SPARK

If you don’t drive your business,
you will be driven out of business

Spark Features

- Spark is written in Scala, and runs on the Java Virtual Machine (JVM)
- Spark can be used from Python, Java, or Scala
- Prerequisite for installing and running spark are:
  - ✓ Jdk 7 or above
  - ✓ Scala 10 or above
  - ✓ Python 2.6 or above
- Spark can be run in 3 different resource management framework
  - ✓ Standalone
  - ✓ Yarn
  - ✓ Mesos

Spark Installation

- Spark Installation doesn't need Hadoop to be mandatorily present. But preferred one in real time is installing Spark on Linux OS on top of existing Hadoop installation
- Spark can be built from source code as well but prefer to install using binary tar balls.

- Download the spark binary tar ball matching our Hadoop version from http://spark.apache.org/downloads.html
- Unpack the downloaded tar ball into **/opt/spark** and provide permissions to this directory.
  - ✓ **sudo mkdir /opt/spark/**
  - ✓ **cp Downloads/spark-*.tgz /opt/spark**
  - ✓ **cd /opt/spark/**
  - ✓ **tar –xzf spark-*.tgz**

- Add spark **bin** folder location to **PATH** environment variable in **.bashrc** file
  - ✓ **gedit ~/.bashrc**
  - ✓ **export SPARK_HOME=/opt/spark/**
  - ✓ **export PATH = $PATH:/opt/spark/bin/:/opt/spark/sbin/**
- Verify installation by starting spark-shell

Spark In QuickStart VM

- But Cloudera QuickStart VM, comes with Spark installed already. Check weather spark daemons are running properly or not in standalone mode.
  - ✓ sudo jps
  - ✓ check master/worker service
  - ✓ sudo service spark-master status/start/restart/stop
  - ✓ sudo service spark-worker status/start/restart/stop
  - ✓ sudo service spark-history-server status/start/restart/stop

- Spark Web UI
  - ✓ Master runs on 18080 port and hostname is **quickstart.cloudera**
  - ✓ Worker runs on 18081 port
  - ✓ History Server runs on 18088
  - ✓ If running in stand alone, spark jobs can be checked on 4040 port

Spark Shells

- Spark comes with shells for Scala, Python and R.
- There is no shell available for Spark with Java.

- Spark Python Shell
  - ✓ pyspark

```
lines = sc.textFile("README.md")
lines.count()
lines.first()
```

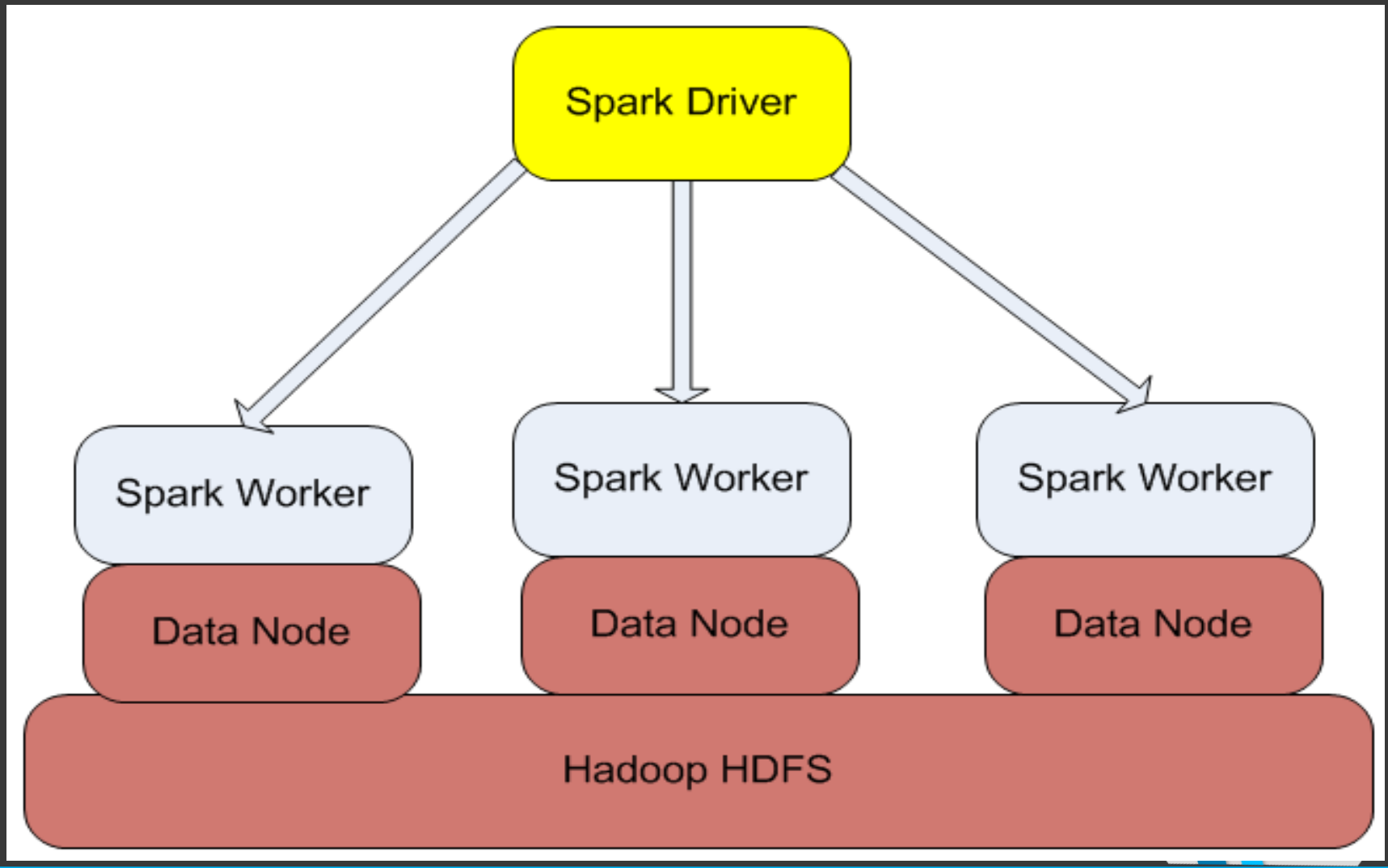
- Spark Scala Shell
  - ✓ spark-shell

```
val lines = sc.textFile("README.md")
lines.count()
lines.first()
```

