APACHE SPARK

Chapter 1

WHY SPARK?

Why Spark?

- In this chapter you will learn
- What problems exist with traditional large-scale computing systems
- How Spark addresses those issues
- Some typical big data questions Spark can be used to answer

Traditional Large-Scale Computation

- Traditionally, computation has been processor bound
  - Relatively small amounts of data
  - Lots of complex processing
- The early solution: bigger computers
  - Faster processor, more memory
  - But even this couldn't keep up

Distributed Systems

- **The better solution: more computers**
  - **Distributed systems - use multiple machines for a single job**

"In pioneer days they used oxen for heavy pulling, and when one ox couldn't budge a log, we didn't try to grow a larger ox. We shouldn't be trying for bigger computers, but for more systems of computers."

-Grace Hopper

Distributed Systems: Challenges

- Challenges with distributed systems
  - Programming complexity
  - Keeping data and processes in sync
  - Finite bandwidth
  - Partial failures

Distributed Systems: The Data Bottleneck

- **Traditionally, data is stored in a central location**
- **Data is copied to processors at runtime**
- **Fine for limited amounts of data**

Distributed Systems: The Data Bottleneck

- **Modern systems have much more data**
  - **terabytes+ a day**
  - **petabytes+ total**
- **We need a new approach…**

Big Data Processing

- Hadoop introduced a radical new approach based on two key concepts
  - Distribute data when the data is stored
  - Run computation where the data is
- Spark takes this new approach to the next level
  - Data is distributed in memory

Distributed Processing with the Spark Framework

Advantages of Spark

- High level programming framework
  - Programmers can focus on logic, not plumbing
- Cluster computing
  - Application processes are distributed across a cluster of worker nodes
  - Managed by a single “master”
  - Scalable and fault tolerant
- Distributed storage
  - Data is distributed when it is stored
  - Replicated for efficiency and fault tolerance
  - “Bring the computation to the data”
- Data in memory
  - Configurable caching for efficient iteration

Scalability

- Increasing load results in a graceful decline in performance
  - Not failure of the system
- Adding nodes adds capacity proportionally

Fault Tolerance

- Node failure is inevitable
- What happens?
  - System continues to function
  - Master re-assigns tasks to a different node
  - Data replication= no loss of data
  - Nodes which recover rejoin the cluster automatically

Who Uses Spark?

- Yahoo!
  - Personalization and ad analytics
- Conviva
  - Real-time video stream optimization
- Technicolor
  - Real-time analytics for telco clients
- Ooyala
  - Cross-device personalized video experience
- Plus…
  - Intel, Groupon, TrendMicro, Autodesk, Nokia, Shopify, ClearStory, Technicolor, and many more…

Common Spark Use Cases

- **Extract/Transform/Load (ETL)**
- **Text mining**
- **Index building**
- **Graph creation and analysis**
- **Pattern recognition**
- **Collaborative filtering**
- **Prediction models**
- **Sentiment analysis**
- **Risk assessment**

Benefits of Spark

- **Previously impossible or impractical analysis**
- **Lower cost**
- **Less time**
- **Greater flexibility**
- **Near-linear scalability**

