

COMP 543: Tools & Models for Data Science

Big Data Part Two: Beyond MapReduce

Chris Jermaine & Risa Myers

Rice University



Most Popular “Pure” MapReduce Software

- Is called Hadoop
 - Runs on JVM (like most Big Data software)
 - Includes MapReduce functionality
 - Plus the Hadoop distributed file system (HDFS)
- Hadoop popularity peaked around 2015...
- Has been declining since then
- Why? Were several issues...

Many Felt MapReduce Too Slow

- Data reread from DFS for each MR job
- Bad for iterative data processing
 - Example: most machine learning uses gradient descent
 - Need to make 100's or 1000's of passes over data
 - Re-evaluating gradient at various points

MR API is Too Restrictive

- Can only do Map
- Or MapReduce
- Everything else must be implemented in terms of those operations
- Unless you cheat
 - Ex: Mappers/Reducers talk to each other using sockets
 - But then why not just go with C/MPI?

Result: MapReduce Used Less and Less

- There are now an entire ecosystem of alternative softwares
 - Spark, Flink, etc.
- For use in both streaming and batch processing applications
- Generally oriented more towards in-memory computing
- Have far more expressive APIs
- We will focus on Spark

- #1 Hadoop MapReduce killer
- What is Spark?
 - Platform for efficient distributed data analytics
- Runs on the JVM
- Written in Scala
 - But has bindings for Java, Scala, Python, R
 - Python nice for data analytics (NumPy, SciPy)... will focus there
- Doesn't do storage
 - Focus exclusively on compute
 - Commonly used with HDFS, S3, HBase, etc.

- Basic abstraction: Resilient Distributed Data Set (RDD)
- **Resilient** - fault tolerance
- **Distributed** - across machines in a cluster
- **Data Set** - “working set of data”
- Read-only
- RDD is a data set buffered in RAM by Spark

- pySpark is interactive!
- RDDs
 - Are lazy: computations are done on demand and only as much as needed
 - Are ephemeral: computations are discarded when no longer used
 - Have lineage: the train of computations to generate an RDD is saved and may be replayed

- To create and load an RDD (in Python shell):

```
myRDD = sc.textFile (someFileName) # sc is the Spark context  
# or else...  
data = [1, 2, 3, 4, 5]  
myRDD = sc.parallelize (data) # or  
myRDD = sc.parallelize (range (20000)) # or ...
```

- Or create an RDD from another RDD ...

Computations: Series of Xforms Over RDDs

■ Example: word count

```
def countWords (fileName):  
    lines = sc.textFile (fileName)  
    tokens = lines.flatMap (lambda line: line.split("_"))  
    instances = tokens.map (lambda word: (word, 1))  
    aggCounts = instances.reduceByKey (lambda a, b: a + b)  
    return aggCounts.top (200, key=lambda p: p[1])
```

■ What transforms do we see here?

- flatMap, map, reduceByKey, top

■ Let's go through them

■ But first, quick review of lambdas...

- Fundamental to programming in Spark

What's a Lambda?

- Basically, a function that that we can pass like a variable
- Key ability: can “capture” its surroundings at creation
- Can also accept parameters

```
def addTwelveToResult (myLambda):  
    return myLambda (3) + 12
```

```
a = 23  
aCoolLambda = lambda x : x + a  
addTwelveToResult (aCoolLambda) # prints 38
```

```
a = 45  
addTwelveToResult (aCoolLambda) # prints ???
```

What's going on here?

- Anytime we see “myLambda” we replace it with the body of the lambda

```
def addTwelveToResult (myLambda):  
    return myLambda (3) + 12
```

```
a = 23  
aCoolLambda = lambda x : x + a  
addTwelveToResult (aCoolLambda)
```

- Results in

```
myLambda (3) + 12  
(x + a) + 12  
(3 + a) + 12  
(3 + 23) + 12
```

- The parentheses matter!

Lambdas and Comprehensions

- Lambdas can return many items
- Lambdas **MUST** return something

```
def sumThem (myLambda):  
    tot = 0  
    for a in myLambda ():  
        tot = tot + a  
    return tot
```

```
x = np.array([1, 2, 3, 4, 5])  
iter = lambda : (j for j in x)  
sumThem (iter) # prints 15
```

Spark Operation: flatMap ()

```
def countWords (fileName):  
  lines = sc.textFile (fileName)  
  tokens = lines.flatMap (lambda line: line.split("_"))
```

- Process every data item in the RDD
- Apply lambda to it
- Lambda argument to return zero or more results
- Can omit, combine or create elements
- Each result added into resulting RDD

Spark Operation: map ()

```
def countWords (fileName):  
    lines = sc.textFile (fileName)  
    tokens = lines.flatMap (lambda line: line.split("_"))  
    instances = tokens.map (lambda word: (word, 1))
```

- Process every data item in the RDD
- Apply lambda to it
- The lambda must return exactly one result
- The returned RDD has a new element with each element replaced by the lam

Spark Operation: reduceByKey ()

```
def countWords (fileName):  
    lines = sc.textFile (fileName)  
    tokens = lines.flatMap (lambda line: line.split("_"))  
    instances = tokens.map (lambda word: (word, 1))  
    aggCounts = instances.reduceByKey (lambda a, b: a + b)
```

- Data must be (*Key*, *Value*) pairs
- Shuffle so that all (*K*, *V*) pairs with same *K* on same machine
- Organize into (*K*, (*V*₁, *V*₂, ..., *V*_{*n*})) pairs
- Use the lambda to “reduce” the list to a single value

