Learning Spark
by Holden Karau et. al.

CHAPTER 05: LOADING AND SAVING YOUR DATA

Overview: Loading and saving your data

- Motivation
- File Formats
  - Text files
  - JSON
  - CSV and TSV
  - SequenceFiles
  - Object Files
  - Hadoop Input and Output Formats
  - File Compression
- Filesystems
  - Local/"Regular" FS
  - Amazon S3
  - HDFS

Overview: Loading and saving your data

- Structured Data with Spark SQL
  - Apache Hive
  - JSON
- Databases
  - Java Database Connectivity
  - Cassandra
  - HBase
  - Elasticsearch

5.1 Motivation

- Spark supports a wide range of input and output sources.
- Spark can access data through the InputFormat and OutputFormat interfaces used by Hadoop MapReduce, which are available for many common file formats and storage systems (e.g., S3, HDFS, Cassandra, HBase, etc.)
- You will want to use higher-level APIs built on top of these raw interfaces.

5.2 File Formats

Table 5-1. Common supported file formats

Format name	Structured	Comments
Text files	No	Plain old text files. Records are assumed to be one per line.
JSON	Semi	Common text-based format, semistructured; most libraries require one record per line.
CSV	Yes	Very common text-based format, often used with spreadsheet applications.
SequenceFiles	Yes	A common Hadoop file format used for key/value data.
Protocol buffers	Yes	A fast, space-efficient multilanguage format.
Object files	Yes	Useful for saving data from a Spark job to be consumed by shared code. Breaks if you change your classes, as it relies on Java Serialization.

5.2.1 Text Files

Example 5-1. Loading a text file in Python

```
input = sc.textFile
("file:///home/holden/repos/spark/README.md")
```

Example 5-2. Loading a text file in Scala

```
val input = sc.textFile
("file:///home/holden/repos/spark/README.md")
```

Example 5-3. Loading a text file in Java

```
JavaRDD<String> input =
sc.textFile("file:///home/holden/repos/spark/README.md")
```

5.2.1 Text Files

Example 5-4. Average value per file in Scala

```
val input = sc.wholeTextFiles("file://home/holden/salesFiles")
val result = input.mapValues{y =>
        val nums = y.split(" ").map(x => x.toDouble)
        nums.sum / nums.size.toDouble
}
```

Example 5-5. Saving as a text file in Python

```
result.saveAsTextFile(outputFile)
```

5.2.2 JSON

Example 5-6. Loading unstructured JSON in Python

```
import json
 data = input.map(lambda x: json.loads(x))
```

5.2.2 JSON

Example 5-7. Loading JSON in Scala

```
import com.fasterxml.jackson.module.scala.DefaultScalaModule
 import com.fasterxml.jackson.module.scala.experimental.ScalaObjectMapper
 import com.fasterxml.jackson.databind.ObjectMapper
 import com.fasterxml.jackson.databind.DeserializationFeature
 ...
 case class Person(name: String, lovesPandas: Boolean) // Must be a top-level class
...
 // Parse it into a specific case class. We use flatMap to handle errors
 // by returning an empty list (None) if we encounter an issue and a
 // list with one element if everything is ok (Some(_)).
 val result= input.flatMap(record => {
try {
 Some(mapper.readValue(record, classOf[Person]))
} catch {
 case e: Exception => None
}})
```

5.2.2 JSON

Example 5-9. Saving JSON in Python

```
(data.filter(lambda x: x['lovesPandas']).map(lambda x:
json.dumps(x)) .saveAsTextFile(outputFile))
```

Example 5-10. Saving JSON in Scala

```
result.filter(p =>
P.lovesPandas).map(mapper.writeValueAsString(_))
.saveAsTextFile(outputFile)
```

