Table of Contents

Apache Spark Notes

Notes prepared for future quick reference and is prepared from the following sources

- Learning Spark
- Spark official documentation
- StackOverflow questions
- My own exercises
- Other internet sources

Exercises

I'd suggest you to turn **ON** only ERROR logging from spark-shell so that the focus is on your commands and data

```
scala> import org.apache.log4j.{Level, Logger}
import org.apache.log4j.{Level, Logger}

scala> Logger.getRootLogger().setLevel(Level.ERROR)
```

If you want to turn **ON** the DEBUG logs from the code provided in these code katas then, run the following command

```
scala> Logger.getLogger("mylog").setLevel(Level.DEBUG)
```

To turn them **OFF**

```
scala> Logger.getLogger("mylog").setLevel(Level.ERROR)
```

Datasets

Star Wars Kid

Read the story from Andy Baio's blog post here.

Video: Original, Mix

Summary:

- Released in 2003
- Downloaded 1.1 million times the first 2 weeks
- Since been downloaded 40 million times since it was moved to YouTube
- On April 29 2003, Andy Baio renamed it to *Star_Wars_Kid.wmv* and posted it on his site
- In 2008, Andy Baio shared the Apache logs for the first 6 months from his site
- File is 1.6 GB unzipped and available via BitTorent

Data:

- Sample: swk_small.log TODO - add link from github
- Original: Torrent link on this website

Univ of Florida

http://www.stat.ufl.edu/~winner/datasets.html

Submitting applications to Spark

From http://stackoverflow.com/questions/34391977/spark-submit-does-automatically-upload-the-jar-to-cluster

This seems to be a common problem, here's what my research into this has turned up. In practice it seems that a lot of people ask this question here and on other forums, so I would think the default behavior is that Spark's driver does NOT push jars around a cluster, just the classpath. The point of confusion, that I along with other newcomers commonly suffer from is this:

The driver does NOT push your jars to the cluster. The master in the cluster DOES push your jars to the workers. In theory. I see various things in the docs for Spark that seem to contradict the idea that if you pass an assembly jar and/or dependencies to spark-submit with --jars, that these jars are pushed out to the cluster. For example, on the plus side,

"When using spark-submit, the application jar along with any jars included with the --jars option will be automatically transferred to the cluster. " That's from http://spark.apache.org/docs/latest/submitting-applications.html#advanced-dependency-management

So far so good right? But that is only talking about once your master has the jars, it can push them to workers.

There's a line on the docs for spark-submit, for the main parameter application-jar. Quoting another answer I wrote:

"I assume they mean most people will get the classpath out through a driver config option. I know most of the docs for spark-submit make it look like the script handles moving your code around the cluster, but it only moves the classpath around for you. The driver does not load the jars to the master. For example in this line from Launching Applications with spark-submit explicitly says you have to move the jars yourself or make them "globally available":

application-jar: Path to a bundled jar including your application and all dependencies. The URL must be globally visible inside of your cluster, for instance, an hdfs:// path or a file:// path that is present on all nodes." From an answer I wrote on Spark workers unable to find JAR on EC2 cluster

So I suspect we are seeing a "bug" in the docs. I would explicitly put the jars on each node, or "globally available" via NFS or HDFS, for another reason from the docs:

From Advanced Dependency Management, it seems to present the best of both worlds, but also a great reason for manually pushing your jars out to all nodes:

local: - a URI starting with local:/ is expected to exist as a local file on each worker node. This means that no network IO will be incurred, and works well for large files/JARs that are pushed to each worker, or shared via NFS, GlusterFS, etc. Bottom line, I am coming to realize that to truly understand what Spark is doing, we need to bypass the docs and read the source code. Even the Java docs don't list all the parameters for things like parallelize().

