ACCUMULATORS

Say you were using Spark to process some logs

```
2016-06-08 12:51:29,517 INFO  [regionserver//192.168.0.118:16201] regionserver.HRegionServer: Stopping infoServer
2016-06-08 12:51:29,521 INFO  [regionserver//192.168.0.118:16201] mortbay.log: Stopped SelectChannelConnector@0.0.0.0:1
2016-06-08 12:51:29,522 INFO  [regionserver//192.168.0.118:16201] regionserver.HRegionServer: aborting server 192.168.0
2016-06-08 12:51:29,522 INFO  [regionserver//192.168.0.118:16201] client.ConnectionManager$HConnectionImplementation: C
2016-06-08 12:51:29,918 INFO  [regionserver//192.168.0.118:16201-SendThread(localhost:2181)] zookeeper.ClientCnxn: Open
(unknown error)
2016-06-08 12:51:30,021 INFO  [regionserver//192.168.0.118:16201] zookeeper.ZooKeeper: Session: 0x0 closed
2016-06-08 12:51:30,022 INFO  [regionserver//192.168.0.118:16201-EventThread] zookeeper.ClientCnxn: EventThread shut do
2016-06-08 12:51:30,024 INFO  [regionserver//192.168.0.118:16201] regionserver.HRegionServer: stopping server 192.168.0
2016-06-08 12:51:30,024 INFO  [regionserver//192.168.0.118:16201] hbase.ChoreService: Chore service for: 192.168.0.118,
2016-06-08 12:51:30,026 INFO  [regionserver//192.168.0.118:16201] ipc.RpcServer: Stopping server on 16201
2016-06-08 12:51:30,375 INFO  [main-SendThread(localhost:2181)] zookeeper.ClientCnxn: Opening socket connection to serv
2016-06-08 12:51:30,375 WARN  [main-SendThread(localhost:2181)] zookeeper.ClientCnxn: Session 0x0 for server null, unex
```

The logs are stored in a text file

```
2016-06-08 12:51:29,517 INFO  [regionserver//192.168.0.118:16201] regionserver.HRegionServer: Stopping infoServer
2016-06-08 12:51:29,521 INFO  [regionserver//192.168.0.118:16201] mortbay.log: Stopped SelectChannelConnector@0.0.0.0:1
2016-06-08 12:51:29,522 INFO  [regionserver//192.168.0.118:16201] regionserver.HRegionServer: aborting server 192.168.0
2016-06-08 12:51:29,522 INFO  [regionserver//192.168.0.118:16201] client.ConnectionManager$HConnectionImplementation: C
2016-06-08 12:51:29,918 INFO  [regionserver//192.168.0.118:16201-SendThread(localhost:2181)] zookeeper.ClientCnxn: Open
(unknown error)
2016-06-08 12:51:30,021 INFO  [regionserver//192.168.0.118:16201] zookeeper.ZooKeeper: Session: 0x0 closed
2016-06-08 12:51:30,022 INFO  [regionserver//192.168.0.118:16201-EventThread] zookeeper.ClientCnxn: EventThread shut do
2016-06-08 12:51:30,024 INFO  [regionserver//192.168.0.118:16201] regionserver.HRegionServer: stopping server 192.168.0
2016-06-08 12:51:30,024 INFO  [regionserver//192.168.0.118:16201] hbase.ChoreService: Chore service for: 192.168.0.118,
2016-06-08 12:51:30,026 INFO  [regionserver//192.168.0.118:16201] ipc.RpcServer: Stopping server on 16201
2016-06-08 12:51:30,375 INFO  [main-SendThread(localhost:2181)] zookeeper.ClientCnxn: Opening socket connection to serv
2016-06-08 12:51:30,375 WARN  [main-SendThread(localhost:2181)] zookeeper.ClientCnxn: Session 0x0 for server null, unex
```

You have a specific set of processing steps for this log

1. Parse the logs

2. Save the parsed logs into a new text file

1. Parse the logs

2. Save the parsed logs into a new text file

At the end of this, you also want to print the number of ERROR messages in the log

Option 1:

1 Action to parse the logs and save them to text file

1. Parse the logs
2. Save the parsed logs into a new text file
3. Print the number of error messages

Option 1:

1 Action to parse the logs and save them to text file

1. Parse the logs
2. Save the parsed logs into a new text file
3. Print the number of error messages

A 2nd Action to compute the count of Error messages

Option 1:

1 Action to parse the logs and save them to text file

A 2nd Action to compute the count of Error messages

Option 2:

Use an accumulator variable

Accumulator variable

An Accumulator is a special type of variable

It is shared among all the nodes of the Spark cluster

Accumulator variable

Broadcast variables are also shared variables

Broadcast variables are immutable though

Accumulators are not!

Accumulator variable

Accumulators have 2 important characteristics

1. Individual nodes can only write to the accumulator

2. The main program (or Spark shell) can only read it's value

1. Individual nodes can only write to the accumulator

While processing an RDD, the nodes can increment the accumulator

1. Individual nodes can only write to the accumulator

The increment is usually triggered by certain events that the user wants to track

Ex: Encountering an ERROR message

1. Individual nodes can only write to the accumulator

The individual nodes cannot read the accumulator's value

The main program is the one where we invoke the processing of an RDD

usually through an Action

2. The main program (or Spark shell) can only read it's value

This is called the driver program and the processing task a Job

2. The main program (or Spark shell) can only read it's value

At the end of a Job

the driver program can request the value of the accumulator

2. The main program (or Spark shell) can only read it's value

The driver program can only

1. Create the accumulator ;

2. Read it's value

2. The main program (or Spark shell) can only read it's value

The driver program cannot change the accumulator once created

2. The main program (or Spark shell) can only read it's value

Accumulator variable

Let's go back to our log example

```
logs=sc.textFile(logsPath)
```

This is the logs RDD

Accumulator variable

```
def processLog(line):

    dateField=line[:24]
    logField=line[24:]
    # some other processing
    #....
    #



    return (dateField,logField)
```

This is the function to parse the logs

Accumulator variable

```
def processLog(line):

    dateField=line[:24]
    logField=line[24:]
    # some other processing
    #....
    #


    return (dateField,logField)
```

It does a bunch of processing

The specifics don't really matter

Accumulator variable

```
def processLog(line):

    dateField=line[:24]
    logField=line[24:]
    # some other processing
    #....
    #


    return (dateField,logField)
```

We'll use this function to process the logs and save them to a file

```
logs.map(processLog).saveAsTextFile(processedLogsPath)
```

This triggers a Job on the Spark cluster

Accumulator variable

```
def processLog(line):

    dateField=line[:24]
    logField=line[24:]
    # some other processing
    #....
    #


    return (dateField,logField)
```

To print a count of ERROR messages in the log, we could trigger another Job

```
logs.map(processLog).saveAsTextFile(processedLogsPath)
```

Or we could use an accumulator instead

Accumulator variable

```
def processLog(line):

    dateField=line[:24]
    logField=line[24:]
    # some other processing
    #....
    #


    return (dateField,logField)
```

Let's create an accumulator variable

```
errCount=sc.accumulator(0)
```

The initial value of the accumulator

```
logs.map(processLog).saveAsTextFile(processedLogsPa
```

Accumulator variable

```
def processLog(line):

    dateField=line[:24]
    logField=line[24:]
    # some other processing
    #....
    #


    return (dateField,logField)
```

This variable will be shared and incremented by all the nodes

```
errCount=sc.accumulator(0)
```

```
logs.map(processLog).saveAsTextFile(processedLogsPa
```

Accumulator variable

```
def processLog(line):

    dateField=line[:24]
    logField=line[24:]
    # some other processing
    #....
    #


    return (dateField,logField)
```

The increment logic has to be part of the function that's used to process the RDD

```
errCount=sc.accumulator(0)
```

```
logs.map(processLog).saveAsTextFile(processedLogsPa
```

Accumulator variable

```
errCount=sc.accumulator(0)
```

```
def processLog(line):
    global errCount
    dateField=line[:24]
    logField=line[24:]
    # some other processing
    #....
    #
    if "ERROR" in line:
        errCount+=1
    return (dateField,logField)
```

This makes sure we are referring to the accumulator variable defined outside the function

```
logs.map(processLog).saveAsTextFile(processedLogsPa
```

Accumulator variable

```
errCount=sc.accumulator(0)
```

```
def processLog(line):
    global errCount
    dateField=line[:24]
    logField=line[24:]
    # some other processing
    #....
    #
    if "ERROR" in line:
        errCount+=1
    return (dateField,logField)
```

Whenever an ERROR message is encountered, the accumulator is incremented

```
logs.map(processLog).saveAsTextFile(processedLogsPa
```

Accumulator variable

```
errCount=sc.accumulator(0)
```

```
def processLog(line):
    global errCount
    dateField=line[:24]
    logField=line[24:]
    # some other processing
    #....
    #
    if "ERROR" in line:
        errCount+=1
    return (dateField,logField)
```

+=

In Python, this is the only operation allowed on accumulators

```
logs.map(processLog).saveAsTextFile(processedLogsPa
```

Accumulator variable

```
errCount=sc.accumulator(0)
```

```
def processLog(line):
    global errCount
    dateField=line[:24]
    logField=line[24:]
    # some other processing
    #....
    #
    if "ERROR" in line:
        errCount+=1
    return (dateField,logField)
```

+=

In Scala, the user can define custom accumulators with other operations

```
logs.map(processLog).saveAsTextFile(processedLogsPa
```

Accumulator variable

```
errCount=sc.accumulator(0)
```

```
def processLog(line):
    global errCount
    dateField=line[:24]
    logField=line[24:]
    # some other processing
    #....
    #
    if "ERROR" in line:
        errCount+=1
    return (dateField,logField)
```

After the Job is completed, you can access the value of the accumulator

```
logs.map(processLog).saveAsTextFile(processedLogsPath)
```

Accumulator variable

```
errCount=sc.accumulator(0)
```

```
def processLog(line):
    global errCount
    dateField=line[:24]
    logField=line[24:]
    # some other processing
    #....
    #
    if "ERROR" in line:
        errCount+=1
    return (dateField,logField)
```

After the Job is completed, you can access the value of the accumulator

```
logs.map(processLog).saveAsTextFile(processedLogsPath)
```

```
print errCount.value
```

```
15
```

SPARK-SUBMIT

So far, we have only used the
Spark REPL environment

This is great for experimentation
and fast feedback

Once you've developed a program/
application to do specific tasks

You can submit this program
to be run on the Spark cluster

In Python - you can submit your
Python scripts

For Java and Scala, you'll need
to build and submit JAR files
with a main function

Spark has a tool for
running all of these

spark-submit

spark-submit

```
>spark-submit Logs.py
```

This will run the Logs.py script on the Spark cluster

Logs.py

```python
from pyspark import SparkConf, SparkContext

conf=SparkConf().setMaster("yarn-client").setAppName("My App")
sc=SparkContext(conf=conf)

logsPath="hdfs:///user/swethakolalapudi/log/hbase.log"

logs=sc.textFile(logsPath)

errCount=sc.accumulator(0)

def processLog(line):
    global errCount
    dateField=line[:24]
    logField=line[24:]
    # some other processing
    if "ERROR" in line:
        errCount+=1
    return (dateField,logField)

logs.map(processLog).saveAsTextFile("hdfs:///user/swethakolalapudi/log/processedLogs.log")

print "There were " + str(errCount.value)+" ERROR lines"
```

This is a script to process some log files

Logs.py

```
from pyspark import SparkConf, SparkContext

conf=SparkConf().setMaster("yarn-client").setAppName("My App")
sc=SparkContext(conf=conf)

logsPath="hdfs:///user/swethakolalapudi/log/hbase.log"

logs=sc.textFile(logsPath)

errCount=sc.accumulator(0)

def processLog(line):
    global errCount
    dateField=line[:24]
    logField=line[24:]
    # some other processing
    if "ERROR" in line:
        errCount+=1
    return (dateField,logField)

logs.map(processLog).saveAsTextFile("hdfs:///user/swethakolalapudi/log/processedLogs.log")

print "There were " + str(errCount.value)+" ERROR lines"
```

This part is exactly the same as what you write in the PySpark shell

Logs.py

```
from pyspark import SparkConf, SparkContext

conf=SparkConf().setMaster("yarn-client").setAppName("My App")
sc=SparkContext(conf=conf)

logsPath="hdfs:///user/swethakolalapudi/log/hbase.log"

logs=sc.textFile(logsPath)

errCount=sc.accumulator(0)

def processLog(line):
    global errCount
    dateField=line[:24]
    logField=line[24:]
    # some other processing
    if "ERROR" in line:
        errCount+=1
    return (dateField,logField)

logs.map(processLog).saveAsTextFile("hdfs:///user/swethakolalapudi/log/processedLogs.log")

print "There were " + str(errCount.value)+" ERROR lines"
```

This is a bit of setup that you need for your script to run

Logs.py

```
from pyspark import SparkConf, SparkContext

conf=SparkConf().setMaster("yarn-client").setAppName("My App")
sc=SparkContext(conf=conf)

logsPath="hdfs:///user/swethakolalapudi/log/hbase.log"

logs=sc.textFile(logsPath)

errCount=sc.accumulator(0)

def processLog(line):
    global errCount
    dateField=line[:24]
    logField=line[24:]
    # some other processing
    if "ERROR" in line:
        errCount+=1
    return (dateField,logField)

logs.map(processLog).saveAsTextFile("hdfs:///user/swethakolalapudi/log/processedLogs.log")

print "There were " + str(errCount.value)+" ERROR lines"
```

This is a bit of setup that you need for your script to run

Logs.py

```
from pyspark import SparkConf, SparkContext

conf=SparkConf().setMaster("yarn-client").setAppName("My App")
sc=SparkContext(conf=conf)
```

```
logsPath="hdfs:///user/swethakolalapudi/log/hbase.log"

logs=sc.textFile(logsPath)

errCount=sc.accumulator(0)

def processLog(line):
    global errCount
    dateField=line[:24]
    logField=line[24:]
    # some other processing
    if "ERROR" in line:
```

Here we are setting up the SparkContext

Logs.py

```
from pyspark import SparkConf, SparkContext

conf=SparkConf().setMaster("yarn-client").setAppName("My App")
sc=SparkContext(conf=conf)
```

```
logsPath="hdfs:///user/swethakolalapudi/log/hbase.log"

logs=sc.textFile(logsPath)

errCount=sc.accumulator(0)

def processLog(line):
    global errCount
    dateField=line[:24]
    logField=line[24:]
    # some other processing
    if "ERROR" in line:
```

The SparkContext represents a connection to the Spark cluster

Logs.py

```
from pyspark import SparkConf, SparkContext

conf=SparkConf().setMaster("yarn-client").setAppName("My App")
sc=SparkContext(conf=conf)
```

```
logsPath="hdfs:///user/swethakolalapudi/log/hbase.log"

logs=sc.textFile(logsPath)

errCount=sc.accumulator(0)

def processLog(line):
    global errCount
    dateField=line[:24]
    logField=line[24:]
    # some other processing
    if "ERROR" in line:
```

Any Spark application has to start with setting up a SparkContext

Logs.py

```
from pyspark import SparkConf, SparkContext

conf=SparkConf().setMaster("yarn-client").setAppName("My App")
sc=SparkContext(conf=conf)
```

```
logsPath="hdfs:///user/vgethakolalapudi/log/hbase.log"

logs=sc.textFile(logsPath)

errCount=sc.accumulator(0)

def processLog(line):
    global errCount
    dateField=line[:24]
    logField=line[24:]
    # some other processing
    if "ERROR" in line:
```

When we use PySpark, the SparkContext is set up for us

Logs.py

```
from pyspark import SparkConf, SparkContext

conf=SparkConf().setMaster("yarn-client").setAppName("My App")
sc=SparkContext(conf=conf)

logsPath="hdfs:///user/swethakolalapudi/log/hbase.log"

logs=sc.textFile(logsPath)

errCount=sc.accumulator(0)

def processLog(line):
    global errCount
    dateField=line[:24]
    logField=line[24:]
    # some other processing
    if "ERROR" in line:
```

In a script/program, we have to set it up ourselves

Logs.py

```
from pyspark import SparkConf, SparkContext

conf=SparkConf().setMaster("yarn-client").setAppName("My App")
sc=SparkContext(conf=conf)

logsPath="hdfs:///user/swethakolalapudi/log/hbase.log"

logs=sc.textFile(logsPath)

errCount=sc.accumulator(0)

def processLog(line):
    global errCount
    dateField=line[:24]
    logField=line[24:]
    # some other processing
    if "ERROR" in line:
```

We start by importing SparkConf and SparkContext

Logs.py

```
from pyspark import SparkConf, SparkContext

conf=SparkConf().setMaster("yarn-client").setAppName("My App")
sc=SparkContext(conf=conf)

logsPath="hdfs:///user/swethakolalapudi/log/hbase.log"

logs=sc.textFile(logsPath)

errCount=sc.accumulator(0)

def processLog(line):
    global errCount
    dateField=line[:24]
    logField=line[24:]
    # some other processing
    if "ERROR" in line:
```

The SparkConf helps us specify the parameters of the Spark Connection

Logs.py

```
from pyspark import SparkConf, SparkContext

conf=SparkConf().setMaster("yarn-client").setAppName("My App")
sc=SparkContext(conf=conf)
```

```
logsPath="hdfs:///user/swethakolalapudi/log/hbase.log"
logs=sc.textFile(logsPath)

errCount=sc.accumulator(0)

def processLog(line):
    global errCount
    dateField=line[:24]
    logField=line[24:]
    # some other processing
    if "ERROR" in line:
```

```
> pyspark --master yarn-client
```

setMaster is similar to the master option when we initialize PySpark

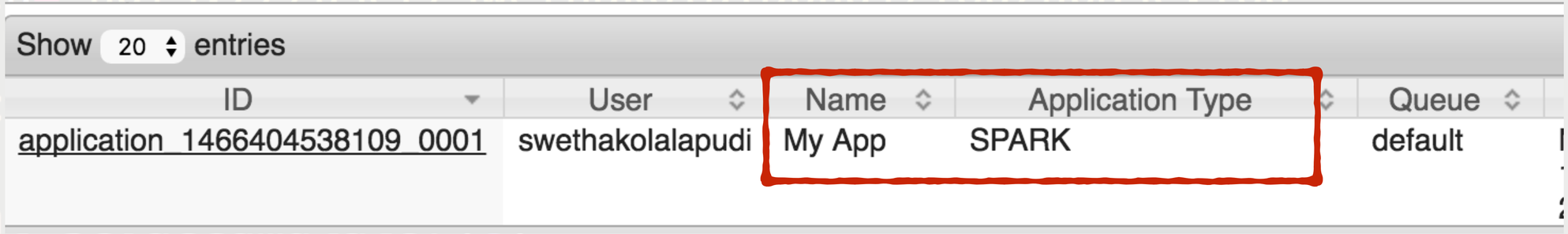
Logs.py

```
from pyspark import SparkConf, SparkContext

conf=SparkConf().setMaster("yarn-client").setAppName("My App")
sc=SparkContext(conf=conf)

logsPath="hdfs:///user/swethakolalapudi/log/hbase.log"
logs=sc.textFile(logsPath)

errCount=sc.accumulator(0)

def processLog(line):
    global errCount
    dateField=line[:24]
    logField=line[24:]
    # some other processing
    if "ERROR" in line:
```

This will be the name of our application

Logs.py

```
from pyspark import SparkConf, SparkContext

conf=SparkConf().setMaster("yarn-client").setAppName("My App")
sc=SparkContext(conf=conf)
```



This is how this application will appear on the YARN web interface

Logs.py

```
from pyspark import SparkConf, SparkContext

conf=SparkConf().setMaster("yarn-client").setAppName("My App")
sc=SparkContext(conf=conf)

logsPath="hdfs:///user/swethakolalapudi/log/hbase.log"

logs=sc.textFile(logsPath)

errCount=sc.accumulator(0)

def processLog(line):
    global errCount
    dateField=line[:24]
    logField=line[24:]
    # some other processing
    if "ERROR" in line:
```

This initializes the SparkContext

Logs.py

```
from pyspark import SparkConf, SparkContext

conf=SparkConf().setMaster("yarn-client").setAppName("My App")
sc=SparkContext(conf=conf)

logsPath="hdfs:///user/swethakolalapudi/log/hbase.log"

logs=sc.textFile(logsPath)

errCount=sc.accumulator(0)

def processLog(line):
    global errCount
    dateField=line[:24]
    logField=line[24:]
    # some other processing
    if "ERROR" in line:
        errCount+=1
    return (dateField,logField)

logs.map(processLog).saveAsTextFile("hdfs:///user/swethakolalapudi/log/processedLogs.log")

print "There were " + str(errCount.value)+" ERROR lines"
```

Now, you can write your code exactly as you would in the PySpark shell

spark-submit

```
>spark-submit Logs.py
```

Then you can run your script on the Spark cluster

SPARK EXECUTION

So far, we've interacted with Spark

1. Using an REPL environment
2. Submitting a script

These are ways for the user to give instructions to Spark

Spark takes the user's instructions

It translates them into tasks that run across a computing cluster

It returns the results to the user

How does this work?

When you run a program (using PySpark or spark-submit)

Spark prints a bunch of messages

First, there are a lot of messages initializing things

```
INFO util.Utils: Successfully started service 'sparkDriver' on port 64191.
INFO slf4j.Slf4jLogger: Slf4jLogger started
INFO Remoting: Starting remoting
INFO Remoting: Remoting started; listening on addresses :[akka.tcp://sparkDriverActorSyst
INFO util.Utils: Successfully started service 'sparkDriverActorSystem' on port 64192.
INFO spark.SparkEnv: Registering MapOutputTracker
INFO spark.SparkEnv: Registering BlockManagerMaster
```

Then you can see the progress of your processing instructions

```
spark.SparkContext: Starting job: saveAsTextFile at
scheduler.DAGScheduler: Got job 0 (saveAsTextFile at
scheduler.DAGScheduler: Submitting ResultStage 0 (MapPar
cluster.YarnScheduler: Adding task set 0.0 with 2 tasks
scheduler.TaskSetManager: Starting task 0.0 in stage 0.0
scheduler.TaskSetManager: Starting task 1.0 in stage 0.0
scheduler.TaskSetManager: Finished task 1.0 in stage 0.0
scheduler.TaskSetManager: Finished task 0.0 in stage 0.0
scheduler.DAGScheduler: ResultStage 0 (saveAsTextFile at
scheduler.DAGScheduler: Job 0 finished: saveAsTextFile
```

There seem to be many components involved in the processing

```
spark.SparkContext: Starting job: saveAsTextFile at
scheduler.DAGScheduler: Got job 0 (saveAsTextFile at
scheduler.DAGScheduler: Submitting ResultStage 0 (MapPar
cluster.YarnScheduler: Adding task set 0.0 with 2 tasks
scheduler.TaskSetManager: Starting task 0.0 in stage 0.0
scheduler.TaskSetManager: Starting task 1.0 in stage 0.0
scheduler.TaskSetManager: Finished task 1.0 in stage 0.0
scheduler.TaskSetManager: Finished task 0.0 in stage 0.0
scheduler.DAGScheduler: ResultStage 0 (saveAsTextFile at
scheduler.DAGScheduler: Job 0 finished: saveAsTextFile
```

What are all of these components doing?

SparkContext

DAGScheduler

YarnScheduler

TaskSetManager

SparkContext

Every Spark application has a driver program

DAGScheduler

YarnScheduler

TaskSetManager

SparkContext

driver program

This could be

The PySpark/Scala shell

A script

A main function
(Java/Scala)

DAGScheduler

YarnScheduler

TaskSetManager

SparkContext

driver program

This program consists of instructions to Spark

Load data into RDDs

Perform operations on RDDS

DAGScheduler

YarnScheduler

TaskSetManager

driver program

SparkContext

Driver programs use a SparkContext to communicate with the Spark cluster

DAGScheduler

YarnScheduler

TaskSetManager

driver program

SparkContext

There are 2 phases involved in the driver program

Initial setup

Job Run

DAGScheduler

YarnScheduler

TaskSetManager

driver program

SparkContext

Initial setup

This happens when you launch the program (the shell or your script)

Job Run

DAGScheduler

YarnScheduler

TaskSetManager

driver program

SparkContext

Initial setup

A Job run is initiated whenever the program has a processing task

Job Run

Ex: An Action like count() or collect()

DAGScheduler

TaskScheduler

TaskSetManager

Initial setup

driver program

SparkContext

Spark needs a separate cluster manager to manage resources across the cluster

This is a plug and play component : Mesos, YARN or Spark Standalone

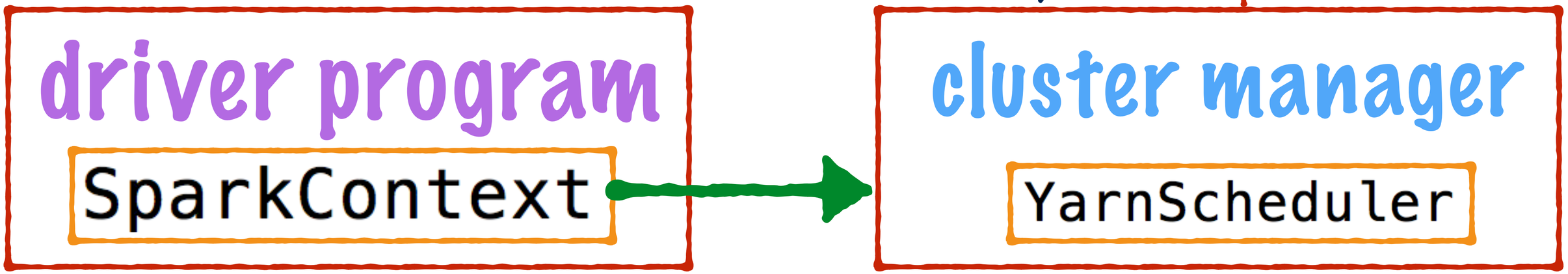
DAGScheduler

YarnScheduler

TaskSetManager

Initial setup

Mesos, YARN or Spark Standalone



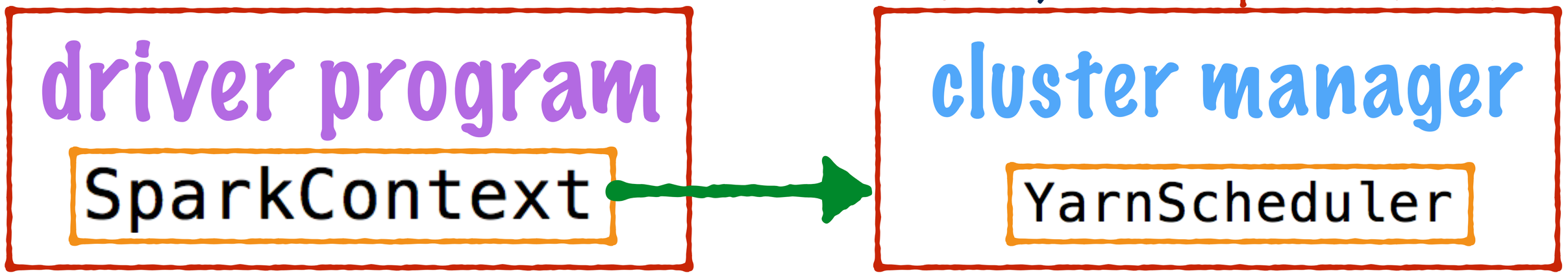
When the driver program starts

It contacts the cluster manager through the SparkContext

DAGScheduler

TaskSetManager

Initial setup



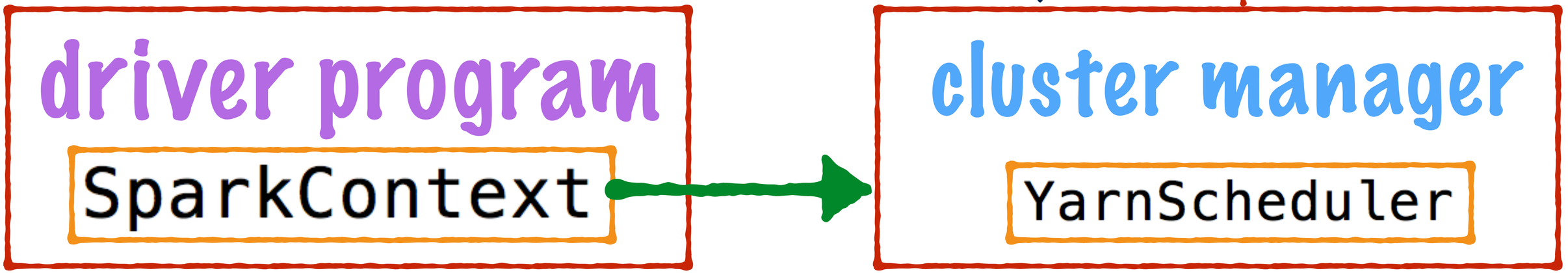
The cluster manager in turn launches Java processes on several nodes in the cluster

