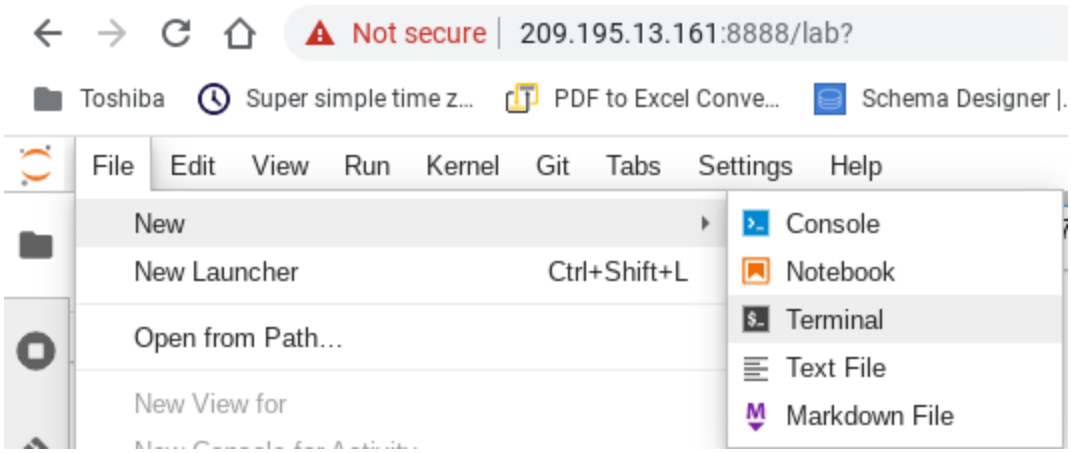
Spark Labs

LAB 1

In Jupyter Labs, launch a new terminal session:



Use wget to download baby names data from the state of New York. It will download as rows.csv but change the name to baby_names.csv:
wget https://health.data.ny.gov/api/views/jxy9-yhdk/rows.csv

```
jovyan@047f202a5007: ~   jovyan@047f202a5007: ~

jovyan@047f202a5007:~$ wget https://health.data.ny.gov/api/views/jxy9-yhdk/rows.csv
--2020-02-21 04:33:48--  https://health.data.ny.gov/api/views/jxy9-yhdk/rows.csv
Resolving health.data.ny.gov (health.data.ny.gov)... 52.206.140.199, 52.206.140.205, 52.206.68.26
Connecting to health.data.ny.gov (health.data.ny.gov)|52.206.140.199|:443... connected.
HTTP request sent, awaiting response... 200 OK
Length: unspecified [text/csv]
Saving to: 'rows.csv'

rows.csv                              [    <=>                                                  ]   1.62M  2.06MB/s    in 0.8s

2020-02-21 04:33:49 (2.06 MB/s) - 'rows.csv' saved [1704359]

jovyan@047f202a5007:~$ ls r*.csv
rows.csv
jovyan@047f202a5007:~$ mv rows.csv baby_names.csv
jovyan@047f202a5007:~$ ls b*.csv
baby_names.csv
jovyan@047f202a5007:~$
```

What is Apache Spark?

Becoming productive with Apache Spark requires an understanding of a few fundamental elements. In this lab, let's explore the fundamentals or the building blocks of Apache Spark. Let's use descriptions and real-world examples in the exploration.

The intention is for you to understand basic Spark concepts. It assumes you are familiar with installing software and unzipping files. Later labs will deeper dive into Apache Spark and example use cases.

The Spark API can be called via Scala, Python or Java. This lab will use Scala, but can be easily translated to other languages and operating systems.

If you have any questions or comments, please leave a comment at the bottom of this lab.

I. Spark Overview

Let's dive into code and working examples first. Then, we'll describe Spark concepts that tie back to the source code examples.

Requirements

* Recent version of Java installed

* Download NY State Baby Names in CSV format from: http://www.healthdata.gov/dataset/baby-names-beginning-2007. (I renamed the csv file to baby_names.csv)

I. Spark with Scala Code examples

The CSV file we will use has following structure:

Year,First Name,County,Sex,Count

2012,DOMINIC,CAYUGA,M,6

2012,ADDISON,ONONDAGA,F,14

2012,JULIA,ONONDAGA,F,15

Let's start our Spark and Scala journey with a working example.

Let's run some code
Launch the spark shell in the terminal: spark-shell

```
jovyan@047f202a5007:~$ spark-shell
20/02/21 04:39:13 WARN NativeCodeLoader: Unable to load native-hadoop library for your platform... u
able
Using Spark's default log4j profile: org/apache/spark/log4j-defaults.properties
Setting default log level to "WARN".
To adjust logging level use sc.setLogLevel(newLevel). For SparkR, use setLogLevel(newLevel).
20/02/21 04:39:34 WARN Utils: Service 'SparkUI' could not bind on port 4040. Attempting port 4041.
Spark context Web UI available at http://047f202a5007:4041
Spark context available as 'sc' (master = local[*], app id = local-1582259975083).
Spark session available as 'spark'.
Welcome to
      ____              __
     / __/__  ___ _____/ /__
    _\ \/ _ \/ _ `/ __/  '_/
   /___/ .__/\_,_/_/ /_/\_\   version 2.4.4
      /_/

Using Scala version 2.11.12 (OpenJDK 64-Bit Server VM, Java 1.8.0_232)
Type in expressions to have them evaluated.
Type :help for more information.

scala>
```

Now execute the code listed in the rest of the lab:

```
scala> val babyNames = sc.textFile("baby_names.csv")
babyNames: org.apache.spark.rdd.RDD[String] = baby_names.csv MappedRDD[1] at textFile
at <console>:12

scala> babyNames.count
res0: Long = 35218

scala> babyNames.first()
res177: String = Year,First Name,County,Sex,Count
```

So, we know there are 35218 rows in the CSV

```
scala> val rows = babyNames.map(line => line.split(","))
rows: org.apache.spark.rdd.RDD[Array[String]] = MappedRDD[17] at map at <console>:14

scala> rows.map(row => row(2)).distinct.count
res22: Long = 62
```

Above, we converted each row in the CSV file to an Array of Strings. Then, we determined there are 62 unique NY State counties over the years of data collected in the CSV.

```
scala> val davidRows = rows.filter(row => row(1).contains("DAVID"))
davidRows: org.apache.spark.rdd.RDD[Array[String]] = FilteredRDD[24] at filter at
<console>:16

scala> davidRows.count
res32: Long = 136
```

There are 136 rows containing the name "DAVID".

```
scala> davidRows.filter(row => row(4).toInt > 10).count()
res41: Long = 89
```

Number of rows where "NAME" DAVID has a "Count" greater than 10

```
scala> davidRows.filter(row => row(4).toInt > 10).map( r => r(2) ).distinct.count
res57: Long = 17
```

