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
  - ✓ start-all.sh / stop-all.sh for starting/stopping both master and worker daemons

- ❑ Spark Web UI
  - ✓ Master runs on 18080 port and hostname is **quickstart.cloudera.** 8080 is latest port number
  - ✓ Worker runs on 18081/8081(latest) 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()
```

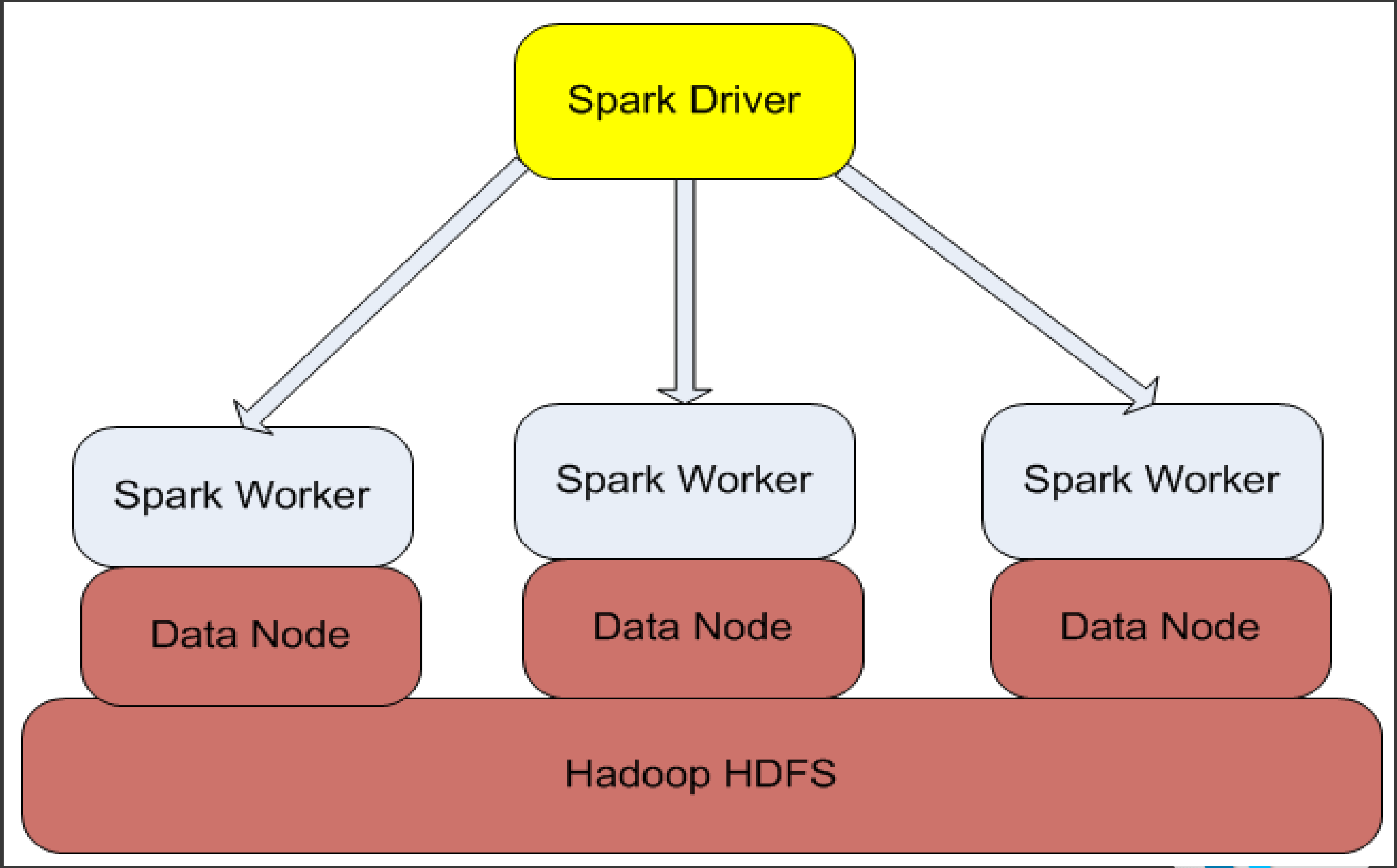
- ❑ Spark Scala Shell
  - ✓ spark-shell

```
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?

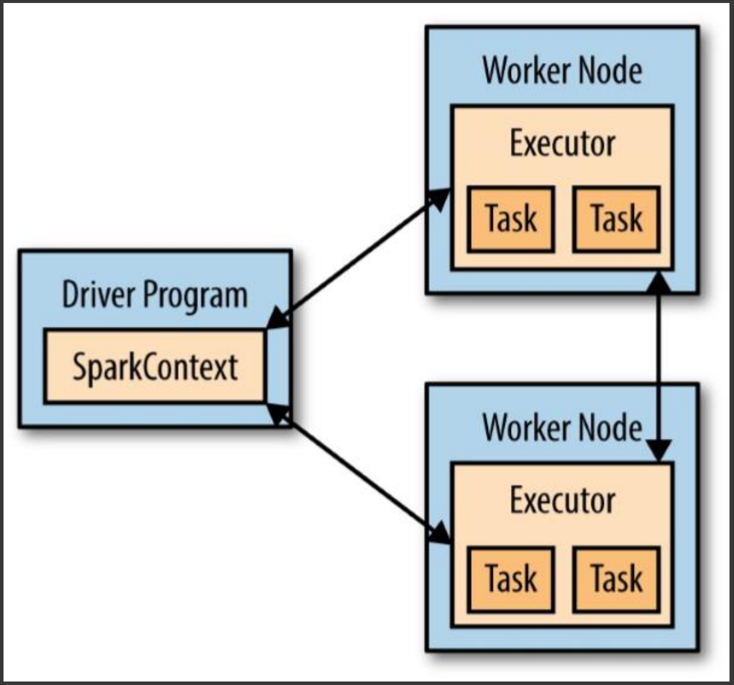
- Spark applications will have a driver program, that launches various parallel operations on the cluster

- 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

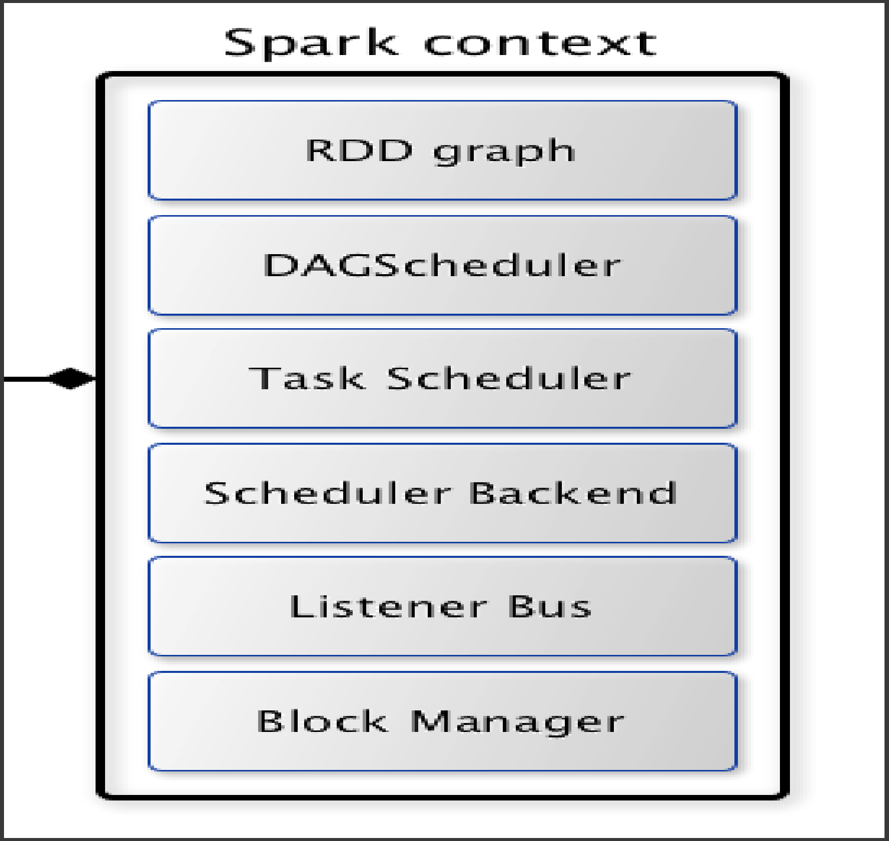
We can use conf.toDebugString or conf.getAll to have the spark.* system properties loaded printed out.

```
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**
- **ammable 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); }
                                    });
```

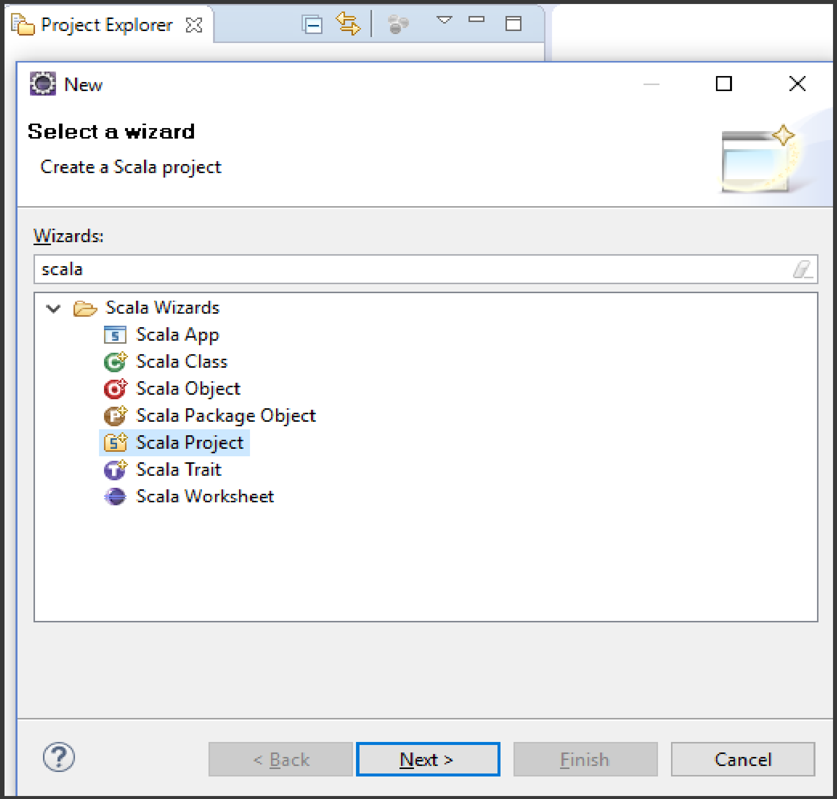
Word count example in scala, java, python

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.



