L03a: Filter, Join & SQL

ANLY 502: Massive Data Fundamentals

Simson Garfinkel & Ghaleb Abdulla

January 25, 2016

Welcome to the midpoint!

Outline for today's class

Administrivia

Midterm!

Student Presentations

PS04 Extra Credit

Final Projects

Student Presentations

L02 — Student Presentations

Josh Kaplan	Paper	Big Data: The End of Privacy or a New Beginning?
Jordan Bramble	Program	TensorFlow: High Scalability Machine Learning
Ron Graf	Program	GraphX

PS04

PS04 EXTRA CREDIT

8. In s3://gu-anly502/maxmind/ you will find the file GeoLite2-Country-Blocks-IPv4.csv and GeoLite2-Country-Locations-en.csv, which together contain the entire MaxMind GeoLite2-by-country dataset. Write a pyspark program that:
    a. Reads the two files and performs a RDD join operation, so that each network prefix is joined with its country.
    b. Converts the dotted-quad notation (e.g. aaa.bbb.ccc.ddd) to an integer and stores this in an array, where each row of the array is:
        (integer network number, country name).
    c. Using Spark broadcast variables, sends this data structure to all of the nodes.
    d. Reads the forensicswiki data for 2012.
    e. Geolocates each line of the forensicswiki data and produce a report of country vs. hits. You'll need to do a binary search through the maxmind data to find the matching network using the bisect algorithm. However, the ip address won't match until you have removed enough of the least significant bits.

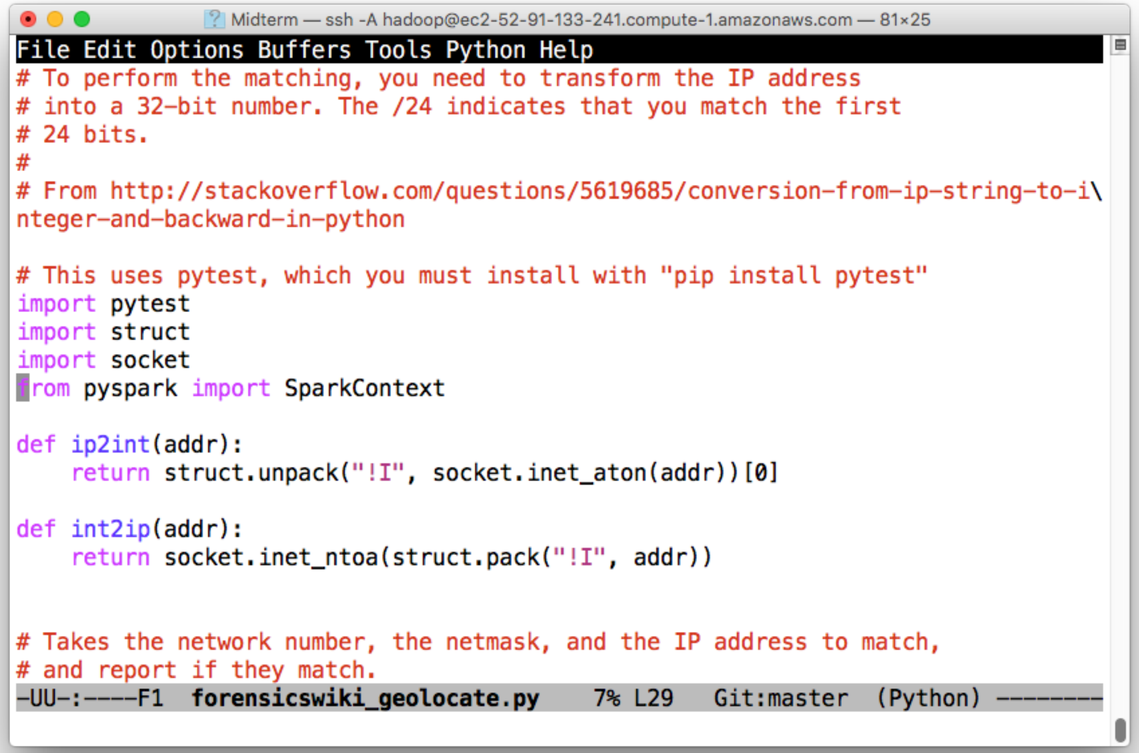
        You can generate a bitmask for cidr block /nn in python with:

        ```
        (0xffffffff ^ (0xffffffff >> nn))
        ```

        You'll need to apply this bitmask to the forensicswiki ipaddress before searching through the maxmind data with **bisect**; start with nn=32 and go down to nn=7.
    f. For each country, determines the three most popular wiki pages.