17 unique counties which have had the name DAVID over 10 times in a given year

```
scala> val names = rows.map(name => (name(1),1))
names: org.apache.spark.rdd.RDD[(String, Int)] = MappedRDD[342] at map at <console>:16

scala> names.reduceByKey((a,b) => a + b).sortBy(_._2).foreach ( println _) // shows
number lines each name appears in file, because how names was created with
rows.map(name => (name(1), 1)); Jacob appears most often, but is actually not the most
popular by total count.

scala> val filteredRows = babyNames.filter(line => !line.contains("Count")).map(line
=> line.split(","))
filteredRows: org.apache.spark.rdd.RDD[Array[String]] = MappedRDD[546] at map at
<console>:14

scala> filteredRows.map ( n => (n(1), n(4).toInt)).reduceByKey((a,b) => a +
b).sortBy(_._2).foreach (println _)
```

The output from the last foreach loop hasn't been shown, but Michael (9187 times) followed by Matthew (7891) and Jaden (7807) have been the top 3 most popular name in

NY from years 2007 through 2012. Also, the first row of the CSV needed to be discarded in order to avoid NumberFormatException

If your results do not show Michael 9187 times, try re-running the last command numerous times. The `println _` function being called from foreach will only print out one partition of the RDD and there is always a minimum of 2 partitions in a Spark RDD. To ensure the entire RDD is printed, send to collect first:

```
filteredRows.map ( n => (n(1), n(4).toInt)).reduceByKey((a,b) => a +
b).sortBy(_._2).collect.foreach (println _)
```

I. Spark Core Overview

The remainder of this lab will cover Spark Core concepts. Spark Core is what makes all other aspects of the Spark ecosystem possible. Later labs will cover other aspects of the Spark ecosystem including Spark SQL, Spark Streaming, MLib and GraphX.

II. Spark Context and Resilient Distributed Datasets

The way to interact with Spark is via a SparkContext (or Spark Session in new versions of Spark). The example used the Spark Console which provides a SparkContext automatically. Did you notice the last line in the REPL?

06:32:25 INFO SparkILoop: Created spark context..
Spark context available as sc.

That's how we're able to use sc from within our example.

After obtaining a SparkContext, developers may interact with Spark via Resilient Distributed Datasets or RDDs. (Update: this changes from RDDs to DataFrames and DataSets in later versions of Spark.)

Resilient Distributed Datasets (RDDs) are an immutable, distributed collection of elements. These collections may be parallelized across a cluster. RDDs are loaded from an external data set or created via a SparkContext. We'll cover both of these scenarios.

In the previous example, we create a RDD via:

```
scala> val babyNames = sc.textFile("baby_names.csv")
```

We also created RDDs other ways as well, which we'll cover a bit later.

When utilizing Spark, you will be doing one of two primary interactions: creating new RDDs or transforming existing RDDs to compute a result. The next section describes these two Spark interactions.

III Spark Actions and Transformations

When working with Spark RDDs, there are two available operations: actions or transformations. An action is an execution which *produces a result*. Examples of actions in previous are countand first.

EXAMPLE OF SPARK ACTIONS

```
babyNames.count() // number of lines in the CSV file
babyNames.first() // first line of CSV
```

EXAMPLE OF SPARK TRANSFORMATIONS

Transformations create new RDDs using existing RDDs. We created a variety of RDDs in our example:

```
scala> val rows = babyNames.map(line => line.split(","))
```

```
scala> val davidRows = rows.filter(row => row(1).contains("DAVID"))
```

IV. Conclusion and Looking Ahead

In this lab, we covered the fundamentals for being productive with Apache Spark. As you witnessed, there are just a few Spark concepts to know before being able to be productive. What do you think of Scala? To many, the use of Scala is more intimidating than Spark itself. Sometimes choosing to use Spark is a way to bolster your Scala-fu as well.

Lab 2

Apache Spark Transformations in Scala Examples

Spark Transformations in Scala Examples

Spark Transformations produce a new Resilient Distributed Dataset (RDD) or DataFrame or DataSet depending on your version of Spark. Resilient distributed datasets are Spark's main and original programming abstraction for working with data distributed across multiple nodes in your cluster. RDDs are automatically parallelized across the cluster.

SCALA SPARK TRANSFORMATIONS FUNCTION EXAMPLES

map

flatMap

filter

mapPartitions

mapPartitionsWithIndex

sample

Hammer Time

union

intersection

distinct

The Keys

groupByKey

reduceByKey

aggregateByKey

sortByKey

join

map(func)

What does it do? Pass each element of the RDD through and into the supplied function; i.e. `func`

```
scala> val rows = babyNames.map(line => line.split(","))
rows: org.apache.spark.rdd.RDD[Array[String]] = MappedRDD[360] at map at
<console>:14
```

What did this example do? Iterates over every line in the babyNames RDD (originally created from baby_names.csv file) and splits into new RDD of Arrays of Strings. The arrays contain a String separated by comma characters in the source RDD (CSV). Makes sense?

flatMap(func)

"Similar to map, but each input item can be mapped to 0 or more output items (so `*func*` should return a Seq rather than a single item)." Whereas themapfunction can be thought of as a one-to-one operation, theflatMapfunction can be considered a one-to-many. Well, that helped me if I considered it that way anyhow.

Compare flatMaptomapin the following

```
scala> sc.parallelize(List(1,2,3)).flatMap(x=>List(x,x,x)).collect
res200: Array[Int] = Array(1, 1, 1, 2, 2, 2, 3, 3, 3)

scala> sc.parallelize(List(1,2,3)).map(x=>List(x,x,x)).collect
```

res201: Array[List[Int]] = Array(List(1, 1, 1), List(2, 2, 2), List(3, 3, 3))

flatMapcan be especially helpful with nested datasets. For example, it may be helpful to think of the RDD source as hierarchical JSON (which may have been converted to case classes or nested collections). This is unlike CSV which should have no hierarchical structural.

By the way, these examples may blur the line between Scala and Spark. Both Scala and Spark have bothmap and flatMapin their APIs. In a sense, the only Spark unique portion of this code example above is the use of `parallelize` from a SparkContext. When calling `parallelize`, the elements of the collection are copied to form a distributed dataset that can be operated on in parallel. Being able to operate in parallel is a Spark feature. But, you knew that already.