New

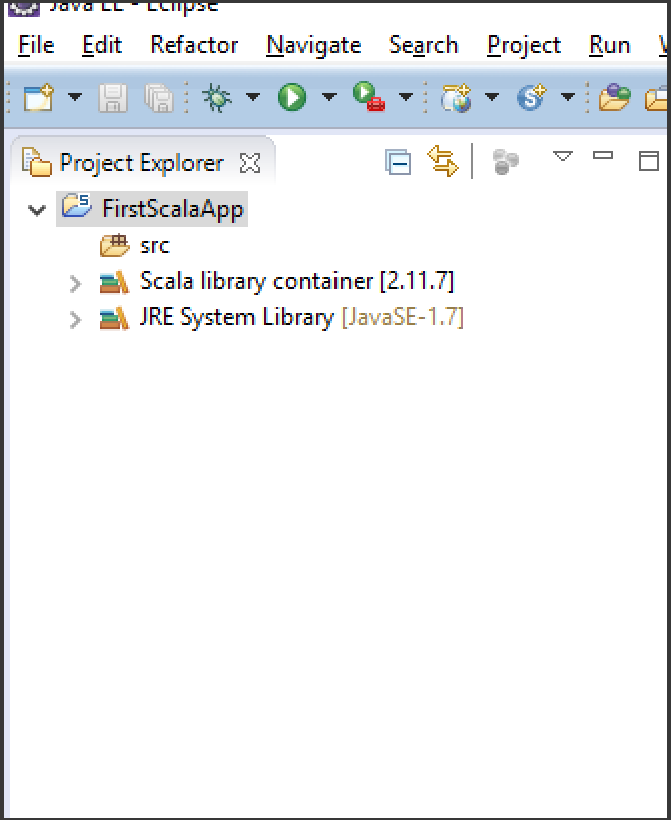
Select a wizard

Create a Scala project

Wizards:

scala

- Scala Wizards
  - Scala App
  - Scala Class
  - Scala Object
  - Scala Package Object
  - Scala Project
  - Scala Trait
  - Scala Worksheet



- FirstScalaApp
  - src
  - Scala library container [2.11.7]
  - JRE System Library [JavaSE-1.7]

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.**

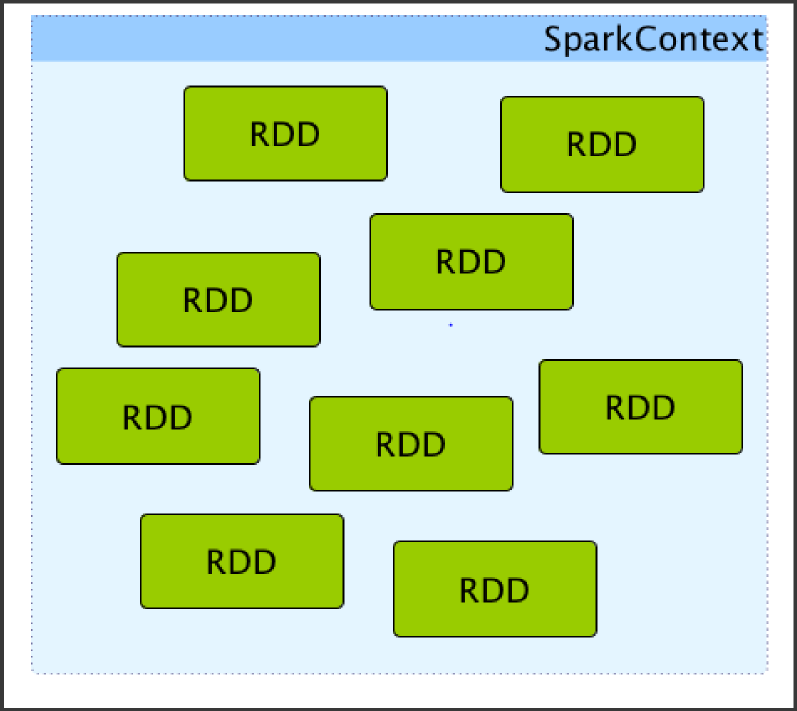
Resilient – It is nothing but if data in In-memory is lost, It will be recreated

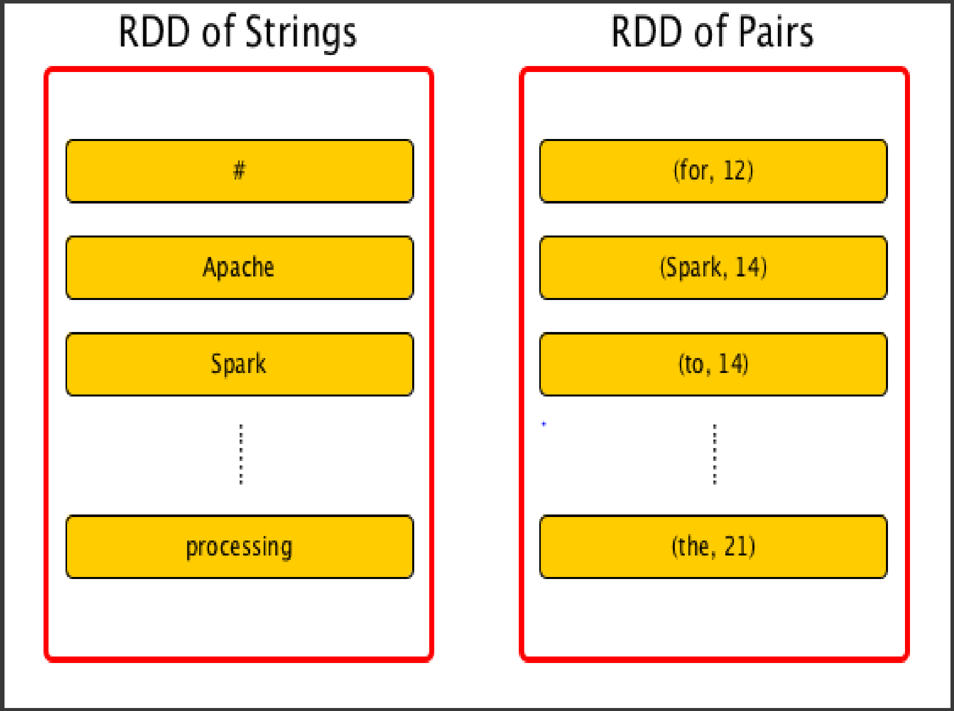
Distributed – Data will be stored across cluster

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.
- By design RDDs can't 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

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 RDDs sharing the same key.	rdd.cogroup(other)	{(1,([2],[])), (3,([4, 6],[9]))}

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))}

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.