DAGScheduler

TaskSetManager

Initial setup

Mesos, YARN or Spark Standalone



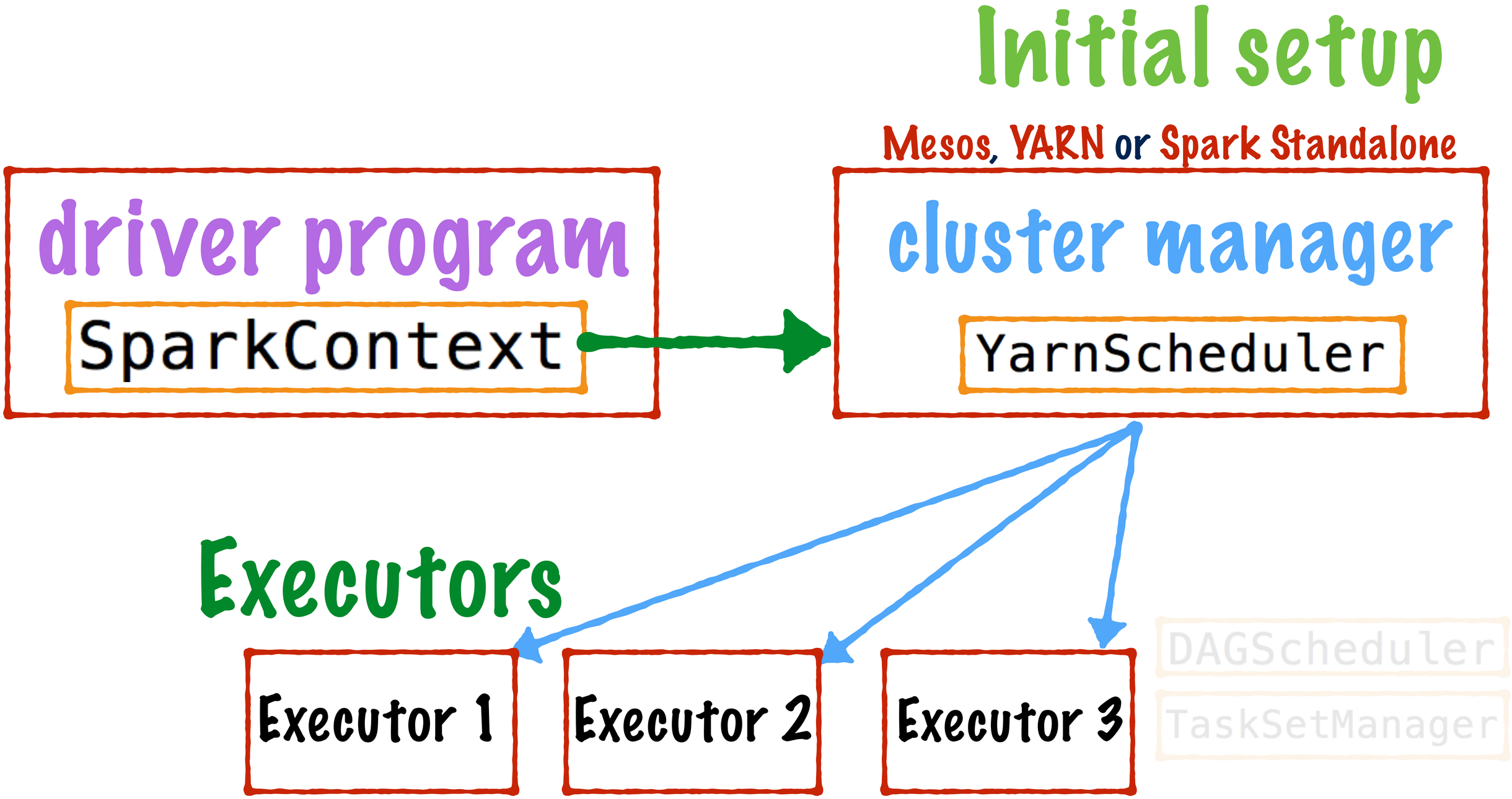
Executors

Java processes

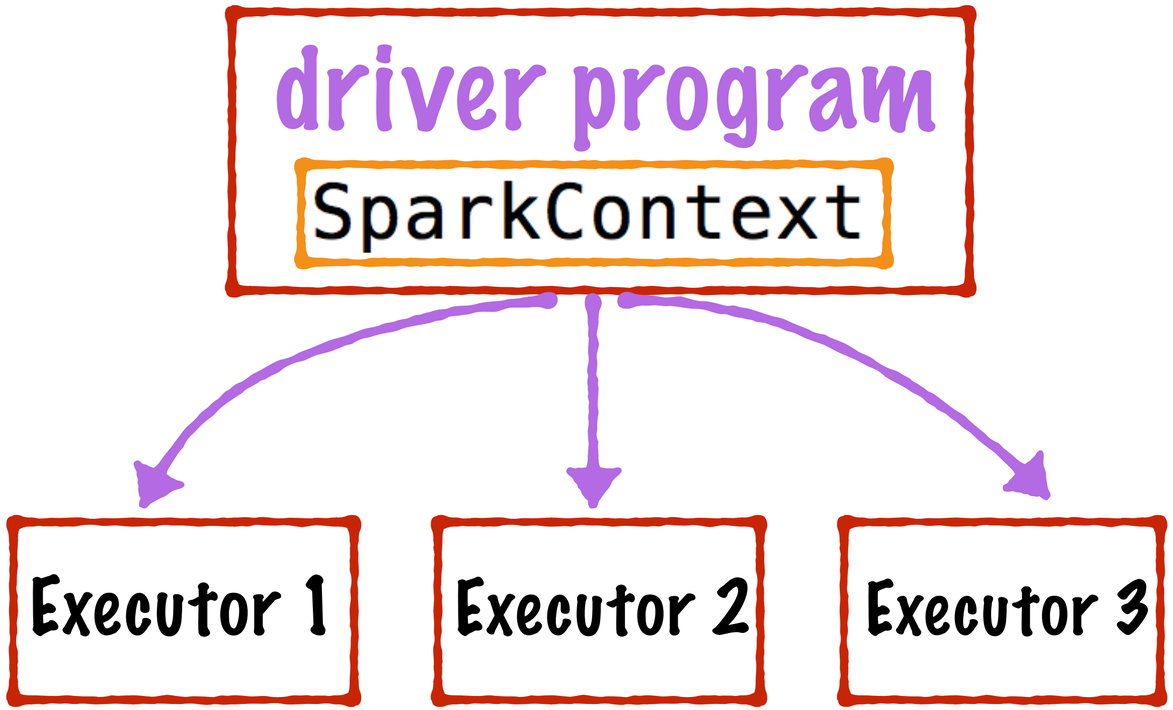
The cluster manager in turn launches on several nodes in the cluster

DAGScheduler

TaskSetManager


Initial setup
Mesos, YARN or Spark Standalone
driver program
SparkContext
cluster manager
YarnScheduler
Executors
Executor 1
Executor 2
Executor 3
DAGScheduler
TaskSetManager

Initial setup

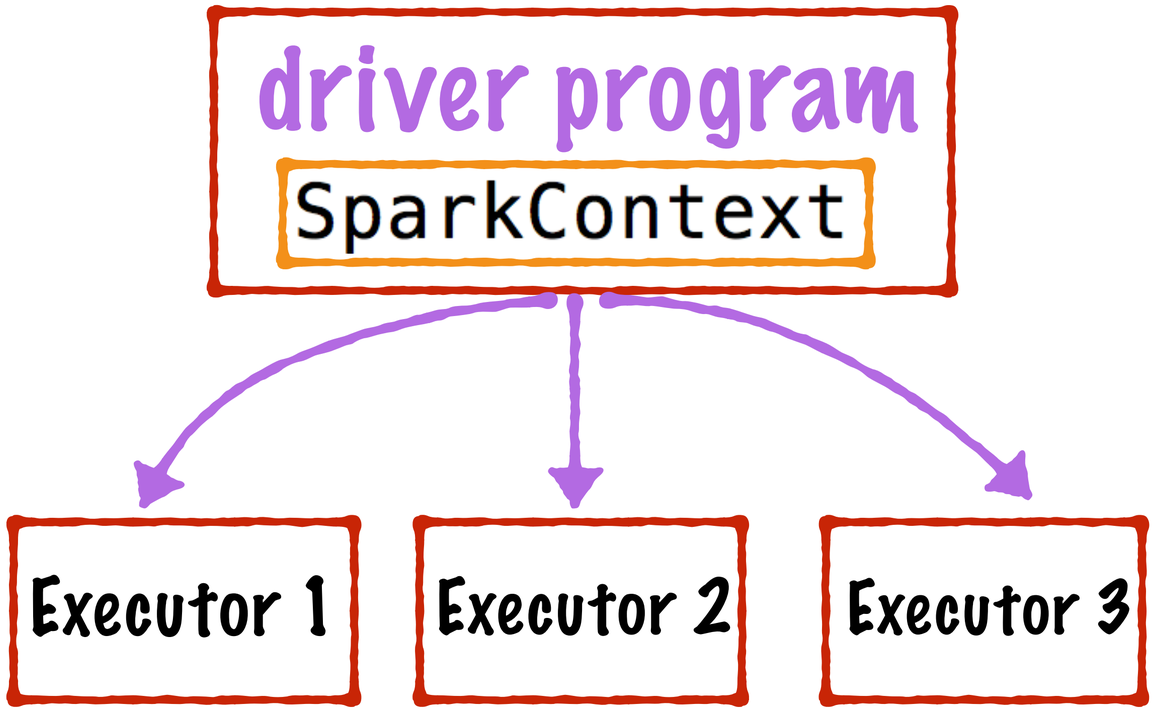


DAGScheduler

TaskSetManager

Once the executors are launched they register themselves with the driver program

Initial setup



DAGScheduler

TaskSetManager

The driver program is now ready for processing user instructions

driver program

SparkContext

Initial setup ✓

Job Run

DAGScheduler

TaskSetManager

Job Run

Whenever there is an action
(or some processing) on an RDD

Executor 1

Executor 2

Executor 3

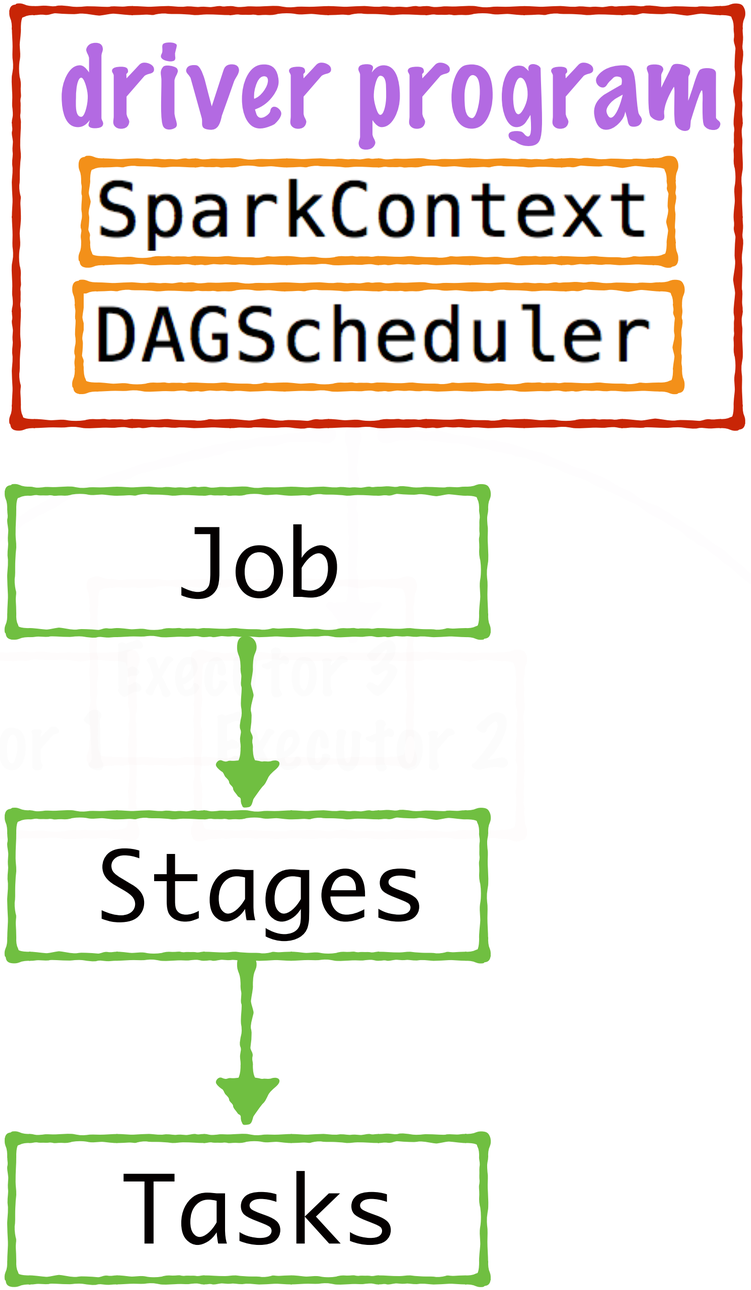
A Job is initiated

When this happens,
SparkContext passes the user
instructions on to a Scheduler

DAGScheduler

TaskSetManager

Job Run



The scheduler breaks down the Job into smaller units of work

Job Run

DAGScheduler

Job

Stages

Tasks

```
flightsParsed.map(lambda x:x.distance).reduce(lambda x,y:x+y)
```

Stage 0

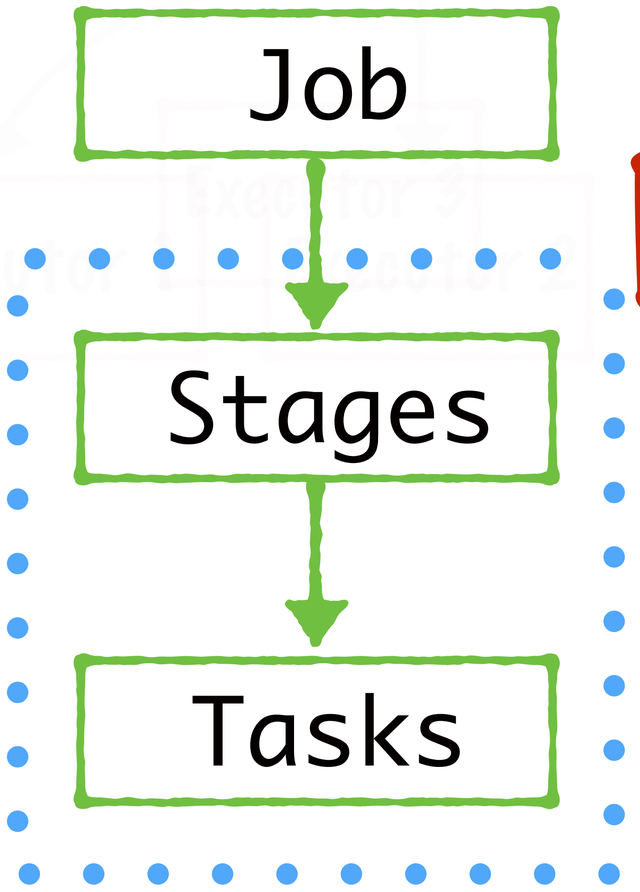
```
map(lambda x:x.distance)
```

Stage 1

```
reduce(lambda x,y:x+y)
```

Tasks on individual partitions of the RDD

Job Run



DAG

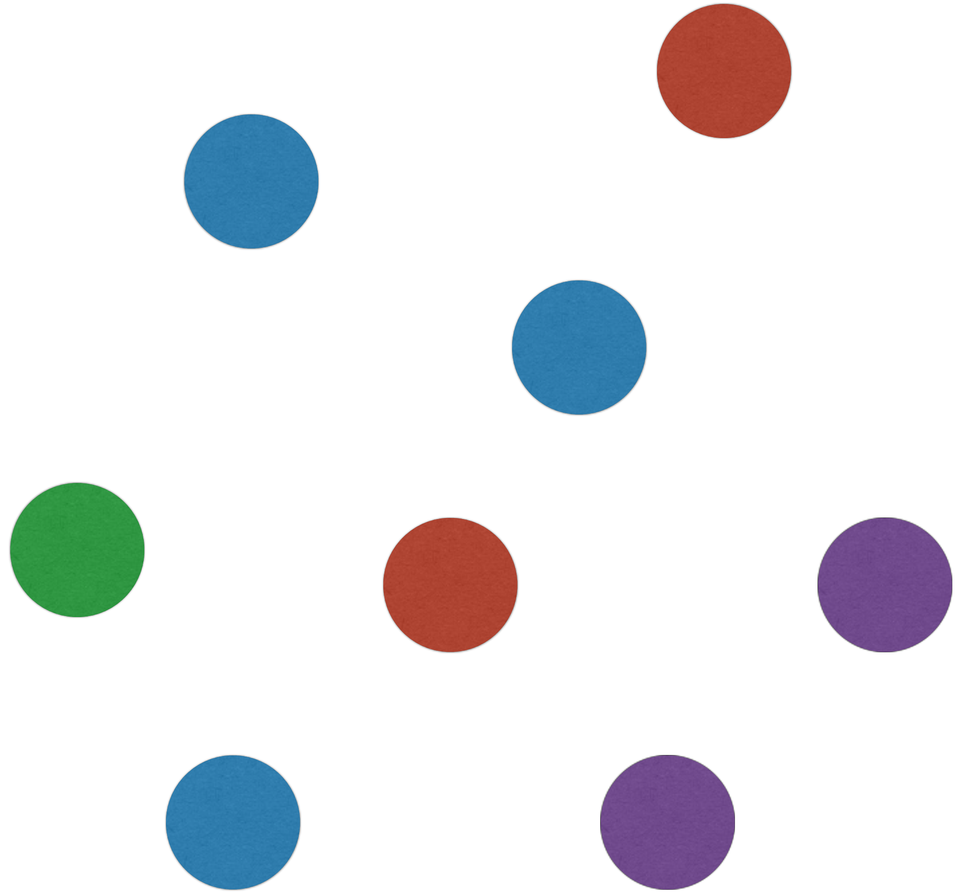
The Stages and Tasks form
a Directed Acyclic Graph

This is a standard way
to represent a workflow

Directed Acyclic Graph

In any workflow

You might have a bunch of tasks

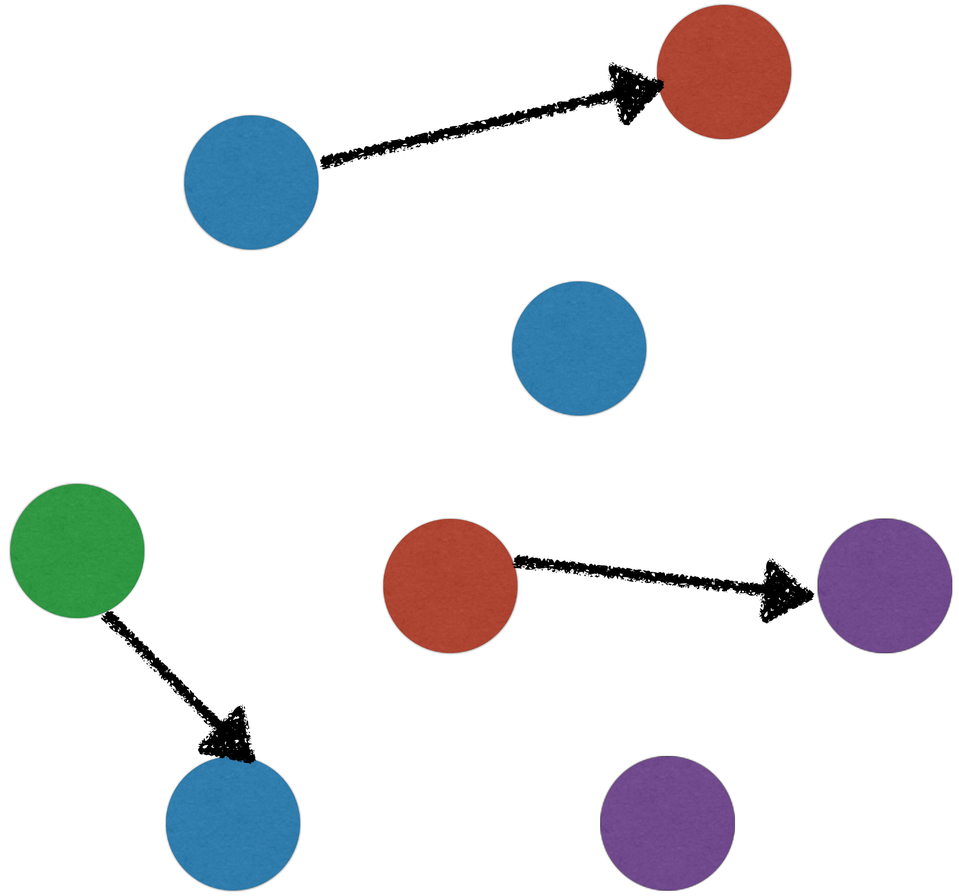


Directed Acyclic Graph

Some tasks are independent

They can be done in parallel

Ex: Applying a map operation on individual partitions of an RDD



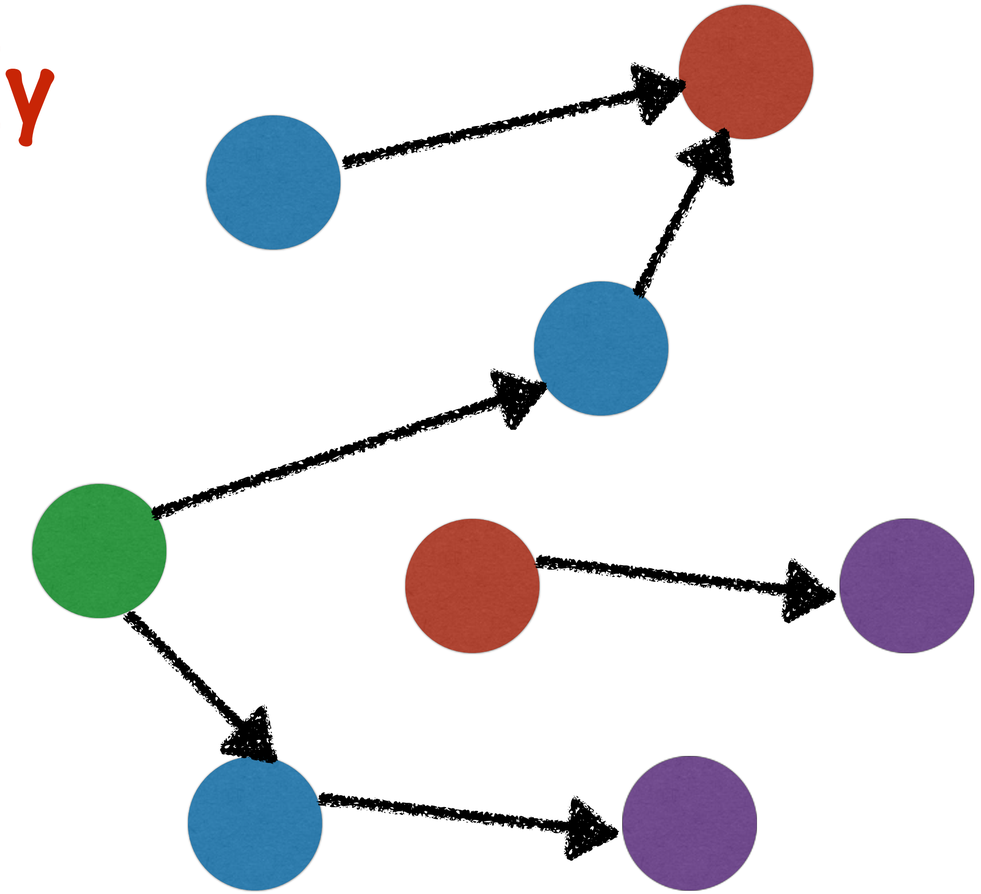
Directed Acyclic Graph

Some tasks are
dependent on each other

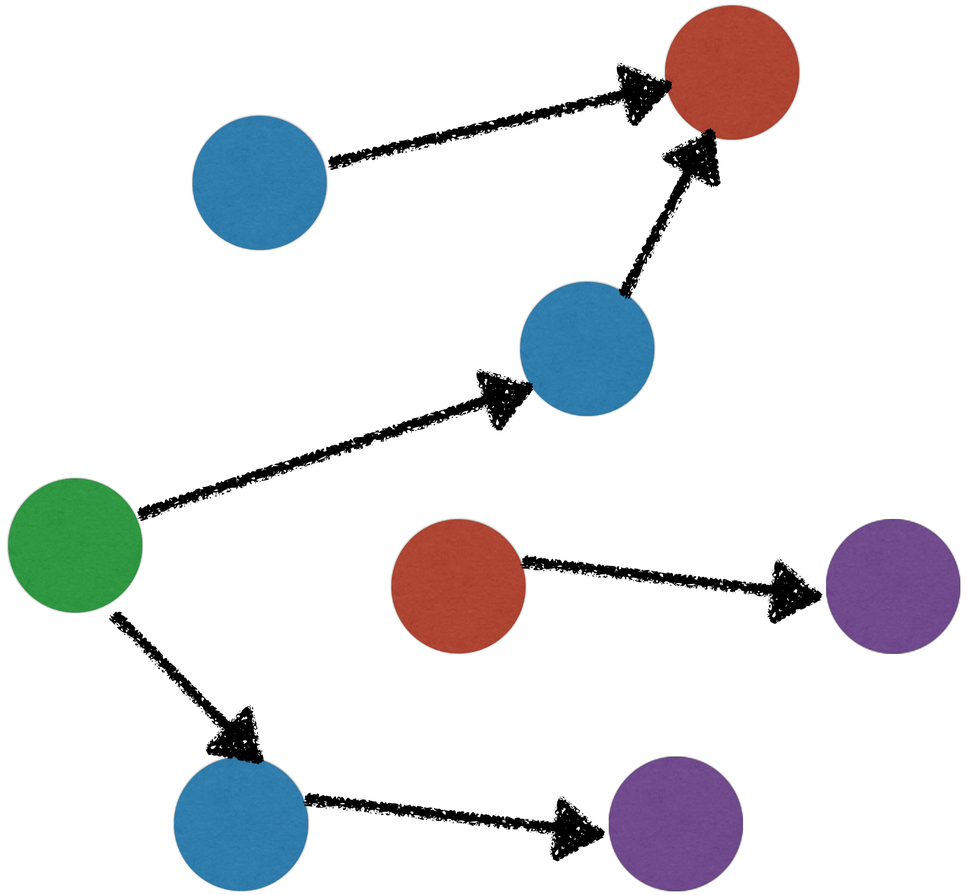
They have to be performed serially

Ex: In reduce,

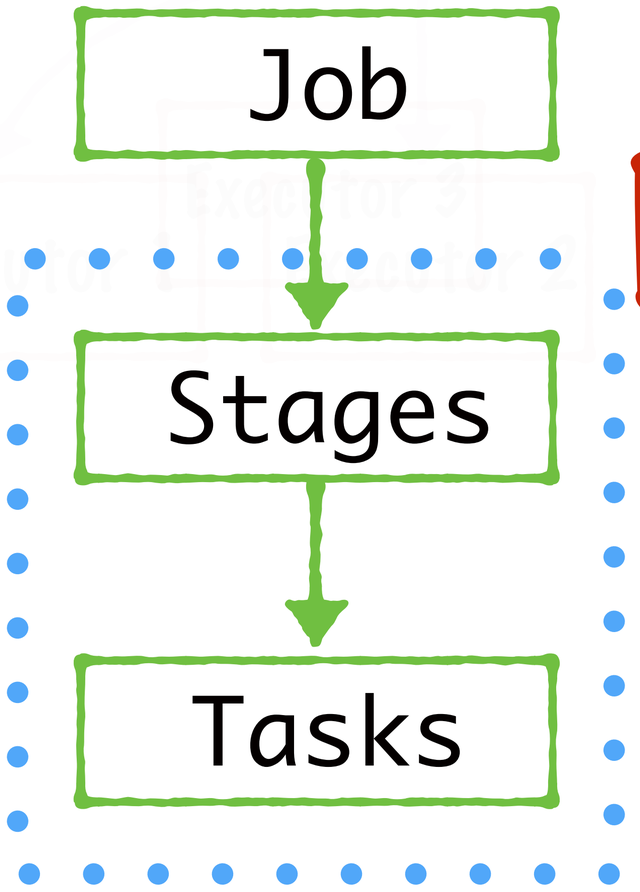
1. individual partitions have to be processed
2. Those results are combined