Adding `collect` to both the `flatMap` and `map` results was shown for clarity. We can focus on Spark aspect (re: the RDD return type) of the example if we don't use `collect` as seen in the following:

```
scala> sc.parallelize(List(1,2,3)).flatMap(x=>List(x,x,x))
res202: org.apache.spark.rdd.RDD[Int] = FlatMappedRDD[373] at flatMap at <console>:13

scala> sc.parallelize(List(1,2,3)).map(x=>List(x,x,x))
res203: org.apache.spark.rdd.RDD[List[Int]] = MappedRDD[375] at map at <console>:13
```

Formal API sans implicit: flatMap[U](f: (T) ⇒ TraversableOnce[U]): RDD[U]

filter(func)

Filter creates a new RDD by passing in the supplied funcused to filter the results. For those people with relational database background or coming from a SQL perspective, it may be helpful think of `filter` as the `where` clause in a SQL statement. In other words, where in SQL is used to filter the desired results according to some criteria such as ... where state = 'MN'

Spark filter examples

```
val file = sc.textFile("catalina.out")
```

val errors = file.filter(line => line.contains("ERROR"))

Formal API: filter(f: (T) ⇒ Boolean): RDD[T]

mapPartitions(func)

Consider mapPartitionsa tool for performance optimization. Now, it won't do much good for you when running examples on your local machine, but don't forget it when running on a Spark cluster. It's the same as map but works with Spark RDD partitions. Remember the first D in RDD is "Distributed" – Resilient Distributed Datasets. Or, put another way, you could say it is distributed over partitions.

```
// from laptop
scala> val parallel = sc.parallelize(1 to 9, 3)
parallel: org.apache.spark.rdd.RDD[Int] = ParallelCollectionRDD[450] at parallelize at
<console>:12

scala> parallel.mapPartitions( x => List(x.next).iterator).collect
res383: Array[Int] = Array(1, 4, 7)

// compare to the same, but with default parallelize
scala> val parallel = sc.parallelize(1 to 9)
parallel: org.apache.spark.rdd.RDD[Int] = ParallelCollectionRDD[452] at parallelize at
<console>:12

scala> parallel.mapPartitions( x => List(x.next).iterator).collect
res384: Array[Int] = Array(1, 2, 3, 4, 5, 6, 7, 8)
```

API: mapPartitions[U](f: (Iterator[T]) ⇒ Iterator[U], preservesPartitioning: Boolean = false)(implicit arg0:ClassTag[U]): RDD[U]

mapPartitionsWithIndex(func)

Similar to mapPartitions but also provides a function with an Int value to indicate the index position of the partition.

```
scala> val parallel = sc.parallelize(1 to 9)
```

```
parallel: org.apache.spark.rdd.RDD[Int] = ParallelCollectionRDD[455] at parallelize at
<console>:12

scala> parallel.mapPartitionsWithIndex( (index: Int, it: Iterator[Int]) => it.toList.map(x
=> index + ", "+x).iterator).collect
res389: Array[String] = Array(0, 1, 1, 2, 2, 3, 3, 4, 4, 5, 5, 6, 6, 7, 7, 8, 7, 9)
```

When learning these APIs on an individual laptop or desktop, it might be helpful to show differences in capabilities and outputs. For example, if we change the above example to use a parallelize'd list with 3 slices, our output changes significantly:

```
scala> val parallel = sc.parallelize(1 to 9, 3)
parallel: org.apache.spark.rdd.RDD[Int] = ParallelCollectionRDD[457] at parallelize at
<console>:12

scala> parallel.mapPartitionsWithIndex( (index: Int, it: Iterator[Int]) => it.toList.map(x
=> index + ", "+x).iterator).collect
res390: Array[String] = Array(0, 1, 0, 2, 0, 3, 1, 4, 1, 5, 1, 6, 2, 7, 2, 8, 2, 9)
```

Formal API signature (implicts stripped) and definition from Spark Scala API docs:

mapPartitionsWithIndex[U](f: (Int, Iterator[T]) ⇒ Iterator[U], preservesPartitioning: Boolean = false): RDD[U]

"Return a new RDD by applying a function to each partition of this RDD, while tracking the index of the original partition.

preservesPartitioning indicates whether the input function preserves the partitioner, which should be false unless this is a pair RDD and the input function doesn't modify the keys."

sample(*withReplacement*,*fraction*, *seed*)

Return a random sample subset RDD of the input RDD

```
scala> val parallel = sc.parallelize(1 to 9)
parallel: org.apache.spark.rdd.RDD[Int] = ParallelCollectionRDD[470] at parallelize at
<console>:12
```

```
scala> parallel.sample(true,.2).count
res403: Long = 3

scala> parallel.sample(true,.2).count
res404: Long = 2

scala> parallel.sample(true,.1)
res405: org.apache.spark.rdd.RDD[Int] = PartitionwiseSampledRDD[473] at sample at
<console>:15
```

Formal API: (withReplacement: Boolean, fraction: Double, seed: Long = Utils.random.nextLong): RDD[T]

The Next Three (AKA: Hammer Time)

Stop. Hammer Time. Does anyone remember that? Remember those pants! Oh man, I wish I could forget it. Google "Hammer Time" if you don't know. Anyhow, where was I?

The next three functions union, intersection and distinct really play well off of each other. "can't Touch this" . See what I did there <- Hah! I threw in some MC Hammer. Hammer Time.

union(a different rdd)

Simple. Return the union of two RDDs

```
scala> val parallel = sc.parallelize(1 to 9)
parallel: org.apache.spark.rdd.RDD[Int] = ParallelCollectionRDD[477] at parallelize at
<console>:12

scala> val par2 = sc.parallelize(5 to 15)
par2: org.apache.spark.rdd.RDD[Int] = ParallelCollectionRDD[478] at parallelize at
<console>:12

scala> parallel.union(par2).collect
res408: Array[Int] = Array(1, 2, 3, 4, 5, 6, 7, 8, 9, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15)
```

intersection(a different rdd)

Simple. Similar to union but return the intersection of two RDDs

```
scala> val parallel = sc.parallelize(1 to 9)
parallel: org.apache.spark.rdd.RDD[Int] = ParallelCollectionRDD[477] at parallelize at
<console>:12

scala> val par2 = sc.parallelize(5 to 15)
par2: org.apache.spark.rdd.RDD[Int] = ParallelCollectionRDD[478] at parallelize at
<console>:12

scala> parallel.intersection(par2).collect
res409: Array[Int] = Array(8, 9, 5, 6, 7)
```

Formal API: intersection(other: RDD[T]): RDD[T]

distinct([numTasks])

Another simple one. Return a new RDD with distinct elements within a source RDD

```
scala> val parallel = sc.parallelize(1 to 9)
parallel: org.apache.spark.rdd.RDD[Int] = ParallelCollectionRDD[477] at parallelize at
<console>:12

scala> val par2 = sc.parallelize(5 to 15)
par2: org.apache.spark.rdd.RDD[Int] = ParallelCollectionRDD[478] at parallelize at
<console>:12

scala> parallel.union(par2).distinct.collect
res412: Array[Int] = Array(1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15)
```

Formal API: distinct(): RDD[T]

The Keys

The group of Spark transformations such as groupByKey, reduceByKey, aggregateByKey, sortByKey, join all act on key, value RDDs. So, this section will be known as "The Keys". Cool name, huh? Well, not really, but it sounded much better than "The Keys and the Values" which for some unexplained reason, triggers memories of "The Young and the Restless". Let's have some fun.

