Columbia University

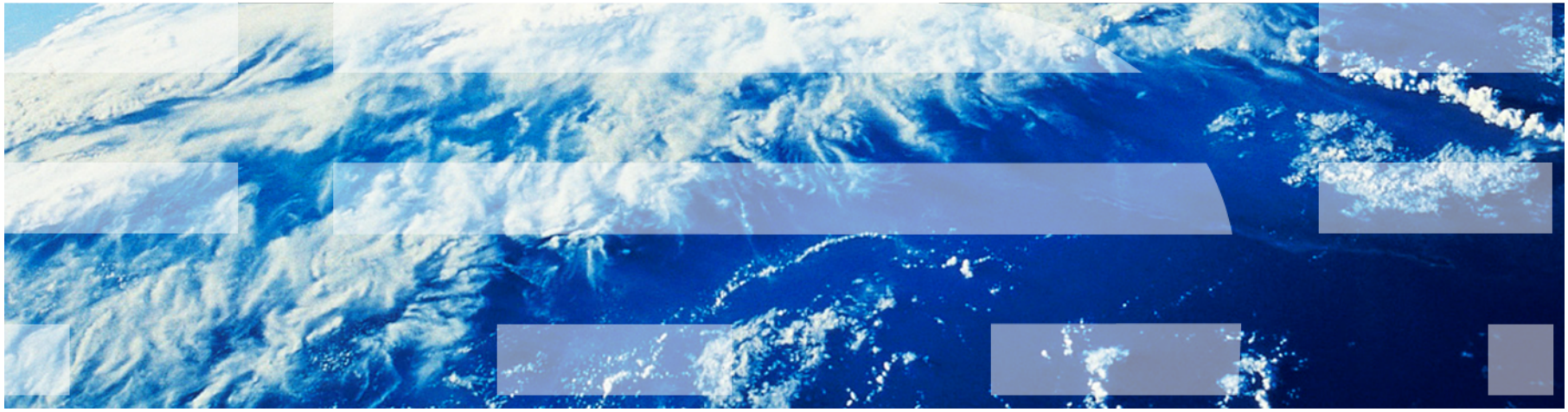
E6893 Big Data Analytics Lecture 7:

Spark and Data Analytics

Ching-Yung Lin, Ph.D.

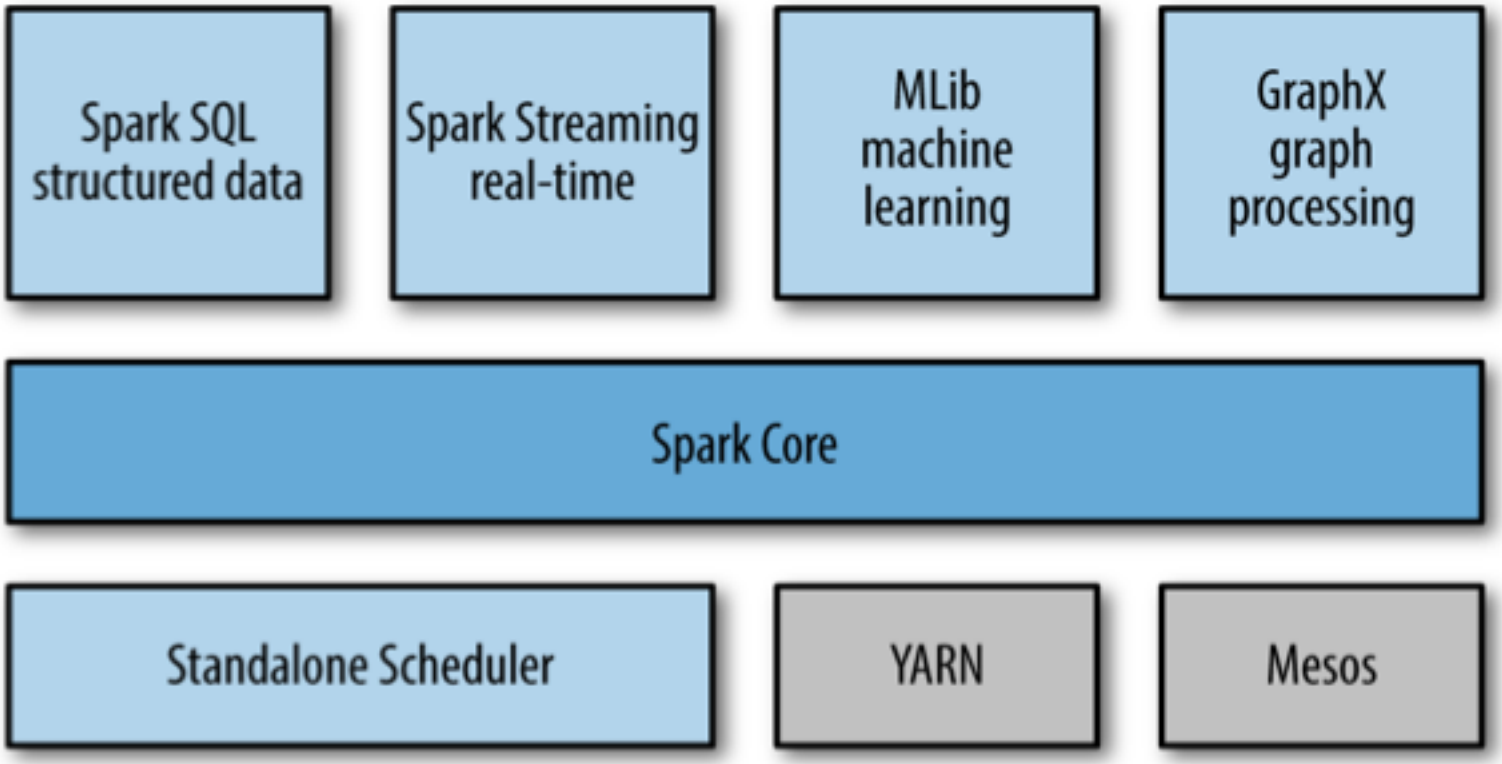
Adjunct Professor, Dept. of Electrical Engineering and Computer Science