Spark v. Hadoop MapReduce

- Spark takes the concepts of MapReduce to the next level
- Higher level API = faster, easier development
  - Low latency = near real-time processing
  - In-memory data storage = up to 100x performance improvement

```
sc.textFile(file) \
  .flatMap(lambda s: s.split()) \
  .map(lambda w: (w,1)) \
  .reduceByKey(lambda v1,v2: v1+v2)
  .saveAsTextFile(output)
```

```
public class WordCount {
  public static void main(String[] args) throw
   Job job = new Job();
    job.setJarByClass(WordCount.class);
    job.setJobName("Word Count");
    FileInputFormat.setInputPaths(job, new Path(args[0]));
    FileOutputFormat.setOutputPath(job, new Path(args[1]));
    job.setMapperClass(WordMapper.class);
    job.setReducerClass(SumReducer.class);
    job.setMapOutputKeyClass(Text.class);
    job.setMapOutputValueClass(IntWritable.class);
    job.setOutputKeyClass(Text.class);
    job.setOutputValueClass(IntWritable.class);
    boolean success = job.waitForCompletion(true);
    System.exit(success ? 0 : 1);
  }
}

public class WordMapper extends Mapper<LongWritable, Text, Text,
IntWritable> {
public void map(LongWritable key, Text value,
Context context) throws IOException, InterruptedException {
    String line = value.toString();
    for (String word : line.split("\\W+")) {
      if (word.length() > 0)
        context.write(new Text(word), new IntWritable(1));
      }
    }
  }
}

public class SumReducer extends Reducer<Text, IntWritable, Text,
IntWritable> {
  public void reduce(Text key, Iterable<IntWritable>
  values, Context context) throws IOException, InterruptedException {
    int wordCount = 0;
    for (IntWritable value : values) {
      wordCount += value.get();
    }
    context.write(key, new IntWritable(wordCount));
  }
}
```

Chapter 2

SPARK BASICS

What is Apache Spark?

- Apache Spark is a fast and general engine for large scale data processing
- Written in Scala
  - Functional programming language that runs in a JVM
- Spark Shell
  - Interactive- for learning or data exploration
  - Python or Scala
- Spark Applications
  - For large scale data processing
  - Python, Scala , or Java

Spark Shell

- The Spark Shell provides interactive data exploration (REPL)
- Writing Spark applications without the shell will be covered later
-
- Python Shell: pyspark Scala Shell: spark-shell

```
$ pyspark

Welcome to
      ____              __
     / __/__  ___ _____/ /__
    _\ \/ _ \/ _ `/ __/  '_/
   /__ / .__/\_,_/_/ /_/\_\   version 0.9.1
      /_/

Using Python version 2.6.6 (r266:84292, Jan
22 2014 09:42:36)
Spark context available as sc.

>>>
```

```
$ spark-shell

Welcome to
      ____              __
     / __/__  ___ _____/ /__
    _\ \/ _ \/ _ `/ __/  '_/
   /___/ .__/\_,_/_/ /_/\_\   version 0.9.1
      /_/

Using Scala version 2.10.3 (Java HotSpot(TM)
64-Bit Server VM, Java 1.7.0_51)
Created spark context..
Spark context available as sc.

scala>
```

REPL: Read/Evaluate/Print Loop

Spark Context

- Every Spark application requires a Spark Context
  - The main entry point to the Spark API
- Spark Shell provides a preconfigured Spark Context called sc

```
Using Python version 2.6.6 (r266:84292, Jan 22 2014 09:42:36)
Spark context available as sc.

>>> sc.appName
u'PySparkShell'
```

```
Using Scala version 2.10.3 (Java HotSpot(TM) 64-Bit Server VM, Java
1.7.0_51)
Created spark context..
Spark context available as sc.

scala> sc.appName
res0: String = Spark shell
```

RDD (Resilient Distributed Dataset)

- **RDD (Resilient Distributed Dataset)**
  - **Resilient - if data in memory is lost, it can be recreated**
  - **Distributed - stored in memory across the cluster**
  - **Dataset - initial data can come from a file or be created programmatically**
- **RDDs are the fundamental unit of data in Spark**
- **Most Spark programming consists of performing operations on RDDs**

Creating an RDD

- Three ways to create an RDD
  - From a file or set of files
  - From data in memory
  - From another RDD

File-Based RDDs

- For file-based RDDS, use Spark Context.textFile
  - Accepts a single file, a wildcard list of files, or a comma-separated list of files
  - Examples
    - sc.textFile("myfile.txt")
    - sc.textFile("mydata/*.log")
    - sc.textFile("myfile1.txt,myfile2.txt")
    - Each line in the file(s) is a separate record in the RDD
- Files are referenced by absolute or relative URI
  - Absolute URI: file:/home/training/myfile.txt
  - Relative URI (uses default file system): myfile.txt

RDD Operations

- Two types of RDD operations
  - Actions - return values
  - Transformations- define a new RDD based on the current one(s)
- Pop quiz:
  - Which type of operation is count()?