Directed Acyclic Graph



Job Run



DAG

The Stages and Tasks form
a Directed Acyclic Graph

This is a standard way
to represent a workflow

driver program

SparkContext

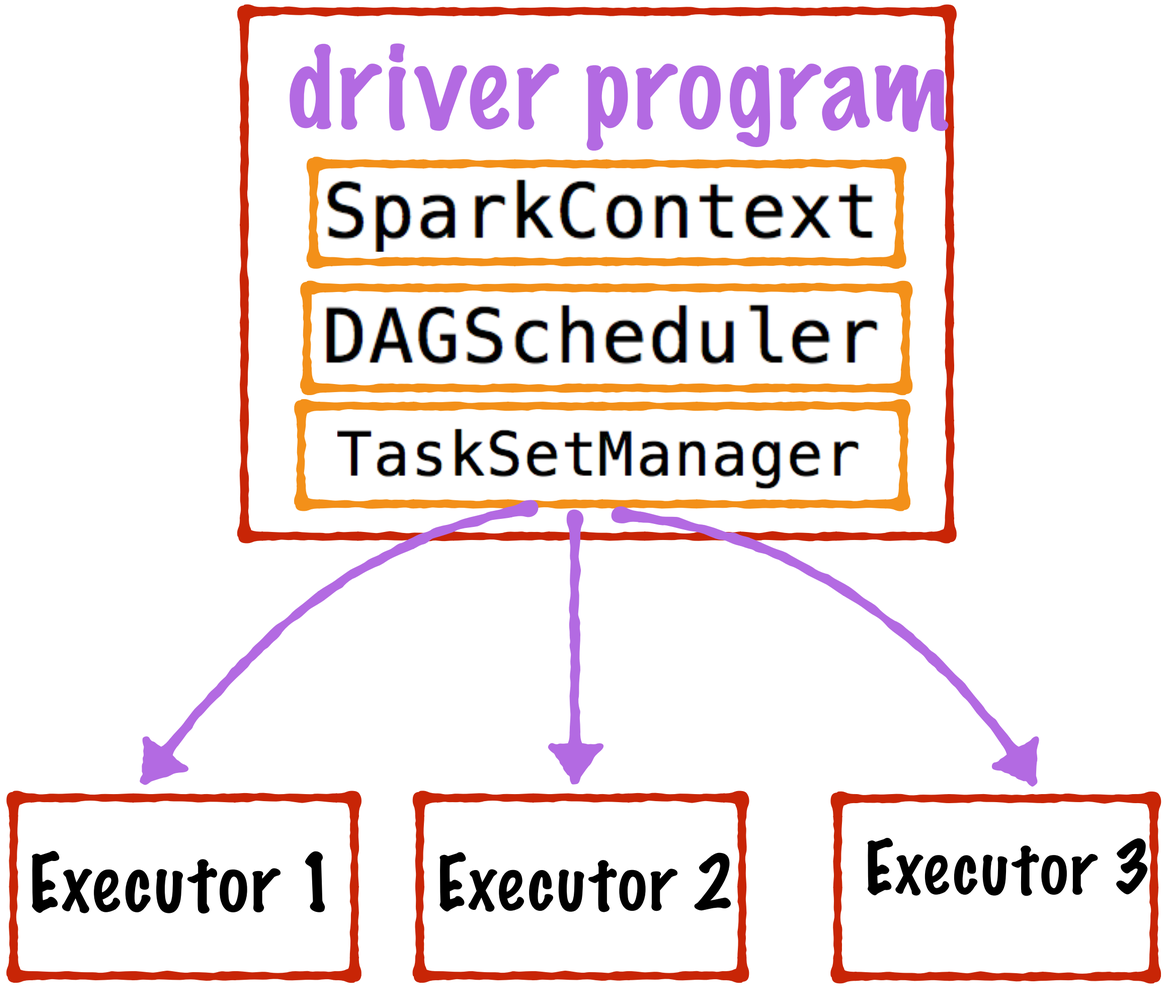
DAGScheduler

Job Run

Tasks

Tasks are the smallest unit of work in a Spark Job

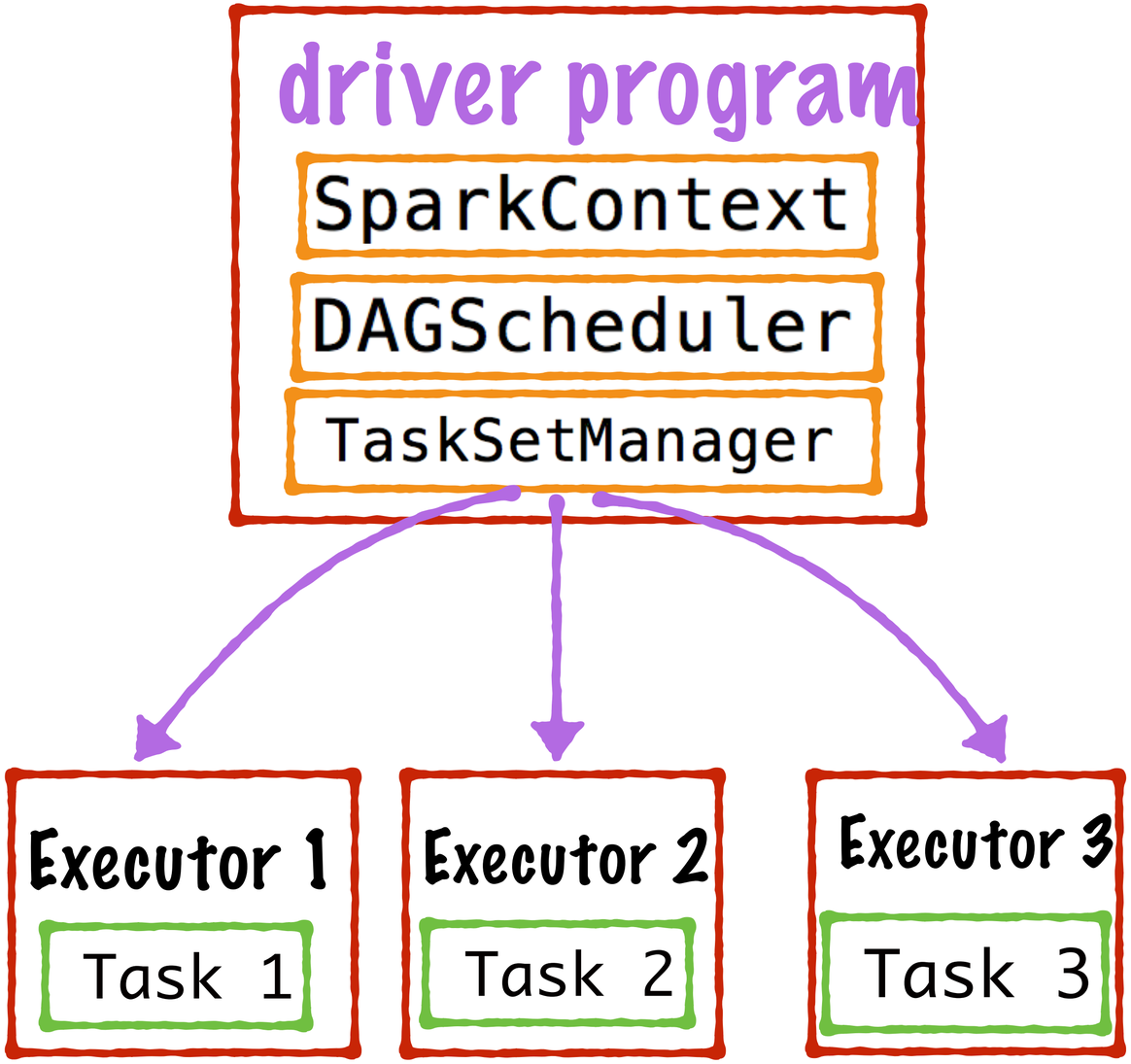
Each task is assigned to an Executor by the Task scheduler



Job Run

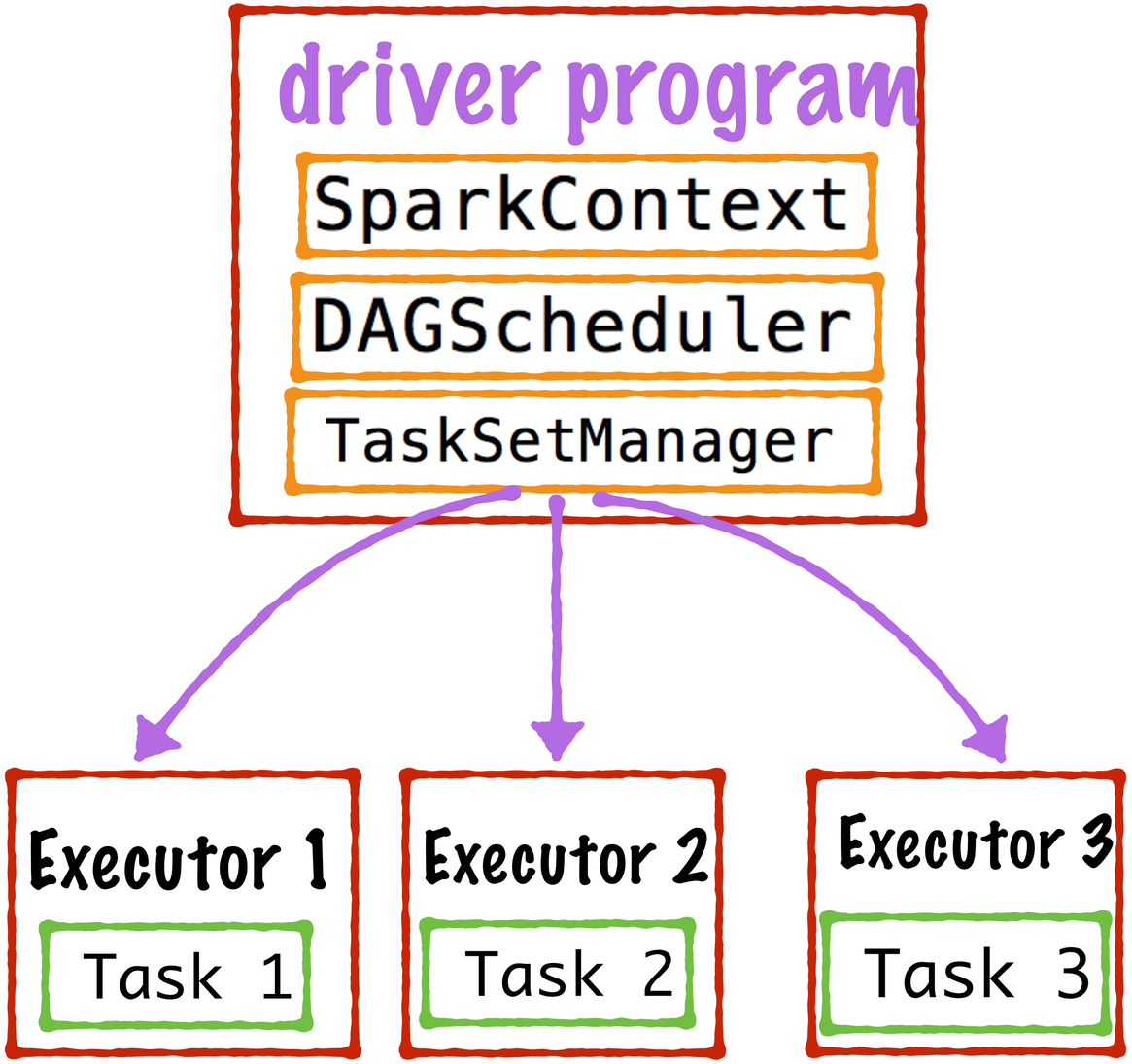
Tasks

Each task is assigned to an Executor by the Task scheduler



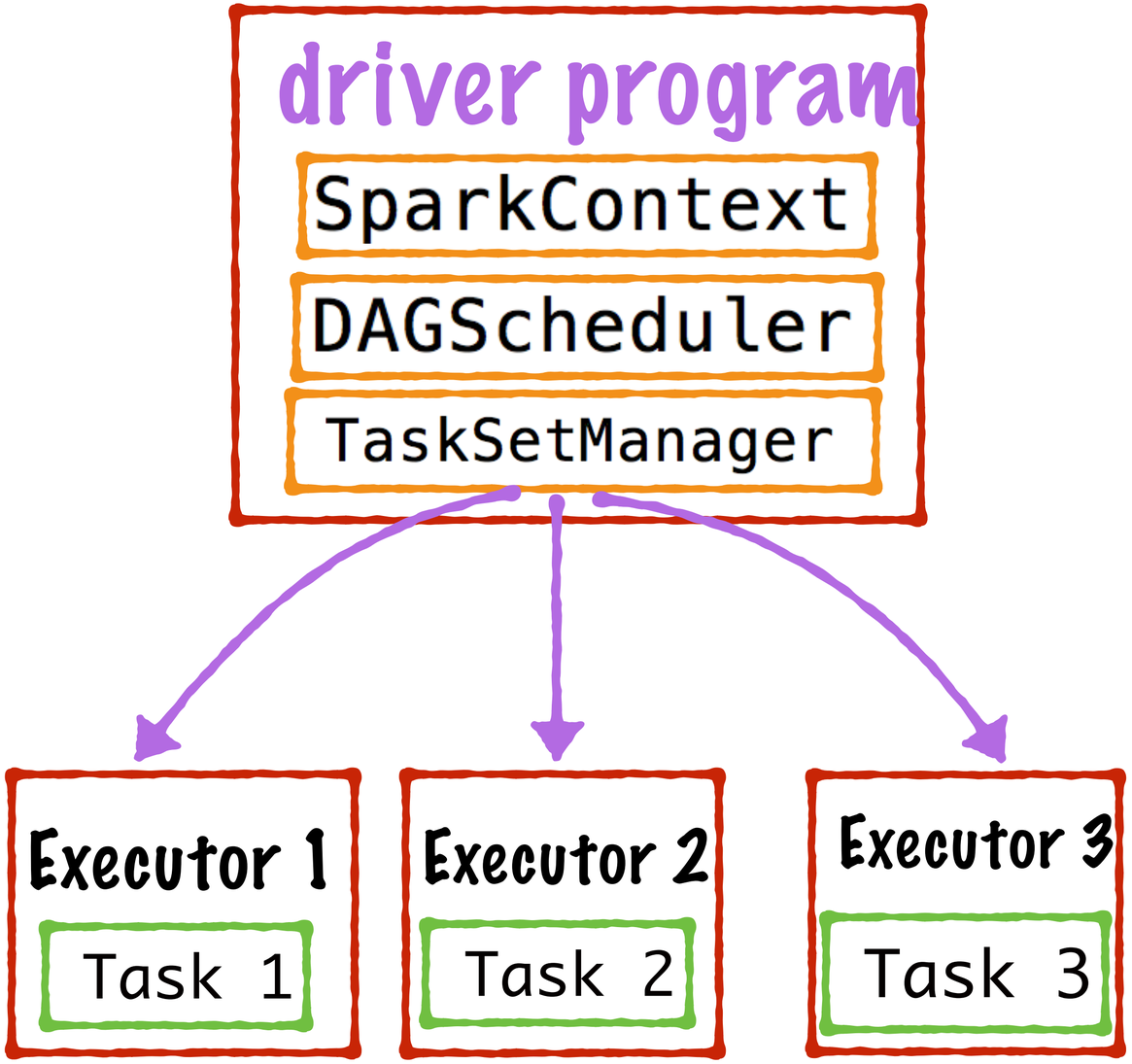
Job Run

The executors run the tasks and send updates back to the driver program



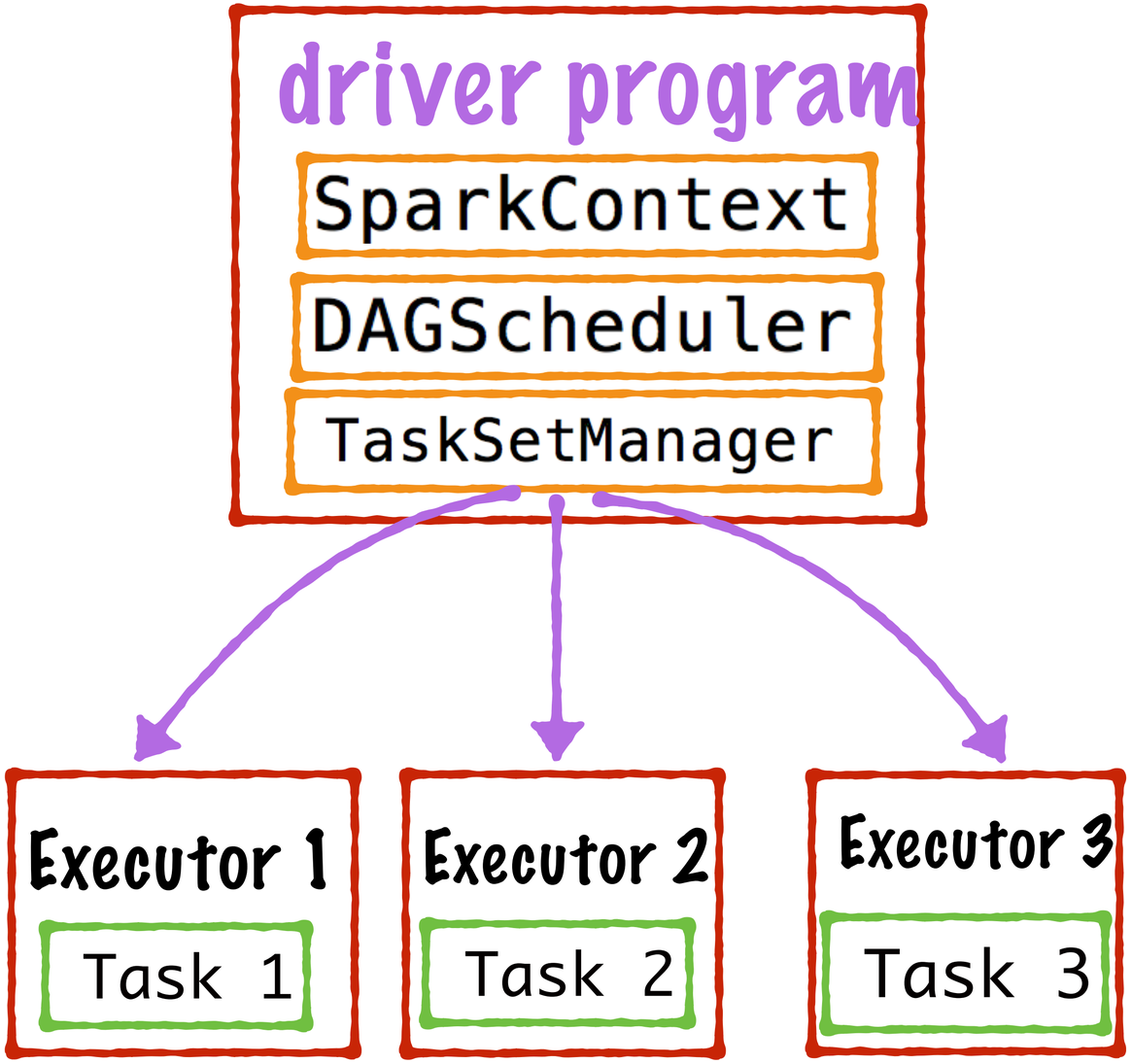
Job Run

When all the tasks in a Stage are completed, the next Stage is launched



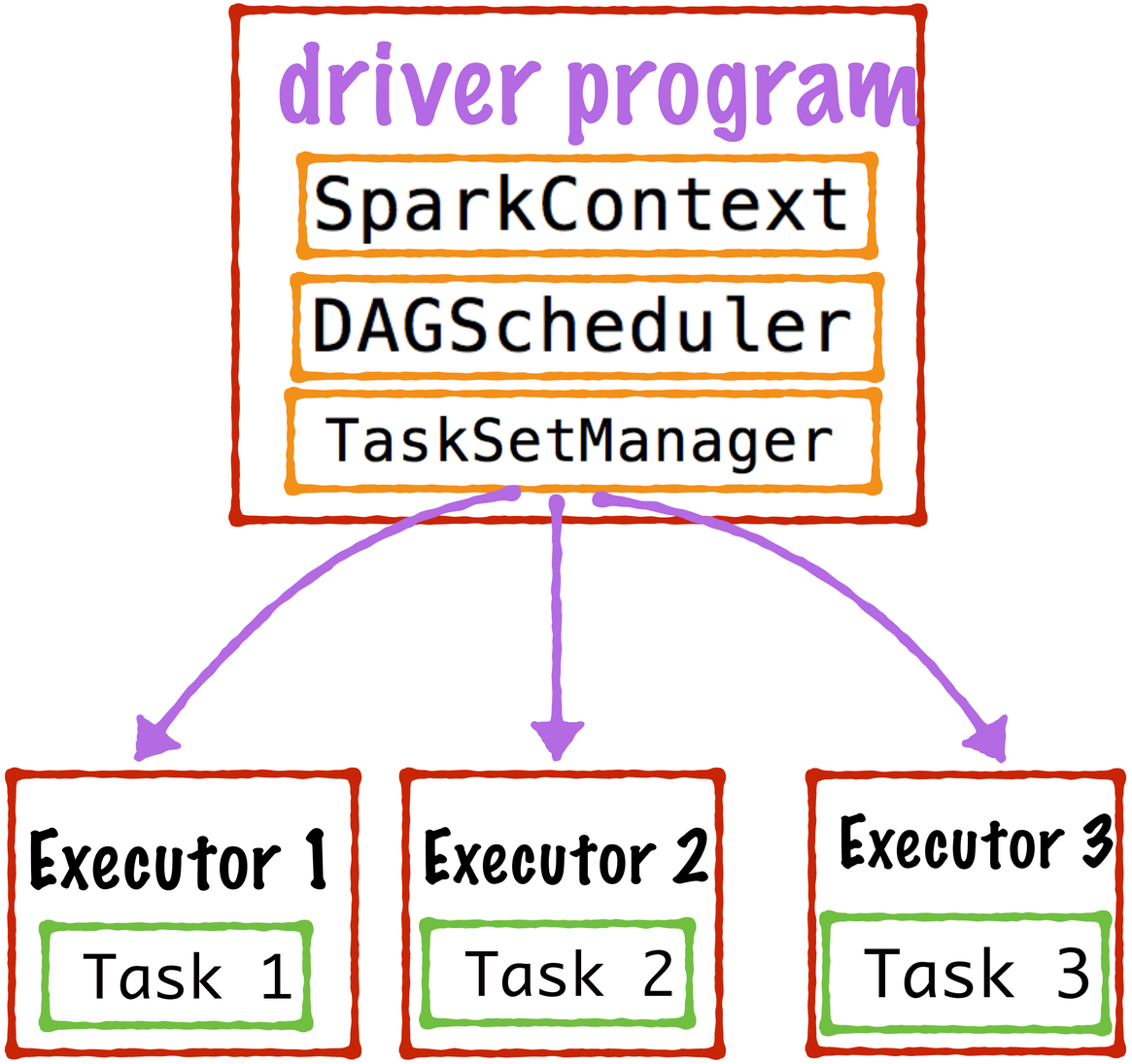
Job Run

When all the Stages are completed, the Job finishes



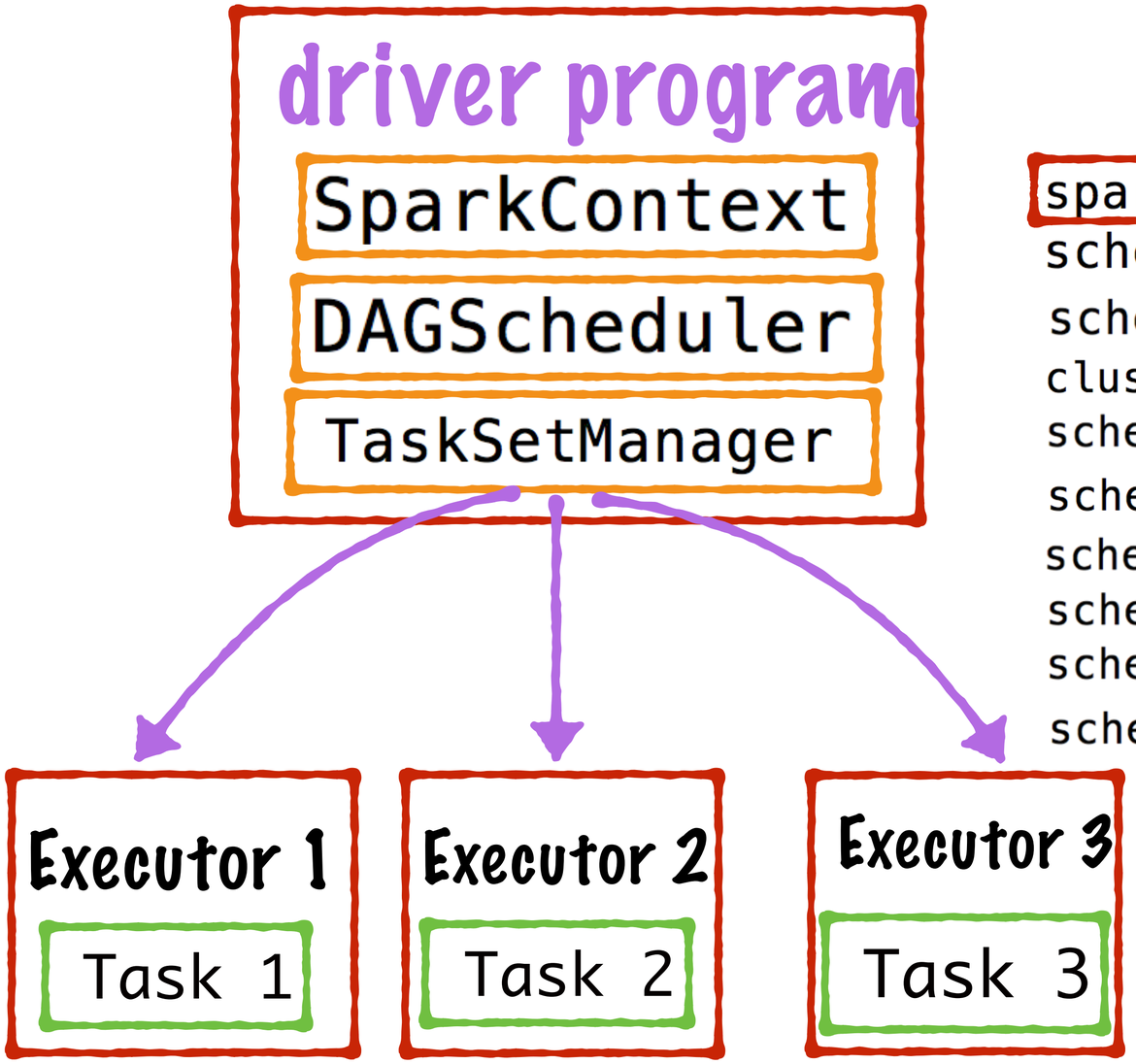
Job Run

Let's go back and look at the status messages Spark prints



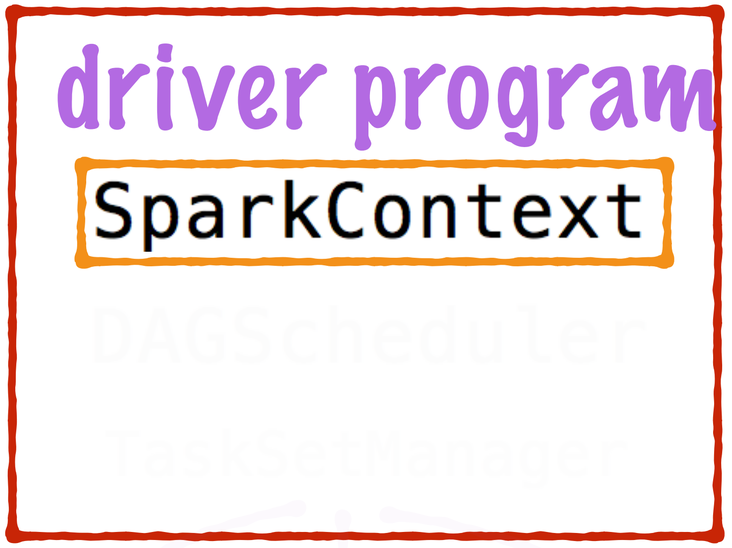
Job Run

Let's go back and look at the status messages Spark prints



Job Run

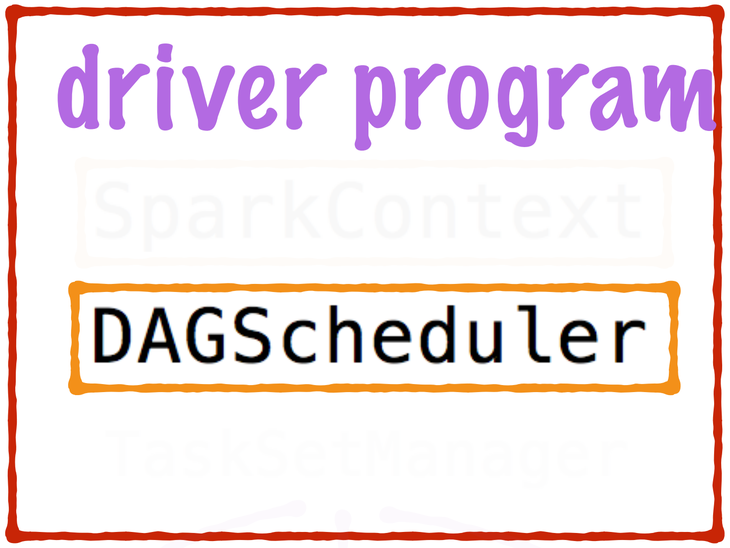
```
spark.SparkContext: Starting job: saveAsTextFile
scheduler.DAGScheduler: Got job 0 (saveAsTextFil
scheduler.DAGScheduler: Submitting ResultStage 0 (
cluster.YarnScheduler: Adding task set 0.0 with 2 tasks
scheduler.TaskSetManager: Starting task 0.0 in stage 0.
scheduler.TaskSetManager: Starting task 1.0 in stage 0.
scheduler.TaskSetManager: Finished task 1.0 in stage 0.
scheduler.TaskSetManager: Finished task 0.0 in stage 0.
scheduler.DAGScheduler: ResultStage 0 (saveAsTextFile a
scheduler.DAGScheduler: Job 0 finished: saveAsTextFile
```



Job Run

```
spark.SparkContext: Starting job: saveAsTextFile
scheduler.DAGScheduler: Got job 0 (saveAsTextFil
scheduler.DAGScheduler: Submitting ResultStage 0 (
cluster.YarnScheduler: Adding task set 0.0 with 2 tasks
scheduler.TaskSetManager: Starting task 0.0 in stage 0.
scheduler.TaskSetManager: Starting task 1.0 in stage 0.
scheduler.TaskSetManager: Finished task 1.0 in stage 0.
scheduler.TaskSetManager: Finished task 0.0 in stage 0.
scheduler.DAGScheduler: ResultStage 0 (saveAsTextFile a
scheduler.DAGScheduler: Job 0 finished: saveAsTextFile
```

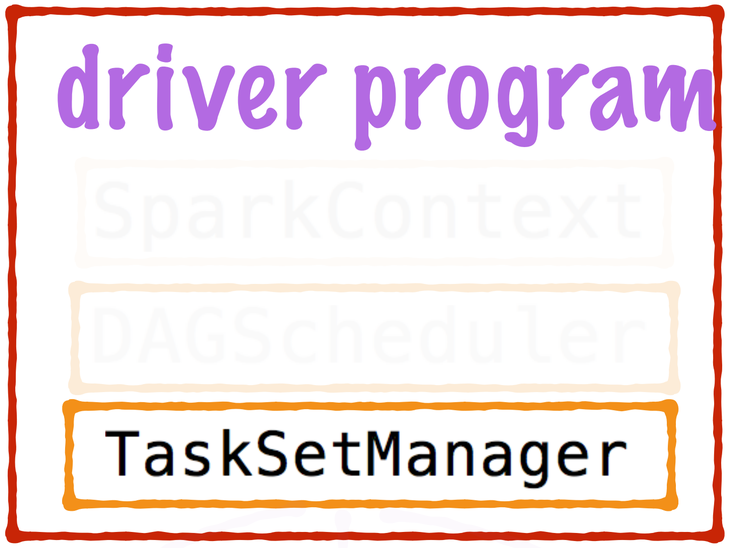
SparkContext starts the Job



Job Run

```
spark.SparkContext: Starting job: saveAsTextFile
scheduler.DAGScheduler: Got job 0 (saveAsTextFil
scheduler.DAGScheduler: Submitting ResultStage 0 (
cluster.YarnScheduler: Adding task set 0.0 with 2 tasks
scheduler.TaskSetManager: Starting task 0.0 in stage 0.
scheduler.TaskSetManager: Starting task 1.0 in stage 0.
scheduler.TaskSetManager: Finished task 1.0 in stage 0.
scheduler.TaskSetManager: Finished task 0.0 in stage 0.
scheduler.DAGScheduler: ResultStage 0 (saveAsTextFile a
scheduler.DAGScheduler: Job 0 finished: saveAsTextFile
```

DAGScheduler takes the Job and constructs a DAG of Stages and tasks



Job Run

```
spark.SparkContext: Starting job: saveAsTextFile
scheduler.DAGScheduler: Got job 0 (saveAsTextFil
scheduler.DAGScheduler: Submitting ResultStage 0 (
cluster.YarnScheduler: Adding task set 0.0 with 2 tasks
scheduler.TaskSetManager: Starting task 0.0 in stage 0.
scheduler.TaskSetManager: Starting task 1.0 in stage 0.
scheduler.TaskSetManager: Finished task 1.0 in stage 0.
scheduler.TaskSetManager: Finished task 0.0 in stage 0.
scheduler.DAGScheduler: ResultStage 0 (saveAsTextFile a
scheduler.DAGScheduler: Job 0 finished: saveAsTextFile
```

TaskSetManager actually schedules and keeps track of the tasks

MAPREDUCE WITH SPARK

We have a large text file

next up previous contents index
Next: Dynamic indexing Up: Index construction Previous: Single-pass in-memory indexing Contents Index

Distributed indexing

Collections are often so large that we cannot perform index construction efficiently on a single machine. This is particularly true of the World Wide Web for which we need large computer clusters [*]to construct any reasonably sized web index. Web search engines, therefore, use distributed indexing algorithms for index construction. The result of the construction process is a distributed index that is partitioned across several machines - either according to term or according to document. In this section, we describe distributed indexing for a
term-partitioned index . Most large search engines prefer a document-partitioned index (which can be easily generated from a term-partitioned index). We discuss this topic further in Section 20.3 (page [*]).