IBM Distinguished Researcher and Chief Scientist, Graph Computing



Reference

Spark Stack



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 | Libraries | Documentation | Examples | Community | FAQ

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 26, 2014)

Registration open for Spark Summit East 2015 (Nov 26, 2014)

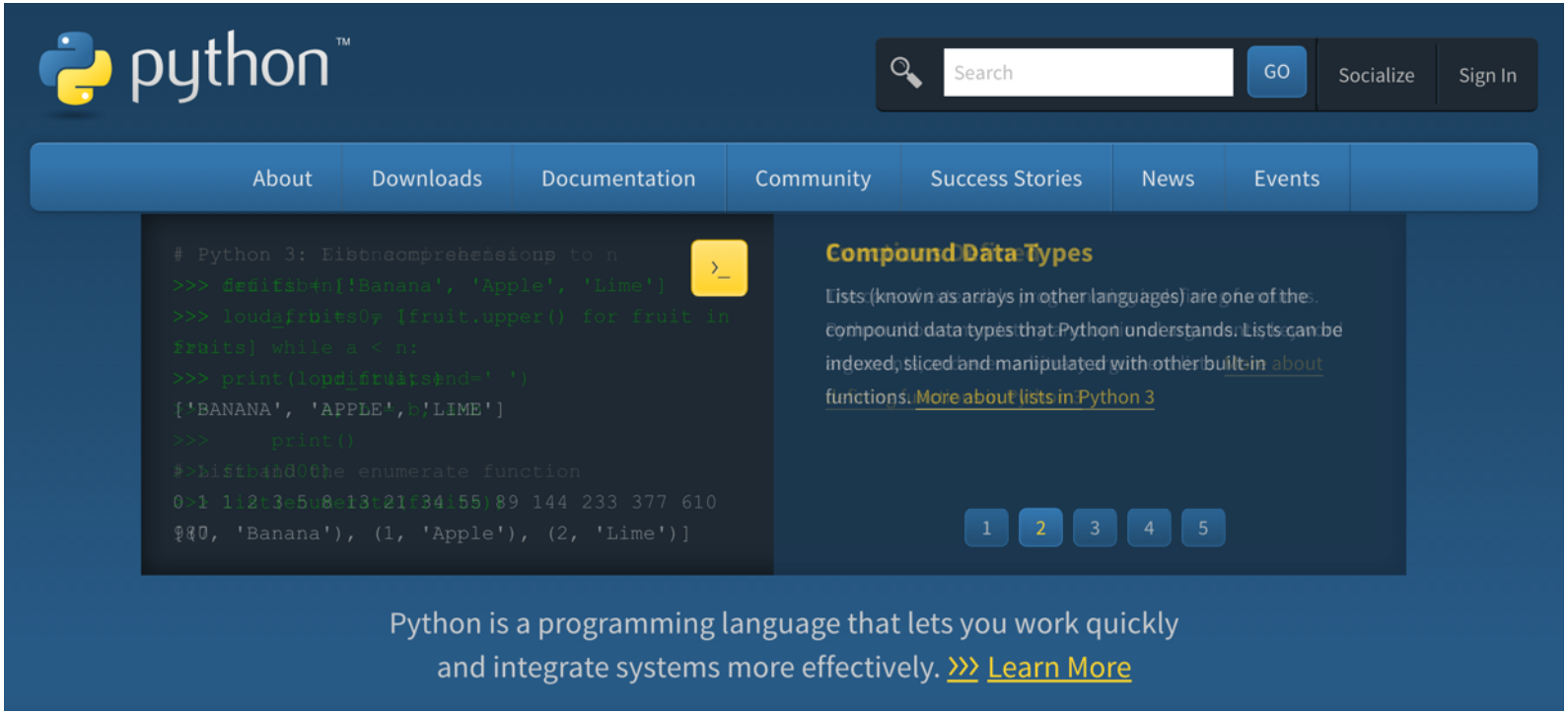
Archive

Download Spark

Built-in Libraries:

- Spark SQL
- Spark Streaming
- MLlib (machine learning)
- GraphX (graph)