- Spark SQL Shell
  - ✓ spark-sql
- Spark R Shell
  - ✓ sparkR

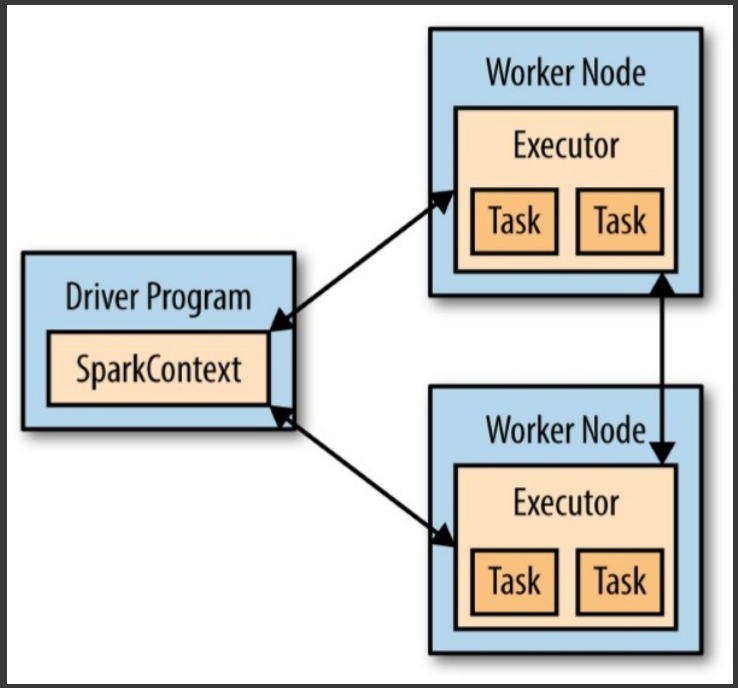
- To exit any shell, press Ctrl-D.

Spark Architecture



How spark works?

- Spark applications will have a driver program, that launches various parallel operations on the cluster
- Driver program contains our application's main function and defines distributed datasets on the cluster, then applies operations to them.
- Driver programs access Spark through a SparkContext object, which represents a connection to a computing cluster.
- In the shell, shell itself is our Driver, and SparkContext is automatically created as sc.
- Then we can run various operations on this RDDs.
- Driver programs typically manage a number of nodes called executors



SparkConf - Configuration for Spark Applications

- First step of any Spark driver application is to create a **SparkContext**
- To create spark context object, we need **SparkConf** object
- The SparkConf stores configuration parameters that our Spark driver application will pass to SparkContext
- Two mandatory settings of any Spark application that have to be defined before this Spark application could be run — **spark.master** and **spark.app.name**

```
import org.apache.spark.SparkConf

val conf = new SparkConf().setAppName("MySparkDriverApp").
setMaster("spark://master:7077").set("spark.executor.memory", "2g")
```

- Start **spark-shell with --conf spark.logConf=true** to log the effective Spark configuration as INFO when SparkContext is started

```
$ spark-shell --conf spark.logConf=true
```

http://spark.apache.org/docs/latest/configuration.html#viewing-spark-properties

SparkConf - Configuration for Spark Applications

```
$ spark-shell --conf spark.logConf=true
...
15/10/19 17:13:49 INFO SparkContext: Running Spark version 1.6.0-
SNAPSHOT
15/10/19 17:13:49 INFO SparkContext: Spark configuration:
spark.app.name=Spark shell
spark.home=/Users/jacek/dev/oss/spark
spark.jars=
spark.logConf=true
spark.master=local[*]
spark.repl.class.uri=http://10.5.10.20:64055
spark.submit.deployMode=client

scala> sc.getConf.getOption("spark.local.dir")
res0: Option[String] = None

scala> sc.getConf.getOption("spark.app.name")
res1: Option[String] = Some(Spark shell)

scala> sc.getConf.get("spark.master")
res2: String = local[*]
```

Setting up Properties

Ways to set up properties for Spark and user programs

- **conf/spark-defaults.conf** - the default
- **--conf or -c** - the command-line option used by spark-shell and spark-submit+
- **SparkConf**

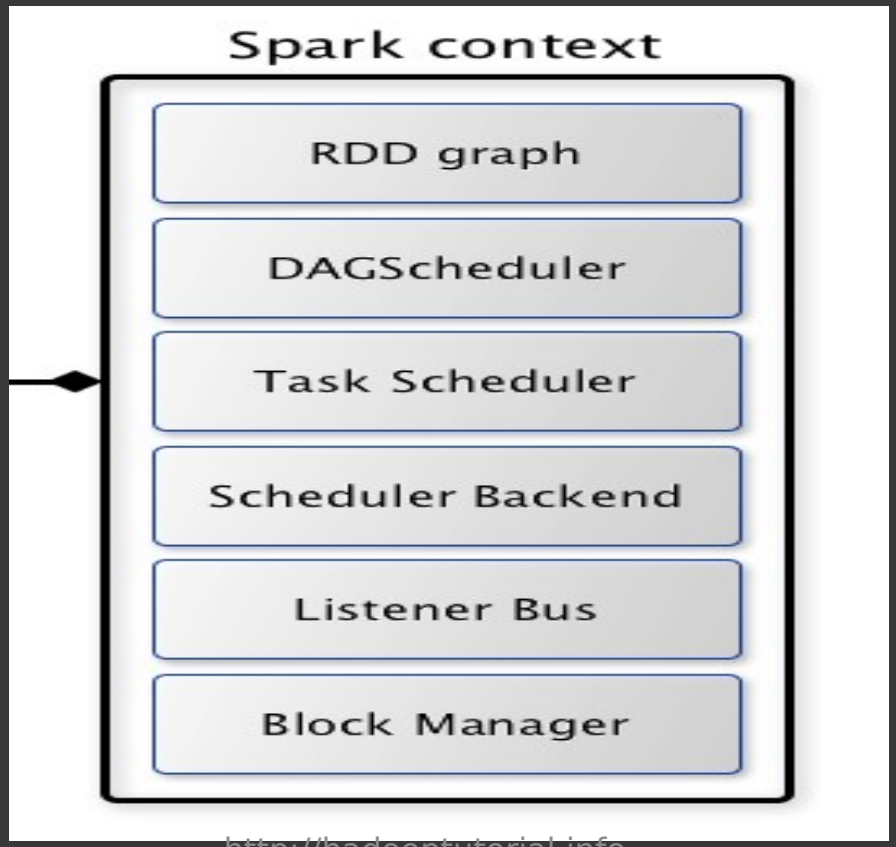
We can use conf.toDebugString or conf.getAll to have the spark.* system properties loaded printed out.

```
scala> val conf= sc.getConf
scala> conf.getAll
res0: Array[(String, String)] = Array((spark.app.name,Spark shell),
(spark.jars,""), (spark.master,local[*]), (spark.submit.deployMode,client))

scala> conf.toDebugString
res1: String =
spark.app.name=Spark shell
spark.jars=
spark.master=local[*]
spark.submit.deployMode=client

scala> println(conf.toDebugString)
spark.app.name=Spark shell
spark.jars=
spark.master=local[*]
spark.submit.deployMode=client
```

SparkContext

- A Spark context is essentially a client of Spark's execution environment and acts as the master of your Spark application (don't get confused with the other meaning of Master in Spark, though)

