Tutorial

Introduction

Spark is a relatively new, general-purpose cluster computing software system. It has APIs in Java, Scala, and Python, allowing for a wide range of options in terms of programming languages.

A Spark application utilizes a **driver program** that runs the main function. The driver also coordinates the parallel efforts of the child programs on the cluster. Each individual node on the cluster consists of multiple **resilient distributed datasets** (RDDs) that are the foundation of Spark (which we will discuss in greater detail). Simply put, RDDs enable users to parallelize their driver program, or run operations on an external dataset (of which anything using a Hadoop InputFormat is supported). The user can also specify that an RDD persists in memory, which greatly speeds up its parallel processing (assuming the memory is large enough to accommodate the RDD). They are also fault tolerant, which is a key feature for any distributed computing system (as we have already established with MapReduce).

Spark also uses **shared variables** that can be accessed from the different nodes. Normally, local variables in a parallelized function are copied over onto the different nodes. On certain occasions, special shared variables may greatly optimize a parallel operation. These come in two flavors:

1. **Broadcast Variables:** These are variables that are cached in memory on each node that uses it. As they are read-only, there is no need to worry about data mismatchs between the different copies over a period of time. This will also ensure that new nodes will receive the same value if more nodes are required for the driver.
2. **Accumulators:** These are variables that will only ever increase in value. This makes it simple to write to from the cluster nodes, as they never need to confirm the original value before writing to it. Only the driver program is able to read the accumulated value.

Deployment to a cluster is typically accomplished by packaging the program into a JAR file (when using Java or Scala), then submitting it using the `./bin/spark-submit` script.

Setup

There are two basic ways to set up a Spark environment: pre-built or source code.

Download
To download Spark, visit http://spark.apache.org/downloads.html

Pre-Built
Select a pre-built package from the download section, and download.

Source Code
After downloading the source code, run

```
export MAVEN_OPTS="-Xmx2g -XX:MaxPermSize=512M -XX:ReservedCodeCacheSize=512m"
```

This will increase the max maven memory to allow building the source.

Then run the build by executing the code

```
mvn -DskipTests clean package
```

This will take a while to complete.

Test Environment
After the build is successful, you may test your environment by running

```
./bin/run-example SparkPi 10
```

Keep in mind
Spark build is best when using scala 2.10.X
The latest scala will cause complications and may not build.
Also, building Spark requires Maven and Java.

A Little About Spark Map, Reduce, and Filter

Spark supports Python, Java, and Scala. But we will be using Scala throughout our tutorial.

Introduction

Spark's map, reduce, and filter functions operate similarly to Scala's closure, or also to pattern matching. Essentially, these functions take an input, map a variable to the input, and apply the variable in a meaningful way. We will see many demonstrations of this throughout the tutorial.

More specifically, these functions operate on every element of a dataset, then match the individual elements to a pattern, and operate on the match. For example, if we have a dataset K where each element is a string, then a filter function may look like

```
 K.filter(_.startWith("Error"))
```

The "_" is a pattern that matches any element, which is a string in this case, and the operation is to filter only the strings that startWith "Error".

We should also describe Scala's closure (function) syntax. It looks like

```
(parameter) => predicate
```

Where predicate determines the algorithm to be performed on the inputs. This is a very powerful and expressive feature when combined with pattern matching. Scala closures are the building blocks of Spark's map, reduce, and filter functions, so we will demonstrate them throughout the tutorial.

Keep in mind that map, reduce, and filter are transformations, so they will need an action to generate a logical output.

Filter

Spark's filter function takes a closure function as its input, applies it to elements of a dataset and returns a dataset where the elements returned true.

Map

Map maps each element in the dataset to the return of the closure.

Reduce

Reduce is an action rather than a transformation. It functions similarly to normal MapReduce function where it aggregates the elements using a given closure.

Spark Shell And WordCount

Shell

To start the shell, enter

```
./bin/spark-shell
```

When you see the scala shell, you've succeeded //INSERT IMAGE HERE

Next we will load a text file

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

For this example, we'll use Spark's own readme file.

Let's filter out the lines with the word Spark in it

```
scala> val linesWithSpark = textFile.filter(line => line.contains("Spark"))
```

Here we can see that an element in a text file is a line. And the filter matches each line in the text to the identifier "line", and runs the boolean function contains to return only the lines containing the word Spark.

We can also cache our data for example

```
scala> linesWithSpark.cache()
```

And then run a count action to see how many lines contained the word "Spark"

```
scala> linesWithSpark.count()
```

You can even chain functions together

```
scala> val linesWithSpark = textFile.filter(line => line.contains("Spark"))
```

Now that we are familiar with the shell, let's look at how we can count the number of occurances of each word in a file.

