scala> **val** lines **=** sc.textFile("README.md")

scala> lines.count()

scala> lines.first()

scala> **val** lines **=** sc.textFile("README.md")

scala> **val** pythonLines **=** lines.filter(line **=>** line.contains("Python"))

scala> pythonLines.first()

Initializing spark

**val** conf **=** **new** **SparkConf**().setMaster("local").setAppName("My App")

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

here local- clustr URL

word count program

*// Create a Scala Spark Context.*

**val** conf **=** **new** **SparkConf**().setAppName("wordCount")

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

*// Load our input data.*

**val** input **=** sc.textFile(inputFile)

*// Split it up into words.*

**val** words **=** input.flatMap(line **=>** line.split(" "))

*// Transform into pairs and count.*

**val** counts **=** words.map(word **=>** (word, 1)).reduceByKey{**case** (x, y) **=>** x + y}

*// Save the word count back out to a text file, causing evaluation.*

counts.saveAsTextFile(outputFile)

Example 2-15. Maven build and run

mvn clean && mvn compile && mvn package

$SPARK\_HOME/bin/spark-submit **\**

--class com.oreilly.learningsparkexamples.mini.java.WordCount **\**

./target/learning-spark-mini-example-0.0.1.jar **\**

./README.md ./wordcounts

parallelize() method in Scala

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

filter() transformation in Scala

**val** inputRDD **=** sc.textFile("log.txt")

**val** errorsRDD **=** inputRDD.filter(line **=>** line.contains("error"))

Scala error count using actions

println("Input had " + badLinesRDD.count() + " concerning lines")

println("Here are 10 examples:")

badLinesRDD.take(10).foreach(println)

Scala function passing

**class** **SearchFunctions**(**val** query**:** **String**) {

**def** isMatch(s**:** **String**)**:** **Boolean** = {

s.contains(query)

}

**def** getMatchesFunctionReference(rdd**:** **RDD**[**String**])**:** **RDD**[**String**] **=** {

*// Problem: "isMatch" means "this.isMatch", so we pass all of "this"*

rdd.map(isMatch)

}

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

*// Problem: "query" means "this.query", so we pass all of "this"*

rdd.map(x **=>** x.split(query))

}

**def** getMatchesNoReference(rdd**:** **RDD**[**String**])**:** **RDD**[**String**] **=** {

*// Safe: extract just the field we need into a local variable*

**val** query\_ **=** **this**.query

rdd.map(x **=>** x.split(query\_))

}

}

Scala squaring the values in an RDD

**val** input **=** sc.parallelize(**List**(1, 2, 3, 4))

**val** result **=** input.map(x **=>** x \* x)

println(result.collect().mkString(","))

flatMap() in Scala, splitting lines into multiple words

**val** lines **=** sc.parallelize(**List**("hello world", "hi"))

**val** words **=** lines.flatMap(line **=>** line.split(" "))

words.first() *// returns "hello"*

reduce() in Scala

**val** sum **=** rdd.reduce((x, y) **=>** x + y)

aggregate() in Scala

**val** result **=** input.aggregate((0, 0))(

(acc, value) **=>** (acc.\_1 + value, acc.\_2 + 1),

(acc1, acc2) **=>** (acc1.\_1 + acc2.\_1, acc1.\_2 + acc2.\_2))

**val** avg **=** result.\_1 / result.\_2.toDouble

persist() in Scala,unpersist()

**val** result **=** input.map(x **=>** x \* x)

result.persist(**StorageLevel**.**DISK\_ONLY**)

println(result.count())

println(result.collect().mkString(","))

Creating a pair RDD using the first word as the key in Scala

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

Simple filter on second element in Scala

pairs.filter{**case** (key, value) **=>** value.length < 20}

. Per-key average with reduceByKey() and mapValues() in Scala

rdd.mapValues(x **=>** (x, 1)).reduceByKey((x, y) **=>** (x.\_1 + y.\_1, x.\_2 + y.\_2))

Word count in Scala

**val** input **=** sc.textFile("s3://...")

**val** words **=** input.flatMap(x **=>** x.split(" "))

**val** result **=** words.map(x **=>** (x, 1)).reduceByKey((x, y) **=>** x + y)

Per-key average using combineByKey() in Scala

**val** result **=** input.combineByKey(

(v) **=>** (v, 1),

(acc**:** (**Int**, **Int**), v) **=>** (acc.\_1 + v, acc.\_2 + 1),

(acc1**:** (**Int**, **Int**), acc2**:** (**Int**, **Int**)) **=>** (acc1.\_1 + acc2.\_1, acc1.\_2 + acc2.\_2)