Caching

- We can use persist() to tell spark to store the partitions
- On node failures (that persist data), lost partitions are recomputed by spark
- In Scala and Java, default persist() stores the data on heap as **unserialized** objects
- We can also **replicate** the data on to multiple nodes to handle node failures without slowdown
- Persistence is at partition level. i.e. there should be enough memory to cache all the data of a partition. Partial caching isn't supported as of 1.5.2
- LRU policy is used to evict cached partitions to make room for new ones. Wen evicting,
    - memory_only ones are recomputed next time when they are accessed
    - memory_and_disk ones are written to disk

TODO

- add examples of various persist mechanisms
- example for tachyon
- tungston

Pair RDD

Exercise 1

Goal

Understand different ways of creating **Pair** RDD's

Problem(s)

Problem 1: Create a Pair RDD of this collection Seq("the", "quick", "brown", "fox", "jumps", "over", "the", "lazy", "dog")?

Problem 2: Create Pair RDD's from external storages.

Answer(s)

Answer 1

In general, we can extract a key from the data by running a map() transformation on it. Pair's are represented using Tuples (key, value).

```
scala> val words=sc.parallelize(Seq("the", "quick", "brown", "fox", "jumps", "over", "
the", "lazy", "dog"))
words: org.apache.spark.rdd.RDD[String] = ParallelCollectionRDD[45] at parallelize at
<console>:27

scala> val wordPair=words.map( w => (w.charAt(0), w) )
wordPair: org.apache.spark.rdd.RDD[(Char, String)] = MapPartitionsRDD[46] at map at <c
onsole>:29

scala> wordPair.foreach(println)
(o,over)
(l,lazy)
(d,dog)
(t,the)
(j,jumps)
(t,the)
(f,fox)
(b,brown)
(q,quick)
```

Answer 2

TODO

Pair RDD - Transformations

Reference: http://spark.apache.org/docs/latest/programming-guide.html#transformations

Excercise 1

Goal

Understand the usage of reduceByKey() transformation.

> reduceByKey(func, [numTasks]): When called on a dataset of (K, V) pairs, returns a dataset of (K, V) pairs where the values for each key are aggregated using the given reduce function func, which must be of type (V,V) => V. Like in groupByKey, the number of reduce tasks is configurable through an optional second argument.

Problem(s)

Problem 1: Calculate page views by day for the star wars video (download link).

Problem 2: Word count

Answer(s)

Answer 1

```
scala> val logs=sc.textFile("file:///c:/_home/swk_small.log")
logs: org.apache.spark.rdd.RDD[String] = MapPartitionsRDD[50] at textFile at <console>:
27

scala> val parsedLogs=logs.map(line => parseAccessLog(line))
parsedLogs: org.apache.spark.rdd.RDD[scala.collection.mutable.Map[String,String]] = Ma
pPartitionsRDD[51] at map at <console>:31

scala> //create a pair RDD and do reduceByKey() transformation

scala> val viewsByDate=parsedLogs.map( m => (m.getOrElse("accessDate","unknown"), 1)).
reduceByKey((x,y) => x+y)
viewsByDate: org.apache.spark.rdd.RDD[(String, Int)] = ShuffledRDD[55] at reduceByKey
at <console>:33

scala> viewsByDate.foreach(println)
(03-02-2003,1)
(05-04-2003,1)
(12-04-2003,1913)
(02-03-2003,1)
(26-02-2003,2)
(07-02-2003,1)
(05-01-2003,1)
(22-03-2003,1)
(19-02-2003,1)
(02-01-2003,2)
(11-04-2003,4162)
(22-01-2003,1)
(21-03-2003,1)
(10-04-2003,3902)
(01-03-2003,1)
(05-02-2003,1)
(18-03-2003,2)
(25-02-2003,1)
(06-02-2003,1)
(09-04-2003,1)
(20-02-2003,1)
(23-03-2003,1)
(06-04-2003,1)
```

Answer 2

Excercise 2

Goal

Understand the usage of foldByKey() transformation.

Problem 1: Calculate page views by day for the star wars video (download link).

