

Spark Introduction

Big Data

February 13, 2017

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
 - See instructions in "EMRSpark.pdf" document

- Get the lab files:

```
wget https://s3.amazonaws.com/1004bigdata2017/lab4files.tar  
tar xvf lab4files.tar
```

- Copy the input file to HDFS:

```
hadoop fs -copyFromLocal sherlock.txt
```



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

DataFrame

Spark SQL

Spark R

MLlib

Spark
Streaming

GraphX

Spark Core Engine

Cluster Management (Standalone, YARN, Mesos)

Distributed Storage (HDFS, S3, etc.)

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 (<i>func</i>)	Return a new distributed dataset formed by passing each element of the source through a function <i>func</i> .
filter (<i>func</i>)	Return a new dataset formed by selecting those elements of the source on which <i>func</i> returns true.
flatMap (<i>func</i>)	Similar to map, but each input item can be mapped to 0 or more output items (so <i>func</i> should return a Seq rather than a single item).
mapPartitions (<i>func</i>)	Similar to map, but runs separately on each partition (block) of the RDD, so <i>func</i> must be of type <code>Iterator[T] => Iterator[U]</code> when running on an RDD of type T.
mapPartitionsWithIndex (<i>func</i>)	Similar to mapPartitions, but also provides <i>func</i> with an integer value representing the index of the partition, so <i>func</i> must be of type <code>(Int, Iterator[T]) => Iterator[U]</code> when running on an RDD of type T.
sample (<i>withReplacement</i> , <i>fraction</i> , <i>seed</i>)	Sample a fraction <i>fraction</i> of the data, with or without replacement, using a given random number generator seed.
union (<i>otherDataset</i>)	Return a new dataset that contains the union of the elements in the source dataset and the argument.
distinct ([<i>numTasks</i>])	Return a new dataset that contains the distinct elements of the source dataset.

groupByKey (<i>[numTasks]</i>)	<p>When called on a dataset of (K, V) pairs, returns a dataset of (K, Seq[V]) pairs.</p> <p>Note: By default, this uses only 8 parallel tasks to do the grouping. You can pass an optional <code>numTasks</code> argument to set a different number of tasks.</p>
reduceByKey (<i>func</i> , <i>[numTasks]</i>)	<p>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 <code>groupByKey</code>, the number of reduce tasks is configurable through an optional second argument.</p>
sortByKey (<i>[ascending]</i> , <i>[numTasks]</i>)	<p>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 <code>ascending</code> argument.</p>
join (<i>otherDataset</i> , <i>[numTasks]</i>)	<p>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.</p>
cogroup (<i>otherDataset</i> , <i>[numTasks]</i>)	<p>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 <code>groupWith</code>.</p>
cartesian (<i>otherDataset</i>)	<p>When called on datasets of types T and U, returns a dataset of (T, U) pairs (all pairs of elements).</p>

Actions

Action	Meaning
reduce (<i>func</i>)	Aggregate the elements of the dataset using a function <i>func</i> (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 (<i>n</i>)	Return an array with the first <i>n</i> elements of the dataset. Note that this is currently not executed in parallel. Instead, the driver program computes all the elements.
takeSample (<i>withReplacement</i> , <i>num</i> , <i>seed</i>)	Return an array with a random sample of <i>num</i> elements of the dataset, with or without replacement, using the given random number generator seed.
saveAsSequenceFile (<i>path</i>)	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 (<i>func</i>)	Run a function <i>func</i> 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

