Apache Spark

Session 6 - Loading & Saving Data

WELCOME - KNOWBIGDATA

- ❏ Expert Instructors
- ❏ CloudLabs
- ❏ Lifetime access to LMS
- ❏ Presentations
- ❏ Class Recording
- ❏ Assignments + Quizzes
- ❏ Project Work
- ❏ Real Life Project
- ❏ Course Completion Certificate
- ❏ 24x7 support
- ❏ KnowBigData - Alumni
  - ❏ Jobs
  - ❏ Stay Abreast (Updated Content, Complimentary Sessions)
  - ❏ Stay Connected

COURSE CONTENT

I	Introduction to Big Data with Apache Spark
II	Downloading Spark and Getting Started
III	Programming with RDDs
IV	Working with Key/Value Pairs
V	Loading and Saving Your Data
VI	Advanced Spark Programming
VII	Running on a Cluster
VIII	Tuning and Debugging Spark
IX	Spark SQL, SparkR
X	Spark Streaming
XI	Machine Learning with MLlib, GraphX

About Instructor?

2014	**KnowBigData**	Founded
2014 2012	**Amazon**	Built High Throughput Systems for Amazon.com site using in-house NoSql.
2012	**InMobi**	Built Recommender that churns 200 TB
2011 2006 2002	**tBits Global**	Founded tBits Global Built an enterprise grade Document Management System
	D.E.Shaw	Built the big data systems before the term was coined
2002	**IIT Roorkee**	Finished B.Tech.

Loading & Saving Data

1. So far we either converted in-memory data
2. Or used the HDFS file
3. Spark supports wide variety of dataset
4. Can access data through InputFormat & OutputFormat
    a. The interfaces used by Hadoop
    b. Which are available for many common file formats and storage systems (e.g., S3, HDFS, Cassandra, HBase, etc.).

Common Data Sources

File formats

+ Text, JSON, SequenceFiles, Protocol buffers.
+ We can also configure compression

Filesystems

+ Local, NFS, HDFS, Amazon S3

Structured data sources through Spark SQL

+ Efficient API for structured data sources, including JSON and Apache Hive
+ Covered later

Databases and key/value stores

+ Built-in and third-party libraries
+ For Cassandra, HBase, Elasticsearch, and JDBC databases.

Common supported file formats

Format Name	Comments
Text files	Plain old text files. Records are assumed to be one per line.
JSON	Common text-based format, semistructured; most libraries require one record per line.
CSV	Very common text-based format, often used with spreadsheet applications.
SequenceFiles	A common Hadoop file format used for key/value data.
Protocol buffers	A fast, space-efficient multilanguage format.
Object files	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.

Handling Text Files

Loading Files

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

Loading Directories

```
input = sc.wholeTextFiles("/home/student/sgiri/ml-100k");
lengths = input.mapValues(lambda x: len(x));
lengths.collect();
```

```
[(u'file:/home/student/sgiri/ml-100k/u1.test', 392629), (u'file:/home/student/sgiri/ml-100k/u.genre', 202), ...]
```

Saving Files

```
result.saveAsTextFile(outputDir)
```

Handling JSON Files

Loading JSON:

+ Load the data as a text file and then map over the values with a JSON
+ Also, we write as text file.
+ We can also load JSON in Spark SQL, we will discuss that later.
+ You can use per partition processing too

Loading JSON Files

1. Download data from: http://jsonstudio.com/wp-content/uploads/2014/02/stocks.zip
2. Unzip it
3. Using tail or head -1 observe the format
4. If needed test it with the interactive shell

```
import json
input = sc.textFile("stocks.json")
def f(x):
    js = json.loads(x);
    return js["Ticker"]

data = input.map(f)
data.collect()
    [u'A', u'AA', u'AADR', u'AAIT', u'AAMC', u'AAME', u'AAN', ... ]
```

Saving JSON Files

```
outJson = data.map(lambda x: json.dumps(x))
data.saveAsTextFile("myoutputdir/");

# Check if the directory is created
ls -l /home/student/sgiri/myoutputdir/
```

Comma / Tab -Separated Values (CSV / TSV)

1. Records are often stored one per line,
2. Fixed number of fields per line
3. Fields are separated by a comma (tab in TSV)
4. We get row number to detect header etc.

