public class LineCountWithFiltering {

public static void main(String[] args) {

SparkConf sparkConf = new SparkConf().setAppName("File Copy");

JavaSparkContext sparkContext = new JavaSparkContext(sparkConf);

// Read the source file

JavaRDD<String> input = sparkContext.textFile(args[0]);

// RDD is immutable, let's create a new RDD which doesn't contain empty lines

// the function needs to return true for the records to be kept

JavaRDD<String> nonEmptyLines = input.filter(new Function<String, Boolean>() {

@Override

public Boolean call(String s) throws Exception {

if(s == null || s.trim().length() < 1) {

return false;

}

return true;

}

});

long count = nonEmptyLines.count();

System.out.println(String.format("Total lines in %s is %d",args[0],count));

}

}

Let's compile the program

$mvn clean package

$~/cots/spark-1.2.0-bin-hadoop2.4/bin/spark-submit --class org.learningspark.simple.LineCountWithFiltering --master local[1] target/learningspark-1.0-SNAPSHOT.jar /Users/ashishpaliwal/open-spource/flume/trunk/CHANGELOG

Important points to note are,

* filter is a transformation operation in Spark hence it is lazily evaluated
* It is a narrow operation as it is not shuffling data from one partition to multiple partitions
* filter accepts predicate as an argument and will filter the elements from source RDD which are not satisfied by predicate function
* **Spark  filter Example Using Java 8**

package com.backtobazics.sparkexamples;

import java.util.Arrays;

import java.util.List;

import org.apache.spark.api.java.JavaRDD;

import org.apache.spark.api.java.JavaSparkContext;

import org.apache.spark.api.java.function.Function;

public class FilterExample {

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

JavaSparkContext sc = new JavaSparkContext();

*//Filter Predicate*

Function<Integer, Boolean> filterPredicate = e -> e % 2 == 0;

*// Parallelized with 2 partitions*

JavaRDD<Integer> rddX = sc.parallelize(

Arrays.asList(1, 2, 3, 4, 5, 6, 7, 8, 9, 10),

2);

*// filter operation will return List of Array in following case*

JavaRDD<Integer> rddY = rddX.filter(filterPredicate);

List<Integer> filteredList = rddY.collect();

}

}

SAVE FILE WITH ORC FORMAT::::::::::::::::::;

public class SparkOrcHiveInsert {

public static void main(String[] args) {

String tableName = "person\_orc";

String tablePath = "/apps/hive/warehouse/" + tableName;

SparkConf conf = new SparkConf().setAppName("ORC Demo").setMaster("local");

JavaSparkContext sc = new JavaSparkContext(conf);

HiveContext hiveContext = new org.apache.spark.sql.hive.HiveContext(sc.sc());

JavaRDD<Person> people = sc.textFile("hdfs://~:8020/tmp/person.txt").map(

new Function<String, Person>() {

public Person call(String line) throws Exception {

return process(line);

}

});

DataFrame schemaPeople = hiveContext.createDataFrame(people, Person.class);

schemaPeople.select("id","name", "age").save(tablePath, "org.apache.spark.sql.hive.orc", SaveMode.Append);

}

private static Person process(String line) {

String[] parts = line.split(",");

Person person = new Person();

person.setId(Integer.parseInt(parts[0].trim()));

person.setName(parts[1]);

person.setAge(Integer.parseInt(parts[2].trim()));

return person;

}

}

Hive table script

create table person\_orc (

id int,

name string,

age int

) stored as orc tblproperties ("orc.compress"="NONE");

~/spark/bin/spark-submit --master local --class com.test.spark.SparkOrcHiveInsert spark-orc-hive-1.0.jar

# Save the RDD to files

RDD's have some built in methods for saving them to disk. Once in files, many of the Hadoop databases can bulk load in data directly from files, as long as they are in a specific format.

In the following code example, we demonstrate the simple .saveAsTextFile() method. This will write the data to simple text files where the .toString() method is called on each RDD element and one element is written per line. The number of files output is equal to the the number of partitions of the RDD being saved. In this sample, the RDD is repartitioned to control the number of output files.

public class LogAnalyzerExportRDD {

// Optionally modify this based as makes sense for your dataset.

public static final int NUM\_PARTITIONS = 2;

public static void main(String[] args) throws IOException {

// Create the spark context.