The distributed index construction method we describe in this section is an application of MapReduce , a general architecture for distributed computing. MapReduce is designed for large computer clusters. The point of a cluster is to solve large computing problems on cheap commodity machines or nodes that are built from standard parts (processor, memory, disk) as opposed to on a supercomputer with specialized hardware. Although hundreds or thousands of machines are available in such clusters, individual machines can fail at any time. One requirement for robust distributed indexing is, therefore, that we divide the work up into chunks that we can easily assign and - in case of failure - reassign. A master node directs the process of assigning and reassigning tasks to individual worker nodes.

The map and reduce phases of MapReduce split up the computing job into chunks that standard machines can process in a short time. The various steps of MapReduce are shown in Figure 4.5 and an example on a collection consisting of two documents is shown in Figure 4.6 . First, the input data, in our case a collection of web pages, are split into n splits where the size of the split is chosen to ensure that the work can be distributed evenly (chunks should not be too large) and efficiently (the total number of chunks we need to manage should not be too large); 16 or 64 MB are good sizes in distributed indexing. Splits are not preassigned to machines, but are instead assigned by the master node on an ongoing basis: As a machine finishes processing one split, it is assigned the next one. If a machine dies or becomes a laggard due to hardware problems, the split it is working on is simply reassigned to another machine.

Figure 4.5: An example of distributed indexing with MapReduce. Adapted from Dean and Ghemawat (2004).
\includegraphics[width=11.5cm]{art/mapreduce2.eps}
In general, MapReduce breaks a large computing problem into smaller parts by recasting it in terms of manipulation of key-value pairs . For indexing, a key-value pair has the form (termID,docID). In distributed indexing, the mapping from terms to termIDs is also distributed and therefore more complex than in single-machine indexing. A simple solution is to maintain a (perhaps precomputed) mapping for frequent terms that is copied to all nodes and to use terms directly (instead of termIDs) for infrequent terms. We do not address this problem here and assume that all nodes share a consistent term $\rightarrow$ termID mapping.

The map phase of MapReduce consists of mapping splits of the input data to key-value pairs. This is the same parsing task we also encountered in BSBI and SPIMI, and we therefore call the machines that execute the map phase parsers . Each parser writes its output to local intermediate files, the segment files (shown as \fbox{a-f\medstrut} \fbox{g-p\medstrut} \fbox{q-z\medstrut} in Figure 4.5).

For the reduce phase , we want all values for a given key to be stored close together, so that they can be read and processed quickly. This is achieved by partitioning the keys into j term partitions and having the parsers write key-value pairs for each term partition into a separate segment file. In Figure 4.5 , the term partitions are according to first letter: a-f, g-p, q-z, and $j=3$. (We chose these key ranges for ease of exposition. In general, key ranges need not correspond to contiguous terms or termIDs.) The term partitions are defined by the person who operates the indexing system (Exercise 4.6). The parsers then write corresponding segment files, one for each term partition. Each term partition thus corresponds to r segments files, where r is the number of parsers. For instance, Figure 4.5 shows three a-f segment files of the a-f partition, corresponding to the three parsers shown in the figure.

Collecting all values (here: docIDs) for a given key (here: termID) into one list is the task of the inverters in the reduce phase. The

Objective: Create a Frequency Distribution of words in the file

We have a large text file

next up previous contents index
Next: Dynamic indexing Up: Index construction Previous: Single-pass in-memory indexing Contents Index

Distributed indexing

Collections are often so large that we cannot perform index construction efficiently on a single machine. This is particularly true of the World Wide Web for which we need large computer clusters [*]to construct any reasonably sized web index. Web search engines, therefore, use distributed indexing algorithms for index construction. The result of the construction process is a distributed index that is partitioned across several machines - either according to term or according to document. In this section, we describe distributed indexing for a
term-partitioned index . Most large search engines prefer a document-partitioned index (which can be easily generated from a term-partitioned index). We discuss this topic further in Section 20.3 (page [*]).

The distributed index construction method we describe in this section is an application of MapReduce , a general architecture for distributed computing. MapReduce is designed for large computer clusters. The point of a cluster is to solve large computing problems on cheap commodity machines or nodes that are built from standard parts (processor, memory, disk) as opposed to on a supercomputer with specialized hardware. Although hundreds or thousands of machines are available in such clusters, individual machines can fail at any time. One requirement for robust distributed indexing is, therefore, that we divide the work up into chunks that we can easily assign and - in case of failure - reassign. A master node directs the process of assigning and reassigning tasks to individual worker nodes.

The map and reduce phases of MapReduce split up the computing job into chunks that standard machines can process in a short time. The various steps of MapReduce are shown in Figure 4.5 and an example on a collection consisting of two documents is shown in Figure 4.6 . First, the input data, in our case a collection of web pages, are split into n splits where the size of the split is chosen to ensure that the work can be distributed evenly (chunks should not be too large) and efficiently (the total number of chunks we need to manage should not be too large); 16 or 64 MB are good sizes in distributed indexing. Splits are not preassigned to machines, but are instead assigned by the master node on an ongoing basis: As a machine finishes processing one split, it is assigned the next one. If a machine dies or becomes a laggard due to hardware problems, the split it is working on is simply reassigned to another machine.

Figure 4.5: An example of distributed indexing with MapReduce. Adapted from Dean and Ghemawat (2004).
\includegraphics[width=11.5cm]{art/mapreduce2.eps}
In general, MapReduce breaks a large computing problem into smaller parts by recasting it in terms of manipulation of key-value pairs . For indexing, a key-value pair has the form (termID,docID). In distributed indexing, the mapping from terms to termIDs is also distributed and therefore more complex than in single-machine indexing. A simple solution is to maintain a (perhaps precomputed) mapping for frequent terms that is copied to all nodes and to use terms directly (instead of termIDs) for infrequent terms. We do not address this problem here and assume that all nodes share a consistent term $\rightarrow$ termID mapping.

The map phase of MapReduce consists of mapping splits of the input data to key-value pairs. This is the same parsing task we also encountered in BSBI and SPIMI, and we therefore call the machines that execute the map phase parsers . Each parser writes its output to local intermediate files, the segment files (shown as \fbox{a-f\medstrut} \fbox{g-p\medstrut} \fbox{q-z\medstrut} in Figure 4.5).

For the reduce phase , we want all values for a given key to be stored close together, so that they can be read and processed quickly. This is achieved by partitioning the keys into j term partitions and having the parsers write key-value pairs for each term partition into a separate segment file. In Figure 4.5 , the term partitions are according to first letter: a-f, g-p, q-z, and $j=3$. (We chose these key ranges for ease of exposition. In general, key ranges need not correspond to contiguous terms or termIDs.) The term partitions are defined by the person who operates the indexing system (Exercise 4.6). The parsers then write corresponding segment files, one for each term partition. Each term partition thus corresponds to r segments files, where r is the number of parsers. For instance, Figure 4.5 shows three a-f segment files of the a-f partition, corresponding to the three parsers shown in the figure.

Collecting all values (here: docIDs) for a given key (here: termID) into one list is the task of the inverters in the reduce phase. The

This is a pretty common task in Natural Language Processing

We have a large text file

next up previous contents index
Next: Dynamic indexing Up: Index construction Previous: Single-pass in-memory indexing Contents Index

Distributed indexing

Collections are often so large that we cannot perform index construction efficiently on a single machine. This is particularly true of the World Wide Web for which we need large computer clusters [*]to construct any reasonably sized web index. Web search engines, therefore, use distributed indexing algorithms for index construction. The result of the construction process is a distributed index that is partitioned across several machines - either according to term or according to document. In this section, we describe distributed indexing for a
term-partitioned index . Most large search engines prefer a document-partitioned index (which can be easily generated from a term-partitioned index). We discuss this topic further in Section 20.3 (page [*]).

The distributed index construction method we describe in this section is an application of MapReduce , a general architecture for distributed computing. MapReduce is designed for large computer clusters. The point of a cluster is to solve large computing problems on cheap commodity machines or nodes that are built from standard parts (processor, memory, disk) as opposed to on a supercomputer with specialized hardware. Although hundreds or thousands of machines are available in such clusters, individual machines can fail at any time. One requirement for robust distributed indexing is, therefore, that we divide the work up into chunks that we can easily assign and - in case of failure - reassign. A master node directs the process of assigning and reassigning tasks to individual worker nodes.

The map and reduce phases of MapReduce split up the computing job into chunks that standard machines can process in a short time. The various steps of MapReduce are shown in Figure 4.5 and an example on a collection consisting of two documents is shown in Figure 4.6 . First, the input data, in our case a collection of web pages, are split into n splits where the size of the split is chosen to ensure that the work can be distributed evenly (chunks should not be too large) and efficiently (the total number of chunks we need to manage should not be too large); 16 or 64 MB are good sizes in distributed indexing. Splits are not preassigned to machines, but are instead assigned by the master node on an ongoing basis: As a machine finishes processing one split, it is assigned the next one. If a machine dies or becomes a laggard due to hardware problems, the split it is working on is simply reassigned to another machine.

Figure 4.5: An example of distributed indexing with MapReduce. Adapted from Dean and Ghemawat (2004).
\includegraphics[width=11.5cm]{art/mapreduce2.eps}
In general, MapReduce breaks a large computing problem into smaller parts by recasting it in terms of manipulation of key-value pairs . For indexing, a key-value pair has the form (termID,docID). In distributed indexing, the mapping from terms to termIDs is also distributed and therefore more complex than in single-machine indexing. A simple solution is to maintain a (perhaps precomputed) mapping for frequent terms that is copied to all nodes and to use terms directly (instead of termIDs) for infrequent terms. We do not address this problem here and assume that all nodes share a consistent term $\rightarrow$ termID mapping.

The map phase of MapReduce consists of mapping splits of the input data to key-value pairs. This is the same parsing task we also encountered in BSBI and SPIMI, and we therefore call the machines that execute the map phase parsers . Each parser writes its output to local intermediate files, the segment files (shown as \fbox{a-f\medstrut} \fbox{g-p\medstrut} \fbox{q-z\medstrut} in Figure 4.5).

For the reduce phase , we want all values for a given key to be stored close together, so that they can be read and processed quickly. This is achieved by partitioning the keys into j term partitions and having the parsers write key-value pairs for each term partition into a separate segment file. In Figure 4.5 , the term partitions are according to first letter: a-f, g-p, q-z, and $j=3$. (We chose these key ranges for ease of exposition. In general, key ranges need not correspond to contiguous terms or termIDs.) The term partitions are defined by the person who operates the indexing system (Exercise 4.6). The parsers then write corresponding segment files, one for each term partition. Each term partition thus corresponds to r segments files, where r is the number of parsers. For instance, Figure 4.5 shows three a-f segment files of the a-f partition, corresponding to the three parsers shown in the figure.

Collecting all values (here: docIDs) for a given key (here: termID) into one list is the task of the inverters in the reduce phase. The

Word	Count
because	1
each	4
figure	9
..	..

How can we do this in Spark?

Word Counts in Spark

We'll start by loading the file into an RDD

```
lines = sc.textFile(textfilePath)
```

Each record in the RDD represents a line in the text file

Word Counts in Spark

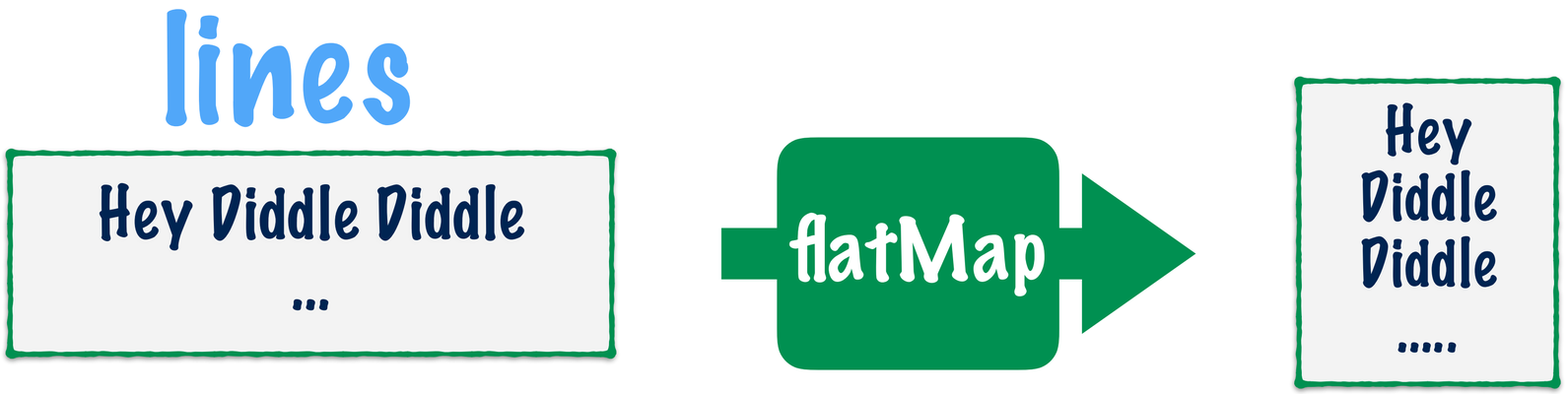
```
lines = sc.textFile(textfilePath)
```

```
wordsRDD=lines.flatMap(lambda x:x.split()).map(lambda x:(x,1))
```

This is the crucial step in this exercise

Word Counts in Spark

```
wordsRDD=lines.flatMap(lambda x:x.split()).map(lambda x:(x,1))
```



It creates an RDD in which each record is a word in the file

Word Counts in Spark

```
wordsRDD=lines.flatMap(lambda x:x.split()).map(lambda x:(x,1))
```

lines

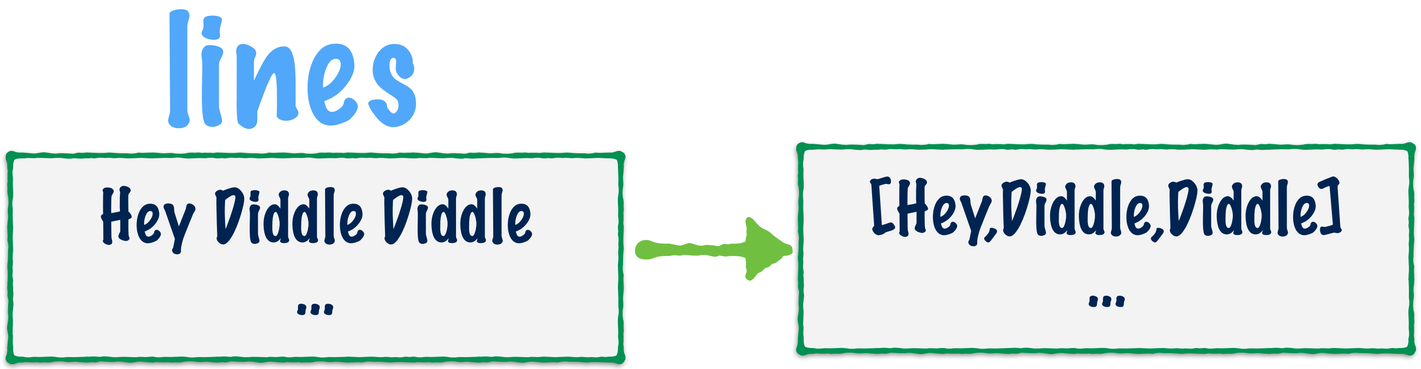
Hey Diddle Diddle
...

Hey,
Diddle,
Diddle,
.....

Let's parse what happened here

Word Counts in Spark

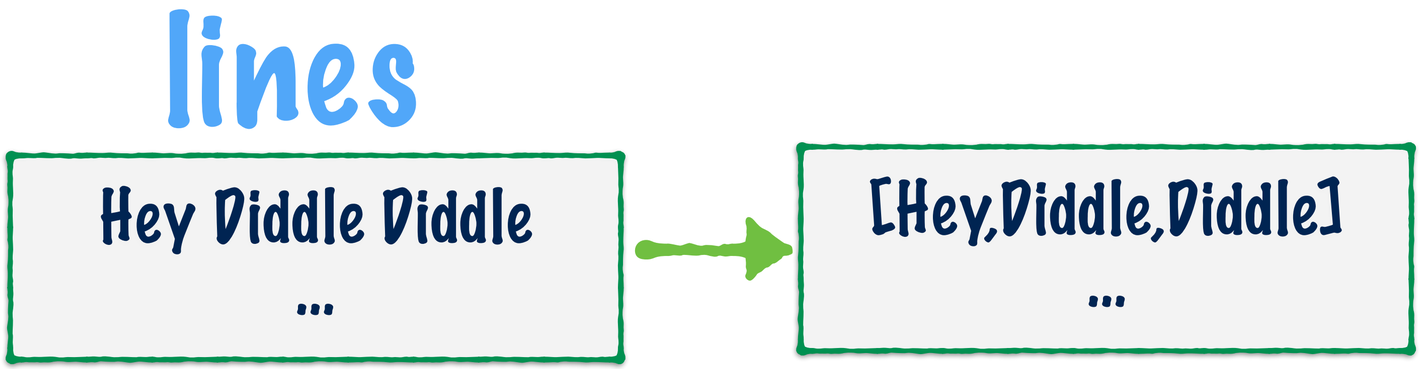
```
wordsRDD=lines.flatMap(lambda x:x.split()).map(lambda x:(x,1))
```



This function creates a list for each record in the lines RDD

Word Counts in Spark

```
wordsRDD=lines.flatMap(lambda x:x.split()).map(lambda x:(x,1))
```

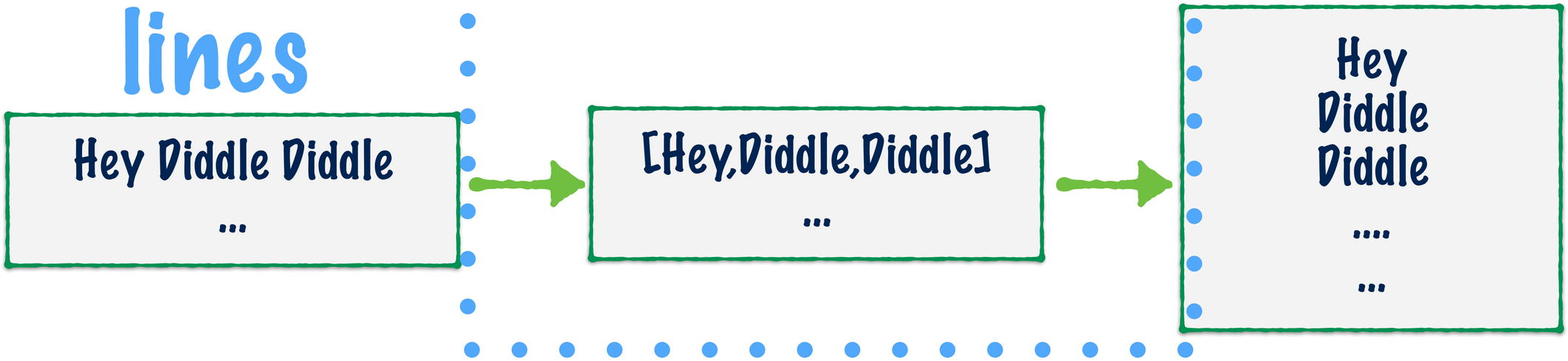


This is exactly what would have happened if we used map instead of flatMap

flatMap goes one step further

Word Counts in Spark

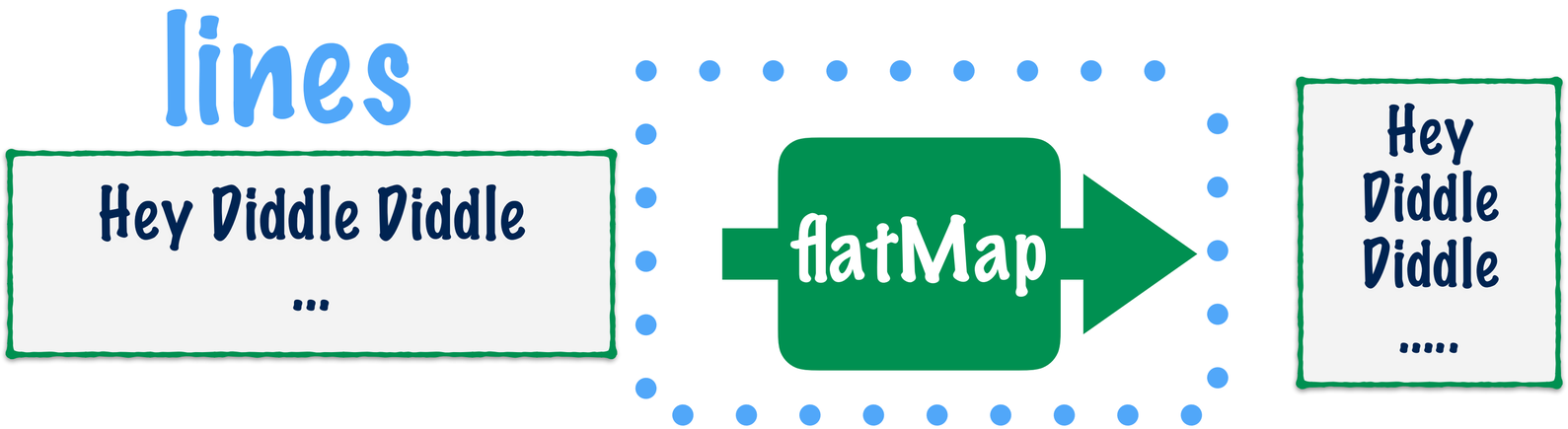
```
wordsRDD=lines.flatMap(lambda x:x.split()).map(lambda x:(x,1))
```



flatMap then creates one record for each element in the list

Word Counts in Spark

```
wordsRDD=lines.flatMap(lambda x:x.split()).map(lambda x:(x,1))
```



flatMap then creates one record for each element in the list

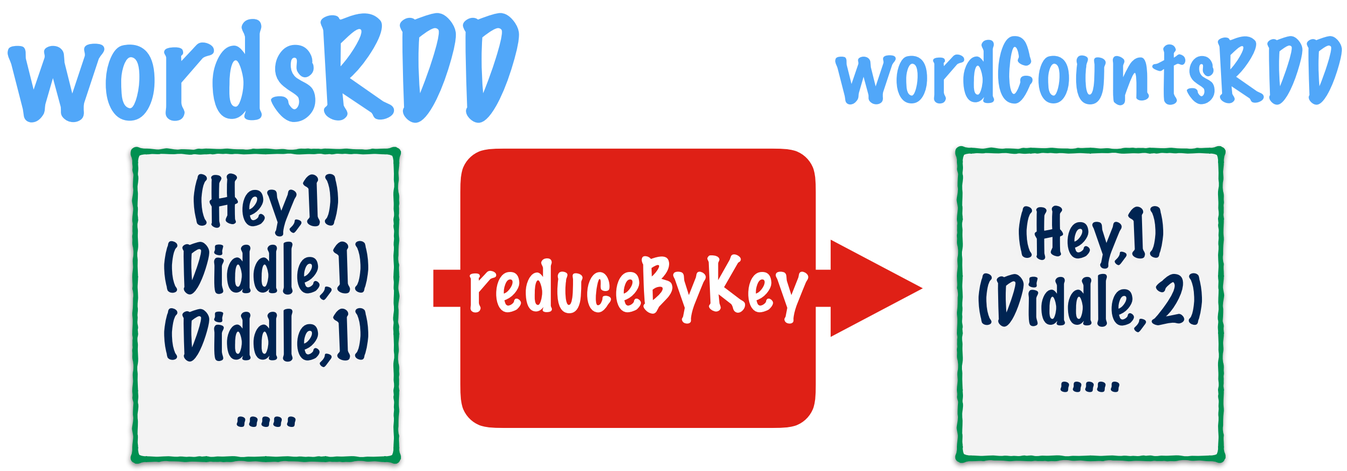
Word Counts in Spark

```
wordsRDD=lines.flatMap(lambda x:x.split()).map(lambda x:(x,1))
```

The map step creates a Pair RDD with the value representing the count

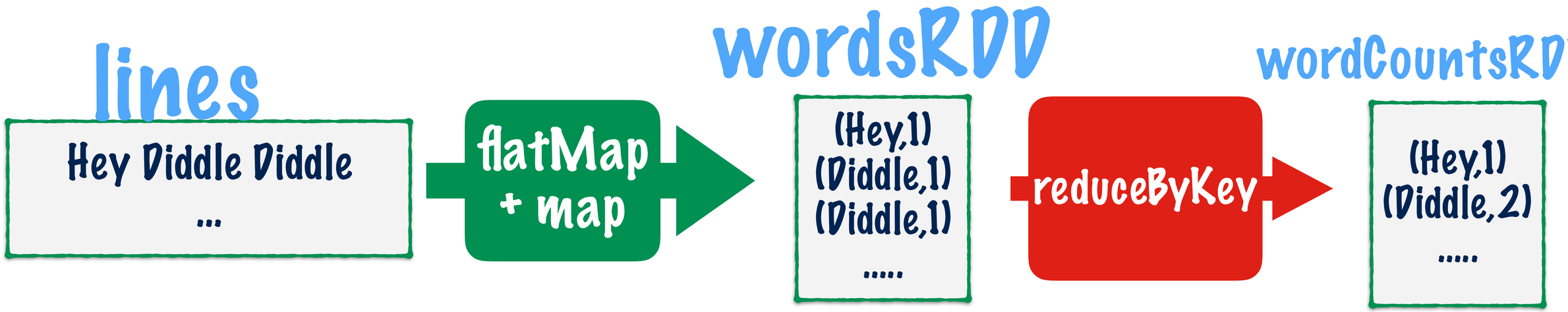
Word Counts in Spark

```
wordCountsRDD=wordsRDD.reduceByKey(lambda x,y:x+y)
```



We can simply use reduceByKey to compute the word counts

Word Counts in Spark



What we just did is a classic example of the MapReduce programming model

MapReduce

MapReduce is a programming model

Invented by Google

MapReduce

Hadoop uses MapReduce
for all it's computing tasks

MapReduce

Distributed computing
can get very complicated

How to manage tasks across multiple nodes?

What to do if a node goes down?

MapReduce

MapReduce abstracts the programmer from all these complications

MapReduce

You just define 2 functions

map() reduce()

Note. These are different from Spark's built in map and reduce operations

MapReduce

map() reduce()

The rest is taken care of by Hadoop!

