```

Now, display the data of “df2”

```
scala> df2.show
```

ID	NAME	AGE	SALARY	COUNTRY
1001	Raju	24	70000	India
1002	Ravi	23	65000	India
1003	Rani	22	60000	India
1004	Roja	25	75000	India

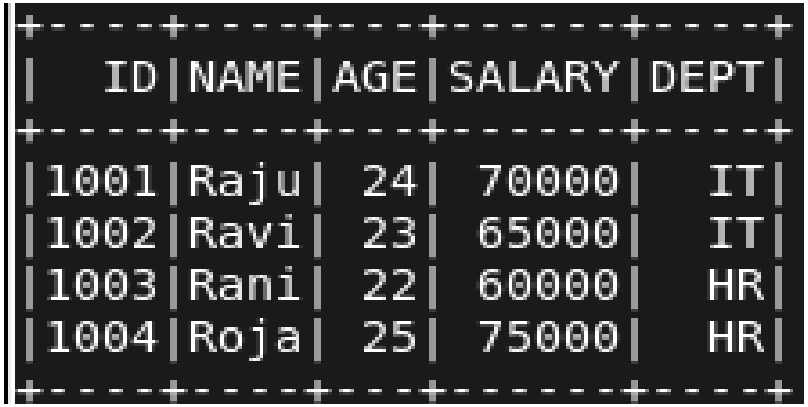
Adding a new column to "Data Frame" with values and with conditions:

→ **Task:** Let us add "DEPT" column where employee IDs less than or equal to 1002 then add the value as "IT" department. If the employee IDs are above 1002 then add "HR" department

scala> val df3 = df1.withColumn("DEPT", when(col("ID") <= 1002, ("IT")). otherwise("HR"))

Display “df3”

scala> df3.show



ID	NAME	AGE	SALARY	DEPT
1001	Raju	24	70000	IT
1002	Ravi	23	65000	IT
1003	Rani	22	60000	HR
1004	Roja	25	75000	HR

Another Example: Adding a new column to "Data Frame" with values and with conditions:

→ **Task:** Add "DESIGNATION" column. If AGE is less than or equal to 25 then DESIGNATION is "Hadoop Developer". If AGE is between 26 and 35 then DESIGNATION is "Junior Data Scientist". If the AGE is above 35 (i.e from 36) then DESIGNATION is "Senior Data Scientist.

Note: The following steps are common

```
[cloudera@quickstart ~]$ cd sparksqlproject
[cloudera@quickstart sparksqlproject]$ sbt
> console
scala>import org.apache.spark._
scala>val conf = new
SparkConf().setAppName("SparkSQLApplication").setMaster("
local[2]")
scala>val sc = new SparkContext(conf)
scala>import org.apache.spark.sql._
scala>val sqlc = new SQLContext(sc)
```

Open another terminal and create the following employees data:

[cloudera@quickstart ~]$ cat > employees

ID,NAME,AGE,SALARY

1001,Raju,24,70000

1002,Ravi,23,65000

1003,Rani,22,60000

1004,Roja,25,75000

1005,Ramu,26,76000

1006,Roopa,35,80000

1007,Ramya,36,85000

Press **ctrl d** to save and come out of **cat** command.

Come back to scala> prompt and do the following steps:

scala>val df_csv = sqlc.read.format("csv").option("header","true").option("inferSchema","true").load("file:///home/cloudera/employees")

scala>val df1 = df_csv.select("ID","NAME","AGE","SALARY")

scala>import org.apache.spark.sql.functions._

Next command:

```
scala>val df4 = df1.withColumn("DESIGNATION", when(col("AGE")<=25, ("Hadoop Developer")).when(col("AGE")>25 and col("AGE")<=35, ("Junior Data Scientist")).otherwise("Senior Data Scientist"))
```

Display “df4”

```
scala>df4.show
```

Another Example:

Task: If AGE is less than or equal to 25 then increment the SALARY for Rs.2000. If AGE is between 26 and 35 the increment the SALARY for Rs.4000. If the AGE is above 35 (i.e from 36) increment the SALARY for Rs.6000

scala> val df5 = df1.withColumn("SALARY", when(col("AGE")<=25, col("SALARY")+2000).when(col("AGE")>25 and col("AGE")<=35, col("SALARY")+4000).otherwise(col("SALARY")+6000))

scala>df5.show

Renaming Column in “Data Frame”:

Task: Rename “ID” column as “IDENTITY NUMBER”

```
scala> val df6 = df1.withColumnRenamed("ID","IDENTITY NUMBER")

scala>df6.show
```

Changing the data type of column in "Data Frame"

Task: Change Integer "SALARY" column to Float data type

First print the schema of "df6"

```
scala> df6.printSchema
root
 |-- IDENTITY NUMBER: integer (nullable = true)
 |-- NAME: string (nullable = true)
 |-- AGE: integer (nullable = true)
 |-- SALARY: integer (nullable = true)
```

Now, change Integer "SALARY" column to Float type.

```
scala> val df7 =
df1.withColumn("SALARY",col("SALARY").cast("Float")
)
```

Now, print schema of “df7”

```
scala> df7.printSchema
root
 |-- ID: integer (nullable = true)
 |-- NAME: string (nullable = true)
 |-- AGE: integer (nullable = true)
 |-- SALARY: float (nullable = true)
```

Creation of Data Frames from collections:

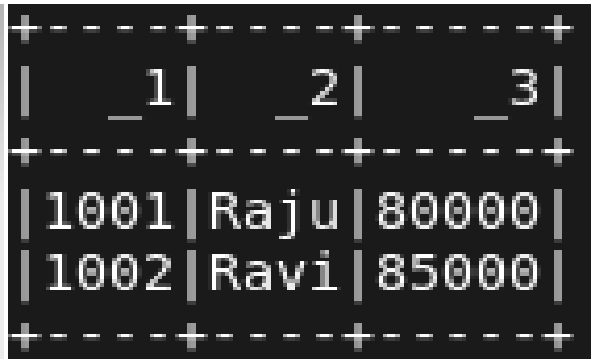
→ Programmer can create the data frames from the collections like Arrays,Lists....

→ The following steps are common:

```
[cloudera@quickstart ~]$ cd sparksqlproject
[cloudera@quickstart sparksqlproject]$ sbt
> console
scala>import org.apache.spark._
scala>val conf = new
SparkConf().setAppName("SparkSQLApplication").set
Master("local[2]")
scala>val sc = new SparkContext(conf)
scala>import org.apache.spark.sql._
scala>val sqlc = new SQLContext(sc)
```

scala> val df1 = sqlc.createDataFrame(Array((1001,"Raju",80000.00),(1002,"Ravi",85000.00)))

scala> df1.show



_1	_2	_3
1001	Raju	80000
1002	Ravi	85000

Note: Here Column headings are _1 _2 _3. But we want to give first column as “ID” and second column as “NAME” and third column as “SALARY”. Then do the following steps:

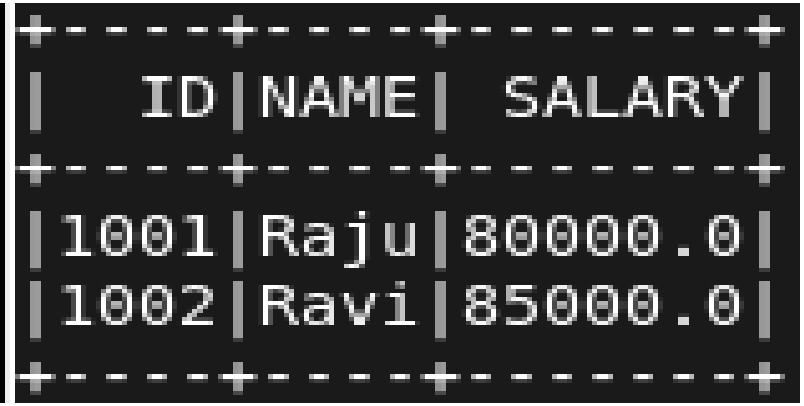
scala> case class employee(ID:Int,NAME:String,SALARY:Double)

Output:

defined class employee

scala> val df1 = sqlc.createDataFrame(Array(employee(1001,"Raju",80000.00),employee(1002,"Ravi",85000.00)))

scala> df1.show

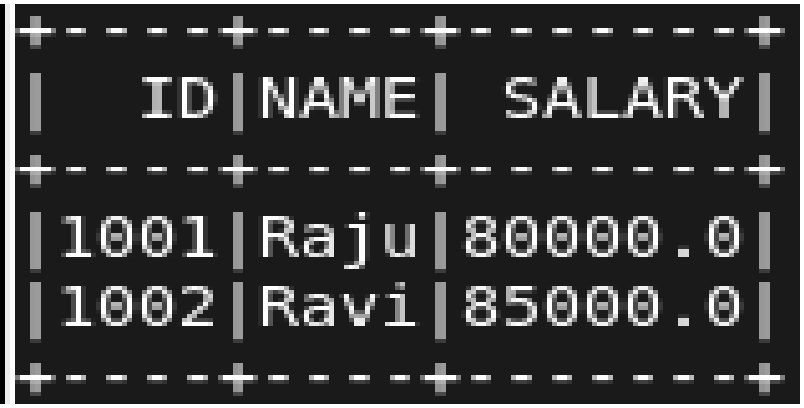


ID	NAME	SALARY
1001	Raju	80000.0
1002	Ravi	85000.0

List Collection:

scala> val df1 = sqlc.createDataFrame(List(employee(1001,"Raju",80000.00),employee(1002,"Ravi",85000.00)))

scala> df1.show



```
+----+----+--------+
|  ID|NAME| SALARY|
+----+----+--------+
|1001|Raju|80000.0|
|1002|Ravi|85000.0|
+----+----+--------+
```

Creating Permanent Table in Hive warehouse through Spark SQL Application (OR) Creation of hive context to create permanent tables:

Open a terminal and start spark application:

[cloudera@quickstart ~]$ spark-shell

Import "hive" package:

scala> import org.apache.spark.sql.hive._

Creating Hive context:

scala> val hivec = new HiveContext(sc)

Open another terminal and create JSON data:

[cloudera@quickstart ~]$ cat > jsondata

{"ID":1001,"NAME":"Raju","SALARY":70000}

{"ID":1002,"NAME":"Ravi","SALARY":80000}

{"ID":1003,"NAME":"Rani","SALARY":75000}

Press **Ctrl d** to save and come out of **cat** command.

Come back to scala> prompt and load the data into a data frame:

scala> val df = hivec.read.format("json").load("file:///home/cloudera/jsondata")

Display “df” data frame:

```
scala>df.show
+----+----+------+
|  ID|NAME|SALARY|
+----+----+------+
|1001|Raju| 70000|
|1002|Ravi| 80000|
|1003|Rani| 75000|
+----+----+------+
```

Create "dftable" under "default" database of Hive Warehouse:

scala> df.write.saveAsTable("default.dftable")

Now, "dftable" is a permanent table which is created under "default" database of hive warehouse

Note: If "dftable" already exists exception comes then give another name to the table and execute again.

Now, check whether "dftable" created under hive warehouse or not:

So, open browser --> Hadoop Menu --> HDFS Namenode Option --> come down of window and click on "legacy UI" link --> Browse the file System link --> user --> hive --> warehouse.
Here, we find "dftable“

Note: So, to save a “Data Frame” as a permanent table in Hive Warehouse then we have to use “hive context”

Spark 1.x	
Spark Context	**Hive Context**
Data Frames and Data Sets can be created	**Data Frames and Data Sets can be created**
Temporary Tables can only be created. Temporary Tables are deleted automatically when session is closed.	**Both Temporary and Permanent Tables can be created. So Permanent tables will be stored in Hive Warehouse**

Note: In Spark 1.x, “Hive” is the default warehouse. In Spark 2.x, “Spark Warehouse” is the default warehouse.

	Spark 1.x	Spark 2.x
Default Warehouse (Without Hive Integration)	No Warehouse	Spark Warehouse
Default Warehouse (With Hive Integration)	Hive Warehouse	Hive Warehouse
Contexts Available	Sql context, Hive Context	Sql context, Hive Context, and Spark Session (sql and hive contexts are depricated)

Spark 1.x			Spark 2.x		
	Sql Context	**Hive Context**	**Sql Context**	**Hive Context**	**Spark Session**
Data Frame, Data Set Creation	Yes	Yes	Yes	Yes	Yes
Creation of Temporary Tables	Yes	Yes	Yes	Yes	Yes
Creation of Permanent Tables	No	Yes	Yes	Yes	yes

Data Sets:

These are storage objects advanced to “Data Frames”.

→ Data Sets allow to perform both RDD Operations and SQL Operations i.e RDD API allows RDD Operations (also called as functional programming) and Data Frame API allows SQL operations.

In Spark 1.x version:

Data Set API contains RDD API (Resilient Distributed Data Set API)

RDD API allows to perform “Functional Programming Operations” i.e. Core programming operations.

Note: API = Application Programming Interface

Definition of API = A set of functions that allow the creation of applications.

In Spark 2.x version:

Data Set API contains “RDD API” and “Data Frame API”. So, “Data Frame API” allows to perform SQL Operations.

Resilient Distributed Data Set (RDD)	Data Frame (DF)	Data Set (DS)
Functional Programming can be done	SQL programming can be done	Functional programming + SQL programming can be done
The data is stored in the form of Strings	The data is stored in the form of rows	The data is stored in the form of objects of class

Resilient Distributed Data Set (RDD)	Data Frame (DF)	Data Set (DS)
RDD uses serialization and “Kryo” Serialization	DF uses Java Serialization and “Kryo” serialization	DS uses “Encoders” and “Tungsten” engine
No “Catalyst Optimizer”	Catalyst Optimizer	Catalyst Optimizer
rdd.toDF rdd.toDS	df.rdd (row RDD is created)	ds.rdd (RDD of objects)

Resilient Distributed Data Set (RDD)	Data Frame (DF)	Data Set (DS)
It does not verify the existence of source file during compilation. So, it checks only syntax.	It verifies file existence during compilation but schema will be evaluated only during execution.	If verifies file existence, and schema evaluation is done at compilation itself. (So, this is best)

Catalyst Optimizer:

→The Spark SQL Engine will submit data frame statements/data set statements to catalyst optimizer. The following plans are prepared by Catalyst Optimizer:

1) Parsed Logical Plan
2) Analyzed Logical Plan
3) Optimized Logical Plan
4) Physical Plan

General Example for "Catalyst Optimizer":

Journey for Ameerpet to shamshabad airport:

1) Find out all possible ways from Ameerpet to Shamshabad airport i.e. **Parsed Logical Plan**

2) Analyze all the ways i.e. **Analyzed Logical Plan**
3) After analysis, select the best way (optimized way) i.e. **Optimized Logical plan**
4) Implement that best way(Optimized Logical Plan) i.e. **Physical Plan**

Apply the above meaning:

→The catalyst optimizer will parse whole stage code of data frame/data set into all possible RDD codes.

→ Each possible RDD code is called as **Logical plan (or) Parsed Logical Plan.**

→ These logical plans are analyzed. So, these are called as **Analyzed Logical plans**

→Among all “Analyzed Logical Plans” the best plan will be selected. Which is called as **Optimized Logical plan**

→ If “Optimized Logical Plan” is put into action then it is called as **Physical Plan.**

Creation of Data Set in spark 1.6.0

→bashrc file contains as follows:

[cloudera@quickstart ~]$ gedit ~/.bashrc

export JAVA_HOME=/usr/java/jdk1.7.0_67-cloudera

export PATH=$JAVA_HOME/bin:$PATH

export SCALA_HOME=/home/cloudera/Downloads/scala-2.10.4

export PATH=$SCALA_HOME/bin:$PATH

Note: Don't Disturb others**.** Save bashrc file and exit from bashrc file and give the source command to update basrc as follows:

[cloudera@quickstart ~]$ source ~/.bashrc

[cloudera@quickstart ~]$ spark-shell

Now, create a Data Set as follows:

scala> val ds = Seq(1, 2, 3).toDS()

Output: ds: org.apache.spark.sql.Dataset[Int] = [value: int]

Now, display “Data Set”:

scala> ds.show

```
+-----+
|value|
+-----+
|    1|
|    2|
|    3|
+-----+
```

Another Example of Data Set:

scala> case class Employee(ID: Int, NAME: String, SALARY: Long)

Output: defined class Employee

scala> val ds = Seq(Employee(1001, "Raju", 80000)).toDS()

Output: ds: org.apache.spark.sql.Dataset[Employee] = [ID: int, NAME: string, SALARY: bigint]

scala> val ds = Seq(Employee(1001, "Raju", 80000), Employee(1002, "Ravi", 85000)).toDS()

Output: ds: org.apache.spark.sql.Dataset[Employee] = [ID: int, NAME: string, SALARY: bigint]

scala> ds.show

```
+----+----+------+
|  ID|NAME|SALARY|
+----+----+------+
|1001|Raju| 80000|
|1002|Ravi| 85000|
+----+----+------+
```

Creating a Data Frame with JSON data in spark 2.3.2 version (spark 2.3.2 is the latest spark version)

Note: Spark Session is available from spark 2.x version onwards.

Download spark 2.3.2 version(Latest version):

Open google on windows operating system→ type: https://spark.apache.org→ Click on "Downloads" → select "2.3.2 (Sep 24 2018)" option from "Choose a Spark release:" list box →

Select "pre-built for Apache Hadoop 2.6" from "Choose a package type:" list box → Click on " spark-2.3.2-bin-hadoop2.6.tgz" link. Now, download starts.

→ After downloading, paste " spark-2.3.2-bin-hadoop2.6.tgz" this downloaded file into "INSTALL" directory of "Cloudera's Home" directory **(Cloudera's Home directory is available on Cloudera Desktop)**

Next go to "INSTALL" directory:

Right Click on "spark-2.3.2-bin-hadoop2.6.tgz" file and select "Extract Here" option then "spark-2.3.2-bin-hadoop2.6" directory will be created.

Next open terminal:

[cloudera@quickstart ~]$ gedit ~/.bashrc

export SPARK_HOME=/home/cloudera/INSTALL/spark-2.3.2-bin-hadoop2.6

export PATH=$SPARK_HOME/bin:$PATH

export JAVA_HOME=/home/cloudera/INSTALL/jdk1.8.0_151

export PATH=$JAVA_HOME/bin:$PATH

Note: Spark 2.3.2 needs JDK 1.8 i.e. Java 1.8 version

Save bashrc file and exit from bashrc file

Next update bashrc file as follows:

[cloudera@quickstart ~]$ source ~/.bashrc

Create JSON data:

[cloudera@quickstart ~]$ cat > jsondata

{"ID":1001,"NAME":"Raju","SALARY":70000}

{"ID":1002,"NAME":"Ravi","SALARY":80000}

{"ID":1003,"NAME":"Rani","SALARY":75000}

press **ctrl d** to save and come out of **cat** command

Start spark application

[cloudera@quickstart ~]$ spark-shell

scala>

Now, Create a Data Frame with JSON data as follows:

```
scala> val df =
spark.read.format("json").load("file:///home/clouder
a/jsondata")
```

Note: In the command spark is "spark session" object

Now, display data frame:

```
scala> df.show
+----+----+------+
|  ID|NAME|SALARY|
+----+----+------+
|1001|Raju| 70000|
|1002|Ravi| 80000|
|1003|Rani| 75000|
+----+----+------+
```

Creation of Data Set:

[cloudera@quickstart ~]$ spark-shell

Creating Employee Class:

scala> case class Employee(ID: Int, NAME: String, SALARY: Long)

Creation of Data Set:

scala> val ds = Seq(Employee(1001,"Raju",70000), Employee(1002,"Ravi",80000)).toDS()

scala> ds.show

```
+----+----+------+
|  ID|NAME|SALARY|
+----+----+------+
|1001|Raju| 70000|
|1002|Ravi| 80000|
+----+----+------+
```

Performing Word Count operation with Data Set (i.e. Performing RDD programming or Functional Programming:

Note: Data Set allows to perform both SQL operations and RDD operations.

Open a Terminal and create the following word count file as follows:

[cloudera@quickstart ~]$ cat wordcount
apple,banana,cherry
banana,apple,cherry
cherry,apple,banana
apple,apple,apple

Press **Ctrl d** to save and come out of **cat** command.

Performing Word Count operation with Data Set (i.e. Performing RDD programming or Functional Programming:

Note: Data Set allows to perform both SQL operations and RDD operations.

Open a Terminal and create the following word count file as follows:

[cloudera@quickstart ~]$ cat wordcount
apple,banana,cherry
banana,apple,cherry
cherry,apple,banana
apple,apple,apple

Press **Ctrl d** to save and come out of **cat** command.

Start Spark Application:

[cloudera@quickstart ~]$ spark-shell

scala> val ds1 = spark.read.format("text").load("file:///home/cloudera/wordcount").as[String]

Output: ds1: org.apache.spark.sql.Dataset[String] = [value: string]

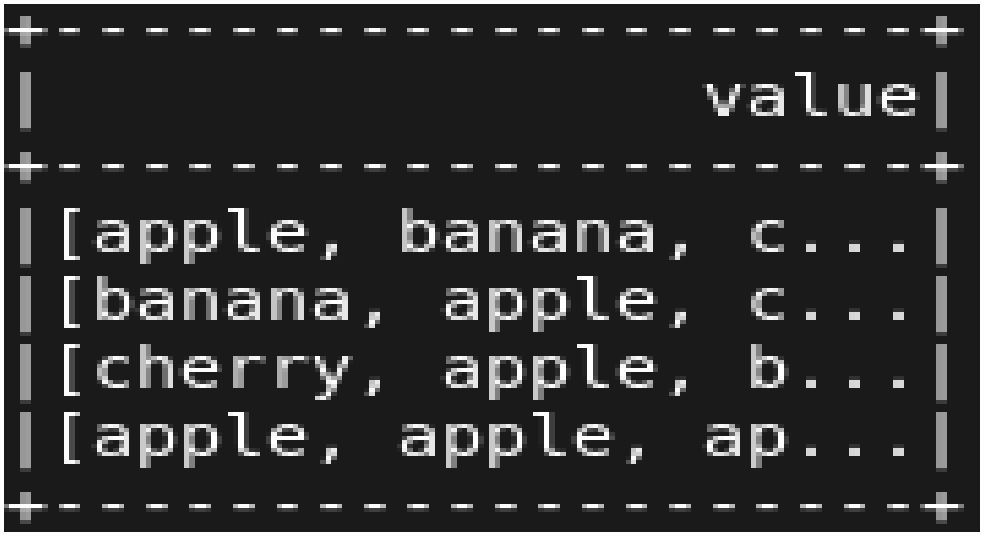
scala> ds1.show

```
+--------------------+
|               value|
+--------------------+
|apple,banana,cherry|
|banana,apple,cherry|
|cherry,apple,banana|
|  apple,apple,apple|
+--------------------+
```

scala> val ds2 = ds1.map(x => x.split(","))

Output: ds2: org.apache.spark.sql.Dataset[Array[String]] = [value: array<string>]

scala> ds2.show



```
+--------------------+
|               value|
+--------------------+
|[apple, banana, c...|
|[banana, apple, c...|
|[cherry, apple, b...|
|[apple, apple, ap...|
+--------------------+
```

scala> val ds3 = ds2.flatMap(x => x)

Output:ds3: org.apache.spark.sql.Dataset[String] = [value: string]

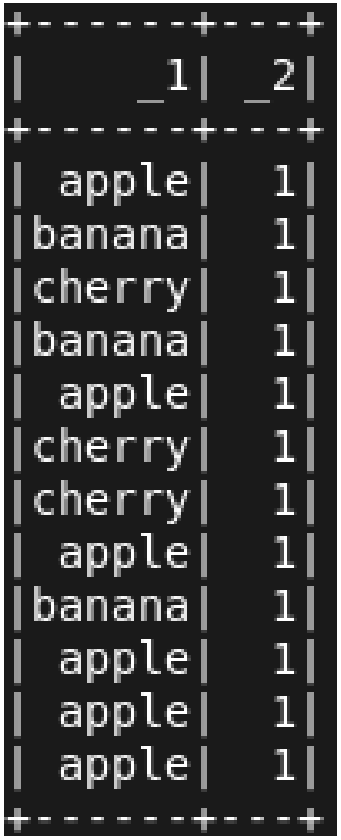
scala> ds3.show