First language to use — Python



Spark's Python Shell (PySpark Shell)

bin/pyspark

```
holden@hmbp2:~/Downloads/spark-1.1.0-bin-hadoop1$ ./bin/pyspark
Python 2.7.6 (default, Mar 22 2014, 22:59:56)
[GCC 4.8.2] on linux2
Type "help", "copyright", "credits" or "license" for more information.
Spark assembly has been built with Hive, including Datanucleus jars on classpath
Using Spark's default log4j profile: org/apache/spark/log4j-defaults.properties
14/11/19 14:33:49 WARN Utils: Your hostname, hmbp2 resolves to a loopback address: 127.0.1.1; using 172.17.42.1 instead (on interface docker0)
14/11/19 14:33:49 WARN Utils: Set SPARK_LOCAL_IP if you need to bind to another address
14/11/19 14:33:49 INFO SecurityManager: Changing view acls to: holden,
14/11/19 14:33:49 INFO SecurityManager: Changing modify acls to: holden,
14/11/19 14:33:49 INFO SecurityManager: SecurityManager: authentication disabled; ui acls disabled; users with view permissions: Set(holden, )
; users with modify permissions: Set(holden, )
14/11/19 14:33:49 INFO Slf4jLogger: Slf4jLogger started
14/11/19 14:33:49 INFO Remoting: Starting remoting
14/11/19 14:33:49 INFO Remoting: Remoting started; listening on addresses :[akka.tcp://sparkDriver@172.17.42.1:35021]
14/11/19 14:33:49 INFO Remoting: Remoting now listens on addresses: [akka.tcp://sparkDriver@172.17.42.1:35021]
14/11/19 14:33:49 INFO Utils: Successfully started service 'sparkDriver' on port 35021.
14/11/19 14:33:49 INFO SparkEnv: Registering MapOutputTracker
14/11/19 14:33:49 INFO SparkEnv: Registering BlockManagerMaster
14/11/19 14:33:49 INFO DiskBlockManager: Created local directory at /tmp/spark-local-20141119143349-5776
14/11/19 14:33:49 INFO Utils: Successfully started service 'Connection manager for block manager' on port 57218.
14/11/19 14:33:49 INFO ConnectionManager: Bound socket to port 57218 with id = ConnectionManagerId(172.17.42.1,57218)
14/11/19 14:33:49 INFO MemoryStore: MemoryStore started with capacity 265.4 MB
14/11/19 14:33:49 INFO BlockManagerMaster: Trying to register BlockManager
14/11/19 14:33:49 INFO BlockManagerMasterActor: Registering block manager 172.17.42.1:57218 with 265.4 MB RAM
14/11/19 14:33:49 INFO BlockManagerMaster: Registered BlockManager
14/11/19 14:33:49 INFO HttpFileServer: HTTP File server directory is /tmp/spark-399c53ec-0be8-4043-9a7d-9345e970576d
14/11/19 14:33:49 INFO HttpServer: Starting HTTP Server
14/11/19 14:33:49 INFO Utils: Successfully started service 'HTTP file server' on port 49008.
14/11/19 14:33:49 INFO Utils: Successfully started service 'SparkUI' on port 4040.
14/11/19 14:33:49 INFO SparkUI: Started SparkUI at http://172.17.42.1:4040
14/11/19 14:33:49 INFO AkkaUtils: Connecting to HeartbeatReceiver: akka.tcp://sparkDriver@172.17.42.1:35021/user/HeartbeatReceiver
Welcome to
      ____              __
     / __/__  ___ _____/ /__
    _\ \/ _ \/ _ `/ __/  '_/
   /__ / .__/\_,_/_/ /_/\_\   version 1.1.0
      /_/

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

```
>>> 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:

```
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:

```
log4j.rootCategory=WARN, console
```

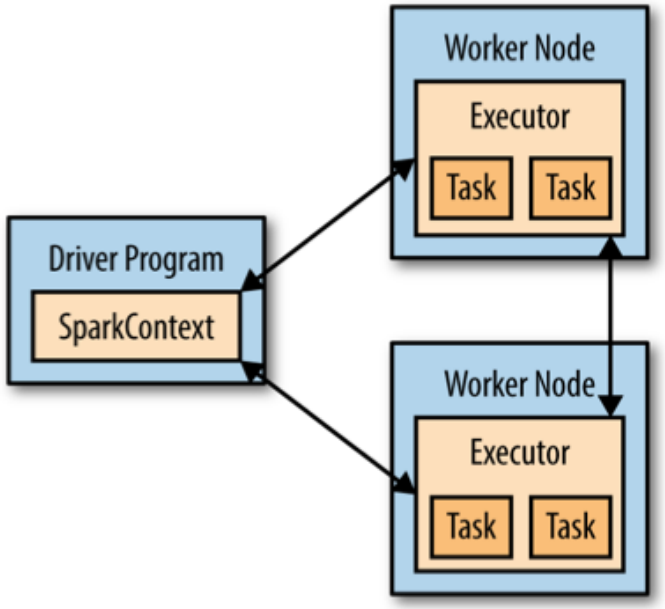
Core Spark Concepts

- 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

```
>>> 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.

```
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

```
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**.

```
>>> 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.

Example 3-4. Persisting an RDD in memory

```
>>> pythonLines.persist

>>> pythonLines.count()
2

>>> pythonLines.first()
u'## 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

```
lines = sc.parallelize(["pandas", "i like pandas"])
```

Spark file loading

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.

