E6893 Big Data Analytics Lecture 6:

Spark and Data Analytics

Ching-Yung Lin, Ph.D.
Adjunct Professor, Dept. of Electrical Engineering and Computer Science
IBM Distinguished Researcher and Chief Scientist, Graph Computing

October 13th, 2016
Homework #2 revision (due October 24th)

1. Recommendation:
   1-1. Choose any two datasets you can get from any public data set. (Example, you can see whether you can use data from Yahoo Labs Ratings and Classification Data, or others.)
   1-2. Try various recommendation algorithms provided by Mahout

2. Clustering:
   Using datasets from:
   1. Online news (e.g., New York Times article in September 2016, or other data sources)
   2. Wikipedia articles
   3. (optional) gather data from Twitter API, try clustering
   Do clustering —> finding related documents

3. Classification:
   3-1: Using the 20 newsgroups data (will be provided by TA), try various classification algorithms provided by Mahout, and discuss their performance
   3-2: Do similar experiments on the Wikipedia data that you downloaded.

4. Install Spark. Run simple word count algorithm.

* Two changes: (1) The submission is delayed to October 24th; (2) You can use either Mahout or Spark MLlib to accomplish Assignment 1, 2, and 3.
(Some potential) Projects

1. **Price checker** (contact: Dr. Jie Lu — jielu@us.ibm.com)

   Problem:
   Given the description of a product, determine the reasonable price range of this kind of product based on information sources available from the Internet. The description may vary in granularity. It may have details such as particular brand/model, or may be very brief/general/vague, e.g. "smart phone".

2. **Robot vision** (contact Dr. Guangnan Ye — gye@us.ibm.com)

   Problem:
   Improving computer vision and speech recognition capability on robots — Nao, Pepper, etc.

3. **Mobile vision** (contact Dr. Larry Lai — larrylai@us.ibm.com)

   Problem:
   OCR, Face Analysis, and Object Recognition, etc, on iOS platform

4. **Distributed graph analytics** (contact Dr. Toyotaro Suzumura — tsuzumura@us.ibm.com)

   Problem:
   Linked big data analysis using distributed graph middleware

5. **Big data visualization** (contact Dr. Conglei Shi—shiconglei@us.ibm.com)

   Problem:
   Explore novel layout and visual analytics technologies on big data
Learning Spark

LIGHTNING-FAST DATA ANALYSIS

Holden Karau, Andy Konwinski, Patrick Wendell & Matei Zaharia
Spark Stack

Spark SQL
Structured data

Spark Streaming
Real-time

MLib
Machine learning

GraphX
Graph processing

Spark Core

Standalone Scheduler

YARN

Mesos
Spark Core

Basic functionality of Spark, including components for:

- Task Scheduling
- Memory Management
- Fault Recovery
- Interacting with Storage Systems
- and more

Home to the API that defines resilient distributed datasets (RDDs) - Spark’s main programming abstraction.

RDD represents a collection of items distributed across many compute nodes that can be manipulated in parallel.
Download Spark

http://spark.apache.org/downloads.html

Download Spark

The latest release of Spark is Spark 1.2.0, released on December 18, 2014 (release notes) (git tag)

1. Chose a Spark release: 1.2.0 (Dec 18 2014)
2. Chose a package type: Pre-built for Hadoop 2.4 and later
3. Chose a download type: Direct Download
4. Download Spark: spark-1.2.0-bin-hadoop2.4.tgz
5. Verify this release using the 1.2.0 signatures and checksums.

Note: Scala 2.11 users should download the Spark source package and build with Scala 2.11 support.