WordCounts

Take the text file like before

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

And now perform wordcount

```
scala> val wordCounts = textFile.flatMap(line => line.split(" ")).map(word => (word, 1)).reduceByKey((a, b) => a + b)
```

FlatMap is the same as map except each key is mapped to zero or more output items. In this case, each line is mapped to a list of words split by space.

Map will take each line as a dataset, match each word to the identifier word. Then map each word to a value 1.

And finally, reduceByKey will take a pair of words that match, since the words are the keys, and aggregate their value. In more details, the reduceByKey function will find elements with matching keys, in this case matching words, and match their values to the pattern of a pair (a, b). Which means that it will take only 2 matching words, and then add these values, aggregating to the same key(word). ReduceByKey is a transformation. And in this case, it will return the a dataset of key(word) paird with the value (sum).

And because reduceByKey is a transformation, you need to get the values of the dataset with collect

```
scala> wordCounts.collect()
```

App Deployment and PiEstimator

App Deployment

Now that we know how to use Spark with the shell, it's useful to look at a way to deploy a Spark application which can be reused. We will use Spark's "SimpleApp" to demonstrate.

```
/* SimpleApp.scala */
import org.apache.spark.SparkContext
import org.apache.spark.SparkContext._
import org.apache.spark.SparkConf

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

Diving into the SimpleApp.scala code, the conf and sc variables are the configuration and the spark context, respectively. The Spark context is the object to use to perform Spark functions.

Next, we see two filter functions, to count the number of As and the number of Bs. To do this, the dataset is the file with lines as the elements. For each element, filter will match the entire element(line) to the identifier(line). Then, a boolean function "contains()" is applied to return only lines with As or Bs. This new dataset is then applied to the "count()" action, which counts the number of elements in the set, which is the number of lines containing As and Bs respectively.

SparkConf is a configuration requirement to make sure that the application will be using Spark. To create the sbt configuration file, simple.sbt, use the following:

```
name := "Simple Project"

version := "1.0"

scalaVersion := "2.10.4"

libraryDependencies += "org.apache.spark" %% "spark-core" % "1.2.1"
```

Finally, the following steps will deploy and run the application.

```
# Your directory layout should look like this
$ find .
.
./simple.sbt
./src
./src/main
./src/main/scala
./src/main/scala/SimpleApp.scala
```

```
./src/main/scala/SimpleApp.scala

# Package a jar containing your application
$ sbt package
...
[info] Packaging {..}/{..}/target/scala-2.10/simple-project_2.10-1.0.jar

# Use spark-submit to run your application
$ YOUR_SPARK_HOME/bin/spark-submit \
  --class "SimpleApp" \
  --master local[4] \
  target/scala-2.10/simple-project_2.10-1.0.jar
...
Lines with a: 46, Lines with b: 23
```

Pi Estimator

Spark comes with a sample pi estimator to demonstrate and to test your Spark environment.

This Pi Estimator algorithm intuitively imagines a unit circle in a unit square from 0 to 1, [(0,0) ... (1,1)]. It randomly picks points in the unit square. The probability of landing a point in the circle is pi/4. So it picks random x and y coordinates, if the point is within the circle, then return 1. We will examine the mapreduce code after the code.

```
package org.apache.spark.examples
import scala.math.random
import org.apache.spark._

