Load and Save Data in Spark

(see also Spark DataFrames)

Nagiza F. Samatova, samatova@csc.ncsu.edu
Professor, Department of Computer Science
North Carolina State University

Senior Scientist, Computer Science & Mathematics Division
Oak Ridge National Laboratory

Loading and Saving Data

- **Odds are that your data doesn't fit on a single machine, so it's time to explore Spark's options for loading and saving**
- **Spark can access data through the InputFormat and OutputFormat interfaces used by Hadoop MapReduce:**
  - available for many common file formats and storage systems (e.g., S3, HDFS, Cassandra, HBase, etc.)
- **More commonly, though, you will want to use higher-level APIs built on top of these raw interfaces.**

Common Sets of Data Sources

- **File formats and filesystems**
  - For data stored in a local or distributed filesystem, such as NFS, HDFS, or Amazon S3, Spark can access a variety of file formats including text, JSON, SequenceFiles, and protocol buffers.
- **Structured data sources through Spark SQL**
  - The Spark SQL module provides a nicer and often more efficient API for structured data sources, including JSON and Apache Hive.
- **Databases and key/value stores**
  - Built-it and third-party libraries for connecting to Cassandra, HBase, Elasticsearch, and JDBC databases

File Formats

- Formats range from unstructured, like text, to semi-structured, like JSON, to structured, like SequenceFiles .
- The input formats that Spark wraps all transparently handle compressed formats based on the file extension.

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.

Loading Text Files with textFile()

- **When loading a *single text file* as an RDD, each input line becomes an element in the RDD:**
  - by calling the **textFile()** function on our SparkContext with the path to the file
  - To control the number of partitions, specify minPartitions

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

- **Loading *multiple text files* at the same time into a pair RDD, with the key being the name and the value being the contents of each file.**

Saving Text Files with saveAsTextFile()

- **saveAsTextFile(): outputs the content of the RDD to text file specified by its path**
  - The path is treated as a directory and Spark will output multiple files underneath that directory.
  - This allows Spark to write the output from multiple nodes.

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

- **With this method we don't get to control which files end up with which segments of our data, but there are other output formats that do allow this.**

Loading JSON File

- **JSON is a popular semi-structured data format.**
- **Load JSON data in one of two ways:**
  - load the data as a text file and then map over the values with a JSON parser
  - use JSON serialization library to write out the values to strings
- **Loading the data as a text file and then parsing the JSON data works assuming that you have <u>one</u> JSON record per row:**
  - if you have multi-line JSON files, you will instead have to load the whole file and then parse each file.
- **If constructing a JSON parser is expensive in your language, you can use mapPartitions() to reuse the parser;**

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

JSON Data with Consistent Schema

- **For JSON data with a consistent schema across records:**
  - Spark SQL can infer their schema and load this data as rows
- **To load JSON data, first create a HiveContext as when using Hive:**
  - No installation of Hive is needed in this case, though—that is, you don't need a hivesite.xml file.
  - Use the **HiveContext.jsonFile()** method to get an RDD of Row objects for the whole file.
  - Apart from using the whole Row object, you can also register this RDD as a table and select specific fields from it.
    - For example, suppose that we had a JSON file containing tweets, one per line:

```
{"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")
```

Saving JSON Files

- **Writing out JSON files is much simpler compared to loading it, because**
  - no need to worry about incorrectly formatted data and
  - the type of the data to be written out is known
- **Take an RDD of structured data, convert it into an RDD of strings, and then write strings with text file API.**
  - Let's say we were running a promotion for people who love pandas. We can take our input from the first step and filter it for the people who love pandas

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

Comma-Separated Values (CSV)

- Comma-separated value (CSV) files are supposed to contain a fixed number of fields per line, and the fields are separated by a comma (or a tab in the case of tab-separated value, or TSV, files).
- Records are often stored one per line, but this is not always the case as records can sometimes span multiple lines.
- CSV and TSV files can sometimes be inconsistent, most frequently with respect to handling newlines, escaping, and rendering non-ASCII characters, or non-integer numbers.
- CSVs cannot handle nested field types natively, so we have to unpack and pack to specific fields manually.

Loading CSV

- **Loading CSV/TSV data is similar to loading JSON data:**
  - first load it as text and then process it
- **The lack of standardization of format leads to different versions of the same library sometimes handling input in different ways.**
- **As with JSON, there are many different CSV libraries,**
  - in Python, we use the included **csv** library
- **If your CSV data happens to not contain newlines in any of the fields, you can load your data with textFile() and parse it**

```
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)
```

Loading CSV in full

- **If there are embedded newlines in fields, we will need to load each file in full and parse the entire segment.**
  - This is unfortunate because if each file is large it can introduce bottlenecks in loading and parsing.

```
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)
```

Saving CSV

- **Writing out CSV/TSV data is quite simple and we can benefit from reusing the output encoding object.**
- **Since in CSV we don't output the field name with each record, to have a consistent output we need to create a mapping:**
  - just write a function that converts the fields to given positions in an array
- **When outputting dictionaries, the CSV writer can do this for us based on the order in which we provide the field names when constructing the writer.**

```
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)
```

Acknowledgements

- **Anatoli Melechko, NCSU**
- **Learning Spark by Holden Karau, Andy Konwinski, Patrick Wendell, Matei Zaharia (http://shop.oreilly.com/product/0636920028512.do)**