Parallel Computing with Apache Spark

Apache Spark is a computing framework for large-scale parallel processing

- Developed by UC Berkeley AMPLab (Now RISELab)
- now maintained by Apache Foundation

Implementations are available in Java, Scala and Python (and R, sort of) and these can be run interactively!

Easily communicates with several other “big data” Apache tools e.g., Hadoop, Mesos, HBase
Can also be run locally or in the cloud (e.g., Amazon EC2)

https://spark.apache.org/docs/0.9.0/index.html
Why use Spark?

“Wait, doesn’t Hadoop/mrjob already do all this stuff?”

Short answer: yes!

Less short answer: Spark is faster and more flexible than Hadoop

and since Spark looks to be eclipsing Hadoop in industry, it is my responsibility to teach it to you!

Spark still follows the MapReduce framework, but is better suited to:

- Interactive sessions
- Caching (i.e., data is stored in RAM on the nodes where it is to be processed, not on disk)
- Repeatedly updating computations (e.g., updates as new data arrive)
- Fault tolerance and recovery
Apache Spark: Overview

Implemented in Scala
- Popular functional programming (sort of…) language
- Runs atop Java Virtual Machine (JVM)
  
  http://www.scala-lang.org/

But Spark can be called from Scala, Java and Python
and from R using SparkR: https://spark.apache.org/docs/latest/sparkr.html

We’ll do all our coding in Python
PySpark: https://spark.apache.org/docs/0.9.0/python-programming-guide.html
but everything you learn can be applied with minimal changes in other supported languages
Running Spark

Option 1: Run in interactive mode
   Type `pyspark` on the command line
   PySpark provides an interface similar to the Python interpreter
   Like what you get when you type `python` on the command line
   Scala, Java and R also provide their own interactive modes

Option 2: Run on a cluster
   Write your code, then launch it via a scheduler
   `spark-submit`
   `https://spark.apache.org/docs/latest/submitting-applications.html#launching-applications-with-spark-submit`
   Similar to running Python `mrjob` scripts with the `-r hadoop` flag
Two Basic Concepts

SparkContext
Object corresponding to a connection to a Spark cluster
   Automatically created in interactive mode
   Must be created explicitly when run via scheduler (We’ll see an example soon)
Stores information about where data is stored
Allows configuration by supplying a SparkConf object

Resilient Distributed Dataset (RDD)
Represents a collection of data
Distributed across nodes in a fault-tolerant way (much like HDFS)
More about RDDs

RDDs are the basic unit of Spark

“a collection of elements partitioned across the nodes of the cluster that can be operated on in parallel.” ([https://spark.apache.org/docs/0.9.0/scala-programming-guide.html#overview](https://spark.apache.org/docs/0.9.0/scala-programming-guide.html#overview))

Elements of an RDD are analogous to <key,value> pairs in MapReduce

RDD is roughly analogous to a dataframe in R

RDD elements are somewhat like rows in a table

Spark can also keep (**persist**, in Spark’s terminology) an RDD in memory

Allows reuse or additional processing later

RDDs are **immutable**, like Python tuples and strings.
RDD operations

Think of RDD as representing a data set

Two basic operations:

**Transformation:** results in another RDD
(e.g., map takes an RDD and applies some function to every element of the RDD)

**Action:** computes a value and reports it to driver program
(e.g., reduce takes all elements and computes some summary statistic)
RDD operations are lazy!

**Transformations** are only carried out once an **action** needs to be computed.

Spark remembers the sequence of transformations to run...
...but doesn’t execute them until it has to
  e.g., to produce the result of a reduce operation for the user.

This allows for gains in efficiency in some contexts
mainly because it avoids expensive intermediate computations
Okay, let’s dive in!

[klevin@flux-hadoop-login1 ~]$ pyspark
Python 2.7.5 (default, Nov  6 2016, 00:28:07)
[GCC 4.8.5 20150623 (Red Hat 4.8.5-11)] on linux2
Type "help", "copyright", "credits" or "license" for more information.
[...a bunch of boot-up information...]
Welcome to

/\_/\           /\_/\                    \/   \                     \\        /\_/\ \_/\ \\
_/\_/\           /\_/\                    /\_/\ \\
  \\

Using Python version 2.7.5 (default, Nov  6 2016 00:28:07)
SparkContext available as sc, HiveContext available as sqlContext.
>>>
Okay, let’s dive in!