```
+-------+
|  value|
+-------+
|  apple|
| banana|
| cherry|
| banana|
|  apple|
| cherry|
| cherry|
|  apple|
| banana|
|  apple|
|  apple|
|  apple|
+-------+
```

scala> val ds4 = ds3.map(x=>(x,1))

Output: ds4: org.apache.spark.sql.Dataset[(String, Int)] = [_1: string, _2: int]

scala> ds4.show

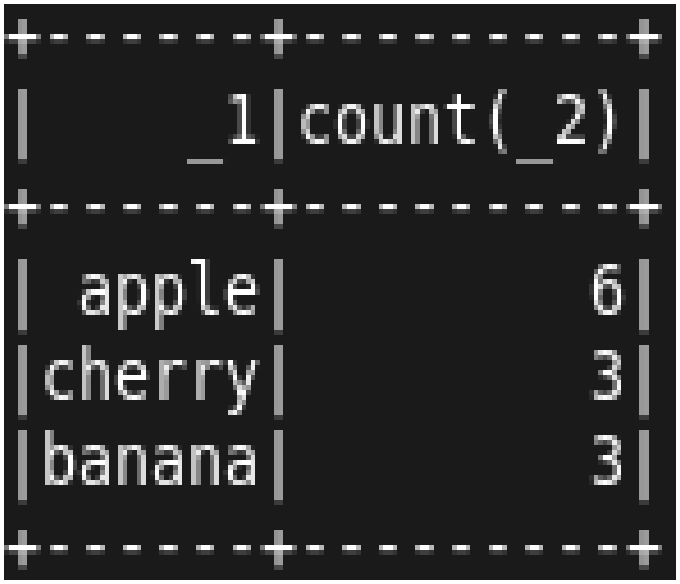


_1	_2
apple	1
banana	1
cherry	1
banana	1
apple	1
cherry	1
cherry	1
apple	1
banana	1
apple	1
apple	1
apple	1

scala> val ds5 = ds4.select("*").groupBy("_1").agg(count("_2"))

Output:ds5: org.apache.spark.sql.DataFrame = [_1: string, count(_2): bigint]

scala> ds5.show



```
+------+---------+
|    _1|count(_2)|
+------+---------+
| apple|        6|
|cherry|        3|
|banana|        3|
+------+---------+
```

Converting An RDD into a Data Set:

First create An RDD:

scala> val r1 = sc.makeRDD(Array((1,2),(3,4)))

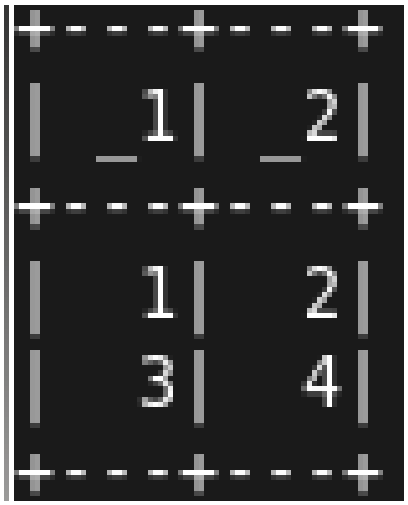
Output: r1: org.apache.spark.rdd.RDD[(Int, Int)] = ParallelCollectionRDD[93] at makeRDD at <console>:24

Convert RDD into Data Set:

scala> val ds = r1.toDS()

Output: ds: org.apache.spark.sql.Dataset[(Int, Int)] = [_1: int, _2: int]

scala> ds.show



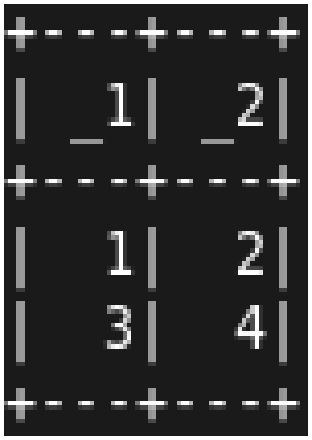
_1	_2
1	2
3	4

Converting RDD into Data Frame

scala> val df = r1.toDF()

Output: df: org.apache.spark.sql.DataFrame = [_1: int, _2: int]

scala> df.show

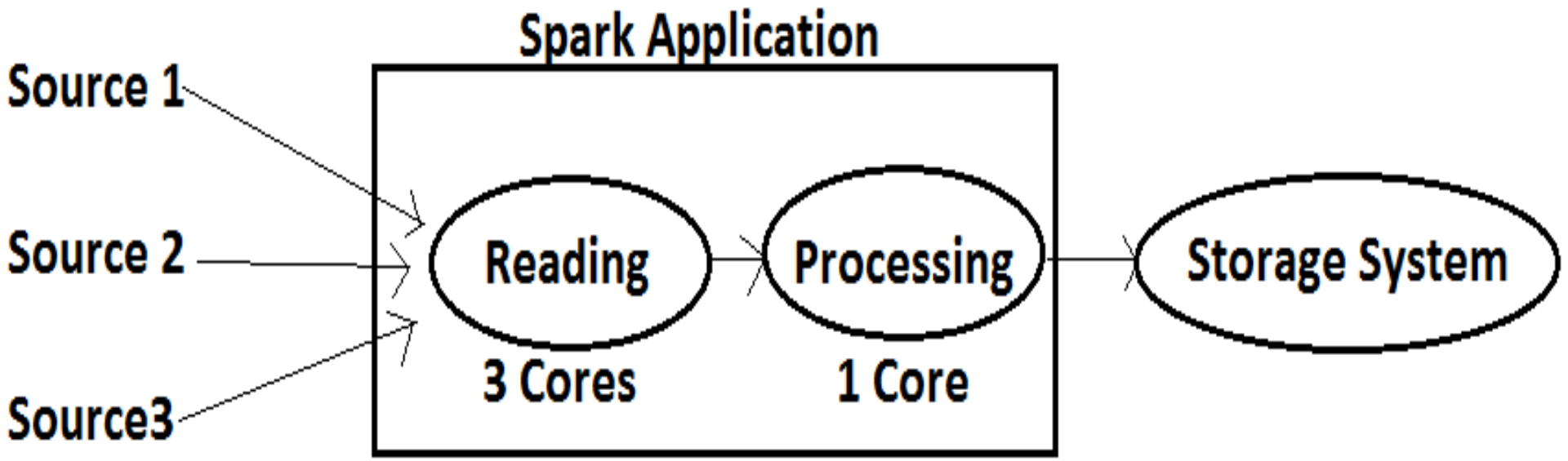


_1	_2
1	2
3	4

SPARK STREAMING

SPARK STREAMING

→ Spark Streaming application is capable to read and process the streaming data.



→ Spark Streaming application stores the data in "DStream" (Discretized Stream)

→We need to create an object to StreamingContext class for developing "spark streaming application" as follows:

```
[cloudera@quickstart ~]$ spark-shell --master local[*]
scala> import org.apache.spark.streaming._
scala> val ssc = new StreamingContext(sc, seconds(20))
```

Note: "ssc" means "spark streaming context"

20 seconds means that the streaming data should be processed for every 20 seconds. (So, 20 seconds gap between reading and processing)

→ If a spark application is receiving the data from one receiver (one Source) then minimum number of cores must be two i.e. one core for receiving the data and another core for processing the data.

→ If there are two receivers then the minimum number of cores must be three i.e. one core for receiver-1, second core for receiver-2, and third core for processing the data.

→ If there are “n” number of receivers are existed then the number of cores must be “n+1”.

→The receivers deposit the data in memory in the form of “Dstreams“ (Discretized Stream)

→Every “DStream“ is converted into sequence of RDDs. (So, “DStream“ means RDD only)

→After certain amount of interval the repository data (deposited data) will get processed.
→Once the process is successful then the data from repository get deleted.
→The interval between data and data processing is called as “Batch interval”
→The receivers create two copies of data in the repository (replication factor is 2)

Check pointing:

→If we don't trust memory then the receivers can write the data to fault tolerance file system like HDFS.

→During the processing of data if any partition is lost then it can be generated from HDFS.

→Once processing is completed then the data get deleted from HDFS. This process is called as **Check pointing**

Micro batch:

→ The batch interval can be mentioned in seconds or minutes.

→ The amount of data which is collected during Batch interval is called as **Micro batch**

Note: 1 second = 1000 milliseconds

1 millisecond = 1000 microseconds

1 microsecond = 1000 nanoseconds

Program1: Word count program (Streaming style, i.e. Words come lively and perform word count lively)

Step-1:

[cloudera@quickstart ~]$ spark-shell --master local[*]

(or) [cloudera@quickstart ~]$ spark-shell

Step-2:

scala> import org.apache.spark.streaming._

Output: import org.apache.spark.streaming._

Step-3:

scala> val ssc=new StreamingContext(sc,Seconds(20))

Output: ssc:

org.apache.spark.streaming.StreamingContext =

org.apache.spark.streaming.StreamingContext@4468fda8

Step-4:

scala> val ds1=ssc.socketTextStream("localhost",12345)

Output: ds1: org.apache.spark.streaming.dstream.ReceiverInputDStream[String] = org.apache.spark.streaming.dstream.SocketInputDStream@624d778e

Step-5:

```
scala> val ds2=ds1.flatMap(x=>x.split("
")).map(x=>(x,1)).reduceByKey(_+_)
```

Output: ds2:

```
org.apache.spark.streaming.dstream.DStream[(String,
Int)] =
org.apache.spark.streaming.dstream.ShuffledDStream
@3506bc8b
```

Step-6:

scala> ds2.print()

Step-7:

scala> ssc.start → starting Spark Streaming application

Step-8:

Open another terminal:

[cloudera@quickstart ~]$ nc -l 12345

After giving the above command start typing the following words:
hadoop
hadoop
hadoop
spark
spark

Now goto the terminal where "ssc.start" is done. Here you see (hadoop,3) (spark,2)
Test like this for as many time as possible.

Step-9:

To stop "12345" terminal press ctrl c

Step-10:

To stop Spark Streaming application, go to the terminal where "ssc.start" is done. Here type ssc.stop(true,false)(note that while you are typing ssc.stop(true,false), these appear in different lines but don't stop typing)Finally type exit i.e. scala> exit

(or)

Simple press **ctrl c** continuously then we will come to cloudera prompt i.e. [cloudera@quickstart ~]$

Question: In Dual core system, can spark application receive the streaming data from two sources?

Answer: No. Because in dual core system, only two cores are available. For receiving streaming data from two sources then two cores are required and third core is required to process the data (So, totally 3 cores are required)

The following steps prove that dual core system can not receive the streaming data from two sources. So, the following steps work only in system which contains minimum 3 cores (two cores for receiving the data and one core for processing the data).

So, if system is having 4 cores then the following steps will be worked:

cloudera@quickstart ~]$ spark-shell --master local[*]

(or) cloudera@quickstart ~]$ spark-shell

scala> import org.apache.spark.streaming._

Output:import org.apache.spark.streaming._

scala> val ssc=new StreamingContext(sc,Seconds(20))

Output: ssc: org.apache.spark.streaming.StreamingContext = org.apache.spark.streaming.StreamingContext@22c239ea

scala> val ds1=ssc.socketTextStream("localhost",12345)

Output: ds1: org.apache.spark.streaming.dstream.ReceiverInputDStream[String] = org.apache.spark.streaming.dstream.SocketInputDStream@3056a7c3

scala> val ds2=ssc.socketTextStream("localhost",123456)

Output: ds2: org.apache.spark.streaming.dstream.ReceiverInputDStream[String] = org.apache.spark.streaming.dstream.SocketInputDStream@50b4adf2

scala> val ds3=ds1.union(ds2)

Output: ds3: org.apache.spark.streaming.dstream.DStream[String] = org.apache.spark.streaming.dstream.UnionDStream@566dd081

scala> val ds4=ds3.flatMap(x=>x.split(" ")).map(x=>(x,1)).reduceByKey(_+_)

Output: ds4: org.apache.spark.streaming.dstream.DStream[(String, Int)] = org.apache.spark.streaming.dstream.ShuffledDStream@59b37acf

scala> ds4.print()

scala> ssc.start

Open another terminal:[cloudera@quickstart ~]$ nc -l 12345

Open one more terminal:[cloudera@quickstart ~]$ nc -l 123456

Come back to : [cloudera@quickstart ~]$ nc -l 12345
Now type:
hadoop
hadoop
hadoop

goto the terminal where "ssc.start" is done. Here you see (hadoop,3)

Now, Come to [cloudera@quickstart ~]$ nc -l 123456
Now type:
spark
spark

goto the terminal where "ssc.start" is done. Here you see (spark,3)
Like this you give words and see the output. (To stop the "12345" & “123456" terminals press ctrl c)
To stop Spark Streaming application: Come to the terminal where "ssc.start" is done. Here type ssc.stop(true,false)

(OR) The following steps also can be done to prove that dual core system can not receive the streaming data from two sources. So, if the system is having minimum 3 cores then only these steps will be executed:

[cloudera@quickstart ~]$ spark-shell

scala> import org.apache.spark.streaming._
Output: import org.apache.spark.streaming._

scala> val ssc=new StreamingContext(sc,Seconds(20))
Output: ssc:
org.apache.spark.streaming.StreamingContext =
org.apache.spark.streaming.StreamingContext@22c2
39ea

scala> val ds1=ssc.socketTextStream("localhost",12345)

Output: ds1: org.apache.spark.streaming.dstream.ReceiverInputDStream[String] = org.apache.spark.streaming.dstream.SocketInputDStream@3056a7c3

scala> val ds2=ds1.flatMap(x=>x.split(" ")).map(x=>(x,1)).reduceByKey(_+_)

Output: ds4: org.apache.spark.streaming.dstream.DStream[(String, Int)] = org.apache.spark.streaming.dstream.ShuffledDStream@59b37acf

scala> ds2.print()

scala> val ds3=ssc.socketTextStream("localhost",123456)

Output: ds2: org.apache.spark.streaming.dstream.ReceiverInputDStream[String] = org.apache.spark.streaming.dstream.SocketInputDStream@50b4adf2

scala> val ds4=ds3.flatMap(x=>x.split(" ")).map(x=>(x,1)).reduceByKey(_+_)

Output: ds4: org.apache.spark.streaming.dstream.DStream[(String, Int)] = org.apache.spark.streaming.dstream.ShuffledDStream@59b37acf

scala> ds4.print()

scala> ssc.start
Open another terminal:[cloudera@quickstart ~]$ nc -l 12345

Open one more terminal:[cloudera@quickstart ~]$ nc -l 123456

Come back to : [cloudera@quickstart ~]$ nc -l 12345
Now type:
 hadoop
hadoop
hadoop

goto the terminal where "ssc.start" is done. Here you see (hadoop,3)
Note: If your system is having dual core(i.e. two cores) then no output, only exceptions come. So, these steps work only in quadratic core (i.e. 4 cores) system.
Now, Come to [cloudera@quickstart ~]$ nc -l 123456
Now type:
spark
spark
goto the terminal where "ssc.start" is done. Here you see (spark,3)
Note: If your system is having dual core(i.e. two cores) then no output, only exceptions come. So, these steps work only in quadratic core (i.e. 4 cores) system.

Like this you give words and see the output. (To stop the "12345" & “123456" terminals press ctrl c)
To stop Spark Streaming application: Come to the terminal where "ssc.start" is done. Here type ssc.stop(true,false)

→ In a cluster mode each receiver will be allocated with one executor.
→ If four cores are there then four executors will be allocated
→ The executors will be executed when application is launched.
→ Local mode means both master and slave runs is same system
→ Collection of Data Nodes is one rack and collection of racks is one Data Centre.
→ If Block is loaded into memory then it is called as partition.

Port numbers and their details:

	YARN	Stand Alone	Mesos	HDFS
Master Process	Resource Manager (RM)	Master	Mesos Master	Name Node, Secondary Name Node
Slave Process	Node Manager	Worker	Mesos Agents	Data Node
Master http port (web UI)	8088	18080 (CDH) CDH = Cloudera Distribution for Hadoop	4040	50070
Master RPC port RPC = Remote Procedure Call	8032	7077	5050	8020

Windowing:

→ By default the streaming application processes only one batch of data for every batch interval (time interval) and if we want to process multiple batches of data for every time interval then it can be implemented by **windowing**.

(((Note: Batch contains the data**)))**

((Note: In dual core system, if window size is two then the same batch of data comes two times (see in output)**))**

→ The number of batches will define the **window size.**

→ If the number of batches are two then the window size is 2.

→ The windowing concept automatically implements persistence of previous batches (i.e. storing the previous batches. Note that batch contains the data)

Example: Window size = 2

Example: Window size = 2, Batch interval/Time interval = 1 minute

Batch Creation	Batch Processing	persistence(Backup)
10 - 10.01: B1	⨯	
10.01 - 10.02: B2	B1	
10.02 - 10.03: B3	B2, B1 (because window size = 2)	B1
10.03 - 10.04: B4	B3, B2	B2
10.04 - 10.05: B5	B4, B3	B3

Note: B = Batch

Example: Window size = 3

Example: Window size = 3, Batch interval/Time interval = 1 minute

Batch Creation	Batch Processing	persistence(Backup)
10 - 10.01: B1	⨯	
10.01 - 10.02: B2	B1	
10.02 - 10.03: B3	B2, B1	B1
10.03 - 10.04: B4	B3, B2, B1	B2, B1
10.04 - 10.05: B5	B4, B3, B2	B3, B2

Note: B = Batch

Implementing “windowing” in dual core system

(To see the number of cores: start menu → type: device manager and press enter key → double click on processors → Here, the number of processors will be displayed**)**

Step-1:

[cloudera@quickstart ~]$ spark-shell --master local[*]

→ Starting spark application in local mode

(or) [cloudera@quickstart ~]$ spark-shell

Step-2:

scala> import org.apache.spark.streaming._

Output: import org.apache.spark.streaming._

Step-3:

scala> val ssc=new StreamingContext(sc,Seconds(20))

Output: ssc:
org.apache.spark.streaming.StreamingContext =
org.apache.spark.streaming.StreamingContext@4468
fda8

Step-4:

scala> val ds1=ssc.socketTextStream("localhost",12345)

Output: ds1: org.apache.spark.streaming.dstream.ReceiverInputDStream[String] = org.apache.spark.streaming.dstream.SocketInputDStream@624d778e

Step-5:

scala> val ds2 = ds1.window(Seconds(40))

Note: Here, window(Seconds(40)) means the window size is two i.e. 20 + 20 So, the first window is processed in 20 seconds and the second window is processed in another 20 seconds.

Similarly, window(Seconds(60)) means the window size is three.

Question: Why 20 + 20 only?

Answer: val ssc=new StreamingContext(sc,Seconds(20)) → Based on this statement “20 + 20” is decided.

Step-6:

scala> ds2.print()

Step-7:

scala> ssc.start --> starting Spark Streaming application

Step-8:

Open another terminal:

[cloudera@quickstart ~]$ nc -l 12345

Note: nc = netcat, -l = localhost, 12345 = port number. It is specified in “val ds1=ssc.socketTextStream("localhost",12345)” statement

After giving the above command start typing the following:

Spark Streaming is a great tool to process the streaming data.

Come back to the terminal where spark streaming application(i.e. “ssc.start” terminal)

Output:

Time: 1542351300000 ms

Spark Streaming is a great tool to process the streaming data.

Time: 1542351320000 ms

Spark Streaming is a great tool to process the streaming data.

Note: why two times? Because of this statement “val ds2 = ds1.window(Seconds(40))”. i.e. same data is displayed for two times with 20 seconds gap.

Converting Streaming Data into Data Frame:

Step1:

[cloudera@quickstart ~]$ spark-shell → Starting spark application in local mode

Step-2:

scala> import org.apache.spark.streaming._

Output: import org.apache.spark.streaming._

Step-3:

scala> val ssc = new StreamingContext(sc,Seconds(20))

Output: ssc: org.apache.spark.streaming.StreamingContext = org.apache.spark.streaming.StreamingContext@20a9f5fb

Step-4: Data Source-1