RDD Operations: Actions

- Some common actions
  - count() - return the number of elements
  - take(n) - return an array of the first n elements
  - collect()- return an array of all elements
  - saveAsTextFile(file) - save to text file(s)

RDD Operations: Transformations

- Transformations create a new RDD from an existing one
- RDDs are immutable
  - Data in an RDD is never changed
  - Transform in sequence to modify the data as needed
- Some common transformations
  - map(function) – creates a new RDD by performing a function on each record in the base RDD
  - filter(function) – creates a new RDD by including or excluding each record in the base RDD according to a boolean function

Example: map and filter Transformations

Lazy Execution

- **Data in RDDs is not processed until an action is performed**

```
> mydata = sc.textFile("purplecow.txt")
> mydata_uc = mydata.map(lambda line:
  line.upper())
> mydata_filt = \
      mydata_uc.filter(lambda line: \
      line.startswith('I'))
> mydata_filt.count()
3
```

Functional Programming in Spark

- Spark depends heavily on the concepts of functional programming
  - Functions are the fundamental unit of programming
  - Functions have input and output only
  - No state or side effects
- Key concepts
  - Passing functions as input to other functions
  - Anonymous functions

Passing Functions as Parameters

- Many RDD operations take functions as parameters
- Pseudocode for the RDD map operation
  - Applies function fn to each record in the RDD

```
RDD {
    map(fn(x)) {
        foreach record in rdd
        emit fn(record)
    }
}
```

Anonymous Functions

- Functions defined in-line without an identifier
  - Best for short, one-off functions
- Supported in many programming languages

Now, please do the following three Hands-On Exercises

1. Viewing the Spark Documentation
2. Using the Spark Shell
3. Getting Started with RDDs

HANDS-ON EXERCISES

Chapter 3

WORKING WITH RDDS

RDDs

- RDDs can hold any type of element
  - Primitive types: integers, characters, booleans, etc.
  - Sequence types: strings, lists, arrays, tuples, dicts, etc. (including nested data types)
  - Scala/Java Objects (if serializable)
  - Mixed types
- Some types of RDDs have additional functionality
  - Pair RDDs
  - RDDs consisting of Key-Value pairs
  - Double RDDs
  - RDDs consisting of numeric data

Creating RDDs From Collections

- You can create RDDs from collections instead of files

```
> randomnumlist = \
    [random.uniform(0,10) for _ in xrange(10000)]
> randomrdd = sc.parallelize(randomnumlist)
> print "Mean: %f" % randomrdd.mean()
```

Some Other General RDD Operations

- Transformations
  - distinct - filter out duplicates
  - union - add all elements of two RDDs into a single new RDD
  - zip - pair the elements of one RDD with another
- Other RDD operations
  - first - return the first element of the RDD
  - for each - apply a function to each element in an RDD
  - top(n) - return the largest n elements using natural ordering
- Sampling operations
  - sample(percent) - create a new RDD with a sampling of elements
  - takeSample (percent) - return an array of sampled elements
- Double RDD operations
  - Statistical functions, e.g. mean, sum, variance, stdev

Pair RDDs

- Pair RDDs are a special form of RDD
  - Each element must be a key-value pair (a two-element tuple)
  - Keys and values can be any type
- Why?
  - Use with Map Reduce algorithms
  - Many additional functions are available for common data processing needs
    - e.g., sorting, joining, grouping, counting, etc.

Creating Pair RDDs

- The first step in most workflows is to get the data into key/value form
  - What should the RDD should be keyed on?
  - What is the value?
- Commonly used functions to create Pair RDDs
  - map
  - flatMap
  - keyBy

Example: Keying Web Logs by User ID

```
> sc.textFile(logfile) \
    .keyBy(lambda line: line.split(' ')[2])
```

```
> sc.textFile(logfile) \
    .keyBy(line => line.split(' ')(2))
```