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



SparkContext offers the following functions:

- **Getting current configuration**
  - SparkConf
  - deployment environment (as master URL)
  - application name
  - deploy mode
  - default level of parallelism
  - Spark user
  - the time (in milliseconds) when SparkContext was created
  - Spark version
- **Setting configuration**
  - mandatory master URL
  - local properties
  - default log level
- **Creating objects**
  - RDDs
  - accumulators
  - broadcast variables
- **Accessing services,** e.g. TaskScheduler, LiveListenerBus, BlockManager, SchedulerBackends, ShuffleManager.
- **Running jobs**
- **Setting up Scheduler Backend, TaskScheduler and DAGScheduler**
- **Closure Cleaning**
- **Submitting Jobs Asynchronously**
- **Unpersisting RDDs, i.e. marking RDDs as non-persistent**
- **Registering SparkListener**
- **Programmable Dynamic Allocation**

Getting Existing or Creating New SparkContext (getOrCreate methods)

```
import org.apache.spark.SparkContext
val sc = SparkContext.getOrCreate()

// Using an explicit SparkConf object
import org.apache.spark.SparkConf
val conf = new SparkConf()
  .setMaster("local[*]")
  .setAppName("SparkMe App")
val sc = SparkContext.getOrCreate(conf)
```

SparkContext Constructors

We can create a SparkContext instance using the below four constructors

```
SparkContext()
SparkContext(conf: SparkConf)
SparkContext(master: String, appName: String, conf: SparkConf)
SparkContext(
  master: String,
  appName: String,
  sparkHome: String = null,
  jars: Seq[String] = Nil,
  environment: Map[String, String] = Map())
```

SparkSQL, HiveContext

- One of Sparks's modules is SparkSQL. SparkSQL can be used to process structured data
- SparkSQL has a SQLContext and a HiveContext.
- HiveContext is a super set of the SQLContext
- Spark community suggest using the HiveContext.
- SparkContext defined as sc and a HiveContext defined as sqlContext.
- HiveContext allows you to execute SQL queries as well as Hive commands.
- If you want to create sqlContext object, below code for it

```
val sqlContext = new org.apache.spark.sql.SQLContext(sc)
```

http://spark.apache.org/docs/latest/configuration.html#viewing-spark-properties

Core programing in spark using Scala

- Let us start with performing basic operations on text file

Step-1: Reading a file.

```
val textFile = sc.textFile("README.md")
    textFile: spark.RDD[String] = spark.MappedRDD@2ee9b6e3
```

Step-2:

```
    textFile.first()
    res1: String = # Apache Spark
```

Step-3: filter by keyword

```
    val linesWithSpark = textFile.filter(line =>
line.contains("Spark"))
    linesWithSpark: spark.RDD[String] = spark.FilteredRDD@7dd4af09
```

Step-4: count the number of lines containing word “spark”

```
    textFile.filter(line => line.contains("Spark")).count()
    res3: Long = 15
```

Word count example in scala, java, python

Scala example :

```
val textFile = sc.textFile("hdfs://...")

val counts = textFile.flatMap(line => line.split(" "))
             .map(word => (word, 1))
             .reduceByKey(_ + _)

counts.saveAsTextFile("hdfs://...")
```

Word count example in scala, java, python

Python example:

```
text_file = sc.textFile("hdfs://...")
counts = text_file.flatMap(lambda line: line.split(" ")) \
         .map(lambda word: (word, 1)) \
         .reduceByKey(lambda a, b: a + b)
counts.saveAsTextFile("hdfs://...")
```

Word count example in scala, java, python

Java example :

```
JavaRDD<String> textFile = sc.textFile("hdfs://...");
JavaRDD<String> words = textFile.flatMap(
new FlatMapFunction<String, String>() {
    public Iterable<String> call(String s) { return
Arrays.asList(s.split(" ")); }        });
JavaPairRDD<String, Integer> pairs =
words.mapToPair (new PairFunction<String, String,
Integer>() {   public Tuple2<String, Integer>
call(String s) {
    return new Tuple2<String, Integer>(s, 1); }
        });
```

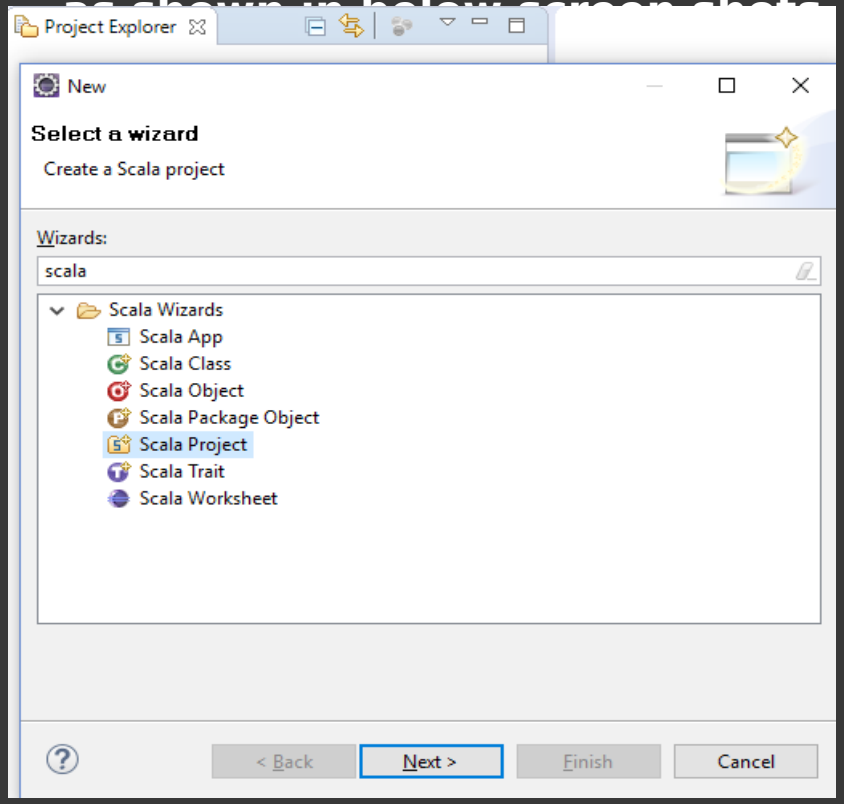
Word count example in scala, java, python