Link with Spark

Spark artifacts are hosted in Maven Central. You can add a Maven dependency with the following coordinates:

groupId: org.apache.spark
artifactId: spark-core_2.10
version: 1.2.0

Latest News

Spark Summit East agenda posted, CFP open for West (Jan 21, 2015)
Spark 1.2.0 released (Dec 18, 2014)
Spark 1.1.1 released (Nov 28, 2014)
Registration open for Spark Summit East 2015 (Nov 26, 2014)

Archive

Download Spark

Built-in Libraries:

- Spark SQL
- Spark Streaming
- MLlib (machine learning)
- GraphX (graph)
First language to use — Python

Python is a programming language that lets you work quickly and integrate systems more effectively. **Learn More**
Spark’s Python Shell (PySpark Shell)

bin/pyspark

```
bin/pyspark
```

Welcome to

```
Using Python version 2.7.6 (default, Mar 22 2014 22:59:56)
SparkContext available as sc.
```
Test installation

*Example 2-1. Python line count*

```python
>>> lines = sc.textFile("README.md") # Create an RDD called lines

>>> lines.count() # Count the number of items in this RDD
127

>>> lines.first() # First item in this RDD, i.e. first line of README.md
u'# Apache Spark'
```
Disable logging

You may find the logging statements that get printed in the shell distracting. You can control the verbosity of the logging. To do this, you can create a file in the `conf` directory called `log4j.properties`. The Spark developers already include a template for this file called `log4j.properties.template`. To make the logging less verbose, make a copy of `conf/log4j.properties.template` called `conf/log4j.properties` and find the following line:

```java
log4j.rootCategory=INFO, console
```

Then lower the log level so that we show only the WARN messages, and above by changing it to the following:

```java
log4j.rootCategory=WARN, console
```
Core Spark Concepts

- At a high level, every Spark application consists of a driver program that launches various parallel operations on a cluster.

- The driver program contains your application’s main function and defines distributed databases on the cluster, then applies operations to them.

- In the preceding example, the driver program was the Spark shell itself.

- Driver programs access Spark through a SparkContext object, which represents a connection to a computing cluster.

- In the shell, a SparkContext is automatically created as the variable called sc.
Driver Programs

Driver programs typically manage a number of nodes called executors.

If we run the count() operation on a cluster, different machines might count lines in different ranges of the file.
Example filtering

```python
>>> lines = sc.textFile("README.md")

>>> pythonLines = lines.filter(lambda line: "Python" in line)

>>> pythonLines.first()

u'## Interactive Python Shell'
```

lambda —> define functions inline in Python.

```python
def hasPython(line):
    return "Python" in line

pythonLines = lines.filter(hasPython)
```
Running as a Standalone Application

In Python, you simply write applications as Python scripts, but you must run them using the `bin/spark-submit` script included in Spark. The `spark-submit` script includes the Spark dependencies for us in Python. This script sets up the environment for Spark’s Python API to function. Simply run your script with the line given in Example 2-6.

Example 2-6. Running a Python script

`bin/spark-submit my_script.py`
Example — word count

```python
import sys
from operator import add

from pyspark import SparkContext

if __name__ == "__main__":
    if len(sys.argv) != 2:
        print >> sys.stderr, "Usage: wordcount <file>"
        exit(-1)
    sc = SparkContext(appName="PythonWordCount")
    lines = sc.textFile(sys.argv[1], 1)
    counts = lines.flatMap(lambda x: x.split(' '))
        .map(lambda x: (x, 1))
        .reduceByKey(add)
    output = counts.collect()
    for (word, count) in output:
        print "%s: %i" % (word, count)

sc.stop()
```
Resilient Distributed Dataset (RDD) Basics

• An RDD in Spark is an immutable distributed collection of objects.

• Each RDD is split into multiple partitions, which may be computed on different nodes of the cluster.

• Users create RDDs in two ways: by loading an external dataset, or by distributing a collection of objects in their driver program.

• Once created, RDDs offer two types of operations: transformations and actions.

```python
>>> lines = sc.textFile("README.md")  # <= create RDD

>>> pythonLines = lines.filter(lambda line: "Python" in line)  # <= transformation

>>> pythonLines.first()  # <= action
u'## Interactive Python Shell'
```

Transformations and actions are different because of the way Spark computes RDDs. Only computes when something is, the first time, in an action.
Persistance in Spark

- By default, RDDs are computed each time you run an action on them.
- If you like to reuse an RDD in multiple actions, you can ask Spark to persist it using RDD.persist().
- RDD.persist() will then store the RDD contents in memory and reuse them in future actions.
- Persisting RDDs on disk instead of memory is also possible.
- The behavior of not persisting by default seems to be unusual, but it makes sense for big data.

```python
Example 3-4. Persisting an RDD in memory

>>> pythonlines.persist

>>> pythonlines.count()
2

>>> pythonlines.first()
'## Interactive Python Shell'
```