I approached this problem with two windows:

Notebook



Playpen

Running emacs (or VIM)
Keep all of your notes and code
Ready to run as a completed program

Running ipyspark (iPython w/ pyspark

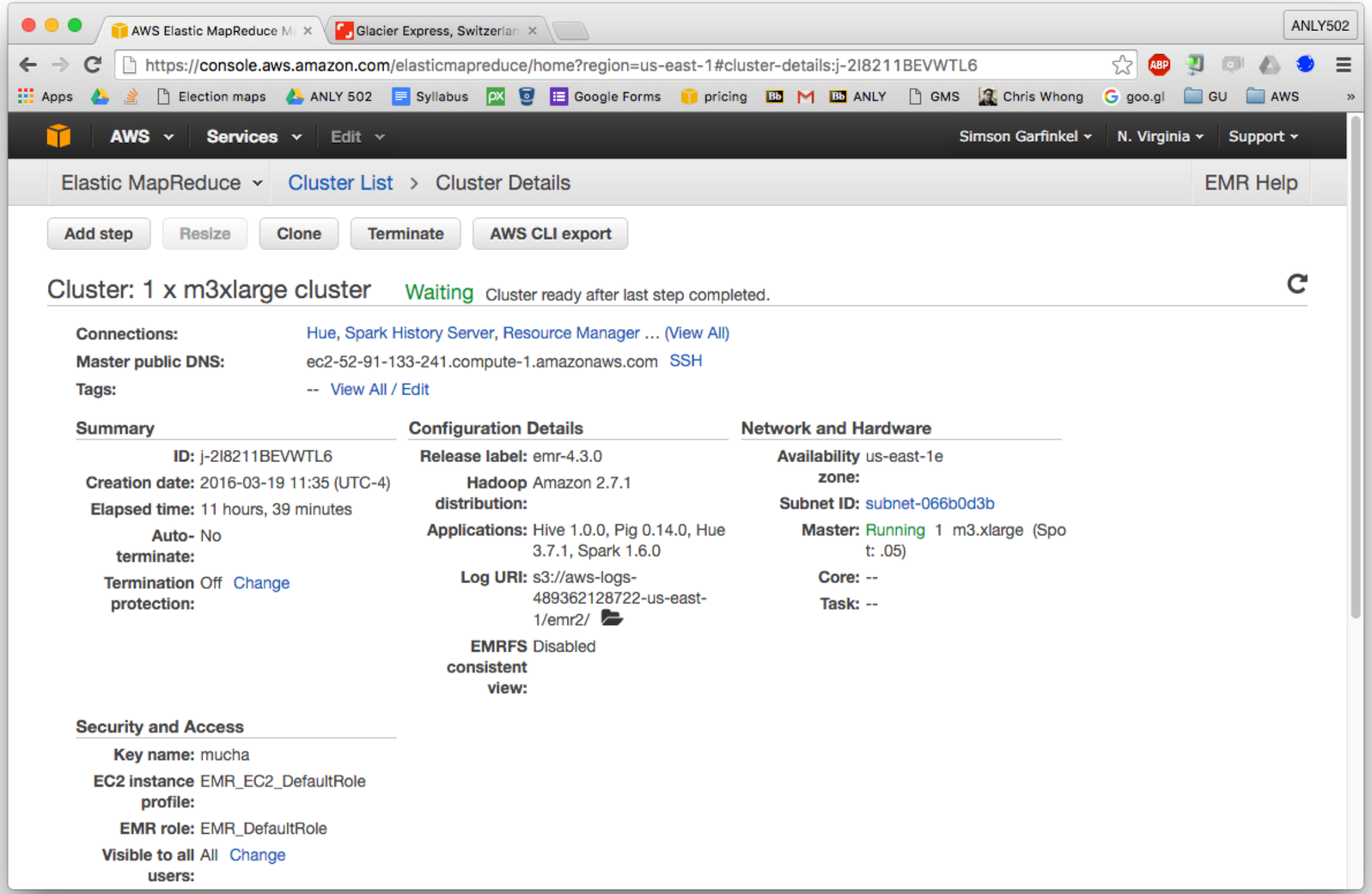
I have a single node m3.xlarge @ $0.05/hour

No pressure!