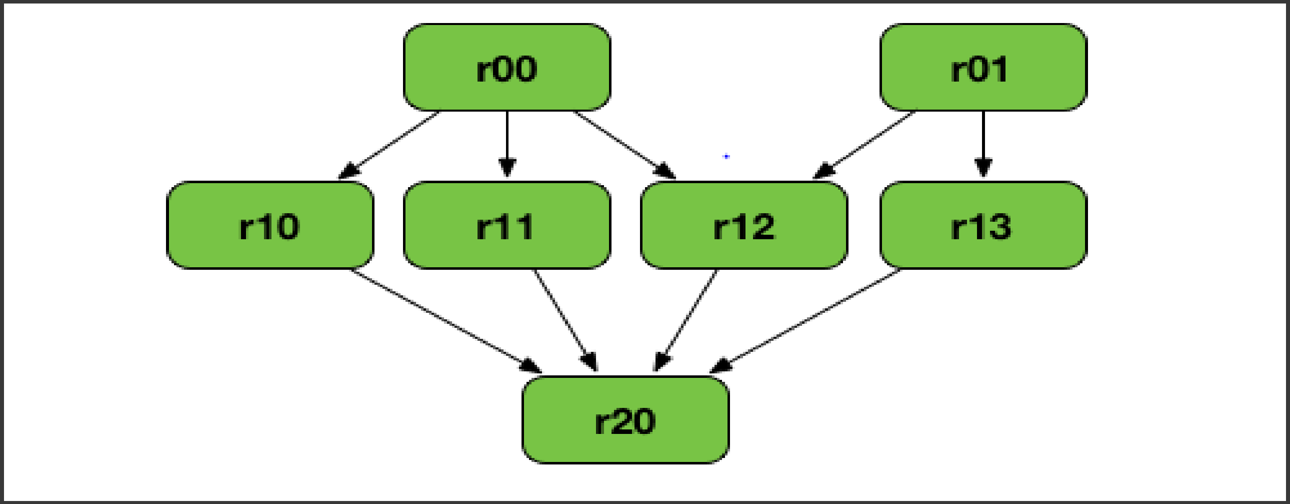
```
scala> input.toDebugString
res85: String =
(2) input.text MappedRDD[292] at textFile at <console>:13
| input.text HadoopRDD[291] at textFile at <console>:13

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 below RDD graph could be the result of the above series of transformations



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.

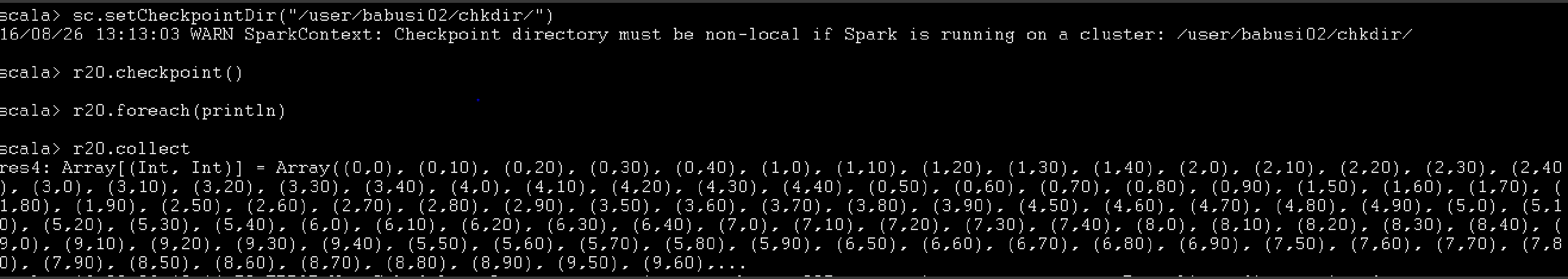
It's up to a Spark application developer to decide when and how to checkpoint using RDD.checkpoint() method.

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

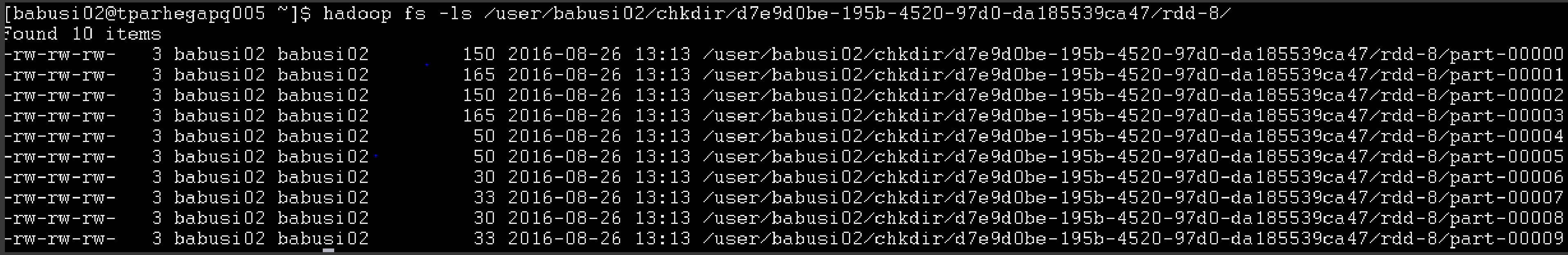
RDD.checkpoint() operation is called, all the information related to RDD checkpointing are in ReliableRDDCheckpointData

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.

When to go for Persisting

Spark processes are lazy, that is, nothing will happen until it's required. To quick answer the question, after val textFile = sc.textFile("/user/emp.txt") is issued, nothing happens to the data, only a HadoopRDD is constructed, using the file as source.

Let's say we transform that data a bit:

```
val wordsRDD = textFile.flatMap(line => line.split("\\W"))
```

Again, nothing happens to the data. Now there's a new RDD wordsRDD that contains a reference to testFile and a function to be applied when needed.

Only when an action is called upon an RDD, like wordsRDD.count, the RDD chain, called lineage will be executed. That is, the data, broken down in partitions, will be loaded by the Spark cluster's executors, the flatMap function will be applied and the result will be calculated.

On a linear lineage, like the one in this example, cache() is not needed. The data will be loaded to the executors, all the transformations will be applied and finally the count will be computed, all in memory - if the data fits in memory.

Passing RDDs to Methods

cache is useful when the lineage of the RDD branches out. Let's say you want to filter the words of the previous example into a count for positive and negative words. You could do this like that:

```
val positiveWordsCount = wordsRDD.filter(word => isPositive(word)).count()
val negativeWordsCount = wordsRDD.filter(word => isNegative(word)).count()
```

Here, each branch issues a reload of the data. Adding an explicit cache statement will ensure that processing done previously is preserved and reused. The job will look like this:

```
val textFile = sc.textFile("/user/emp.txt")
val wordsRDD = textFile.flatMap(line => line.split("\\W"))
wordsRDD.cache()
val positiveWordsCount = wordsRDD.filter(word => isPositive(word)).count()
val negativeWordsCount = wordsRDD.filter(word => isNegative(word)).count()
```

For that reason, cache is said to 'break the lineage' as it creates a checkpoint that can be reused for further processing.

Rule of thumb: Use cache when the lineage of your RDD branches out or when an RDD is used multiple times like in a loop

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)

res10.getStorageLevel
res20: org.apache.spark.storage.StorageLevel = StorageLevel(memory,
deserialized, 1 replicas)

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

In Spark 2.0.0
scala> res10.getStorageLevel                //After unpersist()
res24: org.apache.spark.storage.StorageLevel = StorageLevel(1
replicas)
```

Loading and Saving Data

- Spark can access data through the InputFormat and OutputFormat interfaces used by Hadoop MapReduce, which are available for many common file formats and storage systems (e.g., S3, HDFS, Cassandra, HBase, etc.).

- Spark supports major file formats and loading those files is very simple

File format name	Structured	Comments
Text files	No	Plain old text files. Records are assumed to be one per line.
JSON	Semi	Common text-based format, semistructured; most libraries require one record per line. (Will be discussed part of SQLContext)
CSV	Yes	Very common text-based format, often used with spreadsheet applications.

File Formats in Spark

File format name	Structured	Comments
Sequence Files	Yes	A common Hadoop file format used for key/value data.
Protocol buffers	Yes	A fast, space-efficient Multilanguage format.
Object files	Yes	Useful for saving data from a Spark job to be consumed by shared code. Breaks if you change your classes, as it relies on Java Serialization. saveAsObjectFile on RDD to write and sc.objectFile("path") to read

Loading text files

- For Text files line feed is the delimiter, each input line becomes an element in the RDD
- textFile() accepts path of a file on both HDFS and LFS, It also accepts directory path to read all the files one after the other.

```
val input = sc.textFile("README.md")   //hdfs file path by default
val file = sc.textFile("file:///home/cloudera/spark-1.5.1-bin-hadoop2.4/README.md")
                                        // Local files
```

Saving text files

- The method saveAsTextFile() can be used to store results into text file format.
- The path is treated as a directory and Spark will output multiple files underneath that directory.

```
rdd.saveAsTextFile("/user/cloudera/file_result")  //HDFS
rdd.saveAsTextFile("file:///home/cloudera/file_result")
                                                //LFS
```

Solution For Small Files Processing in Spark

- If we have a directory of small files and need to process all the files at a time in the spark, we could either provide the directory as input to textFile() method, or

- Use wholeTextFile() method which will read each file name & its contents as key/values into pair rdd.
- wholeTextFiles() can be very useful when each file represents a certain time period's data. If we had files representing sales data from different periods, we could easily compute the average for each period.

```
val input = sc.wholeTextFiles("file:///home/cloudera/sales")
val result = input.mapValues{y =>
val nums = y.split(" ").map(x => x.toDouble)
nums.sum / nums.size.toDouble}
```

- Spark supports reading all the files in a given directory and doing wildcard expansion on the input (e.g., part-*.txt). This is useful in selecting specific type of files when datasets are often mixed with multiple files (like success markers) in same directory.

Solution For Small Files Processing in Spark

- Spark supports reading all the files in a given directory and doing wildcard expansion on the input (e.g., part-*.txt). This is useful in selecting specific type of files when datasets are often mixed with multiple files (like success markers) in same directory.

```
val input = sc.wholeTextFiles("file:///home/cloudera/Desktop/*.txt")

scala> input.keys.collect
res40: Array[String] = Array(file:/home/cloudera/Desktop/data.txt,
file:/home/cloudera/Desktop/abc.txt, file:/home/cloudera/Desktop/dimensions.txt)

scala> input.values.collect
res41: Array[String] =
Array(-9999,49775,1,82365242,1,null,null,null,0,0,690…………….

input.flatMapValues{y => y.split(",")}.values.map(x => (x,0)).countByKey
```

Usage of Case Classes

Mapping a file with case class(Bean)

- ❑ Firstly we have to define a case class and load a file and map to the bean as shown below

```
               //Defining a case class
case class  Employee(id: Int, name: String, address: String)
               //Loading a text file and splitting by delimiter
val emp_details = sc.textFile("emp.txt").map(_.split("\t"))   //map returns array[String]
               //Map with Java bean
val emp_details_bean = emp_details.map(p => Employee (p(0).toInt,
               p(1).trim, p(2).trim)) //processing each array  with (index) to convert its data type
```

- ❑ Now we can apply conditions very easily

```
                //applying groupBy
val groupedResult = emp_details_bean.groupBy(_.id)
               //applying filter condition
val filtered = emp_details_bean.filter(p => p.id==123)
filtered.collect
```

Sequence Files

```
val data = sc.parallelize(List(("Panda", 3), ("Kay", 6),
("Snail", 2)))

data.saveAsSequenceFile("/user/cloudera/seq_out")
```

Reading Sequence File

```
val data = sc.sequenceFile("/user/cloudera/seq_out/part-
00000", classOf[org.apache.hadoop.io.Text],
classOf[org.apache.hadoop.io.IntWritable]).map{case (x,
y) => (x.toString, y.get())}

data.collect()
```