Loading and Saving

Example 5-1. Loading a text file in Python

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

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

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

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))
```

Loading and Saving (II)

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)
```

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(outputFile)
```

Loading and Saving (III)

Example 5-20. Loading a SequenceFile in Python

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

File compression options

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
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

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","contactlong":"-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

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

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

callSigns = file.flatMap(extractCallSigns)
callSigns.saveAsTextFile(outputDir + "/callsigns")
print "Blank lines: %d" % blankLines.value
```

Numeric RDD Operations

Table 6-2. Summary statistics available from StatsCounter

Method	Meaning
`count()`	Number of elements in the RDD
`mean()`	Average of the elements
`sum()`	Total
`max()`	Maximum value
`min()`	Minimum value
`variance()`	Variance of the elements
`sampleVariance()`	Variance of the elements, computed for a sample
`stdev()`	Standard deviation
`sampleStdev()`	Sample standard deviation

Removing outliers

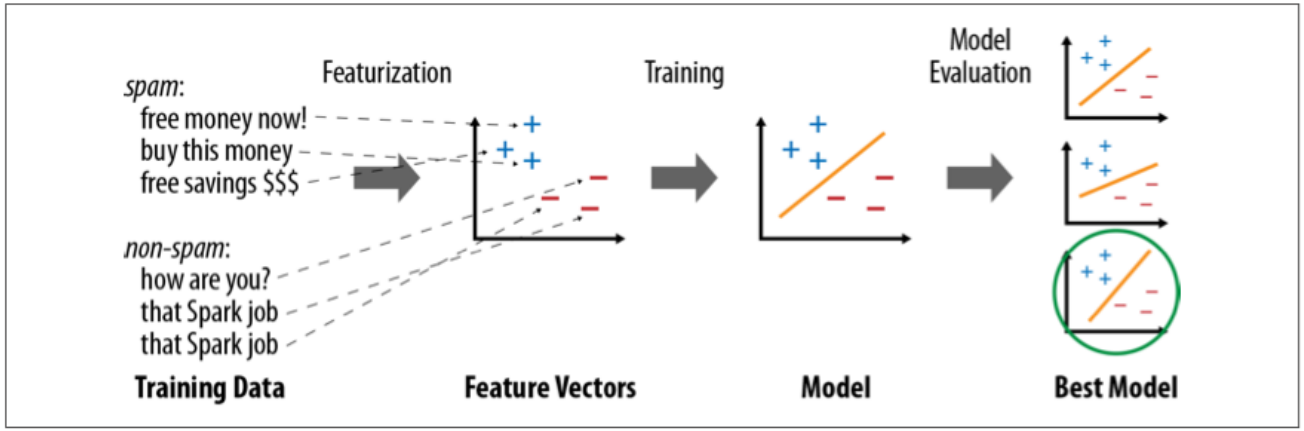
Example 6-19. Removing outliers in Python

```
# 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()
stddev = std.stdev()
mean = stats.mean()
reasonableDistances = distanceNumerics.filter(
  lambda x: math.fabs(x - mean) < 3 * stddev)
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

```
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

```
>>> 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