To summarize, every Spark program and shell session will work as follows:

1. Create some input RDDs from external data.
2. Transform them to define new RDDs using transformations like filter().
3. Ask Spark to persist() any intermediate RDDs that will need to be reused.
4. Launch actions such as count() and first() to kick off a parallel computation, which is then optimized and executed by Spark.
Parallelizing in Spark

The simplest way to create RDDs is to take an existing collection in your program and pass it to SparkContext’s parallelize() method. But, that needs all dataset in memory on one machine.

Example 3-5. parallelize() method in Python

```python
lines = sc.parallelize(['pandas', 'i like pandas'])
```
# Spark file loading

## Table 5-1. Common supported file formats

<table>
<thead>
<tr>
<th>Format name</th>
<th>Structured</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>Text files</td>
<td>No</td>
<td>Plain old text files. Records are assumed to be one per line.</td>
</tr>
<tr>
<td>JSON</td>
<td>Semi</td>
<td>Common text-based format, semistructured; most libraries require one record per line.</td>
</tr>
<tr>
<td>CSV</td>
<td>Yes</td>
<td>Very common text-based format, often used with spreadsheet applications.</td>
</tr>
<tr>
<td>SequenceFiles</td>
<td>Yes</td>
<td>A common Hadoop file format used for key/value data.</td>
</tr>
<tr>
<td>Protocol buffers</td>
<td>Yes</td>
<td>A fast, space-efficient multilanguage format.</td>
</tr>
<tr>
<td>Object files</td>
<td>Yes</td>
<td>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.</td>
</tr>
</tbody>
</table>
Loading and Saving

Example 5-1. Loading a text file in Python

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

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

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

Example 5-6. Loading unstructured JSON in Python

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

Example 5-9. Saving JSON in Python

```python
(data.filter(lambda x: x['lovesPandas']).map(lambda x: json.dumps(x))
 .saveAsTextFile(outputFile))
```
Loading and Saving (II)

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

```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)
```

Example 5-18. Writing CSV in Python

```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)
```
Loading and Saving (III)

Example 5-20. Loading a SequenceFile in Python

```scala
val data = sc.sequenceFile(inFile,
```
### File compression options

<table>
<thead>
<tr>
<th>Format</th>
<th>Splittable</th>
<th>Average compression speed</th>
<th>Effectiveness on text</th>
<th>Hadoop compression codec</th>
<th>Pure Java</th>
<th>Native</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>gzip</td>
<td>N</td>
<td>Fast</td>
<td>High</td>
<td>org.apache.hadoop.io.compress.GzipCodec</td>
<td>Y</td>
<td>Y</td>
<td></td>
</tr>
<tr>
<td>lzo</td>
<td>Y&lt;sup&gt;6&lt;/sup&gt;</td>
<td>Very fast</td>
<td>Medium</td>
<td>com.hadoop.compression.lzo.LzoCodec</td>
<td>Y</td>
<td>Y</td>
<td>LZ0 requires installation on every worker node</td>
</tr>
<tr>
<td>bzip2</td>
<td>Y</td>
<td>Slow</td>
<td>Very high</td>
<td>org.apache.hadoop.io.compress.BZip2Codec</td>
<td>Y</td>
<td>Y</td>
<td>Uses pure Java for splittable version</td>
</tr>
<tr>
<td>zlib</td>
<td>N</td>
<td>Slow</td>
<td>Medium</td>
<td>org.apache.hadoop.io.compress.DefaultCodec</td>
<td>Y</td>
<td>Y</td>
<td>Default compression codec for Hadoop</td>
</tr>
<tr>
<td>Snappy</td>
<td>N</td>
<td>Very Fast</td>
<td>Low</td>
<td>org.apache.hadoop.io.compress.SnappyCodec</td>
<td>N</td>
<td>Y</td>
<td>There is a pure Java port of Snappy but it is not yet available in Spark/Hadoop</td>
</tr>
</tbody>
</table>
Example of Spark Programming

