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., GCP and Amazon EC2)

https://spark.apache.org/docs/latest/
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 is 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)
  https://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/latest/api/python/getting_started/index.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
   - 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](https://spark.apache.org/docs/latest/submitting-applications.html#launching-applications-with-spark-submit)

Similar functionality on Google Cloud Platform

Similar to running Python `mrjob` scripts with the `-r dataproc` 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)
- Maintains 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/latest/rdd-programming-guide.html)

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!

In the slides that follow, I am assuming that we are logged on to a cluster that has a Spark server running. Your homework will walk you through how to set up a cluster like this on Google Cloud Platform.
Okay, let’s dive in!

Spark finishes setting up our interactive session and gives us a prompt like the Python interpreter.

There will be information here (sometimes multiple screens' worth) 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.
Creating an RDD from a file

Welcome to

Welcome to

Spark context Web UI available at
http://stat679-test-server-m.c.stat679-s21-trial.internal:35879
Spark context available as 'sc' (master = yarn, app id = application_1617664534064_0001).
SparkSession available as 'spark'.

```python
>>> sc
<SparkContext master=yarn appName=PySparkShell>
>>> data = sc.textFile('gs://uw-stat679s21-pyspark/ps_demo_file.txt')
>>> data.collect()
['This is just a demo file.', 'Normally, a file this small would have no need for Hadoop.]
```
Creating an RDD from a file

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. Note that PySpark had no trouble finding our file in our GCP storage bucket.

Our first RDD action, `collect()`, gathers the elements of the RDD into a list.
Creating an RDD from a file

Welcome to

```
Welcome to

/ __/__  ___  _____/ /_
/_ \_ _\_\_\_\_\_\_\_
\// . ___/_// /// ///
_/\// _/ \__,_/_/ /_/\_
\// / 

version 3.1.1

```

Spark context Web UI available at
http://stat679-test-server-m.c.stat679-s21-trial.internal:35879
Spark context available as 'sc' (master = yarn, app id = application_1617664534064_0001).
SparkSession available as 'spark'.

```python
>>> sc
<SparkContext master=yarn appName=PySparkShell>
>>> data = sc.textFile('gs://uw-stat679s21-pyspark/ps_demo_file.txt')
>>> data.collect()
['This is just a demo file.', 'Normally, a file this small would have no need for Hadoop.]
```
PySpark keeps track of RDDs
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

[pyspark interactive session]
>>> data = sc.textFile('gs://uw-stat679s21-pyspark/numbers.txt')
>>> data.collect()
['10', '23', '16', '7', '12', '0', '1', '1', '2', '3', '5', '8', '-1', '42', '64', '101', '-101', '3']
>>> stripped = data.map(lambda line: line.strip())
>>> stripped.collect()
['10', '23', '16', '7', '12', '0', '1', '1', '2', '3', '5', '8', '-1', '42', '64', '101', '-101', '3']

Using `strip()` here is redundant: PySpark automatically splits on whitespace when it reads from a text file. 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('gs://uw-stat679s21-pyspark/numbers.txt')
>>> stripped = data.map(lambda line: line.strip())
>>> intdata = stripped.map(lambda n: int(n))
>>> intdata.reduce(lambda x,y: x+y)
196
```
Simple MapReduce task: Summations

```python
>>> data = sc.textFile('gs://uw-stat679s21-pyspark/numbers.txt')
>>> stripped = data.map(lambda line: line.strip())
>>> intdata = stripped.map(lambda n: int(n))
>>> intdata.reduce(lambda x,y: x+y)
196
```
Simple MapReduce task: Summations

```python
>>> data = sc.textFile('gs://uw-stat679s21-pyspark/numbers.txt')
>>> stripped = data.map(lambda line: line.strip())
>>> intdata = stripped.map(lambda n: int(n))
>>> intdata.reduce(lambda x,y: x+y)
196
```
Simple MapReduce task: Summations

```python
>>> data = sc.textFile('gs://uw-stat679s21-pyspark/numbers.txt')
>>> stripped = data.map(lambda line: line.strip())
>>> intdata = stripped.map(lambda n: int(n))
>>> intdata.reduce(lambda x,y: x+y)
196

Warning: RDD laziness also means that if you have an error, you often won’t find out about it until you call an RDD action!