```
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

```
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 logisitic 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

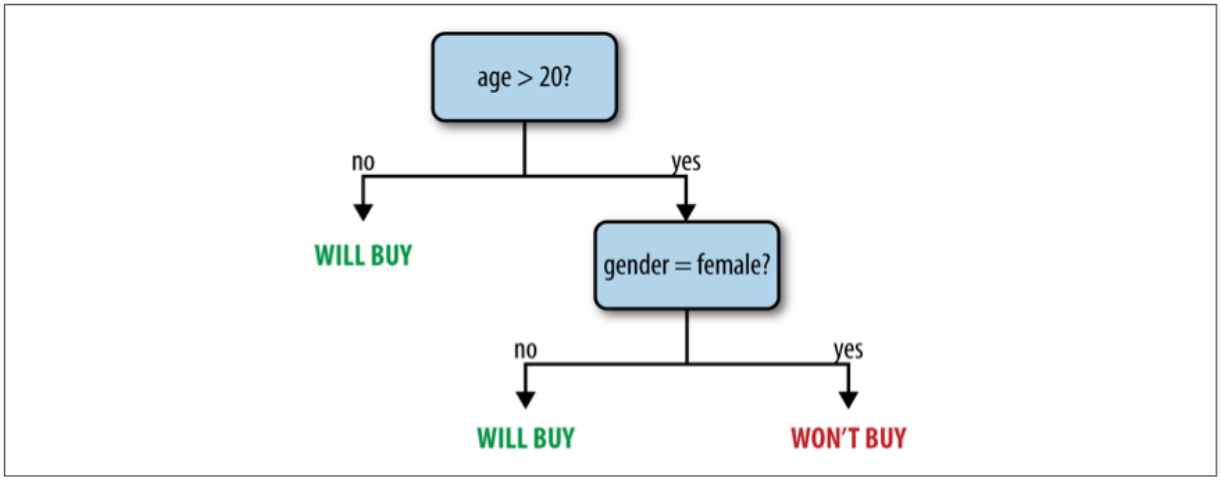


Figure 11-2. An example decision tree predicting whether a user might buy a product

In MLlib, you can train trees using the `mllib.tree.DecisionTree` class, through the static methods `trainClassifier()` and `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.[5] K-means|| is similar to the K-means++ initialization procedure often used in single-node settings.

`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.[6] It is located in the `mllib.recommendation.ALS` class.

Dimensionality Reduction

Principal component analysis

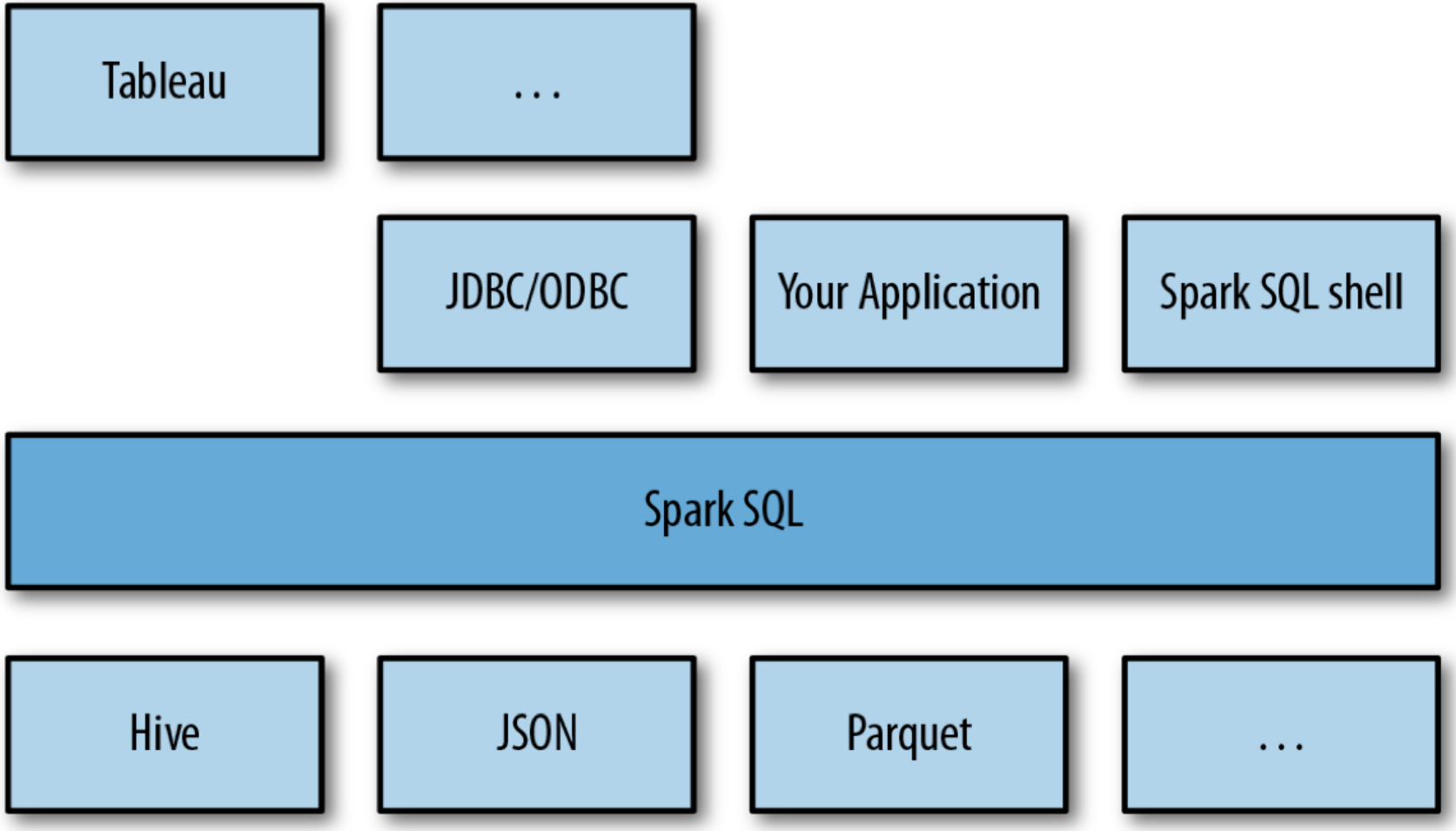
Given a dataset of points in a high-dimensional space, we are often interested in reducing the dimensionality of the points so that they can be analyzed with simpler tools. For example, we might want to plot the points in two dimensions, or just reduce the number of features to train models more effectively.

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.
- Σ 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.

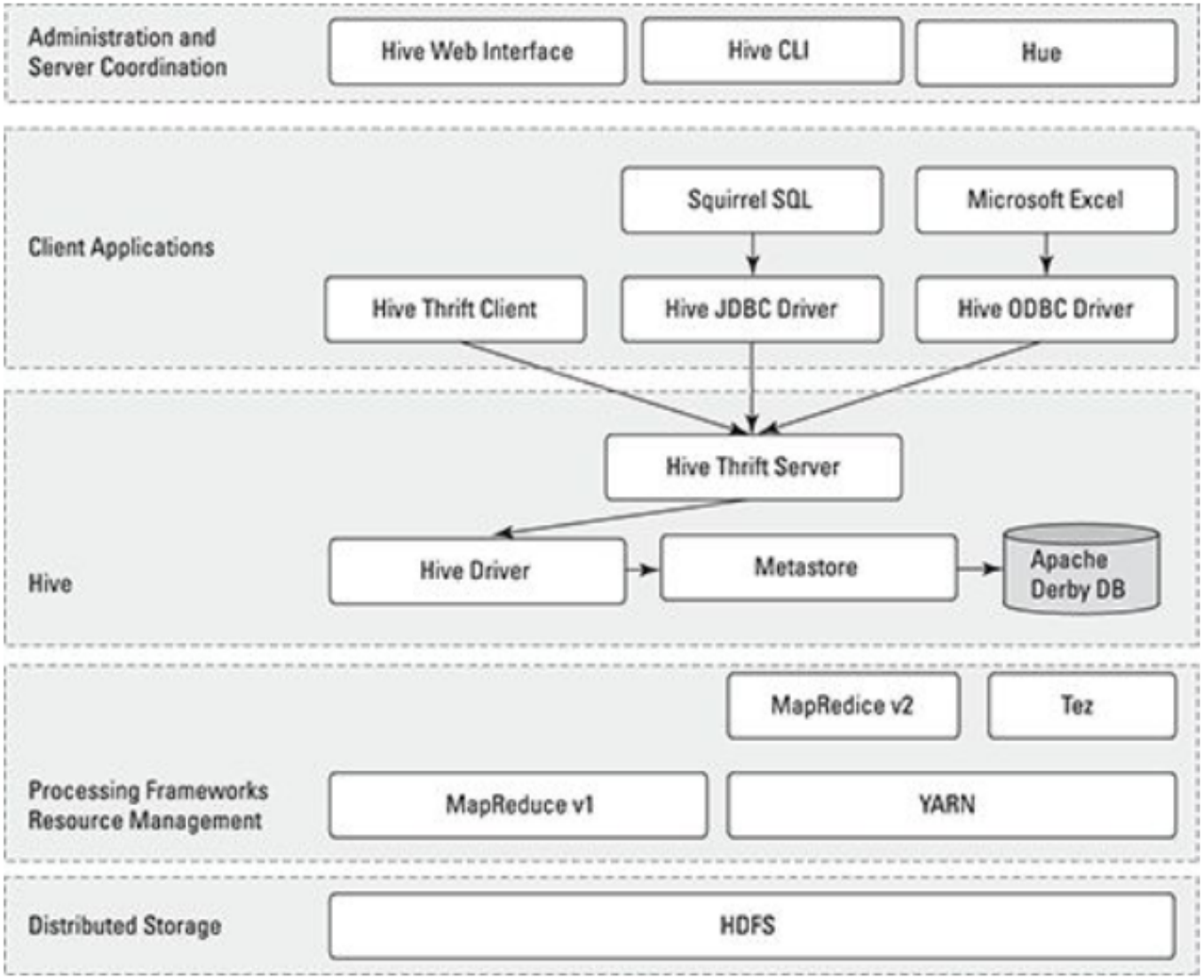
Spark SQL



Spark SQL

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



Using Hive to Create a Table