**Example 6-1. Sample call log entry in JSON, with some fields removed**

```
{"address":"address here","band":"40m","callsign":"KK6JLK","city":"SUNNYVALE","contactlat":"37.384733","contactlng":"-122.032164","county":"Santa Clara","dxcc":"291","fullname":"MATTHEW McPherrin","id":57779,"mode":"FM","mylat":"37.751952821","mylong":"-122.4208688735",...}
```

**Example 6-2. Accumulator empty line count in Python**

```python
def extractCallSigns(line):
    global blankLines  # Make the global variable accessible
    if (line == ""):
        blankLines += 1
    return line.split(" ")

file = sc.textFile(inputFile)
# Create Accumulator[Int] initialized to 0
blankLines = sc.accumulator(0)

callSigns = file.flatMap(extractCallSigns)
callSigns.saveAsTextFile(outputDir + "/callsigns")
print "Blank lines: %d" % blankLines.value
```
Table 6-2. Summary statistics available from StatsCounter

<table>
<thead>
<tr>
<th>Method</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>count()</td>
<td>Number of elements in the RDD</td>
</tr>
<tr>
<td>mean()</td>
<td>Average of the elements</td>
</tr>
<tr>
<td>sum()</td>
<td>Total</td>
</tr>
<tr>
<td>max()</td>
<td>Maximum value</td>
</tr>
<tr>
<td>min()</td>
<td>Minimum value</td>
</tr>
<tr>
<td>variance()</td>
<td>Variance of the elements</td>
</tr>
<tr>
<td>sampleVariance()</td>
<td>Variance of the elements, computed for a sample</td>
</tr>
<tr>
<td>stdev()</td>
<td>Standard deviation</td>
</tr>
<tr>
<td>sampleStdev()</td>
<td>Sample standard deviation</td>
</tr>
</tbody>
</table>
Removing outliers

**Example 6-19. Removing outliers in Python**

```python
# Convert our RDD of strings to numeric data so we can compute stats and
# remove the outliers.
distanceNumerics = distances.map(lambda string: float(string))
stats = distanceNumerics.stats()
stderr = std.stdev()
mean = stats.mean()
reasonableDistances = distanceNumerics.filter(
    lambda x: math.fabs(x - mean) < 3 * stderr)
print reasonableDistances.collect()
```
Machine Learning Library in Spark — MLlib

An example of using MLlib for text classification task, e.g., identifying spammy emails.

1. Start with an RDD of strings representing your messages.
2. Run one of MLlib’s feature extraction algorithms to convert text into numerical features (suitable for learning algorithms); this will give back an RDD of vectors.
3. Call a classification algorithm (e.g., logistic regression) on the RDD of vectors; this will give back a model object that can be used to classify new points.
4. Evaluate the model on a test dataset using one of MLlib’s evaluation functions.
Example: Spam Detection

Example 11-1. Spam classifier in Python

```python
from pyspark.mllib.regression import LabeledPoint
from pyspark.mllib.feature import HashingTF
from pyspark.mllib.classification import LogisticRegressionWithSGD

spam = sc.textFile("spam.txt")
normal = sc.textFile("normal.txt")

# Create a HashingTF instance to map email text to vectors of 10,000 features.
tf = HashingTF(numFeatures = 10000)
# Each email is split into words, and each word is mapped to one feature.
spamFeatures = spam.map(lambda email: tf.transform(email.split(" ")))
normalFeatures = normal.map(lambda email: tf.transform(email.split(" ")))

# Create LabeledPoint datasets for positive (spam) and negative (normal) examples.
positiveExamples = spamFeatures.map(lambda features: LabeledPoint(1, features))
negativeExamples = normalFeatures.map(lambda features: LabeledPoint(0, features))
trainingData = positiveExamples.union(negativeExamples)
trainingData.cache() # Cache since Logistic Regression is an iterative algorithm.

# Run Logistic Regression using the SGD algorithm.
model = LogisticRegressionWithSGD.train(trainingData)

# Test on a positive example (spam) and a negative one (normal). We first apply
# the same HashingTF feature transformation to get vectors, then apply the model.
posTest = tf.transform("O M G GET cheap stuff by sending money to ...".split(" "))
negTest = tf.transform("Hi Dad, I started studying Spark the other ...".split(" "))
print "Prediction for positive test example: %g" % model.predict(posTest)
print "Prediction for negative test example: %g" % model.predict(negTest)
```
Feature Extraction Example — TF-IDF