Hey, it might be important for some of you to note here. You're going to work much with key, value structured data in your Spark adventures.

The following key functions are available through org.apache.spark.rdd.PairRDDFunctions which are operations available only on RDDs of key-value pairs. "These operations are automatically available on any RDD of the right type (e.g. RDD[(Int, Int)] through implicit conversions when you import org.apache.spark.SparkContext._."

For the following, we're going to use the baby_names.csv file introduced in the previous lab What is Apache Spark?

All the following examples presume the baby_names.csv file has been loaded and split such as:

```
scala> val babyNames = sc.textFile("baby_names.csv")
babyNames: org.apache.spark.rdd.RDD[String] = baby_names.csv MappedRDD[495] at
textFile at <console>:12

scala> val rows = babyNames.map(line => line.split(","))
rows: org.apache.spark.rdd.RDD[Array[String]] = MappedRDD[496] at map at
<console>:14
```

groupByKey([numTasks])

"When called on a dataset of (K, V) pairs, returns a dataset of (K, Iterable<V>) pairs. "

The following groups all names to counties in which they appear over the years.

```
scala> val namesToCounties = rows.map(name => (name(1),name(2)))
namesToCounties: org.apache.spark.rdd.RDD[(String, String)] = MappedRDD[513] at map
at <console>:16
```

scala> namesToCounties.groupByKey.collect
res429: Array[(String, Iterable[String])] = Array((BRADEN,CompactBuffer(SUFFOLK, SARATOGA, SUFFOLK, ERIE, SUFFOLK, SUFFOLK, ERIE)), (MATTEO,CompactBuffer(NEW YORK, SUFFOLK, NASSAU, KINGS, WESTCHESTER, WESTCHESTER, KINGS, SUFFOLK, NASSAU, QUEENS, QUEENS, NEW YORK, NASSAU, QUEENS, KINGS, SUFFOLK, WESTCHESTER, WESTCHESTER, SUFFOLK, KINGS, NASSAU, QUEENS, SUFFOLK, NASSAU, WESTCHESTER)), (HAZEL,CompactBuffer(ERIE, MONROE, KINGS, NEW YORK, KINGS, MONROE, NASSAU, SUFFOLK, QUEENS, KINGS, SUFFOLK, NEW YORK, KINGS, SUFFOLK)), (SKYE,CompactBuffer(NASSAU, KINGS, MONROE, BRONX, KINGS, KINGS, NASSAU)), (JOSUE,CompactBuffer(SUFFOLK, NASSAU, WESTCHESTER, BRONX, KINGS, QUEENS, SUFFOLK, QUEENS, NASSAU, WESTCHESTER, BRONX, BRONX, QUEENS, SUFFOLK, KINGS, WESTCHESTER, QUEENS, NASSAU, SUFFOLK, BRONX, KINGS, ...

The above example was created from baby_names.csv file which was introduced in previous lab What is Apache Spark?

reduceByKey(*func*, [*numTasks*])

Operates on (K,V) pairs of course, but the func must be of type (V,V) => V

Let's sum the yearly name counts over the years in the CSV. Notice we need to filter out the header row.

scala> val filteredRows = babyNames.filter(line => !line.contains("Count")).map(line => line.split(","))
filteredRows: org.apache.spark.rdd.RDD[Array[String]] = MappedRDD[546] at map at <console>:14

scala> filteredRows.map(n => (n(1),n(4).toInt)).reduceByKey((v1,v2) => v1 + v2).collect
res452: Array[(String, Int)] = Array((BRADEN,39), (MATTEO,279), (HAZEL,133), (SKYE,63), (JOSUE,404), (RORY,12), (NAHLA,16), (ASIA,6), (MEGAN,581), (HINDY,254), (ELVIN,26), (AMARA,10), (CHARLOTTE,1737), (BELLA,672), (DANTE,246), (PAUL,712), (EPHRAIM,26), (ANGIE,295), (ANNABELLA,38), (DIAMOND,16), (ALFONSO,6), (MELISSA,560), (AYANNA,11), (ANIYAH,365), (DINAH,5), (MARLEY,32), (OLIVIA,6467), (MALLORY,15), (EZEQUIEL,13), (ELAINE,116), (ESMERALDA,71), (SKYLA,172),

(EDEN,199), (MEGHAN,128), (AHRON,29), (KINLEY,5), (RUSSELL,5), (TROY,88), (MORDECHAI,521), (JALIYAH,10), (AUDREY,690), (VALERIE,584), (JAYSON,285), (SKYLER,26), (DASHIELL,24), (SHAINDEL,17), (AURORA,86), (ANGELY,5), (ANDERSON,369), (SHMUEL,315), (MARCO,370), (AUSTIN,1345), (MITCHELL,12), (SELINA,187), (FATIMA,421), (CESAR,292), (CAR...

Formal API: reduceByKey(func: (V, V) ⇒ V): RDD[(K, V)]

And for the last time, the above example was created from baby_names.csv file which was introduced in previous lab What is Apache Spark?

aggregateByKey(*zeroValue*)(*seqOp*, *combOp*, [*numTasks*])

Ok, I admit, this one drives me a bit nuts. Why wouldn't we just use reduceByKey? I don't feel smart enough to know when to use aggregateByKey over reduceByKey. For example, the same results may be produced:

```
scala> val filteredRows = babyNames.filter(line => !line.contains("Count")).map(line => line.split(","))
filteredRows: org.apache.spark.rdd.RDD[Array[String]] = MappedRDD[546] at map at <console>:14

scala> filteredRows.map(n => (n(1),n(4).toInt)).reduceByKey((v1,v2) => v1 + v2).collect
res452: Array[(String, Int)] = Array((BRADEN,39), (MATTEO,279), (HAZEL,133), (SKYE,63), (JOSUE,404), (RORY,12), (NAHLA,16), (ASIA,6), (MEGAN,581), (HINDY,254), (ELVIN,26), (AMARA,10), (CHARLOTTE,1737), (BELLA,672), (DANTE,246), (PAUL,712), (EPHRAIM,26), (ANGIE,295), (ANNABELLA,38), (DIAMOND,16), (ALFONSO,6), (MELISSA,560), (AYANNA,11), (ANIYAH,365), (DINAH,5), (MARLEY,32), (OLIVIA,6467), (MALLORY,15), (EZEQUIEL,13), (ELAINE,116), (ESMERALDA,71), (SKYLA,172), (EDEN,199), (MEGHAN,128), (AHRON,29), (KINLEY,5), (RUSSELL,5), (TROY,88), (MORDECHAI,521), (JALIYAH,10), (AUDREY,690), (VALERIE,584), (JAYSON,285), (SKYLER,26), (DASHIELL,24), (SHAINDEL,17), (AURORA,86), (ANGELY,5), (ANDERSON,369), (SHMUEL,315), (MARCO,370), (AUSTIN,1345), (MITCHELL,12), (SELINA,187), (FATIMA,421), (CESAR,292), (CAR...


scala> filteredRows.map ( n => (n(1), n(4))).aggregateByKey(0)((k,v) => v.toInt+k, (v,k) => k+v).sortBy(_._2).collect
```

