Spark Introduction

Big Data

March 28, 2016

Some slides adapted from Matei Zaharia, UC Berkeley AmpCamp 2012, “Parallel Programming with Spark”

Setup

- You are encouraged to work in groups
- ssh into HPC's DUMBO
- If you have problems logging in, you can use AWS:
  - Start an EMR cluster with 1 node (master only), using your key pair
  - Once instance is ready, ssh into the master node

- Get the lab files:

```
wget https://s3.amazonaws.com/bigdataclassecc/lab8files.tar.gz
tar -xvf lab8files.tar.gz
```

What is Spark?

- Spark is a fast and expressive cluster computing framework compatible with Apache Hadoop
- Improves efficiency through
  - In-memory computing primitives
  - General computation graphs
- Improves usability through rich Scala, Java, and Python APIs and interactive shell
- Goal: work with distributed collections as you would with local ones

Why Use Spark?

- Speed: up to 100x faster than Hadoop MapReduce in memory, up to 10x faster on disk
- Ease of use: supports different languages for developing applications using Spark; runs on Hadoop, Mesos, standalone, or in the cloud
- Generality: Combine SQL, streaming, and complex analytics into one platform

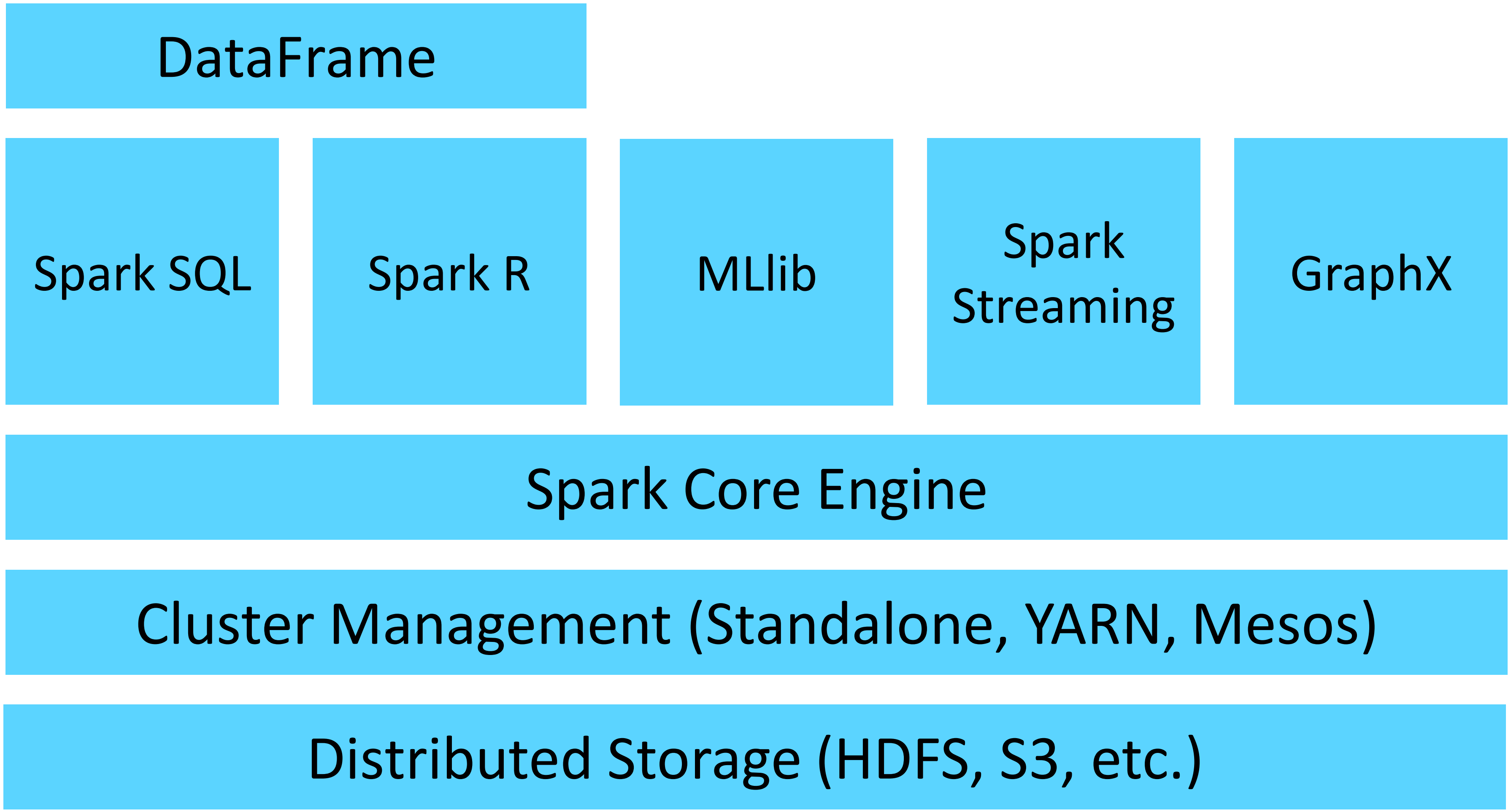
MapReduce Limitations

- MapReduce ok for one-pass computations, but inefficient for apps that require multipass computations and algorithms (ML)
- To run complex jobs, need to string together series of MapReduce jobs in sequence
- Job output of each step needs to be stored in local file system before next step can begin
- Hadoop requires integration of several tools for different big data use cases

Spark Advantages

- Less expensive shuffles in the data processing
- Uses an advanced DAG execution engine that supports cyclic data flow and in-memory computing
- Designed to work both in-memory and on-disk
- Lazy evaluation of big data queries – optimizes overall data processing workflow
- Concise and consistent APIs in Scala, Java, Python; interactive shell for Scala and Python

The Spark Platform



Resilient Distributed Datasets

- Key data structure: Resilient Distributed Datasets (RDDs)
  - Immutable collections of objects spread across a cluster
  - Built through parallel transformations (map, filter, etc)
  - Automatically rebuilt on failure
  - Controllable persistence (e.g., caching in RAM) for reuse

- 2 types of RDDs (both operated on by same methods):
  - parallelized collections : take existing collection and run functions on it in parallel
  - Hadoop datasets : run functions on each record of a file in HDFS

Main Primitives

- Resilient distributed datasets (RDDs)