Loading CSV

```
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)
```

Not good for the situations where record contains newline. For that load in full

Loading CSV in full in Python

```
import csv
import StringIO

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(inputDir).flatMap(loadRecords)
```

Not good if the files is huge that don't fit in memory

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(outputFile)
```

Tab Separated Files

```
reader = csv.DictReader(input, fieldnames=["name", "favoriteAnimal"],
dialect="excel-tab")
```

```
writer = csv.DictWriter(output, fieldnames=["name", "favoriteAnimal"],
dialect="excel-tab")
```

SequenceFiles

- Popular Hadoop format
- Composed of flat files with key/value pairs.
- Has Sync markers
  - Allow to seek to a point
  - Then resynchronize with the record boundaries
  - Allows Spark to efficiently read in parallel from multiple nodes

Loading SequenceFiles

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

Saving SequenceFiles

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

Loading/Saving SequenceFiles - Example

```
>>> rdd = sc.parallelize([('key1', 1.0), ('key2', 2.0), ('key3', 3.0)])
>>> rdd.saveAsSequenceFile('/tmp/pysequencefile1/')

>>> f = sc.sequenceFile('/tmp/pysequencefile1/')
>>> f.collect()
[(u'key2', 2.0), (u'key3', 3.0), (u'key1', 1.0)]
```

Object Files

- Simple wrapper around SequenceFiles
- Values are written out using Java Serialization.
- Intended to be used for Spark jobs communicating with other Spark jobs
- Can also be quite slow.
- Saving - saveAsObjectFile() on an RDD
- Loading - objectFile() on SparkContext
- Require almost no work to save almost arbitrary objects.
- Not available in python using pickle file instead
- If you change the objects, old files may not be valid

Pickle File

- Python way of handling object files
- Uses Python's pickle serialization library
- Saving - saveAsPickleFile() on an RDD
- Loading - pickleFile() on SparkContext
- Can also be quite slow as Object Fiels

pickleFile Example

pickleFile(name, minPartitions=None)[source]

Load an RDD previously saved using RDD.saveAsPickleFile method.

```
>>> tmpFile = NamedTemporaryFile(delete=True)
>>> tmpFile.close()
>>> sc.parallelize(range(10)).saveAsPickleFile(tmpFile.name, 5)
>>> sorted(sc.pickleFile(tmpFile.name, 3).collect())
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

Non-filesystem data sources

- Access Hadoop-supported storage formats
- Many key/value stores provide Hadoop input formats
- Example providers:HBase, MongoDB
- Takes a Configuration object on which you set the Hadoop properties
- Older: hadoopFile() / saveAsHadoopFile()
- Newer: newAPIHadoopDataset() / saveAsNewAPIHadoopDataset()

Hadoop Input and Output Formats - Old API

hadoopFile(path, inputFormatClass, keyClass, valueClass, keyConverter=None, valueConverter=None, conf=None, batchSize=0)

Read an 'old' Hadoop InputFormat with arbitrary key and value class from HDFS, a local file system (available on all nodes), or any Hadoop-supported file system URI. The mechanism is the same as for sc.sequenceFile.

A Hadoop configuration can be passed in as a Python dict. This will be converted into a Configuration in Java.

Parameters:

path – path to Hadoop file
inputFormatClass – fully qualified classname of Hadoop InputFormat (e.g. "org.apache.hadoop.mapred.TextInputFormat")
keyClass – fully qualified classname of key Writable class (e.g. "org.apache.hadoop.io.Text")
valueClass – fully qualified classname of value Writable class (e.g. "org.apache.hadoop.io.LongWritable")
keyConverter – (None by default)
valueConverter – (None by default)
conf – Hadoop configuration, passed in as a dict (None by default)
batchSize – The number of Python objects represented as a single Java object. (default 0, choose batchSize automatically)

Hadoop Input and Output Formats - New API

newAPIHadoopFile(path, inputFormatClass, keyClass, valueClass, keyConverter=None, valueConverter=None, conf=None, batchSize=0)

Read a 'new API' Hadoop InputFormat with arbitrary key and value class from HDFS, a local file system (available on all nodes), or any Hadoop-supported file system URI. The mechanism is the same as for sc.sequenceFile.

A Hadoop configuration can be passed in as a Python dict. This will be converted into a Configuration in Java

Parameters:

path – path to Hadoop file
inputFormatClass – fully qualified classname of Hadoop InputFormat (e.g. "org.apache.hadoop.mapreduce.lib.input.TextInputFormat")
keyClass – fully qualified classname of key Writable class (e.g. "org.apache.hadoop.io.Text")
valueClass – fully qualified classname of value Writable class (e.g. "org.apache.hadoop.io.LongWritable")
keyConverter – (None by default)
valueConverter – (None by default)
conf – Hadoop configuration, passed in as a dict (None by default)
batchSize – The number of Python objects represented as a single Java object. (default 0, choose batchSize automatically)

Protocol buffers

- Developed at Google for internal RPCs
- Open sourced
- Structured data - fields & types of fields defined
- Fast for encoding and decoding (20-100x than XML)
- Take up the minimum space (3-10x than xml)
- Defined using a domain-specific language
- Compiler generates accessor methods in variety of languages
- Consist of fields: optional, required, or repeated
- While parsing
  - A missing optional field => success
  - A missing required field => failure
- So, make new fields as optional (remember object file failures?)

