INTRODUCTION TO APACHE SPARK

Apache Spark 101

Apache Spark does the same basic thing as Hadoop, which is run calculations on data and store the results across a distributed file system. Spark has either moved ahead or has reached par with Hadoop in terms of projects and users. A major reason for this is Spark operates much faster than Hadoop because it processes MapReduce operations in memory. One recent study said ½ of big data consulting dollars went to Hadoop projects but that Spark had more installations. Since the software is free, it's difficult to tell.

Spark also has added Spark Streaming to give it the same ability to read streaming data as LinkedIn's Kafka and Twitter's Apache Storm.

Spark has items that Hadoop does not. For example, Spark has an interactive shell. That means you can walk through datasets and code one line at a time, fixing errors as you go, which is an easy way to do development. Spark has shells in Scala, Python, and R. But you can also code Spark programs in Java. There is just no REPL (read-eval-print-loop) command line interpreter for Java.

Spark also has a machine learning library, Spark ML. Data scientists writing algorithms using Python probably use scikit-learn. R programmers uses packages from CRAN. But those does not do what Spark ML does, which is work across a distributed architecture. Instead those work only on datasets on a local server. So they would not scale without limit as Spark ML would.

Let's get started with Apache Spark by introducing some concepts and then writing code.

(This article is part of our Hadoop Guide. Use the right-hand menu to navigate.)

RDD basics

The main Spark data structures are the RDD (resilient distributed dataset) and Data Frames.

Let's explain RDDs with an example.

Spark is written in Scala, which is a functional programming language. Programmers who are not familiar with that compact notation will find it strange and often difficult to understand. But here we will not use the functional-programming coding style in order to make the Scala code easier to understand. If you only know Python, that does not matter as the Scala syntax is simple enough. What is important are the Spark functions. Those are the same in any language.

What Scala does well is work with Java. It runs on the Java JVM and lets programmers instantiate Java objects. So you can use Java objects in your code.

First, let's open the Spark Scala shell:

```
spark-shell
```

When you do that it starts up Spark and establishes a SparkContext which is the basic object upon which all others are based. SparkContext knows the details of the Spark installation and configuration, like what port processes are running on. If you were writing stand-alone jobs instead of the command line shell you would instantiate org.apache.spark. SparkContext with code plus have to start Spark manually.

Here we use Scala to create a List and make an RDD from that. First make a List of Ints.

```
var a = List(1,2,3,4,5,6)/
```

New we make an RDD:

```
var rddA = sc.parallelize(a)
```

Note: You can leave off the Scala semicolon in most commands. But if you get stuck in the shell you will have to enter one of those to make it quit prompting for values. You do not want to hit (-control-)(-c-) as that would exit the spark-shell.

Parallelize just means to create the object across the distributed architecture so that it can be worked on in parallel. In other words, it becomes an RDD. You need to keep that in mind as items that you create in, say, alphabetical order, will get out of alphabetical order as they are worked on in parallel. This can produce some strange and unwanted results. But there are ways to handle that.

Now we run the **map** function. Map just means run some operation over every element in an iterable object. A List is definitely iterable. So we can multiply each element in the List by 2 like this:

```
var mapA = a.map(n => n*2)
```

Note: The notation (x => y) means declare a variable x and run operation y on it. You can make up any name for the variables. x just stands for some element in the list. If the element was an Array of 2 elements you would write ((a,b) => function(a,b)).

Spark echoes the results:

```
mapA: List = List(2, 4, 6, 8, 10, 12)
```

Now we can sum the items in the list using **reduce**:

```
var reduceA = mapA.reduce( (a,b) => a + b)
```

Spark answers:

```
reduceA: Int = 42
```

The whole purpose of reduce is to **return** one single value, unlike **map** which creates a new collection. The **reduce** operations work on adjacent elements in pairs. So (a,b) => a + b first adds 2 + 4 = 6 then 6 + 8 = 15 until we get to 42.

Printing RDDs

There are several ways to print items. Here is one:

```
rddA.collect.foreach(println);
1
2
3
4
5
6
```

The thing to notice here is the collect command. Spark is a distributed architecture. Collect causes Spark to reach out to all nodes in the cluster and retrieve them to the cluster where you are running the spark-shell. If you did that with a really large dataset it would overload the memory of the machine.

So, to debug working on a really large dataset, it would better print to just **take** a few using:

```
rddA.take(5).foreach(println);
```

Data frames and SQL and reading from a text file

Data Frames are the next main Spark data structure. Suppose we have this comma-delimited data that shows the fishing catch in kilos by boat and species for a fishing fleet:

```
species,vessel,kilos
mackerel,Sue,800
mackerel,Ellen,750
tuna,Mary,650
tuna,Jane,160
flounder,Sue,30
flounder,Ellen,40
```

Delete the first line and then read in the comma-delimited file like shown below.