  - Immutable, partitioned collections of objects
- Transformations (e.g., map, filter, groupBy, join)
  - Lazy operations to build RDDs from other RDDs
- Actions (e.g., count, collect, save)
  - Return a result or write it to storage

Transformations

Transformation	Meaning
map(*func*)	Return a new distributed dataset formed by passing each element of the source through a function *func*.
filter(*func*)	Return a new dataset formed by selecting those elements of the source on which *func* returns true.
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).
mapPartitions(*func*)	Similar to map, but runs separately on each partition (block) of the RDD, so *func* must be of type Iterator[T] => Iterator[U] when running on an RDD of type T.
mapPartitionsWithIndex(*func*)	Similar to mapPartitions, but also provides *func* with an integer value representing the index of the partition, so *func* must be of type (Int, Iterator[T]) => Iterator[U] when running on an RDD of type T.
sample(*withReplacement*, *fraction*, *seed*)	Sample a fraction *fraction* of the data, with or without replacement, using a given random number generator seed.
union(*otherDataset*)	Return a new dataset that contains the union of the elements in the source dataset and the argument.
distinct([*numTasks*]))	Return a new dataset that contains the distinct elements of the source dataset.

groupByKey([*numTasks*])	When called on a dataset of (K, V) pairs, returns a dataset of (K, Seq[V]) pairs. **Note:** By default, this uses only 8 parallel tasks to do the grouping. You can pass an optional `numTasks` argument to set a different number of tasks.
reduceByKey(*func*, [*numTasks*])	When called on a dataset of (K, V) pairs, returns a dataset of (K, V) pairs where the values for each key are aggregated using the given reduce function. Like in `groupByKey`, the number of reduce tasks is configurable through an optional second argument.
sortByKey([*ascending*], [*numTasks*])	When called on a dataset of (K, V) pairs where K implements Ordered, returns a dataset of (K, V) pairs sorted by keys in ascending or descending order, as specified in the boolean `ascending` argument.
join(*otherDataset*, [*numTasks*])	When called on datasets of type (K, V) and (K, W), returns a dataset of (K, (V, W)) pairs with all pairs of elements for each key.
cogroup(*otherDataset*, [*numTasks*])	When called on datasets of type (K, V) and (K, W), returns a dataset of (K, Seq[V], Seq[W]) tuples. This operation is also called `groupWith`.
cartesian(*otherDataset*)	When called on datasets of types T and U, returns a dataset of (T, U) pairs (all pairs of elements).