Example 11-7. Using HashingTF in Python

```python
>>> from pyspark.mllib.feature import HashingTF

>>> sentence = "hello hello world"
>>> words = sentence.split()  # Split sentence into a list of terms
>>> tf = HashingTF(10000)  # Create vectors of size S = 10,000
>>> tf.transform(words)
SparseVector(10000, {3065: 1.0, 6861: 2.0})

>>> rdd = sc.wholeTextFiles("data").map(lambda (name, text): text.split())
>>> tfVectors = tf.transform(rdd)  # Transforms an entire RDD
```

Example 11-8. Using TF-IDF in Python

```python
from pyspark.mllib.feature import HashingTF, IDF

# Read a set of text files as TF vectors
rdd = sc.wholeTextFiles("data").map(lambda (name, text): text.split())
tf = HashingTF()
tfVectors = tf.transform(rdd).cache()

# Compute the IDF, then the TF-IDF vectors
idf = IDF()
idfModel = idf.fit(tfVectors)
tfIdfVectors = idfModel.transform(tfVectors)
```
Linear Regression

Example 11-10. Linear regression in Python

```python
from pyspark.mllib.regression import LabeledPoint
from pyspark.mllib.regression import LinearRegressionWithSGD

points = # (create RDD of LabeledPoint)
model = LinearRegressionWithSGD.train(points, iterations=200, intercept=True)
print "weights: %s, intercept: %s" % (model.weights, model.intercept)
```
Classifiers

**Logistic regression**

Logistic regression is a binary classification method that identifies a linear separating plane between positive and negative examples. In MLlib, it takes LabeledPoints with label 0 or 1 and returns a LogisticRegressionModel that can predict new points.

**Support Vector Machines**

Support Vector Machines, or SVMs, are another binary classification method with linear separating planes, again expecting labels of 0 or 1. They are available through the SVMWithSGD class, with similar parameters to linear and logistic regression. The returned SVMModel uses a threshold for prediction like LogisticRegressionModel.

**Naive Bayes**

Naive Bayes is a multiclass classification algorithm that scores how well each point belongs in each class based on a linear function of the features. It is commonly used in text classification with TF-IDF features, among other applications. MLlib implements Multinomial Naive Bayes, which expects nonnegative frequencies (e.g., word frequencies) as input features.
Decision trees and Random Forests

Figure 11-2. An example decision tree predicting whether a user might buy a product

In MLlib, you can train trees using the \texttt{mllib.tree.DecisionTree} class, through the static methods \texttt{trainClassifier()} and \texttt{trainRegressor()}. 
Clustering

**K-means**

MLlib includes the popular K-means algorithm for clustering, as well as a variant called K-means|| that provides better initialization in parallel environments. K-means|| is similar to the K-means++ initialization procedure often used in single-node settings.

```java
initializationMode
   The method to initialize cluster centers, which can be either “k-means||” or “random”; k-means|| (the default) generally leads to better results but is slightly more expensive.

maxIterations
   Maximum number of iterations to run (default: 100).

runs
   Number of concurrent runs of the algorithm to execute. MLlib’s K-means supports running from multiple starting positions concurrently and picking the best result, which is a good way to get a better overall model (as K-means runs can stop in local minima).
```

Like other algorithms, you invoke K-means by creating a `mllib.clustering.KMeans` object (in Java/Scala) or calling `KMeans.train` (in Python).
Collaborative Filtering

Collaborative Filtering and Recommendation

Collaborative filtering is a technique for recommender systems wherein users’ ratings and interactions with various products are used to recommend new ones. Collaborative filtering is attractive because it only needs to take in a list of user/product interactions: either “explicit” interactions (i.e., ratings on a shopping site) or “implicit” ones (e.g., a user browsed a product page but did not rate the product).