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('gs://uw-stat679s21-pyspark/numbers.txt')
>>> stripped = data.map(lambda line: line.strip())
>>> intdata = stripped.map(lambda n: int(n))
>>> intdata.reduce(lambda x, y: x+y)
196
```
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/latest/rdd-programming-guide.html#transformations
RDD.map()

```python
>>> data = sc.textFile('gs://uw-stat679s21-pyspark/numbers.txt').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]
>>> doubles = data.map(lambda n: 2*n)
>>> doubles.collect()
[20, 46, 32, 14, 24, 0, 2, 2, 4, 6, 10, 16, -2, 84, 128, 202, -202, 6]
>>> sc.addPyFile('gs://uw-stat679s21-pyspark/poly.py')
>>> from poly import *
>>> data.map(polynomial).collect()
[101, 530, 257, 50, 145, 1, 2, 2, 5, 10, 26, 65, 2, 1765, 4097, 10202, 10202, 10]

poly.py

```
>>> data = sc.textFile('gs://uw-stat679s21-pyspark/numbers.txt').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]
>>> doubles = data.map(lambda n: 2*n)
>>> doubles.collect()
[20, 46, 32, 14, 24, 0, 2, 2, 4, 6, 10, 16, -2, 84, 128, 202, -202, 6]
>>> sc.addPyFile('gs://uw-stat679s21-pyspark/poly.py')
>>> from poly import *
>>> data.map(polynomial).collect()
[101, 530, 257, 50, 145, 1, 2, 2, 5, 10, 26, 65, 3, 176, 4097, 10202, 10202, 10]
```python
>>> data = sc.textFile('gs://uw-stat679s21-pyspark/numbers.txt').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]
>>> doubles = data.map(lambda n: 2*n)
>>> doubles.collect()
[20, 46, 32, 14, 24, 0, 2, 2, 4, 6, 10, 16, -2, 84, 128, 202, -202, 6]
>>> sc.addPyFile('gs://uw-stat679s21-pyspark/poly.py')
>>> from poly import *
>>> data.map(polynomial).collect()
[101, 530, 257, 50, 145, 1, 2, 2, 5, 10, 26, 65, 2, 1765, 4097, 10202, 10202, 10]
>>>
```

**poly.py**

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

This file is stored in a Google Cloud storage bucket, so we have to specify its path.
RDD.filter()

```python
>>> data = sc.textFile('gs://uw-stat679s21-pyspark/numbers.txt').map(lambda n: int(n))
>>> evens = data.filter(lambda n: n%2==0)
>>> evens.collect()
[10, 16, 12, 0, 2, 8, 42, 64]
>>> odds = data.filter(lambda n: n%2!=0)
>>> odds.collect()
[23, 7, 1, 1, 3, 5, -1, 101, -101, 3]
>>> sc.addPyFile('gs://uw-stat679s21-pyspark/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`. 

`prime.py`

```python
def is_prime(n):
    if n < 1:  # Primes must be naturals.
        return False
    import math
    if n==1:
        return False
    for x in range(2,max([3,int(math.sqrt(n))])):
        if n%x==0:
            return False
    return True
```
```python
>>> data = sc.textFile('gs://uw-stat679s21-pyspark/numbers.txt').map(lambda n: int(n))
>>> samp = data.sample(False, 0.5)
>>> samp.collect()
[12, 5, -1, 42, 101, -101]
>>> samp = data.sample(True, 0.5)
>>> samp.collect()
[10, 10, 23, 7, 2, 42, 101, 3]
```
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 via serialization, which you already know about from the pickle module:
https://docs.python.org/3/library/pickle.html
Database-like file

[keith@m ~]$ gsutil cat gs://uw-stat679s21-pyspark/scientists.txt
John Bardeen 3.1 EE 1908
Eugene Wigner 3.2 Physics 1902
Albert Einstein 4.0 Physics 1879
Ronald Fisher 3.25 Statistics 1890
Emmy Noether 2.9 Physics 1882
Leonard Euler 3.9 Mathematics 1707
Jerzy Neyman 3.5 Statistics 1894
Ky Fan 3.55 Mathematics 1914
[keith@m ~]$
Database-like file

```python
>>> data = sc.textFile('gs://uw-stat679s21-pyspark/scientists.txt')
>>> data.collect()
>>> data2 = data.map(lambda line: line.split())
>>> data2.collect()
```
Database-like file

On initial read, each line is a single element in the RDD.