SparkConf conf = new SparkConf().setAppName("Log Analyzer SQL");

JavaSparkContext sc = new JavaSparkContext(conf);

if (args.length < 2) {

System.out.println("Must specify an access logs file and an output file.");

System.exit(-1);

}

String inputFile = args[0];

String outputDirectory = args[1];

JavaRDD<ApacheAccessLog> accessLogs = sc.textFile(inputFile)

.map(ApacheAccessLog::parseFromLogLine)

.repartition(NUM\_PARTITIONS); // Optionally, change this.

accessLogs.saveAsTextFile(outputDirectory);

sc.stop();

}

}

GROUPBY:::::::::::::::::::::::::;

##### Spark groupBy Example Using Scala

*// Bazic groupBy example in scala*

scala> val x = sc.parallelize(Array("Joseph", "Jimmy", "Tina",

| "Thomas", "James", "Cory",

| "Christine", "Jackeline", "Juan"), 3)

x: org.apache.spark.rdd.RDD[String] = ParallelCollectionRDD[16] at parallelize at <console>:21

*// create group per first character*

scala> val y = x.groupBy(word => word.charAt(0))

y: org.apache.spark.rdd.RDD[(Char, Iterable[String])] = ShuffledRDD[18] at groupBy at <console>:23

scala> y.collect

res0: Array[(Char, Iterable[String])] = Array((T,CompactBuffer(Tina, Thomas)), (C,CompactBuffer(Cory,

Christine)), (J,CompactBuffer(Joseph, Jimmy, James, Jackeline, Juan)))

*// Another short syntax*

scala> val y = x.groupBy(\_.charAt(0))

y: org.apache.spark.rdd.RDD[(Char, Iterable[String])] = ShuffledRDD[3] at groupBy at <console>:23

scala> y.collect

res1: Array[(Char, Iterable[String])] = Array((T,CompactBuffer(Tina, Thomas)), (C,CompactBuffer(Cory,

Christine)), (J,CompactBuffer(Joseph, Jimmy, James, Jackeline, Juan)))

##### Spark groupBy Example Using Java 8

package com.backtobazics.sparkexamples;

import java.util.Arrays;

import org.apache.spark.api.java.JavaPairRDD;

import org.apache.spark.api.java.JavaRDD;

import org.apache.spark.api.java.JavaSparkContext;

public class GroupByExample {

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

JavaSparkContext sc = new JavaSparkContext();

*// Parallelized with 2 partitions*

JavaRDD<String> rddX = sc.parallelize(

Arrays.asList("Joseph", "Jimmy", "Tina",

"Thomas", "James", "Cory",

"Christine", "Jackeline", "Juan"), 3);

JavaPairRDD<Character, Iterable<String>> rddY = rddX.groupBy(word -> word.charAt(0));

System.out.println(rddY.collect());

}

}

*// Output:*

*// [(J,[Joseph, Jimmy, James, Jackeline, Juan]), (T,[Tina, Thomas]), (C,[Cory, Christine])]*

# Apache Spark reduceByKey Example

This function has three variants

1. ***reduceByKey(function)***
2. ***reduceByKey(function, [numPartition])***
3. ***reduceByKey(partitioner, function)***

* Variants 1 will generate hash-partitioned output with existing partitioner
* Variants 2 will generate hash-partitioned output with number of partitions given by numPartition
* Variants 3 will generate output using Partitioner object referenced by partitioner

##### Spark reduceByKey Example Using Scala

*// Bazic reduceByKey example in scala*

*// Creating PairRDD x with key value pairs*

scala> val x = sc.parallelize(Array(("a", 1), ("b", 1), ("a", 1),

| ("a", 1), ("b", 1), ("b", 1),

| ("b", 1), ("b", 1)), 3)

x: org.apache.spark.rdd.RDD[(String, Int)] = ParallelCollectionRDD[1] at parallelize at <console>:21

*// Applying reduceByKey operation on x*

scala> val y = x.reduceByKey((accum, n) => (accum + n))

y: org.apache.spark.rdd.RDD[(String, Int)] = ShuffledRDD[2] at reduceByKey at <console>:23

scala> y.collect

res0: Array[(String, Int)] = Array((a,3), (b,5))

*// Another way of applying associative function*

scala> val y = x.reduceByKey(\_ + \_)

y: org.apache.spark.rdd.RDD[(String, Int)] = ShuffledRDD[3] at reduceByKey at <console>:23

scala> y.collect

res1: Array[(String, Int)] = Array((a,3), (b,5))

*// Define associative function separately*

scala> def sumFunc(accum:Int, n:Int) = accum + n

sumFunc: (accum: Int, n: Int)Int

scala> val y = x.reduceByKey(sumFunc)

y: org.apache.spark.rdd.RDD[(String, Int)] = ShuffledRDD[4] at reduceByKey at <console>:25

scala> y.collect

res2: Array[(String, Int)] = Array((a,3), (b,5))

##### Spark reduceByKey Example Using Java 8

package com.backtobazics.sparkexamples;

import java.util.Arrays;

import org.apache.spark.api.java.JavaPairRDD;

import org.apache.spark.api.java.JavaRDD;

import org.apache.spark.api.java.JavaSparkContext;

import org.apache.spark.api.java.function.Function2;

import scala.Tuple2;

public class ReduceByKeyExample {

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

JavaSparkContext sc = new JavaSparkContext();

*//Reduce Function for sum*

Function2<Integer, Integer, Integer> reduceSumFunc = (accum, n) -> (accum + n);

*// Parallelized with 2 partitions*

JavaRDD<String> x = sc.parallelize(

Arrays.asList("a", "b", "a", "a", "b", "b", "b", "b"),

3);