Alternating Least Squares

MLlib includes an implementation of Alternating Least Squares (ALS), a popular algorithm for collaborative filtering that scales well on clusters. It is located in the mllib.recommendation.ALS class.
Dimensionality Reduction

**Principal component analysis**

Given a dataset of points in a high-dimensional space, we are often interested in reducing the dimensionality of the points so that they can be analyzed with simpler tools. For example, we might want to plot the points in two dimensions, or just reduce the number of features to train models more effectively.

**Singular value decomposition**

MLlib also provides the lower-level singular value decomposition (SVD) primitive. The SVD factorizes an $m \times n$ matrix $A$ into three matrices $A \approx U \Sigma V^T$, where:

- $U$ is an orthonormal matrix, whose columns are called left singular vectors.
- $\Sigma$ is a diagonal matrix with nonnegative diagonals in descending order, whose diagonals are called singular values.
- $V$ is an orthonormal matrix, whose columns are called right singular vectors.
Spark SQL

- Tableau
- ... 
- JDBC/ODBC
- Your Application
- Spark SQL shell

Spark SQL

- Hive
- JSON
- Parquet
- ...
Spark SQL can be built with or without Apache Hive, the Hadoop SQL engine. Spark SQL with Hive support allows us to access Hive tables, UDFs (user-defined functions), SerDes (serialization and deserialization formats), and the Hive query language (HiveQL). Hive query language (HQL) It is important to note that including the Hive libraries does not require an existing Hive installation. In general, it is best to build Spark SQL with Hive support to access these features. If you download Spark in binary form, it should already be built with Hive support. If you are building Spark from source, you should run sbt/sbt -Phive assembly.
Apache Hive
Hive’s operation modes
Using HiveQL for Spark SQL

When programming against Spark SQL we have two entry points depending on whether we need Hive support. The recommended entry point is the HiveContext to provide access to HiveQL and other Hive-dependent functionality. The more basic SQLContext provides a subset of the Spark SQL support that does not depend on Hive. The separation exists for users who might have conflicts with including all of the Hive dependencies. Using a HiveContext does not require an existing Hive setup.

HiveQL is the recommended query language for working with Spark SQL. Many resources have been written on HiveQL, including *Programming Hive* and the online *Hive Language Manual*. In Spark 1.0 and 1.1, Spark SQL is based on Hive 0.12,
This is the Hive Language Manual.

- **Commands and CLIs**
  - Commands
  - Hive CLI
  - Variable Substitution
  - Beeline CLI for HiveServer2
  - HCatalog CLI

- **File Formats**
  - Avro Files
  - ORC Files
  - Parquet
  - Compressed Data Storage
  - LZO Compression

- **Data Types**
Using Spark SQL — Steps and Example

Example 9-5. Python SQL imports

```python
# Import Spark SQL
from pyspark.sql import HiveContext, Row
```

Example 9-8. Constructing a SQL context in Python

```python
hiveCtx = HiveContext(sc)
```

Example 9-11. Loading and querying tweets in Python

```python
input = hiveCtx.jsonFile(inputFile)
# Register the input schema RDD
input.registerTempTable("tweets")
# Select tweets based on the retweetCount
topTweets = hiveCtx.sql("""SELECT text, retweetCount FROM tweets ORDER BY retweetCount LIMIT 10""")
```
Query testtweet.json

Get it from Learning Spark Github ==> https://github.com/databricks/learning-spark/tree/master/files

```python
>>> print(topTweets.collect())
[Row(text=u'Adventures With Coffee, Code, and Writing.', retweetCount=0)]
```
SchemaRDD

Both loading data and executing queries return SchemaRDDs. SchemaRDDs are similar to tables in a traditional database. Under the hood, a SchemaRDD is an RDD composed of Row objects with additional schema information of the types in each column. Row objects are just wrappers around arrays of basic types (e.g., integers and strings).
Row Objects

Row objects represent records inside SchemaRDDs, and are simply fixed-length arrays of fields.