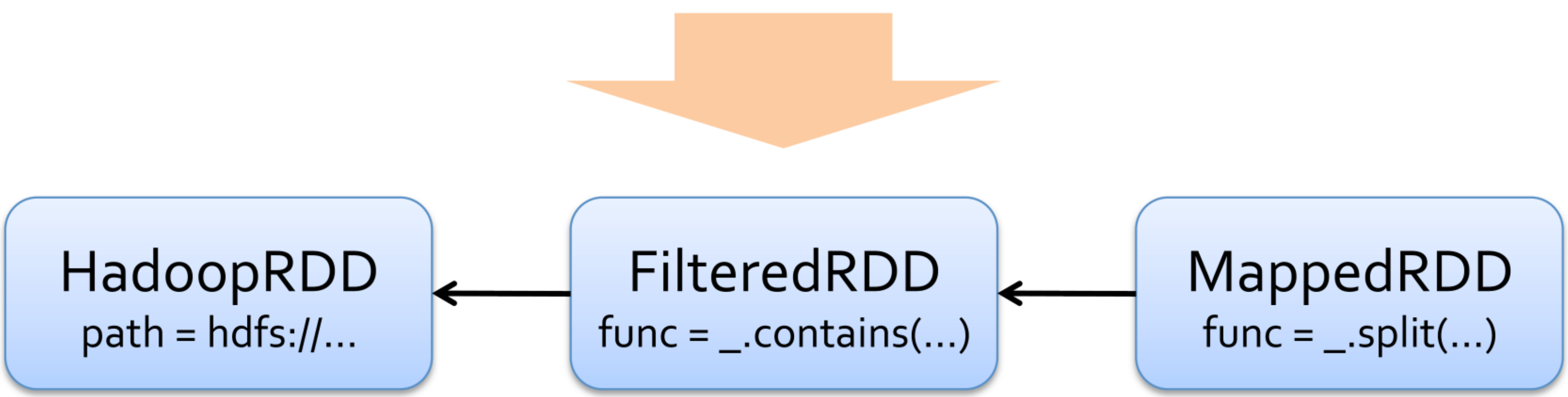
Actions

Action	Meaning
reduce(*func*)	Aggregate the elements of the dataset using a function *func* (which takes two arguments and returns one). The function should be commutative and associative so that it can be computed correctly in parallel.
collect()	Return all the elements of the dataset as an array at the driver program. This is usually useful after a filter or other operation that returns a sufficiently small subset of the data.
count()	Return the number of elements in the dataset.
first()	Return the first element of the dataset (similar to take(1)).
take(*n*)	Return an array with the first *n* elements of the dataset. Note that this is currently not executed in parallel. Instead, the driver program computes all the elements.
takeSample(*withReplacement*, *num*, *seed*)	Return an array with a random sample of *num* elements of the dataset, with or without replacement, using the given random number generator seed.
saveAsSequenceFile(*path*)	Write the elements of the dataset as a Hadoop SequenceFile in a given path in the local filesystem, HDFS or any other Hadoop-supported file system. This is only available on RDDs of key-value pairs that either implement Hadoop's Writable interface or are implicitly convertible to Writable (Spark includes conversions for basic types like Int, Double, String, etc).
countByKey()	Only available on RDDs of type (K, V). Returns a \`Map\` of (K, Int) pairs with the count of each key.
foreach(*func*)	Run a function *func* on each element of the dataset. This is usually done for side effects such as updating an accumulator variable (see below) or interacting with external storage systems.