*// PairRDD parallelized with 3 partitions*

*// mapToPair function will map JavaRDD to JavaPairRDD*

JavaPairRDD<String, Integer> rddX =

x.mapToPair(e -> new Tuple2<String, Integer>(e, 1));

*// New JavaPairRDD*

JavaPairRDD<String, Integer> rddY = rddX.reduceByKey(reduceSumFunc);

*//Print tuples*

for(Tuple2<String, Integer> element : rddY.collect()){

System.out.println("("+element.\_1+", "+element.\_2+")");

}

}

}

*// Output:*

*// (b, 5)*

*// (a, 3)*

# Apache Spark flatMap Example

Important points to note are,

* flatMap is a transformation operation in Spark hence it is lazily evaluated
* It is a narrow operation as it is not shuffling data from one partition to multiple partitions
* Output of flatMap is flatten
* flatMap parameter function should return array, list or sequence (any subtype of scala.TraversableOnce)

Let’s take some examples,

##### Spark flatMap Example Using Scala

scala> val x = sc.parallelize(List("spark rdd example", "sample example"), 2)

*// map operation will return Array of Arrays in following case : check type of res0*

scala> val y = x.map(x => x.split(" ")) *// split(" ") returns an array of words*

scala> y.collect

res0: Array[Array[String]] = Array(Array(spark, rdd, example), Array(sample, example))

*// flatMap operation will return Array of words in following case : Check type of res1*

scala> val y = x.flatMap(x => x.split(" "))

scala> y.collect

res1: Array[String] = Array(spark, rdd, example, sample, example)

*// rdd y can be re written with shorter syntax in scala as*

scala> val y = x.flatMap(\_.split(" "))

scala> y.collect

res2: Array[String] = Array(spark, rdd, example, sample, example)

##### Spark flatMap Example Using Java 8

*// Basic map example in Java 8*

package com.backtobazics.sparkexamples;

import java.util.Arrays;

import java.util.List;

import org.apache.spark.api.java.JavaRDD;

import org.apache.spark.api.java.JavaSparkContext;

public class FlatMapExample {

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

JavaSparkContext sc = new JavaSparkContext();

*// Parallelized with 2 partitions*

JavaRDD<String> rddX = sc.parallelize(

Arrays.asList("spark rdd example", "sample example"),

2);

*// map operation will return List of Array in following case*

JavaRDD<String[]> rddY = rddX.map(e -> e.split(" "));

List<String[]> listUsingMap = rddY.collect();

*// flatMap operation will return list of String in following case*

JavaRDD<String> rddY2 = rddX.flatMap(e -> Arrays.asList(e.split(" ")));

List<String> listUsingFlatMap = rddY2.collect();

}

}

# Apache Spark reduce Example

Important points to note are,

* reduce is an action operation in Spark hence it triggers execution of DAG and gets execute on final RDD
* It is a wide operation as it is shuffling data from multiple partitions and reduces to a single value
* It accepts a Commutative and Associative function as an argument
  + The parameter function should have two arguments of the same data type
  + The return type of the function also must be same as argument types

##### Spark reduce Example Using Scala

*// reduce numbers 1 to 10 by adding them up*

scala> val x = sc.parallelize(1 to 10, 2)

scala> val y = x.reduce((accum,n) => (accum + n))

y: Int = 55

*// shorter syntax*

scala> val y = x.reduce(\_ + \_)

y: Int = 55

*// same thing for multiplication*

scala> val y = x.reduce(\_ \* \_)

y: Int = 3628800

##### Spark reduce Example Using Java 8

package com.backtobazics.sparkexamples;

import java.util.Arrays;

import org.apache.spark.api.java.JavaRDD;

import org.apache.spark.api.java.JavaSparkContext;

import org.apache.spark.api.java.function.Function2;

public class ReduceExample {

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

JavaSparkContext sc = new JavaSparkContext();

*//Reduce Function for cumulative sum*

Function2<Integer, Integer, Integer> reduceSumFunc = (accum, n) -> (accum + n);

*//Reduce Function for cumulative multiplication*

Function2<Integer, Integer, Integer> reduceMulFunc = (accum, n) -> (accum \* n);

*// Parallelized with 2 partitions*

JavaRDD<Integer> rddX = sc.parallelize(

Arrays.asList(1, 2, 3, 4, 5, 6, 7, 8, 9, 10),

2);

*// cumulative sum*

Integer cSum = rddX.reduce(reduceSumFunc);

*// another way to write*

Integer cSumInline = rddX.reduce((accum, n) -> (accum + n));

*// cumulative multiplication*

Integer cMul = rddX.reduce(reduceMulFunc);

*// another way to write*

Integer cMulInline = rddX.reduce((accum, n) -> (accum \* n));

System.out.println("cSum: " + cSum + ", cSumInline: " + cSumInline +

"\ncMul: " + cMul + ", cMulInline: " + cMulInline);

}

}

*// Output:*

*// cSum: 55, cSumInline: 55*

*// cMul: 3628800, cMulInline: 3628800*