```
 (A) $ $HIVE_HOME/bin hive --service cli
(B) hive> set hive.cli.print.current.db=true;
(C) hive (default)> CREATE DATABASE ourfirstdatabase;
OK
Time taken: 3.756 seconds
(D) hive (default)> USE ourfirstdatabase;
OK
Time taken: 0.039 seconds
(E) hive (ourfirstdatabase)> CREATE TABLE our_first_table (
                   > FirstName     STRING,
                   > LastName      STRING,
                   > EmployeeId     INT);
OK
Time taken: 0.043 seconds
hive (ourfirstdatabase)> quit;
(F) $ ls /home/biadmin/Hive/warehouse/ourfirstdatabase.db
our_first_table
```

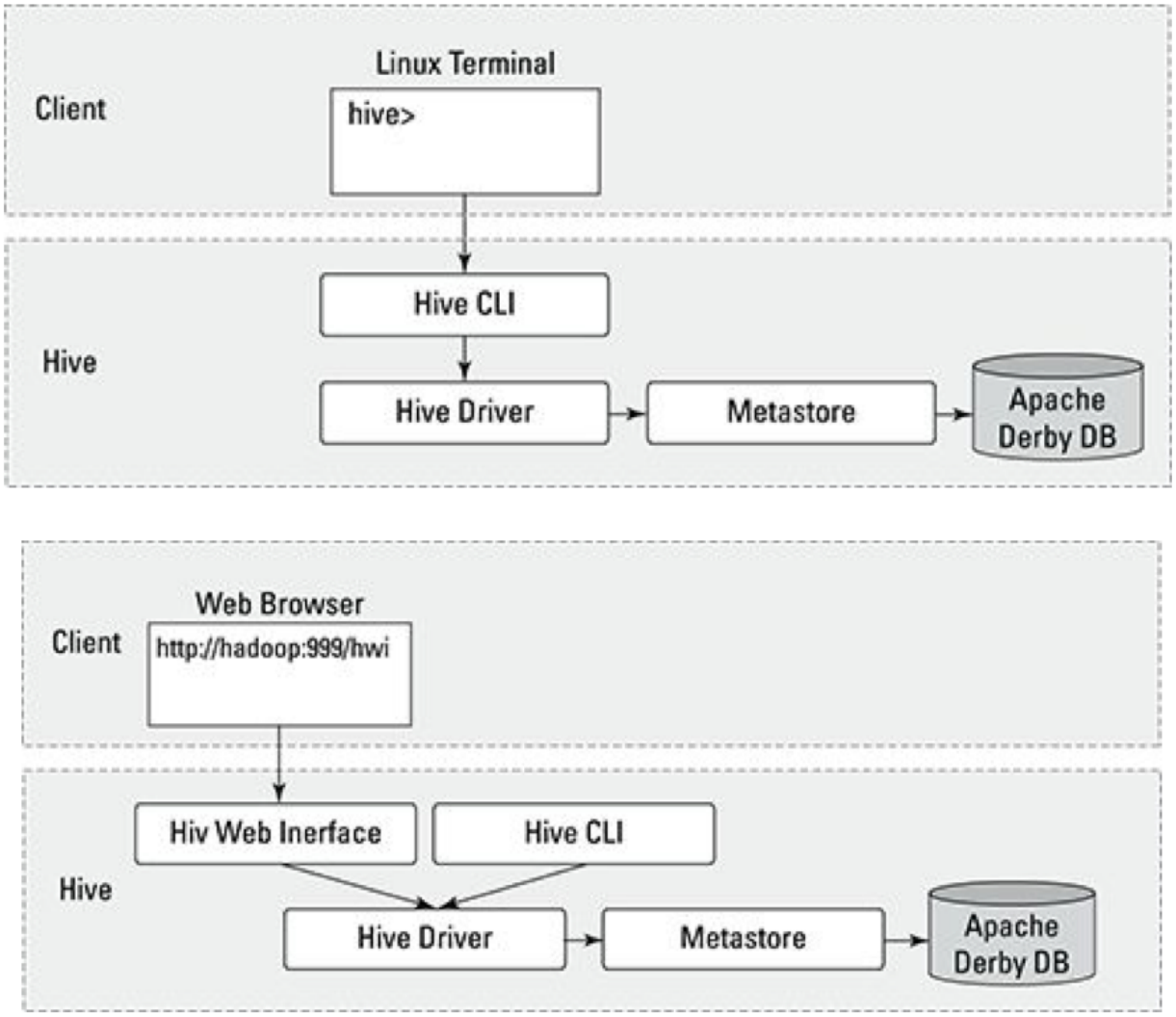
Creating, Dropping, and Altering DBs in Apache Hive