Protocol buffers - Example

```
package tutorial;
message Person {
  required string name = 1;
  required int32 id = 2;
  optional string email = 3;
  enum PhoneType {
    MOBILE = 0;
    HOME = 1;
    WORK = 2;
  }
  message PhoneNumber {
    required string number = 1;
    optional PhoneType type = 2 [default = HOME];
  }
  repeated PhoneNumber phone = 4;
}

message AddressBook {
  repeated Person person = 1;
}
```

Protocol buffers - Steps

1. Download and install protocol buffer compiler
2. pip install protobuf
3. protoc -I=$SRC_DIR --python_out=$DST_DIR $SRC_DIR/addressbook.proto
4. create objects
5. Convert those into protocol buffers
6. See this project

File Compression

1. To Save Storage & Network Overhead
2. With most hadoop output formats we can specify compression codecs
3. Compression should not require the whole file at once
4. Each worker can find start of record => splitable

File Compression Options

Format	Splittable	Speed	Effectiveness on text	Hadoop compression codec	comments
gzip	N	Fast	High	org.apache.hadoop.io.com.GzipCodec	
lzo	*Y*	*V. Fast*	*Medium*	*com.hadoop.compression.lzo.LzoCodec*	*LZO requires installation on every worker node*
bzip2	*Y*	*Slow*	*V. High*	*org.apache.hadoop.io.com.BZip2Codec*	*Uses pure Java for splittable version*
zlib	N	Slow	Medium	org.apache.hadoop.io.com.DefaultCodec	Default compression codec for Hadoop
Snappy	N	V. Fast	Low	org.apache.hadoop.io.com.SnappyCodec	There is a pure Java port of Snappy but it is not yet available in Spark/ Hadoop

Loading + Saving Data: File Systems

Local/“Regular” FS

1. rdd = sc.textFile("**file**:///home/holden/happypandas.gz")
2. The path has to be available on all nodes.
   Otherwise, load it locally and distribute using sc.parallelize

Amazon S3

1. Popular option
2. Good if nodes are inside EC2
3. Use path in all input methods (textFile, hadoopFile etc)
   s3n://bucket/path-within-bucket
4. Set Env. Vars: AWS_ACCESS_KEY_ID AWS_SECRET_ACCESS_KEY

HDFS

1. The Hadoop Distributed File System
2. Spark and HDFS can be collocated on the same machines
3. Spark can take advantage of this data locality to avoid network overhead
4. In all i/o methods, use path: hdfs://master:port/path
5. Use only the version of spark w.r.t HDFS version

Structured Data with Spark SQL

1. For data that has schema
2. Consistent set of field for all records
3. Spark SQL run SQL query on Data Source &
4. Gives back an RDD of Row objects, one per record.
5. In java, row.getXXX() Methods
6. In python, row["XXX"]

Apache Hive

- Common structured data source on Hadoop
- Can store tables in a variety of formats inside HDFS or other storages
- In plain text to column-oriented formats

Steps:

1. To connect, copy hive-site.xml to Spark's ./conf/
2. Create HiveContext() object as entry point
3. Write query using HQL

Apache Hive - Example

```
cp /etc/hive/conf/hive-site.xml spark/conf
spark/bin/pyspark
```

```
from pyspark.sql import HiveContext
hiveCtx = HiveContext(sc)

#see table at
# http://hadoop1.knowbigdata.com:8000/beeswax/table/default/employee
rows = sqlContext.sql("SELECT name,sal FROM employee where
name='sandeep'")
firstRow = rows.first()
print firstRow.name
rowsLocal = rows.collect()
for row in rowsLocal:
...      print row.name
...
```

JSON

```
{"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")
```

JSON (example 2 from HDFS JSON)

```
hiveCtx = HiveContext(sc)
stocks = hiveCtx.jsonFile("hdfs://hadoop1.knowbigdata.com/data/spark/stocks.json")
stocks.registerTempTable("stocks")
results = hiveCtx.sql('select Ticker from stocks')
```

Databases

#Database Querying
spark/bin/pyspark --driver-library-path 'mysql-connector-java-5.1.36-bin.jar' --driver-class-path 'mysql-connector-java-5.1.36-bin.jar'

```
from pyspark.sql import SQLContext
sqlctx = SQLContext(sc)
df = sqlctx.load(
        source='jdbc',
        driver='com.mysql.jdbc.Driver',
        url='jdbc:mysql://hadoop1.knowbigdata.com/test?user=root&password=',
        dbtable='sales')
df.collect()
```

NOTE: This has been replaced by dataframe. We will discuss those later.

Apache Spark

Thank you.

+1 419 665 3276 (US)
+91 803 959 1464 (IN)

reachus@knowbigdata.com

Subscribe to our Youtube channel for latest videos - https://www.youtube.com/channel/UCxugRFe5wETYA7nMH6VGyEA