There's a gist of aggregateByKey as well.

sortByKey([*ascending*], [*numTasks*])

This simply sorts the (K,V) pair by K. Try it out. See examples above on where babyNames originates.

```
scala> val filteredRows = babyNames.filter(line => !line.contains("Count")).map(line => line.split(","))
filteredRows: org.apache.spark.rdd.RDD[Array[String]] = MappedRDD[546] at map at <console>:14

scala>  filteredRows.map ( n => (n(1), n(4))).sortByKey().foreach (println _)

scala>  filteredRows.map ( n => (n(1), n(4))).sortByKey(false).foreach (println _) // opposite order
```

join(*otherDataset*, [*numTasks*])

If you have relational database experience, this will be easy. It's joining of two datasets. Other joins are available as well such as *leftOuterJoin* and *rightOuterJoin*.

```
scala> val names1 = sc.parallelize(List("abe", "abby", "apple")).map(a => (a, 1))
names1: org.apache.spark.rdd.RDD[(String, Int)] = MappedRDD[1441] at map at <console>:14

scala> val names2 = sc.parallelize(List("apple", "beatty", "beatrice")).map(a => (a, 1))
names2: org.apache.spark.rdd.RDD[(String, Int)] = MappedRDD[1443] at map at <console>:14

scala> names1.join(names2).collect
res735: Array[(String, (Int, Int))] = Array((apple,(1,1)))

scala> names1.leftOuterJoin(names2).collect
res736: Array[(String, (Int, Option[Int]))] = Array((abby,(1,None)), (apple,(1,Some(1))), (abe,(1,None)))

scala> names1.rightOuterJoin(names2).collect
```

res737: Array[(String, (Option[Int], Int))] = Array((apple,(Some(1),1)), (beatty,(None,1)), (beatrice,(None,1)))

SPARK TRANSFORMATIONS EXAMPLES IN SCALA CONCLUSION

As mentioned at the top, the way to really get a feel for your Spark API options with Spark Transformations is to perform these examples in your own environment. "hands on the keyboard" as some people refer to it. If you have any questions or suggestions, let me know. And keep an eye on the Spark with Scala labs page for the latest labs using Scala with Spark.

Featured image credit https://flic.kr/p/8R8uP9

Lab 3

Apache Spark Examples of Actions in Scala

Spark Action Examples in Scala

When using Spark API "action" functions, a **result** is produced back to the Spark Driver. Computing this result will trigger any of the RDDs, DataFrames or DataSets needed in order to produce the result. Recall Spark Transformations such as map, flatMap, and other transformations are used to create RDDs, DataFrames or DataSets are *lazily initialized*. A call to a Spark action will trigger the RDDs, DataFrames or DataSets to be formed in memory or hydrated or initialized or you may choose to say it other ways as well. "hydrated" is the fancy way to say it, but be wary of people who use the term "hydrated".

As mentioned elsewhere on this site, if you are new to Scala with Spark, you are encouraged to perform these examples in your environment. Don't just read through them. Type them out on your keyboard. Now, I don't mean to sound bossy here, but typing these examples really will help you learn. Trust me, I'm a doctor. Actually, I'm not a doctor, but it sounds better than just "Trust me" without the doctor part.

Once you are comfortable here, check out other Scala with Spark labs and let me know if you have any questions or suggestions. Thanks, from Todd... who is not a doctor by the way.

reduce

collect

count

first

take

takeSample

countByKey

saveAsTextFile

reduce(func)

Aggregate the elements of a dataset through *func*

```
scala> val names1 = sc.parallelize(List("abe", "abby", "apple"))
names1: org.apache.spark.rdd.RDD[String] = ParallelCollectionRDD[1467] at parallelize at
<console>:12

scala> names1.reduce((t1,t2) => t1 + t2)
res778: String = abbyabeapple

scala> names1.flatMap(k => List(k.size) ).reduce((t1,t2) => t1 + t2)
res779: Int = 12

// another way to show

scala> val names2 = sc.parallelize(List("apple", "beatty", "beatrice")).map(a => (a, a.size))
names2: org.apache.spark.rdd.RDD[(String, Int)] = MappedRDD[1473] at map at <console>:14

scala> names2.flatMap(t => Array(t._2)).reduce(_ + _)
res783: Int = 19
```

map API signature with stripped implicits: map[U](f: (T) ⇒ U): RDD[U]

collect(func)

collect returns the elements of the dataset as an array back to the driver program.

collect is often used in previously provided examples such as Spark Transformation Examples in order to show the values of the return. The REPL, for example, will print the values of the array back to the console. This can be helpful in debugging programs.

Examples

scala> **sc.parallelize(List(1,2,3)).flatMap(x=>List(x,x,x)).collect**
res200: Array[Int] = Array(1, 1, 1, 2, 2, 2, 3, 3, 3)

scala> **sc.parallelize(List(1,2,3)).map(x=>List(x,x,x)).collect**
res201: Array[List[Int]] = Array(List(1, 1, 1), List(2, 2, 2), List(3, 3, 3))

Formal API : collect(): Array[T]

Return an array containing all of the elements in this RDD.

count()

Number of elements in the RDD

scala> **val names2 = sc.parallelize(List("apple", "beatty", "beatrice"))**
names2: org.apache.spark.rdd.RDD[String] = ParallelCollectionRDD[1476] at parallelize at <console>:12

scala> **names2.count**
res784: Long = 3

first()

Return the first element in the RDD

scala> **val names2 = sc.parallelize(List("apple", "beatty", "beatrice"))**
names2: org.apache.spark.rdd.RDD[String] = ParallelCollectionRDD[1477] at parallelize at <console>:12

scala> **names2.first**
res785: String = apple

take(n)

From Spark Programming Guide,"Return an array with the first *n* elements of the dataset."

scala> **val names2 = sc.parallelize(List("apple", "beatty", "beatrice"))**
names2: org.apache.spark.rdd.RDD[String] = ParallelCollectionRDD[1478] at parallelize at <console>:12

scala> **names2.take(2)**
res786: Array[String] = Array(apple, beatty)

takeSample(withReplacement: Boolean, n:Int, [seed:Int]): Array[T]

Similar to take, in return type of array with size of n. Includes boolean option of with or without replacement and random generator seed.