User ID

```
56.38.234.188 - 99788 "GET /KBDOC-00157.html HTTP/1.0" …
56.38.234.188 - 99788 "GET /theme.css HTTP/1.0" …
203.146.17.59 - 25254 "GET /KBDOC-00230.html HTTP/1.0" …
…
```

```
(99788,56.38.234.188 - 99788 "GET /KBDOC-00157.html…)
(99788,56.38.234.188 - 99788 "GET /theme.css…)
(25254,203.146.17.59 - 25254 "GET /KBDOC-00230.html…)
…
```

Pairs With Complex Values

- **How would you do this?**
  - **Input: a list of postal codes with latitude and longitude**
  - **Output: postal code (key) and lat/long pair (value)**

Pairs With Complex Values

```
> sc.textFile(file) \
  .map(lambda line: line.split()) \
  .map(lambda fields: (fields[0],(fields[1],fields[2])))
```

```
00210   43.005895  -71.013202
00211   43.005895  -71.013202
00212   43.005895  -71.013202
00213   43.005895  -71.013202
00214   43.005895  -71.013202
…
```

```
(00210,(43.005895,-71.013202))
(00211,(43.005895,-71.013202))
(00212,(43.005895,-71.013202))
(00213,(43.005895,-71.013202))
…
```

Mapping Single Rows to Multiple Pairs

- How would you do this?
  - Input: order numbers with a list of SKUs in the order
  - Output: order (key) and sku (value)

Input Data

```
00001   sku010:sku933:sku022
00002   sku912:sku331
00003   sku888:sku022:sku010:sku594
00004   sku411
```

Pair RDD

(00001,sku010)
(00001,sku933)
(00001,sku022)
(00002,sku912)
(00002,sku331)
(00003,sku888)
…

Creating a Pair RDD

```
> sc.textFile(file) \
  .map(lambda line: line.split('\t')) \
  .flatMap(lambda fields:(fields[0],fields[1].split(':')))
```

00001	sku010:sku933:sku022
00002	sku912:sku331
00003	sku888:sku022:sku010:sku594
00004	sku411

(00001,sku010:sku933:sku022)
(00002,sku912:sku331)
(00003,sku888:sku022:sku010:sku594)
(00004,sku411)

(00001,sku010)
(00001,sku933)
(00001,sku022)
(00002,sku912)
(00002,sku331)
(00003,sku888)
…

MapReduce

- MapReduce is a common programming model
  - Easily applicable to distributed processing of large data sets
- Hadoop MapReduce is the major implementation
  - Somewhat limited
    - Each job has one Map phase, one Reduce phase
    - Job output is saved to files
- Spark implements MapReduce with much greater flexibility
  - Map and Reduce functions can be interspersed
  - Results stored in memory
    - Operations can easily be chained

MapReduce in Spark

- MapReduce in Spark works on Pair RDDs
- Map
  - Operates on one record at a time
  - "Maps" each record to one or more new records
  - map and flatMap
- Reduce
  - Works on Map output
  - Consolidates multiple records
  - reduceByKey

Example: Word Count

```
> counts = sc.textFile(file) \
    .flatMap(lambda line: line.split()) \
    .map(lambda word: (word,1)) \
    .reduceByKey(lambda v1,v2: v1+v2)
```

ReduceByKey

ReduceByKey functions must be
- Binary – combines values from two keys
- Commutative – x+y = y+x
- Associative – (x+y)+z = x+(y+z)

```
> counts = sc.textFile(file) \
  .flatMap(lambda line: line.split()) \
  .map(lambda word: (word,1)) \
  .reduceByKey(lambda v1,v2: v1+v2)
```

Word Count Recap (the Scala Version)

```
> val counts = sc.textFile(file) \
  .flatMap(line => line.split("\\W")) \
  .map(word => (word,1)) \
  .reduceByKey((v1,v2) => v1+v2)
```

OR

```
> val counts = sc.textFile(file) \
  .flatMap(_.split("\\W")) \
  .map(_,1)) \
  .reduceByKey(_+_)
```

Why Do We Care About Counting Words?