).map{ **case** (key, value) **=>** (key, value.\_1 / value.\_2.toFloat) }

result.collectAsMap().map(println(**\_**))

reduceByKey() with custom parallelism in Scala

**val** data **=** **Seq**(("a", 3), ("b", 4), ("a", 1))

sc.parallelize(data).reduceByKey((x, y) **=>** x + y) *// Default parallelism*

sc.parallelize(data).reduceByKey((x, y) **=>** x + y) *// Custom parallelism*

Scala shell inner join

1

storeAddress **=** {

(**Store**("Ritual"), "1026 Valencia St"), (**Store**("Philz"), "748 Van Ness Ave"),

(**Store**("Philz"), "3101 24th St"), (**Store**("Starbucks"), "Seattle")}

storeRating **=** {

(**Store**("Ritual"), 4.9), (**Store**("Philz"), 4.8))}

storeAddress.join(storeRating) == {

(**Store**("Ritual"), ("1026 Valencia St", 4.9)),

(**Store**("Philz"), ("748 Van Ness Ave", 4.8)),

(**Store**("Philz"), ("3101 24th St", 4.8))}

leftOuterJoin() and rightOuterJoin()

storeAddress.leftOuterJoin(storeRating) ==

{(**Store**("Ritual"),("1026 Valencia St",**Some**(4.9))),

(**Store**("Starbucks"),("Seattle",**None**)),

(**Store**("Philz"),("748 Van Ness Ave",**Some**(4.8))),

(**Store**("Philz"),("3101 24th St",**Some**(4.8)))}

storeAddress.rightOuterJoin(storeRating) ==

{(**Store**("Ritual"),(**Some**("1026 Valencia St"),4.9)),

(**Store**("Philz"),(**Some**("748 Van Ness Ave"),4.8)),

(**Store**("Philz"), (**Some**("3101 24th St"),4.8))}

Scala simple application

*// Initialization code; we load the user info from a Hadoop SequenceFile on HDFS.*

*// This distributes elements of userData by the HDFS block where they are found,*

*// and doesn't provide Spark with any way of knowing in which partition a*

*// particular UserID is located.*

**val** sc **=** **new** **SparkContext**(...)

**val** userData **=** sc.sequenceFile[**UserID**, **UserInfo**]("hdfs://...").persist()

*// Function called periodically to process a logfile of events in the past 5 minutes;*

**Data Partitioning (Advanced) | 61**

*// we assume that this is a SequenceFile containing (UserID, LinkInfo) pairs.*

**def** processNewLogs(logFileName**:** **String**) {

**val** events **=** sc.sequenceFile[**UserID**, **LinkInfo**](logFileName)

**val** joined **=** userData.join(events)*// RDD of (UserID, (UserInfo, LinkInfo)) pairs*

**val** offTopicVisits **=** joined.filter {

**case** (userId, (userInfo, linkInfo)) **=>** *// Expand the tuple into its components*

!userInfo.topics.contains(linkInfo.topic)

}.count()

println("Number of visits to non-subscribed topics: " + offTopicVisits)

}

Scala custom partitioner

**val** sc **=** **new** **SparkContext**(...)

**val** userData **=** sc.sequenceFile[**UserID**, **UserInfo**]("hdfs://...")

.partitionBy(**new** **HashPartitioner**(100)) *// Create 100 partitions*

.persist()

Determining partitioner of an RDD

scala> **val** pairs **=** sc.parallelize(**List**((1, 1), (2, 2), (3, 3)))

pairs**:** **spark.RDD**[(**Int**, **Int**)] **=** **ParallelCollectionRDD**[0] at parallelize at <console**>:**12

scala> pairs.partitioner

res0**:** **Option**[**spark.Partitioner**] **=** **None**

scala> **val** partitioned **=** pairs.partitionBy(**new** spark.**HashPartitioner**(2))

partitioned**:** **spark.RDD**[(**Int**, **Int**)] **=** **ShuffledRDD**[1] at partitionBy at <console**>:**14

scala> partitioned.partitioner

res1**:** **Option**[**spark.Partitioner**] **=** **Some**([spark.**HashPartitioner@**5147788d](mailto:spark.HashPartitioner@5147788d))

benefit from

partitioning are cogroup(), groupWith(), join(), leftOuterJoin(), rightOuter

Join(), groupByKey(), reduceByKey(), combineByKey(), and lookup().