5.2.3 CSV/TSV

Example 5-12. Loading CSV with textFile() in Python

```
import csv
 import StringIO
 ...
 def loadRecord(line):
"""Parse a CSV line"""
input = StringIO.StringIO(line)
 reader = csv.DictReader(input, fieldnames=["name",
"favouriteAnimal"]) return reader.next()
input = sc.textFile(inputFile).map(loadRecord)
```

5.2.3 CSV/TSV

Example 5-13. Loading CSV with textFile() in Scala

```
import Java.io.StringReader
import au.com.bytecode.opencsv.CSVReader...
 val input = sc.textFile(inputFile)
 val result = input.map{ line =>
val reader = new CSVReader(new StringReader(line));
reader.readNext(); }
```

5.2.3 CSV/TSV

Example 5-15. Loading CSV in full in Python

```
def loadRecords(fileNameContents):
 """Load all the records in a given file"""
 input = StringIO.StringIO(fileNameContents[1])
 reader = csv.DictReader(input, fieldnames=["name",
"favoriteAnimal"]) return reader
fullFileData= sc.wholeTextFiles(inputFile).flatMap(loadRecords)
```

5.2.3 CSV/TSV

Example 5-16. Loading CSV in full in Scala

```
case class Person(name: String, favoriteAnimal: String)
val input = sc.wholeTextFiles(inputFile) val result = input.flatMap{
case (_, txt) =>
val reader = new CSVReader(new StringReader(txt));
reader.readAll().map(x => Person(x(0), x(1))) }
```

5.2.3 CSV/TSV

Example 5-18. Writing CSV in Python

```
def writeRecords(records):
 """Write out CSV lines"""
 output = StringIO.StringIO()
 writer = csv.DictWriter(output, fieldnames=["name",
"favoriteAnimal"]) for record in records:
writer.writerow(record) return [output.getvalue()]
pandaLovers.mapPartitions(writeRecords).saveAsTextFile(outputFil
e)
```

5.2.3 CSV/TSV

Example 5-19. Writing CSV in Scala

```
pandaLovers.map(person => List(person.name,
person.favoriteAnimal).toArray) .mapPartitions{people =>
val stringWriter = new StringWriter();
val csvWriter = new CSVWriter(stringWriter);
csvWriter.writeAll(people.toList) Iterator(stringWriter.toString)
}.saveAsTextFile(outFile)
```

5.2.4 SequenceFiles

- SequenceFiles are a popular Hadoop format composed of flat files with key/value pairs.
- SequenceFiles are a common input/output format for Hadoop MapReduce jobs.
- SequenceFiles consist of elements that implement Hadoop's Writable interface, as Hadoop uses a custom serialization framework.

5.2.4 SequenceFiles

Table 5-2. Corresponding Hadoop Writable types

Scala type	Java type	Hadoop Writable
`Int`	`Integer`	`IntWritable` or `VIntWritable`[2]
`Long`	`Long`	`LongWritable` or `VLongWritable`[2]
`Float`	`Float`	`FloatWritable`
`Double`	`Double`	`DoubleWritable`
`Boolean`	`Boolean`	`BooleanWritable`
`Array[Byte]`	`byte[]`	`BytesWritable`
`String`	`String`	`Text`
`Array[T]`	`T[]`	`ArrayWritable<TW>`[3]
`List[T]`	`List<T>`	`ArrayWritable<TW>`[3]
`Map[A, B]`	`Map<A, B>`	`MapWritable<AW, BW>`[3]

5.2.4 SequenceFiles

Example 5-20. Loading a SequenceFile in Python

```
val data = sc.sequenceFile(inFile,
 "org.apache.hadoop.io.Text",
"org.apache.hadoop.io.IntWritable")
```

Example 5-21. Loading a SequenceFile in Scala

```
val data = sc.sequenceFile(inFile, classOf[Text],
classOf[IntWritable]). map{case (x, y) => (x.toString,
y.get())}
```

5.2.4 SequenceFiles

Example 5-23. Saving a SequenceFile in Scala

```
val data = sc.parallelize(List(("Panda", 3), ("Kay", 6),
("Snail", 2))) data.saveAsSequenceFile(outputFile)
```

5.2.5 Object Files

- Object files are a deceptively simple wrapper around SequenceFiles that allows us to save our RDDs containing just values. Unlike with SequenceFiles, with object files the values are written out using Java Serialization.
- Using Java Serialization for object files has a number of implications. Unlike with normal SequenceFiles, the output will be different than Hadoop outputting the same objects. Unlike the other formats, object files are mostly intended to be used for Spark jobs communicating with other Spark jobs. Java Serialization can also be quite slow.