scala> **val teams = sc.parallelize(List("twins", "brewers", "cubs", "white sox", "indians", "bad news bears"))**
teams: org.apache.spark.rdd.RDD[String] = ParallelCollectionRDD[1482] at parallelize at <console>:12

scala> **teams.takeSample(true, 3)**
res789: Array[String] = Array(white sox, twins, white sox)

scala> **teams.takeSample(true, 3)**
res790: Array[String] = Array(cubs, white sox, twins)

countByKey()

This is only available on RDDs of (K,V) and returns a hashmap of (K, count of K)

scala> **val hockeyTeams = sc.parallelize(List("wild", "blackhawks", "red wings", "wild", "oilers", "whalers", "jets", "wild"))**
hockeyTeams: org.apache.spark.rdd.RDD[String] = ParallelCollectionRDD[3] at parallelize at <console>:12

scala> **hockeyTeams.map(k => (k,1)).countByKey**
res0: scala.collection.Map[String,Long] = Map(jets -> 1, blackhawks -> 1, red wings -> 1, oilers -> 1, whalers -> 1, wild -> 3)

CountByKey would have been helpful to show in most popular baby names spark example from an earlier lab.

```
scala> val babyNamesToTotalCount = sc.textFile("baby_names.csv").map(line =>
line.split(",")).map(n => (n(1), n(4)))
babyNamesToTotalCount: org.apache.spark.rdd.RDD[(String, String)] = MappedRDD[21] at map at
<console>:12

scala> babyNamesToTotalCount.countByKey
res2: scala.collection.Map[String,Long] = Map(JADEN -> 65, KACPER -> 2, RACHEL -> 63,
JORDYN -> 33, JANA -> 1, CESIA -> 1, IBRAHIM -> 22, LUIS -> 65, DESMOND -> 5, AMANI -> 6,
ELIMELECH -> 7, LILA -> 39, NEYMAR -> 1, JOSUE -> 31, LEELA -> 1, DANNY -> 25, GARY -> 3,
SIMA -> 10, GOLDY -> 14, SADIE -> 41, MARY -> 40, LINDSAY -> 10, JAKOB -> 2, AHARON -> 5,
LEVI -> 39, MADISYN -> 3, HADASSAH -> 5, MALIA -> 10, ANTONIA -> 2, RAIZY -> 16, ISAIAS ->
1, AMINA -> 9, DECLAN -> 33, GILLIAN -> 1, ARYANA -> 1, GRIFFIN -> 25, BRYANNA -> 6,
SEBASTIEN -> 1, JENCARLOS -> 1, ELSA -> 1, HANA -> 3, MASON -> 194, SAVANNA -> 6,
ROWAN -> 6, DENNIS -> 15, JEROME -> 1, BROOKLYNN -> 2, MIRANDA -> 11, KYLER -> 1,
HADLEY -> 2, STEPHANIE -> 46, CAMILA -> 45, MAKENNA -> 3, CARMINE -> 5, KATRINA -> 1,
AMALIA -> 1, EN...
```

saveAsTextFile(filepath)

Write out the elements of the data set as a text file in a filepath directory on the *filesystem*, HDFS or any other Hadoop-supported file system.

```
scala> val onlyInterestedIn = sc.textFile("baby_names.csv").map(line => line.split(",")).map(n
=> (n(1), n(4)))
onlyInterestedIn: org.apache.spark.rdd.RDD[(String, String)] = MappedRDD[27] at map at
<console>:12

scala> onlyInterestedIn.saveAsTextFile("results.csv")

$ ls -al results.csv/
total 824
drwxr-xr-x   8 ernestolee  staff     272 Dec  3 06:53 .
drwxr-xr-x@ 17 ernestolee  staff     578 Dec  3 06:54 ..
-rw-r--r--   1 ernestolee  staff       8 Dec  3 06:53 ._SUCCESS.crc
-rw-r--r--   1 ernestolee  staff    1600 Dec  3 06:53 .part-00000.crc
-rw-r--r--   1 ernestolee  staff    1588 Dec  3 06:53 .part-00001.crc
-rw-r--r--   1 ernestolee  staff       0 Dec  3 06:53 _SUCCESS
-rw-r--r--   1 ernestolee  staff  203775 Dec  3 06:53 part-00000
```

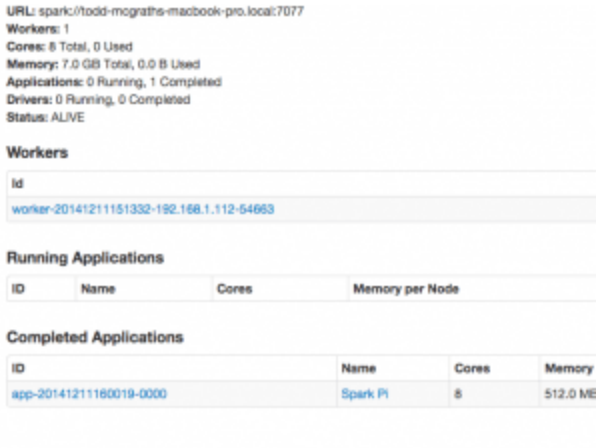
-rw-r--r-- 1 ernestolee staff 202120 Dec 3 06:53 part-00001

SPARK ACTIONS WITH SCALA CONCLUSION

Did you type these examples above as I suggested? I hope so. And do let me know if you have any questions or suggestions for improvement.

Lab 4

Apache Spark Cluster Part 2: Deploy Scala Program to Spark Cluster



How do you deploy a Scala program to a Spark Cluster? In this lab, we'll cover how to build, deploy and run a Scala driver program to a Spark Cluster. The focus will be on a simple example in order to gain confidence and set the foundation for more advanced examples in the future. To keep things interesting, we're going to add some SBT and Sublime 3 editor for fun.

This lab assumes Scala and SBT experience, but if not, it's a chance to gain further understanding of the Scala language and simple build tool (SBT).

Requirements to Deploy to Spark Cluster

- You need to have SBT installed
- Make sure your Spark cluster master and at least one worker is running. Refer to the previous lab, Running a local Standalone Spark Cluster if you run into any issues.

Deploy Scala Program to Spark Cluster Steps

1. Create a directory for the project: mkdir sparksample

2. Create some directories for SBT:

cd sparksample

mkdir project

mkdir src/main/scala

Ok, so you should now be in the sparksample directory and have project/ and src/ dirs.

(3. We're going to sprinkle this **Spark lab** with using Sublime 3 text editor and SBT plugins. So, this step isn't necessary for deploying a scala program to a spark cluster. This is an optional step.)

In any text editor, create a plugins.sbt file in projects directory.