- Word count is challenging over massive amounts of data
  - Using a single compute node would be too time-consuming
  - Number of unique words could exceed available memory
- Statistics are often simple aggregate functions
  - Distributive in nature
  - e.g., max, min, sum, count
- MapReduce breaks complex tasks down into smaller elements which can be executed in parallel
- Many common tasks are very similar to word count
  - e.g., log file analysis

Pair RDD Operations

- In addition to map and reduce functions, Spark has several operations specific to Pair RDDs
- Examples
  - countByKey-return a map with the count of occurrences of each key
  - groupByKey - group all the values for each key in an RDD
  - sortByKey - sort in ascending or descending order
  - join - return an RDD containing all pairs with matching keys in two RDDs

Example: Pair RDD Operations

Example: Joining by Key

```
> movies = moviegross.join(movieyear)
```

RDD: **moviegross**

(Casablanca,$3.7M)
(Star Wars,$775M)
(Annie Hall,$38M)
(Argo,$232M)
…

RDD: **movieyear**

(Casablanca,1942)
(Star Wars,1977)
(Annie Hall,1977)
(Argo,2012)
…

(Casablanca,[$3.7M,1942])
(Star Wars,[$775M,1977])
(Annie Hall,[$38M,1977])
(Argo,[$232M,2012])
…

Using Join

- A common programming pattern
  - 1. Map separate datasets into key-value Pair RDDs
  - 2. Join by key
  - 3. Map joined data into the desired format
  - 4. Save, display, or continue processing…

Map Web Log Requests to (docid,userid)

```
> import re
> def getRequestDoc(s):
   return re.search(r'KBDOC-[0-9]*',s).group()

> kbreqs = sc.textFile(logfile) \
    .filter(lambda line: 'KBDOC-' in line) \
    .map(lambda line: (getRequestDoc(line),line.split(' ')[2])) \
    .distinct()
```

```
56.38.234.188 - 99788 "GET /KBDOC-00157.html HTTP/1.0" …
56.38.234.188 - 99788 "GET /theme.css HTTP/1.0" …
203.146.17.59 - 25254 "GET /KBDOC-00230.html HTTP/1.0"
221.78.60.155 - 45402 "GET /titanic_4000_sales.html
65.187.255.81 - 14242 "GET /KBDOC-00107.html HTTP/1.
…
```

kbreqs

(KBDOC-00157,99788)
(KBDOC-00203,25254)
(KBDOC-00107,14242)
…

Map KB Index to (docid,userid)

```
> kblist = sc.textFile(kblistfile) \
  .map(lambda line: line.split(':')) \
  .map(lambda fields: (fields[0],fields[1]))
```

```
KBDOC-00157:Ronin Novelty Note 3 - Back up files
KBDOC-00230:Sorrento F33L - Transfer Contacts
KBDOC-00050:Titanic 1000 - Transfer Contacts
KBDOC-00107:MeeToo 5.0 - Transfer Contacts
KBDOC-00206:iFruit 5A - overheats
…
```

kblist

```
(KBDOC-00157,Ronin Novelty Note 3 - Back up files)
(KBDOC-00230,Sorrento F33L - Transfer Contacts)
(KBDOC-00050,Titanic 1000 - Transfer Contacts)
(KBDOC-00107,MeeToo 5.0 - Transfer Contacts)
…
```

Join By Key docid

```
> titlereqs = kbreqs.join(kblist)
```

kbreqs

(KBDOC-00157,99788)
(KBDOC-00230,25254)
(KBDOC-00107,14242)
…

kblist

(KBDOC-00157,Ronin Novelty Note 3 - Back up files)
(KBDOC-00230,Sorrento F33L - Transfer Contacts)
(KBDOC-00050,Titanic 1000 - Transfer Contacts)
(KBDOC-00107,MeeToo 5.0 - Transfer Contacts)
…

(KBDOC-00157,(99788,Ronin Novelty Note 3 - Back up files))
(KBDOC-00230,(25254,Sorrento F33L - Transfer Contacts))
(KBDOC-00107,(14242,MeeToo 5.0 - Transfer Contacts))
…

Map Result to Desired Format (userid,title)

```
> titlereqs = kbreqs.join(kblist) \
    .map(lambda (docid,(userid,title)): (userid,title))
```

