# Learning Spark

## 5. Loading and Saving your Data

손준영
(wiseosho@gmail.com)

# Contents

- File formats
- File systems
- Spark SQL
- Databases

# File Formats

| Format Name | Splittable | Structured | Comment |
|---|---|---|---|
| Text file | yes | no | Plain old text file, one / line |
| JSON | yes | semi | Common text based format, semi-structured, splittable if one record per line |
| CSV | yes | yes | Very common text based format, used with spreadsheet applications |
| Sequence files | y | y | A common Hadoop file format used for key-value data |
| Protocol buffers | y | y | A fast space-efficient multi-alnguage format |
| Object Files | y | y | Useful for saving data from a Spark job to be consumed by shared code.<br>Breaks if you change your classes, as it relies on Java Seerialization |

# File Formats

| Format Name | Splittable | Structured | Comment |
|---|---|---|---|
| Text file | yes | no | Plain old text file, one / line |
| JSON | yes | semi | Common text based format, semi-structured, splittable if one record per line |
| CSV | yes | yes | Very common text based format, used with spreadsheet applications |
| Sequence files | y | y | A common Hadoop file format used for key-value data |
| Protocol buffers | y | y | A fast space-efficient multi-alnguage format |
| Object Files | y | y | Useful for saving data from a Spark job to be consumed by shared code.<br>Breaks if you change your classes, as it relies on Java Seerialization |

# File Formats
# Text files

- **Load One text file**
  - Element / each line in file.
- **Load Multiple text file**
  - Pair RDD
  - Key : Filename / Value : Contents of file
-

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

# File Formats
# JSON

- **Load as Text and mapping value with JSON Parser**
  - Assuming 1 JSON / Row
  - Mulit line → Parse whole file content.
  - mapPartitions to reuse parser
- **JSON serialization library for saving Data as JSON format**
- **Used libraries:**
  - Jackson for Java and Scalar
  - json(inbuilt) for Python

# File Formats
# Load / Save JSON Data

- **Example 5-6. Loading unstructured JSON in Python**
-

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

**Example 5-9. Saving JSON in Python**

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

# File Formats
# Comma Seperated Values(CSV)

- A sample per line
- Fileds seperated by special character(, ;)
- No field names for each row rec.(←> JSON)
- Field name should be known before outputting

# File Formats
# Load CSV File in python

**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)
```

# File Formats
# Save into 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(outputFile)
```

# File Formats
# Sequence Files

- **Hadoop file system**
- **Flat files w/ (Key:Value) pairs**
- **Sync Markers**
  - Parallel IO from Multiple Nodes
- **Writable Interface.**
  - Hadoop custom serialization framework

# Writable Formats

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

# Load & Save Sequence File

**sequenceFile(path, keyClass, valueClass, minPartitions)**

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

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

```
saveAsSequenceFile(path)
```

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

# Hadoop File Format

- **Input and Output formats.**
  - Supply both New & Old api
  - NewAPIHadoopFile(filename, format, keyClass, valueClass, (conf))
  - HadoopFile()
  - ex) KeyValueTextInputFormat
    - reading in key/value data from text files

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

}
```

# Elephant Bird Package

- **Support # of Data format**
  - JSON, Lucene, Protocol Buffer
  - Work with all of Hadoop file API's
  - Ex) LZO-compressed JSON Data with Lzo JsonInputFormat

```
val input = sc.newAPIHadoopFile(inputFile,
classOf[LzoJsonInputFormat],

 classOf[LongWritable], classOf[MapWritable], conf)

// Each MapWritable in "input" represents a JSON object
```

# Save Hadoop File format

- SaveAsNewAPIHadoopFile
- ex) saving sequencefile in Java

```
JavaPairRDD<String, Integer> rdd =
sc.parallelizePairs(input);

JavaPairRDD<Text, IntWritable> result =
rdd.mapToPair(new ConvertToWritableTypes());

result.saveAsHadoopFile(fileName, Text.class,
IntWritable.class,

  SequenceFileOutputFormat.class);
```

# File Systems

- **Local/"Regular" FS**
  - requires that the files are available on all the nodes in your cluster.
  - Network file systems are usable
    - NFS, AFS, MapR's NFS layer
  - Recommended to use shared filesystem
    - HDFS, NFS, S3

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

# HDFS

- **Distributed filesystem**
- **Work on commodity hardware**
- **resilient to node failure**
- **High data throughput**
- **Spark & HDFS can be collocated on the same machines**
- **hdfs://master:port/path**
- **Version dependency between HDFS & Spark**
  - Default HDFS: 1.0.4

# Structured Data with Spark SQL

- **Have a schema**
  - Consistent set of fields across data records
  - Read only needed data from common sources
- **Apache Hive, JSON**

# Apache Hive

- Store tables in a variety of formats
- Copy *hive-sit.xml* into spark's *./conf/*

```
from pyspark.sql import HiveContext

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

# Load JSON by Spark SQL

- **Works for data with *consistent schema***
- **Use HiveContext.jsonFile**
  - Use the HiveContext.jsonFile method
  - Get RDD of **Row** objects for the whole file
  - RDD as a table and select specific fields from it.
- **Example Data Set**

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

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

# Database

- JavaDatabaseConnectivity(JDBC)
- Casandra
- Hbase
- ElasticSearch

# Hbase

```
import org.apache.hadoop.hbase.HbaseConfiguration
import org.apache.hadoop.hbase.client.Result
  - #Value Type
import org.apache.hadoop.hbase.io.ImmutableBytesWritable
  - # Key Type
import org.apache.hadoop.hbase.mapreduce.TableInputFormat
  - # Take Hbase and return Key/Value set

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])
```

# Elasticsearch-Output

- Ignores the path information depend on configuration

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

# Elasticsearch - Input

```
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) }
```

# Java Database Connectivity

- **Org.apache.spark.rdd.jdbcRDD**

```
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)
```

1. Establish a connection to database
2. provide query, set range of Bound(lowerBound, upperBound..)
3. convert each row data

# Cassandra

- **Use open source spark Cassandra connector ( from DataStax)**
- **CassandraRow object == Spark SQL Row object**
- **In Java & Scala only.**
-

# Requirements for Cassandra connector

- **Sbt & Maven requirements**

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

- **Maven requirements**

```
<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>
```

# Setting the Cassandra property

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

val sc = new SparkContext(conf)
```

```
Spark.cassandra.connection.host
Spark.cassandra.auth.username
Spark.cassandra.auth.password
```

# Load the entire table as RDD w/ key/value

```
// 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()
```

Thank you