MapReduce

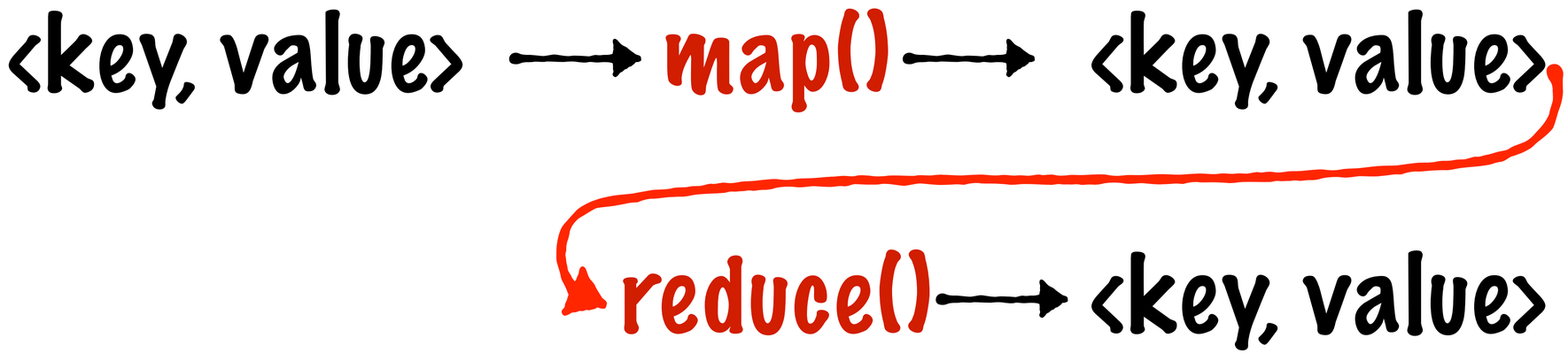
map() reduce()

This paradigm is
driven by a key insight

MapReduce

key insight map() reduce()

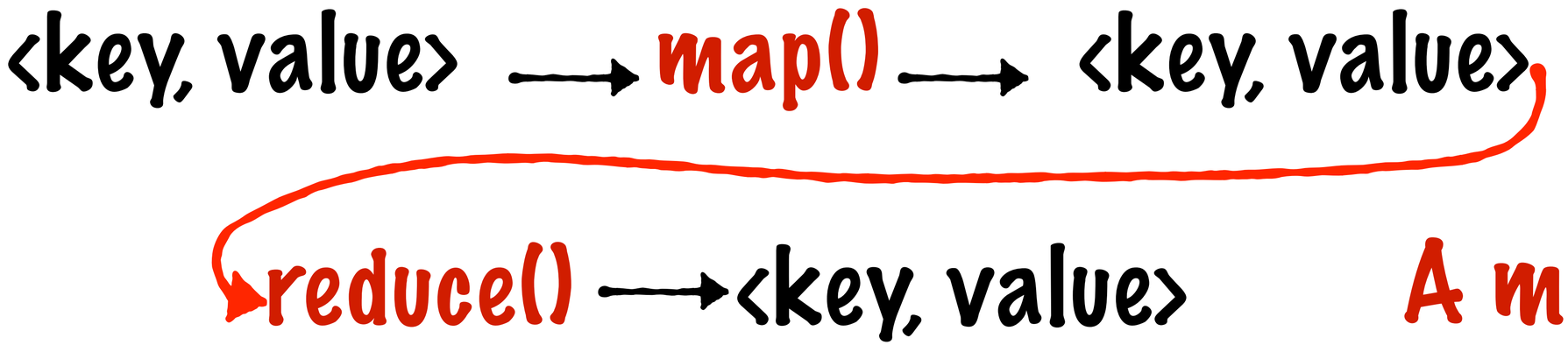
ANY data processing task can be parallelized , if you express it as



MapReduce

key insight

map() reduce()

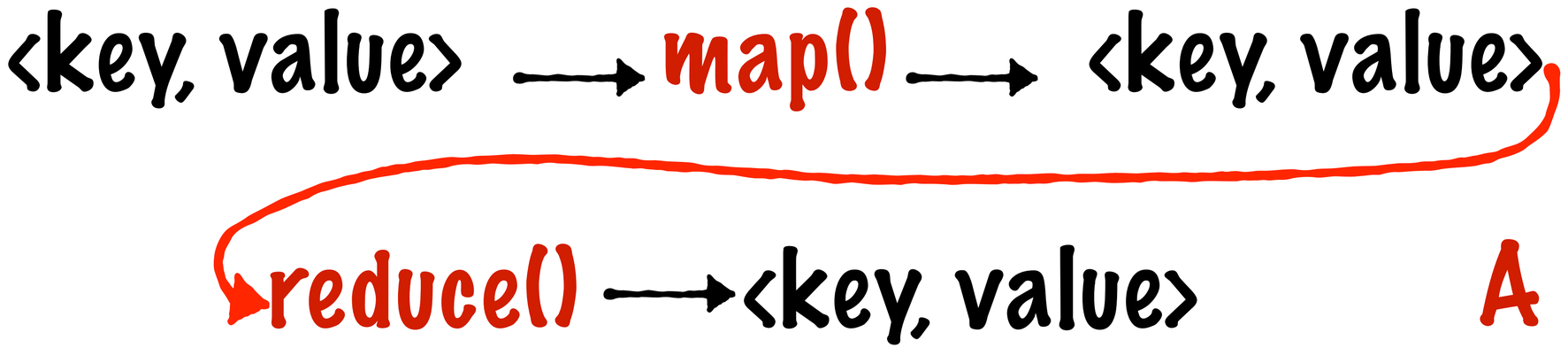


A map() task that transforms a key, value pair to a set of key, value pairs

MapReduce

key insight

map() reduce()

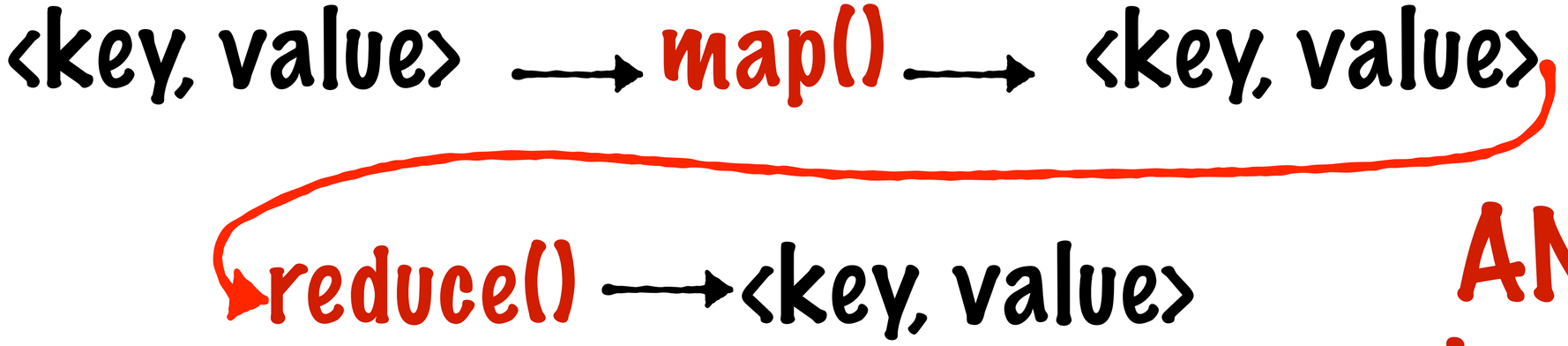


A reduce() task that combines values which have the same key

MapReduce

key insight

map() reduce()

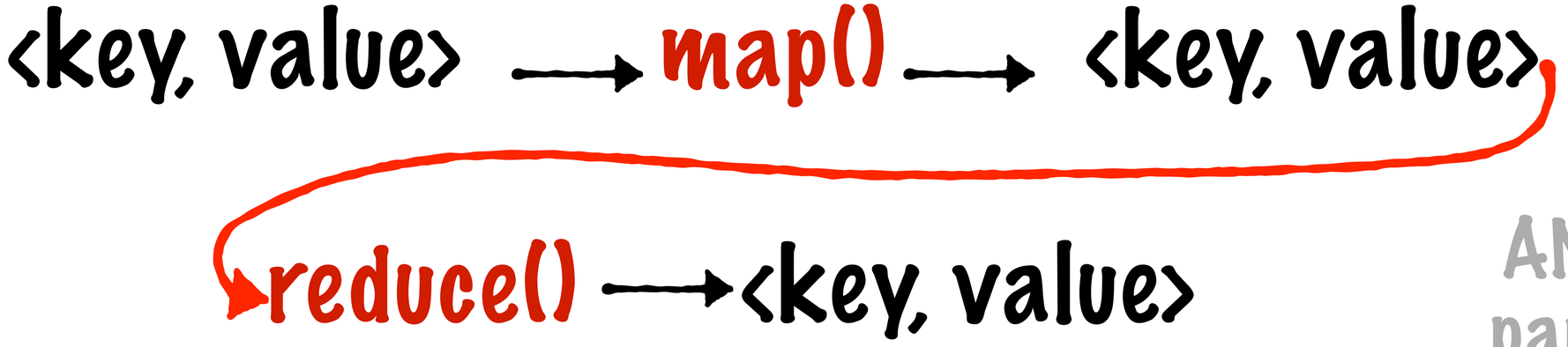


ANY task can be parallelized if it's expressed in this form

MapReduce

key insight

map() reduce()



ANY task can be parallelized if it's expressed in this form

or a chain of such transformations

MapReduce

While MapReduce is very powerful ,
it is also a little restrictive

In Hadoop, every task needs to be broken down into Map and Reduce tasks

This makes it difficult to intuitively express very complex tasks

MapReduce

With Spark, the user does not have to break down tasks into map and reduce

However, some tasks lend themselves beautifully to the MapReduce model

Word Counts in text documents,
Building inverted indices

MapReduce

However, some tasks lend themselves beautifully to the MapReduce model

Word Counts in text documents,
Building inverted indices

..and so, Spark allows users to express these tasks using the MapReduce model

MapReduce

Hadoop	Spark
Map	flatMap
Reduce	reduceByKey

JAVA API

So far, we have used
the Python API for
working with RDDs

Python lends itself really
well to data analysis

But sometimes you may want
to use the Spark Java API

For instance: to set up complex
class hierarchies or for type safety

The Java API is very similar to the Python API

All the usual operations like map, filter etc are available

The Java API uses
Function objects to
represent functions

These Function objects
are passed to map, filter
and other RDD operations

Let's go back to our
Flight delays example

This time we'll use Java
to manipulate the data

Flight delays using Java

```
public class parseRows {
    public static void main(String[] args) throws Exception {

        //Setting up the SparkContext
        SparkConf conf = new SparkConf().setAppName("parseRows");
        JavaSparkContext sc = new JavaSparkContext(conf);

        //Loading the data from airports file
        JavaRDD<String> airports = sc.textFile("hdfs:///user/swethakolalapudi/flightDelaysData/airports.csv");


        //The function that will be passed to filter
        class NotHeader implements Function<String,Boolean> {
            public NotHeader() {
            }

            public Boolean call(String s) throws Exception {
                return !s.contains("Description");
            }
        }

        //Filter transformation
        JavaRDD<String> airportsFiltered = airports.filter(new NotHeader());


        //Map transformation
        JavaRDD<List<String>> airportsParsed = airportsFiltered.map(new Function<String, List<String>>() {
            public List<String> call(String s) throws Exception {
                List<String> parsedRow = new ArrayList<String>();
                parsedRow.add(s.substring(0,s.indexOf(",")).replace("\"",""));
                parsedRow.add(s.substring(s.indexOf(",")+1,s.length()-1).replace("\"",""));
                return parsedRow;
            }
        });

        //Lambda function
        JavaRDD<Integer> rowLengths = airportsFiltered.map(s -> s.length());

        //Print out the results
        for (List<String> s:airportsParsed.collect()){
            System.out.println(StringUtils.join(s, "|"));
        }


        //A Pair function - used to create Pair RDDs
        PairFunction<List<String>,String,String> arrayToPair = new PairFunction<List<String>, String, String>() {
            @Override
            public Tuple2<String, String> call(List<String> strings) throws Exception {
                return new Tuple2<>(strings.get(0),strings.get(1));
            }
        };

        //Create a Pair RDD from airports
        JavaPairRDD<String,String> airportsLookup = airportsParsed.mapToPair(arrayToPair);

        //Use it to lookup a particular airport description
        System.out.println(airportsLookup.lookup("PPG"));

    }
}
```

Here is a class to

1. Parse the rows of the airports file

2. Create a Pair RDD for looking up the Airport description

Parse the rows

```
public class parseRows {
    public static void main(String[] args) throws Exception {

        //Setting up the SparkContext
        SparkConf conf = new SparkConf().setAppName("parseRows");
        JavaSparkContext sc = new JavaSparkContext(conf);

        //Loading the data from airports file
        JavaRDD<String> airports = sc.textFile("hdfs:///user/swethakolalapudi/flightDelaysData/airports.csv");


        //The function that will be passed to filter
        class NotHeader implements Function<String,Boolean> {
            public NotHeader() {
            }

            public Boolean call(String s) throws Exception {
                return !s.contains("Description");
            }
        }

        //Filter transformation
        JavaRDD<String> airportsFiltered = airports.filter(new NotHeader());


        //Map transformation
        JavaRDD<List<String>> airportsParsed = airportsFiltered.map(new Function<String, List<String>>() {
            public List<String> call(String s) throws Exception {
                List<String> parsedRow = new ArrayList<String>();
                parsedRow.add(s.substring(0,s.indexOf(",")).replace("\"",""));
                parsedRow.add(s.substring(s.indexOf(",")+1,s.length()-1).replace("\"",""));
                return parsedRow;
            }
        });

        //Lambda function
        JavaRDD<Integer> rowLengths = airportsFiltered.map(s -> s.length());

        //Print out the results
        for (List<String> s:airportsParsed.collect()){
            System.out.println(StringUtils.join(s, "|"));
        }


        //A Pair function - used to create Pair RDDs
        PairFunction<List<String>,String,String> arrayToPair = new PairFunction<List<String>, String, String>() {
            @Override
            public Tuple2<String, String> call(List<String> strings) throws Exception {
                return new Tuple2<>(strings.get(0),strings.get(1));
            }
        };

        //Create a Pair RDD from airports
        JavaPairRDD<String,String> airportsLookup = airportsParsed.mapToPair(arrayToPair);

        //Use it to lookup a particular airport description
        System.out.println(airportsLookup.lookup("PPG"));

    }
}
```

All our code is enclosed in the main function

Parse the rows

```
public class parseRows {
    public static void main(String[] args) throws Exception {
//Setting up the SparkContext
SparkConf conf = new SparkConf().setAppName("parseRows");
JavaSparkContext sc = new JavaSparkContext(conf);
        //Loading the data from airports file
        JavaRDD<String> airports = sc.textFile("hdfs:///user/swethakolalapudi/flightDelaysData/airports.csv");

        //The function that will be passed to filter
        class NotHeader implements Function<String,Boolean> {
            public NotHeader() {
            }

            public Boolean call(String s) throws Exception {
                return !s.contains("Description");
            }
        }

        //Filter transformation
        JavaRDD<String> airportsFiltered = airports.filter(new NotHeader());


        //Map transformation
        JavaRDD<List<String>> airportsParsed = airportsFiltered.map(new Function<String, List<String>>() {
            public List<String> call(String s) throws Exception {
                List<String> parsedRow = new ArrayList<String>();
                parsedRow.add(s.substring(0,s.indexOf(",")).replace("\"",""));
                parsedRow.add(s.substring(s.indexOf(",")+1,s.length()-1).replace("\"",""));
                return parsedRow;
            }
        });

        //Lambda function
        JavaRDD<Integer> rowLengths = airportsFiltered.map(s -> s.length());

        //Print out the results
        for (List<String> s:airportsParsed.collect()){
            System.out.println(StringUtils.join(s, "|"));
        }


        //A Pair function - used to create Pair RDDs
        PairFunction<List<String>,String,String> arrayToPair = new PairFunction<List<String>, String, String>() {
            @Override
            public Tuple2<String, String> call(List<String> strings) throws Exception {
                return new Tuple2<>(strings.get(0),strings.get(1));
            }
        };

        //Create a Pair RDD from airports
        JavaPairRDD<String,String> airportsLookup = airportsParsed.mapToPair(arrayToPair);

        //Use it to lookup a particular airport description
        System.out.println(airportsLookup.lookup("PPG"));

    }
}
```

First we need to set up a SparkContext

Parse the rows

```
public class parseRows {
    public static void main(String[] args) throws Exception {
//Setting up the SparkContext
SparkConf conf = new SparkConf().setAppName("parseRows");
JavaSparkContext sc = new JavaSparkContext(conf);
        //Loading the data from airports file
        JavaRDD<String> airports = sc.textFile("hdfs:///user/swethakolalapudi/flightDelaysData/airports.csv");

        //The function that will be passed to filter
        class NotHeader implements Function<String,Boolean> {
            public NotHeader() {
            }

            public Boolean call(String s) throws Exception {
                return !s.contains("Description");
            }
        }

        //Filter transformation
        JavaRDD<String> airportsFiltered = airports.filter(new NotHeader());


        //Map transformation
        JavaRDD<List<String>> airportsParsed = airportsFiltered.map(new Function<String, List<String>>() {
            public List<String> call(String s) throws Exception {
                List<String> parsedRow = new ArrayList<String>();
                parsedRow.add(s.substring(0,s.indexOf(",")).replace("\"",""));
                parsedRow.add(s.substring(s.indexOf(",")+1,s.length()-1).replace("\"",""));
                return parsedRow;
            }
        });

        //Lambda function
        JavaRDD<Integer> rowLengths = airportsFiltered.map(s -> s.length());

        //Print out the results
        for (List<String> s:airportsParsed.collect()){
            System.out.println(StringUtils.join(s, "|"));
        }


        //A Pair function - used to create Pair RDDs
        PairFunction<List<String>,String,String> arrayToPair = new PairFunction<List<String>, String, String>() {
            @Override
            public Tuple2<String, String> call(List<String> strings) throws Exception {
                return new Tuple2<>(strings.get(0),strings.get(1));
            }
        };

        //Create a Pair RDD from airports
        JavaPairRDD<String,String> airportsLookup = airportsParsed.mapToPair(arrayToPair);

        //Use it to lookup a particular airport description
        System.out.println(airportsLookup.lookup("PPG"));

    }
}
```

The SparkContext represents a connection to the Spark cluster

Parse the rows

```
public class parseRows {
    public static void main(String[] args) throws Exception {
//Setting up the SparkContext
SparkConf conf = new SparkConf().setAppName("parseRows");
JavaSparkContext sc = new JavaSparkContext(conf);

        //Loading the data from airports file
        JavaRDD<String> airports = sc.textFile("hdfs:///user/swethakolalapudi/flightDelaysData/airports.csv");


        //The function that will be passed to filter
        class NotHeader implements Function<String,Boolean> {
            public NotHeader() {
            }

            public Boolean call(String s) throws Exception {
                return !s.contains("Description");
            }
        }

        //Filter transformation
        JavaRDD<String> airportsFiltered = airports.filter(new NotHeader());


        //Map transformation
        JavaRDD<List<String>> airportsParsed = airportsFiltered.map(new Function<String, List<String>>() {
            public List<String> call(String s) throws Exception {
                List<String> parsedRow = new ArrayList<String>();
                parsedRow.add(s.substring(0,s.indexOf(",")).replace("\"",""));
                parsedRow.add(s.substring(s.indexOf(",")+1,s.length()-1).replace("\"",""));
                return parsedRow;
            }
        });

        //Lambda function
        JavaRDD<Integer> rowLengths = airportsFiltered.map(s -> s.length());

        //Print out the results
        for (List<String> s:airportsParsed.collect()){
            System.out.println(StringUtils.join(s, "|"));
        }


        //A Pair function - used to create Pair RDDs
        PairFunction<List<String>,String,String> arrayToPair = new PairFunction<List<String>, String, String>() {
            @Override
            public Tuple2<String, String> call(List<String> strings) throws Exception {
                return new Tuple2<>(strings.get(0),strings.get(1));
            }
        };

        //Create a Pair RDD from airports
        JavaPairRDD<String,String> airportsLookup = airportsParsed.mapToPair(arrayToPair);

        //Use it to lookup a particular airport description
        System.out.println(airportsLookup.lookup("PPG"));

    }
}
```

The SparkContext needs to be setup using a SparkConf

Parse the rows

```
public class parseRows {
    public static void main(String[] args) throws Exception {
//Setting up the SparkContext
SparkConf conf = new SparkConf().setAppName("parseRows");
JavaSparkContext sc = new JavaSparkContext(conf);

        //Loading the data from airports file
        JavaRDD<String> airports = sc.textFile("hdfs:///user/swethakolalapudi/flightDelaysData/airports.csv");


        //The function that will be passed to filter
        class NotHeader implements Function<String,Boolean> {
            public NotHeader() {
            }

            public Boolean call(String s) throws Exception {
                return !s.contains("Description");
            }
        }

        //Filter transformation
        JavaRDD<String> airportsFiltered = airports.filter(new NotHeader());


        //Map transformation
        JavaRDD<List<String>> airportsParsed = airportsFiltered.map(new Function<String, List<String>>() {
            public List<String> call(String s) throws Exception {
                List<String> parsedRow = new ArrayList<String>();
                parsedRow.add(s.substring(0,s.indexOf(",")).replace("\"",""));
                parsedRow.add(s.substring(s.indexOf(",")+1,s.length()-1).replace("\"",""));
                return parsedRow;
            }
        });

        //Lambda function
        JavaRDD<Integer> rowLengths = airportsFiltered.map(s -> s.length());

        //Print out the results
        for (List<String> s:airportsParsed.collect()){
            System.out.println(StringUtils.join(s, "|"));
        }


        //A Pair function - used to create Pair RDDs
        PairFunction<List<String>,String,String> arrayToPair = new PairFunction<List<String>, String, String>() {
            @Override
            public Tuple2<String, String> call(List<String> strings) throws Exception {
                return new Tuple2<>(strings.get(0),strings.get(1));
            }
        };

        //Create a Pair RDD from airports
        JavaPairRDD<String,String> airportsLookup = airportsParsed.mapToPair(arrayToPair);

        //Use it to lookup a particular airport description
        System.out.println(airportsLookup.lookup("PPG"));

    }
}
```

You can set the master (cluster manager) and the app name using the SparkConf

Parse the rows

```
public class parseRows {
    public static void main(String[] args) throws Exception {
//Setting up the SparkContext
SparkConf conf = new SparkConf().setAppName("parseRows");
JavaSparkContext sc = new JavaSparkContext(conf);
        //Loading the data from airports file
        JavaRDD<String> airports = sc.textFile("hdfs:///user/swethakolalapudi/flightDelaysData/airports.csv");

        //The function that will be passed to filter
        class NotHeader implements Function<String,Boolean> {
            public NotHeader() {
            }

            public Boolean call(String s) throws Exception {
                return !s.contains("Description");
            }
        }

        //Filter transformation
        JavaRDD<String> airportsFiltered = airports.filter(new NotHeader());


        //Map transformation
        JavaRDD<List<String>> airportsParsed = airportsFiltered.map(new Function<String, List<String>>() {
            public List<String> call(String s) throws Exception {
                List<String> parsedRow = new ArrayList<String>();
                parsedRow.add(s.substring(0,s.indexOf(",")).replace("\"",""));
                parsedRow.add(s.substring(s.indexOf(",")+1,s.length()-1).replace("\"",""));
                return parsedRow;
            }
        });

        //Lambda function
        JavaRDD<Integer> rowLengths = airportsFiltered.map(s -> s.length());

        //Print out the results
        for (List<String> s:airportsParsed.collect()){
            System.out.println(StringUtils.join(s, "|"));
        }


        //A Pair function - used to create Pair RDDs
        PairFunction<List<String>,String,String> arrayToPair = new PairFunction<List<String>, String, String>() {
            @Override
            public Tuple2<String, String> call(List<String> strings) throws Exception {
                return new Tuple2<>(strings.get(0),strings.get(1));
            }
        };

        //Create a Pair RDD from airports
        JavaPairRDD<String,String> airportsLookup = airportsParsed.mapToPair(arrayToPair);

        //Use it to lookup a particular airport description
        System.out.println(airportsLookup.lookup("PPG"));

    }
}
```

Once you have a SparkContext, you can use it to create an RDD

Parse the rows

```
public class parseRows {
    public static void main(String[] args) throws Exception {

        //Setting up the SparkContext
        SparkConf conf = new SparkConf().setAppName("parseRows");
        JavaSparkContext sc = new JavaSparkContext(conf);

//Loading the data from airports file
JavaRDD<String> airports = sc.textFile("hdfs:///user/swethakolalapudi/flightDelaysData/airports.csv");
        //The function that will be passed to filter
        class NotHeader implements Function<String,Boolean> {
            public NotHeader() {
            }

            public Boolean call(String s) throws Exception {
                return !s.contains("Description");
            }
        }

        //Filter transformation
        JavaRDD<String> airportsFiltered = airports.filter(new NotHeader());


        //Map transformation
        JavaRDD<List<String>> airportsParsed = airportsFiltered.map(new Function<String, List<String>>() {
            public List<String> call(String s) throws Exception {
                List<String> parsedRow = new ArrayList<String>();
                parsedRow.add(s.substring(0,s.indexOf(",")).replace("\"",""));
                parsedRow.add(s.substring(s.indexOf(",")+1,s.length()-1).replace("\"",""));
                return parsedRow;
            }
        });

        //Lambda function
        JavaRDD<Integer> rowLengths = airportsFiltered.map(s -> s.length());

        //Print out the results
        for (List<String> s:airportsParsed.collect()){
            System.out.println(StringUtils.join(s, "|"));
        }


        //A Pair function - used to create Pair RDDs
        PairFunction<List<String>,String,String> arrayToPair = new PairFunction<List<String>, String, String>() {
            @Override
            public Tuple2<String, String> call(List<String> strings) throws Exception {
                return new Tuple2<>(strings.get(0),strings.get(1));
            }
        };

        //Create a Pair RDD from airports
        JavaPairRDD<String,String> airportsLookup = airportsParsed.mapToPair(arrayToPair);

        //Use it to lookup a particular airport description
        System.out.println(airportsLookup.lookup("PPG"));

    }
}
```

The method to create the RDD from a text file

Parse the rows

```
public class parseRows {
    public static void main(String[] args) throws Exception {

        //Setting up the SparkContext
        SparkConf conf = new SparkConf().setAppName("parseRows");
        JavaSparkContext sc = new JavaSparkContext(conf);

 //Loading the data from airports file
JavaRDD<String> airports = sc.textFile("hdfs:///user/swethakolalapudi/flightDelaysData/airports.csv");
        //The function that will be passed to filter
        class NotHeader implements Function<String,Boolean> {
            public NotHeader() {
            }

            public Boolean call(String s) throws Exception {
                return !s.contains("Description");
            }
        }

        //Filter transformation
        JavaRDD<String> airportsFiltered = airports.filter(new NotHeader());


        //Map transformation
        JavaRDD<List<String>> airportsParsed = airportsFiltered.map(new Function<String, List<String>>() {
            public List<String> call(String s) throws Exception {
                List<String> parsedRow = new ArrayList<String>();
                parsedRow.add(s.substring(0,s.indexOf(",")).replace("\"",""));
                parsedRow.add(s.substring(s.indexOf(",")+1,s.length()-1).replace("\"",""));
                return parsedRow;
            }
        });

        //Lambda function
        JavaRDD<Integer> rowLengths = airportsFiltered.map(s -> s.length());

        //Print out the results
        for (List<String> s:airportsParsed.collect()){
            System.out.println(StringUtils.join(s, "|"));
        }


        //A Pair function - used to create Pair RDDs
        PairFunction<List<String>,String,String> arrayToPair = new PairFunction<List<String>, String, String>() {
            @Override
            public Tuple2<String, String> call(List<String> strings) throws Exception {
                return new Tuple2<>(strings.get(0),strings.get(1));
            }
        };

        //Create a Pair RDD from airports
        JavaPairRDD<String,String> airportsLookup = airportsParsed.mapToPair(arrayToPair);

        //Use it to lookup a particular airport description
        System.out.println(airportsLookup.lookup("PPG"));

    }
}
```

The path of the file to be used

Parse the rows

```
public class parseRows {
    public static void main(String[] args) throws Exception {

        //Setting up the SparkContext
        SparkConf conf = new SparkConf().setAppName("parseRows");
        JavaSparkContext sc = new JavaSparkContext(conf);

 //Loading the data from airports file
JavaRDD<String> airports = sc.textFile("hdfs:///user/swethakolalapudi/flightDelaysData/airports.csv");
        //The function that will be passed to filter
        class NotHeader implements Function<String,Boolean>{
            public NotHeader() {
            }

            public Boolean call(String s) throws Exception {
                return !s.contains("Description");
            }
        }

        //Filter transformation
        JavaRDD<String> airportsFiltered = airports.filter(new NotHeader());


        //Map transformation
        JavaRDD<List<String>> airportsParsed = airportsFiltered.map(new Function<String, List<String>>() {
            public List<String> call(String s) throws Exception {
                List<String> parsedRow = new ArrayList<String>();
                parsedRow.add(s.substring(0,s.indexOf(",")).replace("\"",""));
                parsedRow.add(s.substring(s.indexOf(",")+1,s.length()-1).replace("\"",""));
                return parsedRow;
            }
        });

        //Lambda function
        JavaRDD<Integer> rowLengths = airportsFiltered.map(s -> s.length());

        //Print out the results
        for (List<String> s:airportsParsed.collect()){
            System.out.println(StringUtils.join(s, "|"));
        }


        //A Pair function - used to create Pair RDDs
        PairFunction<List<String>,String,String> arrayToPair = new PairFunction<List<String>, String, String>() {
            @Override
            public Tuple2<String, String> call(List<String> strings) throws Exception {
                return new Tuple2<>(strings.get(0),strings.get(1));
            }
        };

        //Create a Pair RDD from airports
        JavaPairRDD<String,String> airportsLookup = airportsParsed.mapToPair(arrayToPair);

        //Use it to lookup a particular airport description
        System.out.println(airportsLookup.lookup("PPG"));

    }
}
```

The type of RDD

Parse the rows

```
public class parseRows {
    public static void main(String[] args) throws Exception {

        //Setting up the SparkContext
        SparkConf conf = new SparkConf().setAppName("parseRows");
        JavaSparkContext sc = new JavaSparkContext(conf);

 //Loading the data from airports file
JavaRDD<String> airports = sc.textFile("hdfs:///user/swethakolalapudi/flightDelaysData/airports.csv");
        //The function that will be passed to filter
        class NotHeader implements Function<String,Boolean>{
            public NotHeader() {
            }

            public Boolean call(String s) throws Exception {
                return !s.contains("Description");
            }
        }

        //Filter transformation
        JavaRDD<String> airportsFiltered = airports.filter(new NotHeader());


        //Map transformation
        JavaRDD<List<String>> airportsParsed = airportsFiltered.map(new Function<String, List<String>>() {
            public List<String> call(String s) throws Exception {
                List<String> parsedRow = new ArrayList<String>();
                parsedRow.add(s.substring(0,s.indexOf(",")).replace("\"",""));
                parsedRow.add(s.substring(s.indexOf(",")+1,s.length()-1).replace("\"",""));
                return parsedRow;
            }
        });

        //Lambda function
        JavaRDD<Integer> rowLengths = airportsFiltered.map(s -> s.length());

        //Print out the results
        for (List<String> s:airportsParsed.collect()){
            System.out.println(StringUtils.join(s, "|"));
        }


        //A Pair function - used to create Pair RDDs
        PairFunction<List<String>,String,String> arrayToPair = new PairFunction<List<String>, String, String>() {
            @Override
            public Tuple2<String, String> call(List<String> strings) throws Exception {
                return new Tuple2<>(strings.get(0),strings.get(1));
            }
        };

        //Create a Pair RDD from airports
        JavaPairRDD<String,String> airportsLookup = airportsParsed.mapToPair(arrayToPair);

        //Use it to lookup a particular airport description
        System.out.println(airportsLookup.lookup("PPG"));

    }
}
```

The JavaRDD is a generic class with 1 type parameter

Parse the rows

```
public class parseRows {
    public static void main(String[] args) throws Exception {

        //Setting up the SparkContext
        SparkConf conf = new SparkConf().setAppName("parseRows");
        JavaSparkContext sc = new JavaSparkContext(conf);

 //Loading the data from airports file
JavaRDD<String> airports = sc.textFile("hdfs:///user/swethakolalapudi/flightDelaysData/airports.csv");
        //The function that will be passed to filter()
        class NotHeader implements Function<String,Boolean> {
            public NotHeader() {
            }

            public Boolean call(String s) throws Exception {
                return !s.contains("Description");
            }
        }

        //Filter transformation
        JavaRDD<String> airportsFiltered = airports.filter(new NotHeader());


        //Map transformation
        JavaRDD<List<String>> airportsParsed = airportsFiltered.map(new Function<String, List<String>>() {
            public List<String> call(String s) throws Exception {
                List<String> parsedRow = new ArrayList<String>();
                parsedRow.add(s.substring(0,s.indexOf(",")).replace("\"",""));
                parsedRow.add(s.substring(s.indexOf(",")+1,s.length()-1).replace("\"",""));
                return parsedRow;
            }
        });

        //Lambda function
        JavaRDD<Integer> rowLengths = airportsFiltered.map(s -> s.length());

        //Print out the results
        for (List<String> s:airportsParsed.collect()){
            System.out.println(StringUtils.join(s, "|"));
        }


        //A Pair function - used to create Pair RDDs
        PairFunction<List<String>,String,String> arrayToPair = new PairFunction<List<String>, String, String>() {
            @Override
            public Tuple2<String, String> call(List<String> strings) throws Exception {
                return new Tuple2<>(strings.get(0),strings.get(1));
            }
        };

        //Create a Pair RDD from airports
        JavaPairRDD<String,String> airportsLookup = airportsParsed.mapToPair(arrayToPair);

        //Use it to lookup a particular airport description
        System.out.println(airportsLookup.lookup("PPG"));

    }
}
```