Hadoop File APIs

Old API

To read file using old Hadoop API, hadoopFile() function on spark context object, as shown below

```
val input = sc.hadoopFile[Text, Text,
KeyValueTextInputFormat](inputFile)
```

```
import org.apache.hadoop.mapred.KeyValueTextInputFormat
import org.apache.hadoop.io.Text

val input = sc.hadoopFile[Text, Text,
org.apache.hadoop.mapred.KeyValueTextInputFormat]("/user/cloudera/mr_out/part-r-00000")

input.keys.map(x => x.toString).collect
input. .map{case (x, y) => (x.toString, y.toString)    }
println(input.collect().toList)
```

Hadoop File APIs

New API

To read file using new Hadoop API, newAPIHadoopFile() function on spark context object, as shown below

```
val input = sc.newAPIHadoopFile(inputFile, classOf[LzoJsonInputFormat],
            classOf[LongWritable], classOf[MapWritable], conf)
```

Hadoop File APIs Example

```
import org.apache.hadoop.io.LongWritable
import org.apache.hadoop.io.Text
import org.apache.hadoop.conf.Configuration
import org.apache.hadoop.mapreduce.lib.input.TextInputFormat

val conf = new Configuration

conf.set("textinputformat.record.delimiter", "\t")

val data = sc.newAPIHadoopFile("emp.txt",
classOf[TextInputFormat], classOf[LongWritable],
classOf[Text], conf).map(_._2.toString)

data.take(5)
```

Loading ORC files

```
Create external table emp_txt(e_id int, name string, address string) row format delimited
fields terminated by ',' stored as textfile location '/user/cloudera/users/' ;

load data local inpath '/home/cloudera/emp.txt' into table emp_txt;

Create external table emp_orc(e_id int, name string, address string) stored as orcfile
location '/user/cloudera/orc/' ;

insert overwrite table emp_orc select * from emp_txt;

case class  Employee(e_id: Int, name: String, address: String)
```

ORC Files in Spark

- Loading ORC files

```
//Defining a case class
case class Employee(e_id: Int, name: String, address: String)

//Loading a ORC file and we have to mention key and value types
val employee_details =
sc.hadoopFile("/user/cloudera/orc/000000_0",
classOf[org.apache.hadoop.hive.ql.io.orc.OrcInputFormat],clas
sOf[org.apache.hadoop.io.NullWritable],classOf[org.apache.had
oop.hive.ql.io.orc.OrcStruct])
```

- Loading of Parquet files & Json Files will be done easily with SqlContext and will discuss during SparkSQL.

Loading ORC files

```
val employee_details =
sc.hadoopFile("hdfs://quickstart.cloudera:8020/user/cloudera/orc",
classOf[org.apache.hadoop.hive.ql.io.orc.OrcInputFormat],classOf[org
.apache.hadoop.io.NullWritable],classOf[org.apache.hadoop.hive.ql.io
.orc.OrcStruct])

val employee_details_string = employee_details.map(pair =>
pair._2.toString)
```

//Map with Java bean

```
val retailBeanList = employee_details_string.map(_.split(",")).map(p
=> Employee((p(0).trim.substring(1,p(0).length)).toInt ,
p(1).trim,p(2).trim.dropRight(1)))

retailBeanList.foreach(emp => println(emp.id+"------"+emp.name+"----
---"+emp.address))
```

Spark Advanced concepts

Shared variables

- The concept of shared variables is similar to distributed cache in Hadoop
- Shared variables will be accessible by all nodes
- Shared variables maintain a separate copy on each node
- Shared variables will not support any update operation on each node
- There are two types of shared variables
  - Broadcast variables
  - Accumulators

- Spark automatically sends all variables referenced in our closures to the worker nodes. While this is convenient, it can also be inefficient because
  a) The default task launching mechanism is optimized for small task sizes, and
  b) you might, in fact, use the same variable in multiple parallel operations, but Spark will send it separately for each operation

Broadcast variables

- Broadcast variables allow the programmer to keep a read-only variable cached on each machine rather than shipping a copy of it with tasks

- Spark also attempts to distribute broadcast variables using efficient broadcast algorithms to reduce communication cost

- The variable will be sent to each node only once, and should be treated as read-only (updates will not be propagated to other nodes)
- In such cases, you won’t be able to change the value of the broadcast variable except within the driver code

- Broadcast variables are:
  - ✓ Immutable
  - ✓ Distributed to the cluster
  - ✓ Fit in memory

Broadcast variables Limitations

- Broadcast variables have to be able to fit in memory on one machine. That means that they definitely should NOT be anything super large, like a large table or massive collection.
- Secondly, broadcast variables are immutable, meaning that they cannot be changed later on.
- This may seem inconvenient but it truly suits their use case.

Broadcast variables

We can declare broadcast variables in following way

```
scala> val broadcastVar = sc.broadcast(Array(1, 2, 3))
          broadcastVar:
          org.apache.spark.broadcast.Broadcast[Array[Int]] =
Broadcast(0)

scala> broadcastVar.value
          res0: Array[Int] = Array(1, 2, 3)
```

Broadcast variables

```
val empsRdd = sc.parallelize(Seq(("bhaskar",100), ("Ramesh",102),
("Ravi",103), ("Rakesh",104)))

val toolsRdd = sc.parallelize(Seq((100,"BigData"),(101,"C#"),
(103,"Java"), (102,"Scala"), (104,"Oracle")))

val broadcastedTools = sc.broadcast(toolsRdd.collectAsMap())

val empsWithTools = empsRdd.mapPartitions({row =>
 row.map(x => (x._1, x._2, broadcastedTools.value.getOrElse(x._2, -
1)))
}, preservesPartitioning = true)

//val empsWithTools = empsRdd.map(row => (row._1,
row._2,broadcastedTools.value.getOrElse(row._2,-1)))

empsWithTools.take(5)
```

Accumulators

- Accumulators are variables that are only “added” through an associative operation and can therefore be efficiently supported in parallel
- They can be used to implement counters (as in MapReduce) or sums
- When we normally pass functions to Spark, such as a map() function or a condition for filter(), they can use variables defined outside them in the driver program, but each task running on the cluster gets a new copy of each variable, and updates from these copies are not propagated back to the driver
- The code below shows an accumulator being used to add up the elements of an array

```
 val accum = sc.accumulator(0, "My Accumulator")
      accum: spark.Accumulator[Int] = 0
sc.parallelize(Array(1, 2, 3, 4)).foreach(x => accum += x)
      10/09/29 18:41:08 INFO SparkContext: Tasks finished in
0.317106 s
accum.value
      res2: Int = 10
```

Accumulators

```
val file = sc.textFile("README.md")
//Create an Accumulator[Int] initialized to 0
val blankLines = sc.accumulator(0)

var count = 0;   // Regular variable to increment in each task
file.foreach(line => {if (line.trim.length < 1)
{blankLines += 1; count += 1;}
})

println("Blank lines: " + blankLines.value + " But Count is: " + count)
```

Of course, it is possible to aggregate values from an entire RDD back to the driver program using actions like reduce(), but sometimes we need a simple way to aggregate values that, in the process of transforming an RDD, are generated at different scale or granularity than that of the RDD itself
often we expect some percentage of our data to be corrupted, or allow for the backend to fail some number of times. To prevent producing garbage output when there are too many errors, we can use a counter for valid records and a counter for invalid records

Accumulators

```
inrdd.foreach(line => { if (line.trim.length < 1) {blankLines += 1;
count += 1; println("blankLines : " + blankLines + " count : " +
count)} } )  // Work prints both the counters

inrdd.foreach(line => { if (line.trim.length < 1) {blankLines += 1;
count += 1; println("blankLines : " + blankLines.value + " count : " +
count)} } )  // Error : Can't read accumulator value in task
```

Accumulators

- Note that tasks on worker nodes cannot access the accumulator's value()—from the point of view of these tasks, accumulators are write-only variables. This allows accumulators to be implemented efficiently, without having to communicate every update.

- We create them in the driver by calling the SparkContext.accumulator(initialValue) method, which produces an accumulator holding an initial value. The return type is an org.apache.spark.Accumulator[T] object, where T is the type of initialValue.

- Worker code in Spark closures can add to the accumulator with its += method The driver program can call the value property on the accumulator to access its value.

- The value of our accumulators is available only in the driver program

Accumulators and Fault Tolerance

- Spark automatically deals with failed or slow machines by re-executing failed or slow tasks.
- For example, if the node running a partition of a map() operation crashes, Spark will rerun it on another node;
- And even if the node does not crash but is simply much slower than other nodes, Spark can pre-emptively launch a “speculative” copy of the task on another node, and take its result if that finishes.

- How does this interact with accumulators? The end result is that for *accumulators used in actions, Spark applies each task’s update to each accumulator only once*. Thus, if we want a reliable absolute value counter, regardless of failures or multiple evaluations, we must put it inside an action like foreach().

- For accumulators used in RDD transformations instead of actions, this guarantee does not exist. An accumulator update within a transformation can occur more than once.

Working on a Per-Partition Basis

- The Concept of per-partition is nothing but to avoid redoing setup work for each data item
- mapPartitions() can be used as an alternative to map() & foreach()
- mapPartitions() is called once for each Partition unlike map() & foreach() which is called for each element in the RDD

- The main advantage being that, we can do initialization on Per-Partition basis instead of per-element basis(as done by map() & foreach())

- Consider the case of Initializing a database. If we are using map() or foreach(), the number of times we would need to initialize will be equal to the no of elements in RDD. Whereas if we use mapPartitions(), the no of times we would need to initialize would be equal to number of Partitions

- We get Iterator as an argument for mapPartition, through which we can iterate through all the elements in a Partition.