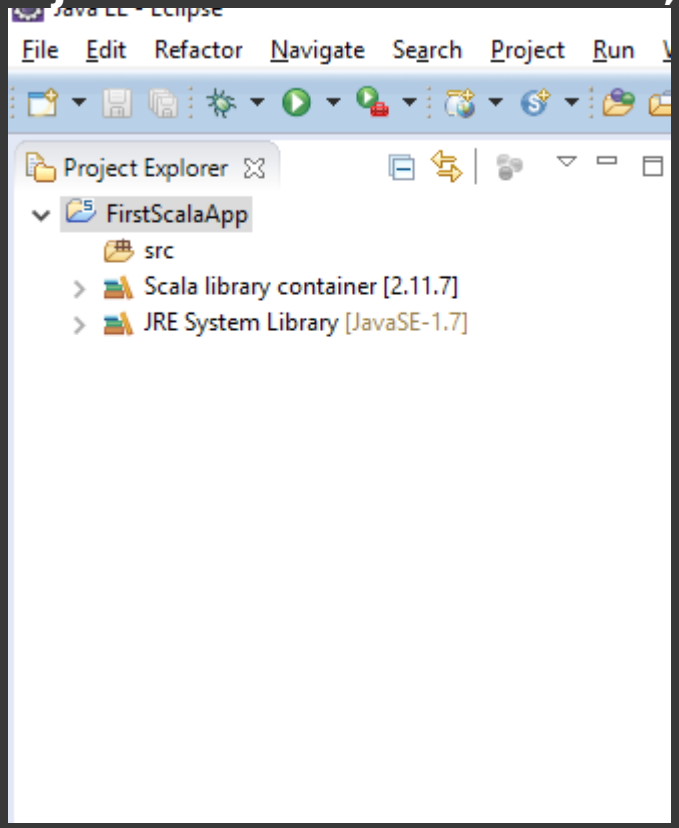
Java example:

```
text_file = sc.textFile("hdfs://...")

JavaPairRDD<String, Integer> counts =
        pairs.reduceByKey(new Function2<Integer,
Integer, Integer>()
        {
    public Integer call(Integer a, Integer b)
        { return a + b; }
        });
counts.saveAsTextFile("hdfs://...");
```

Building Spark Scala applications using maven build

- Create a new Scala project in Eclipse for Scala.
- Right click on project explorer -> new -> type scala in search box
- Now select scala project and give the project name and click finish, as shown in below screen shots





Building scala applications using maven build

- Now right click on the project, and go to Configure.
- Select Convert to Maven Project.
- The project will convert to a Maven for Scala.
- Open the pom.xml (double click).
- Select the Dependencies tab.
- Choose Add…
- Insert the dependence as per your project requirement.
- Here we are inserting spark dependency.

  - Group ID : org.apache.spark
  - Artifact ID : spark-core_2.11
  - Version : 1.3.0

- Save the changes, wait for a download to complete.
- Under a src (default package), create a New Scala File, name it “SparkPi”
- To test the spark, cut-and-paste the following code:

Building scala applications using maven build

```
import scala.math.random
import org.apache.spark._
/** Computes an approximation to pi */
object SparkPi {
def main(args: Array[String]) {
val conf = new SparkConf().setAppName("Spark Pi")
.setMaster("local")

val spark = new SparkContext(conf)
val slices = if (args.length > 0) args(0).toInt else 2
val n = math.min(100000L * slices, Int.MaxValue).toInt // avoid
overflow
val count = spark.parallelize(1 until n, slices).map { i =>
val x = random * 2 - 1
val y = random * 2 - 1
if (x*x + y*y < 1) 1 else 0
}.reduce(_ + _)
println("Pi is roughly " + 4.0 * count / n)
spark.stop()
}
}
```

- Run it as “Scala Application”
- The program should display a PI value among the other lines.

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-
4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <groupId>SimpleSpark</groupId>
  <artifactId>SimpleSpark</artifactId>
  <version>0.0.1-SNAPSHOT</version>
<build>
<sourceDirectory>src</sourceDirectory>
<plugins>
     <plugin>
     <groupId>org.scala-tools</groupId>
     <artifactId>maven-scala-plugin</artifactId>
     <version>2.15.2</version>
     <executions>
     <execution>
     <goals>
     <goal>compile</goal>
     <goal>testCompile</goal>
     </goals>
     </execution>
     </executions>
     </plugin>
     <plugin>
     <artifactId>maven-compiler-plugin</artifactId>
     <configuration>
     <source>1.7</source>
     <target>1.7</target>
     </configuration>
     </plugin>
```

```
      <plugin>
      <artifactId>maven-assembly-plugin</artifactId>
      <configuration>
      <descriptorRefs>
      <descriptorRef>jar-with-dependencies</descriptorRef>
      </descriptorRefs>
      </configuration>

      <executions>
      <execution>
      <id>make-assembly</id>
      <phase>package</phase>
      <goals>
      <goal>single</goal>
      </goals>
      </execution>
      </executions>
      </plugin>
      </plugins>
      </build>
  <dependencies>
  <dependency>
  <groupId>org.apache.spark</groupId>
  <artifactId>spark-core_2.11</artifactId>
  <version>1.3.0</version>
  </dependency>
  </dependencies>
</project>