Note: Spark version 2.0 adds the DataBricks spark-csv module to make working with CSV files, easier including those with headers. We do not use that here as we want to illustrate basic functions.

```
var fishCSV =
```

```
sc.textFile("/home/walker/Downloads/fish.csv").map(_.split(","));
```

Above we ran the map function over the collection created by textFile. Then we read each line and split it into an Array of strings using the _.split(",") function. The _ is a placeholder in Scala. We could have written map(l => l.split(",")) instead.

Here is the first element. Note that Data Frames use different commands to print out their results than RDDs.

```
fishCSV.first
```

```
res4: Array = Array(mackerel, Sue, 800)
```

Now, fishCSV has no column names, so it cannot have a SQL schema. So create a class to contain that.

```
case class Catch(species: String, vessel: String, amount: Int);
```

Then map through the collection of Arrays and pass the 3 elements species, vessel, and kilos to the Catch constructor:

```
val f = fishCSV.map(w => Catch(w(0),w(1),w(2).toInt));
```

We have:

```
f.first
```

```
res7: Catch = Catch(mackerel,Sue,800)
```

Now create a SQLContext.

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

The we use createDataFrame method to create a data frame from the RDD.

```
val catchDF = sqlContext.createDataFrame(f);
```

Now we can use show to display the results:

catchDF.show

species	vessel	amount
mackerel	Sue	800
mackerel	Ellen	750
tuna	Mary	650
tuna	Jane	160
flounder	Sue	30
flounder	Ellene	40

To query that with SQL first we have to registerTempTable:

```
catchDF.registerTempTable("catchDF")
```

Notice the quote marks around catchDF. Now we can query the columns just as if we were working with a relational database. We want to list which boats caught tuna:

```
val tunaCatch = sqlContext.sql("select vessel from catchDF where species =
'tuna'");

tunaCatch.show
```

vessel
Mary
Jane

Broadcasters and accumulators

When you first learn Spark or Pig or any other language used to work with big data, the first program you usually learn is the word count program. That iterates over a text file and then uses map and reduce to count the occurrences of each word.

But that calculation can be incorrect when you are running in cluster mode instead of local model because of the distributed nature of Spark. As we showed above, you can use collect to bring the data all back in one place to put those back into the proper order. But collect is not something you would use with a large amount of data. Instead we have **broadcasters** and **accumulators**.

A **broadcast** variable is a static variable that is broadcast to all the nodes. So each node has a read-only copy of some data that it needs to do further calculations. The main reason for doing this is efficiency as they are stored in serial format to make transferring that data faster. For example, if each node needs a copy of a price list, then calculate that once and send it out to each node instead of having each node create one of those.

You create a broadcast variable like this:

```
val broadcast =sc.broadcast(Array (1,2,3));
```

An **accumulator**, as the name implies, gathers the results of addition operations across the cluster at the main node. This works, because of the associative property of arithmetic a+b = b+a. So it does not matter in what order items are added.

The accumulator is given some initial value and the nodes in the cluster update that as they are running. So it's one way to keep track of the progress of calculations running across the cluster. Of course, that is just one use case. Also know that programmers can use accumulators on any object for which they have defined the + method.

You declare an accumulator like this:

```
val accum = sc.accumulator(0)
```

Transformations

Spark has different set operations including flatMap, union, Intersection, groupbyKey, reduceByKey, and others. Below we show **reduceByKey**.

This is similar to the regular reduce operation. Except it runs a reduce function on elements with a common key. So the elements have to be in K,V (key, value) format. Here we have the Array e where the value (c,1) is repeated 2 times. So if we sum that we expect to see (c,2), which we do:

```
var e = Array(("a",1), ("b",1), ("c",1), ("c",1));
    var er = sc.parallelize(e);
val d = er.reduceByKey((x, y) => x + y);
d.collect();

res59: Array = Array((a,1), (b,1), (c,2))
```

Save data

Spark is designed to be **resilient**.That means it will preserve data in memory even when nodes crash, which they will do when they run out of memory. So it keeps track of data and recalculates datasets as needed so as not to lose them. You can persist data permanently to storage using:

- **SaveAsTextFile**—write the data as text to local the file system or Hadoop.
- **SaveasObjectFile**—store as serialized Java objects. In other words preserve the object as a type, e.g. java.util. Arrays, but store it in a efficient byte format so that it need not be converted back to a Scala object when read back into memory. Remember than Scala and Java are pretty much the same since Scala runs on the Java JVM. (Because of that a few of the Spark commands are available in Python.)
- **SaveAsSequenceFile**—write as a Hadoop sequence file.

Persist data

Persist() or cache() will keep objects in memory available to the node so they do not have to be recomputed if any of the nodes crash of the partition is used for something else or otherwise lost.

MEMORY_ONLY—store data as Java objects in memory.

MEMORY_AND_DISK—store in memory. What does not fit save to disk.

MEMORY_ONLY_SER—store as serialized Java objects, meaning write to disk as Java objects in byte format.

So those are the basic Spark concepts to get you started. As an exercise you could rewrite the Scala code here in Python, if you prefer to use Python. And for further reading you could read about Spark Streaming and Spark ML (machine learning).