RDD Fault Tolerance

- RDDs track the series of transformations used to build them (their *lineage*) to recomputed lost data
- Example:

```
messages = textFile(…).filter(_.contains("error")).map(_.split('\t')(2))
```



Learning Spark

- Easiest way to start: use Spark interpreter (spark-shell for scala, pyspark for python)
  - Runs in local mode on 1 thread by default

SparkContext

- Main entry point to Spark functionality
- Created for you in interactive shell as variable sc

Creating RDDs

```
//Turn a Scala collection into an RDD
sc.parallelize(List(1,2,3))

//Load text file from local FS, HDFS, or S3
sc.textFile(“file.txt”)
sc.textFile(“directory/*.txt”)
sc.textFile(“hdfs://…”)

//Use any existing Hadoop InputFormat
sc.hadoopFile(keyClass, valClass, inputFmt,
conf)
```

Basic Transformations

```
val nums = sc.parallelize(List(1,2,3))

//Pass each element through a function
val squares = nums.map(x => x*x)    // {1,4,9}

//Keep elements passing a predicate
val even = squares.filter(_ % 2 == 0)    //{4}

//Map each element to zero or more others
nums.flatMap(x => 1 to x) // => {1,1,2,1,2,3}
```

Basic Actions

```
val nums = sc.parallelize(List(1,2,3))

//Retrieve RDD contents as a local collection
nums.collect()       // => Array(1,2,3)

//Return first k elements
nums.take(2)         // => Array(1,2)

//Count number of elements
nums.count()         // => 3

//Merge elements with an associative function
nums.reduce(_ + _)           // => 6

//Write elements to a text file
nums.saveAsTextFile(“file.txt”)
```

Working with Key-Value Pairs

- Spark's "distributed reduce" transformations operate on RDDs of key-value pairs

Scala pair syntax:

```
val pair = (a,b)
```

Accessing pair elements:

```
pair._1  // => a
pair._2  // => b
```

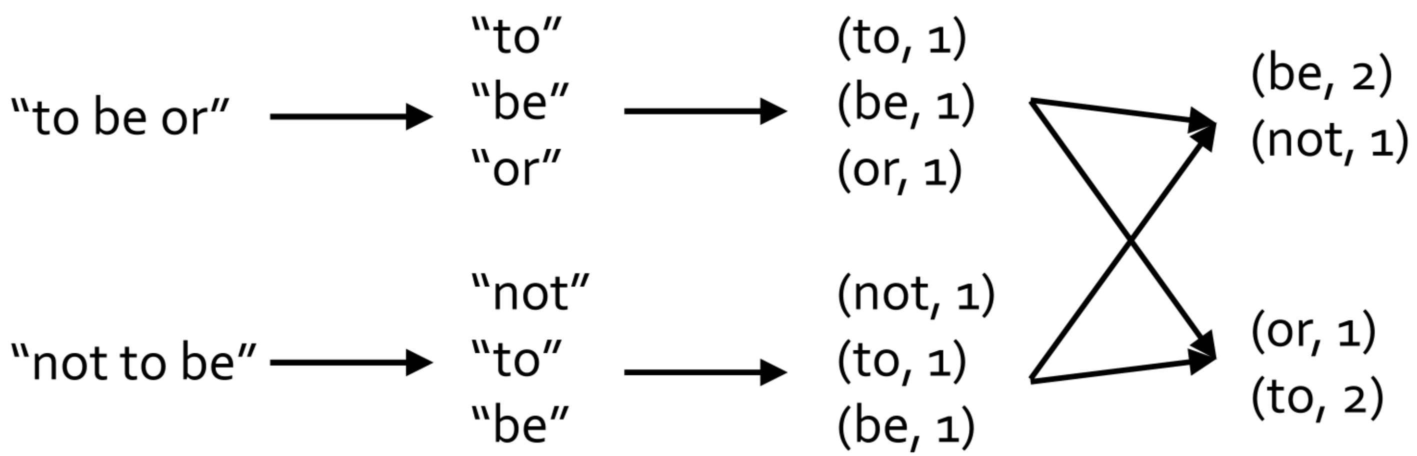
Some Key-Value Operations

```
val pets = sc.parallelize(
                List((“cat”, 1), (“dog”, 1), (“cat”, 2)))

pets.reduceByKey(_ + _) //=> {(cat,3), (dog, 1)}
pets.groupByKey() // => {(cat, Seq(1,2)), (dog, Seq(1)}
pets.sortByKey()  // => {(cat,1), (cat,2), dog(1)}
```