(KBDOC-00157,(99788,Ronin Novelty Note 3 - Back up files))
(KBDOC-00230,(25254,Sorrento F33L - Transfer Contacts))
(KBDOC-00107,(14242,MeeToo 5.0 - Transfer Contacts))
…

(99788,Ronin Novelty Note 3 - Back up files)
(25254,Sorrento F33L - Transfer Contacts)
(14242,MeeToo 5.0 - Transfer Contacts)
…

Continue Processing- Group by User ID

```
> titlereqs = kbreqs.join(kblist) \
    .map(lambda (docid,(userid,title)): (userid,title)) \
    .groupByKey()
```

```
(99788,Ronin Novelty Note 3 - Back up files)
(25254,Sorrento F33L - Transfer Contacts)
(14242,MeeToo 5.0 - Transfer Contacts)
…
```

```
(99788,[Ronin Novelty Note 3 - Back up files,
        Ronin S3 - overheating])
(25254,[Sorrento F33L - Transfer Contacts])
(14242,[MeeToo 5.0 - Transfer Contacts,
        MeeToo 5.1 - Back up files,
        iFruit 1 - Back up files,
        MeeToo 3.1 - Transfer Contacts])
…
```

Example Output

```
> for (userid,titles) in titlereqs.take(10):
    print 'user id: ',userid
    for title in titles: print '\t',title

user id: 99788
  Ronin Novelty Note 3 - Back up files
  Ronin S3 - overheating
user id: 25254
  Sorrento F33L - Transfer Contacts
user id: 14242
  MeeToo 5.0 - Transfer Contacts
  MeeToo 5.1 - Back up files
  iFruit 1 - Back up files
  MeeToo 3.1 - Transfer Contacts
…
```

```
(99788,[Ronin Novelty Note 3 - Back up files,
        Ronin S3 - overheating])
(25254,[Sorrento F33L - Transfer Contacts])
(14242,[MeeToo 5.0 - Transfer Contacts,
        MeeToo 5.1 - Back up files,
        iFruit 1 - Back up files,
        MeeToo 3.1 - Transfer Contacts])
…
```

Aside: Anonymous Function Parameters

Python

```
> map(lambda (docid,(userid,title)): (userid,title))
```

Scala

```
> map(pair => (pair._2._1,pair._2._2))
```

OR

```
> map{case (docid,(userid,title)) => (userid,title)}
```

Other Pair Operations

- Some other pair operations
  - keys - return an RDD of just the keys, without the values
  - values - return an RDD of just the values, without keys
  - lookup(key)
  - leftOuterJoin, rightOuterJoin - join, including keys defined only in the left or right RDDs respectively
  - mapValues, flatMapValues - execute a function on just the values, keeping the key the same

- See the PairRDDFunctions class Scaladoc for a full list

HANDS-ON EXERCISE: WORKING WITH PAIR RDDS

Chapter 4

THE HADOOP DISTRIBUTED FILE SYSTEM (HDFS)

Distributed Processing with the Spark Framework

Big Data Processing with Spark

- Three key concepts
  - Distribute data when the data is stored - HDFS
  - Run computation where the data is - HDFS and Spark
  - Cache data in memory - Spark

HDFS Basic Concepts

- HDFS is a filesystem written in Java
  - Based on Google's GFS
- Sits on top of a native filesystem
- Provides redundant storage for massive amounts of data
  - Using readily-available, industry-standard computers

HDFS Basic Concepts

- HDFS performs best with a 'modest' number of large files
  - Millions, rather than billions, of files
  - Each file typically 100MB or more
- Files in HDFS are 'write once'
  - No random writes to files are allowed
- HDFS is optimized for large, streaming reads of files
  - Rather than random reads

How Files Are Stored

- **NameNode stores metadata**

Storing and Retrieving Files

Storing and Retrieving Files

Storing and Retrieving Files

Storing and Retrieving Files

HDFS NameNode Availability

- The NameNode daemon must be running at all times
  - If the NameNode stops, the cluster becomes inaccessible
- HDFS is typically set up for High Availability
  - Two NameNodes: Active and Standby"
- Small clusters may use ‘Classic mode’
  - One NameNode
  - One “helper” node called the Secondary NameNode
  - Bookkeeping, not backup