5.2.5 Object Files

- Saving an object file is as simple as calling saveAsObjectFile on an RDD. Reading an object file back is also quite simple: the function objectFile() on the SparkCon- text takes in a path and returns an RDD.
- Object files are not available in Python, but the Python RDDs and SparkContext support methods called saveAsPickleFile() and pickleFile() instead. These use Python's pickle serialization library.

5.2.6 Hadoop Input and Output Formats

Example 5-24. Loading KeyValueTextInputFormat() with old-style API in Scala

```
val input = sc.hadoopFile[Text, Text,
KeyValueTextInputFormat](inputFile).map{ case (x,
y) => (x.toString, y.toString)
}
```

5.2.6 Hadoop Input and Output Formats

Example 5-25. Loading LZO-compressed JSON with Elephant Bird in Scala

```
val input = sc.newAPIHadoopFile(inputFile,
classOf[LzoJsonInputFormat], classOf[LongWritable],
classOf[MapWritable], conf)
// Each MapWritable in "input" represents a JSON
object
```

5.2.7 File Compression

Table 5-3. Compression options

Format	Splittable	Average compression speed	Effectiveness on text	Hadoop compression codec	Pure Java	Native	Comments
gzip	N	Fast	High	`org.apache.hadoop.io.compress.GzipCodec`	Y	Y	
lzo	Y [6]	Very fast	Medium	`com.hadoop.compression.lzo.LzoCodec`	Y	Y	LZO requires installation on every worker node
bzip2	Y	Slow	Very high	`org.apache.hadoop.io.compress.BZip2Codec`	Y	Y	Uses pure Java for splittable version
zlib	N	Slow	Medium	`org.apache.hadoop.io.compress.DefaultCodec`	Y	Y	Default compression codec for Hadoop

5.2.7 File Compression

Format	Splittable	Average compression speed	Effectiveness on text	Hadoop compression codec	Pure Java	Native	Comments
Snappy	N	Very Fast	Low	`org.apache.hadoop.io.compress.SnappyCodec`	N	Y	There is a pure Java port of Snappy but it is not yet available in Spark/ Hadoop

5.3 File Systems

- Spark supports a large number of filesystems for reading and writing to, which we can use with any of the file formats we want.

5.3.1 Local/"Regular" FS

- While Spark supports loading files from the local filesystem, it requires that the files are available at the same path on all nodes in your cluster.

Example 5-29. Loading a compressed text file from the local filesystem in Scala

```
val rdd = sc.textFile
("file:///home/holden/happypandas.gz")
```

5.3.2 Amazon S3

- To access S3 in Spark, you should first set the AWS_ACCESS_KEY_ID and AWS_SECRET_ACCESS_KEY environment variables to your S3 credentials.
- Then pass a path starting with s3n:// to Spark's file input methods, of the form s3n://bucket/path-within-bucket.

5.3.3 HDFS

- Using Spark with HDFS is as simple as specifying hdfs://master:port/path for your input and output.

5.4 Structured data with Spark SQL

- Spark SQL supports multiple structured data sources as input, and because it understands their schema.
- We give Spark SQL a SQL query to run on the data source (selecting some fields or a function of the fields), and we get back an RDD of Row objects, one per record.

5.4.1 Apache Hive

- To connect Spark SQL to an existing Hive installation, you need to provide a Hive configuration. You do so by copying your *hive-site.xml* file to Spark's *./conf/* directory.