- In this case we can use mapPartitions() function, which gives us an iterator of the elements in each partition of the input RDD and expects us to return an iterator of our results

Working on a Per-Partition Basis

- In addition to mapPartitions(), Spark has a number of other per-partition operators, listed below,

Function name	We are called with	We return	Function signature on RDD[T]
mapPartitions()	Iterator of the elements in that partition	Iterator of our return elements	f: (Iterator[T]) → Iterator[U]
mapPartitionsWithIndex()	Integer of partition number, and Iterator of the elements in that partition	Iterator of our return elements	f: (Int, Iterator[T]) → Iterator[U]
foreachPartition()	Iterator of the elements	Nothing	f: (Iterator[T]) → Unit

Working on a Per-Partition Basis

In this example, we will use mapPartitionsWithIndex(), which apart from similar to mapPartitions() also provides an index to track the Partition No

```
val rdd1 =  sc.parallelize(List("yellow","red","blue","cyan","black"), 3)

val mapped =   rdd1.mapPartitionsWithIndex {
              // 'index' represents the Partition No
              // 'iterator' to iterate through all elements in the partition
                 (index, iterator) => {
                      println("Called in Partition -> " + index)
                      val myList = iterator.toList
                      // In a normal use case, we will do the
                      // the initialization(ex : initializing database)
                      // before iterating through each element
                      myList.map(x => x + " -> " + index).iterator
                      }
                 }

mapped.collect()
```

Numeric RDD Operations

- Spark provides several descriptive statistics operations on RDDs containing numeric data.
- Few of them are listed below,

Method	Meaning
count()	Number of elements in the RDD
mean()	Average of the elements
sum()	Total
max()	Maximum value
min()	Minimum value
variance()	Variance of the elements
sampleVariance()	Variance of the elements, computed for a sample
stdev()	Standard deviation
sampleStdev()	Sample standard deviation

Spark Local (pseudo-cluster)

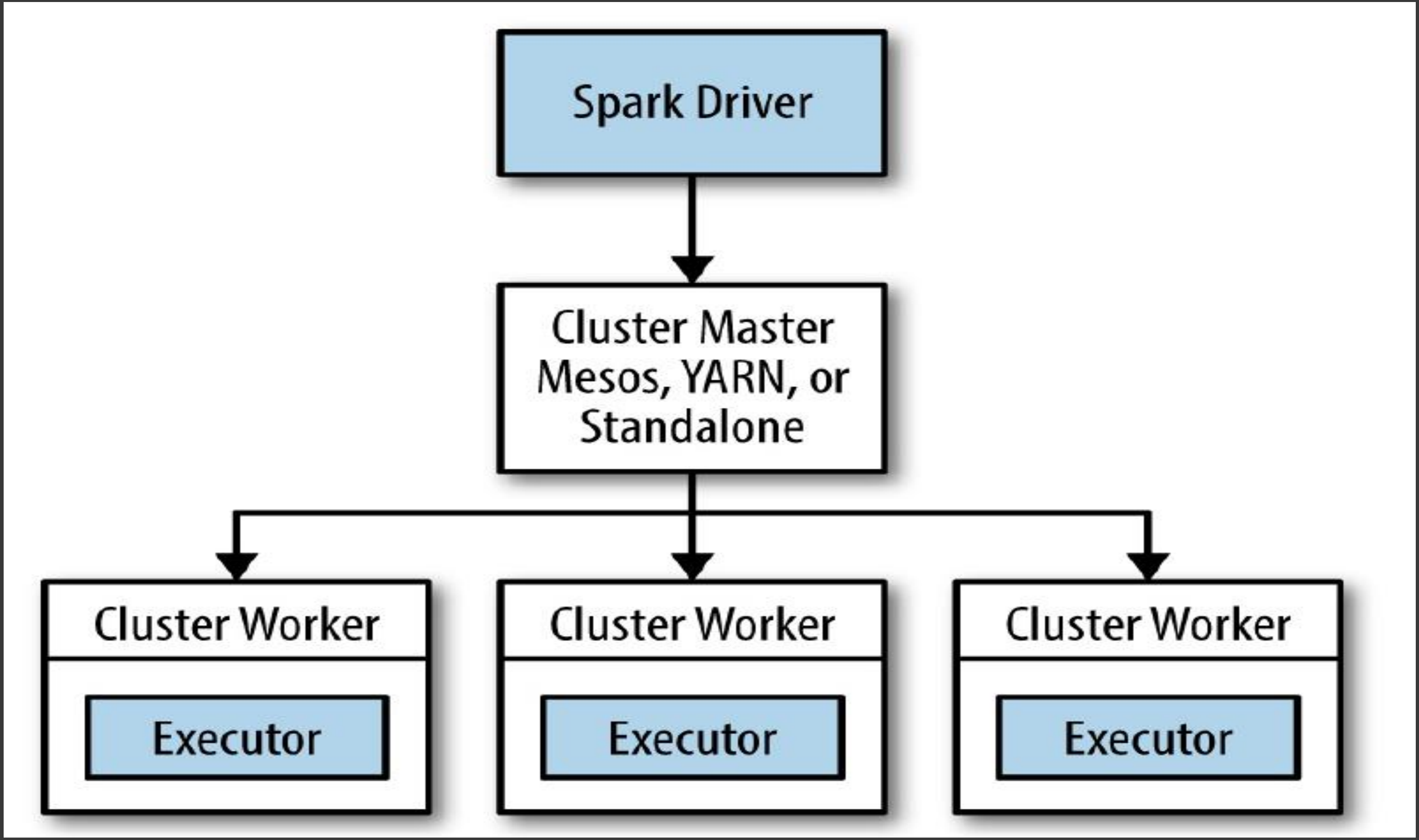
- You can run Spark in local mode. In this non-distributed single-JVM deployment mode, Spark spawns all the execution components - driver, executor, backend, and master - in the same JVM.
- The default parallelism is the number of threads as specified in the master URL.
- This is the only mode where a driver is used for execution
- The local mode is very convenient for testing, debugging or demonstration purposes

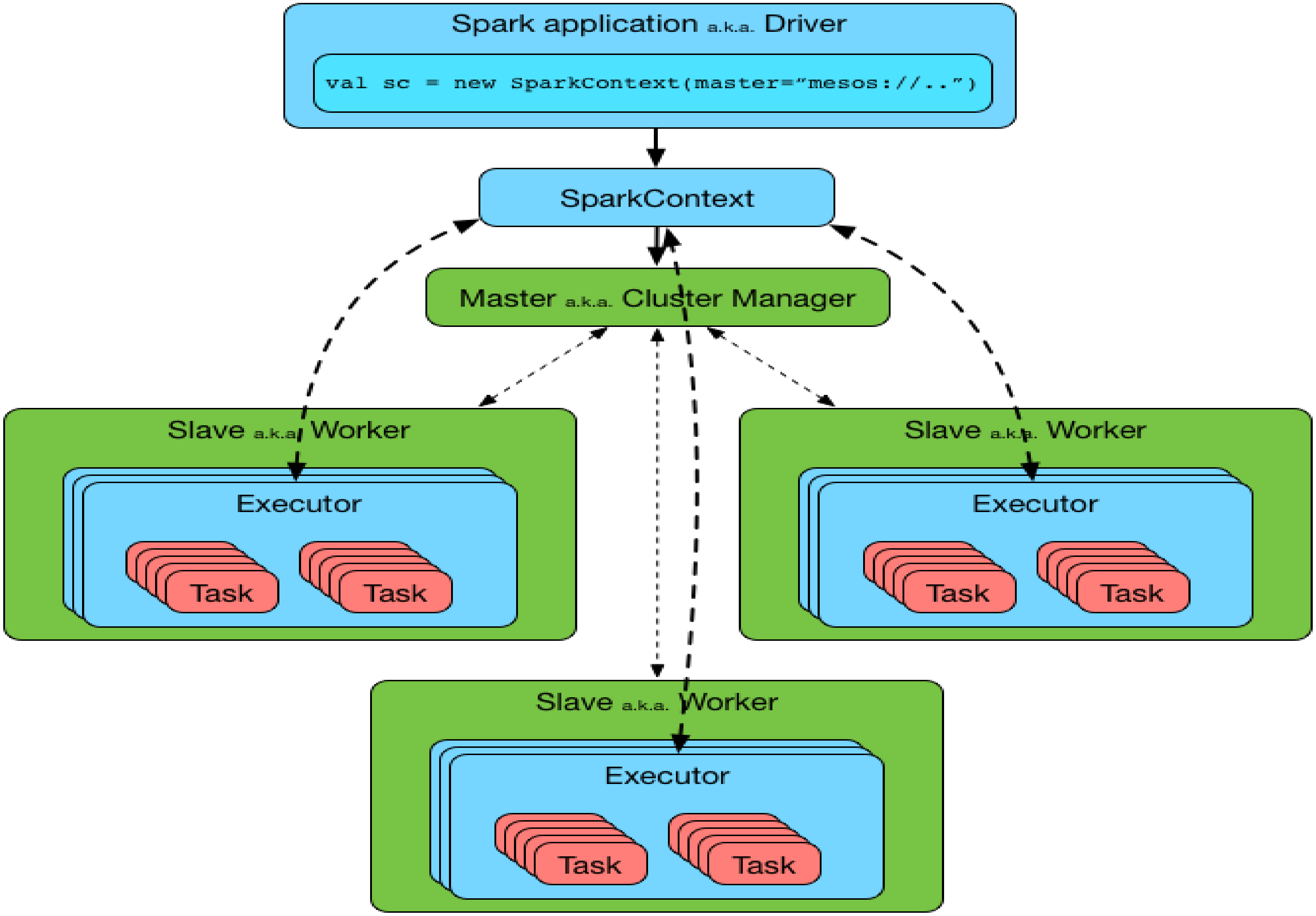
```
scala> sc.isLocal
res0: Boolean = true

scala> sc.master
res1: String = local[*]
```

Spark Runtime Architecture

- In distributed mode, Spark uses a master/slave architecture with one central coordinator and many distributed workers.
- The central coordinator is called the driver.
- The driver communicates with a potentially large number of distributed workers called executors.
- The driver runs in its own Java process and each executor is a separate Java process.
- A driver and its executors are together termed a ***Spark application***.
- A Spark application is launched on a set of machines using an external service called a ***cluster manager.***