Options for Accessing HDFS

- From the command line
  - FsShell: hdfs dfs"
- In Spark
  - By URI, e.g. hdfs://host:port/file…
- Other programs
  - Java API
  - Used by Hadoop MapReduce,
  - Hue, Sqoop,
  - Flume, etc.
  - RESTful interface

hdfs dfs Examples

```
$ hdfs dfs -put foo.txt foo.txt
```

- This will copy the file to `/user/`*`username`*`/foo.txt`

- **Get a directory listing of the user's home directory in HDFS**

```
$ hdfs dfs -ls
```

- **Get a directory listing of the HDFS root directory**

```
$ hdfs dfs -ls /
```

hdfs dfs Examples

```
$ hdfs dfs -cat /user/fred/bar.txt
```

- **Copy that file to the local disk, named as `baz.txt`**

```
$ hdfs dfs -get /user/fred/bar.txt baz.txt
```

- **Create a directory called `input` under the user's home directory**

```
$ hdfs dfs -mkdir input
```

Note: copyFromLocal is a synonym for put; copyToLocal is a synonym for get

Example: HDFS in Spark

- **Specify HDFS files in Spark by URI**
  - **hdfs://hdfs-host[:port]/path**
  - **Default port is 8020**

```
> mydata = sc.textFile \
  ("hdfs://hdfs-host:port/user/training/purplecow.txt")


> mydata.map(lambda s: s.upper()).\
  saveAsTextFile \
  ("hdfs://hdfs-host:port/user/training/purplecowuc")
```

Using HDFS By Default

- If Hadoop configuration files are on Spark's classpath, Spark will use HDFS by default
  - e.g. /etc/hadoop/conf
- Paths are relative to user's home HDFS directory

HANDS-ON EXERCISE: USING HDFS

Chapter 5

SPARK ON A CLUSTER

Spark Cluster Options

- Spark can run
  - Locally
    - No distributed processing
    - Locally with multiple worker threads
  - On a cluster
    - Spark Standalone
    - Apache Hadoop YARN (Yet Another Resource Negotiator)

Distributed Processing with the Spark Framework

The Spark Driver Program

- A Spark Driver
  - The "main" program
    - Either the Spark Shell or a Spark application
  - Creates a Spark Context configured for the cluster
  - Communicates with Cluster Manager to distribute tasks to executors

Starting the Spark Shell on a Cluster

- Set the Spark Shell master to
  - url - the URL of the cluster manager
  - local[*] - run with as many threads as cores (default)
  - local[n] - run locally with n worker threads
  - local - run locally without distributed processing
- This configures the SparkContext.master property

Python

```
$ MASTER=spark://masternode:7077 pyspark
```

Scala

```
$ spark-shell --master spark://masternode:7077
```

Spark Standalone Daemons

- Spark Standalone daemons
  - Spark Master
    - One per cluster
    - Manages applications, distributes individual tasks to Spark Workers
  - Spark Worker
    - One per worker node
    - Starts and monitors Executors for applications

Running Spark on a Standalone Cluster

Running Spark on a Standalone Cluster

Spark Standalone Web UI

Spark Standalone clusters offer a Web UI to monitor the cluster

- http://masternode:uiport
- e.g., in our class environment, http://localhost:18080

Spark Standalone Web UI: Application Overview

Spark Standalone Web UI: Worker Detail

Supported Cluster Resource Managers

- Spark Standalone
  - Included with Spark
  - Easy to install and run
  - Limited configurability and scalability
  - Useful for testing, development, or small systems
- Hadoop YARN
  - Included in HDP
  - Most common for production sites
  - Allows sharing cluster resources with other applications (MapReduce, etc.)

Client Mode and Cluster Mode

- By default, the driver program runs outside the cluster
  - Called "client" deploy mode
  - Most common
  - Required for interactive use (e.g., the Spark Shell)
- It is also possible to run the driver program on a worker node in the cluster
  - Called cluster deploy mode

HANDS-ON EXERCISE: RUNNING SPARK ON A CLUSTER