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

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

$$
\begin{align*}
\text{data} &= \text{sc.textFile('gs://uw-stat679s21-pyspark/scientists.txt')} \\
\text{data2} &= \text{data.map}(\lambda \text{line: line.split()}) \\
\text{fields} &= \text{data2.map}(\lambda \text{t: t[3]}) \\
\text{fields\_distinct} &= \text{fields.distinct()} \\
\text{fields\_distinct.collect()} &\Rightarrow \begin{cases} \\
'Economics', 'Statistics', 'Physics', 'Mathematics' \\
\end{cases}
\end{align*}
$$
Each tuple is of the form 
(first_name, last_name, GPA, field, birth_year)

RDD.distinct() does just what you think it does!
```python
>>> data = sc.textFile('gs://uw-stat679s21-pyspark/numbers_weird.txt')
>>> data.collect()
['10 23 16', '7 12', '0', '1 1 2 3 5 8', '-1 42', '64 101 -101', '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).

[Link to PySpark documentation](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 concatenating those lists. 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
```python
>>> data = sc.textFile('gs://uw-stat679s21-pyspark/ps_demo_file.txt')
>>> data_flat = data.flatMap(lambda line: line.split())
>>> words = data_flat.map(lambda w: w.lower())
>>> words.collect()
['this', 'is', 'just', 'a', 'demo', 'file.', 'normally,', 'a', 'file',
'this', 'small', 'would', 'have', 'no', 'reason', 'to', 'be', 'on',
hdfs.]
>>> uniqwords = words.distinct()
>>> uniqwords.count()
17
```
`RDD.countByKey()`

```python
>>> data = sc.textFile('gs://uw-stat679s21-pyspark/ps_demo_file.txt')
>>> data_flat = data.flatMap(lambda line: line.split())
>>> words = data_flat.map(lambda w: (w.lower(), 0))
>>> words.countByKey()
defaultdict(<class 'int'>, {'this': 2, 'is': 1, 'just': 1, 'a': 2, 'demo': 1, 'file.': 1, 'normally.': 1, 'file': 1, 'small': 1, 'would': 1, 'have': 1, 'no': 1, 'need': 1, 'for': 1, 'had oop.': 1})
```

**Note:** In the example above, each word had a key 0, but note that in the dictionary produced by `countByKey`, the values correspond to how many times that key appeared. This is because `countByKey()` counts how many times each key appears and ignores their values.
Sidenote: 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

You won’t need these in this course, but they’re extremely useful for more complicated jobs, especially ones that are not embarrassingly parallel.
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 a cluster and run there.

  - Cluster with PySpark server: instead of pyspark, use spark-submit
  
    This will be cluster-specific, so we won’t discuss it here

  - GCP: submit pyspark job to a Dataproc server

  https://cloud.google.com/sdk/gcloud/reference/dataproc/jobs/submit/pyspark
Submitting to the queue: `spark-submit`

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

# This script takes two arguments, an input file and output directory.
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('WordCount')
sc = SparkContext(conf=conf)

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

# Save the results in the specified output directory.
wordcounts.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 file and output file.
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('WordCount')
sc = SparkContext(conf=conf)

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

# Save the results in the specified output directory.
wordcounts.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`

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

# This script takes two arguments, an input file and output directory.
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('WordCount')
sc = SparkContext(conf=conf)

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

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

Load the modules we need and read an input filename and output location from the command line.
Submitting to the queue: spark-submit

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

# This script takes two arguments, an input file and output directory.
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('WordCount')
sc = SparkContext(conf=conf)

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

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

Load the modules we need and read an input filename and output location from the command line.

Do the actual work. Read in the data, create (word,1) keys, and sum up the counts for each key.
Submitting to the queue: spark-submit

```
[keith@cluster ~]$ spark-submit ps_wordcount.py hdfs:/stat679/demo.txt wc_demo
```

Run our `ps_wordcount.py` script on the file `demo.txt`, stored on HDFS, and store the output in a directory called `wc_demo` (which will also be created on HDFS).

This will submit our script to be run on the PySpark server (assuming we have one available on the cluster we ssh’d to…). We are also assuming that our data is on HDFS. This will only work if you are on a cluster that has HDFS configured.
Submitting to the queue: \texttt{spark-submit}

[keith@cluster ~]$ spark-submit ps_wordcount.py hdfs:/stat679/demo.txt wc_demo
[...lots of status information from Spark...]
[keith@cluster ~]$ hdfs dfs -ls wc_demo/

List the contents of our newly-created directory \texttt{wc_demo} on HDFS. If all went well, this should contain the results of our computation.
Submitting to the queue: spark-submit

[keith@cluster ~]$ spark-submit ps_wordcount.py hdfs:/stat679/demo.txt wc_demo
[...lots of status information from Spark...]
[keith@cluster ~]$ hdfs dfs -ls wc_demo/
Found 3 items
-rw-r--r--  3 klevin hdfs  0 2019-03-12 11:58 wc_demo/_SUCCESS
-rw-r--r--  3 klevin hdfs  94 2019-03-12 11:58 wc_demo/part-00000
-rw-r--r--  3 klevin hdfs 108 2019-03-12 11:58 wc_demo/part-00001
[keith@cluster ~]$
Submitting to the queue: `spark-submit`