Scala PageRank

*// Assume that our neighbor list was saved as a Spark objectFile*

**val** links **=** sc.objectFile[(**String**, **Seq**[**String**])]("links")

.partitionBy(**new** **HashPartitioner**(100))

.persist()

*// Initialize each page's rank to 1.0; since we use mapValues, the resulting RDD*

*// will have the same partitioner as links*

**var** ranks **=** links.mapValues(v **=>** 1.0)

*// Run 10 iterations of PageRank*

**for** (i **<-** 0 until 10) {

**val** contributions **=** links.join(ranks).flatMap {

**case** (pageId, (links, rank)) **=>**

links.map(dest **=>** (dest, rank / links.size))

}

ranks **=** contributions.reduceByKey((x, y) **=>** x + y).mapValues(v **=>** 0.15 + 0.85\*v)

}

*// Write out the final ranks*

ranks.saveAsTextFile("ranks")

shows how we would write the domain-name-based partitioner

sketched previously, which hashes only the domain name of each URL.

Example 4-26. Scala custom partitioner

**class** **DomainNamePartitioner**(numParts**:** **Int**) **extends** **Partitioner** {

**override** **def** numPartitions**:** **Int** = numParts

**override** **def** getPartition(key**:** **Any**)**:** **Int** = {

**val** domain **=** **new** **Java**.net.**URL**(key.toString).getHost()

**val** code **=** (domain.hashCode % numPartitions)

**if** (code < 0) {

code + numPartitions *// Make it non-negative*

} **else** {

code

}

}

*// Java equals method to let Spark compare our Partitioner objects*

**override** **def** equals(other**:** **Any**)**:** **Boolean** = other **match** {

**case** dnp**:** **DomainNamePartitioner** =>

dnp.numPartitions == numPartitions

**case** **\_** **=>**

**false**

}

}

Loading a text file in Scala

**val** input **=** sc.textFile("file:///home/holden/repos/spark/README.md")

Example 5-4. Average value per file in Scala

**val** input **=** sc.wholeTextFiles("file://home/holden/salesFiles")

**val** result **=** input.mapValues{y **=>**

**val** nums **=** y.split(" ").map(x **=>** x.toDouble)

nums.sum / nums.size.toDouble

}

result.saveAsTextFile(outputFile)

Loading JSON in Scala

**import** **com.fasterxml.jackson.module.scala.DefaultScalaModule**

**import** **com.fasterxml.jackson.module.scala.experimental.ScalaObjectMapper**

**import** **com.fasterxml.jackson.databind.ObjectMapper**

**import** **com.fasterxml.jackson.databind.DeserializationFeature**

...

**case** **class** **Person**(name**:** **String**, lovesPandas**:** **Boolean**) *// Must be a top-level class*

...

*// Parse it into a specific case class. We use flatMap to handle errors*

*// by returning an empty list (None) if we encounter an issue and a*

*// list with one element if everything is ok (Some(\_)).*

**val** result **=** input.flatMap(record **=>** {

**try** {

**Some**(mapper.readValue(record, classOf[**Person**]))

} **catch** {

**case** e**:** **Exception** => **None**

}})

Saving JSON in Scala

result.filter(p **=>** P.lovesPandas).map(mapper.writeValueAsString(**\_**))

.saveAsTextFile(outputFile)

Loading CSV with textFile() in Scala

**import** **Java.io.StringReader**

**import** **au.com.bytecode.opencsv.CSVReader**

...

**val** input **=** sc.textFile(inputFile)

**val** result **=** input.map{ line **=>**

**val** reader **=** **new** **CSVReader**(**new** **StringReader**(line));

reader.readNext();

}

Writing CSV in Scala

pandaLovers.map(person **=>** **List**(person.name, person.favoriteAnimal).toArray)

.mapPartitions{people **=>**

**val** stringWriter **=** **new** **StringWriter**();

**File Formats | 79**

**val** csvWriter **=** **new** **CSVWriter**(stringWriter);

csvWriter.writeAll(people.toList)

**Iterator**(stringWriter.toString)

}.saveAsTextFile(outFile)

Loading a SequenceFile in Scala

**val** data **=** sc.sequenceFile(inFile, classOf[**Text**], classOf[**IntWritable**]).

map{**case** (x, y) **=>** (x.toString, y.get())}

On the SparkContext we

can call sequenceFile(path, keyClass, valueClass, minPartitions)