`reduceByKey` also automatically implements combiners on the map side

Example: Word Count

```
val lines = sc.textFile(“hamlet.txt”)

val counts = lines.flatMap(line => line.split(“ ”))
                  .map(word => (word, 1))
                  .reduceByKey(_ + _)
```



Other Key-Value Operations

```
val visits = sc.parallelize(List(
      (“index.html”, “1.2.3.4”),
      (“about.html”, “3.4.5.6”),
      (“index.html”, “1.3.3.1”)))
val pageNames = sc.parallelize(List(
     (“index.html”, “Home”), (“about.html”, “About”)))
visits.join(pageNames)
// (“index.html”, (“1.2.3.4”, “Home”))
// (“index.html”, (“1.3.3.1”, “Home”))
// (“about.html”, (“3.4.5.6”, “About”))
visits.cogroup(pageNames)
// (“index.html”, (Seq(“1.2.3.4”, “1.3.3.1”), Seq(“Home”)))
// (“about.html”, (Seq(“3.4.5.6”), Seq(“About”)))
```

Controlling the Number of Reduce Tasks

- All the pair RDD operations take an optional second parameter for number of tasks
- Examples:

```
words.reduceByKey(_ + _, 5)
words.groupByKey(5)
visits.join(pageViews, 5)
```

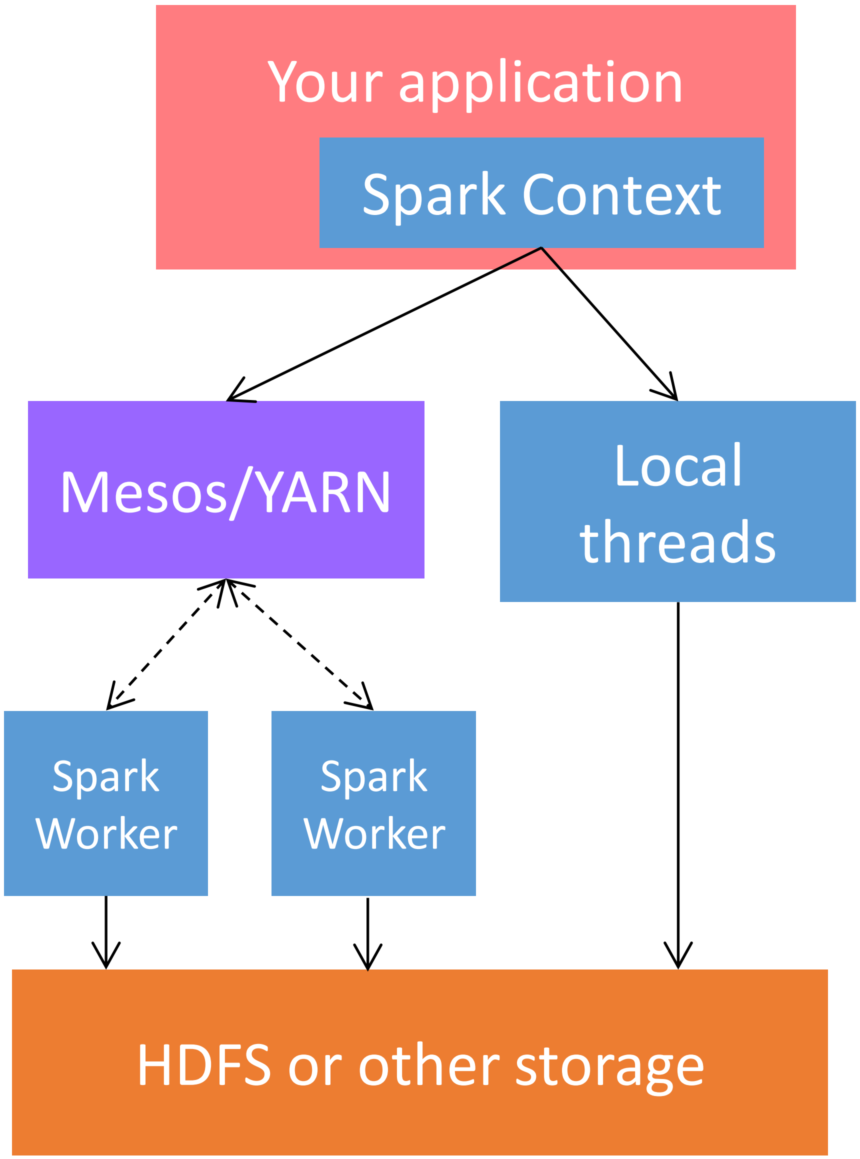
Other RDD Operations

- `sample()` : deterministically sample a subset
- `union()` : merge two RDDs
- `cartesian()` : cross product
- `pipe()` : pass through external program