Add the sublime plugin according to: Added sublime plugin:https://github.com/orrsella/sbt-sublime)

4. Create a SBT file in root directory. For this lab, the root directory is sparksample/. Name the file "sparksample.sbt" with the following content

```
name := "Spark Sample"

version := "1.0"

scalaVersion := "2.10.3"

libraryDependencies += "org.apache.spark" %% "spark-core" % "1.1.1"
```

5. Create a file named SparkPi.scala in the src/main/scala directory. Because this is an introductory lab, let's keep things simple and cut-and-paste this code from the Spark samples. The code is:

```
import scala.math.random
```

```
import org.apache.spark._

/** Computes an approximation to pi */
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 = 100000 * slices
  val count = spark.parallelize(1 to 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()
 }
}
```

6. Start SBT from a command prompt: sbt

Running sbt may trigger many file downloads of 3rd party library jars. It depends on if you attempted something similar with SBT in the past and whether your local cache already has the files.

(If you want to continue with Sublime example, run the 'gen-sublime' command from SBT console and open the Sublime project. In the next step, step 6, you can create the sample Scala code in Sublime.)

7. In SBT console, run 'package' to create a jar. The jar will be created in the target/ directory. Note the name of the generated jar; if you follow the previous sparksample.sbt step exactly, the filename will be spark-sample_2.10-1.0.jar

8. Exit SBT, or in a different terminal window, call the "spark-submit" script with the appropriate –master arg value. For example:

```
../spark-1.6.1-bin-hadoop2.4/bin/spark-submit --class "SparkPi" --master
spark://todd-mcgraths-macbook-pro.local:7077
target/scala-2.10/spark-sample_2.10-1.0.jar
```

So, in this example, it's safe to presume I have the following directory structure:

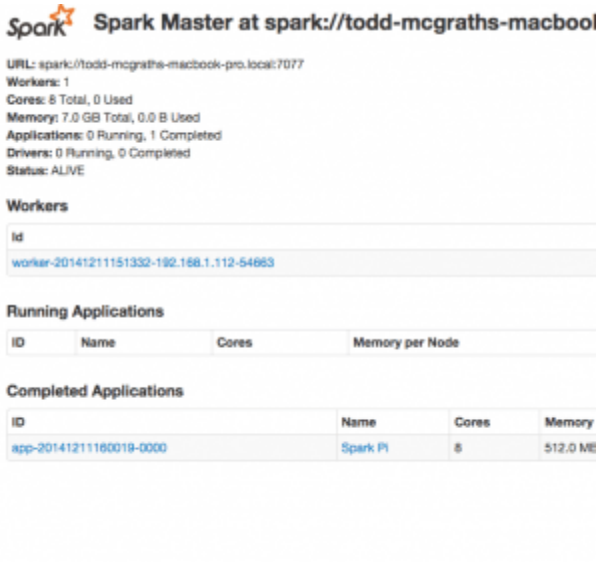
parentdir

-spark-1.6.1-bin/hadoop2.4

-sparksample

We can assume this because I'm running ../spark-1.6.0-bin-hadoop2.4/bin/spark-submit from the sparksample directory.

9. You should see output "Pi is roughly..." and if you goto Spark UI, you should see the "Spark Pi" in completed applications:



Completed Application after running in Spark Cluster

CONCLUSION

That's it. You've built, deployed and ran a Scala driver program to Spark Cluster. Simple, I know, but with this experience, you are in good position to move to more complex examples and use cases. Let me know if you have any questions in the comments below.

Lab 5

Spark SQL CSV Examples in Scala

In this Spark SQL lab, we will use Spark SQL with a CSV input data source. We will continue to use the baby names CSV source file as used in the previous What is Spark lab. This lab presumes the reader is familiar with using SQL with relational databases and would like to know how to use Spark SQL in Spark.

OVERVIEW

Earlier versions of Spark SQL required a certain kind of Resilient Distributed Data set called SchemaRDD. Starting with Spark 1.3, Schema RDD was renamed to DataFrame.

DataFrames are composed of Row objects accompanied with a schema which describes the data types of each column. A DataFrame may be considered similar to a table in a traditional relational database. A DataFrame may be created from a variety of input sources including CSV text files.

This intro to Spark SQL lab will use a CSV file from a previous Spark lab.

Download the CSV version of baby names file here:

https://health.data.ny.gov/api/views/jxy9-yhdk/rows.csv?accessType=DOWNLOAD

For this and other Spark labs, the file has been named baby_names.csv

Methodology

We're going to use the spark shell and the spark-csv package available from Spark Packages to make our lives easier. It is described as a "library for parsing and querying CSV data with Apache Spark, for Spark SQL and DataFrames" This library is compatible with Spark 1.3 and above.

Spark SQL CSV Example lab Part 1

1. Depending on your version of Scala, start the spark shell with a packages command line argument.

At time of this writing, scala 2.10 version:

SPARK_HOME/bin/spark-shell --packages com.databricks:spark-csv_2.10:1.3.0

where SPARK_HOME is the root directory of your Spark directory; i.e. ~/Development/spark-1.4.1-bin-hadoop2.4 or c:/dev/spark-1.4.1-bin-hadoop2.4

At time of this writing, scala 2.11 version:

SPARK_HOME/bin/spark-shell --packages com.databricks:spark-csv_2.11:1.3.0

2. Using the available sqlContext from the shell load the CSV read, format, option and load functions

Welcome to

```
      ____              __
     / __/__  ___ _____/ /__
    _\ \/ _ \/ _ `/ __/  '_/
   /___/ .__/\_,_/_/ /_/\_\   version 1.4.1
      /_/
```

Using Scala version 2.10.4 (Java HotSpot(TM) 64-Bit Server VM, Java 1.7.0_79)
Type in expressions to have them evaluated.
Type :help for more information.
Spark context available as sc.
SQL context available as sqlContext.

```
scala> val baby_names =
sqlContext.read.format("com.databricks.spark.csv").option("header",
"true").option("inferSchema", "true").load("baby_names.csv")
baby_names: org.apache.spark.sql.DataFrame = [Year: int, First Name: string, County:
string, Sex: string, Count: int]
```

In the above code, we are specifying the desire to use *com.databricks.spark.csv* format from the package we passed to the shell in step 1. "header" set to true signifies the first row has column names. "inferSchema" instructs Spark to attempt to infer the schema of the CSV and finally load function passes in the path and name of the CSV source file. In this example, we can tell the baby_names.csv file is in the same directory as where the spark-shell script was launched.