Answer(s)

Answer 1

```
scala> val logs=sc.textFile("file:///c:/_home/swk_small.log")
logs: org.apache.spark.rdd.RDD[String] = MapPartitionsRDD[74] at textFile at <console>:
27

scala> val parsedLogs=logs.map(line => parseAccessLog(line))
parsedLogs: org.apache.spark.rdd.RDD[scala.collection.mutable.Map[String,String]] = Ma
pPartitionsRDD[75] at map at <console>:31

scala> val viewDates=parsedLogs.map( m => (m.getOrElse("accessDate","unknown"), 1))
viewDates: org.apache.spark.rdd.RDD[(String, Int)] = MapPartitionsRDD[77] at map at <c
onsole>:33

scala> val viewsByDate=viewDates.foldByKey(0)((x,y)=> x+y)
viewsByDate: org.apache.spark.rdd.RDD[(String, Int)] = ShuffledRDD[78] at foldByKey at
 <console>:35

scala> viewsByDate.foreach(println)
(03-02-2003,1)
(26-02-2003,2)
(05-04-2003,1)
(05-01-2003,1)
(19-02-2003,1)
(11-04-2003,4162)
(21-03-2003,1)
(01-03-2003,1)
(12-04-2003,1913)
(18-03-2003,2)
(02-03-2003,1)
(06-02-2003,1)
(20-02-2003,1)
(07-02-2003,1)
(23-03-2003,1)
(22-03-2003,1)
(06-04-2003,1)
(02-01-2003,2)
(22-01-2003,1)
(10-04-2003,3902)
(05-02-2003,1)
(25-02-2003,1)
(09-04-2003,1)
```

Excercise 3

Goal

Understand the usage of combineByKey() transformation.

> def combineByKey[C] (createCombiner: V => C, mergeValue: (C, V) => C, mergeCombiners: (C, C) => C): RDD[(K, C)]

Problem 1: From here

Answer(s)

Answer 1

```
scala> val a = sc.parallelize(List("dog","cat","gnu","salmon","rabbit","turkey","wolf",
"bear","bee"), 3)
a: org.apache.spark.rdd.RDD[String] = ParallelCollectionRDD[87] at parallelize at <con
sole>:27

scala> val b = sc.parallelize(List(1,1,2,2,2,1,2,2,2), 3)
b: org.apache.spark.rdd.RDD[Int] = ParallelCollectionRDD[88] at parallelize at <consol
e>:27

scala> val c = b.zip(a)
c: org.apache.spark.rdd.RDD[(Int, String)] = ZippedPartitionsRDD2[89] at zip at <conso
le>:31

scala> c.collect
res78: Array[(Int, String)] = Array((1,dog), (1,cat), (2,gnu), (2,salmon), (2,rabbit),
 (1,turkey), (2,wolf), (2,bear), (2,bee))

scala> def createCombiner(v:String):List[String] = List[String](v)
createCombiner: (v: String)List[String]

scala> def mergeValue(acc:List[String], value:String) : List[String] = value :: acc
mergeValue: (acc: List[String], value: String)List[String]

scala> def mergeCombiners(acc1:List[String], acc2:List[String]) : List[String] = acc2
::: acc1
mergeCombiners: (acc1: List[String], acc2: List[String])List[String]

scala> val result=c.combineByKey(createCombiner,mergeValue,mergeCombiners)
result: org.apache.spark.rdd.RDD[(Int, List[String])] = ShuffledRDD[91] at combineByKe
y at <console>:39

scala> result.collect
res80: Array[(Int, List[String])] = Array((1,List(turkey, cat, dog)), (2,List(bee, bea
r, wolf, rabbit, salmon, gnu)))
```

Parallelism

Every RDD has a fixed number of partitions that determine the degree of parallelism to use when executing operations on RDD