Run as => Maven Build => clean install

$ spark-submit --class SparkPi SimpleSpark-0.0.1-SNAPSHOT-jar-with-dependencies.jar
</input_dir /output_dir>
```

RDDs

Before going to execute an example, we have to know about RDD in spark.

- **RDD –** ***Resilient Distributed Datasets*****.**
- **RDDs represent a collection of items distributed across many compute nodes that can be manipulated in parallel.**
- **In spark, a unit of data is considered as an RDD.**

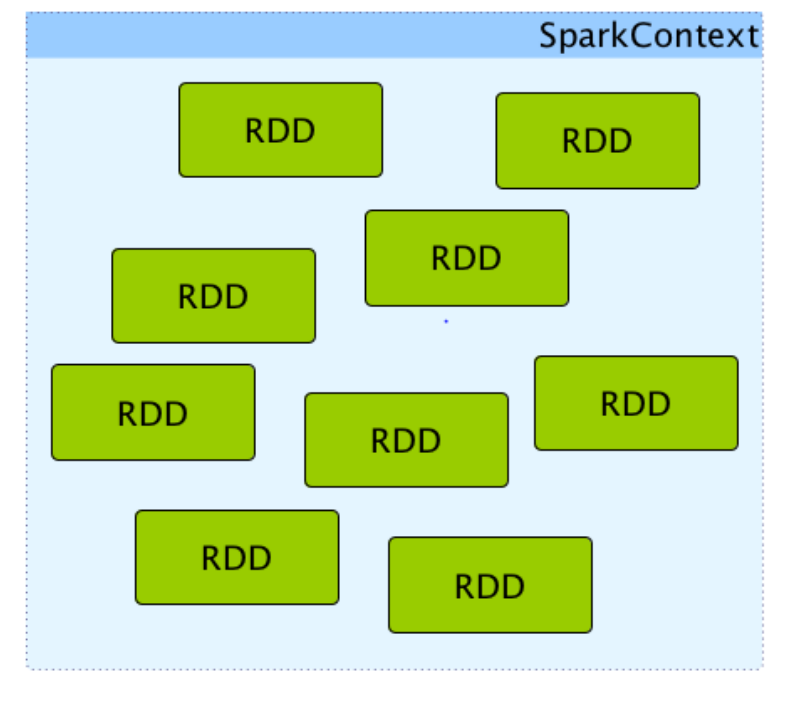
Resilient – It is nothing but if data in In-memory is lost, It will be recreated

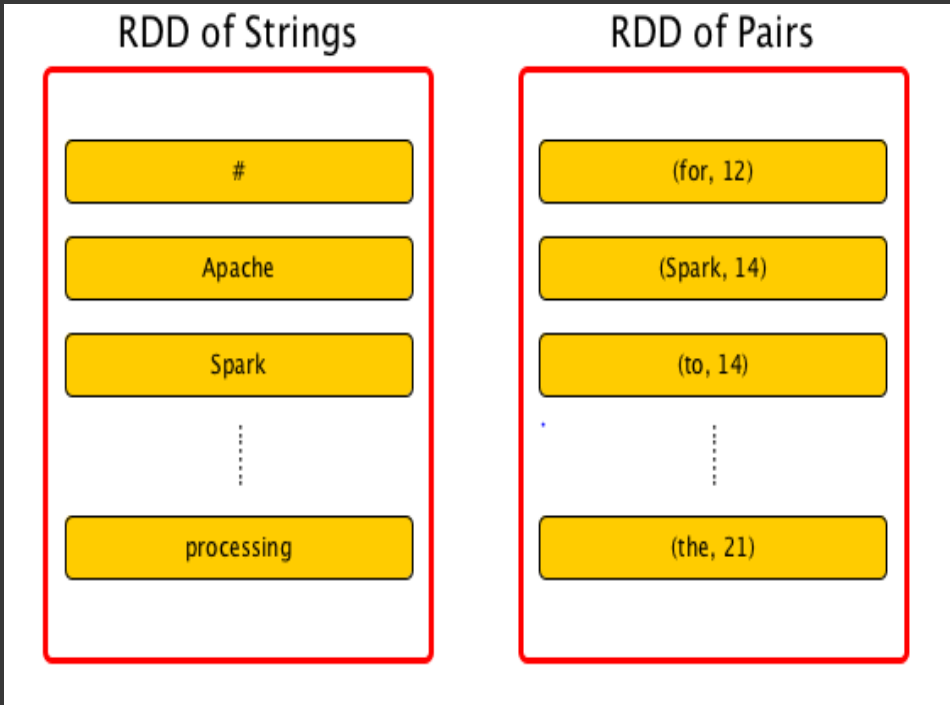
Distributed – Data will be stored across cluster

Datasets – It is nothing but group or collection of data which will come from file or streaming data or from any other source.

- RDD's are immutable
- They are generic means, it can store any type of data (Say Int, char, Boolean or Custom datatypes)
- Each RDD is split into multiple 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

- We can use Spark context to create RDDs.
- When an RDD is created, it belongs to and is completely owned by the Spark context it originated from.
- RDDs can't by design be shared between SparkContexts.





RDD Features

- **In-Memory**, i.e. data inside RDD is stored in memory as much (size) and long (time) as possible.
- **Immutable** or Read-Only, i.e. it does not change once created and can only be transformed using transformations to new RDDs.
- **Lazy evaluated**, i.e. the data inside RDD is not available or transformed until an action is executed that triggers the execution.
- **Cacheable**, i.e. you can hold all the data in a persistent "storage" like memory (default and the most preferred) or disk (the least preferred due to access speed).
- **Parallel**, i.e. process data in parallel.
- **Typed**, i.e. values in a RDD have types, e.g. RDD[Long] or RDD[(Int, String)].
- **Partitioned**, i.e. the data inside a RDD is partitioned (split into partitions) and then distributed across nodes in a cluster (one partition per JVM that may or may not correspond to a single node).

Creation of RDD's

- ✓ The simplest way to create RDDs is by SparkContext's parallelize() method

```
val lines = sc.parallelize(List("pandas", "i like pandas"))
```

- ✓ A more common way to create RDDs is to load data from external storage

```
val lines = sc.textFile("/path/to/README.md")
```

- ✓ Using makeRDD on SparkContext

```
sc.makeRDD(0 to 1000)
```

Partitions

- By default, one partition is created for each HDFS block while reading HDFS files into Spark RDDs.
- Ideally, we would get the same number of partitions as many blocks we see in HDFS for any HDFS file, but if the lines in your file are too long (longer than the block size), there will be fewer partitions
- Each of these partitions (default size 64 or 128 MB) will be stored in RAM memory of executors by their block managers
- Every RDD has a fixed number of partitions that determine the degree of parallelism to use when executing operations on the RDD
- Spark will always try to infer a sensible default value for partitions based on the size of our cluster and not beyond the count of cpu cores available
- We can get default parallelism value by sc.defaultParallelism
- But if we want to tune the level of parallelism for better performance during aggregations or grouping operations, we can ask Spark to use a specific number of partitions

Partitions

- The maximum size of a partition is ultimately limited by the available memory of an executor

- In the first RDD transformation, e.g. reading from a file using sc.textFile(path, partition), the partition parameter will be applied to all further transformations and actions on this RDD

- For compressed files default number of partitions is 1 only as Spark disables splitting on compressed files , we can use rdd.repartition(n) once after the file is loaded. It uses coalesce (combine) and shuffle to redistribute data

```
scala> rdd.coalesce(numPartitions=8, shuffle=false)
res10: …

Scala> res10.toDebugString
scala> rdd.coalesce(numPartitions=8, shuffle=true)
res11:…
scala> res11.toDebugString
```

- HashPartitioner is the default partitioner for coalesce operation when shuffle is allowed

Shuffling

- Shuffling is a process of redistributing data across partitions, I.e. data transfer between stages
- By default, shuffling doesn't change the number of partitions, but changes their content
- Avoid shuffling at all cost
- Avoid `groupByKey` and use `reduceByKey` or `combineByKey` instead.
  - ✓ `groupByKey` shuffles all the data, which is slow.
  - ✓ `reduceByKey` shuffles only the results of sub-aggregations in each partition of the data

Tuning the level of parallelism

```
val data = Seq(("a", 3), ("b", 4), ("a", 1))
sc.parallelize(data).reduceByKey((x, y) => x + y) // Default parallelism

INFO scheduler.DAGScheduler: Got job 7 (collect at <console>:24) with 1 output partitions
INFO scheduler.TaskSchedulerImpl: Adding task set 10.0 with 1 tasks
INFO scheduler.TaskSetManager: Finished task 0.0 in stage 11.0 (TID 20) in 38 ms on localhost (1/1)

sc.parallelize(data).reduceByKey((x, y) => x + y,10) // Custom parallel

INFO scheduler.DAGScheduler: Got job 8 (collect at <console>:24) with 10 output partitions
INFO scheduler.TaskSchedulerImpl: Adding task set 13.0 with 10 tasks
INFO scheduler.TaskSetManager: Finished task 9.0 in stage 13.0 (TID 31) in 27 ms on localhost (10/10)
```

- Most of the operators in spark accept a second parameter giving the number of partitions to use when creating the grouped or aggregated RDD

```
scala> sc.textFile("README.md").getNumPartitions
res0: Int = 1
scala> sc.textFile("README.md", 5).getNumPartitions
res1: Int = 5
scala> val ints = sc.parallelize(1 to 100, 4)
scala> ints.partitions.size  // 4
```

Transformation on RDDs

RDDs can be used for loading external dataset or for any transformation or to perform any action

- Transformations construct a new RDD from a previous one.
- For example, if use filter() transformation new RDD will be created.