Example 9-14. Accessing the text column in the topTweets SchemaRDD in Python

topTweetText = topTweets.map(lambda row: row.text)

<table>
<thead>
<tr>
<th>Spark SQL/HiveQL type</th>
<th>Scala type</th>
<th>Java type</th>
<th>Python</th>
</tr>
</thead>
<tbody>
<tr>
<td>STRUCT&lt;COL1:</td>
<td>Row</td>
<td>Row</td>
<td>Row</td>
</tr>
<tr>
<td>COL1_TYPE, ...&gt;</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
### Types stored by Schema RDDs

<table>
<thead>
<tr>
<th>Spark SQL/HiveQL type</th>
<th>Scala type</th>
<th>Java type</th>
<th>Python</th>
</tr>
</thead>
<tbody>
<tr>
<td>TINYINT</td>
<td>Byte</td>
<td>Byte/byte</td>
<td>int/long (in range of -128 to 127)</td>
</tr>
<tr>
<td>SMALLINT</td>
<td>Short</td>
<td>Short/short</td>
<td>int/long (in range of -32768 to 32767)</td>
</tr>
<tr>
<td>INT</td>
<td>Int</td>
<td>Int/int</td>
<td>int or long</td>
</tr>
<tr>
<td>BIGINT</td>
<td>Long</td>
<td>Long/long</td>
<td>long</td>
</tr>
<tr>
<td>FLOAT</td>
<td>Float</td>
<td>Float/float</td>
<td>float</td>
</tr>
<tr>
<td>DOUBLE</td>
<td>Double</td>
<td>Double/Double</td>
<td>float</td>
</tr>
<tr>
<td>STRING</td>
<td>String</td>
<td>String</td>
<td>string</td>
</tr>
<tr>
<td>BINARY</td>
<td>Array[Byte]</td>
<td>byte[]</td>
<td>bytestring</td>
</tr>
<tr>
<td>BOOLEAN</td>
<td>Boolean</td>
<td>Boolean/boolean</td>
<td>bool</td>
</tr>
<tr>
<td>TIMESTAMP</td>
<td>java.sql.Timestamp</td>
<td>java.sql.Timestamp</td>
<td>datetime.datetime</td>
</tr>
<tr>
<td>ARRAY&lt;DATA_TYPE&gt;</td>
<td>Seq</td>
<td>List</td>
<td>list, tuple, or array</td>
</tr>
<tr>
<td>MAP&lt;KEY_TYPE, VAL_TYPE&gt;</td>
<td>Map</td>
<td>Map</td>
<td>dict</td>
</tr>
</tbody>
</table>
Look at the Schema

```python
>>> input.printSchema()
root
|-- contributorsIDs: array (nullable = true)
  |-- element: string (containsNull = false)
|-- createdAt: string (nullable = true)
|-- currentUserRetweetId: integer (nullable = true)
|-- hashtagEntities: array (nullable = true)
  |-- element: string (containsNull = false)
|-- id: long (nullable = true)
|-- inReplyToStatusId: integer (nullable = true)
|-- inReplyToUserID: integer (nullable = true)
|-- isFavorited: boolean (nullable = true)
|-- isPossiblySensitive: boolean (nullable = true)
|-- isTruncated: boolean (nullable = true)
|-- mediaEntities: array (nullable = true)
  |-- element: string (containsNull = false)
|-- retweetCount: integer (nullable = true)
|-- source: string (nullable = true)
|-- text: string (nullable = true)
|-- urlEntities: array (nullable = true)
  |-- element: string (containsNull = false)
```

(not a complete screen shot)
Another way to create SchemaRDD

Example 9-28. Creating a SchemaRDD using Row and named tuple in Python

```python
happyPeopleRDD = sc.parallelize([Row(name="holden", favouriteBeverage="coffee")])
happyPeopleSchemaRDD = hiveCtx.inferSchema(happyPeopleRDD)
happyPeopleSchemaRDD.registerTempTable("happy_people")
```
JDBC Server

Spark SQL provides JDBC connectivity, which is useful for connecting business intelligence tools to a Spark cluster and for sharing a cluster across multiple users.

The server can be launched with `sbin/start-thriftserver.sh` in your Spark directory (Example 9-31). This script takes many of the same options as `spark-submit`. By default it listens on `localhost:10000`, but we can change these with either environment variables (`HIVE_SERVER2_THRIFT_PORT` and `HIVE_SERVER2_THRIFT_BIND_HOST`), or with Hive configuration properties (`hive.server2.thrift.port` and `hive.server2.thrift.bind.host`). You can also specify Hive properties on the command line with `--hiveconf property=value`.