Tuning

- When performing aggregations and grouping operations, you can tell Spark to use a specific number of partitions. For e.g. reduceByKey(func, customParallelismOrPartitionsToSue)
- If you want to change the partitioning of an RDD outside of these grouping or aggregation operations, then you can use repartition() or coaleesce()
- repartition() reshuffles the data acroos network to create a new set of partitions
- coalesce() avoids reshuffling data ONLY if we are decreasing the number of existing RDD partitions (Use rdd.partitions.size() to see the current RDD partitions)
- Tackle performance intensive commands like collect(), groupByKey(), reduceByKey()
- See this post to see a small e.g. of how coalesc works
- We can run multiple jobs within a single SparkContext parallerly by using threads. Look at the answers from Patrick Liu and yuanbosoft in this post
    - Within each Spark application, multiple “jobs” (Spark actions) may be running concurrently if they were submitted by different threads. This is common if your application is serving requests over the network.
- Improving Garbage Collection
    - Below JDK 1.7 update 4
        - CMS - Concurrent Mark and Sweep
            - focus is on low-latency (doesn't do compaction)
            - hence use it for real-time streaming
        - Parallel GC
            - Focus is on higher-throughput but results in higher pause-times (performs whole-heap only compaction.
            - hence use it for batch
    - Above or equal to JDK 1.7 update 4
        - G1
            - Primarily meant for server-side and multi-core machines with large memory
            -

Problems

Problem 1: The following is data of few users. It cotains four columns (userid, date, item1 and item2). Find each user's latest day's records?

Input

```
val data="""
user date      item1 item2
1    2015-12-01 14  5.6
1    2015-12-01 10  0.6
1    2015-12-02 8   9.4
1    2015-12-02 90  1.3
2    2015-12-01 30  0.3
2    2015-12-01 89  1.2
2    2015-12-30 70  1.9
2    2015-12-31 20  2.5
3    2015-12-01 19  9.3
3    2015-12-01 40  2.3
3    2015-12-02 13  1.4
3    2015-12-02 50  1.0
3    2015-12-02 19  7.8
"""
```

Expected output

- For user 1, the *latest date* is 2015-12-02 and he has *two records for that particular* date.
- For user 2, the *latest date* is 2015-12-31 and he has *two records for that particular* date.
- For user 3, the *latest date* is 2015-12-02 and he has *three records for that particular* date.

Problem 2: From the tweet data set here, find the following (This is my own solution version of excellent article: Getting started with Spark in practice)

- all the tweets by user

- how many tweets each user has
- all the persons mentioned on tweets
- Count how many times each person is mentioned
- Find the 10 most mentioned persons
- Find all the hashtags mentioned on a tweet
- Count how many times each hashtag is mentioned
- Find the 10 most popular Hashtags

Input sample

```
{
   "id":"572692378957430785",
   "user":"Srkian_nishu :)",
   "text":"@always_nidhi @YouTube no i dnt understand bt i loved the music nd their da
nce awesome all the song of this mve is rocking",
   "place":"Orissa",
   "country":"India"
}
{
   "id":"572575240615796737",
   "user":"TagineDiningGlobal",
   "text":"@OnlyDancers Bellydancing this Friday with live music call 646-373-6265 htt
p://t.co/BxZLdiIVM0",
   "place":"Manhattan",
   "country":"United States"
}
```

Problem 3: Demonstrate data virtualization capabilities of SparkSQL by joining data across different data stores i.e. rdbms, parquet, avro

Solutions

Please create a pull request to provide more elegant solutions. Specify the improvements.