```
transformation: RDD => RDD
transformation: RDD => Seq[RDD]
```

- Transformed RDDs are computed lazily, only when you use them in an action.
- In the above text file read example, filter() transformation will not applied until count() action is called.
- Actions, on the other hand, used to produce results. For example, if use first() action will give you first element in RDD
- Spark scans the file only until it finds the first matching line; it doesn't even read the whole file

Transformation on RDDs

- Spark's RDDs are by default recomputed each time we run an action on them. If we would like to reuse an RDD in multiple actions, we can ask Spark to persist it using RDD.persist()

- Spark internally records metadata to indicate that this operation has been requested
- Rather than thinking of an RDD as containing specific data, think of each RDD as consisting of instructions on how to compute the data that we build up through transformations

- **map(func) -** Returns a new distributed dataset, formed by passing each element of the source through a function func
- **filter(func) -** Returns a new dataset formed by selecting those elements of the source on which func returns true
- **flatMap(func) -** Similar to map, but each input item can be mapped to 0 or more output items (so func should return a Seq rather than a single item)
- **mapPartitions(func) -** Similar to map, but runs separately on each partition of the RDD, so func must be of type Iterator<T> ⇒ Iterator<U> when running on an RDD of type T
- From the scaladoc of org.apache.spark.rdd.RDD

Basic RDD transformations on an RDD containing {1, 2, 3, 3}

Function name	Purpose	Example	Result
Map()	Apply a function to each element in the RDD and return an RDD of the result	rdd.map(x => x + 1)	{2, 3, 4, 4}
flatMap()	Apply a function to each element in the RDD and return an RDD of the contents of the iterators returned. Often used to extract words.	rdd.flatMap(x => x.to(3))	{1, 2, 3, 2, 3, 3, 3}
filter()	Return an RDD consisting of only elements that pass the condition passed to filter().	rdd.filter(x => x != 1)	{2, 3, 3}
distinct()	Remove duplicates.	rdd.distinct()	{1, 2, 3}
sample(withReplacement, fraction)	Sample an RDD, with or without replacement.	rdd.sample(false,0.5)	Nondeterministic

Two-RDD transformations on RDDs containing {1, 2, 3} and {3, 4, 5}

Function name	Purpose	Example	Result
union()	Produce an RDD containing elements from both RDDs.	rdd.union(other)	{1, 2, 3, 3, 4, 5}
intersection()	RDD containing only elements found in both RDDs.	rdd.intersection(other)	{3}
subtract()	Remove the contents of one RDD (e.g., remove training data).	rdd.subtract(other)	{1, 2}
cartesian()	Cartesian product with the other RDD.	rdd.cartesian(other)	{(1, 3), (1, 4), ... (3,5)}

Types of RDD's

- ParallelCollectionRDD - an RDD of a collection of elements, it is the result of SparkContext.parallelize and SparkContext.makeRDD methods
- MapPartitionsRDD - an RDD that applies the provided function f to every partition of the parent RDD, it is the result of the following transformations
  - ✓ map
  - ✓ flatMap
  - ✓ filter
  - ✓ mapPartitions
  - ✓ mapPartitionsWithIndex
  - ✓ PairRDDFunctions.mapValues
  - ✓ PairRDDFunctions.flatMapValues
- HadoopRDD - an RDD that provides core functionality for reading data stored in HDFS, HadoopRDD is result of calling the following methods
  - ✓ hadoopFile
  - ✓ textFile
  - ✓ sequenceFile

Action on RDDs

- Actions are the second type of RDD's which will do some operation on existing dataset.
- In the above example count() function is an action RDD which will produce some result for an existing dataset.
- Actions are RDD operations that produce non-RDD values

```
action: RDD => a value
```

- Actions are one of two ways to send data from executors to the driver (the other being accumulators)

- **Reduce(func)** - Aggregate the elements of the dataset using a function func (which takes two arguments and returns one). The function should be commutative and associative so that it can be computed correctly in parallel
- **collect() -** Returns all the elements of the dataset as an array at the driver program. This is usually useful after a filter or other operation that returns a sufficiently small subset of the data

Action RDD's

- **count() -** Returns the number of elements in the dataset
- **first() -** Returns the first element of the dataset (similar to take (1))
- **take(n) -** Returns an array with the first n elements of the dataset
- **Max**
- **Min**
- **saveAsTextFile , saveAsHadoopFile**

- Actions run jobs using SparkContext.runJob
- Before calling an action, Spark does closure/function cleaning (using SparkContext.clean) to make it ready for serialization and sending over the wire to executors

Basic actions on RDDs containing {1, 2, 3, 3}

Function name	Purpose	Example	Result
collect()	Return all elements from the RDD.	rdd.collect()	{1, 2, 3, 3}
count()	Number of elements in the RDD.	rdd.count()	4
countByValue()	Number of times each element occurs in the RDD.	rdd.countByValue()	{(1, 1), (2, 1), (3, 2)}
take(num)	Return num elements from the RDD.	rdd.take(2)	{1, 2}
top(num)	Return the top num elements the RDD.	rdd.top(2)	{3,3}

Basic actions on RDDs containing {1, 2, 3, 3}

Function name	Purpose	Example	Result
takeSample(withReplacement, num)	Return num elements at random.	Rdd.takeSample(false,1)	Nondeterministic
reduce(func)	Combine the elements of the RDD together in parallel (e.g., sum).	rdd.reduce((x, y) => x + y)	9
fold(zero)(func)	Same as reduce() but with the provided zero value.	rdd.fold(0)((x, y) => x + y)	9
foreach(func)	Apply the provided function to each element of the RDD.	rdd.foreach(func)	Nothing

Debugging in spark

We will learn debugging in spark using a sample example.

input.txt, the source file for our example

```
## input.txt ##
INFO This is a message with content
INFO This is some other content
(empty line)
INFO Here are more messages
WARN This is a warning
(empty line)
ERROR Something bad happened
WARN More details on the bad thing
INFO back to normal messages
```

Debugging in Spark

Processing text data in the Scala Spark shell

```
// Read input file
scala> val input = sc.textFile("input.txt")
// Split into words and remove empty lines

scala> val tokenized = input.
  map(line => line.split(" ")).
  filter(words => words.size > 0)