Saving a SequenceFile in Scala

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

data.saveAsSequenceFile(outputFile)

Creating a HiveContext and selecting data in Scala

**import** **org.apache.spark.sql.hive.HiveContext**

**val** hiveCtx **=** **new** org.apache.spark.sql.hive.**HiveContext**(sc)

**val** rows **=** hiveCtx.sql("SELECT name, age FROM users")

**val** firstRow **=** rows.first()

println(firstRow.getString(0)) *// Field 0 is the name*

JSON loading with Spark SQL in Scala

**val** tweets **=** hiveCtx.jsonFile("tweets.json")

tweets.registerTempTable("tweets")

**val** results **=** hiveCtx.sql("SELECT user.name, text FROM tweets")

JdbcRDD in Scala

**def** createConnection() **=** {

**Class**.forName("com.mysql.jdbc.Driver").newInstance();

**DriverManager**.getConnection("jdbc:mysql://localhost/test?user=holden");

}

**def** extractValues(r**:** **ResultSet**) **=** {

(r.getInt(1), r.getString(2))

}

**val** data **=** **new** **JdbcRDD**(sc,

createConnection, "SELECT \* FROM panda WHERE ? <= id AND id <= ?",

lowerBound **=** 1, upperBound **=** 3, numPartitions **=** 2, mapRow **=** extractValues)

println(data.collect().toList)

Loading the entire table as an RDD with key/value data in Scala

*// Implicits that add functions to the SparkContext & RDDs.*

**import** **com.datastax.spark.connector.\_**

*// Read entire table as an RDD. Assumes your table test was created as*

*// CREATE TABLE test.kv(key text PRIMARY KEY, value int);*

**val** data **=** sc.cassandraTable("test" , "kv")

*// Print some basic stats on the value field.*

data.map(row **=>** row.getInt("value")).stats()

Saving to Cassandra in Scala

**val** rdd **=** sc.parallelize(**List**(**Seq**("moremagic", 1)))

rdd.saveToCassandra("test" , "kv", **SomeColumns**("key", "value"))

Scala example of reading from HBase

**import** **org.apache.hadoop.hbase.HBaseConfiguration**

**import** **org.apache.hadoop.hbase.client.Result**

**import** **org.apache.hadoop.hbase.io.ImmutableBytesWritable**

**import** **org.apache.hadoop.hbase.mapreduce.TableInputFormat**

**val** conf **=** **HBaseConfiguration**.create()

conf.set(**TableInputFormat**.**INPUT\_TABLE**, "tablename") *// which table to scan*

**val** rdd **=** sc.newAPIHadoopRDD(

conf, classOf[**TableInputFormat**], classOf[**ImmutableBytesWritable**], classOf[**Result**])

Accumulator empty line count in Scala

**val** sc **=** **new** **SparkContext**(...)

**val** file **=** sc.textFile("file.txt")

**val** blankLines **=** sc.accumulator(0) *// Create an Accumulator[Int] initialized to 0*

**val** callSigns **=** file.flatMap(line **=>** {

**if** (line == "") {

blankLines += 1 *// Add to the accumulator*

}

line.split(" ")

})

callSigns.saveAsTextFile("output.txt")

println("Blank lines: " + blankLines.value)

Country lookup with Broadcast values in Scala

*// Look up the countries for each call sign for the*

*// contactCounts RDD. We load an array of call sign*

*// prefixes to country code to support this lookup.*

**val** signPrefixes **=** sc.broadcast(loadCallSignTable())

**val** countryContactCounts **=** contactCounts.map{**case** (sign, count) **=>**

**val** country **=** lookupInArray(sign, signPrefixes.value)

(country, count)

}.reduceByKey((x, y) **=>** x + y)

countryContactCounts.saveAsTextFile(outputDir + "/countries.txt")

Removing outliers in Scala

*// Now we can go ahead and remove outliers since those may have misreported locations*

*// first we need to take our RDD of strings and turn it into doubles.*

**val** distanceDouble **=** distance.map(string **=>** string.toDouble)

**val** stats **=** distanceDoubles.stats()

**val** stddev **=** stats.stdev

**val** mean **=** stats.mean

**val** reasonableDistances **=** distanceDoubles.filter(x **=>** math.abs(x-mean) < 3 \* stddev)

println(reasonableDistance.collect().toList

Submitting an application with extra arguments

bin/spark-submit --master spark://host:7077 --executor-memory 10g my\_script.py

General format for spark-submit

bin/spark-submit [options] <app jar | python file> [app options]

able 7-2. Common flags for spark-submit

**Flag Explanation**

--master Indicates the cluster manager to connect to. The options for this flag are described in Table 7-1.