There will be a lot of information (multiple screens’ worth!) here about establishing a Spark session. You can safely ignore this information, for now, but if you're running your own Spark cluster this is where you'll need to look when it comes time to troubleshoot.

Spark finishes setting up our interactive session and gives us a prompt like the Python interpreter.
Creating an RDD from a file

Welcome to

  /  __/__  ___  ____/ /__
 /  _\_\  _\ `\ _\_\ _\/_
/   /_/ /_/ /_/ /_/

version 1.6.0

Using Python version 2.7.5 (default, Nov 6 2016 00:28:07)
SparkContext available as sc, HiveContext available as sqlContext.

```python
>>> sc
<pyspark.context.SparkContext object at 0x2d73350>
```

```python
>>> data = sc.textFile('/var/stat700002f17/demo_file.txt')
```

```python
>>> data.collect()
[u'This is just a demo file.', u'Normally, a file this small would have no reason to be on HDFS.]
```
Welcome to  

Welcome to Spark version 1.6.0

Using Python version 2.7.5 (default, Nov 6 2016 00:28:07)
SparkContext available as sc, HiveContext available as sqlContext.

```python
>>> sc
< pyspark.context.SparkContext object at 0x2d73350>
```

```python
>>> data = sc.textFile('/var/stat700002f17/demo_file.txt')
```

```python
>>> data.collect()
[u'\nThis is just a demo file.', u'\nNormally, a file this small would have no reason to be on HDFS.\n']
```

SparkContext is automatically created by the PySpark interpreter, and saved in the variable `sc`. When we write a job to be run on the cluster, we will have to define `sc` ourselves.

This creates an RDD from the given file. PySpark assumes that we are referring to a file on HDFS.

Our first RDD action, `collect()`, gathers the elements of the RDD into a list.
PySpark keeps track of RDDs

Welcome to

```
/ __/__  ___  ____/ /__
\ \/ _ / _ `/ __/  '_/
/__ / .__/_/ /_/
_/ /
```

version 1.6.0

Using Python version 2.7.5 (default, Nov  6 2016 00:28:07)
SparkContext available as sc, HiveContext available as sqlContext.

```python
>>> data = sc.textFile('/var/stat700002f17/demo_file.txt')
>>> data
/var/stat700002f17/demo_file.txt MapPartitionsRDD[1] at textFile at NativeMethodAccessorImpl.java:-2
```
PySpark keeps track of RDDs

Using Python version 2.7.5 (default, Nov 6 2016 00:28:07)
SparkContext available as sc, HiveContext available as sqlContext.
>>> data = sc.textFile('/var/stat700002f17/demo_file.txt')
>>> data
/var/stat700002f17/demo_file.txt MapPartitionsRDD[1] at textFile at
NativeMethodAccessorImpl.java:-2

PySpark keeps track of where the original data resides. MapPartitionsRDD is like an array of all the RDDs that we’ve created (though it’s not a variable you can access).
Simple MapReduce task: Summations

I have a file containing some numbers. Let's add them up using PySpark.
Simple MapReduce task: Summations

Using Python version 2.7.5 (default, Nov 6 2016 00:28:07)
SparkContext available as sc, HiveContext available as sqlContext.

```python
>>> data = sc.textFile('/user/klevin/numbers.txt')
>>> data.collect()
[u'10', u'23', u'16', u'7', u'12', u'0', u'1', u'1', u'2', u'3', u'5', u'8', u'-1', u'42', u'64', u'101', u'-101', u'3']
```

Using `strip()` here is redundant--PySpark automatically splits on whitespace when it reads from a textfile. This is again just to show an example.

**Reminder:** `collect()` is an RDD action that produces a list of the RDD elements.
Simple MapReduce task: Summations

```python
>>> data = sc.textFile('/var/stat700002f17/numbers.txt')
>>> data = data.map(lambda line: line.strip())
>>> intdata = data.map(lambda n: int(n))
>>> intdata.reduce(lambda x,y: x+y)
196
```
Simple MapReduce task: Summations

```python
>>> data = sc.textFile('/var/stat700002f17/numbers.txt')
>>> data = data.map(lambda line: line.strip())
>>> intdata = data.map(lambda n: int(n))
>>> intdata.reduce(lambda x,y: x+y)
```

PySpark doesn't actually perform any computations on the data until this line.