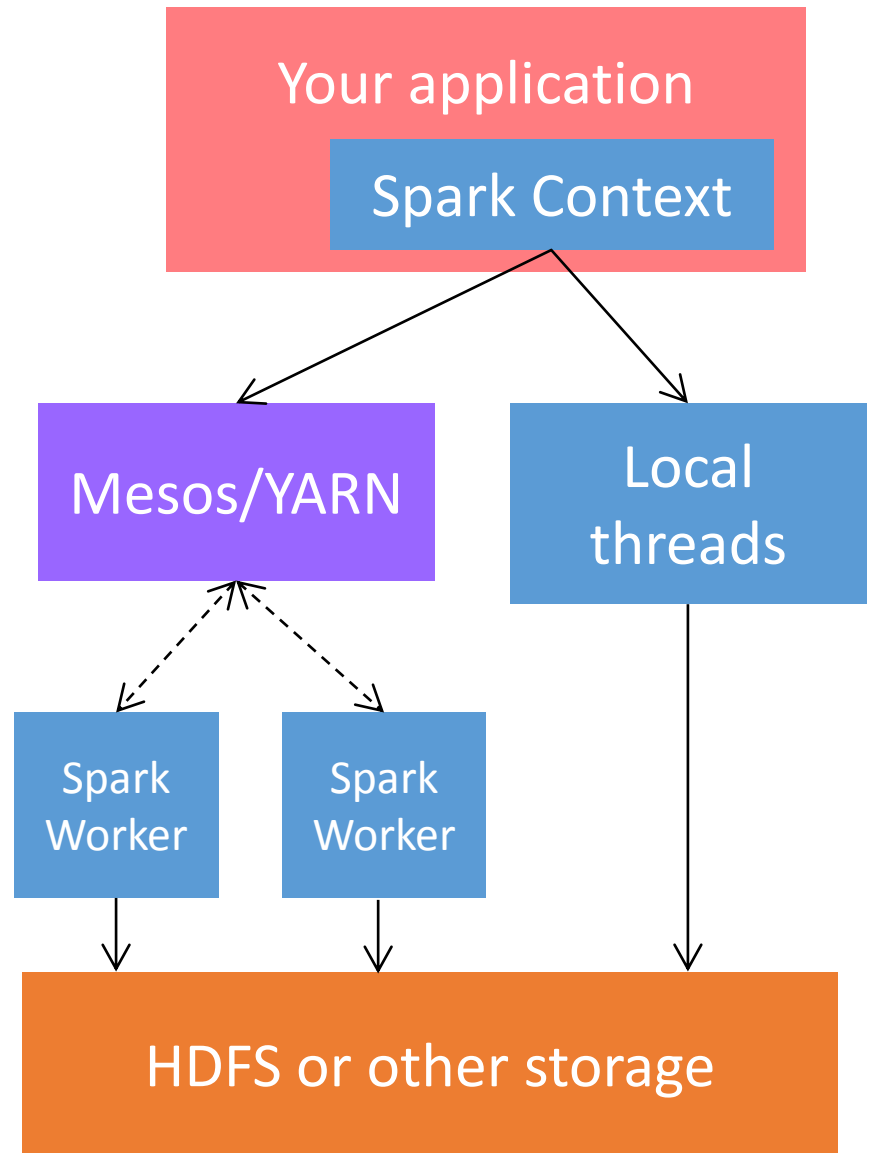
- RDDs track the series of transformations used to build them (their *lineage*) to recompute lost data
- Example:

```
messages = sc.textFile(...) \  
            .filter(lambda x: x.contains("error")) \  
            .map(lambda x: x.split('\t'))
```



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
- Runs in local mode on 1 thread by default
- PySpark API: <http://spark.apache.org/docs/1.6.0/api/python/>
- On EC2: <http://spark.apache.org/docs/2.1.0/api/python/>

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)
```


SparkContext

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

Let's try it: start PySpark

- To start PySpark interactive shell, type
`pyspark`
- Same on both EMR master and DUMBO

Creating RDDs

Turn a python list into an RDD

```
nums = sc.parallelize([1,2,3])
```

Load text file

```
mytxt = sc.textFile("sherlock.txt")
```

Basic Transformations

```
nums = sc.parallelize([1,2,3])
```

```
# Pass each element through a function
```

```
squares = nums.map(lambda x: x*x)
```

```
squares.collect() # [1,4,9]
```

```
# Keep elements passing a predicate
```

```
even = squares.filter(lambda x: x % 2 == 0)
```

```
even.collect() # [4]
```

```
# Map each element to zero or more others
```

```
expand = nums.flatMap(lambda x: range(1,x+1))
```

```
expand.collect() # [1,1,2,1,2,3]
```

Basic Actions

```
# Retrieve RDD contents as a local collection  
nums.collect()    # [1,2,3]
```

```
# Return first k elements  
nums.take(2)      # [1,2]
```

```
# Count number of elements  
nums.count()      # 3
```

```
# Merge elements with an associative function  
nums.reduce(lambda x, y: x + y)  # 6
```

```
# Write elements to a text file (on HDFS)  
nums.saveAsTextFile("file.txt")
```

Some Key-Value Operations