```
scala> val ds1 =
ssc.socketTextStream("localhost",3456)
```

Output: ds1:

```
org.apache.spark.streaming.dstream.ReceiverInputDS
tream[String] =
org.apache.spark.streaming.dstream.SocketInputDStr
eam@7809b43a
```

Step-5: Converting Streaming data (in the form of RDD) into Data Frame

```
scala> val ds2 = ds1.foreachRDD{x =>
     val df = x.toDF()
     df.show
     }
```

output: ds2: Unit = ()

Step-6:

scala> ssc.start → Starting spark streaming application

Step-7:

Open another terminal:

[cloudera@quickstart ~]$ nc -l 3456

Note: Type the following lines

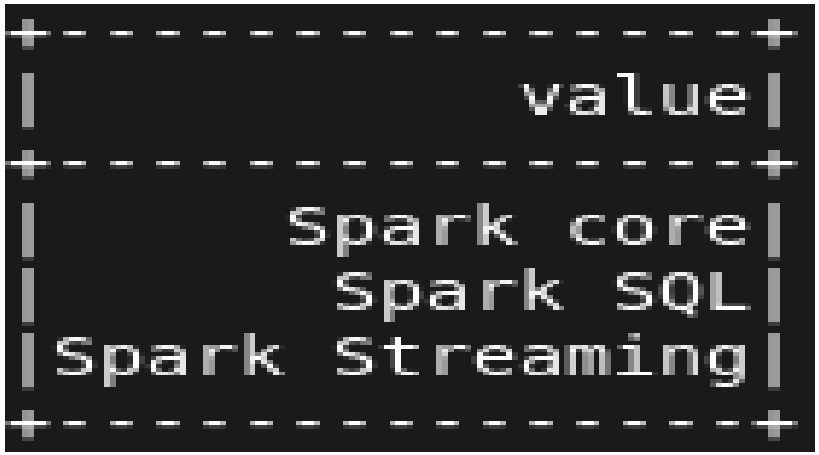
Spark Core

Spark SQL

Spark Streaming

Go to the terminal where spark streaming application is running (i.e. “ssc.start” command is given)

Output:



Note: press **ctrl c** continuously until you see the cloudera prompt i.e. **[cloudera@quickstart ~]$**

Converting Streaming Data into Data Set:

Step1:

[cloudera@quickstart ~]$ spark-shell → Starting spark application in local mode

Step-2:

scala> import org.apache.spark.streaming._

Output: import org.apache.spark.streaming._

Step-3:

scala> val ssc = new StreamingContext(sc,Seconds(20))

Output: ssc: org.apache.spark.streaming.StreamingContext = org.apache.spark.streaming.StreamingContext@20a9f5fb

Step-4: Data Source-1

```
scala> val ds1 =
ssc.socketTextStream("localhost",3456)
```

Output: ds1:

```
org.apache.spark.streaming.dstream.ReceiverInputDS
tream[String] =
org.apache.spark.streaming.dstream.SocketInputDStr
eam@7809b43a
```

Step-5: Converting Streaming data (in the form of RDD) into Data Set

```
scala> val ds2 = ds1.foreachRDD{x =>
     val ds = x.toDS()
     ds.show
     }
```

output: ds2: Unit = ()

Step-6:

scala> ssc.start → Starting spark streaming application

Step-7:

Open another terminal:

[cloudera@quickstart ~]$ nc -l 3456

Note: Type the following lines

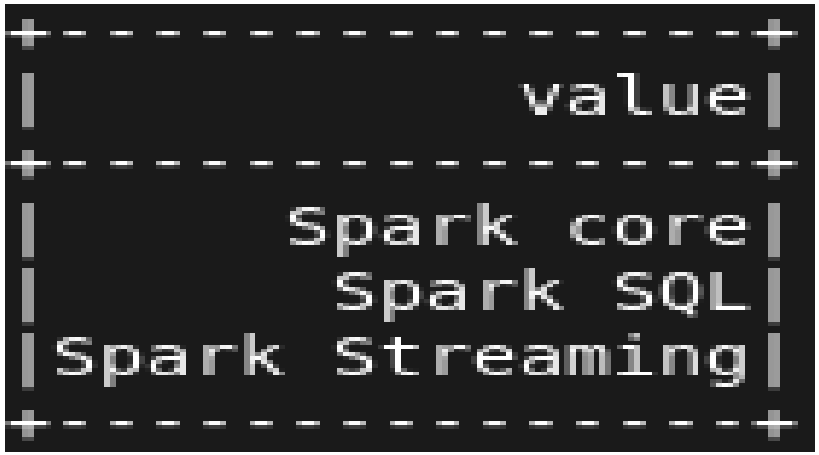
Spark Core

Spark SQL

Spark Streaming

Go to the terminal where spark streaming application is running (i.e. “ssc.start” command is given)

Output:



Note: press **ctrl c** continuously until you see the cloudera prompt i.e. **[cloudera@quickstart ~]$**