5.4.1 Apache Hive

Example 5-30. Creating a HiveContext and selecting data in Python

```
from pyspark.sql import HiveContext
hiveCtx = HiveContext(sc)
 rows = hiveCtx.sql("SELECT name, age FROM users")
firstRow = rows.first()
 print firstRow.name
```

5.4.1 Apache Hive

Example 5-31. Creating a HiveContext and selecting data in Scala

```
import org.apache.spark.sql.hive.HiveContext
val hiveCtx = new
org.apache.spark.sql.hive.HiveContext(sc) val rows =
hiveCtx.sql("SELECT name, age FROM users")
val firstRow =
rows.first()println(firstRow.getString(0)) // Field 0 is
the name
```

5.4.1 Apache Hive

Example 5-32. Creating a HiveContext and selecting data in Java

```
import org.apache.spark.sql.hive.HiveContext; import
org.apache.spark.sql.Row;
 import org.apache.spark.sql.SchemaRDD;
HiveContext hiveCtx = new HiveContext(sc);
 SchemaRDD rows = hiveCtx.sql("SELECT name, age
FROM users");
 Row firstRow = rows.first();
System.out.println(firstRow.getString(0)); // Field 0 is the
name
```

5.4.2 JSON

Example 5-33. Sample tweets in JSON

{"user": {" name": "Holden", "location": "San Francisco"}, "text": "Nice day out today"} {"user": {" name": "Matei", "location": "Berkeley"}, "text": "Even nicer here :)"}

5.4.2 JSON

Example 5-34. JSON loading with Spark SQL in Python

```
tweets = hiveCtx.jsonFile("tweets.json")
tweets.registerTempTable("tweets") results =
hiveCtx.sql("SELECT user.name, text FROM tweets")
```

5.4.2 JSON

Example 5-35. JSON loading with Spark SQL in Scala

```
val tweets = hiveCtx.jsonFile("tweets.json")
tweets.registerTempTable("tweets")val results =
hiveCtx.sql("SELECT user.name, text FROM tweets")
```

Example 5-36. JSON loading with Spark SQL in Java

```
SchemaRDD tweets = hiveCtx.jsonFile(jsonFile);
tweets.registerTempTable("tweets"); SchemaRDD results =
hiveCtx.sql("SELECT user.name, text FROM tweets");
```

5.5 Databases

- Spark can access several popular databases using either their Hadoop connectors or custom Spark connectors.

5.5.1 JDBC

Example 5-37. JdbcRDD in Scala

```
def createConnection() = {
Class.forName("com.mysql.jdbc.Driver").newInstance();
DriverManager.getConnection("jdbc:mysql://localhost/test?
user=holden");
}
def extractValues(r: ResultSet) = { (r.getInt(1), r.getString(2))
}
val data = new JdbcRDD(sc,
 createConnection, "SELECT * FROM panda WHERE ? <= id
AND id <= ?", lowerBound = 1, upperBound = 3, numPartitions
= 2, mapRow = extractValues)
println(data.collect().toList)
```

5.5.2 Cassandra

Example 5-38. sbt requirements for Cassandra connector

```
"com.datastax.spark" %% "spark-cassandra-connector" % "1.0.0-rc5", "com.datastax.spark" %% "spark-cassandra-connector-java" % "1.0.0-rc5"
```

Example 5-39. Maven requirements for Cassandra connector

```
<dependency> <!-- Cassandra -->
<groupId>com.datastax.spark</groupId> <artifactId>spark-cassandra-connector</artifactId> <version>1.0.0-rc5</version>
</dependency>
 <dependency> <!-- Cassandra -->
<groupId>com.datastax.spark</groupId> <artifactId>spark-cassandra-connector-java</artifactId> <version>1.0.0-rc5</version>
</dependency>
```