Test your understanding:
Why is this the case?
Simple MapReduce task: Summations

```python
>>> data = sc.textFile('/var/stat700002f17/numbers.txt')
>>> data = data.map(lambda line: line.strip())
>>> intdata = data.map(lambda n: int(n))
>>> intdata.reduce(lambda x,y: x+y)
196
```

PySpark doesn't actually perform any computations on the data until this line.

**Test your understanding:**

Why is this the case?

**Answer:** Because PySpark RDD operations are lazy! PySpark doesn't perform any computations until we actually ask it for something via an *RDD action*. 
Simple MapReduce task: Summations

```python
>>> data = sc.textFile('/var/stat700002f17/numbers.txt')
>>> data = data.map(lambda line: line.strip())
>>> intdata = data.map(lambda n: int(n))
>>> intdata.reduce(lambda x,y: x+y)
196
```
Simple MapReduce job: Summations

```python
>>> data = sc.textFile('/var/stat700002f17/numbers.txt')
>>> data = data.map(lambda line: line.strip())
>>> intdata = data.map(lambda n: int(n))
>>> intdata.reduce(lambda x,y: x+y)
196
```
Getting help

>>> help()

Welcome to Python 2.7!  This is the online help utility.

If this is your first time using Python, you should definitely check out the tutorial on the Internet at http://docs.python.org/2.7/tutorial/.

Enter the name of any module, keyword, or topic to get help on writing Python programs and using Python modules. To quit this help utility and return to the interpreter, just type "quit".

To get a list of available modules, keywords, or topics, type "modules", "keywords", or "topics". Each module also comes with a one-line summary of what it does; to list the modules whose summaries contain a given word such as "spam", type "modules spam".

help>
Getting help

>>> help(map)
Help on built-in function map in module __builtin__:

map(...)
    map(function, sequence[, sequence, ...]) -> list

Return a list of the results of applying the function to the items of the argument sequence(s). If more than one sequence is given, the function is called with an argument list consisting of the corresponding item of each sequence, substituting None for missing values when not all sequences have the same length. If the function is None, return a list of the items of the sequence (or a list of tuples if more than one sequence).

(END)

Or ask about a specific function or object with help(object)
Example RDD Transformations

map: apply a function to every element of the RDD

filter: retain only the elements satisfying a condition

flatMap: apply a map, but “flatten” the structure (details in a few slides)

sample: take a random sample from the elements of the RDD

distinct: remove duplicate entries of the RDD

reduceByKey: on RDD of (K, V) pairs, return RDD of (K, V) pairs
values for each key are aggregated using the given reduce function.

More: https://spark.apache.org/docs/0.9.0/scala-programming-guide.html#transformations
```python
def polynomial(x):
    return x**2 + 1
```
RDD.map()

```python
>>> data = sc.textFile('/var/stat700002f17/numbers.txt')
>>> data = data.map(lambda n: int(n))
>>> data.collect()
[10, 23, 16, 7, 12, 0, 1, 1, 2, 3, 5, 8, -1, 42, 64, 101, -101, 3]

>>> sc.addPyFile('poly.py')
[...status messages redacted...]

>>> from poly import *

>>> data.map(polynomial).collect()
[...status messages redacted...]
[101, 530, 257, 50, 145, 1, 2, 2, 5, 10, 26, 65, 2, 1765, 4097, 10202, 10202, 10]
```

**poly.py**

```python
1    def polynomial(x):
2        return x**2 + 1
```
**RDD.map()**

```python
>>> data = sc.textFile('/var/stat700002f17/numbers.txt')
>>> data = data.map(lambda n: int(n))
>>> data.collect()
[10, 23, 16, 7, 12, 0, 1, 1, 2, 3, 5, 8, -1, ...]
>>> sc.addPyFile('poly.py')
[...status messages redacted...]
>>> from poly import *
>>> data.map(polynomial).collect()
[...status messages redacted...]
[101, 530, 257, 50, 145, 1, 2, 2, 5, 10, 26, 65, 2, 1765, 4097, 10202, 10202, 10]
```