--deploy-mode Whether to launch the driver program locally (“client”) or on one of the worker machines inside the

cluster (“cluster”). In client mode spark-submit will run your driver on the same machine where

spark-submit is itself being invoked. In cluster mode, the driver will be shipped to execute on a

worker node in the cluster. The default is client mode.

--class The “main” class of your application if you’re running a Java or Scala program.

--name A human-readable name for your application. This will be displayed in Spark’s web UI.

--jars A list of JAR files to upload and place on the classpath of your application. If your application depends

on a small number of third-party JARs, you can add them here.

--files A list of files to be placed in the working directory of your application. This can be used for data files

that you want to distribute to each node.

--py-files A list of files to be added to the PYTHONPATH of your application. This can contain .py, .egg, or .zip

*ubmitting a Python application in YARN client mode*

$ export HADOP\_CONF\_DIR=/opt/hadoop/conf

$ ./bin/spark-submit **\**

--master yarn **\**

--py-files somelib-1.2.egg,otherlib-4.4.zip,other-file.py **\**

--deploy-mode client **\**

--name "Example Program" **\**

--queue exampleQueue **\**

--num-executors 40 **\**

--executor-memory 10g **\**

my\_script.py "options" "to your application" "go here"

Creating an application using a SparkConf in Scala

*// Construct a conf*

**val** conf **=** **new** **SparkConf**()

conf.set("spark.app.name", "My Spark App")

conf.set("spark.master", "local[4]")

conf.set("spark.ui.port", "36000") *// Override the default port*

*// Create a SparkContext with this configuration*

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

Setting configuration values at runtime using flags

$ bin/spark-submit **\**

--class com.example.MyApp **\**

--master local[4] **\**

--name "My Spark App" **\**

--conf spark.ui.port=36000 **\**

myApp.jar

Example 9-6. Constructing a SQL context in Scala

**val** sc **=** **new** **SparkContext**(...)

**val** hiveCtx **=** **new** **HiveContext**(sc)

Loading and quering tweets in Scala

**val** input **=** hiveCtx.jsonFile(inputFile)

*// Register the input schema RDD*

input.registerTempTable("tweets")

*// Select tweets based on the retweetCount*

**val** topTweets **=** hiveCtx.sql("SELECT text, retweetCount FROM

tweets ORDER BY retweetCount LIMIT 10")

Accessing the text column (also first column) in the topTweets

SchemaRDD in Scala

**val** topTweetText **=** topTweets.map(row **=>** row.getString(0))

Hive load in Scala

**import** **org.apache.spark.sql.hive.HiveContext**

**val** hiveCtx **=** **new** **HiveContext**(sc)

**val** rows **=** hiveCtx.sql("SELECT key, value FROM mytable"**val** keys **=** rows.map(row **=>** row.getInt(0))

Parquet load in Python

*# Load some data in from a Parquet file with field's name and favouriteAnimal*

rows = hiveCtx.parquetFile(parquetFile)

names = rows.map(**lambda** row: row.name)

**print** "Everyone"

**print** names.collect()

Parquet query in Python

*# Find the panda lovers*

tbl = rows.registerTempTable("people")

pandaFriends = hiveCtx.sql("SELECT name FROM people WHERE favouriteAnimal = **\"**panda**\"**")

**print** "Panda friends"

**print** pandaFriends.map(**lambda** row: row.name).collect()

Creating a SchemaRDD from case class in Scala

**case** **class** **HappyPerson**(handle**:** **String**, favouriteBeverage**:** **String**)

...

*// Create a person and turn it into a Schema RDD*

**val** happyPeopleRDD **=** sc.parallelize(**List**(**HappyPerson**("holden", "coffee")))

*// Note: there is an implicit conversion*

*// that is equivalent to sqlCtx.createSchemaRDD(happyPeopleRDD)*

happyPeopleRDD.registerTempTable("happy\_people")

Scala string length UDF

registerFunction("strLenScala", (**\_:** **String**).length)

**val** tweetLength **=** hiveCtx.sql("SELECT strLenScala('tweet') FROM tweets LIMIT 10")

streaming