```
(1) $ $HIVE_HOME/bin hive --service cli
 (2) hive> set hive.cli.print.current.db=true;
 (3) hive (default)> USE ourfirstdatabase;
 (4) hive (ourfirstdatabase)> ALTER DATABASE
 ourfirstdatabase SET DBPROPERTIES
 ('creator'='Bruce Brown', 'created_for'='Learning Hive
 DDL');
 OK
 Time taken: 0.138 seconds
 (5) hive (ourfirstdatabase)> DESCRIBE DATABASE
 EXTENDED ourfirstdatabase;
 OK
 ourfirstdatabase                    file:/home/biad
 min/Hive/warehouse/ourfirstdatabase.db   {created_f
 or=Learning Hive DDL, creator=Bruce Brown}
 Time taken: 0.084 seconds, Fetched: 1 row(s)CREATE
 (DATABASE|SCHEMA) [IF NOT EXISTS]
 database_name
 (6) hive (ourfirstdatabase)> DROP DATABASE
 ourfirstdatabase CASCADE;
 OK
 Time taken: 0.132 seconds
```

Another Hive Example

```
(A) CREATE TABLE IF NOT EXISTS FlightInfo2007 (
Year SMALLINT, Month TINYINT, DayofMonth TINYINT, DayOfWeek TINYINT,
DepTime SMALLINT, CRSDepTime SMALLINT, ArrTime SMALLINT, CRSArrTime SMALLINT,
UniqueCarrier STRING, FlightNum STRING, TailNum STRING,
ActualElapsedTime SMALLINT, CRSElapsedTime SMALLINT,
AirTime SMALLINT, ArrDelay SMALLINT, DepDelay SMALLINT,
Origin STRING, Dest STRING,Distance INT,
TaxiIn SMALLINT, TaxiOut SMALLINT, Cancelled SMALLINT,
CancellationCode STRING, Diverted SMALLINT,
CarrierDelay SMALLINT, WeatherDelay SMALLINT,
NASDelay SMALLINT, SecurityDelay SMALLINT, LateAircraftDelay SMALLINT)
COMMENT 'Flight InfoTable'
ROW FORMAT DELIMITED
FIELDS TERMINATED BY ','
LINES TERMINATED BY '\n'
STORED AS TEXTFILE
TBLPROPERTIES ('creator'='Bruce Brown', 'created_at'='Thu Sep 19 10:58:00 EDT 2013');
```

Hive's operation modes

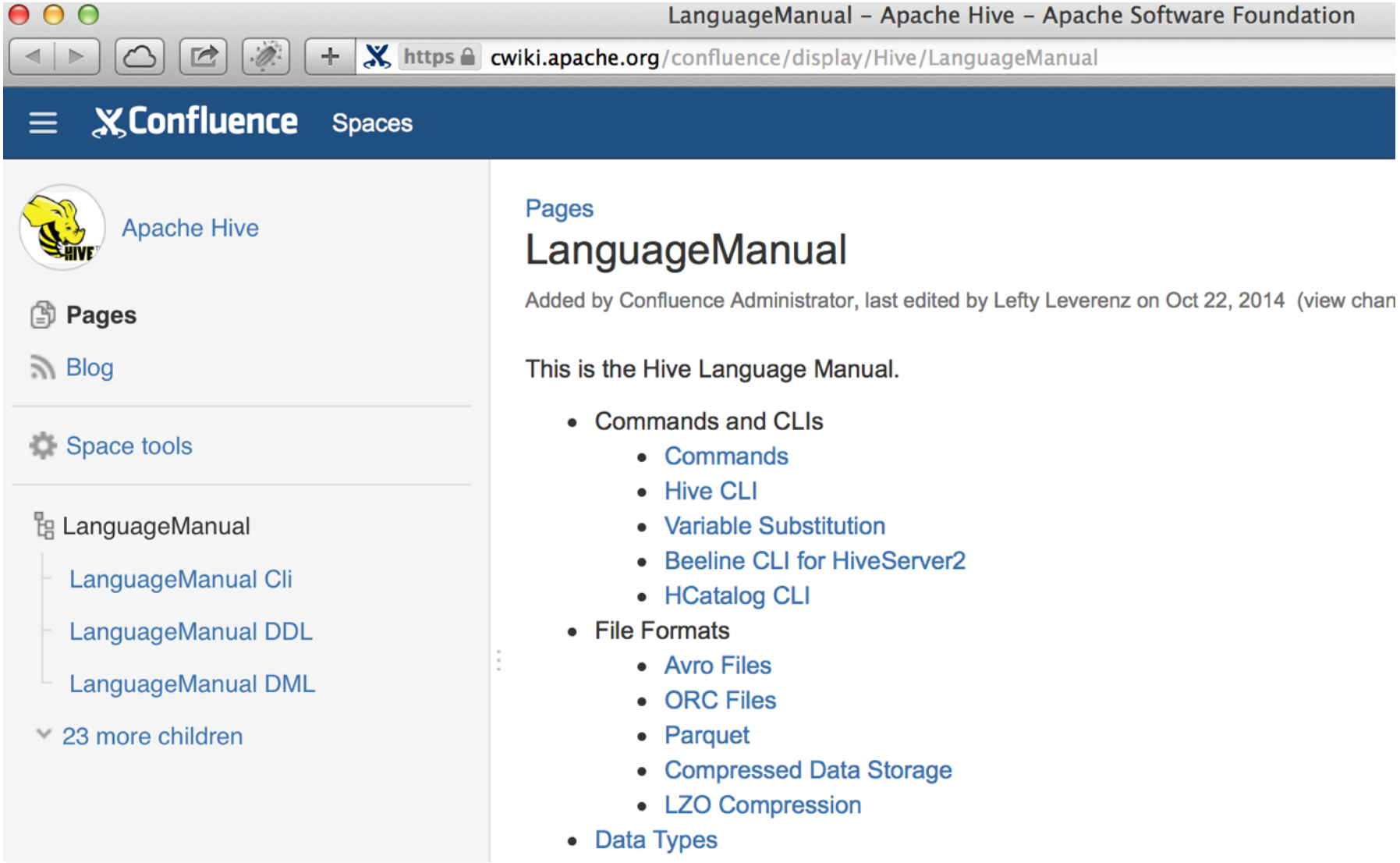


Using HiveQL for Spark SQL

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,

Hive Language Manual



Using Spark SQL — Steps and Example

Example 9-5. Python SQL imports

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

Example 9-8. Constructing a SQL context in Python

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

Example 9-11. Loading and quering tweets in 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