*map() takes a function as an argument, whether that function is defined elsewhere or simply by a lambda expression.*

### poly.py

```python
1    def polynomial(x):
2        return x**2 + 1
```

*This file is saved in the directory where I launched *pyspark*. If it's somewhere else, we have to specify the path to it.*
 RDD.filter()

```python
>>> data = sc.textFile('/var/stat700002f17/numbers.txt').map(lambda n: int(n))
>>> evens = data.filter(lambda n: n%2==0)
>>> evens.collect()
[...output messages redacted...]
[10, 16, 12, 0, 2, 8, 42, 64]
>>> odds = data.filter(lambda n: n%2!=0)
>>> odds.collect()
[...output messages redacted...]
[23, 7, 1, 1, 3, 5, -1, 101, -101, 3]
>>> sc.addPyFile('prime.py')
>>> from prime import is_prime
>>> primes = data.filter(is_prime)
>>> primes.collect()
[23, 7, 3, 5, 101, 3]
```

`filter()` takes a Boolean function as an argument, and retains only the elements that evaluate to true.
RDD.sample()

```python
>>> data = sc.textFile('/var/stat700002f17/numbers.txt').map(lambda n: int(n))
>>> samp = data.sample(False, 0.5)
>>> samp.collect()
[16, 7, 0, 1, -101, 3]
>>> samp = data.sample(True, 0.5)
>>> samp.collect()
[10, 12, 8, 8, 8, -101, -101]
```

sample(withReplacement, fraction, seed)
last argument is optional.

RDD.sample() is mostly useful for testing on small subsets of your data.
Dealing with more complicated elements

What if the elements of my RDD are more complicated than just numbers?...

Example: if I have a comma-separated database-like file

Short answer: RDD elements are always tuples

But what about really complicated elements?
Recall that PySpark RDDs are immutable. This means that if you want your RDD to contain, for example, python dictionaries, you need to do a bit of extra work to turn Python objects into strings. This is called serialization. We won’t use it in the homework, but it’s good to be aware of it: https://docs.python.org/2/library/pickle.html
Database-like file

[klewin@flux-hadoop-login1 pyspark]$ hdfs dfs -cat hdfs:///var/stat700002f17/scientists.txt
Claude Shannon 3.1 EE 1916
Eugene Wigner 3.2 Physics 1902
Albert Einstein 4.0 Physics 1879
Ronald Fisher 3.25 Statistics 1890
Max Planck 2.9 Physics 1858
Leonard Euler 3.9 Mathematics 1707
Jerzy Neyman 3.5 Statistics 1894
Ky Fan 3.55 Mathematics 1914
[klewin@flux-hadoop-login1 pyspark]$
Database-like file

```python
>>> data = sc.textFile('/var/stat700002f17/scientists.txt')
>>> data.collect()
>>> data = data.map(lambda line: line.split())
>>> data.collect()
[[u'Claude', u'Shannon', u'3.1', u'EE', u'1916'], [u'Eugene', u'Wigner', u'3.2', u'Physics', u'1902'], [u'Albert', u'Einstein', u'4.0', u'Physics', u'1879'], [u'Ronald', u'Fisher', u'3.25', u'Statistics', u'1890'], [u'Max', u'Planck', u'2.9', u'Physics', u'1858'], [u'Leonard', u'Euler', u'3.9', u'Mathematics', u'1707'], [u'Jerzy', u'Neyman', u'3.5', u'Statistics', u'1894'], [u'Ky', u'Fan', u'3.55', u'Mathematics', u'1914']]```
On initial read, each line is a single element in the RDD.

After splitting each element on whitespace, we have what we want-- each element is a tuple of strings.

Note: RDD.collect() returns a list, but internal to the RDD, the elements are tuples, not lists.
RDD.distinct()