$0.05/hour = $1.20/day

Personal git repository

- git commit -m 'message' -a
- git push (to bitbucket)



Step 1 — Join the two tables
Look at the data

```
  -rw-rw-r--  1 hadoop hadoop     235 Mar 19 16:11 GEOLITE2_COPYRIGHT.txt
  -rw-rw-r--  1 hadoop hadoop 6666094 Mar 19 16:11 GeoLite2-Country-Blocks-IPv4.csv
  -rw-rw-r--  1 hadoop hadoop    9275 Mar 19 16:11 GeoLite2-Country-Locations-en.csv

GeoLite2-Country-Blocks-IPv4.csv:

network,geoname_id,registered_country_geoname_id,represented_country_geoname_id,is_anonymous_proxy,is_satellite_pr
ovider
1.0.0.0/24,2077456,2077456,,0,0
1.0.1.0/24,1814991,1814991,,0,0
1.0.2.0/23,1814991,1814991,,0,0
1.0.4.0/22,2077456,2077456,,0,0
1.0.8.0/21,1814991,1814991,,0,0
1.0.16.0/20,1861060,1861060,,0,0

geoname_id,locale_code,continent_code,continent_name,country_iso_code,country_name
49518,en,AF,Africa,RW,Rwanda
51537,en,AF,Africa,SO,Somalia
69543,en,AS,Asia,YE,Yemen
99237,en,AS,Asia,IQ,Iraq
102358,en,AS,Asia,SA,"Saudi Arabia"
130758,en,AS,Asia,IR,Iran
146669,en,EU,Europe,CY,Cyprus
149590,en,AF,Africa,TZ,Tanzania
163843,en,AS,Asia,SY,Syria
174982,en,AS,Asia,AM,Armenia
```

Simple join:

Country-Blocks-IPv4

```
1.0.0.0/24,2077456,2077456,,0,0
1.0.1.0/24,1814991,1814991,,0,0
1.0.2.0/23,1814991,1814991,,0,0
...
```

Country-Locations-en

```
...
2029969,en,AS,Asia,MN,Mongolia
2077456,en,OC,Oceania,AU,Australia
2078138,en,AS,Asia,CX,"Christmas Island"
...
```

To do this join:

- Create RDD A = (ID,[Country-Blocks-IPv4])
- Create RDD B = (ID,[Country Name])
- Join with RDD.joinByKey()
- Produce a new RDD with each row being (Country-Block-IPv4, Country Name)

Try it out in iPython, then copy to the text editor

```
In [4]: geolite_ipblocks = sc.textFile("s3://gu-anly502/maxmind/GeoLite2-Country-Blocks-IPv4.csv")

In [5]: geolite_ipblocks.take(5)
Out[5]:
[u'network,geoname_id,registered_country_geoname_id,represented_country_geoname_id,is_anonymous_proxy,is_satellite_provider',
 u'1.0.0.0/24,2077456,2077456,,0,0',
 u'1.0.1.0/24,1814991,1814991,,0,0',
 u'1.0.2.0/23,1814991,1814991,,0,0',
 u'1.0.4.0/22,2077456,2077456,,0,0']

(That looks good – but it has a header. Can we get rid of it?)

In [6]: geolite_ipblocks_raw = sc.textFile("s3://gu-anly502/maxmind/GeoLite2-Country-Blocks-IPv4.csv")

In [7]: header               = geolite_ipblocks_raw.first()

In [8]:     geolite_ipblocks     = geolite_ipblocks_raw.\
   ...:                            filter(lambda line:line != header).\
   ...:                            map(lambda line:line.split(","))

In [9]: geolite_ipblocks.take(5)

Out[9]:
[[u'1.0.0.0/24', u'2077456', u'2077456', u'', u'0', u'0'],
 [u'1.0.1.0/24', u'1814991', u'1814991', u'', u'0', u'0'],
 [u'1.0.2.0/23', u'1814991', u'1814991', u'', u'0', u'0'],
 [u'1.0.4.0/22', u'2077456', u'2077456', u'', u'0', u'0'],
 [u'1.0.8.0/21', u'1814991', u'1814991', u'', u'0', u'0']]
```

Note: Numbers are *strings*.