```
{"createdAt":"Nov 4, 2014 4:56:59 PM","id":529799371026485248,"text":"Adventures With
Coffee, Code, and Writing.","source":"\u003ca href\u003d\"http://twitter.com\"
rel\u003d\"nofollow\"\u003eTwitter Web
Client\u003c/a\u003e","isTruncated":false,"inReplyToStatusId":-1,"inReplyToUserId":-1,"
isFavorited":false,"retweetCount":0,"isPossiblySensitive":false,"contributorsIDs":[],"
userMentionEntities":[],"urlEntities":[],"hashtagEntities":[],"mediaEntities":[],"
currentUserRetweetId":-1,"user":{"id":15594928,"name":"Holden
Karau","screenName":"holdenkarau","location":"","description":"","descriptionURLEntities":
[],"isContributorsEnabled":false,"profileImageUrl":"http://pbs.twimg.com/profile_images/
3005696115/2036374bbadbed85249cdd50aac6e170_normal.jpeg","profileImageUrlHttps":"https://
pbs.twimg.com/profile_images/3005696115/2036374bbadbed85249cdd50aac6e170_normal.jpeg","
isProtected":false,"followersCount":1231,"profileBackgroundColor":"C0DEED","
profileTextColor":"333333","profileLinkColor":"0084B4","profileSidebarFillColor":"DDEEF6",
"profileSidebarBorderColor":"FFFFFF","profileUseBackgroundImage":true,"showAllInlineMedia"
:false,"friendsCount":600,"createdAt":"Aug 5, 2011 9:42:44
AM","favouritesCount":1095,"utcOffset":-3,"profileBackgroundImageUrl":"","
profileBackgroundImageUrlHttps":"","profileBannerImageUrl":"","profileBackgroundTiled":
true,"lang":"en","statusesCount":6234,"isGeoEnabled":true,"isVerified":false,"translator":
false,"listedCount":0,"isFollowRequestSent":false}}
```


```
>>> 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)
```

Spark SQL/HiveQL type	Scala type	Java type	Python
STRUCT<COL1: COL1_TYPE, ...>	Row	Row	Row

Types stored by Schema RDDs

Spark SQL/HiveQL type	Scala type	Java type	Python
TINYINT	Byte	Byte/byte	int/long (in range of –128 to 127)
SMALLINT	Short	Short/short	int/long (in range of –32768 to 32767)
INT	Int	Int/int	int or long
BIGINT	Long	Long/long	long
FLOAT	Float	Float/float	float
DOUBLE	Double	Double/double	float
DECIMAL	Scala.math.BigDecimal	Java.math.BigDecimal	decimal.Decimal
STRING	String	String	string
BINARY	Array[Byte]	byte[]	bytearray
BOOLEAN	Boolean	Boolean/boolean	bool
TIMESTAMP	java.sql.TimeStamp	java.sql.TimeStamp	datetime.datetime
ARRAY<DATA_TYPE>	Seq	List	list, tuple, or array
MAP<KEY_TYPE, VAL_TYPE>	Map	Map	dict

Look at the Schema

```
>>> 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

```
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

```
# 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")
```