```python
>>> data = sc.textFile('/var/stat700002f17/scientists.txt')
>>> data = data.map(lambda line: line.split())
>>> fields = data.map(lambda t: t[3]).distinct()
>>> fields.collect()
[u'EE', u'Physics', u'Mathematics', u'Statistics']
```
```python
>>> data = sc.textFile('/var/stat700002f17/scientists.txt')
>>> data = data.map(lambda line: line.split())
>>> fields = data.map(lambda t: t[3]).distinct()
>>> fields.collect()
[u'EE', u'Physics', u'Mathematics', u'Statistics']
```

Each tuple is of the form (first_name, last_name, GPA, field, birth_year)

RDD.distinct() does just what you think it does!
RDD.flatMap()

```python
>>> data = sc.textFile('/var/stat700002f17/numbers_rows.txt')
>>> data.collect()
[u'10 23 16', u'7 12', u'0', u'1 1 2 3 5 8', u'-1 42', u'64 101 -101', u'3']
```

Same list of numbers, but they’re not one per line, anymore...

From PySpark documentation:

**flatMap** (*func*) Similar to map, but each input item can be mapped to 0 or more output items (so *func* should return a Seq rather than a single item).

[https://spark.apache.org/docs/latest/rdd-programming-guide.html#transformations](https://spark.apache.org/docs/latest/rdd-programming-guide.html#transformations)
So we can think of `flatMap()` as producing a list for each element in the RDD, and then appending those lists together. But crucially, the output is another RDD, **not** a list. This kind of operation is called **flattening**, and it’s a common pattern in functional programming.
Example RDD Actions

reduce: aggregate elements of the RDD using a function

collect: return all elements of the RDD as an array at the driver program.

count: return the number of elements in the RDD.

countByKey: Returns <key, int> pairs with count of each key.
          Only available on RDDs with elements of the form <key,value>

More: https://spark.apache.org/docs/0.9.0/scala-programming-guide.html#actions
>>> data = sc.textFile('/var/stat700002f17/demo_file.txt')
>>> data = data.flatMap(lambda line:line.split())
>>> data = data.map(lambda w: w.lower())
>>> data.collect()
[u'this', u'is', u'just', u'a', u'demo', u'file.', u'normally,', u'a', u'file', u'this', u'small', u'would', u'have', u'no', u'reason', u'to', u'be', u'on', u'hdfs. ']
>>> uniqwords = data.distinct()
>>> uniqwords.count()
17
>>> data = sc.textFile('/var/stat700002f17/demo_file.txt')
>>> data = data.flatMap(lambda line: line.split())
>>> data = data.map(lambda w: (w.lower(),1))
>>> data.countByKey()
defaultdict(<type 'int'>, {u'a': 2, u'be': 1, u'file': 1, u'hdfs.': 1, u'would': 1, u'just': 1, u'no': 1, u'this': 2, u'demo': 1, u'is': 1, u'to': 1, u'reason': 1, u'have': 1, u'small': 1, u'normally,': 1, u'on': 1, u'file.': 1})
>>>
Running PySpark on the Cluster

So far, we’ve just been running in interactive mode.

**Problem:** Interactive mode is good for prototyping and testing…
...but not so well-suited for running large jobs.

**Solution:** PySpark can also be submitted to the grid and run there. Instead of `pyspark`, we use `spark-submit` on the Fladoop grid. Instructions specific to Fladoop can be found here:

Two preliminaries

Before we can talk about running jobs on the cluster...

1) **UNIX groups**
   How we control who can and can’t access files

2) **Queues on compute clusters**
   How we know who has to pay for compute time
UNIX Groups

On UNIX-like systems, files are owned by users

Sets of users, called **groups**, can be granted special permissions

On UNIX/Linux/MacOS:

```
[klevin@flux-hadoop-login1 pyspark]$ ls -l ..
total 166
drwxr-xr-x 2 klevin statistics  25 Oct 27 12:07 hadoop_stuff
-rw-r--r-- 1 klevin statistics  29 Oct 27 12:09 homework2.tex
drwxr-xr-x 2 klevin statistics 217 Nov 11 16:38 HW3
-rw-r--r-- 1 klevin statistics  0 Oct 27 10:59 hw3.tex
drwxr-xr-x 2 klevin statistics 131 Nov 13 10:31 mrjob_demo
-rw-r--r-- 1 klevin statistics  14 Oct 27 12:22 myfile.txt
drwxr-xr-x 3 klevin statistics 335 Nov 16 12:19 pyspark
```
UNIX Groups

On UNIX-like systems, files are owned by users.

Sets of users, called **groups**, can be granted special permissions.

On UNIX/Linux/MacOS:

```
[klevin@flux-hadoop-login1 pyspark]$ ls -l ..
total 166
drwxr-xr-x 2 klevin statistics  25 Oct 27 12:07 hadoop_stuff
-rwr-r--r-- 1 klevin statistics  29 Oct 27 12:09 homework2.tex
drwxr-xr-x 2 klevin statistics 217 Nov 11 16:38 HW3
-rwr-r--r-- 1 klevin statistics  0 Oct 27 12:59 hw3.tex
drwxr-xr-x 2 klevin statistics 131 Nov 13 10:31 mrjob_demo
-rwr-r--r-- 1 klevin statistics  14 Oct 27 12:22 myfile.txt
drwxr-xr-x 3 klevin statistics 335 Nov 16 12:19 pyspark
```

These lines are permission information.

---

**Legend**

- `d`: directory
- `r`: read access
- `w`: write access
- `x`: execute access
UNIX Groups

On UNIX-like systems, files are owned by users.

Sets of users, called **groups**, can be granted special permissions.

On UNIX/Linux/MacOS:

```
[klevin@flux-hadoop-login1 pyspark]$ ls -l ..
total 166
  drwxr-xr-x 2 klevin statistics 25 Oct 27 12:07 hadoop_stuff
  -rw-r--r-- 1 klevin statistics 29 Oct 27 12:09 homework2.tex
  drwxr-xr-x 2 klevin statistics 317 Nov 11 16:38 HW3
  -rw-r--r-- 1 klevin statistics 10 Oct 27 10:59 hw3.tex
  drwxr-xr-x 2 klevin statistics 131 Nov 13 10:31 mrjob_demo
  -rw-r--r-- 1 klevin statistics 14 Oct 27 12:22 myfile.txt
  drwxr-xr-x 3 klevin statistics 335 Nov 16 12:19 pyspark
```

This column lists what group owns the file.

**Legend**

- `d`: directory
- `r`: read access
- `w`: write access
- `x`: execute access
UNIX Groups

On UNIX-like systems, files are owned by users

Sets of users, called **groups**, can be granted special permissions

On UNIX/Linux/MacOS:

```
[klevin@flux-hadoop-login1 pyspark]$ ls -l ..
total 166
drwxr-xr-x 2 klevin statistics  25 Oct 27 12:07 hadoop_stuff
-rw-r--r-- 1 klevin statistics  29 Oct 27 12:09 homework2.tex
drwxr-xr-x 2 klevin statistics 217 Nov 11 16:38 HW3
-rw-r--r-- 1 klevin statistics   0 Oct 27 10:59 hw3.tex
```

These specific columns specify group permissions. Anyone in the statistics group has these permissions on these files.
Cluster computing: queues

Compute cluster is a shared resource

How do we know who has to pay for what?

Flux operates what are called allocations, which are like pre-paid accounts

When you submit a job, you submit to a queue
   Like a line that you stand in to wait for your job to be run
   One line for each class, lab, etc

Our class has its own queue: stat700-002-f17
Submitting to the queue: spark-submit

```python
from pyspark import SparkConf, SparkContext
import sys

# This script takes two arguments, an input and output
if len(sys.argv) != 3:
    print('Usage: ' + sys.argv[0] + ' <in> <out> ')
    sys.exit(1)
inputlocation = sys.argv[1]
outputlocation = sys.argv[2]

# Set up the configuration and job context
conf = SparkConf().setAppName('Summation')
sc = SparkContext(conf=conf)

# Read in the dataset and immediately transform all the lines in arrays
data = sc.textFile(inputlocation)
data = data.flatMap(lambda line: line.split())
data = data.map(lambda w: (w.lower(), 1))
data = data.reduceByKey(lambda x, y: x+y)

# Save the results in the specified output directory.
data.saveAsTextFile(outputlocation)
sc.stop()  # Let Spark know that the job is done.
```
Submitting to the queue: `spark-submit`