3. Register a temp table

```
scala> baby_names.registerTempTable("names")
```

4. We're now ready to query using SQL such as finding the distinct years in the CSV

```
scala> val distinctYears = sqlContext.sql("select distinct Year from names")
distinctYears: org.apache.spark.sql.DataFrame = [Year: int]

scala> distinctYears.collect.foreach(println)
[2007]
[2008]
[2009]
[2010]
[2011]
[2012]
[2013]
```

5. It might be handy to know the schema

```
scala> baby_names.printSchema
root
|-- Year: integer (nullable = true)
|-- First Name: string (nullable = true)
|-- County: string (nullable = true)
|-- Sex: string (nullable = true)
|-- Count: integer (nullable = true)
```

Spark SQL CSV Example lab Part 2

But, let's try some more advanced SQL, such as determining the names which appear most often in the data

```
scala> val popular_names = sqlContext.sql("select distinct(`First Name`), count(County) as cnt from names group by `First Name` order by cnt desc LIMIT 10")
popular_names: org.apache.spark.sql.DataFrame = [First Name: string, cnt: bigint]

scala> popular_names.collect.foreach(println)
[JACOB,284]
[EMMA,284]
[LOGAN,270]
[OLIVIA,268]
[ISABELLA,259]
[SOPHIA,249]
[NOAH,247]
[MASON,245]
[ETHAN,239]
[AVA,234]
```

But as we learned from Spark transformation and action labs, this doesn't necessarily imply the most popular names. We need to utilize the Count column value:

```
scala> val popular_names = sqlContext.sql("select distinct(`First Name`), sum(Count) as cnt from names group by `First Name` order by cnt desc LIMIT 10")
popular_names: org.apache.spark.sql.DataFrame = [First Name: string, cnt: bigint]

scala> popular_names.collect.foreach(println)
[MICHAEL,10391]
[ISABELLA,9106]
[MATTHEW,9002]
[JAYDEN,8948]
[JACOB,8770]
[JOSEPH,8715]
[SOPHIA,8689]
[DANIEL,8412]
[ANTHONY,8399]
[RYAN,8187]
```

SPARK SQL FURTHER REFERENCE

Check out Spark SQL with Scala labs for more Spark SQL with Scala including Spark SQL with JSON and Spark SQL with JDBC. And for labs in Scala, see Spark labs in Scala page.

Featured image credit https://flic.kr/p/3CrmX

Lab 6

Spark SQL JSON Examples

This lab covers using Spark SQL with a JSON file input data source in Scala. If you are interested in using Python instead, check out Spark SQL JSON in Python lab page.

Spark SQL JSON Overview

We will show examples of JSON as input source to Spark SQL's SQLContext. This Spark SQL lab with JSON has two parts. Part 1 focus is the "happy path" when using JSON with Spark SQL. Part 2 covers a "gotcha" or something you might not expect when using Spark SQL JSON data source.

By the way, If you are not familiar with Spark SQL, a couple of references include a summary of Spark SQL chapter lab and the first Spark SQL CSV lab.

Methodology

We build upon the previous baby_names.csv file as well as a simple file to get us started which I've called customers.json. Here is a gist of customers.json.

Spark SQL JSON Example lab Part 1

1. Start the spark shell

$SPARK_HOME/bin/spark–shell

2. Load the JSON using the jsonFile function from the provided sqlContext. The following assumes you have customers.json in the same directory as from where the spark-shell script was called.

```
RBH12103:spark-1.4.1-bin-hadoop2.4 tmcgrath$ bin/spark-shell
Welcome to
      ____              __
     / __/__  ___ _____/ /__
    _\ \/ _ \/ _ `/ __/  '_/
   /___/ .__/\_,_/_/ /_/\_\   version 1.4.1
      /_/

Using Scala version 2.10.4 (Java HotSpot(TM) 64-Bit Server VM, Java 1.7.0_79)
Type in expressions to have them evaluated.
Type :help for more information.
Spark context available as sc.
SQL context available as sqlContext.

scala> val customers = sqlContext.jsonFile("customers.json")
warning: there were 1 deprecation warning(s); re-run with -deprecation for details
customers: org.apache.spark.sql.DataFrame = [address:
struct<city:string,state:string,street:string,zip:string>, first_name: string, last_name:
string]
```

3. Register the data as a temp table to ease our future SQL queries

```
scala> customers.registerTempTable("customers")
```

4. We are now in a position to run some Spark SQL

```
scala> val firstCityState = sqlContext.sql("SELECT first_name, address.city, address.state
FROM customers")
firstCityState: org.apache.spark.sql.DataFrame = [first_name: string, city: string, state:
string]

scala> firstCityState.collect.foreach(println)
[James,New Orleans,LA]
[Josephine,Brighton,MI]
[Art,Bridgeport,NJ]
```

Ok, we started with a simple example, but the real world is rarely this simple. So, in part 2, we'll cover a more complex example.

Spark SQL JSON Example lab Part 2

If you run the customers.json from part 1 through http://jsonlint.com, it will not validate. You might be surprised to know that creating invalid JSON for Part 1 was intentional. Why? We needed JSON source which works well with Spark SQL out of the box.

If you read the Spark SQL documentation closely:

"Note that the file that is offered as *a json file* is not a typical JSON file. Each line must contain a separate, self-contained valid JSON object. As a consequence, a regular multi-line JSON file will most often fail."

But, what happens if we have valid JSON?

In this part of the Spark SQL JSON lab, we'll cover how to use valid JSON as an input source for Spark SQL.

As input, we're going to convert the baby_names.csv file to baby_names.json. There are many CSV to JSON conversion tools available... just search for "CSV to JSON converter".

I converted and reduced the baby_names.csv to the following:

```
[{
        "Year": "2013",
        "First Name": "DAVID",
        "County": "KINGS",
        "Sex": "M",
        "Count": "272"
}, {
        "Year": "2013",
        "First Name": "JAYDEN",
        "County": "KINGS",
        "Sex": "M",
        "Count": "268"
}, {
        "Year": "2013",
        "First Name": "JAYDEN",
```

```
        "County": "QUEENS",
        "Sex": "M",
        "Count": "219"
}, {
        "Year": "2013",
        "First Name": "MOSHE",
        "County": "KINGS",
        "Sex": "M",
        "Count": "219"
}, {
        "Year": "2013",
        "First Name": "ETHAN",
        "County": "QUEENS",
        "Sex": "M",
        "Count": "216"
}]
```

I saved this as a file called baby_names.json

Steps

1. Start the spark-shell from the same directory containing the baby_names.json file

2. Load the JSON using the Spark Context *wholeTextFiles* method which produces a PairRDD. Use *map* to create the new RDD using the value portion of the pair.

```
scala> val jsonRDD = sc.wholeTextFiles("baby_names.json").map(x => x._2)
jsonRDD: org.apache.spark.rdd.RDD[String] = MapPartitionsRDD[10] at map at
<console>:21
```

3. Read in this RDD as JSON and confirm the schema

```
scala> val namesJson = sqlContext.read.json(jsonRDD)
namesJson: org.apache.spark.sql.DataFrame = [Count: string, County: string, First Name:
string, Sex: string, Year: string]

scala> namesJson.printSchema
root
 |-- Count: string (nullable = true)
 |-- County: string (nullable = true)
 |-- First Name: string (nullable = true)
 |-- Sex: string (nullable = true)
 |-- Year: string (nullable = true)
```

```
scala>
```

Lab 7