Let's get the ID as the first element of the IP blocks RDD:

```
Out[9]:
[[u'1.0.0.0/24', u'2077456', u'2077456', u'', u'0', u'0'],
 [u'1.0.1.0/24', u'1814991', u'1814991', u'', u'0', u'0'],
 [u'1.0.2.0/23', u'1814991', u'1814991', u'', u'0', u'0'],
 [u'1.0.4.0/22', u'2077456', u'2077456', u'', u'0', u'0'],
 [u'1.0.8.0/21', u'1814991', u'1814991', u'', u'0', u'0']]

In [11]: geolite_ipblocks_by_countryid = geolite_ipblocks.map(lambda x:(x[1],x))

In [12]: geolite_ipblocks_by_countryid.take(3)

16/13/09 20:01:24 INFO SparkContext: Starting job: runJob at PythonRDD.scala:393
16/13/09 20:01:24 INFO DAGScheduler: Got job 3 (runJob at PythonRDD.scala:393) with 1 output partitions
16/13/09 20:01:24 INFO DAGScheduler: Final stage: ResultStage 3 (runJob at PythonRDD.scala:393)
16/13/09 20:01:24 INFO DAGScheduler: Parents of final stage: List()
16/13/09 20:01:24 INFO DAGScheduler: Missing parents: List()
16/13/09 20:01:24 INFO DAGScheduler: Submitting ResultStage 3 (PythonRDD[7] at RDD at PythonRDD.scala:43), which has no
missing parents
...
Out[12]:
[(u'2077456', [u'1.0.0.0/24', u'2077456', u'2077456', u'', u'0', u'0']),
 (u'1814991', [u'1.0.1.0/24', u'1814991', u'1814991', u'', u'0', u'0']),
 (u'1814991', [u'1.0.2.0/23', u'1814991', u'1814991', u'', u'0', u'0'])]
```

Let's get the ID as the first element of the Country-Locations RDD

```
In [13]: geolite_locations_raw  = sc.textFile("s3://gu-anly502/maxmind/GeoLite2-Country-Locations-en.csv")

In [14]: header                 = geolite_locations_raw.first()

In [15]: geolite_locations      = geolite_locations_raw.filter(lambda line:line != header)

In [16]: geolite_locations.take(3)

Out[16]:
[u'49518,en,AF,Africa,RW,Rwanda',
 u'51537,en,AF,Africa,SO,Somalia',
 u'69543,en,AS,Asia,YE,Yemen']

In [17]:     geolite_locations2     = geolite_locations.map(lambda line:line.split(","))

In [18]:     geolite_locations_by_countryid = geolite_locations2.map(lambda x: x[0],x[5])
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-18-8fac34d58581> in <module>()
----> 1 geolite_locations_by_countryid = geolite_locations2.map(lambda x: x[0],x[5])

NameError: name 'x' is not defined

In [19]: geolite_locations_by_countryid = geolite_locations2.map(lambda x: (x[0],x[5]))

In [20]: geolite_locations_by_countryid.take(3)

Out[20]: [(u'49518', u'Rwanda'), (u'51537', u'Somalia'), (u'69543', u'Yemen')]
```

Whoops!