// Extract the first word from each line (the log level) and
do a count

scala> val counts = tokenized.
| map(words => (words(0), 1)).
| reduceByKey{ (a, b) => a + b }
```

Debugging in Spark

Spark provides a **toDebugString()** method which is used to debug the spark jobs.

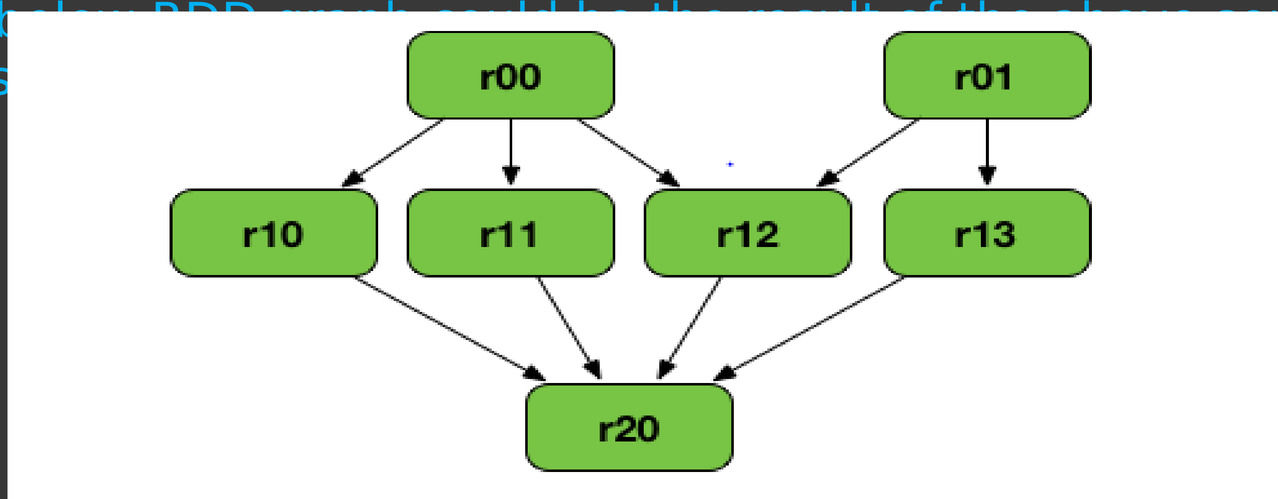
```
scala> input.toDebugString
res85: String =
(2) input.text MappedRDD[292] at textFile at <console>:13
| input.text HadoopRDD[291] at textFile at <console>:13

scala> counts.toDebugString
res84: String =
(2) ShuffledRDD[296] at reduceByKey at <console>:17
+-(2) MappedRDD[295] at map at <console>:17
| FilteredRDD[294] at filter at <console>:15
| MappedRDD[293] at map at <console>:15
| input.text MappedRDD[292] at textFile at <console>:13
| input.text HadoopRDD[291] at textFile at <console>:13
```

RDD Lineage

```
val r00 = sc.parallelize(0 to 9)
val r01 = sc.parallelize(0 to 90 by 10)
val r10 = r00 cartesian r01
val r11 = r00.map(n => (n, n))
val r12 = r00 zip r01
val r13 = r01.keyBy(_ / 20)
val r20 = Seq(r11, r12, r13).foldLeft(r10)(_ union _)
```

r20.toDebugString // open shell like $ ./bin/spark-shell -c spark.logLineage=true

The b … ies of
trans …



RDD CheckPointing

Checkpointing is a process of truncating RDD lineage graph and saving it to a reliable distributed (HDFS) or local file system.

Two types of checkpointing:

- **reliable** - in Spark (core), RDD checkpointing that saves the actual intermediate RDD data to a reliable distributed file system, e.g. HDFS.
- **local** - in Spark Streaming - RDD checkpointing that truncates RDD lineage graph.

It's up to a Spark application developer to decide when and how to checkpoint using RDD.checkpoint() method.

Before checkpointing is used, we need to set the checkpoint directory using sparkContext.setCheckpointDir(directory: String) method

RDD.checkpoint() operation is called, all the information related to RDD checkpointing are in ReliableRDDCheckpointData

RDD.localCheckpoint() that marks the RDD for local checkpointing using Spark's existing caching layer

RDD CheckPointing in Real Time Cluster

```
scala> sc.setCheckpointDir("/user/babusi02/chkdir/")
16/08/26 13:13:03 WARN SparkContext: Checkpoint directory must be non-local if Spark is running on a cluster: /user/babusi02/chkdir/

scala> r20.checkpoint()

scala> r20.foreach(println)

scala> r20.collect
res4: Array[(Int, Int)] = Array((0,0), (0,10), (0,20), (0,30), (0,40), (1,0), (1,10), (1,20), (1,30), (1,40), (2,0), (2,10), (2,20), (2,30), (2,40
), (3,0), (3,10), (3,20), (3,30), (3,40), (4,0), (4,10), (4,20), (4,30), (4,40), (0,50), (0,60), (0,70), (0,80), (0,90), (1,50), (1,60), (1,70), (
1,80), (1,90), (2,50), (2,60), (2,70), (2,80), (2,90), (3,50), (3,60), (3,70), (3,80), (3,90), (4,50), (4,60), (4,70), (4,80), (4,90), (5,0), (5,1
0), (5,20), (5,30), (5,40), (6,0), (6,10), (6,20), (6,30), (6,40), (7,0), (7,10), (7,20), (7,30), (7,40), (8,0), (8,10), (8,20), (8,30), (8,40), (
9,0), (9,10), (9,20), (9,30), (9,40), (5,50), (5,60), (5,70), (5,80), (5,90), (6,50), (6,60), (6,70), (6,80), (6,90), (7,50), (7,60), (7,70), (7,8
0), (7,90), (8,50), (8,60), (8,70), (8,80), (8,90), (9,50), (9,60),...
```