object SparkPi {
def main(args: Array[String]) {
val conf = new SparkConf().setAppName("Spark Pi")
val spark = new SparkContext(conf)
val slices = if (args.length > 0) args(0).toInt else 2
val n = math.min(100000L * slices, Int.MaxValue).toInt // avoid overflow
val count = spark.parallelize(1 until n, slices).map { i =>
val x = random * 2 - 1
val y = random * 2 - 1
if (x*x + y*y < 1) 1 else 0
}.reduce(_ + _)
println("Pi is roughly " + 4.0 * count / n)
spark.stop()
}
}
```

How MapReduce Worked
Going through the code line by line, we see the slices variable which controls how many points we will generate to estimate Pi. The n variable uses slice for exactly this purpose.

Next, we see the *parallelize* function, which distributes the dataset found in its first argument, which is a sequence from 1 to n, and distribute it to slices number of instances for cluster computation.

Now that we have a sequence of numbers as the dataset, each element is a number. The *map* function maps each number to the identifier i as the key. And for each i, it generates a point with x and y, then compute whether or not it falls in the unit circle. If so, it gives the key a value 1, else 0.

And finally, the *reduce* function aggregates every key/value pair, which currently is the iteration number as the key and whether or not it fell within the unit circle as the value. The pattern _+_ takes two key/value pair, and matches each value to a _. Then the addition operation is performed, and the aggregate is returned by the reduce action as a value.

From this example, you can see that Spark supports standard library imports. This is also true for Java. For this Pi Estimator, the math library has been imported.

You can get a list of actions and transformations in the API or find the most frequently used actions and transformations in the programming guide here.

Resilient Distributed Datasets (RDDs)

As mentioned before, Spark is implemented on top of RDDs. These RDDs are key to understanding how Spark is able to claim more efficient memory management while still performing as well as the other big distributed systems. The basic idea is that RDDs keep track of coarse-grained transformations that have been applied to it, rather than fine-grained operations to its shared state. That is to say, a fine-grained operation would involve updating specific cells in a shared table, which would involve propagating this change across all other nodes that use it so that it can be recovered in the event of failure. A coarse-grained approach involves using one operation to update most, if not all, elements. This allows the driver to simply keep track of all transformations on an RDD over time, allowing an RDD to easily rebuild from failure by utilizing other RDDs. This fault recovery process, involving the RDD's lineage, will be discussed in greater detail shortly. In spite of this potential loss of generality, RDDs are powerful and flexible enough to handle the majority of parallelizable computations.

Formally speaking, an RDD is actually immutable - any modifications to an RDD are actually a result of a transformation creating a new RDD after running a function through a group of elements in the old one. In fact, RDDs may not even need to stay in memory. It keeps track of how it was derived

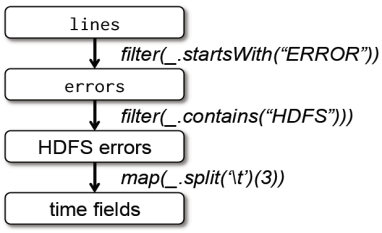
from other data through the aforementioned transformations, known as its **lineage**, and is able to compute specific partitions of itself on the fly. The user can also specify which RDDs may be queried more often, and therefore should be stored in memory instead of being represented as a series of transformations. This is known as **persistence**, which greatly speeds up execution if a user knows beforehand that a certain dataset will be referenced multiple times. An RDD can also be **partitioned** across multiple machines, which is another aspect of RDDs that can be set by the user.

RDD Operations

There are two kinds of operations one can perform on RDDs. The first is a **transformation**, which modifies an existing dataset, and the second is an **action**, which will compute a value from a dataset and return it to the driver program. An action will first load up the relevant partitions of the RDD into memory before processing it - this means that RDDs are not normally stored in memory until needed, which greatly reduces the amount of RAM needed to process large datasets efficiently. The classic transformation and action example would be `map` and `reduce`: `map` takes a function and applies it to every element in a dataset, which is how a transformation works. `reduce` takes a dataset and aggregates it through a function - this is analogous to how an action operates and passes its value on to the driver program.

Transformations in Spark are **lazy**, which means that the dataset is not modified by the transformations immediately. The transformation sequence is saved and only applied when an action requests the value for the driver program. Additionally, in the event of failure, an RDD is able to recover lost data by checking its transformation history, efficiently restoring itself without relying on a previous checkpoint and rolling back any progress made in the process.

RDD Lineage and Operations Example



To get a better sense of how lineage works, consider this example. A user needs to go through 1 TB of log files to search for instances of a specific error. He can define an RDD from this data that is on disk, making sure not to load up 1 TB of memory onto RAM. This leads to the first RDD, `lines`. From there, the user creates a new RDD `errors` by executing a transformation, as shown by the first transformation in this lineage. This filters out all of the log messages that start with the string "ERROR". Note that because the transformations are lazy, the clusters will not actually go through and do any work when the transformation has been requested. The `errors` RDD's lineage simply states that it was derived from the base RDD `lines` in the manner listed, and will compute the necessary sections when requested by an action. In this case, because the user would like to quickly query `errors` and any RDDs that may be derived from `errors`, the user can run a `.count()` action on the RDD. This is an action that will go through all of the elements in `errors`, loading everything onto memory. From there, the user goes on to craft a second filter transformation, looking for all of the error messages that involve the HDFS in some way.