Review the Join documentation:

```
In [21]: help(geolite_ipblocks_by_countryid.join)
```

Review the documentation

```
Help on method join in module pyspark.rdd:

join(self, other, numPartitions=None) method of pyspark.rdd.PipelinedRDD instance
    Return an RDD containing all pairs of elements with matching keys in
    C{self} and C{other}.

    Each pair of elements will be returned as a (k, (v1, v2)) tuple, where
    (k, v1) is in C{self} and (k, v2) is in C{other}.

    Performs a hash join across the cluster.

    >>> x = sc.parallelize([("a", 1), ("b", 4)])
    >>> y = sc.parallelize([("a", 2), ("a", 3)])
    >>> sorted(x.join(y).collect())
    [('a', (1, 2)), ('a', (1, 3))]
(END)
```

Try it out:

```
In [23]: geolite = geolite_ipblocks_by_countryid.join(geolite_locations_by_countryid)

In [24]: geolite.take(5)

16/03/19 20:07:31 INFO SparkContext: Starting job: runJob at PythonRDD.scala:393
16/03/19 20:07:31 INFO DAGScheduler: Registering RDD 17 (join at <ipython-input-23-9bb598391b7a>:1)
16/03/19 20:07:31 INFO DAGScheduler: Got job 7 (runJob at PythonRDD.scala:393) with 1 output partitions
16/03/19 20:07:31 INFO DAGScheduler: Final stage: ResultStage 8 (runJob at PythonRDD.scala:393)
...

16/03/19 20:07:36 INFO DAGScheduler: ResultStage 8 (runJob at PythonRDD.scala:393) finished in 0.200 s
16/03/19 20:07:36 INFO DAGScheduler: Job 7 finished: runJob at PythonRDD.scala:393, took 4.176280 s
Out[24]:
[(u'49518',
  ([u'196.12.140.0/22', u'49518', u'934292', u'', u'0', u'0'], u'Rwanda')),
 (u'49518',
  ([u'196.12.144.0/22', u'49518', u'934292', u'', u'0', u'0'], u'Rwanda')),
 (u'49518',
  ([u'196.12.153.0/24', u'49518', u'934292', u'', u'0', u'0'], u'Rwanda')),
 (u'49518',
  ([u'196.44.240.0/20', u'49518', u'49518', u'', u'0', u'0'], u'Rwanda')),
 (u'49518',
  ([u'196.49.7.0/24', u'49518', u'49518', u'', u'0', u'0'], u'Rwanda'))]
```

Note: .take(50) took 0.362948 seconds (less, because data were already cached)

Next step: How do we use this to geolocate an IP address

We have this:

```
[(u'49518',
  ([u'196.12.140.0/22', u'49518', u'934292', u'', u'0', u'0'], u'Rwanda')),
```

We need a data structure that given

- 196.12.140.15 → "Rwanda"
- 196.12.141.1 → "Rwanda"
- 196.12.142.15 → "Rwanda"
- but...
- 196.12.145.1 → ""

To do this, we need to understand:

- CIDR notation
- Binary searching with bisect

Internet Addresses (IPv4)

IP Address — An address of a computer for the Internet Protocol

IPv4 — 32 bit addresses

Dotted Quad Notation: 141.161.99.73

Binary Notation: 10001101 10100001 01100011 01001010

(141) (161) (99) (74)

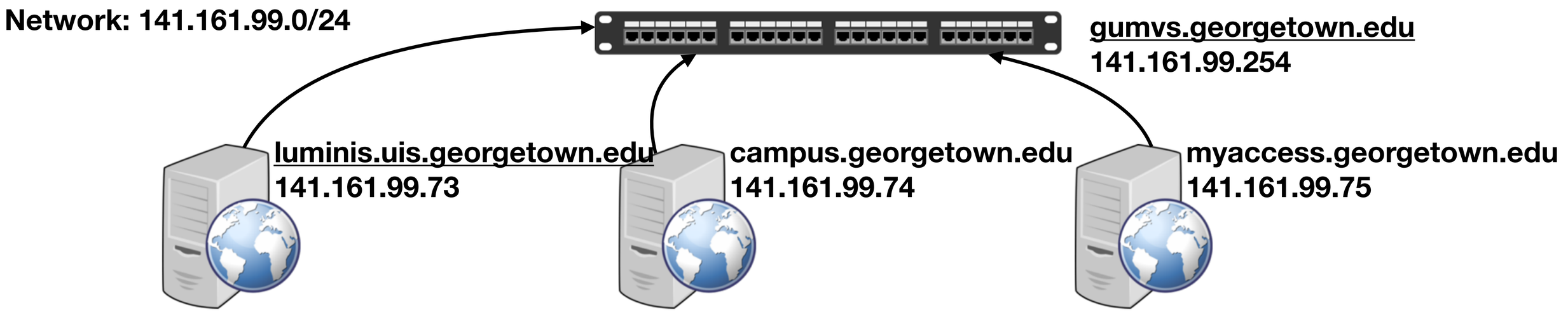
(64+8+4+1) (128+32+1) (64+32+2+1) (64+8+2)

campus.georgetown.edu
141.161.99.74

c-69-143-188-132.hsd1.va.comcast.net
69.143.188.132

http://www.csgnetwork.com/ipaddconv.html

Each address has two parts: [NETWORK][HOST]