Solution 1

```
//Step 1. Prepare dataset

val input = """user date      item1 item2
1    2015-12-01 14  5.6
1    2015-12-01 10  0.6
1    2015-12-02 8   9.4
1    2015-12-02 90  1.3
2    2015-12-01 30  0.3
2    2015-12-01 89  1.2
2    2015-12-30 70  1.9
2    2015-12-31 20  2.5
3    2015-12-01 19  9.3
3    2015-12-01 40  2.3
3    2015-12-02 13  1.4
3    2015-12-02 50  1.0
3    2015-12-02 19  7.8
"""

val inputLines=sc.parallelize(input.split("\\r?\\n"))
//filter the header row
val data=inputLines.filter(l=> !l.startsWith("user") )
data.foreach(println)

//Step 2. Find the latest date of each user

val keyByUser=data.map(line => { val a = line.split("\\s+"); ( a(0), line ) })
//For each user, find his latest date
val latestByUser = keyByUser.reduceByKey( (x,y) => if(x.split("\\s+")(1) > y.split("\\
s+")(1)) x else y )
latestByUser.foreach(println)

//Step 3. Join the original data with the latest date to get the result

val latestKeyedByUserAndDate = latestByUser.map( x => (x._1 + ":"+x._2.split("\\s+")(1
), x._2))
val originalKeyedByUserAndDate = data.map(line => { val a = line.split("\\s+"); ( a(0)
 +":"+a(1), line ) })
val result=latestKeyedByUserAndDate.join(originalKeyedByUserAndDate)
result.foreach(println)

//Step 4. Transform the result into the format you desire

def createCombiner(v:(String,String)):List[(String,String)] = List[(String,String)](v)
def mergeValue(acc:List[(String,String)], value:(String,String)) : List[(String,String
)] = value :: acc
def mergeCombiners(acc1:List[(String,String)], acc2:List[(String,String)]) : List[(Str
ing,String)] = acc2 ::: acc1
//use combineByKey
val transformedResult=result.mapValues(l=> { val a=l._2.split(" +"); (a(2),a(3)) } ).c
ombineByKey(createCombiner,mergeValue,mergeCombiners)
transformedResult.foreach(println)
```

Solution 2

Setup

A single tweet looks as below

```
{
   "id":"572692378957430785",
   "user":"Srkian_nishu :)",
   "text":"@always_nidhi @YouTube no i dnt understand bt i loved the music nd their da
nce awesome all the song of this mve is rocking",
   "place":"Orissa",
   "country":"India"
}
```

Let us create a case class to represent each tweet

```
case class Tweet(id:String, user:String, text:String, place:String, country:String)
```

Transform the file into this RDD[Tweet].

For this we use Jackson library which comes bundled with Apache Spark already. We will not use object binding feature (mapper.readValue()) due to this issue. Instead, we will use TreeModel API (mapper.readTree)

```
/**
    Create a holder object with a transient lazy field to hold the ObjectMapper.
    This way, the ObjectMapper will be instantiated at the destination and we can shar
e the instance.

    NOTE: This is just to show the functionality. Ideally we would like to use mapPart
itions
**/
object Holder extends Serializable {
  import com.fasterxml.jackson.databind.{DeserializationFeature, ObjectMapper}
  import com.fasterxml.jackson.module.scala.DefaultScalaModule

  @transient lazy val mapper = new ObjectMapper()
  mapper.registerModule(DefaultScalaModule)
  mapper.configure(DeserializationFeature.FAIL_ON_UNKNOWN_PROPERTIES, false)
}

val tweets=sc.textFile("file:///c:/_home/reduced-tweets.json").map( line => {
  val t = Holder.mapper.readTree(line)
  Tweet(t.path("id").asText, t.path("user").asText, t.path("text").asText, t.path("pla
ce").asText, t.path("country").asText)
})
```

all the tweets by user

how many tweets each user has

all the persons mentioned on tweets

Count how many times each person is mentioned

Find the 10 most mentioned persons

Find all the hashtags mentioned on a tweet

Count how many times each hashtag is mentioned

Find the 10 most popular Hashtags

Techniques & Best Practices

General

- Set spark.app.id when running on YARN. It enables monitoring. More details.
- Follow best practices for Scala
- Minimize network data shuffling by using shared variables, and favoring reduce operations (which reduce the data locally before shuffling across the network) over grouping (which shuffles all the data across the network)
- Properly configure your temp work dir cleanups

Serialization

- Closure serialization
  - Every task run from Driver to Worker gets serialized
  - The closure and its environment gets serialized to worker **every time it is used. Not just once per node.** This might slow things down considerably, specially when you have large variables in environment. This is where broadcast variables can help.
  - Default java serialization mechanism is used to serialize closures.
- Result serialization: Every result from every task gets serialized at some point
- In general, tasks larger than about 20 KB are probably worth optimizing
- If your objects are large, you may also need to increase the `spark.kryoserializer.buffer` config. This value needs to be large enough to hold the largest object you will serialize.