**Example 9-31. Launching the JDBC server**

```
./sbin/start-thriftserver.sh --master sparkMaster
```

**Example 9-32. Connecting to the JDBC server with Beeline**

```
holden@hmbp2:~/repos/spark$ ./bin/beeline -u jdbc:hive2://localhost:10000
Spark assembly has been built with Hive, including Datanucleus jars on classpath
scan complete in 1ms
Connecting to jdbc:hive2://localhost:10000
Connected to: Spark SQL (version 1.2.0-SNAPSHOT)
```
User-Defined Functions (UDF)

UDFs allow you to register custom functions in Python, Java, and Scala to call within SQL.

This is a very popular way to expose advanced functionality to SQL users in an organization, so that these users can call into it without writing code.

Example 9-36. Python string length UDF

```python
# Make a UDF to tell us how long some text is
hiveCtx.registerFunction("strLenPython", lambda x: len(x), IntegerType())
lengthSchemaRDD = hiveCtx.sql("SELECT strLenPython('text') FROM tweets LIMIT 10")
```
Spark Streaming

Many applications benefit from acting on data as soon as it arrives. For example, an application might track statistics about page views in real time, train a machine learning model, or automatically detect anomalies. Spark Streaming is Spark’s module for such applications. It lets users write streaming applications using a very similar API to batch jobs, and thus reuse a lot of the skills and even code they built for those.

Much like Spark is built on the concept of RDDs, Spark Streaming provides an abstraction called DStreams, or discretized streams. A DStream is a sequence of data arriving over time. Internally, each DStream is represented as a sequence of RDDs arriving at each time step (hence the name “discretized”).

In Spark 1.1, Spark Streaming is available only in Java and Scala. Spark 1.2 has limited Python support.
Spark Streaming architecture

- **Input Data Streams**
- **Spark Streaming**
  - **Spark**
  - **Batches of Input Data**
  - **Results pushed to external systems**

- **Server running at localhost:7777**
  - **Newline separated text**
  - **Lines DStream**
  - **Data from time 0 to 1**
  - **Data from time 1 to 2**
  - **Data from time 2 to 3**
  - **Data from time 3 to 4**

- **Error Lines DStream**
  - **Error Lines from time 0 to 1**
  - **Error Lines from time 1 to 2**
  - **Error Lines from time 2 to 3**
  - **Error Lines from time 3 to 4**

- **Filter Transformation**
Spark Streaming with Spark’s components

- Driver Program
  - StreamingContext
    - Spark jobs to process received data
  - SparkContext
    - Tasks to process received data

- Worker Node
  - Executor
    - Long Task
    - Receiver
  - Task

- Input stream
- Data replicated to another worker node
- Output results in batches
Try these examples

Databricks Reference Apps

At Databricks, we are developing a set of reference applications that demonstrate how to use Apache Spark. This book/repo contains the reference applications.

- View the code in the Github Repo here: https://github.com/databricks/reference-apps
- Read the documentation here: http://databricks.gitbooks.io/databricks-spark-reference-applications/
- Submit feedback or issues here: https://github.com/databricks/reference-apps/issues

The reference applications will appeal to those who want to learn Spark and learn better by example. Browse the applications, see what features of the reference applications are similar to the features you want to build, and refashion the code samples for your needs. Additionally, this is meant to be a practical guide for using Spark in your systems, so the applications mention other technologies that are compatible with Spark - such as what file systems to use for storing your massive data sets.

- **Log Analysis Application** - The log analysis reference application contains a series of tutorials for learning Spark by example as well as a final application that can be used to monitor Apache access logs. The examples use Spark in batch mode, cover Spark SQL, as well as Spark Streaming.

- **Twitter Streaming Language Classifier** - This application demonstrates how to fetch and train a language classifier for Tweets using Spark MLLib. Then Spark Streaming is used to call the trained classifier and filter out live tweets that match a specified cluster. To build this example go into the twitter_classifier/scala and follow the direction in the README.
E6893 Big Data Analytics:

Demo Session for Mahout and Spark

Eric Johnson