- For more details see Spark Programming Guide: https://spark.apache.org/docs/0.9.0/scala-programming-guide.html

Software Components

- Spark runs as a library in your program
- Runs tasks locally or on Mesos/YARN
- Accesses storage systems via Hadoop InputFormat API (can use HDFS, S3)



PySpark

- The Spark Python API (PySpark) exposes the Spark programming model to Python
- Key differences between the Python and Scala APIs:
  - Python is dynamically typed, so RDDs can hold objects of multiple types
  - PySpark does not yet support a few API calls such as lookup and non-text input files
- RDDs support the same methods as their Scala counterparts, but take Python functions and return Python collection types.
- Supports interactive use
- https://spark.apache.org/docs/0.9.0/api/pyspark/index.html

PySpark

- Short functions can be passed to RDD methods using Python's lambda syntax:

```
logData = sc.textFile(logFile).cache()
errors = logData.filter(lambda line: “ERROR” in line)
```

- Can also pass functions that are defined with the `def` keyword (useful for longer functions):

```
def is_error(line):
      return “ERROR” in line
errors = logData.filter(is_error)
```

Let’s try it: start PySpark

- First, copy wikipedia.txt, courses.txt, and professors.txt to HDFS
  - e.g.,

```
hadoop fs -copyFromLocal wikipedia.txt
```

- To start PySpark interactive shell, type

```
pyspark
```

  - Same on both EMR master and DUMBO

Example: Word Count

Once PySpark starts, type the following lines to interactively run Word Count:

```
text_file = sc.textFile("wikipedia.txt")
counts = text_file.flatMap(lambda line: line.split(" "))
             .map(lambda word: (word.encode('utf-8'), 1))
             .reduceByKey(lambda a, b: a + b)
counts.saveAsTextFile("output")
```

To see output, press Ctrl+D to exit PySpark, then type

```
hadoop fs -get output
```

To change the number of reducers, you can type the same thing, but with

```
reduceByKey(lambda a, b: a + b, 1)
```

Example: Pi Estimation

- Spark can also be used for compute intensive tasks
- Example: estimate pi by “throwing darts” at a circle. Pick random points in the unit square ((0,0) to (1,1)) and see how many fall inside the unit circle. This fraction should be pi/4, from which we get our estimate.
- Start PySpark and type the following lines:

```
from random import random
NUM_SAMPLES = 100000

def sample(p):
    x, y = random(), random()
    return 1 if x*x + y*y < 1 else 0

count = sc.parallelize(xrange(0,NUM_SAMPLES)).map(sample)
             .reduce(lambda a, b: a + b)
print "Pi is roughly %f" % (4.0 * count / NUM_SAMPLES)
```

Submitting a Spark Job

- Wordcount example

- If you didn't before, put the data file on HDFS:

```
hadoop fs -copyFromLocal wikipedia.txt
```

- Run the wordcount Python program using Spark:

```
spark-submit wordcount.py wikipedia.txt >>
wordcountoutput.txt
```

- Output can be found in wordcountoutput.txt

Deliverable

- Fill in the crossprod.py python file with code that uses Spark to compute the cross product of the course and professor tables.
  - Look at the wordcount.py example to see how to do output
    - Note that you need to use the collect() function like in wordcount.py because RDDs are not iterable

- To run the code, you can use the command:

```
spark-submit crossprod.py courses.txt professors.txt >>
crossproductoutput.txt
```

- courses.txt and professors.txt must be on HDFS (use, e.g., hadoop fs –copyFromLocal courses.txt)
- Output will be written to crossproductoutput.txt on local file system
- Submit your crossprod.py file to NYU Classes. **Due Monday, April 4, 2016, 12:00pm (noon)**