Spark Runtime Architecture




Spark application a.k.a. Driver
val sc = new SparkContext(master="mesos://..")
SparkContext
Master a.k.a. Cluster Manager
Slave a.k.a. Worker
Executor
Task
Task
Slave a.k.a. Worker
Executor
Task
Task
Slave a.k.a. Worker
Executor
Task
Task

Driver

1. The driver is the process where the main() method of your program runs.
2. It is the process running the user code that creates a SparkContext, creates RDDs, and performs transformations and actions

3. The Spark driver is responsible for converting a user program into units of physical executions called tasks
4. At a high level, all Spark programs follow the same structure: they create RDDs from input, derive new RDDs from those using transformations, and perform actions to collect or save data. A Spark program implicitly creates a logical DAG of operations
5. When the driver runs, it converts this logical graph into a physical execution plan

6. Spark performs several optimizations, such as "pipelining" map transformations together to merge them, and converts the execution graph into a set of stages. Each stage, in turn, consists of multiple tasks
7. Tasks are the smallest unit of work in Spark; a typical user program can launch hundreds or thousands of individual tasks

Driver

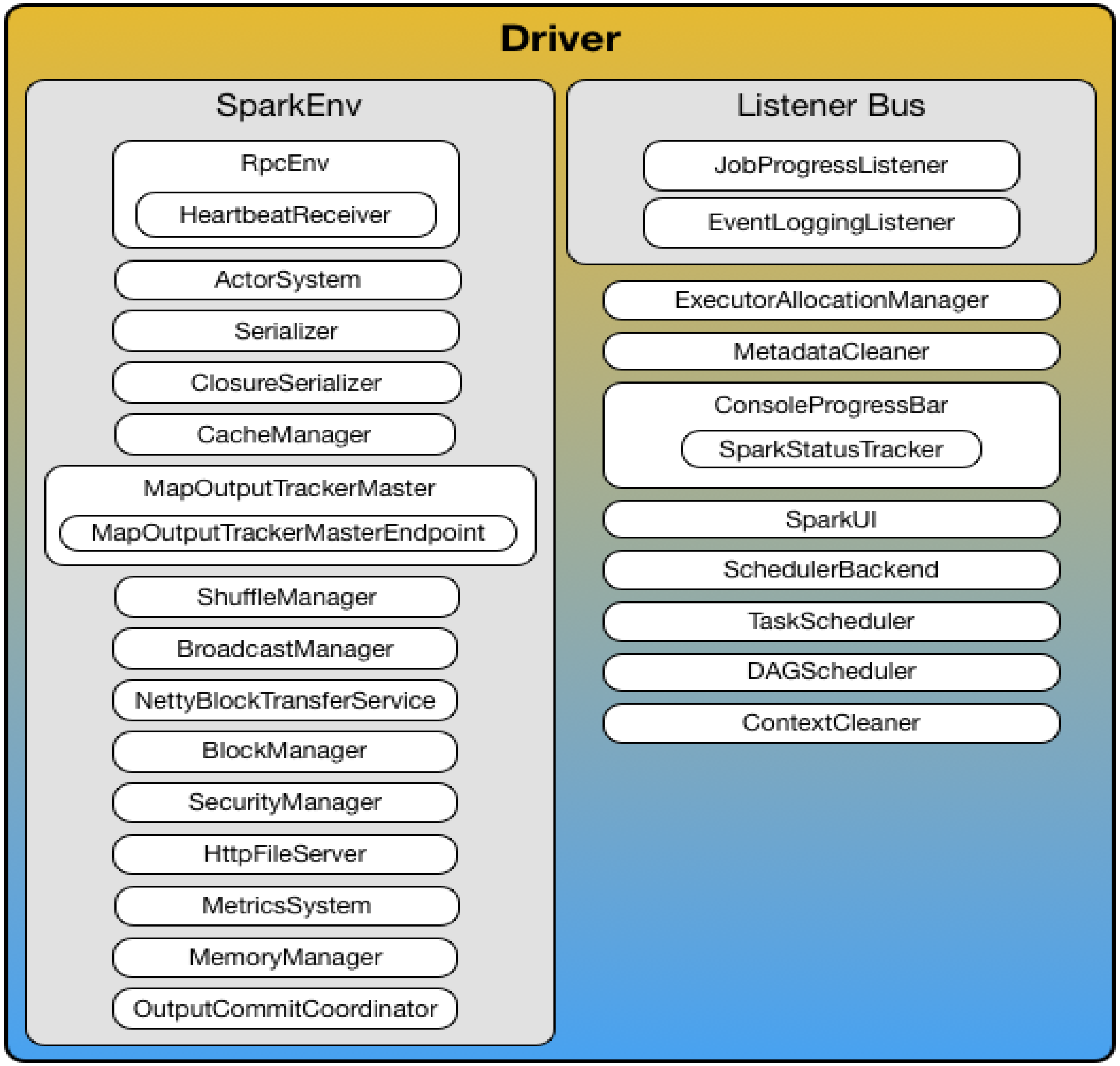
1. Given a physical execution plan, a Spark driver must coordinate the scheduling of individual tasks on executors. When executors are started they register themselves with the driver, so it has a complete view of the application's executors at all times

2. Each executor is a process that is capable of running tasks and storing RDD data

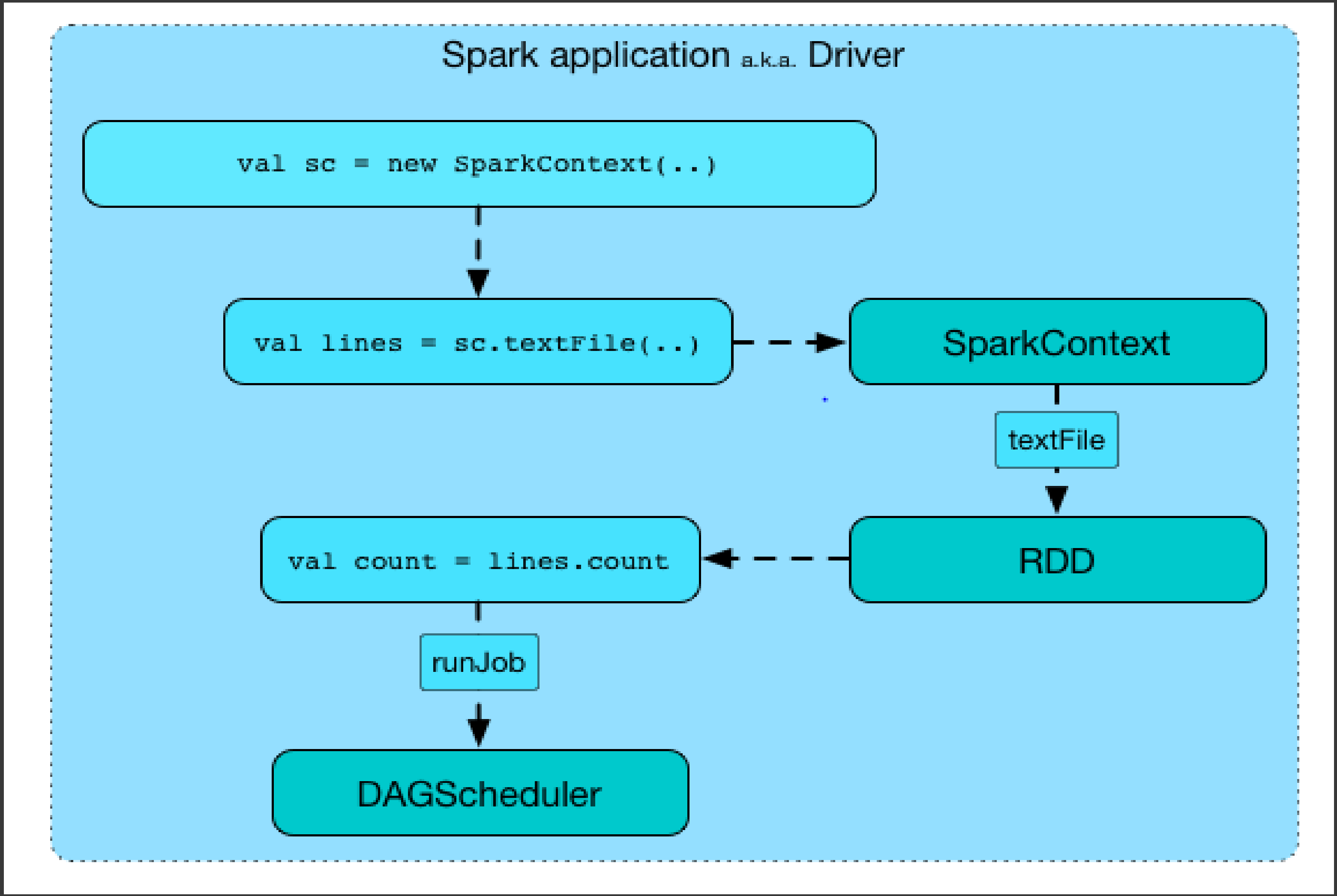
3. The Spark driver will look at the current set of executors and try to schedule each task in an appropriate location, based on data placement.