5.5.2 Cassandra

Example 5-40. Setting the Cassandra property in Scala

```
val conf = new SparkConf(true)
.set("spark.cassandra.connection.host ", "hostname")
val sc = new SparkContext(conf)
```

Example 5-41. Setting the Cassandra property in Java

```
SparkConf conf = new SparkConf(true)
.set("spark.cassandra.connection.host", cassandraHost);
JavaSparkContext sc = new JavaSparkContext(
sparkMaster, "basicquerycassandra", conf);
```

5.5.2 Cassandra

Example 5-42. Loading the entire table as an RDD with key/value data in Scala

```
// Implicits that add functions to the SparkContext & RDDs .
import com.datastax.spark.connector._
// Read entire table as an RDD .Assumes your table test was created as // CREATE TABLE test.kv(key text PRIMARY KEY, value int);val data = sc.cassandraTable("test", "kv")
 // Print some basic stats on the value field.
data.map(row => row.getInt("value")).stats()
```

5.5.2 Cassandra

Example 5-43. Loading the entire table as an RDD with key/value data in Java

```
import com.datastax.spark.connector.CassandraRow;
 import static
com.datastax.spark.connector.CassandraJavaUtil.javaFunctions;
// Read entire table as an RDD. Assumes your table test was created as
 // CREATE TABLE test.kv(key text PRIMARY KEY, value
int);JavaRDD<CassandraRow> data =
javaFunctions(sc).cassandraTable("test" , " kv"); // Print some basic stats.


System.out.println(data.mapToDouble(new
DoubleFunction<CassandraRow>() { public double call(CassandraRow row) {
return row.getInt("value"); }
}). stats());
```

5.5.2 Cassandra

Example 5-44. Saving to Cassandra in Scala

```
val rdd = sc.parallelize(List(Seq("moremagic", 1)))
rdd.saveToCassandra("test" , "kv", SomeColumns
("key", " value"))
```

5.5.3 HBase

Example 5-45. Scala example of reading from HBase

```
import org.apache.hadoop.hbase.HBaseConfiguration import
org.apache.hadoop.hbase.client.Result import
org.apache.hadoop.hbase.io.ImmutableBytesWritable import
org.apache.hadoop.hbase.mapreduce.TableInputFormat
val conf = HBaseConfiguration.create()
conf.set(TableInputFormat.INPUT_TABLE, "tablename") //
which table to scan
val rdd = sc.newAPIHadoopRDD(
 conf, classOf[TableInputFormat],
classOf[ImmutableBytesWritable], classOf[Result])
```

5.5.4 Elastic Search

Example 5-46. Elasticsearch output in Scala

```
val jobConf= new JobConf(sc.hadoopConfiguration)
jobConf.set("mapred.output.format.class",
"org.elasticsearch.hadoop.mr.EsOutputFormat")
jobConf.setOutputCommitter(classOf[FileOutputCommitte
r])
jobConf.set(ConfigurationOptions.ES_RESOURCE_WRIT
E, "twitter/tweets")
jobConf.set(ConfigurationOptions.ES_NODES,
"localhost") FileOutputFormat.setOutputPath(jobConf,
new Path("-")) output.saveAsHadoopDataset(jobConf)
```

5.5.4 Elastic Search

Example 5-47. Elasticsearch input in Scala

```
def mapWritableToInput(in: MapWritable): Map[String, String] = {
in.map{case (k, v) => (k.toString, v.toString)}.toMap
}
val jobConf = new JobConf(sc.hadoopConfiguration)
jobConf.set(ConfigurationOptions.ES_RESOURCE_READ, args(1))
jobConf.set(ConfigurationOptions.ES_NODES, args(2))
 val currentTweets = sc.hadoopRDD(jobConf,
classOf[EsInputFormat[Object, MapWritable]], classOf[Object],
classOf[MapWritable]) // Extract only the map

// Convert the MapWritable[Text, Text] to Map[String, String]
val tweets = currentTweets.map{ case (key, value) =>
mapWritableToInput(value) }
```