```python
from pyspark import SparkConf, SparkContext
import sys

# This script takes two arguments, an input and output path
if len(sys.argv) != 3:
    print('Usage: ' + sys.argv[0] + ' <in> <out>"
    sys.exit(1)
inputlocation = sys.argv[1]
outputlocation = sys.argv[2]

# Set up the configuration and job context
conf = SparkConf().setAppName('Summation')
sc = SparkContext(conf=conf)

# Read in the dataset and immediately transform all the lines in arrays
data = sc.textFile(inputlocation)
data = data.flatMap(lambda line: line.split())
data = data.map(lambda w: (w.lower(), 1))
data = data.reduceByKey(lambda x, y: x+y)

# Save the results in the specified output directory.
data.saveAsTextFile(outputlocation)
sc.stop()  # Let Spark know that the job is done.
```

We’re not in an interactive session, so the SparkContext isn’t set up automatically. SparkContext is set up using a SparkConf object, which specifies configuration information. For our purposes, it’s enough to just give it a name, but in general there is a lot of information we can pass via this object.
Submitting to the queue: spark-submit

[klevin@flux-hadoop-login1 pyspark]$ spark-submit --master yarn-client --queue stat700-002-f17 ps_wordcount.py /var/stat700002f17/demo_file.txt
wc_demo
[...lots of status information from Spark...]
[klevin@flux-hadoop-login1 pyspark]$ hdfs dfs -ls wc_demo/
Found 3 items
-rw-r-----   3 klevin hadoop   0 2017-11-16 15:36 wc_demo/_SUCCESS
-rw-r-----   3 klevin hadoop  130 2017-11-16 15:36 wc_demo/part-00000
-rw-r-----   3 klevin hadoop  89 2017-11-16 15:36 wc_demo/part-00001
[klevin@flux-hadoop-login1 pyspark]$ hdfs dfs -cat wc_demo/*
(u'a', 2)
(u'file', 1)
(u'hdfs.', 1)
[...]
(u'file.', 1)
[klevin@flux-hadoop-login1 pyspark]$
Submitting to the queue: `spark-submit`

Specifying the master and the queue are both mandatory, but there are other additional options we could supply. Most importantly:
- `--num-executors 35`
- `--executor-memory 5g`
- `--executor-cores 4`

More: [https://spark.apache.org/docs/latest/submitting-applications.html](https://spark.apache.org/docs/latest/submitting-applications.html)
Submitting to the queue: `spark-submit`

Larger-scale example (runs on all of Google ngrams):

**Warning**: make sure you provide enough executors or this will take a long time!
Shared Variables

Spark supports shared variables!

Allows for (limited) communication between parallel jobs

Two types:

- **Broadcast variables**: used to communicate value to all nodes

- **Accumulators**: nodes can only “add”
  - (or multiply, or… any operation on a monoid)

https://en.wikipedia.org/wiki/Monoid
https://spark.apache.org/docs/latest/rdd-programming-guide.html#accumulators
Shared Variables

Spark supports shared variables!

Allows for (limited) communication between parallel jobs

Two types:

- **Broadcast variables**: used to communicate value to all nodes

- **Accumulator**: nodes can only “add”
  (or multiply, or… any operation on a **monoid**)

  https://en.wikipedia.org/wiki/Monoid
  https://spark.apache.org/docs/latest/rdd-programming-guide.html#accumulators

You won’t need these in your homework, but they’re extremely useful for more complicated jobs, especially ones that are not embarrassingly parallel.
Readings (this lecture)

Required:
Spark programming guide:
https://spark.apache.org/docs/0.9.0/scala-programming-guide.html
PySpark programming guide:
https://spark.apache.org/docs/0.9.0/python-programming-guide.html

Recommended:
Spark MLlib (Spark machine learning library):
https://spark.apache.org/docs/latest/ml-guide.html
Spark GraphX (Spark library for processing graph data)
https://spark.apache.org/graphx/
Readings (next lecture)

Required:
TensorFlow tutorial: Getting Start with TensorFlow
https://www.tensorflow.org/get_started/get_started
This is the whitepaper that originally introduced TensorFlow.

Recommended:
Assorted tutorials on statistical and neural models in TensorFlow:
https://www.tensorflow.org/tutorials/