	141	161	99	*
	10001101	10100001	01100011	????????
Netmask:	255	255	255	0
	11111111	11111111	11111111	00000000

24 1's

Every host on this network matches 141.161.99.0/24

Original IP address allocation scheme: network classes

1981 - 1993

Name	Netmask	CIDR	# of network bits	# of host bits	# of hosts*
Class A	255.0.0.0	/8	8	24	16,777,215
Class B	255.255.0.0	/16	16	16	65,534
Class C	255.255.255.0	/24	24	8	254

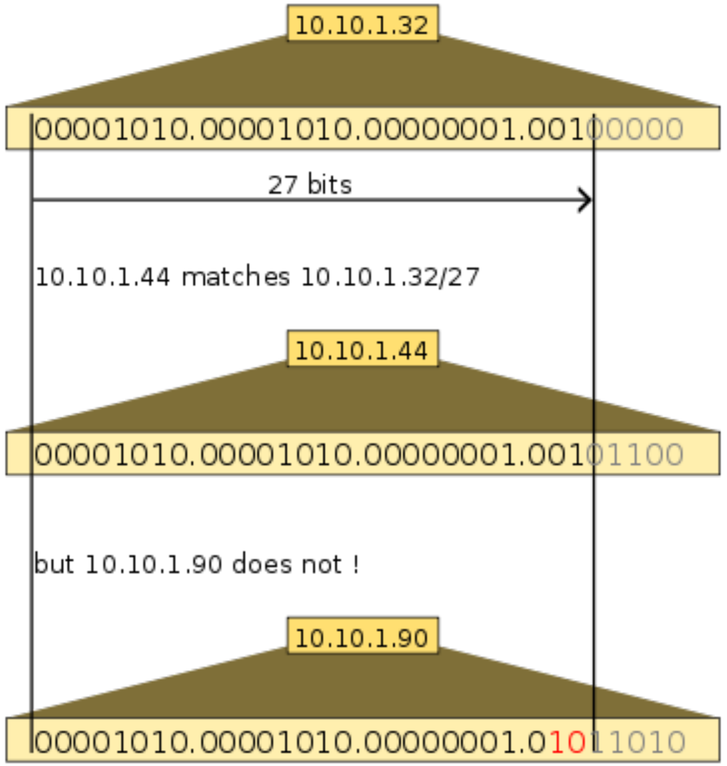
https://en.wikipedia.org/wiki/Classful_network

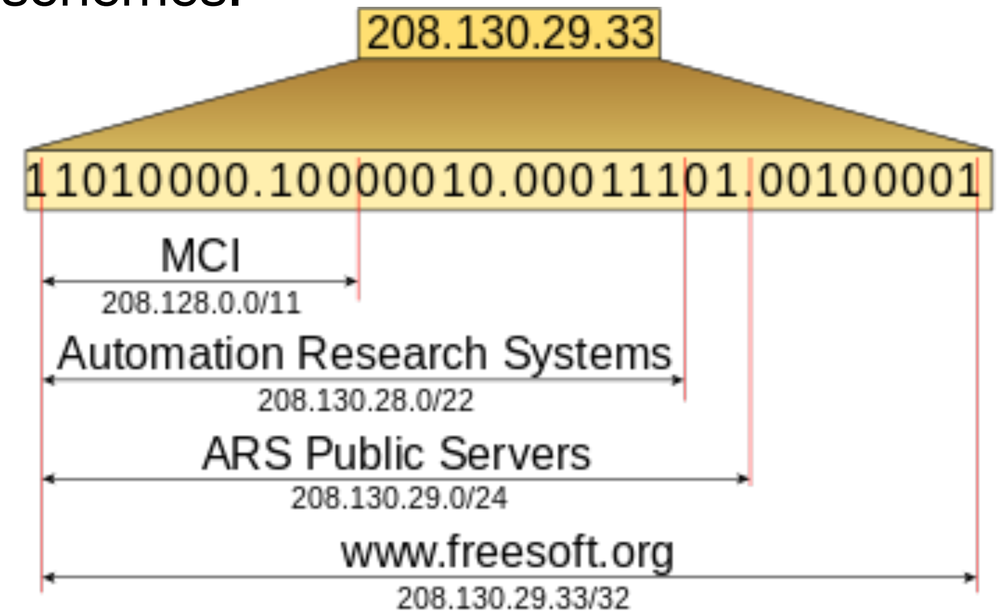
***Note: the host with all 0's is reserved for the network, and all 1's is reserved as the network broadcast address**