In this case, the RDD is a String RDD

Parse the rows

```
public class parseRows {
    public static void main(String[] args) throws Exception {

        //Setting up the SparkContext
        SparkConf conf = new SparkConf().setAppName("parseRows");
        JavaSparkContext sc = new JavaSparkContext(conf);

//Loading the data from airports file
JavaRDD<String> airports = sc.textFile("hdfs:///user/swethakolalapudi/flightDelaysData/airports.csv");
        //The function that will be passed to filter
        class NotHeader implements Function<String,Boolean>{
            public NotHeader() {
            }

            public Boolean call(String s) throws Exception {
                return !s.contains("Description");
            }
        }

        //Filter transformation
        JavaRDD<String> airportsFiltered = airports.filter(new NotHeader());


        //Map transformation
        JavaRDD<List<String>> airportsParsed = airportsFiltered.map(new Function<String, List<String>>() {
            public List<String> call(String s) throws Exception {
                List<String> parsedRow = new ArrayList<String>();
                parsedRow.add(s.substring(0,s.indexOf(",")).replace("\"",""));
                parsedRow.add(s.substring(s.indexOf(",")+1,s.length()-1).replace("\"",""));
                return parsedRow;
            }
        });

        //Lambda function
        JavaRDD<Integer> rowLengths = airportsFiltered.map(s -> s.length());

        //Print out the results
        for (List<String> s:airportsParsed.collect()){
            System.out.println(StringUtils.join(s, "|"));
        }


        //A Pair function - used to create Pair RDDs
        PairFunction<List<String>,String,String> arrayToPair = new PairFunction<List<String>, String, S
            @Override
            public Tuple2<String, String> call(List<String> strings) throws Exception
                return new Tuple2<>(strings.get(0),strings.get(1));
            }
        };

        //Create a Pair RDD from airports
        JavaPairRDD<String,String> airportsLookup = airportsParsed.mapToPair(arrayToPair);

        //Use it to lookup a particular airport description
        System.out.println(airportsLookup.lookup("PPG"));

    }
}
```

Now, you can apply transformations and actions on the airports RDD

Parse the rows

```
//Loading the data from airports file
JavaRDD<String> airports = sc.textFile("hdfs:///user/swethakolalapudi/flightDelaysData/airports.csv");
```

All the operations like map, filter, reduce, aggregate are available

Parse the rows

```
//Loading the data from airports file
JavaRDD<String> airports = sc.textFile("hdfs:///user/swethakolalapudi/flightDelaysData/airports.csv");
```

map, filter, reduce, aggregate

These operations need a function that will be applied on the records in the RDD

Parse the rows

```
public class parseRows {
    public static void main(String[] args) throws Exception {

        //Setting up the SparkContext
        SparkConf conf = new SparkConf().setAppName("parseRows");
        JavaSparkContext sc = new JavaSparkContext(conf);

//Loading the data from airports file
JavaRDD<String> airports = sc.textFile("hdfs:///user/swethakolalapudi/flightDelaysData/airports.csv");

        //The function that will be passed to filter
        class NotHeader implements Function<String,Boolea
            public NotHeader() {
            }

            public Boolean call(String s) throws Exception {
                return !s.contains("Description");
            }
        }

        //Filter transformation
        JavaRDD<String> airportsFiltered = airports.filter(new NotHeader());


        //Map transformation
        JavaRDD<List<String>> airportsParsed = airportsFiltered.map(new Function<String, List<String>>() {
            public List<String> call(String s) throws Exception {
                List<String> parsedRow = new ArrayList<Stri
                parsedRow.add(s.substring(0,s.indexOf(",")
                parsedRow.add(s.substring(s.indexOf(",")+1,s.l
                return parsedRow;
            }
        });

        //Lambda function
        JavaRDD<Integer> rowLengths = airportsFiltered.map(s -> s.length());

        //Print out the results
        for (List<String> s:airportsParsed.collect()){
            System.out.println(StringUtils.join(s, "|
        }


        //A Pair function - used to create Pair RDDs
        PairFunction<List<String>,String,String> array
            @Override
            public Tuple2<String, String> call(List<String> strings) throws Exception {
                return new Tuple2<>(strings.get(0),strings.get(1));
            }
        };

        //Create a Pair RDD from airports
        JavaPairRDD<String,String> airportsLookup = airportsParsed.mapToPair(arrayToPair);

        //Use it to lookup a particular airport description
        System.out.println(airportsLookup.lookup("PPG"));

    }
}
```

map, filter, reduce, aggregate

The function is represented by a subclass of the Function interface

Parse the rows

```
//Loading the data from airports file
JavaRDD<String> airports = sc.textFile("hdfs:///user/swethakolalapudi/flightDelaysData/airports.csv");
```

map, filter, reduce, aggregate

The function can be passed using

1. A subclass of Function Interface
2. An anonymous Function class
3. Lambda expressions

Parse the rows

```
public class parseRows {
    public static void main(String[] args) throws Exception {

        //Setting up the SparkContext
        SparkConf conf = new SparkConf().setAppName("parseRows");
        JavaSparkContext sc = new JavaSparkContext(conf);

        //Loading the data from airports file
        JavaRDD<String> airports = sc.textFile("hdfs:///user/swethakolalapudi/flightDelaysData/airports.csv");

    //The function that will be passed to filter
    class NotHeader implements Function<String,Boolean> {
      public NotHeader() {}

      public Boolean call(String s) throws Exception {
         return !s.contains("Description");
             }
          }

        //Filter transformation
        JavaRDD<String> airportsFiltered = airports.filter(new NotHeader());


        //Map transformation
        JavaRDD<List<String>> airportsParsed = airportsFiltered.map(new Function<String, List<String>>() {
            public List<String> call(String s) throws Exception {
                List<String> parsedRow = new ArrayList<String>();
                parsedRow.add(s.substring(0,s.indexOf(",")).replace("\"",""));
                parsedRow.add(s.substring(s.indexOf(",")+1,s.length()-1).replace("\"",""));
                return parsedRow;
            }
        });

        //Lambda function
        JavaRDD<Integer> rowLengths = airportsFiltered.map(s -> s.length());

        //Print out the results
        for (List<String> s:airportsParsed.collect()){
            System.out.println(StringUtils.join(s, "|"));
        }


        //A Pair function - used to create Pair RDDs
        PairFunction<List<String>,String,String> arrayToPair = new PairFunction<List<String>, String, String>() {
            @Override
            public Tuple2<String, String> call(List<String> strings) throws Exception {
                return new Tuple2<>(strings.get(0),strings.get(1));
            }
        };

        //Create a Pair RDD from airports
        JavaPairRDD<String,String> airportsLookup = airportsParsed.mapToPair(arrayToPair);

        //Use it to lookup a particular airport description
        System.out.println(airportsLookup.lookup("PPG"));

    }
}
```

Here is a function to filter out the header row

Parse the rows

```
public class parseRows {
    public static void main(String[] args) throws Exception {

        //Setting up the SparkContext
        SparkConf conf = new SparkConf().setAppName("parseRows");
        JavaSparkContext sc = new JavaSparkContext(conf);

        //Loading the data from airports file
        JavaRDD<String> airports = sc.textFile("hdfs:///user/swethakolalapudi/flightDelaysData/airports.csv");

    //The function that will be passed to filter
    class NotHeader implements Function<String,Boolean> {
      public NotHeader() {}

      public Boolean call(String s) throws Exception {
         return !s.contains("Description");
             }
          }

        //Filter transformation
        JavaRDD<String> airportsFiltered = airports.filter(new NotHeader());


        //Map transformation
        JavaRDD<List<String>> airportsParsed = airportsFiltered.map(new Function<String, List<String>>() {
            public List<String> call(String s) throws Exception {
                List<String> parsedRow = new ArrayList<String>();
                parsedRow.add(s.substring(0,s.indexOf(",")).replace("\"",""));
                parsedRow.add(s.substring(s.indexOf(",")+1,s.length()-1).replace("\"",""));
                return parsedRow;
            }
        });

        //Lambda function
        JavaRDD<Integer> rowLengths = airportsFiltered.map(s -> s.length());

        //Print out the results
        for (List<String> s:airportsParsed.collect()){
            System.out.println(StringUtils.join(s, "|"));
        }


        //A Pair function - used to create Pair RDDs
        PairFunction<List<String>,String,String> arrayToPair = new PairFunction<List<String>, String, String>() {
            @Override
            public Tuple2<String, String> call(List<String> strings) throws Exception {
                return new Tuple2<>(strings.get(0),strings.get(1));
            }
        };

        //Create a Pair RDD from airports
        JavaPairRDD<String,String> airportsLookup = airportsParsed.mapToPair(arrayToPair);

        //Use it to lookup a particular airport description
        System.out.println(airportsLookup.lookup("PPG"));

    }
}
```

Spark functions need to implement the Function interface

Parse the rows

```
public class parseRows {
    public static void main(String[] args) throws Exception {

        //Setting up the SparkContext
        SparkConf conf = new SparkConf().setAppName("parseRows");
        JavaSparkContext sc = new JavaSparkContext(conf);

        //Loading the data from airports file
        JavaRDD<String> airports = sc.textFile("hdfs:///user/swethakolalapudi/flightDelaysData/airports.csv");

    //The function that will be passed to filter

    class NotHeader implements Function<String,Boolean> {
      public NotHeader() {}

      public Boolean call(String s) throws Exception {
          return !s.contains("Description");
              }
           }

        //Filter transformation
        JavaRDD<String> airportsFiltered = airports.filter(new NotHeader());

        //Map transformation
        JavaRDD<List<String>> airportsParsed = airportsFiltered.map(new Function<String, List<String>>() {
            public List<String> call(String s) throws Exception {
                List<String> parsedRow = new ArrayList<String>();
                parsedRow.add(s.substring(0,s.indexOf(",")).replace("\"",""));
                parsedRow.add(s.substring(s.indexOf(",")+1,s.length()-1).replace("\"",""));
                return parsedRow;
            }
        });

        //Lambda function
        JavaRDD<Integer> rowLengths = airportsFiltered.map(s -> s.length());

        //Print out the results
        for (List<String> s:airportsParsed.collect()){
            System.out.println(StringUtils.join(s, "|"));
        }

        //A Pair function - used to create Pair RDDs
        PairFunction<List<String>,String,String> arrayToPair = new PairFunction<List<String>, String, String>() {
            @Override
            public Tuple2<String, String> call(List<String> strings) throws Exception {
                return new Tuple2<>(strings.get(0),strings.get(1));
            }
        };

        //Create a Pair RDD from airports
        JavaPairRDD<String,String> airportsLookup = airportsParsed.mapToPair(arrayToPair);

        //Use it to lookup a particular airport description
        System.out.println(airportsLookup.lookup("PPG"));

    }
}
```

The **Function** interface has 2 type parameters

Parse the rows

```
public class parseRows {
    public static void main(String[] args) throws Exception {

        //Setting up the SparkContext
        SparkConf conf = new SparkConf().setAppName("parseRows");
        JavaSparkContext sc = new JavaSparkContext(conf);

        //Loading the data from airports file
        JavaRDD<String> airports = sc.textFile("hdfs:///user/swethakolalapudi/flightDelaysData/airports.csv");

        //Filter transformation
        JavaRDD<String> airportsFiltered = airports.filter(new NotHeader());

        //Map transformation
        JavaRDD<List<String>> airportsParsed = airportsFiltered.map(new Function<String, List<String>>() {
            public List<String> call(String s) throws Exception {
                List<String> parsedRow = new ArrayList<String>();
                parsedRow.add(s.substring(0,s.indexOf(",")).replace("\"",""));
                parsedRow.add(s.substring(s.indexOf(",")+1,s.length()-1).replace("\"",""));
                return parsedRow;
            }
        });

        //Lambda function
        JavaRDD<Integer> rowLengths = airportsFiltered.map(s -> s.length());

        //Print out the results
        for (List<String> s:airportsParsed.collect()){
            System.out.println(StringUtils.join(s, "|"));
        }

        //A Pair function - used to create Pair RDDs
        PairFunction<List<String>,String,String> arrayToPair = new PairFunction<List<String>, String, String>() {
            @Override
            public Tuple2<String, String> call(List<String> strings) throws Exception {
                return new Tuple2<>(strings.get(0),strings.get(1));
            }
        };

        //Create a Pair RDD from airports
        JavaPairRDD<String,String> airportsLookup = airportsParsed.mapToPair(arrayToPair);

        //Use it to lookup a particular airport description
        System.out.println(airportsLookup.lookup("PPG"));

    }
}
```

```
//The function that will be passed to filter

class NotHeader implements Function<String, Boolean> {
  public NotHeader() {}

  public Boolean call(String s) throws Exception {
     return !s.contains("Description");
         }
      }
```

The Input type

Parse the rows

```
public class parseRows {
    public static void main(String[] args) throws Exception {

        //Setting up the SparkContext
        SparkConf conf = new SparkConf().setAppName("parseRows");
        JavaSparkContext sc = new JavaSparkContext(conf);

        //Loading the data from airports file
        JavaRDD<String> airports = sc.textFile("hdfs:///user/swethakolalapudi/flightDelaysData/airports.csv");

    //The function that will be passed to filter
    class NotHeader implements Function<String, Boolean> {
      public NotHeader() {}

      public Boolean call(String s) throws Exception {
         return !s.contains("Description");
             }
          }

        //Filter transformation
        JavaRDD<String> airportsFiltered = airports.filter(new NotHeader());

        //Map transformation
        JavaRDD<List<String>> airportsParsed = airportsFiltered.map(new Function<String, List<String>>() {
            public List<String> call(String s) throws Exception {
                List<String> parsedRow = new ArrayList<String>();
                parsedRow.add(s.substring(0,s.indexOf(",")).replace("\"",""));
                parsedRow.add(s.substring(s.indexOf(",")+1,s.length()-1).replace("\"",""));
                return parsedRow;
            }
        });

        //Lambda function
        JavaRDD<Integer> rowLengths = airportsFiltered.map(s -> s.length());

        //Print out the results
        for (List<String> s:airportsParsed.collect()){
            System.out.println(StringUtils.join(s, "|"));
        }

        //A Pair function - used to create Pair RDDs
        PairFunction<List<String>,String,String> arrayToPair = new PairFunction<List<String>, String, String>() {
            @Override
            public Tuple2<String, String> call(List<String> strings) throws Exception {
                return new Tuple2<>(strings.get(0),strings.get(1));
            }
        };

        //Create a Pair RDD from airports
        JavaPairRDD<String,String> airportsLookup = airportsParsed.mapToPair(arrayToPair);

        //Use it to lookup a particular airport description
        System.out.println(airportsLookup.lookup("PPG"));

    }
}
```

This should match the type of the RDD it will be applied on

airports is a String RDD

Parse the rows

```
public class parseRows {
    public static void main(String[] args) throws Exception {

        //Setting up the SparkContext
        SparkConf conf = new SparkConf().setAppName("parseRows");
        JavaSparkContext sc = new JavaSparkContext(conf);

        //Loading the data from airports file
        JavaRDD<String> airports = sc.textFile("hdfs:///user/swethakolalapudi/flightDelaysData/airports.csv");

        //Filter transformation
        JavaRDD<String> airportsFiltered = airports.filter(new NotHeader());

        //Map transformation
        JavaRDD<List<String>> airportsParsed = airportsFiltered.map(new Function<String, List<String>>() {
            public List<String> call(String s) throws Exception {
                List<String> parsedRow = new ArrayList<String>();
                parsedRow.add(s.substring(0,s.indexOf(",")).replace("\"",""));
                parsedRow.add(s.substring(s.indexOf(",")+1,s.length()-1).replace("\"",""));
                return parsedRow;
            }
        });

        //Lambda function
        JavaRDD<Integer> rowLengths = airportsFiltered.map(s -> s.length());

        //Print out the results
        for (List<String> s:airportsParsed.collect()){
            System.out.println(StringUtils.join(s, "|"));
        }

        //A Pair function - used to create Pair RDDs
        PairFunction<List<String>,String,String> arrayToPair = new PairFunction<List<String>, String, String>() {
            @Override
            public Tuple2<String, String> call(List<String> strings) throws Exception {
                return new Tuple2<>(strings.get(0),strings.get(1));
            }
        };

        //Create a Pair RDD from airports
        JavaPairRDD<String,String> airportsLookup = airportsParsed.mapToPair(arrayToPair);

        //Use it to lookup a particular airport description
        System.out.println(airportsLookup.lookup("PPG"));

    }
}
```

```
//The function that will be passed to filter

class NotHeader implements Function<String,Boolean> {
  public NotHeader() {}

  public Boolean call(String s) throws Exception {
     return !s.contains("Description");
         }
     }
```

This is the return type of the function

It is Boolean for functions that will be passed to filter

Parse the rows

```
public class parseRows {
    public static void main(String[] args) throws Exception {

        //Setting up the SparkContext
        SparkConf conf = new SparkConf().setAppName("parseRows");
        JavaSparkContext sc = new JavaSparkContext(conf);

        //Loading the data from airports file
        JavaRDD<String> airports = sc.textFile("hdfs:///user/swethakolalapudi/flightDelaysData/airports.csv");

        //The function that will be passed to filter
        class NotHeader implements Function<String,Boolean> {
          public NotHeader() {}

          public Boolean call(String s) throws Exception {
             return !s.contains("Description");
                 }
           }

        //Filter transformation
        JavaRDD<String> airportsFiltered = airports.filter(new NotHeader());


        //Map transformation
        JavaRDD<List<String>> airportsParsed = airportsFiltered.map(new Function<String, List<String>>() {
            public List<String> call(String s) throws Exception {
                List<String> parsedRow = new ArrayList<String>();
                parsedRow.add(s.substring(0,s.indexOf(",")).replace("\"",""));
                parsedRow.add(s.substring(s.indexOf(",")+1,s.length()-1).replace("\"",""));
                return parsedRow;
            }
        });

        //Lambda function
        JavaRDD<Integer> rowLengths = airportsFiltered.map(s -> s.length());

        //Print out the results
        for (List<String> s:airportsParsed.collect()){
            System.out.println(StringUtils.join(s, "|"));
        }

        //A Pair function - used to create Pair RDDs
        PairFunction<List<String>,String,String> arrayToPair = new PairFunction<List<String>, String, String>() {
            @Override
            public Tuple2<String, String> call(List<String> strings) throws Exception {
                return new Tuple2<>(strings.get(0),strings.get(1));
            }
        };

        //Create a Pair RDD from airports
        JavaPairRDD<String,String> airportsLookup = airportsParsed.mapToPair(arrayToPair);

        //Use it to lookup a particular airport description
        System.out.println(airportsLookup.lookup("PPG"));

    }
}
```

All functions should implement the call method

Parse the rows

```
public class parseRows {
    public static void main(String[] args) throws Exception {

        //Setting up the SparkContext
        SparkConf conf = new SparkConf().setAppName("parseRows");
        JavaSparkContext sc = new JavaSparkContext(conf);

        //Loading the data from airports file
        JavaRDD<String> airports = sc.textFile("hdfs:///user/swethakolalapudi/flightDelaysData/airports.csv");

    //The function that will be passed to filter
    class NotHeader implements Function<String,Boolean> {
      public NotHeader() {}

      public Boolean call(String s) throws Exception {
         return !s.contains("Description");
             }
          }

        //Filter transformation
        JavaRDD<String> airportsFiltered = airports.filter(new NotHeader());


        //Map transformation
        JavaRDD<List<String>> airportsParsed = airportsFiltered.map(new Function<String, List<String>>() {
            public List<String> call(String s) throws Exception {
                List<String> parsedRow = new ArrayList<String>();
                parsedRow.add(s.substring(0,s.indexOf(",")).replace("\"",""));
                parsedRow.add(s.substring(s.indexOf(",")+1,s.length()-1).replace("\"",""));
                return parsedRow;
            }
        });

        //Lambda function
        JavaRDD<Integer> rowLengths = airportsFiltered.map(s -> s.length());

        //Print out the results
        for (List<String> s:airportsParsed.collect()){
            System.out.println(StringUtils.join(s, "|"));
        }


        //A Pair function - used to create Pair RDDs
        PairFunction<List<String>,String,String> arrayToPair = new PairFunction<List<String>, String, String>() {
            @Override
            public Tuple2<String, String> call(List<String> strings) throws Exception {
                return new Tuple2<>(strings.get(0),strings.get(1));
            }
        };

        //Create a Pair RDD from airports
        JavaPairRDD<String,String> airportsLookup = airportsParsed.mapToPair(arrayToPair);

        //Use it to lookup a particular airport description
        System.out.println(airportsLookup.lookup("PPG"));

    }
}
```

This is where the Logic of the function will is implemented

Parse the rows

```
public class parseRows {
    public static void main(String[] args) throws Exception {

        //Setting up the SparkContext
        SparkConf conf = new SparkConf().setAppName("parseRows");
        JavaSparkContext sc = new JavaSparkContext(conf);

        //Loading the data from airports file
        JavaRDD<String> airports = sc.textFile("hdfs:///user/swethakolalapudi/flightDelaysData/airports.csv");

    //The function that will be passed to filter
    class NotHeader implements Function<String,Boolean> {
      public NotHeader() {}

      public Boolean call(String s) throws Exception {
         return !s.contains("Description");
             }
          }

        //Filter transformation
        JavaRDD<String> airportsFiltered = airports.filter(new NotHeader());


        //Map transformation
        JavaRDD<List<String>> airportsParsed = airportsFiltered.map(new Function<String, List<String>>() {
            public List<String> call(String s) throws Exception {
                List<String> parsedRow = new ArrayList<String>();
                parsedRow.add(s.substring(0,s.indexOf(",")).replace("\"",""));
                parsedRow.add(s.substring(s.indexOf(",")+1,s.length()-1).replace("\"",""));
                return parsedRow;
            }
        });

        //Lambda function
        JavaRDD<Integer> rowLengths = airportsFiltered.map(s -> s.length());

        //Print out the results
        for (List<String> s:airportsParsed.collect()){
            System.out.println(StringUtils.join(s, "|"));
        }


        //A Pair function - used to create Pair RDDs
        PairFunction<List<String>,String,String> arrayToPair = new PairFunction<List<String>, String, String>() {
            @Override
            public Tuple2<String, String> call(List<String> strings) throws Exception {
                return new Tuple2<>(strings.get(0),strings.get(1));
            }
        };

        //Create a Pair RDD from airports
        JavaPairRDD<String,String> airportsLookup = airportsParsed.mapToPair(arrayToPair);

        //Use it to lookup a particular airport description
        System.out.println(airportsLookup.lookup("PPG"));

    }
}
```

The input type and return type of the call method should match the Function's type parameters

Parse the rows

```
public class parseRows {
    public static void main(String[] args) throws Exception {

        //Setting up the SparkContext
        SparkConf conf = new SparkConf().setAppName("parseRows");
        JavaSparkContext sc = new JavaSparkContext(conf);

        //Loading the data from airports file
        JavaRDD<String> airports = sc.textFile("hdfs:///user/swethakolalapudi/flightDelaysData/airports.csv");


        //The function that will be passed to filter
        class NotHeader implements Function<String,Boolean> {
            public NotHeader() {
            }

            public Boolean call(String s) throws Exception {
                return !s.contains("Description");
            }
        }

//Filter transformation
JavaRDD<String> airportsFiltered = airports.filter(new NotHeader());


        //Map transformation
        JavaRDD<List<String>> airportsParsed = airportsFiltered.map(new Function<String, List<String>>() {
            public List<String> call(String s) throws Exception {
                List<String> parsedRow = new ArrayList<String>();
                parsedRow.add(s.substring(0,s.indexOf(",")).replace("\"",""));
                parsedRow.add(s.substring(s.indexOf(",")+1,s.length()-1).replace("\"",""));
                return parsedRow;
            }
        });

        //Lambda function
        JavaRDD<Integer> rowLengths = airportsFiltered.map(s -> s.length());

        //Print out the results
        for (List<String> s:airportsParsed.collect()){
            System.out.println(StringUtils.join(s, "|"));
        }


        //A Pair function - used to create Pair RDDs
        PairFunction<List<String>,String,String> arrayToPair = new PairFunction<List<String>, String, String>() {
            @Override
            public Tuple2<String, String> call(List<String> strings) throws Exception {
                return new Tuple2<>(strings.get(0),strings.get(1));
            }
        };

        //Create a Pair RDD from airports
        JavaPairRDD<String,String> airportsLookup = airportsParsed.mapToPair(arrayToPair);

        //Use it to lookup a particular airport description
        System.out.println(airportsLookup.lookup("PPG"));

    }
}
```

You can use this function in a filter transformation

Parse the rows

```
public class parseRows {
    public static void main(String[] args) throws Exception {

        //Setting up the SparkContext
        SparkConf conf = new SparkConf().setAppName("parseRows");
        JavaSparkContext sc = new JavaSparkContext(conf);

        //Loading the data from airports file
        JavaRDD<String> airports = sc.textFile("hdfs:///user/swethakolalapudi/flightDelaysData/airports.csv");


        //The function that will be passed to filter
        class NotHeader implements Function<String,Boolean> {
            public NotHeader() {
            }

            public Boolean call(String s) throws Exception {
                return !s.contains("Description");
            }
        }

//Filter transformation
JavaRDD<String> airportsFiltered = airports.filter(new NotHeader());


        //Map transformation
        JavaRDD<List<String>> airportsParsed = airportsFiltered.map(new Function<String, List<String>>() {
            public List<String> call(String s) throws Exception {
                List<String> parsedRow = new ArrayList<String>();
                parsedRow.add(s.substring(0,s.indexOf(",")).replace("\"",""));
                parsedRow.add(s.substring(s.indexOf(",")+1,s.length()-1).replace("\"",""));
                return parsedRow;
            }
        });

        //Lambda function
        JavaRDD<Integer> rowLengths = airportsFiltered.map(s -> s.length());

        //Print out the results
        for (List<String> s:airportsParsed.collect()){
            System.out.println(StringUtils.join(s, "|"));
        }


        //A Pair function - used to create Pair RDDs
        PairFunction<List<String>,String,String> arrayToPair = new PairFunction<List<String>, String, String>() {
            @Override
            public Tuple2<String, String> call(List<String> strings) throws Exception {
                return new Tuple2<>(strings.get(0),strings.get(1));
            }
        };

        //Create a Pair RDD from airports
        JavaPairRDD<String,String> airportsLookup = airportsParsed.mapToPair(arrayToPair);

        //Use it to lookup a particular airport description
        System.out.println(airportsLookup.lookup("PPG"));

    }
}
```

You can also pass a function using an anonymous class

Parse the rows

```
public class parseRows {
    public static void main(String[] args) throws Exception {

        //Setting up the SparkContext
        SparkConf conf = new SparkConf().setAppName("parseRows");
        JavaSparkContext sc = new JavaSparkContext(conf);

        //Loading the data from airports file
        JavaRDD<String> airports = sc.textFile("hdfs:///user/swethakolalapudi/flightDelaysData/airports.csv");


        //The function that will be passed to filter
        class NotHeader implements Function<String,Boolean> {
            public NotHeader() {
            }

            public Boolean call(String s) throws Exception {
                return !s.contains("Description");
            }
        }

        //Filter transformation
        JavaRDD<String> airportsFiltered = airports.filter(new NotHeader());


    //Map transformation
JavaRDD<List<String>> airportsParsed = airportsFiltered.map(new Function<String, List<String>>() {
                public List<String> call(String s) throws Exception {
                    List<String> parsedRow = new ArrayList<String>();
                    parsedRow.add(s.substring(0,s.indexOf(",")).replace("\"",""));
                    parsedRow.add(s.substring(s.indexOf(",")+1,s.length()-1).replace("\"",""));
                    return parsedRow;
                }
            });


        //Lambda function
        JavaRDD<Integer> rowLengths = airportsFiltered.map(s -> s.length());

        //Print out the results
        for (List<String> s:airportsParsed.collect()){
            System.out.println(StringUtils.join(s, "|"));
        }


        //A Pair function - used to create Pair RDDs
        PairFunction<List<String>,String,String> arrayToPair = new PairFunction<List<String>, String, String>() {
            @Override
            public Tuple2<String, String> call(List<String> strings) throws Exception {
                return new Tuple2<>(strings.get(0),strings.get(1));
            }
        };

        //Create a Pair RDD from airports
        JavaPairRDD<String,String> airportsLookup = airportsParsed.mapToPair(arrayToPair);

        //Use it to lookup a particular airport description
        System.out.println(airportsLookup.lookup("PPG"));

    }
}
```

Here is a map transformation using an anonymous class

Parse the rows

```
public class parseRows {
    public static void main(String[] args) throws Exception {

        //Setting up the SparkContext
        SparkConf conf = new SparkConf().setAppName("parseRows");
        JavaSparkContext sc = new JavaSparkContext(conf);

        //Loading the data from airports file
        JavaRDD<String> airports = sc.textFile("hdfs:///user/swethakolalapudi/flightDelaysData/airports.csv");


        //The function that will be passed to filter
        class NotHeader implements Function<String,Boolean> {
            public NotHeader() {
            }

            public Boolean call(String s) throws Exception {
                return !s.contains("Description");
            }
        }

        //Filter transformation
        JavaRDD<String> airportsFiltered = airports.filter(new NotHeader());


        //Map transformation
        JavaRDD<List<String>> airportsParsed = airportsFiltered.map(new Function<String, List<String>>() {
                    public List<String> call(String s) throws Exception {
                        List<String> parsedRow = new ArrayList<String>();
                        parsedRow.add(s.substring(0,s.indexOf(",")).replace("\"",""));
                        parsedRow.add(s.substring(s.indexOf(",")+1,s.length()-1).replace("\"",""));
                        return parsedRow;
                    }
                });


        //Lambda function
        JavaRDD<Integer> rowLengths = airportsFiltered.map(s -> s.length());

        //Print out the results
        for (List<String> s:airportsParsed.collect()){
            System.out.println(StringUtils.join(s, "|"));
        }


        //A Pair function - used to create Pair RDDs
        PairFunction<List<String>,String,String> arrayToPair = new PairFunction<List<String>, String, String>() {
            @Override
            public Tuple2<String, String> call(List<String> strings) throws Exception {
                return new Tuple2<>(strings.get(0),strings.get(1));
            }
        };

        //Create a Pair RDD from airports
        JavaPairRDD<String,String> airportsLookup = airportsParsed.mapToPair(arrayToPair);

        //Use it to lookup a particular airport description
        System.out.println(airportsLookup.lookup("PPG"));

    }
}
```