4. When tasks execute, they may have a side effect of storing cached data. The driver also tracks the location of cached data and uses it to schedule future tasks that access that data

5. The driver exposes information about the running Spark application through a web interface which by default is available at port 4040, *http://localhost:4040*


Driver
SparkEnv
RpcEnv
HeartbeatReceiver
ActorSystem
Serializer
ClosureSerializer
CacheManager
MapOutputTrackerMaster
MapOutputTrackerMasterEndpoint
ShuffleManager
BroadcastManager
NettyBlockTransferService
BlockManager
SecurityManager
HttpFileServer
MetricsSystem
MemoryManager
OutputCommitCoordinator
Listener Bus
JobProgressListener
EventLoggingListener
ExecutorAllocationManager
MetadataCleaner
ConsoleProgressBar
SparkStatusTracker
SparkUI
SchedulerBackend
TaskScheduler
DAGScheduler
ContextCleaner

Spark Job Execution Flow


Spark application a.k.a. Driver
val sc = new SparkContext(..)
val lines = sc.textFile(..)
SparkContext
textFile
RDD
val count = lines.count
runJob
DAGScheduler

Executors

1. Spark executors are worker processes responsible for running the individual tasks in a given Spark Job
2. Executors are launched once at the beginning of a Spark application and typically run for the entire lifetime of an application, though Spark applications can continue if executors fail
3. Executors have two roles
   - ✓ They run the tasks that make up the application and return results to the driver
   - ✓ They provide in-memory storage for RDDs that are cached by user programs, through a service called the Block Manager that lives within each executor

Cluster Manager

1. Spark depends on a cluster manager to launch executors and, in certain cases, to launch the driver also
2. The cluster manager is a pluggable component in Spark
3. This allows Spark to run on top of different external managers, such as YARN and Mesos, as well as its built-in Standalone cluster manager
4. spark-submit can run the driver within the cluster in YARN worker node, while others (Mesos & Stand alone) it can run it only on your local machine

Standalone Cluster Manager

1. If we want to run itself on a set of machines, the built-in Standalone mode is the easiest way to deploy it
2. Spark’s Standalone manager consists of a master and multiple workers, each with a configured amount of memory and CPU cores
3. While submitting spark application, we can choose how much memory its executors will use, as well as the total number of cores across all executors

Standalone Cluster Manager

- To submit an application to the Standalone cluster manager, pass spark://masternode:7077 as the master argument to spark-submit

- This cluster URL is also shown in the Standalone cluster manager's web UI, at http://masternode:8080

- For example, suppose that you have a 20-node cluster with 4-core machines, and you submit an application with --executor-memory 1G and --total-executor-cores 8. Then Spark will launch 8 executors, each with 1 GB of RAM, on different machines

- Standalone mode supports the failure of worker nodes. If we also want the master of the cluster to be highly available, Spark supports using Apache ZooKeeper to keep multiple standby masters and switch to a new one when any of them fails

YARN Cluster Manager

- YARN is a cluster manager that allows diverse data processing frameworks to run on a shared resource pool, and is typically installed on the same nodes as the HDFS

- When running on YARN we'll need to set the HADOOP_CONF_DIR environment variable to point the location of our Hadoop configuration directory, which contains information about the cluster
- Spark's interactive shell can work on YARN as well; simply set HADOOP_CONF_DIR and pass --master yarn to these applications

```
$ export HADOOP_CONF_DIR="/etc/hadoop/conf/"
$ spark-shell --master yarn
scala> val file = sc.textFile("README.md")
scala> file.count()
```

- Now we can refer the spark job from YARN RM UI → http://quickstart.cloudera:8088/
  SparkApplication will be continuously running under YARN for spark-shell, all the jobs submitted through spark-shell can be view through

YARN Web UI → ApplicationID → ApplicationMaster Url (All Spark Jobs Page same like host:4040 in standalone mode)

YARN Cluster Manager

- When running on YARN, Spark applications use a fixed number of executors, which we can set via the --num-executors flag to spark-submit, spark-shell, and so on. By default, this is only two, so we will likely need to increase it

- Some YARN clusters are configured to schedule applications into multiple “queues” for resource management purposes. Use the --queue option to select your queue name.

Spark Runtime Architecture

We will discuss the complete architecture by exact steps that occur when you run a Spark application on a cluster:

1. The user submits an application using spark-submit
2. spark-submit launches the driver program and invokes the main() method specified by the user
3. The driver program contacts the cluster manager to ask for resources to launch executors
4. The cluster manager launches executors on behalf of the driver program
5. The driver process runs through the user application. Based on the RDD actions and transformations in the program, the driver sends work to executors in the form of tasks
6. Tasks are run on executor processes to compute and save results
7. If the driver’s main() method exits or it calls SparkContext.stop(), it will terminate the executors and release resources from the cluster manager

Spark-submit

When spark-submit is called with just JAR as parameter, it simply runs the supplied Spark program locally

```
spark-submit --class mainclass sparkPI.jar
```

we wanted to submit this program to a Spark Standalone cluster. We can provide extra flags with the address of a Standalone cluster and a specific size of each executor process

```
spark-submit --class mainclass --master spark://localhost:7077
--executor-memory 1g sparkWordCount.jar input.txt output.txt
```

Common flags for spark-submit

Flag	Explanation
--master	Indicates the cluster manager to connect to.
--deploymode	Whether to launch the driver program locally (“client”) or on one of the worker machines inside the cluster (“cluster”). In client mode spark-submit will run your driver on the same machine where spark-submit is itself being invoked. In cluster mode, the driver will be shipped to execute on a worker node in the cluster. The default is client mode.
--class	The “main” class of your application if you’re running a Java or Scala program.
--name	A human-readable name for your application. This will be displayed in Spark’s web UI.
--jars	A list of JAR files to upload and place on the classpath of your application. If your application depends on a small number of third-party JARs, you can add them here.

Common flags for spark-submit

Flag	Explanation
--files	A list of files to be placed in the working directory of your application. This can be used for data files that you want to distribute to each node.
--py-files	A list of files to be added to the PYTHONPATH of your application. This can contain *.py*, *.egg*, or *.zip* files.
-- executormemory	The amount of memory to use for executors, in bytes. Suffixes can be used to specify larger quantities such as “512m” (512 megabytes) or “15g” (15 gigabytes).
--drivermemory	The amount of memory to use for the driver process, in bytes. Suffixes can be used to specify larger quantities such as “512m” (512 megabytes) or “15g” (15 gigabytes).

Possible values for the --master flag in spark-submit

Value	Explanation
spark://host:port	Connect to a Spark Standalone cluster at the specified port. By default Spark Standalone masters use port 7077
mesos://host:port	Connect to a Mesos cluster master at the specified port. By default Mesos masters listen on port 5050
yarn	Connect to a YARN cluster.
yarn-client	YARN mode but driver runs in client machine
yarn-cluster	YARN mode with driver running on cluster worker machine
local	Run in local mode with a single core, I.e one thread
local[N]	Run in local mode with N cores, i.e. N threads
local[*]	Run in local mode and use as many cores as the machine has.

Spark-Submit Example with flags

In cluster deploy mode, we can also specify --supervise to make sure that the **driver is automatically restarted if it fails with non-zero exit code**.

```
# Run application locally on 3 cores
$ spark-submit \
  --class org.apache.spark.examples.SparkPi \
  --master local[3] \
  /path/to/examples.jar

# Run on a Spark standalone cluster in client deploy mode
$ spark-submit \
  --class org.apache.spark.examples.SparkPi \
  --master spark://207.184.161.138:7077 \
  --executor-memory 20G \
  --total-executor-cores 100 \
  /path/to/examples.jar
```

Spark-Submit Example with flags

```
# Run on a Spark standalone cluster in cluster deploy mode with supervise
$ spark-submit \
  --class org.apache.spark.examples.SparkPi \
  --master spark://207.184.161.138:7077 \
  --deploy-mode cluster
  --supervise
  --executor-memory 20G \
  --total-executor-cores 100 \
  /path/to/examples.jar

# Run on a YARN cluster
$ export HADOOP_CONF_DIR="/etc/Hadoop/conf"
$ spark-submit \
  --class org.apache.spark.examples.SparkPi \
  --master yarn \
  --deploy-mode cluster \  # can be client for client mode
  --executor-memory 20G \
  --num-executors 50 \
  /path/to/examples.jar
```

Spark-Submit Example with flags

Here order of –flags is flexible, but after –flags, the first argument should be .jar file and followed by your program optional arguments.

```
$ spark-submit \
--master yarn-cluster\
--deploy-mode cluster \
--num-executors ${SPARK_NUM_EXECUTORS} \
--class com.databricks.examples.SparkExample \
--name "Example Program" \
--jars dep1.jar,dep2.jar,dep3.jar \
--total-executor-cores 300 \
--executor-memory 10g \
--driver-memory ${SPARK_DRIVER_MEMORY} \
--driver-cores ${SPARK_NUM_EXECUTOR_CORES} \
myApp.jar "options" "to your application" "go here"
```

```
#Local Mode
$ spark-submit --class BasicLoadNums
/home/cloudera/workspace2/SparkPiProject/target/SimpleSpark-
0.0.1-SNAPSHOT-jar-with-dependencies.jar local[*]
file:///home/cloudera/workspace2/SparkPiProject/input/input_load
_nums.txt

#Standalone Mode
$ spark-submit --class BasicLoadNums --master
spark://quickstart.cloudera:7077
/home/cloudera/workspace2/SparkPiProject/target/SimpleSpark-
0.0.1-SNAPSHOT-jar-with-dependencies.jar
spark://quickstart.cloudera:7077
file:///home/cloudera/workspace2/SparkPiProject/input/input_load
_nums.txt

#Yarn Mode
$ spark-submit --class BasicLoadNums --master yarn
/home/cloudera/workspace2/SparkPiProject/target/SimpleSpark-
0.0.1-SNAPSHOT-jar-with-dependencies.jar yarn
file:///home/cloudera/workspace2/SparkPiProject/input/input_load
_nums.txt
```

Spark Tuning Using Configuration

- Tuning Spark often simply means changing the Spark application’s runtime configuration.
- The primary configuration mechanism in Spark is the ***SparkConf*** class.
- Below example explains clearly,

```
// Construct a conf
val conf = new SparkConf()
conf.set("spark.app.name", "My Spark App")
conf.set("spark.master", "local[4]")
conf.set("spark.ui.port", "36000") //Override the default port
// Create a SparkContext with this configuration
val sc = new SparkContext(conf)
```

RDD Directed Acyclic Graph

Lets discuss a few internals of RDD DAGs

Consider below input.txt file

```
##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
```