```
[babusi02@tparhegapq005 ~]$ hadoop fs -ls /user/babusi02/chkdir/d7e9d0be-195b-4520-97d0-da185539ca47/rdd-8/
Found 10 items
-rw-rw-rw-   3 babusi02 babusi02        150 2016-08-26 13:13 /user/babusi02/chkdir/d7e9d0be-195b-4520-97d0-da185539ca47/rdd-8/part-00000
-rw-rw-rw-   3 babusi02 babusi02        165 2016-08-26 13:13 /user/babusi02/chkdir/d7e9d0be-195b-4520-97d0-da185539ca47/rdd-8/part-00001
-rw-rw-rw-   3 babusi02 babusi02        150 2016-08-26 13:13 /user/babusi02/chkdir/d7e9d0be-195b-4520-97d0-da185539ca47/rdd-8/part-00002
-rw-rw-rw-   3 babusi02 babusi02        165 2016-08-26 13:13 /user/babusi02/chkdir/d7e9d0be-195b-4520-97d0-da185539ca47/rdd-8/part-00003
-rw-rw-rw-   3 babusi02 babusi02         50 2016-08-26 13:13 /user/babusi02/chkdir/d7e9d0be-195b-4520-97d0-da185539ca47/rdd-8/part-00004
-rw-rw-rw-   3 babusi02 babusi02         50 2016-08-26 13:13 /user/babusi02/chkdir/d7e9d0be-195b-4520-97d0-da185539ca47/rdd-8/part-00005
-rw-rw-rw-   3 babusi02 babusi02         30 2016-08-26 13:13 /user/babusi02/chkdir/d7e9d0be-195b-4520-97d0-da185539ca47/rdd-8/part-00006
-rw-rw-rw-   3 babusi02 babusi02         33 2016-08-26 13:13 /user/babusi02/chkdir/d7e9d0be-195b-4520-97d0-da185539ca47/rdd-8/part-00007
-rw-rw-rw-   3 babusi02 babusi02         30 2016-08-26 13:13 /user/babusi02/chkdir/d7e9d0be-195b-4520-97d0-da185539ca47/rdd-8/part-00008
-rw-rw-rw-   3 babusi02 babusi02         33 2016-08-26 13:13 /user/babusi02/chkdir/d7e9d0be-195b-4520-97d0-da185539ca47/rdd-8/part-00009
```

Passing RDDs to Methods

We can pass RDDs as function arguments as shown below,

```
def getMatchesFieldReference(rdd: RDD[String]): RDD[String] = {

rdd.map(x => x.split(query))
}
```

Caching an RDD

Sometimes we may wish to use the same RDD multiple times. For example,

```
val result = input.map(x => x*x)
println(result.count())
println(result.collect().mkString(","))
```

In the above we are calling count() and collect() functions, here if we don’t cache/persist the RDD, it will extra over head to spark, that it will read data two times.

Persist RDDs

- In above code if we use, result.persist() the result RDD data will be cached, so spark will not read content in RDD two times and it improves performance

- Difference between `cache` and `persist` operations is purely syntactic. cache is a synonym of persist or persist(MEMORY_ONLY), i.e. cache is just persist with the default storage level MEMORY_ONLY

- If we want to unpersist the RDD then use, result.unpersist()

Persist RDDs

Number _2 in the name denotes 2 replicas

Level	Space used	CPU time	In memory	On disk
MEMORY_ONLY / MEMORY_ONLY_2	High	Low	Y	N
MEMORY_ONLY_SER / MEMORY_ONLY_SER_2	Low	High	Y	N
MEMORY_AND_DISK / MEMORY_AND_DISK_2	High	Medium	Some	Some
MEMORY_AND_DISK_SER / MEMORY_AND_DISK_SER_2	Low	High	Some	Some
DISK_ONLY / DISK_ONLY_2	Low	High	N	Y

Persist RDDs

```
Scala> lines.persist()

scala> lines.getStorageLevel
res0: org.apache.spark.storage.StorageLevel = StorageLevel(disk=false,
memory=true, offheap=false, deserialized=true, replication=1)

Scala> lines.unpersist()

scala> lines.getStorageLevel
res0: org.apache.spark.storage.StorageLevel = StorageLevel(disk=false,
memory=false, offheap=false, deserialized=false, replication=1)
```

Pair RDDs

- Spark provides special operations on RDDs containing key/value pairs. These RDDs are called pair RDDs
- They will be useful for many real-time scenarios and will have separate methods
- For example, pair RDDs have a reduceByKey() method that can aggregate data separately for each key, and a join() method that can merge two RDDs together by grouping elements with the same key

Creating Pair RDD's

There many ways to create pair RDDs. For example,

```
val pairs = lines.map(x => (x.split(" ")(0), x))
```

Transformations on Pair RDD (example: {(1, 2), (3, 4), (3, 6)})

Function name	Purpose	Example	Result
reduceByKey(func)	Combine values with the same key.	rdd.reduceByKey((x, y) => x + y)	{(1, 2), (3, 10)}
groupByKey()	Group values with the same key.	rdd.groupByKey()	{(1, [2]), (3, [4, 6])}
mapValues(func)	Apply a function to each value of a pair RDD without changing the ky.	rdd.mapValues (x => x+1)	{(1, 3), (3, 5), (3, 7)}
flatMapValues(func)	Apply a function that returns an iterator to each value of a pair RDD, and for each element returned, produce a key/value entry with the old key.	rdd.flatMapValues(x => (x to 5)	{(1, 2), (1, 3), (1, 4), (1, 5), (3, 4), (3, 5)}

Transformations on Pair RDD (example: {(1, 2), (3, 4), (3, 6)})

Function name	Purpose	Example	Result
keys()	Return an RDD of just the keys.	rdd.keys()	{1, 3, 3}
values()	Return an RDD of just the values.	rdd.values()	{2, 4, 6}
sortByKey()	Return an RDD sorted by the key.	rdd.sortByKey()	{(1, 2), (3, 4), (3, 6)}

Transformations on two pair RDDs (rdd = {(1, 2), (3, 4), (3, 6)} other = {(3, 9)})

Function name	Purpose	Example	Result
subtractByKey()	Remove elements with a key present in the other RDD.	rdd.subtractByKey(other)	{(1, 2)}
join()	Perform an inner join between two RDDs.	rdd.join(other)	{(3, (4, 9)), (3, (6, 9))}
rightOuterJoin()	Perform a join between two RDDs where the key must be present in the first RDD.	rdd.rightOuterJoin(other)	{(3, (Some(4),9)), (3, (Some(6),9))}
leftOuterJoin()	Perform a join between two RDDs where the key must be present in the other RDD.	rdd.leftOuterJoin(other)	{(1, (2,None)), (3, (4,Some(9))), (3, (6,Some(9)))}
cogroup()	Group data from both	rdd.cogroup(oth	{(1,([2],[])),

Actions on pair RDDs

Action	Meaning
countByKey()	Count the number of elements for each key.
collectAsMap()	Collect the result as a map to provide easy lookup.
lookup(key)	Return all values associated with the provided key.
takeOrdered(n, [ordering])	Return the first n elements of the RDD using either their natural order or a custom comparator.

Joins on Pair RDDs

For example consider following data sets

storeAddress = { (Store("Ritual"), "1026 Valencia St"), (Store("Philz"), "748 Van Ness Ave"), (Store("Philz"), "3101 24th St"), (Store("Starbucks"), "Seattle")}

storeRating = { (Store("Ritual"), 4.9), (Store("Philz"), 4.8))}

Now appliying join funcation

storeAddress.join(storeRating) == { (Store("Ritual"), ("1026 Valencia St", 4.9)), (Store("Philz"), ("748 Van Ness Ave", 4.8)), (Store("Philz"), ("3101 24th St", 4.8))}

THANK YOU