Broadcast and Accumulator

- Both are ties to a single SparkContext and hence can't be shared across contexts
- According to a study, broadcasting data of around 1GB to larger workers (40 max) reduces the performance considerably. Refer to this paper.

Logging

- A Spark Executor writes all its logs (including INFO, DEBUG) to *stderr*. The *stdout* is empty

Shuffling

- This setting configures the serializer used for not only shuffling data between worker nodes but also when serializing RDDs to disk

Monitoring Spark Apps

- Web UI
    - Spark launches as embedded web-app by default to monitor a running application. this is launched on the Driver
        - It launches the monitoring app on 4040 by default.
        - If multiple apps are running simultaneously then they bind to successive ports
        - REST interface
        - Publish metrics to aggregation systems like Graphite, Ganglia and JMX

Scheduling

- By default FIFO is used
- Can be configured to use FAIR scheduler by changing *spark.scheduler.mode*. Spark assigns tasks between jobs in a “round robin” fashion in FAIR mode

SparkSQL

- Exposes 3 processing interfaces: SQL, HiveQL and language integrated queries
- Can be used in 2 modes
  - as a library
    - data processing tasks are expressed as SQL, HiveQL or integrated queries in a Scala app
    - 3 key abstractions
      - *SQLContext*
      - *HiveContext*
      - *DataFrame*
  - as a distributed SQL execution engine
    - allows multiple users to **share** a single spark cluster
    - allows centralized caching across all jobs and across multiple data stores
- DataFrame
  - represents a distributed collection of rows organized into named columns
    - Executing SQL queries provides us a DataFrame as the result
    - is schema aware i.e knows names and types of columns
    - provides methods for processing data
    - can be easily converted to regular RDD
    - can be registered as a temporary table to run SQL/HiveQL
    - 2 ways to create a DF
      - from existing RDD
        - *toDF* if we can infer schema using a case class
        - *createDataFrame* is you want to pass the Schema explicitly (StructType, StructField)
      - from external DataSources
        - single unified interace to create DF, either from data stores (MySQL, PostgresSQL, Oracle, Cassandra) or from files (JSON, Parquet, ORC, CSV, HDFS, local, S3)
        - built-in support for JSON, Parquet, ORC, JDBC, Hive
  - uses *DataFrameReader* to read data from external datasources
    - can specify partiotioning, format and data source specific options
    - SQL/HiveContext has a factory method called *read()* that returns an instance of DataFrameReader
  - uses DataFrameWriter to write data to external datasources
    - can specify partiotioning, format and data source specific options
  - build-in functions

        - optimized for faster execution through code generation
        - import org.apache.spark.sql.functions._
        -
- Optimization techniques used
    - Reduced Disk IO
        - Skip rows
            - if a data store (for e.g. Parquet, ORC) maintains statistical info about data (min, max of a particular column in a group) then SparkSQL can take advantage of it to skip these groups
        - Skip columns (Use support of Parquet)
        - skip non-required partitions
        - Predicate Push down i.e. pushing filtering predicates to run natively on data stores like Cassandra, DB etc.
    - In-memory columnar caching
        - cache only required columns from any data store
        - compress the cached columns (using snappy) to minimize memory usage and GC
            - SparkSQL can automatically select a compression codec for a column based on data type
        - columnar format enables using of efficient compression techniques
            - run length encoding
            - delta encoding
            - dictionary encoding
    - Query optimization
        - uses both cost-based (in the physical planning phase) and rule-based (in the logical optimization phase) optimization
        - can even optimize across functions
        - code generation

References

- https://ogirardot.wordpress.com/2015/01/09/changing-sparks-default-java-serialization-to-kryo/
- https://ogirardot.wordpress.com/2014/12/11/apache-spark-memory-management-and-graceful-degradation/