This parses the row and splits it into a list which is (Airport Code, Airport Description)

Parse the rows

```
public class parseRows {
    public static void main(String[] args) throws Exception {

        //Setting up the SparkContext
        SparkConf conf = new SparkConf().setAppName("parseRows");
        JavaSparkContext sc = new JavaSparkContext(conf);

        //Loading the data from airports file
        JavaRDD<String> airports = sc.textFile("hdfs:///user/swethakolalapudi/flightDelaysData/airports.csv");


        //The function that will be passed to filter
        class NotHeader implements Function<String,Boolean> {
            public NotHeader() {
            }

            public Boolean call(String s) throws Exception {
                return !s.contains("Description");
            }
        }

        //Filter transformation
        JavaRDD<String> airportsFiltered = airports.filter(new NotHeader());


        //Map transformation

JavaRDD<List<String>> airportsParsed = airportsFiltered.map(new Function<String, List<String>>() {
            public List<String> call(String s) throws Exception {
                List<String> parsedRow = new ArrayList<String>();
                parsedRow.add(s.substring(0,s.indexOf(",")).replace("\"",""));
                parsedRow.add(s.substring(s.indexOf(",")+1,s.length()-1).replace("\"",""));
                return parsedRow;
            }
        });



        //Lambda function
        JavaRDD<Integer> rowLengths = airportsFiltered.map(s -> s.length());

        //Print out the results
        for (List<String> s:airportsParsed.collect()){
            System.out.println(StringUtils.join(s, "|"));
        }


        //A Pair function - used to create Pair RDDs
        PairFunction<List<String>,String,String> arrayToPair = new Pair[illegible]Function<List<String>, String, String>() {
            @Override
            public Tuple2<String, String> call(List<String> strings) throws Exception {
                return new Tuple2<>(strings.get(0),strings.get(1));
            }
        };

        //Create a Pair RDD from airports
        JavaPairRDD<String,String> airportsLookup = airportsParsed.mapToPair(arrayToPair);

        //Use it to lookup a particular airport description
        System.out.println(airportsLookup.lookup("PPG"));

    }
}
```

This parsing logic is enclosed in an anonymous class

Parse the rows

```
public class parseRows {
    public static void main(String[] args) throws Exception {

        //Setting up the SparkContext
        SparkConf conf = new SparkConf().setAppName("parseRows");
        JavaSparkContext sc = new JavaSparkContext(conf);

        //Loading the data from airports file
        JavaRDD<String> airports = sc.textFile("hdfs:///user/swethakolalapudi/flightDelaysData/airports.csv");


        //The function that will be passed to filter
        class NotHeader implements Function<String,Boolean> {
            public NotHeader() {
            }

            public Boolean call(String s) throws Exception {
                return !s.contains("Description");
            }
        }

        //Filter transformation
        JavaRDD<String> airportsFiltered = airports.filter(new NotHeader());


        //Map transformation
JavaRDD<List<String>> airportsParsed = airportsFiltered.map(new Function<String, List<String>>() {
            public List<String> call(String s) throws Exception {
                List<String> parsedRow = new ArrayList<String>();
                parsedRow.add(s.substring(0,s.indexOf(",")).replace("\"",""));
                parsedRow.add(s.substring(s.indexOf(",")+1,s.length()-1).replace("\"",""));
                return parsedRow;
            }
        });



        //Lambda function
        JavaRDD<Integer> rowLengths = airportsFiltered.map(s -> s.leng

        //Print out the results
        for (List<String> s:airportsParsed.collect()){
            System.out.println(StringUtils.join(s, "|"));
        }


        //A Pair function - used to create Pair RDDs
        PairFunction<List<String>,String,String> arrayToPair = new PairFun
            @Override
            public Tuple2<String, String> call(List<String> strings) throws Exception {
                return new Tuple2<>(strings.get(0),strings.get(1));
            }
        };

        //Create a Pair RDD from airports
        JavaPairRDD<String,String> airportsLookup = airportsParsed.mapToPair(arrayToPair);

        //Use it to lookup a particular airport description
        System.out.println(airportsLookup.lookup("PPG"));

    }
}
```

The function operates on a String RDD

Parse the rows

```
public class parseRows {
    public static void main(String[] args) throws Exception {

        //Setting up the SparkContext
        SparkConf conf = new SparkConf().setAppName("parseRows");
        JavaSparkContext sc = new JavaSparkContext(conf);

        //Loading the data from airports file
        JavaRDD<String> airports = sc.textFile("hdfs:///user/swethakolalapudi/flightDelaysData/airports.csv");


        //The function that will be passed to filter
        class NotHeader implements Function<String,Boolean> {
            public NotHeader() {
            }

            public Boolean call(String s) throws Exception {
                return !s.contains("Description");
            }
        }

        //Filter transformation
        JavaRDD<String> airportsFiltered = airports.filter(new NotHeader());


        //Map transformation
JavaRDD<List<String>> airportsParsed = airportsFiltered.map(new Function<String, List<String>>() {
            public List<String> call(String s) throws Exception {
                List<String> parsedRow = new ArrayList<String>();
                parsedRow.add(s.substring(0,s.indexOf(",")).replace("\"",""));
                parsedRow.add(s.substring(s.indexOf(",")+1,s.length()-1).replace("\"",""));
                return parsedRow;
            }
        });



        //Lambda function
        JavaRDD<Integer> rowLengths = airportsFiltered.map(s -> s.len
        //Print out the results
        for (List<String> s:airportsParsed.collect()){
            System.out.println(StringUtils.join(s, "|"));
        }


        //A Pair function - used to create Pair RDDs
        PairFunction<List<String>,String,String> arrayToPair = new PairFunctio
            @Override
            public Tuple2<String, String> call(List<String> strings) throws Exception {
                return new Tuple2<>(strings.get(0),strings.get(1));
            }
        };

        //Create a Pair RDD from airports
        JavaPairRDD<String,String> airportsLookup = airportsParsed.mapToPa

        //Use it to lookup a particular airport description
        System.out.println(airportsLookup.lookup("PPG"));

    }
}
```

It converts the Strings into a Lists of Strings

Parse the rows

```
public class parseRows {
    public static void main(String[] args) throws Exception {

        //Setting up the SparkContext
        SparkConf conf = new SparkConf().setAppName("parseRows");
        JavaSparkContext sc = new JavaSparkContext(conf);

        //Loading the data from airports file
        JavaRDD<String> airports = sc.textFile("hdfs:///user/swethakolalapudi/flightDelaysData/airports.csv");


        //The function that will be passed to filter
        class NotHeader implements Function<String,Boolean> {
            public NotHeader() {
            }

            public Boolean call(String s) throws Exception {
                return !s.contains("Description");
            }
        }

        //Filter transformation
        JavaRDD<String> airportsFiltered = airports.filter(new NotHeader());

    //Map transformation

JavaRDD<List<String>> airportsParsed = airportsFiltered.map(new Function<String, List<String>>() {
                                   public List<String> call(String s) throws Exception {
            List<String> parsedRow = new ArrayList<String>();
            parsedRow.add(s.substring(0,s.indexOf(",")).replace("\"",""));
            parsedRow.add(s.substring(s.indexOf(",")+1,s.length()-1).replace("\"",""));
            return parsedRow;
        }
                });


        //Lambda function
        JavaRDD<Integer> rowLengths = airportsFiltered.map(s -> s.length());

        //Print out the results
        for (List<String> s:airportsParsed.collect()){
            System.out.println(StringUtils.join(s, "|"));
        }


        //A Pair function - used to create Pair RDDs
        PairFunction<List<String>,String,String> arrayToPair = new PairFunction<List<String>, String, String>() {
            @Override
            public Tuple2<String, String> call(List<String> strings) throws Exception {
                return new Tuple2<>(strings.get(0),strings.get(1));
            }
        };

        //Create a Pair RDD from airports
        JavaPairRDD<String,String> airportsLookup = airportsParsed.mapToPair(arrayToPair);

        //Use it to lookup a particular airport description
        System.out.println(airportsLookup.lookup("PPG"));

    }
}
```

The logic needs to be implemented in the call method

Parse the rows

```
public class parseRows {
    public static void main(String[] args) throws Exception {

        //Setting up the SparkContext
        SparkConf conf = new SparkConf().setAppName("parseRows");
        JavaSparkContext sc = new JavaSparkContext(conf);

        //Loading the data from airports file
        JavaRDD<String> airports = sc.textFile("hdfs:///user/swethakolalapudi/flightDelaysData/airports.csv");


        //The function that will be passed to filter
        class NotHeader implements Function<String,Boolean> {
            public NotHeader() {
            }

            public Boolean call(String s) throws Exception {
                return !s.contains("Description");
            }
        }

        //Filter transformation
        JavaRDD<String> airportsFiltered = airports.filter(new NotHeader());


    //Map transformation
JavaRDD<List<String>> airportsParsed = airportsFiltered.map(new Function<String, List<String>>() {

        public List<String> call(String s) throws Exception {
                        List<String> parsedRow = new ArrayList<String>();
                        parsedRow.add(s.substring(0,s.indexOf(",")).replace("\"",""));
                        parsedRow.add(s.substring(s.indexOf(",")+1,s.length()-1).replace("\"",""));
                        return parsedRow;
                    }
                });


        //Lambda function
        JavaRDD<Integer> rowLengths = airportsFiltered.map(s -> s.length());

        //Print out the results
        for (List<String> s:airportsParsed.collect()){
            System.out.println(StringUtils.join(s, "|"));
        }


        //A Pair function - used to create Pair RDDs
        PairFunction<List<String>,String,String> arrayToPair = new PairFunction<List<String>, String, String>() {
            @Override
            public Tuple2<String, String> call(List<String> strings) throws Exception {
                return new Tuple2<>(strings.get(0),strings.get(1));
            }
        };

        //Create a Pair RDD from airports
        JavaPairRDD<String,String> airportsLookup = airportsParsed.mapToPair(arrayToPair);

        //Use it to lookup a particular airport description
        System.out.println(airportsLookup.lookup("PPG"));

    }
}
```

This logic takes the **String** representing a row and **returns a List**

Parse the rows

```
public class parseRows {
    public static void main(String[] args) throws Exception {

        //Setting up the SparkContext
        SparkConf conf = new SparkConf().setAppName("parseRows");
        JavaSparkContext sc = new JavaSparkContext(conf);

        //Loading the data from airports file
        JavaRDD<String> airports = sc.textFile("hdfs:///user/swethakolalapudi/flightDelaysData/airports.csv");


        //The function that will be passed to filter
        class NotHeader implements Function<String,Boolean> {
            public NotHeader() {
            }

            public Boolean call(String s) throws Exception {
                return !s.contains("Description");
            }
        }

        //Filter transformation
        JavaRDD<String> airportsFiltered = airports.filter(new NotHeader());

        //Map transformation
        JavaRDD<List<String>> airportsParsed = airportsFiltered.map(new Function<String, List<String>>() {
                    public List<String> call(String s) throws Exception {
                        List<String> parsedRow = new ArrayList<String>();
                        parsedRow.add(s.substring(0,s.indexOf(",")).replace("\"",""));
                        parsedRow.add(s.substring(s.indexOf(",")+1,s.length()-1).replace("\"",""));
                        return parsedRow;
                    }
                });

        //Lambda function
        JavaRDD<Integer> rowLengths = airportsFiltered.map(s -> s.length());

        //Print out the results
        for (List<String> s:airportsParsed.collect()){
            System.out.println(StringUtils.join(s, "|"));
        }

        //A Pair function - used to create Pair RDDs
        PairFunction<List<String>,String,String> arrayToPair = new PairFunction<List<String>, String, String>() {
            @Override
            public Tuple2<String, String> call(List<String> strings) throws Exception {
                return new Tuple2<>(strings.get(0),strings.get(1));
            }
        };

        //Create a Pair RDD from airports
        JavaPairRDD<String,String> airportsLookup = airportsParsed.mapToPair(arrayToPair);

        //Use it to lookup a particular airport description
        System.out.println(airportsLookup.lookup("PPG"));

    }
}
```

Anonymous classes are great when the function is only needed for **1 time use**

Parse the rows

```
public class parseRows {
    public static void main(String[] args) throws Exception {

        //Setting up the SparkContext
        SparkConf conf = new SparkConf().setAppName("parseRows");
        JavaSparkContext sc = new JavaSparkContext(conf);

        //Loading the data from airports file
        JavaRDD<String> airports = sc.textFile("hdfs:///user/swethakolalapu[illegible]aysData/airports.dat");


        //The function that will be passed to filter
        class NotHeader implements Function<String,Boolean> {
            public NotHeader() {
            }

            public Boolean call(String s) throws Exception {
                return !s.contains("Description");
            }
        }

        //Filter transformation
        JavaRDD<String> airportsFiltered = airports.filter(new NotHeader());


        //Map transformation
        JavaRDD<List<String>> airportsParsed = airportsFiltered.map(new Function<String, List<String>>() {
            public List<String> call(String s) throws Exception {
                List<String> parsedRow = new ArrayList<String>();
                parsedRow.add(s.substring(0,s.indexOf(",")).replace("\"",""));
                parsedRow.add(s.substring(s.indexOf(",")+1,s.length()-1).replace("\"",""));
                return parsedRow;
            }
        });

        //Lambda function
        JavaRDD<Integer> rowLengths = airportsFiltered.map(s -> s.length());

        //Print out the results
        for (List<String> s:airportsParsed.collect()){
            System.out.println(StringUtils.join(s, "|"));
        }


        //A Pair function - used to create Pair RDDs
        PairFunction<List<String>,String,String> arrayToPair = new PairFunction<List<String>, String, String>() {
            @Override
            public Tuple2<String, String> call(List<String> strings) throws Exception {
                return new Tuple2<>(strings.get(0),strings.get(1));
            }
        };

        //Create a Pair RDD from airports
        JavaPairRDD<String,String> airportsLookup = airportsParsed.mapToPair(arrayToPair);

        //Use it to lookup a particular airport description
        System.out.println(airportsLookup.lookup("PPG"));

    }
}
```

In Java 8, you have support for Lambda functions

These can be used for Spark operations

Parse the rows

```
public class parseRows {
    public static void main(String[] args) throws Exception {

        //Setting up the SparkContext
        SparkConf conf = new SparkConf().setAppName("parseRows");
        JavaSparkContext sc = new JavaSparkContext(conf);

        //Loading the data from airports file
        JavaRDD<String> airports = sc.textFile("hdfs:///user/swethakolalapudi/fl[illegible]/airports");


        //The function that will be passed to filter
        class NotHeader implements Function<String,Boolean> {
            public NotHeader() {
            }

            public Boolean call(String s) throws Exception {
                return !s.contains("Description");
            }
        }

        //Filter transformation
        JavaRDD<String> airportsFiltered = airports.filter(new NotHeader());


        //Map transformation
        JavaRDD<List<String>> airportsParsed = airportsFiltered.map(new Function<String, List<String>>() {
            public List<String> call(String s) throws Exception {
                List<String> parsedRow = new ArrayList<String>();
                parsedRow.add(s.substring(0,s.indexOf(",")).replace("\"",""));
                parsedRow.add(s.substring(s.indexOf(",")+1,s.length()-1).replace("\"",""));
                return parsedRow;
            }
        });

        //Lambda function
        JavaRDD<Integer> rowLengths = airportsFiltered.map(s -> s.length());

        //Print out the results
        for (List<String> s:airportsParsed.collect()){
            System.out.println(StringUtils.join(s, "|"));
        }


        //A Pair function - used to create Pair RDDs
        PairFunction<List<String>,String,String> arrayToPair = new PairFunction<List<String>, String, String>() {
            @Override
            public Tuple2<String, String> call(List<String> strings) throws Exception {
                return new Tuple2<>(strings.get(0),strings.get(1));
            }
        };

        //Create a Pair RDD from airports
        JavaPairRDD<String,String> airportsLookup = airportsParsed.mapToPair(arrayToPair);

        //Use it to lookup a particular airport description
        System.out.println(airportsLookup.lookup("PPG"));

    }
}
```

This operation transforms an RDD of Strings into a numeric RDD of row lengths

Parse the rows

```
//Loading the data from airports file
JavaRDD<String> airports = sc.textFile("hdfs:///user/swethakolalapudi/flightDelaysData/airports.csv");
```

map, filter, reduce, aggregate

The function can be passed using

1. A subclass of Function Interface
2. An anonymous Function class
3. Lambda expressions

Parse the rows

```
public class parseRows {
    public static void main(String[] args) throws Exception {

        //Setting up the SparkContext
        SparkConf conf = new SparkConf().setAppName("parseRows");
        JavaSparkContext sc = new JavaSparkContext(conf);

        //Loading the data from airports file
        JavaRDD<String> airports = sc.textFile("hdfs:///user/swethakolalapudi/flightDelaysData/airports.csv");


        //The function that will be passed to filter
        class NotHeader implements Function<String,Boolean> {
            public NotHeader() {
            }

            public Boolean call(String s) throws Exception {
                return !s.contains("Description");
            }
        }

        //Filter transformation
        JavaRDD<String> airportsFiltered = airports.filter(new NotHeader());


        //Map transformation
        JavaRDD<List<String>> airportsParsed = airportsFiltered.map(new
            public List<String> call(String s) throws Exception {
                List<String> parsedRow = new ArrayList<String>();
                parsedRow.add(s.substring(0,s.indexOf(",")).replace("\"",""));
                parsedRow.add(s.substring(s.indexOf(",")+1,s.length()-1).replace("\"",""));
                return parsedRow;
            }
        });

        //Lambda function
        JavaRDD<Integer> rowLengths = airportsFiltered.map(s -> s.length());

        //Print out the results
                for (List<String> s:airportsParsed.collect()){
                    System.out.println(StringUtils.join(s, "|"));
                }


        //A Pair function - used to create Pair RDDs
        PairFunction<List<String>,String,String> arrayToPair = new PairFunction<List<String>, String, String>() {
            @Override
            public Tuple2<String, String> call(List<String> strings) throws Exception {
                return new Tuple2<>(strings.get(0),strings.get(1));
            }
        };

        //Create a Pair RDD from airports
        JavaPairRDD<String,String> airportsLookup = airportsParsed.mapToPair(arrayToPair);

        //Use it to lookup a particular airport description
        System.out.println(airportsLookup.lookup("PPG"));

    }
}
```

Here is an action to get all the elements of the parsed RDD and then print them

Flight delays using Java

```java
public class parseRows {
    public static void main(String[] args) throws Exception {

        //Setting up the SparkContext
        SparkConf conf = new SparkConf().setAppName("parseRows");
        JavaSparkContext sc = new JavaSparkContext(conf);

        //Loading the data from airports file
        JavaRDD<String> airports = sc.textFile("hdfs:///user/swethakolalapudi/flightDelaysData/airports.csv");


        //The function that will be passed to filter
        class NotHeader implements Function<String,Boolean> {
            public NotHeader() {
            }

            public Boolean call(String s) throws Exception {
                return !s.contains("Description");
            }
        }

        //Filter transformation
        JavaRDD<String> airportsFiltered = airports.filter(new NotHeader());


        //Map transformation
        JavaRDD<List<String>> airportsParsed = airportsFiltered.map(new Function<String, List<String>>() {
            public List<String> call(String s) throws Exception {
                List<String> parsedRow = new ArrayList<String>();
                parsedRow.add(s.substring(0,s.indexOf(",")).replace("\"",""));
                parsedRow.add(s.substring(s.indexOf(",")+1,s.length()-1).replace("\"",""));
                return parsedRow;
            }
        });

        //Lambda function
        JavaRDD<Integer> rowLengths = airportsFiltered.map(s -> s.length());

        //Print out the results
        for (List<String> s:airportsParsed.collect()){
            System.out.println(StringUtils.join(s, "|"));
        }


        //A Pair function - used to create Pair RDDs
        PairFunction<List<String>,String,String> arrayToPair = new PairFunction<List<String>, String, String>() {
            @Override
            public Tuple2<String, String> call(List<String> strings) throws Exception {
                return new Tuple2<>(strings.get(0),strings.get(1));
            }
        };

        //Create a Pair RDD from airports
        JavaPairRDD<String,String> airportsLookup = airportsParsed.mapToPair(arrayToPair);

        //Use it to lookup a particular airport description
        System.out.println(airportsLookup.lookup("PPG"));

    }
}
```

Here is a class to

1. Parse the rows of the airports file ✓

2. Create a Pair RDD for looking up the Airport description

Create a Pair RDD

```
public class parseRows {
    public static void main(String[] args) throws Exception {

        //Setting up the SparkContext
        SparkConf conf = new SparkConf().setAppName("parseRows");
        JavaSparkContext sc = new JavaSparkContext(conf);

        //Loading the data from airports file
        JavaRDD<String> airports = sc.textFile("hdfs:///user/swethakolalapudi/flightDelaysData/airports.csv");


        //The function that will be passed to filter
        class NotHeader implements Function<String,Boolean> {
            public NotHeader() {
            }

            public Boolean call(String s) throws Exception {
                return !s.contains("Description");
            }
        }

        //Filter transformation
        JavaRDD<String> airportsFiltered = airports.filter(new NotHeader());


        //Map transformation
        JavaRDD<List<String>> airportsParsed = airportsFiltered.map(new Function<String, List<String>>() {
            public List<String> call(String s) throws Exception {
                List<String> parsedRow = new ArrayList<String>();
                parsedRow.add(s.substring(0,s.indexOf(",")).replace("\"",""));
                parsedRow.add(s.substring(s.indexOf(",")+1,s.length()-1).replace("\"",""));
                return parsedRow;
            }
        });

        //Lambda function
        JavaRDD<Integer> rowLengths = airportsFiltered.map(s -> s.length());

        //Print out the results
        for (List<String> s:airportsParsed.collect()){
            System.out.println(StringUtils.join(s, "|"));
        }
```

In Python and Scala, if an RDD contains tuples, it's automatically inferred as a Pair RDD

```
//A Pair function - used to create Pair RDDs
PairFunction<List<String>,String,String> arrayToPair = new PairFunction<List<String>, String,
String>() {
                @Override
                public Tuple2<String, String> call(List<String> strings) throws Exception {
                    return new Tuple2<>(strings.get(0),strings.get(1));
                }
            };
```

```
        //Create a Pair RDD from airports
        JavaPairRDD<String,String> airportsLookup = airportsParsed.mapToPair(arrayToPair);

        //Use it to lookup a particular airport description
        System.out.println(airportsLookup.lookup("PPG"));

    }
}
```

Create a Pair RDD

```
public class parseRows {
    public static void main(String[] args) throws Exception {

        //Setting up the SparkContext
        SparkConf conf = new SparkConf().setAppName("parseRows");
        JavaSparkContext sc = new JavaSparkContext(conf);

        //Loading the data from airports file
        JavaRDD<String> airports = sc.textFile("hdfs:///user/swethakolalapudi/flightDelaysData/airports.csv");


        //The function that will be passed to filter
        class NotHeader implements Function<String,Boolean> {
            public NotHeader() {
            }

            public Boolean call(String s) throws Exception {
                return !s.contains("Description");
            }
        }

        //Filter transformation
        JavaRDD<String> airportsFiltered = airports.filter(new NotHeader());


        //Map transformation
        JavaRDD<List<String>> airportsParsed = airportsFiltered.map(new Function<String, List<String>>() {
            public List<String> call(String s) throws Exception {
                List<String> parsedRow = new ArrayList<String>();
                parsedRow.add(s.substring(0,s.indexOf(",")).replace("\"",""));
                parsedRow.add(s.substring(s.indexOf(",")+1,s.length()-1).replace("\"",""));
                return parsedRow;
            }
        });

        //Lambda function
        JavaRDD<Integer> rowLengths = airportsFiltered.map(s -> s.length());

        //Print out the results
        for (List<String> s:airportsParsed.collect()){
            System.out.println(StringUtils.join(s, "|"));
        }
```

In Java, you have to explicitly create Pair RDDs

```
//A Pair function - used to create Pair RDDs
PairFunction<List<String>,String,String> arrayToPair = new PairFunction<List<String>, String,
String>() {
                @Override
                public Tuple2<String, String> call(List<String> strings) throws Exception {
                    return new Tuple2<>(strings.get(0),strings.get(1));
                }
            };
```

```
        //Create a Pair RDD from airports
        JavaPairRDD<String,String> airportsLookup = airportsParsed.mapToPair(arrayToPair);

        //Use it to lookup a particular airport description
        System.out.println(airportsLookup.lookup("PPG"));

    }
}
```

Create a Pair RDD

```
public class parseRows {
    public static void main(String[] args) throws Exception {

        //Setting up the SparkContext
        SparkConf conf = new SparkConf().setAppName("parseRows");
        JavaSparkContext sc = new JavaSparkContext(conf);

        //Loading the data from airports file
        JavaRDD<String> airports = sc.textFile("hdfs:///user/swethakolalapudi/flightDelaysData/airports.csv");


        //The function that will be passed to filter
        class NotHeader implements Function<String,Boolean> {
            public NotHeader() {
            }

            public Boolean call(String s) throws Exception {
                return !s.contains("Description");
            }
        }

        //Filter transformation
        JavaRDD<String> airportsFiltered = airports.filter(new NotHeader());


        //Map transformation
        JavaRDD<List<String>> airportsParsed = airportsFiltered.map(new Function<String, List<String>>() {
            public List<String> call(String s) throws Exception {
                List<String> parsedRow = new ArrayList<String>();
                parsedRow.add(s.substring(0,s.indexOf(",")).replace("\"",""));
                parsedRow.add(s.substring(s.indexOf(",")+1,s.length()-1).replace("\"",""));
                return parsedRow;
            }
        });

        //Lambda function
        JavaRDD<Integer> rowLengths = airportsFiltered.map(s -> s.length());

        //Print out the results
        for (List<String> s:airportsParsed.collect()){
            System.out.println(StringUtils.join(s, "|"));
        }

//A Pair function - used to create Pair RDDs
PairFunction<List<String>,String,String> arrayToPair = new PairFunction<List<String>, String,
String>() {
                @Override
                public Tuple2<String, String> call(List<String> strings) throws Exception {
                    return new Tuple2<>(strings.get(0),strings.get(1));
                }
            };

        //Create a Pair RDD from airports
        JavaPairRDD<String,String> airportsLookup = airportsParsed.mapToPair(arrayToPair);

        //Use it to lookup a particular airport description
        System.out.println(airportsLookup.lookup("PPG"));

    }
}
```

Pair RDDs are normally created using a PairFunction

Create a Pair RDD

```
public class parseRows {
    public static void main(String[] args) throws Exception {

        //Setting up the SparkContext
        SparkConf conf = new SparkConf().setAppName("parseRows");
        JavaSparkContext sc = new JavaSparkContext(conf);

        //Loading the data from airports file
        JavaRDD<String> airports = sc.textFile("hdfs:///user/swethakolalapudi/flightDelaysData/airports.csv");


        //The function that will be passed to filter
        class NotHeader implements Function<String,Boolean> {
            public NotHeader() {
            }

            public Boolean call(String s) throws Exception {
                return !s.contains("Description");
            }
        }

        //Filter transformation
        JavaRDD<String> airportsFiltered = airports.filter(new NotHeader());


        //Map transformation
        JavaRDD<List<String>> airportsParsed = airportsFiltered.map(new Function<String, List<String>>() {
            public List<String> call(String s) throws Exception {
                List<String> parsedRow = new ArrayList<String>();
                parsedRow.add(s.substring(0,s.indexOf(",")).replace("\"",""));
                parsedRow.add(s.substring(s.indexOf(",")+1,s.length()-1).replace("\"",""));
                return parsedRow;
            }
        });

        //Lambda function
        JavaRDD<Integer> rowLengths = airportsFiltered.map(s -> s.length());

        //Print out the results
        for (List<String> s:airportsParsed.collect()){
            System.out.println(StringUtils.join(s, "|"));
        }

//A Pair function - used to create Pair RDDs
PairFunction<List<String>,String,String> arrayToPair = new PairFunction<List<String>, String, String>() {
            @Override
            public Tuple2<String, String> call(List<String> strings) throws Exception {
                return new Tuple2<>(strings.get(0),strings.get(1));
            }
        };

        //Create a Pair RDD from airports
        JavaPairRDD<String,String> airportsLookup = airportsParsed.mapToPair(arrayToPair);

        //Use it to lookup a particular airport description
        System.out.println(airportsLookup.lookup("PPG"));

    }
}
```

A PairFunction has 3 type parameters

Create a Pair RDD

```
public class parseRows {
    public static void main(String[] args) throws Exception {

        //Setting up the SparkContext
        SparkConf conf = new SparkConf().setAppName("parseRows");
        JavaSparkContext sc = new JavaSparkContext(conf);

        //Loading the data from airports file
        JavaRDD<String> airports = sc.textFile("hdfs:///user/swethakolalapudi/flightDelaysData/airports.csv");


        //The function that will be passed to filter
        class NotHeader implements Function<String,Boolean> {
            public NotHeader() {
            }

            public Boolean call(String s) throws Exception {
                return !s.contains("Description");
            }
        }

        //Filter transformation
        JavaRDD<String> airportsFiltered = airports.filter(new NotHeader());


        //Map transformation
        JavaRDD<List<String>> airportsParsed = airportsFiltered.map(new Function<String, List<String>>() {
            public List<String> call(String s) throws Exception {
                List<String> parsedRow = new ArrayList<String>();
                parsedRow.add(s.substring(0,s.indexOf(",")).replace("\"",""));
                parsedRow.add(s.substring(s.indexOf(",")+1,s.length()-1).replace("\"",""));
                return parsedRow;
            }
        });

        //Lambda function
        JavaRDD<Integer> rowLengths = airportsFiltered.map(s -> s.length());

        //Print out the results
        for (List<String> s:airportsParsed.collect()){
            System.out.println(StringUtils.join(s, "|"));
        }

//A Pair function s used to create Pair RDDs
PairFunction<List<String>,String,String> arrayToPair = new PairFunction<List<String>, String, String>() {
            @Override
            public Tuple2<String, String> call(List<String> strings) throws Exception {
                return new Tuple2<>(strings.get(0),strings.get(1));
            }
        };

        //Create a Pair RDD from airports
        JavaPairRDD<String,String> airportsLookup = airportsParsed.mapToPair(arrayToPair);

        //Use it to lookup a particular airport description
        System.out.println(airportsLookup.lookup("PPG"));

    }
}
```