```
pets = sc.parallelize(\n[("cat", 1), ("dog", 1), ("cat", 2)])
```

```
pets.reduceByKey(lambda x, y: x + y).collect()\n# [(dog, 1), (cat, 3)]
```

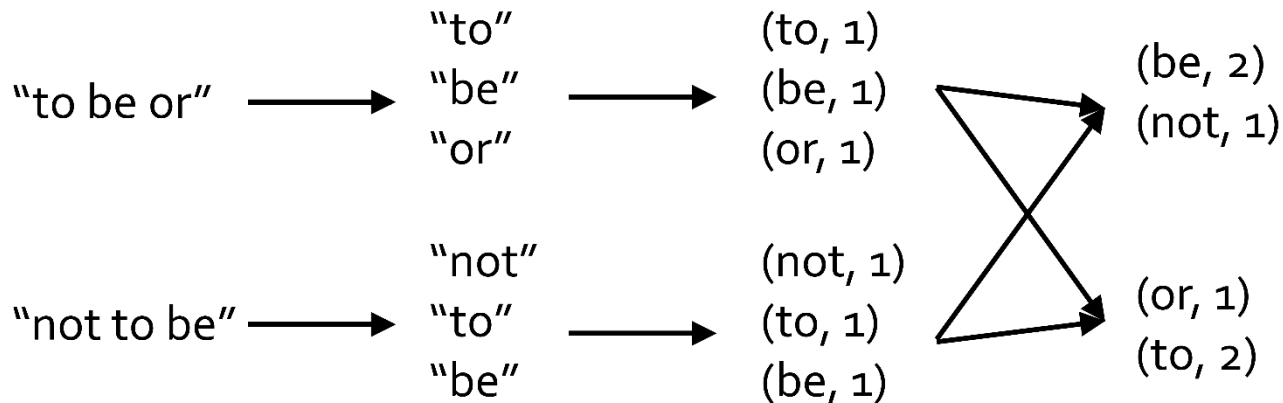
```
pets.groupByKey().collect()\n# [(cat, Seq(1, 2)), (dog, Seq(1))]
```

```
pets.sortByKey().collect()\n# [(cat, 1), (cat, 2), (dog, 1)]
```

`reduceByKey` also automatically implements combiners on the map side

Example: Word Count

```
lines = sc.textFile("sherlock.txt")
counts = lines.flatMap(lambda line: line.split(" ")) \
    .map(lambda word: (word.encode('utf-8'),1)) \
    .reduceByKey(lambda x,y: x+y)
counts.saveAsTextFile("wordcount.out")
```



Note: file written to HDFS. After quitting pyspark, you could type
hadoop fs -getmerge wordcount.out wordcount.out

Other Key-Value Operations

```
visits = sc.parallelize( \
    [("index.html", "1.2.3.4"), \
     ("about.html", "3.4.5.6"), \
     ("index.html", "1.3.3.1")])
pageNames = sc.parallelize( \
    [("index.html", "Home"), ("about.html", "About")])
visits.join(pageNames).collect()
# ("index.html", ("1.2.3.4", "Home"))
# ("index.html", ("1.3.3.1", "Home"))
# ("about.html", ("3.4.5.6", "About"))
visits.cogroup(pageNames).collect()
# ("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 (don't type):
 - `words.reduceByKey(lambda x, y : x + y, 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 (version on dumbo):
<http://spark.apache.org/docs/1.6.0/programming-guide.html>
- For version on EC2 nodes:
<http://spark.apache.org/docs/2.1.0/programming-guide.html>

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.

Example: Pi Estimation

```
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

- Exit pyspark (Ctrl+D)
- If you didn't before, put the data file on HDFS:
`hadoop fs -copyFromLocal sherlock.txt`
- Run the wordcount Python program using Spark:
`spark-submit wordcount.py sherlock.txt`
- Output can be found in wc.out
`hadoop fs -getmerge wc.out wc.out`

Bigrams

- A bigram is a list of two consecutive words in a sentence, e.g., (word1, word2)
- For example:
“There was sentence one. Here is sentence two.”

Bigrams: (there, was), (was, sentence), (sentence, one), (here, is), (is, sentence), (sentence, two)

- Some applications: generating text, predictive texting app, document similarity
- Similar structure to wordcount program, but now we are counting bigrams instead

Deliverable

- Write a Spark program `bigram.py` that computes the frequency of each bigram in `sherlock.txt`, and outputs the (bigram, count) pairs for the top 100 most frequent bigrams.
- Submit your `bigram.py` file and output file (named whatever you want) to NYU Classes. **Due Monday, February 27, 2017, 6:00pm**

Tips

- Use the `saveAsTextFile()` function to save your RDD
 - Then use “`hadoop fs -getmerge...`” to get from HDFS
- To run the code, you can use the command:

```
spark-submit bigram.py sherlock.txt
```

- `sherlock.txt` must be on HDFS - use, e.g.,

```
hadoop fs -copyFromLocal sherlock.txt
```

- The function

```
sortBy(lambda x: x[1], False)
```

can be called on an RDD of key-value pairs to sort by descending value

- The `take()` function and `sc.parallelize()` function from earlier slides may be useful.

Bigram.py format

- Problem: values in each partition of our initial RDD describe lines from the file rather than sentences. Sentences may be split over multiple lines.
 - The `glom()` RDD method is used to create a single entry for each document containing the list of all lines
 - We can then join the lines up, then resplit them into sentences using "." as the separator, using `flatMap` so that every object in our RDD is now a sentence.

```
sentences = lines.glom() \
                .map(lambda x: " ".join(x)) \
```

`sentences` is an RDD with one object = one sentence.