Spark Operation: top ()

```
def countWords (fileName):  
    lines = sc.textFile (fileName)  
    tokens = lines.flatMap (lambda line: line.split("_"))  
    instances = tokens.map (lambda word: (word, 1))  
    aggCounts = instances.reduceByKey (lambda a, b: a + b)  
    return aggCounts.top (200, key=lambda p: p[1])
```

- Data must be (*Key*, *Value*) pairs
- Takes two params... first is the number of list items to return
- Second (optional): lambda to use to obtain key for comparison
- Note: `top` collects an RDD, moving from cloud to local
- So result is **not** an RDD

An Important Note

- Spark uses lazy evaluation...
- If I run this code:

```
lines = sc.textFile (fileName)
tokens = lines.flatMap (lambda line: line.split("_"))
instances = tokens.map (lambda word: (word, 1))
aggCounts = instances.reduceByKey (lambda a, b: a + b)
```

- Nothing happens! (Other than Spark remembers the ops)
 - Spark does not execute until an attempt made to collect an RDD
 - When we hit `top()`, then all of these are executed
- Why do this?
 - By waiting until last possible second, we can “pipeline”
 - Only operations that require a shuffle can't be pipelined

Some Other, More Advanced Ops

- `groupByKey()`, `join()`, `reduce()`, `aggregate()`

groupByKey ()

- Data must be $(Key, Value)$ pairs
- Shuffle so that all (K, V) pairs with same K onto the same machine
- Organize into $(K, \langle V_1, V_2, \dots, V_n \rangle)$ pairs
- Store each list as a `ResultIterable` for future processing
- Like `reduceByKey ()` but without the reduce

- Example:

- *rddOne* is $\{(red, 9), (blue, 7), (red, 12), (green, 4)\}$

- *rddTwo* is $\{(blue, up), (green, down), (green, behind)\}$

- Result of join is $\{(blue, (7, up)), (green, (4, down)), (green, (4, behind))\}$

- Can blow up RDD size if join is many-to-many

- Requires expensive shuffle!

- ? How do we write the join?

- Unlike past operations, this is not a transform from RDD to RDD
 - This is like `top()`
 - It moves the result back to Python
- Repeatedly applies a lambda to each item in the RDD to get single result

```
>>> myData = sc.parallelize (range(20000))  
>>> myData.reduce (lambda a, b: a + b)  
199990000
```

- With `reduce()` you aggregate directly, can be restrictive...
- Example: RDD is $\{(red, 9), (blue, 7), (red, 12), (green, 4)\}$
 - Want: **dictionary where value is sum for each unique color**
 - Cannot use `reduce()`
 - It only “sums” up the items in the input RDD directly
 - Two inputs and output must be the same type
 - How do I get the desired output dictionary by defining $+$ in $((red, 9) + (blue, 7)) + (red, 12)) + (green, 4)$?

aggregate() (Continued)

- Data must be $(Key, Value)$ pairs
- Organize into $(K, \langle V_1, V_2, \dots, V_n \rangle)$ pairs
- Then aggregate the list, like `reduce()`
- `aggregate()` takes three arguments
 - The “zero” to init the aggregation
 - Lambda that takes X_1, X_2 and aggregates them, where X_1 already aggregated, X_2 not
 - Lambda that takes X_1, X_2 and aggregates them, where both aggregated

aggregate() (Example)

```
1 def add (dict, tuple):
2     result = {}
3     for key in dict:
4         result[key] = dict[key]
5     if (tuple[0] in result):
6         result[tuple[0]] += tuple[1]
7     else:
8         result[tuple[0]] = tuple[1]
9     return result
```

Goal: Add tuple to dict and return a new dictionary

- 1 Define add that takes a dictionary and a tuple
- 2 initialize the result dictionary
- 3 For each item in the dictionary we are given
- 4 Create an entry in our result dictionary with dict's value
- 5 If we've seen the tuple key before
- 6 Accumulate the value
- 7 If this is the first time we see the tuple key
- 8 Create an entry for it and initialize with the value
- 9 Return our result dictionary

aggregate() (Example)

```
1 def combine (dict1, dict2):
2     result = {}
3     for key in dict1:
4         result[key] = dict1[key]
5     for key in dict2:
6         if (key in result):
7             result[key] += dict2[key]
8         else:
9             result[key] = dict2[key]
10    return result
```

Return a new dictionary that contains all the keys in dict1 and dict2 with the total counts

Goal:

- 1 Define combine that takes 2 dictionaries
- 2 initialize the result dictionary
- 3 For each item in dict1 that we are given
- 4 Create an entry in our result dictionary with dict1's value
- 5 For each key, value in dict2 that we are given
- 6 If we've seen the key before
- 7 Accumulate the value
- 8 If this is the first time we see the tuple key
- 9 Create an entry for it and initialize with the value
- 10 Return our result dictionary

aggregate() (Example)

```
>>> myRdd = sc.parallelize ([('red', 9), ('blue', 7),  
... ('red', 12), ('green', 4)])  
>>> myRdd.aggregate ( {}, lambda x, y: add (x, y),  
... lambda x, y: combine (x, y))  
  
{'blue': 7, 'green': 4, 'red': 21}
```

Closing Thoughts

- When is Spark/MapReduce a better option than HPC?
 - When your pipeline is heavily data-oriented
 - Or when your compute is (relatively) loosely coupled
- Key benefits compared to HPC
 - Built in fault tolerance
 - Better support for BIG data
 - Much higher programmer productivity
- Will continue to take market share from HPC
 - You see academic papers with both MPI, Spark implementations
 - But not everything can move to Spark

Questions?

`https:
//spark.apache.org/docs/latest/rdd-programming-guide.html`

`https://spark.apache.org/docs/latest/api/python/`