The input RDD type

Create a Pair RDD

```
public class parseRows {
    public static void main(String[] args) throws Exception {

        //Setting up the SparkContext
        SparkConf conf = new SparkConf().setAppName("parseRows");
        JavaSparkContext sc = new JavaSparkContext(conf);

        //Loading the data from airports file
        JavaRDD<String> airports = sc.textFile("hdfs:///user/swethakolalapudi/flightDelaysData/airports.csv");


        //The function that will be passed to filter
        class NotHeader implements Function<String,Boolean> {
            public NotHeader() {
            }

            public Boolean call(String s) throws Exception {
                return !s.contains("Description");
            }
        }

        //Filter transformation
        JavaRDD<String> airportsFiltered = airports.filter(new NotHeader());


        //Map transformation
        JavaRDD<List<String>> airportsParsed = airportsFiltered.map(new Function<String, List<String>>() {
            public List<String> call(String s) throws Exception {
                List<String> parsedRow = new ArrayList<String>();
                parsedRow.add(s.substring(0,s.indexOf(",")).replace("\"",""));
                parsedRow.add(s.substring(s.indexOf(",")+1,s.length()-1).replace("\"",""));
                return parsedRow;
            }
        });

        //Lambda function
        JavaRDD<Integer> rowLengths = airportsFiltered.map(s -> s.length());

        //Print out the results
        for (List<String> s:airportsParsed.collect()){
            System.out.println(StringUtils.join(s, "|"));
        }
```

The **key type** and **value type** of the **output** Pair RDD

```
//A Pair function - used to create Pair RDDs
PairFunction<List<String>,String,String> arrayToPair = new PairFunction<List<String>, String, String>() {
            @Override
            public Tuple2<String, String> call(List<String> strings) throws Exception {
                return new Tuple2<>(strings.get(0),strings.get(1));
            }
        };

        //Create a Pair RDD from airports
        JavaPairRDD<String,String> airportsLookup = airportsParsed.mapToPair(arrayToPair);

        //Use it to lookup a particular airport description
        System.out.println(airportsLookup.lookup("PPG"));

    }
}
```

Create a Pair RDD

```
public class parseRows {
    public static void main(String[] args) throws Exception {

        //Setting up the SparkContext
        SparkConf conf = new SparkConf().setAppName("parseRows");
        JavaSparkContext sc = new JavaSparkContext(conf);

        //Loading the data from airports file
        JavaRDD<String> airports = sc.textFile("hdfs:///user/swethakolalapudi/flightDelaysData/airports.csv");


        //The function that will be passed to filter
        class NotHeader implements Function<String,Boolean> {
            public NotHeader() {
            }

            public Boolean call(String s) throws Exception {
                return !s.contains("Description");
            }
        }

        //Filter transformation
        JavaRDD<String> airportsFiltered = airports.filter(new NotHeader());


        //Map transformation
        JavaRDD<List<String>> airportsParsed = airportsFiltered.map(new Function<String, List<String>>() {
            public List<String> call(String s) throws Exception {
                List<String> parsedRow = new ArrayList<String>();
                parsedRow.add(s.substring(0,s.indexOf(",")).replace("\"",""));
                parsedRow.add(s.substring(s.indexOf(",")+1,s.length()-1).replace("\"",""));
                return parsedRow;
            }
        });

        //Lambda function
        JavaRDD<Integer> rowLengths = airportsFiltered.map(s -> s.length());

        //Print out the results
        for (List<String> s:airportsParsed.collect()){
            System.out.println(StringUtils.join(s, "|"));
        }


//A Pair function s used to create Pair RDDs
PairFunction<List<String>,String,String> arrayToPair = new PairFunction<List<String>, String, String>() {
    @Override
    public Tuple2<String, String> call(List<String> strings) throws Exception {
        return new Tuple2<>(strings.get(0),strings.get(1));
    }
};
        //Create a Pair RDD from airports
        JavaPairRDD<String,String> airportsLookup = airportsParsed.mapToPair(arrayToPair);

        //Use it to lookup a particular airport description
        System.out.println(airportsLookup.lookup("PPG"));

    }
}
```

Just like with a Function class, the logic of pair creation is implemented in the call method

Create a Pair RDD

```
public class parseRows {
    public static void main(String[] args) throws Exception {

        //Setting up the SparkContext
        SparkConf conf = new SparkConf().setAppName("parseRows");
        JavaSparkContext sc = new JavaSparkContext(conf);

        //Loading the data from airports file
        JavaRDD<String> airports = sc.textFile("hdfs:///user/swethakolalapudi/flightDelaysData/airports.csv");


        //The function that will be passed to filter
        class NotHeader implements Function<String,Boolean> {
            public NotHeader() {
            }

            public Boolean call(String s) throws Exception {
                return !s.contains("Description");
            }
        }

        //Filter transformation
        JavaRDD<String> airportsFiltered = airports.filter(new NotHeader());


        //Map transformation
        JavaRDD<List<String>> airportsParsed = airportsFiltered.map(new Function<String, List<String>>() {
            public List<String> call(String s) throws Exception {
                List<String> parsedRow = new ArrayList<String>();
                parsedRow.add(s.substring(0,s.indexOf(",")).replace("\"",""));
                parsedRow.add(s.substring(s.indexOf(",")+1,s.length()-1).replace("\"",""));
                return parsedRow;
            }
        });

        //Lambda function
        JavaRDD<Integer> rowLengths = airportsFiltered.map(s -> s.length());

        //Print out the results
        for (List<String> s:airportsParsed.collect()){
            System.out.println(StringUtils.join(s, "|"));
        }


//A Pair function, used to create Pair RDDs
PairFunction<List<String>,String,String> arrayToPair = new PairFunction<List<String>, String, String>() {
    @Override
    public Tuple2<String, String> call(List<String> strings) throws Exception {
        return new Tuple2<>(strings.get(0),strings.get(1));
    }
};
        //Create a Pair RDD from airports
        JavaPairRDD<String,String> airportsLookup = airportsParsed.mapToPair(arrayToPair);

        //Use it to lookup a particular airport description
        System.out.println(airportsLookup.lookup("PPG"));

    }
}
```

The call method should return a tuple

Create a Pair RDD

```
public class parseRows {
    public static void main(String[] args) throws Exception {

        //Setting up the SparkContext
        SparkConf conf = new SparkConf().setAppName("parseRows");
        JavaSparkContext sc = new JavaSparkContext(conf);

        //Loading the data from airports file
        JavaRDD<String> airports = sc.textFile("hdfs:///user/swethakolalapudi/flightDelaysData/airports.csv");


        //The function that will be passed to filter
        class NotHeader implements Function<String,Boolean> {
            public NotHeader() {
            }

            public Boolean call(String s) throws Exception {
                return !s.contains("Description");
            }
        }

        //Filter transformation
        JavaRDD<String> airportsFiltered = airports.filter(new NotHeader());


        //Map transformation
        JavaRDD<List<String>> airportsParsed = airportsFiltered.map(new Function<String, List<String>>() {
            public List<String> call(String s) throws Exception {
                List<String> parsedRow = new ArrayList<String>();
                parsedRow.add(s.substring(0,s.indexOf(",")).replace("\"",""));
                parsedRow.add(s.substring(s.indexOf(",")+1,s.length()-1).replace("\"",""));
                return parsedRow;
            }
        });

        //Lambda function
        JavaRDD<Integer> rowLengths = airportsFiltered.map(s -> s.length());

        //Print out the results
        for (List<String> s:airportsParsed.collect()){
            System.out.println(StringUtils.join(s, "|"));
        }


//A Pair function s used to create Pair RDDs
PairFunction<List<String>,String,String> arrayToPair = new PairFunction<List<String>, String, String>() {
    @Override
    public Tuple2<String, String> call(List<String> strings) throws Exception {
        return new Tuple2<>(strings.get(0),strings.get(1));
    }
};
        //Create a Pair RDD from airports
        JavaPairRDD<String,String> airportsLookup = airportsParsed.mapToPair(arrayToPair);

        //Use it to lookup a particular airport description
        System.out.println(airportsLookup.lookup("PPG"));

    }
}
```

The tuple's type parameters **should match** the key type and value type

Parse the rows

```
public class parseRows {
    public static void main(String[] args) throws Exception {

        //Setting up the SparkContext
        SparkConf conf = new SparkConf().setAppName("parseRows");
        JavaSparkContext sc = new JavaSparkContext(conf);

        //Loading the data from airports file
        JavaRDD<String> airports = sc.textFile("hdfs:///user/swethakolalapudi/flight-data/airports.csv");


        //The function that will be passed to filter
        class NotHeader implements Function<String,Boolean> {
            public NotHeader() {
            }

            public Boolean call(String s) throws Exception {
                return !s.contains("Description");
            }
        }

        //Filter transformation
        JavaRDD<String> airportsFiltered = airports.filter(new NotHeader());


        //Map transformation
        JavaRDD<List<String>> airportsParsed = airportsFiltered.map(new Function<String, List<String>>() {
            public List<String> call(String s) throws Exception {
                List<String> parsedRow = new ArrayList<String>();
                parsedRow.add(s.substring(0,s.indexOf(",")).replace("\"",""));
                parsedRow.add(s.substring(s.indexOf(",")+1,s.length()-1).replace("\"",""));
                return parsedRow;
            }
        });

        //Lambda function
        JavaRDD<Integer> rowLengths = airportsFiltered.map(s -> s.length());

        //Print out the results
        for (List<String> s:airportsParsed.collect()){
            System.out.println(StringUtils.join(s, "|"));
        }


        //A Pair function - used to create Pair RDDs
        PairFunction<List<String>,String,String> arrayToPair = new PairFunction<List<String>, String, String>() {
            @Override
            public Tuple2<String, String> call(List<String> strings) throws Exception {
                return new Tuple2<>(strings.get(0),strings.get(1));
            }
        };

    //Create a Pair RDD from airports
    JavaPairRDD<String,String> airportsLookup =
    airportsParsed.mapToPair(arrayToPair);


        //Use it to lookup a particular airport description
        System.out.println(airportsLookup.lookup("PPG"));

    }
}
```

Once you have a PairFunction, you can use it to create a Pair RDD

Parse the rows

```
public class parseRows {
    public static void main(String[] args) throws Exception {

        //Setting up the SparkContext
        SparkConf conf = new SparkConf().setAppName("parseRows");
        JavaSparkContext sc = new JavaSparkContext(conf);

        //Loading the data from airports file
        JavaRDD<String> airports = sc.textFile("hdfs:///user/swethakolalapudi/flightDelaysData/airports.csv");

        //The function that will be passed to filter
        class NotHeader implements Function<String,Boolean> {
            public NotHeader() {
            }

            public Boolean call(String s) throws Exception {
                return !s.contains("Description");
            }
        }

        //Filter transformation
        JavaRDD<String> airportsFiltered = airports.filter(new NotHeader());

        //Map transformation
        JavaRDD<List<String>> airportsParsed = airportsFiltered.map(new Function<String, List<String>>() {
            public List<String> call(String s) throws Exception {
                List<String> parsedRow = new ArrayList<String>();
                parsedRow.add(s.substring(0,s.indexOf(",")).replace("\"",""));
                parsedRow.add(s.substring(s.indexOf(",")+1,s.length()-1).replace("\"",""));
                return parsedRow;
            }
        });

        //Lambda function
        JavaRDD<Integer> rowLengths = airportsFiltered.map(s -> s.length());

        //Print out the results
        for (List<String> s:airportsParsed.collect()){
            System.out.println(StringUtils.join(s, "|"));
        }

        //A Pair function - used to create Pair RDDs
        PairFunction<List<String>,String,String> arrayToPair = new PairFunction<List<String>, String, String>() {
            @Override
            public Tuple2<String, String> call(List<String> strings) throws Exception {
                return new Tuple2<>(strings.get(0),strings.get(1));
            }
        };
```

The PairFunction object is passed to the mapToPair operation

```
    //Create a Pair RDD from airports
JavaPairRDD<String,String> airportsLookup =
airportsParsed.mapToPair(arrayToPair);
```

```
        //Use it to lookup a particular airport description
        System.out.println(airportsLookup.lookup("PPG"));

    }
}
```

Parse the rows

```
public class parseRows {
    public static void main(String[] args) throws Exception {

        //Setting up the SparkContext
        SparkConf conf = new SparkConf().setAppName("parseRows");
        JavaSparkContext sc = new JavaSparkContext(conf);

        //Loading the data from airports file
        JavaRDD<String> airports = sc.textFile("hdfs:///user/swethakolalapudi/flightDelaysData/airports.csv");

        //The function that will be passed to filter
        class NotHeader implements Function<String,Boolean> {
            public NotHeader() {
            }

            public Boolean call(String s) throws Exception {
                return !s.contains("Description");
            }
        }

        //Filter transformation
        JavaRDD<String> airportsFiltered = airports.filter(new NotHeader());

        //Map transformation
        JavaRDD<List<String>> airportsParsed = airportsFiltered.map(new Function<String, List<String>>() {
            public List<String> call(String s) throws Exception {
                List<String> parsedRow = new ArrayList<String>();
                parsedRow.add(s.substring(0,s.indexOf(",")).replace("\"",""));
                parsedRow.add(s.substring(s.indexOf(",")+1,s.length()-1).replace("\"",""));
                return parsedRow;
            }
        });

        //Lambda function
        JavaRDD<Integer> rowLengths = airportsFiltered.map(s -> s.length());

        //Print out the results
        for (List<String> s:airportsParsed.collect()){
            System.out.println(StringUtils.join(s, "|"));
        }

        //A Pair function - used to create Pair RDDs
        PairFunction<List<String>,String,String> arrayToPair = new PairFunction<List<String>, String, String>() {
            @Override
            public Tuple2<String, String> call(List<String> strings) throws Exception {
                return new Tuple2<>(strings.get(0),strings.get(1));
            }
        };

    //Create a Pair RDD from airports
    JavaPairRDD<String,String> airportsLookup =
    airportsParsed.mapToPair(arrayToPair);


        //Use it to lookup a particular airport description
        System.out.println(airportsLookup.lookup("PPG"));

    }
}
```

The mapToPair operation is exclusive to Java

Parse the rows

```
public class parseRows {
    public static void main(String[] args) throws Exception {

        //Setting up the SparkContext
        SparkConf conf = new SparkConf().setAppName("parseRows");
        JavaSparkContext sc = new JavaSparkContext(conf);

        //Loading the data from airports file
        JavaRDD<String> airports = sc.textFile("hdfs:///user/swethako[illegible]

        //The function that will be passed to filter
        class NotHeader implements Function<String,Boolean> {
            public NotHeader() {
            }

            public Boolean call(String s) throws Exception {
                return !s.contains("Description");
            }
        }

        //Filter transformation
        JavaRDD<String> airportsFiltered = airports.filter(new NotHeader());


        //Map transformation
        JavaRDD<List<String>> airportsParsed = airportsFiltered.map(new Function<String, List<String>>() {
            public List<String> call(String s) throws Exception {
                List<String> parsedRow = new ArrayList<String>();
                parsedRow.add(s.substring(0,s.indexOf(",")).replace("\"",""));
                parsedRow.add(s.substring(s.indexOf(",")+1,s.length()-1).replace("\"",""));
                return parsedRow;
            }
        });

        //Lambda function
        JavaRDD<Integer> rowLengths = airportsFiltered.map(s -> s.length());

        //Print out the results
        for (List<String> s:airportsParsed.collect()){
            System.out.println(StringUtils.join(s, "|"));
        }


        //A Pair function - used to create Pair RDDs
        PairFunction<List<String>,String,String> arrayToPair = new PairFunction<List<String>, String, String>() {
            @Override
            public Tuple2<String, String> call(List<String> strings) throws Exception {
                return new Tuple2<>(strings.get(0),strings.get(1));
            }
        };

        //Create a Pair RDD from airports


JavaPairRDD<String,String> airportsLookup = airportsParsed.mapToPair(arrayToPair);


        //Use it to lookup a particular airport description
        System.out.println(airportsLookup.lookup("PPG"));

    }
}
```

Once we have a Pair RDD, all Pair RDD operations apply as usual

keys, values, mapValues, reduceByKey, lookup etc

RUNNING YOUR JAVA PROGRAM

You can run your Java code using spark-submit

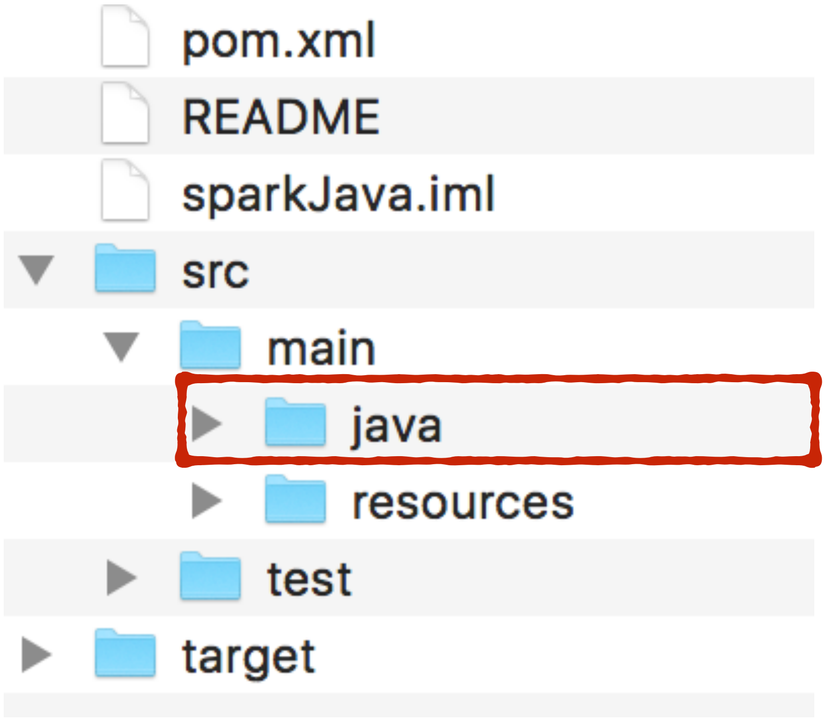
1. Build a JAR using Maven

Maven is necessary as it adds the dependencies required for Spark

2. Submit your JAR to spark-submit

1. Build a JAR using Maven

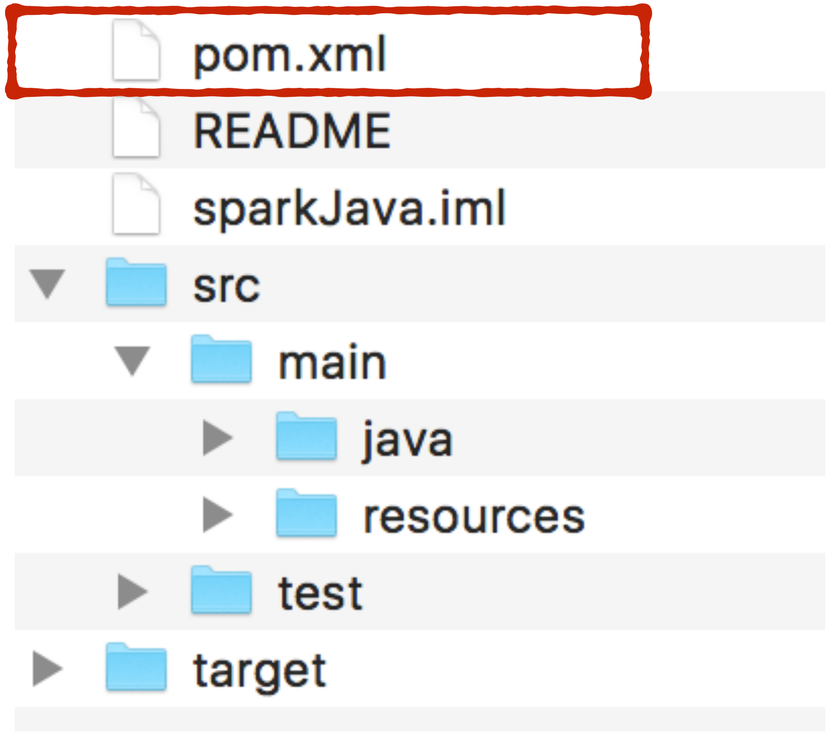
Your source code should be set up using the following directory structure



Keep your main class in src/main/java

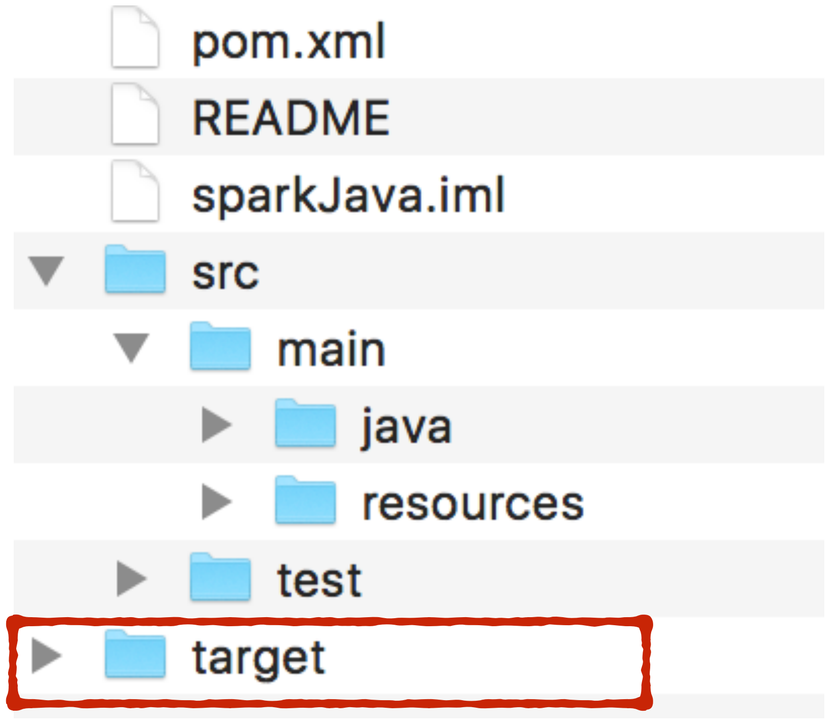
1. Build a JAR using Maven

Set up an pom.xml file



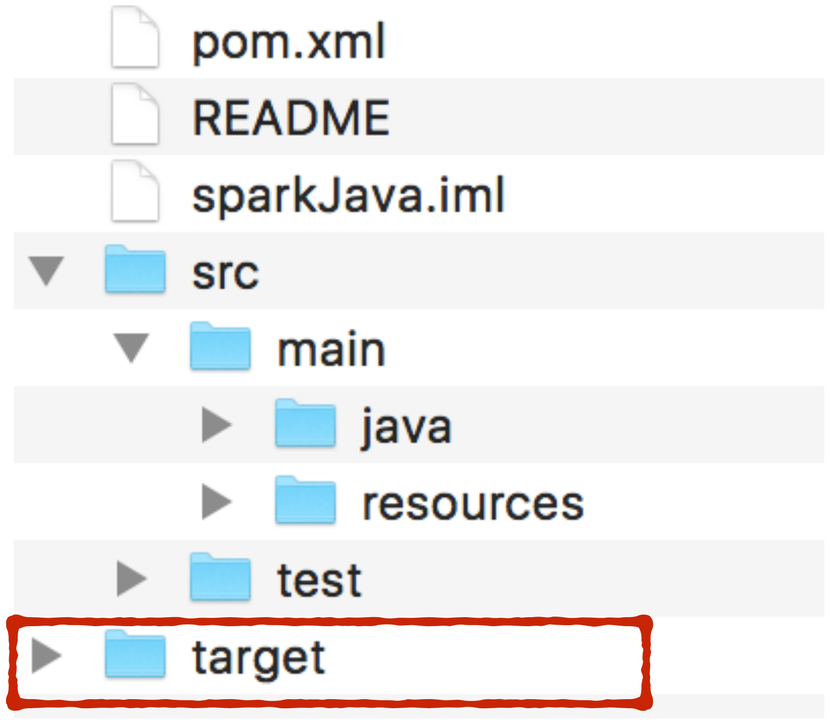
Maven uses this to build a JAR

1. Build a JAR using Maven



The JAR will be saved in the target folder

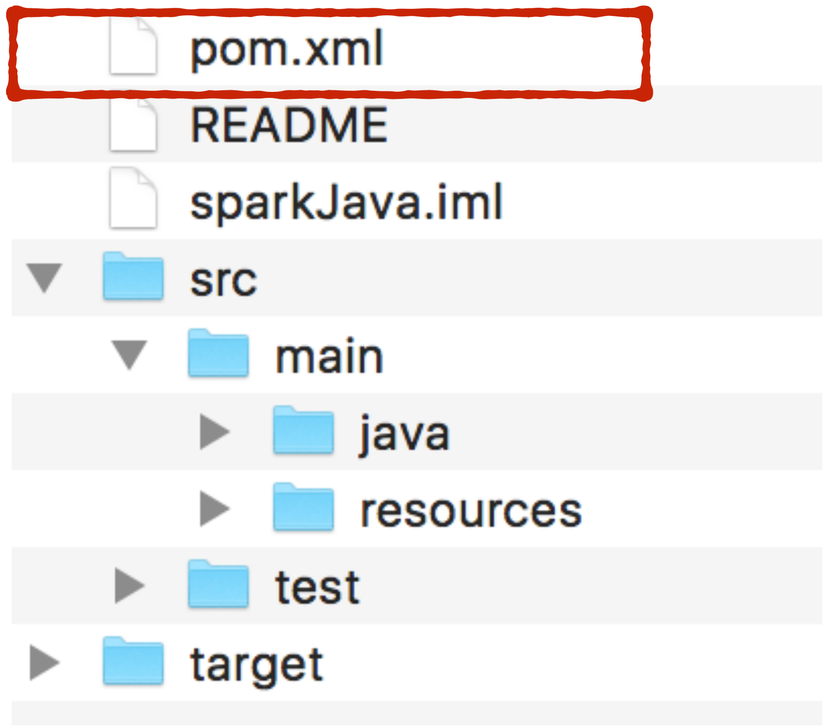
1. Build a JAR using Maven



If you use an IDE like IntelliJ or Eclipse, the directory structure will be set up for you

1. Build a JAR using Maven

This is what your pom.xml should look like




```
<?xml version="1.0" encoding="UTF-8"?>
<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>loonycorn</groupId>
    <artifactId>spark-example</artifactId>
    <version>1.0</version>
    <dependencies>

        <dependency>
            <groupId>org.apache.spark</groupId>
            <artifactId>spark-core_2.10</artifactId>
            <version>1.6.1</version>
            <scope>provided</scope>
        </dependency>

    </dependencies>

    <properties>
        <java.version>1.8
        </java.version>
    </properties>
<build>
    <pluginManagement>
        <plugins>
            <plugin>
                <groupId>
                    org.apache.maven.plugins
                </groupId>
                <artifactId>
                    maven-compiler-plugin
                </artifactId>
                <version>
                    3.1
                </version>
                <configuration>
```

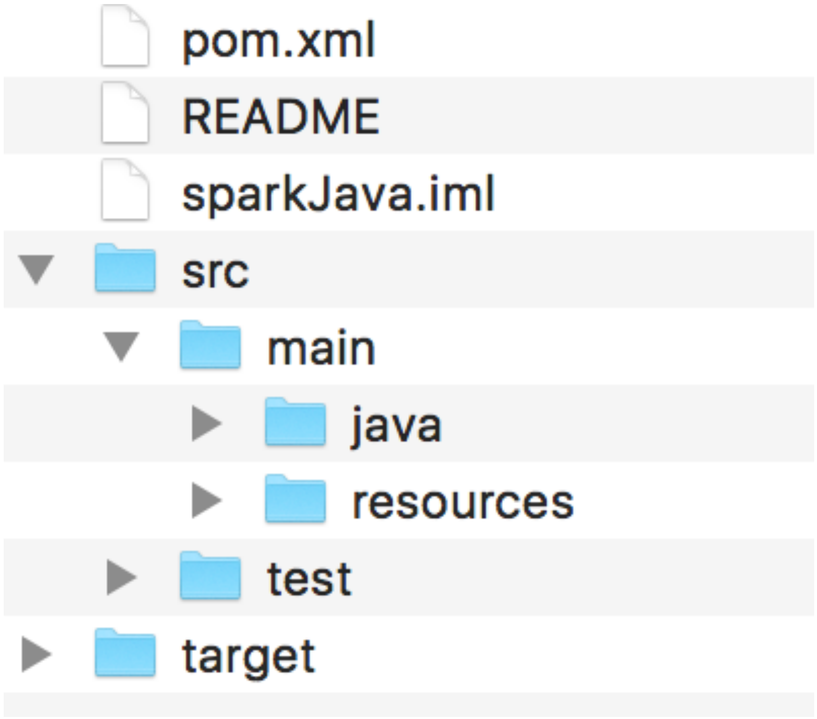
1. Build a JAR using Maven

pom.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<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>loonycorn</groupId>
    <artifactId>spark-example</artifactId>
    <version>1.0</version>

    <dependencies>

        <dependency>
            <groupId>org.apache.spark</groupId>
            <artifactId>spark-core_2.10</artifactId>
            <version>1.6.1</version>
            <scope>provided</scope>
        </dependency>

    </dependencies>

    <properties>
        <java.version>1.8
        </java.version>
    </properties>
<build>
    <pluginManagement>
        <plugins>
            <plugin>
                <groupId>
                    org.apache.maven.plugins
                </groupId>
                <artifactId>
                    maven-compiler-plugin
                </artifactId>
                <version>
```

This sets up the Spark dependency

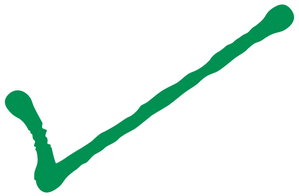
1. Build a JAR using Maven



To build the JAR run the following at the command line in this folder

```
>mvn clean && mvn compile && mvn package
```

You can run your Java code using spark-submit

1. Build a JAR using Maven 

Maven is necessary as it adds the dependencies required for Spark

2. Submit your JAR to spark-submit

2. Submit your JAR to spark-submit

To submit your JAR and
run your program

```
>spark-submit --class parseRows ./target/spark-example-1.0.jar
```

The path to your JAR file

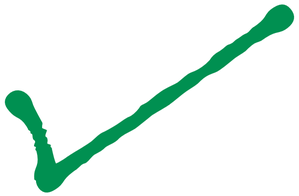
2. Submit your JAR to spark-submit

To submit your JAR and run your program

```
>spark-submit --class parseRows ./target/spark-example-1.0.jar
```

The path of the class within the src/main/java folder

You can run your Java code using spark-submit

1. Build a JAR using Maven 

Maven is necessary as it adds the dependencies required for Spark

2. Submit your JAR to spark-submit