```
[keith@cluster ~]$ spark-submit ps_wordcount.py hdfs:/stat679/demo.txt wc_demo
[...lots of status information from Spark...]
[keith@cluster ~]$ hdfs dfs -ls wc_demo/
Found 3 items
-rw-r--r--   3 klevin hdfs 0 2019-03-12 11:58 wc_demo/_SUCCESS
-rw-r--r--   3 klevin hdfs 94 2019-03-12 11:58 wc_demo/part-00000
-rw-r--r--   3 klevin hdfs 108 2019-03-12 11:58 wc_demo/part-00001

[keith@cluster ~]$ hdfs dfs -cat wc_demo/*
```

PySpark splits our script’s output into pieces, called `part-#####`. The file `_SUCCESS` is an empty file to signal that everything ran successfully. To get our actual answer, we need to combine these different “part” files.
Submitting to the queue: spark-submit

[keith@cluster ~]$ spark-submit ps_wordcount.py hdfs:/stat679/demo.txt wc_demo
[...lots of status information from Spark...]
[keith@cluster ~]$ hdfs dfs -ls wc_demo/
Found 3 items
-rw-r--r-- 3 klevin hdfs 0 2019-03-12 11:58 wc_demo/_SUCCESS
-rw-r--r-- 3 klevin hdfs 94 2019-03-12 11:58 wc_demo/part-00000
-rw-r--r-- 3 klevin hdfs 108 2019-03-12 11:58 wc_demo/part-00001
[keith@cluster ~]$ hdfs dfs -cat wc_demo/*
('this', 2)
('is', 1)
('just', 1)
[...]
('hdfs.', 1)
[keith@cluster ~]$
Submitting to the queue: spark-submit

Just like our example run locally, only this time it ran on the Spark server, working with a file stored on HDFS.
Submitting to the queue: `spark-submit`

```
[keith@cluster ~]$ spark-submit ps_wordcount.py hdfs:/stat679/demo.txt wc_demo
[...lots of status information from Spark...]
[keith@cluster ~]$ hdfs dfs -ls wc_demo/
Found 3 items
-rw-r--r--   3 klevin hdfs 0 2019-03-12 11:58 wc_demo/_SUCCESS
-rw-r--r--   3 klevin hdfs 94 2019-03-12 11:58 wc_demo/part-00000
-rw-r--r--   3 klevin hdfs 108 2019-03-12 11:58 wc_demo/part-00001
[keith@cluster ~]$ hdfs dfs -cat wc_demo/*
('this', 2)
('is', 1)
('just', 1)
[...
('hdfs.', 1)
[keith@cluster ~]$
```

Of course, this is a fictional example-- the specifics of how you call PySpark will depend on how your cluster is configured. For example, you may need to pass flag arguments to spark-submit to specify a `queue` or which resource manager to use.

More: [https://spark.apache.org/docs/latest/submitting-applications.html](https://spark.apache.org/docs/latest/submitting-applications.html)
PySpark on GCP

Step 1: spin up a compute cluster

```
gcloud dataproc clusters create CLUSTNAME --region=REGION
```
Details: [https://cloud.google.com/dataproc/docs/guides/create-cluster](https://cloud.google.com/dataproc/docs/guides/create-cluster)

Step 2: submit your job to the cluster

```
gcloud dataproc jobs submit pyspark --cluster=CLUSTER --region=REG ps_script.py -- SCRIPT-ARGS
```
Details: [https://cloud.google.com/sdk/gcloud/reference/dataproc/jobs/submit/pyspark](https://cloud.google.com/sdk/gcloud/reference/dataproc/jobs/submit/pyspark)
PySpark on GCP

**Step 1:** spin up a compute cluster

gcloud dataproc clusters create CLUSTNAME --region=REGION

Details: [https://cloud.google.com/dataproc/docs/guides/create-cluster](https://cloud.google.com/dataproc/docs/guides/create-cluster)

**Step 2:** submit your job to the cluster

gcloud dataproc jobs submit pyspark --cluster=CLUSTER --region=REG ps_script.py -- SCRIPT-ARGS


This -- is important! It tells gcloud that you are done giving arguments to the dataproc submission process, and the remaining arguments are to be passed to Python.