RDD Directed Acyclic Graph

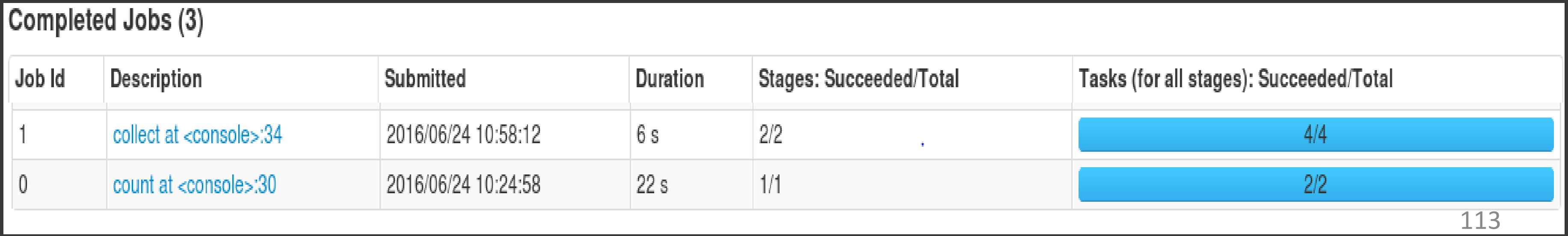
```
val rdd = sc.textFile("file:///home/cloudera/input.txt")

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

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

counts.collect

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
```

- Before we perform an action, RDDs simply store metadata that will be used to compute them later
- Spark's scheduler creates a physical execution plan (equivalent to logical plan displayed in **rdd.toDebugString**) to compute the RDDs needed for performing the action

- Spark's scheduler starts at the final RDD being computed (in this case, counts) and works backward to find what it must compute. It visits that RDD's parents, its parents' parents, and so on, recursively to develop a physical plan necessary to compute all ancestor RDD

- In simplest case, DAG scheduler outputs a computation ***stage*** for **each RDD** in this graph and the *stage* can have *tasks* for **each partition in that RDD**. Those stages are then executed in reverse order to compute the final required RDD

- In more complex cases, the physical set of stages will not be an exact 1:1 correspondence to the RDD graph. This can occur when the scheduler performs pipelining, or collapsing of multiple RDDs into a single stage. Pipelining occurs when RDDs can be computed from their parents without data movement


Completed Jobs (3)

Job Id	Description	Submitted	Duration	Stages: Succeeded/Total	Tasks (for all stages): Succeeded/Total
1	collect at <console>:34	2016/06/24 10:58:12	6 s	2/2	4/4
0	count at <console>:30	2016/06/24 10:24:58	22 s	1/1	2/2

RDD Cache Advantage

- The lineage output shown in previous slide uses indentation levels to show where RDDs are going to be pipelined together into physical stages. RDDs that exist at the same level of indentation as their parents will be pipelined during physical execution.

- For instance, when we are computing counts, even though there are a large number of parent RDDs, there are only two levels of indentation shown. This indicates that the physical execution will require only two stages

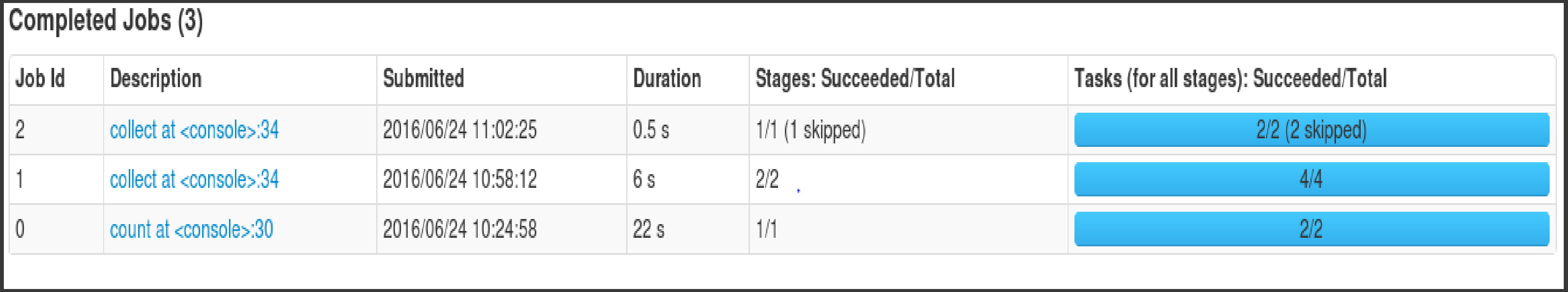
- But if we cache the counts rdd, then any of next actions doesn’t need all the previous stages to be recomputed every time

- Spark can “short-circuit” in this case and just begin computing based on the persisted RDD

```
scala> counts.cache()
scala> counts.collect
```

RDD Cache Advantage

- This will not submit two stages job but just 1 stage job to collect the elements and you can observe the time difference with two counts in the below screenshot



Completed Jobs (3)

Job Id	Description	Submitted	Duration	Stages: Succeeded/Total	Tasks (for all stages): Succeeded/Total
2	collect at <console>:34	2016/06/24 11:02:25	0.5 s	1/1 (1 skipped)	2/2 (2 skipped)
1	collect at <console>:34	2016/06/24 10:58:12	6 s	2/2	4/4
0	count at <console>:30	2016/06/24 10:24:58	22 s	1/1	2/2

Data Serialization

- Serialization plays an important role in the performance of any distributed application.
- Spark provides two types of serialization
  - Java serialization
  - Kryo serialization
- **Java serialization(Default):** By default, Spark serializes objects using Java's ObjectOutputStream framework, and can work with any class you create that implements java.io.Serializable
- Java serialization is flexible but often quite slow, and leads to large serialized formats for many classes

Data Serialization

- Kryo serialization: Spark can also use the Kryo library (version 2) to serialize objects more quickly.
- It is significantly faster and more compact than Java serialization (often as much as 10x).
- We can switch to using Kryo by initializing your job with a SparkConf and calling conf.set("spark.serializer", "org.apache.spark.serializer.KryoSerializer"). This setting configures the serializer used for not only shuffling data between worker nodes but also when serializing RDDs to disk.

Data Serialization

- But kryo does not support all Serializable types and requires you to register the classes as shown below
- To register your own custom classes with Kryo, use the registerKryoClasses method.

```
      val conf = new
SparkConf().setMaster(...).setAppName(...)
       conf.registerKryoClasses(Array(classOf[MyClass1],
      classOf[MyClass2]))
val sc = new SparkContext(conf)
```

Building Spark Applications Using SBT

```
$ curl https://bintray.com/sbt/rpm/rpm | sudo tee
/etc/yum.repos.d/bintray-sbt-rpm.repo
$ sudo yum install sbt
[cloudera@quickstart ~]$ find .
./src
./src/main
./src/main/scala
./src/main/scala/SimpleApp.scala
./build.sbt or simple.sbt

[cloudera@quickstart ~]$ cd simple-spark-app/
[cloudera@quickstart simple-spark-app]$ sbt package
```

Building Spark Applications Using SBT

```
import org.apache.spark.SparkContext
import org.apache.spark.SparkContext._
import org.apache.spark.SparkConf

object SimpleApp {
 def main(args: Array[String]) {
   val logFile = "file:///home/cloudera/README.md"
   val conf = new SparkConf().setAppName("Simple Application")
   val sc = new SparkContext(conf)
   val logData = sc.textFile(logFile, 2).cache()
   val numAs = logData.filter(line => line.contains("a")).count()
   val numBs = logData.filter(line => line.contains("b")).count()
   println("Lines with a: %s, Lines with b: %s".format(numAs, numBs))
 }}
```

Building Spark Applications Using SBT

```
/* simple.sbt */ or build.sbt
name := "Simple Project"

version := "1.0"

scalaVersion := "2.10.4"

libraryDependencies += "org.apache.spark" %% "spark-core" %
"1.3.0"

resolvers += "Akka Repository" at "http://repo.akka.io/releases/"
```

Declaring a dependency looks like this, where groupId, artifactId, and revision are strings:

```
libraryDependencies += groupID % artifactID % revision
```

Of course, sbt (via Ivy) has to know where to download the module. If your module is in one of the default repositories sbt comes with, this will just work. For example, Apache Derby is in the standard Maven2 repository:

```
libraryDependencies += "org.apache.derby" % "derby" % "10.4.1.3"
```

If you type that in build.sbt and then update, sbt should download Derby to ~/.ivy2/cache/org.apache.derby/. (By the way, update is a dependency of compile so there’s no need to manually type update most of the time.)

Of course, you can also use ++= to add a list of dependencies all at once:

```
libraryDependencies ++= Seq(
  groupID % artifactID % revision,
  groupID % otherID % otherRevision
)
```

In rare cases you might find reasons to use := with libraryDependencies as well.

Getting the right Scala version with %%

If you use groupID %% artifactID % revision rather than groupID % artifactID % revision (the difference is the double %% after the groupID), sbt will add your project's Scala version to the artifact name. This is just a shortcut. You could write this without the %%:

```
libraryDependencies += "org.scala-tools" % "scala-stm_2.11.1" % "0.3"
```

Assuming the scalaVersion for your build is 2.11.1, the following is identical (note the double %% after "org.scala-tools"):

```
libraryDependencies += "org.scala-tools" %% "scala-stm" % "0.3"
```

The idea is that many dependencies are compiled for multiple Scala versions, and you'd like to get the one that matches your project to ensure binary compatibility.

Building Spark Applications Using SBT

```
curl https://bintray.com/sbt/rpm/rpm | sudo tee
/etc/yum.repos.d/bintray-sbt-rpm.repo
[cloudera@quickstart simple-spark-app]$ sbt package
[cloudera@quickstart simple-spark-app]$ cd target/scala-
2.10/simple-project_2.10-1.0.jar

$ spark-submit --class SimpleApp simple-project_2.10-
1.0.jar
```

THANK YOU