Looking at this graph, we can see how an RDD is able to efficiently recover from failures, as it does not need to periodically store multiple copies of itself over the different nodes. Instead, it simply refers back to its base RDD, then follows its lineage of transformations until it reaches the state that it was in before the failure.

RDD Advantages

Aspect	RDDs	Distr. Shared Mem.
Reads	Coarse- or fine-grained	Fine-grained
Writes	Coarse-grained	Fine-grained
Consistency	Trivial (immutable)	Up to app / runtime
Fault recovery	Fine-grained and low-overhead using lineage	Requires checkpoints and program rollback
Straggler mitigation	Possible using backup tasks	Difficult
Work placement	Automatic based on data locality	Up to app (runtimes aim for transparency)
Behavior if not enough RAM	Similar to existing data flow systems	Poor performance (swapping?)

Table 1: Comparison of RDDs with distributed shared memory.

Table 1 lists out a quick summary of how an RDD-based system stacks up with a generic Distributed Shared Memory (DSM) system. Most importantly, it shows how the fault tolerance of RDDs can be so much more efficient than that of a DSM by using lineage to track its RDDs. This prevents the necessity of periodic checkpoints and required full-system rollbacks when critical nodes fail, which are typical fault recovery methods for DSMs. In addition, RDDs are capable of running extra copies of slow tasks, similar to how MapReduce deals with stragglers. In DSM, two copies of the same task would be accessing the same memory locations, creating contention and potential data inconsistencies across the board.

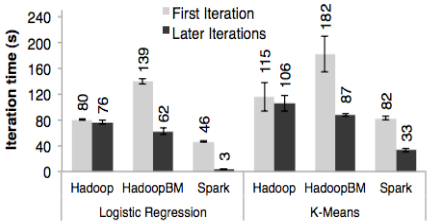
Though RDDs can be generalized to address a wide range of parallelized problems, it is also because most distributed systems today follow a MapReduce-type of framework, in which one function is applied over a large batch of elements. In the event that fine grained updates are required, RDDs would no longer be a practical choice. An example of such includes a storage system over multiple users for a web app, in which it would not be practical to assume that every user would enjoy having the contents of their storage modified to match everyone else's.

How Spark Stacks Up

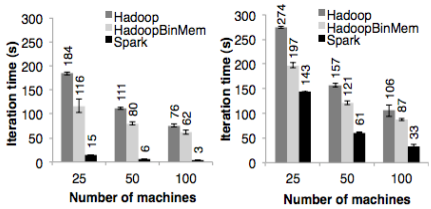
RDDs were first proposed in a paper published by UC Berkeley. There, they describe how they implements these resilient datasets (which have now become an integral part of Spark itself). The researchers were also able to show that Spark can run up to 20 times faster than Hadoop for certain iterative applications, and can scan a 1 TB dataset wth only 5 - 7 seconds of latency. Fault recovery was noticeably efficient, as Spark is able to recover just by building the failed partitions. Specific examples are as follow:

Iterative Machine Learning Algorithm Evaluation

The authors compared their RDDs in Spark against the standard Hadoop and a Hadoop deployment known as HadoopBinMem (which converts text data into binary after the initial iteration). They implemented two such machine learning algorithms, one for a logistic regression and the other for k-means. Dataset sizes were 100 GB on clusters of 25 - 100 machines. While the k-means algorithm involves a lot of computation, the regression is more sensitive to I/O and other kinds of overhead. These are qualities that will be taken into consideration when evaluating the results.



This graph depicts the time difference between the initial iteration of each system and the average of subsequent iterations thereafter. We see that Spark consistently outperformed both versions of Hadoop in both algorithms. In the case of HadoopBM, it incurred a significant overhead cost just to convert the text input into binary format, but subsequent iterations show it outperforming Hadoop on average. However, neither Hadoop versions were able to keep up with Spark once the initial overhead of reading in the data was completed.



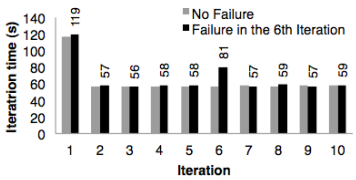
(a) Logistic Regression (b) K-Means

This graph only shows the average running time of the subsequent iterations, but compares the performance of each one over the different numbers of machines used. Again, we see that Spark greatly outperforms Hadoop and HadoopBM by a factor of over 20x on the logistic regression. The margin is not as great on the k-means algorithm, which is more influenced by computations, but it is still significant.

The researchers investigated several factors that may have contributed to Hadoop being so much slower than Spark. They looked at possible overhead from the Hadoop software stack, overhead of HDFS when serving data, and the deserialization overhead cost to convert binary into in-memory Java objects (in the case of HadoopBM). By running simple no-op commands on Hadoop, they found that these jobs still incurred about 25s of overhead involving setup, starting up tasks, and cleanup. They also found that HDFS would run checksums and copy into memory multiple times when serving data. For the final point, they compared reading either text or binary data from in-memory HDFS, then compared the results to reading text or binary data from an in-memory

file. The results showed that parsing binary input was about 7 seconds faster than parsing text input. However, both of the results still ran slower than an RDD that was directly stored in memory as a Java object, which is how Spark is able to avoid these overhead costs and outperform Hadoop for these algorithms that only require coarse-grained transformations.

Fault Recovery Evaluation



To test how Spark performed in the event of failure, the authors introduced a failure in the 6th iteration when running the k-means algorithm. The graph shows that while Spark did have to compensate for the sudden failure, it was able to recover within a meaningful amount of time and continue the successive iterations with no problem. This is because the other machines were able to reconstruct the downed RDD through its lineage, and did not have to resort to rolling back to a previous state and restarting several iterations (as would typically be the case for a checkpoint-based fault tolerance mechanism, depending on how often checkpoints were made).

The paper and all the relevant figures used in this tutorial can be found here. Further examples and implementation details are also found in that paper.