CIDR — Classless Inter-Domain Routing

CIDR allows any number of network bits to be used as a "block."

Network addressing matches administrative allocation schemes.





https://en.wikipedia.org/wiki/Classless_Inter-Domain_Routing

Most networks are geographically located.
Maxmind distributes its database as two tables: [networks → ID] [ID → Name]

GeoLite2-Country-Blocks-IPv4.csv

```
1.0.0.0/24,2077456,2077456,,0,0
1.0.1.0/24,1814991,1814991,,0,0
1.0.2.0/23,1814991,1814991,,0,0
1.0.4.0/22,2077456,2077456,,0,0
1.0.8.0/21,1814991,1814991,,0,0
1.0.16.0/20,1861060,1861060,,0,0
1.0.32.0/19,1814991,1814991,,0,0
1.0.64.0/18,1861060,1861060,,0,0
1.0.128.0/17,1605651,1605651,,0,0
1.1.0.0/24,1814991,1814991,,0,0
1.1.1.0/24,2077456,2077456,,0,0
1.1.2.0/23,1814991,1814991,,0,0
1.1.4.0/22,1814991,1814991,,0,0
1.1.8.0/21,1814991,1814991,,0,0
```

GeoLite2-Country-Locations-en.csv

```
1831722,en,AS,Asia,KH,Cambodia
1835841,en,AS,Asia,KR,"Republic of Korea"
1861060,en,AS,Asia,JP,Japan
1873107,en,AS,Asia,KP,"North Korea"
1880251,en,AS,Asia,SG,Singapore
1899402,en,OC,Oceania,CK,"Cook Islands"
1966436,en,OC,Oceania,TL,"East Timor"
2017370,en,EU,Europe,RU,Russia
2029969,en,AS,Asia,MN,Mongolia
2077456,en,OC,Oceania,AU,Australia
2078138,en,AS,Asia,CX,"Christmas Island"
2080185,en,OC,Oceania,MH,"Marshall Islands"
2088628,en,OC,Oceania,PG,"Papua New Guinea"
```

We need a function that takes an IP address and finds the matching entry in the GeoLite2-Country-Blocks-IPv4 table

Proposed algorithm: try matching 32 bits, then 31 bits, etc.

campus.georgetown.edu
141.161.99.74

```
141.128.0.0/15,6252001,6252001,,0,0

6252001,en,NA,"North America",US,"United States"
```

141.161.99.74 = 10001101 10100001 01100011 01001010 = 2376164170 = A
141.128.0.0 = 10001101 10000000 00000000 00000000 = 2373976064 = B
/15 = 11111111 11111110 00000000 00000000 = 4294836224 = MASK

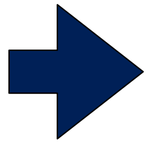
Note: MASK = /nn = (0xFFFFFFFF ^ (0xFFFFFFFF >> nn))

Problem: we don't know what /nn is when trying to match.
Proposed Solution: Search for B/32, B/31, B/30, B/29, etc...

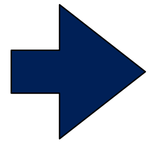
Better Solution: Convert all Maxmind networks to binary (integers) and search the array.

campus.georgetown.edu
141.161.99.74

```
141.127.64.0/18,660013,6252001,,0,0
141.127.128.0/17,660013,6252001,,0,0
141.128.0.0/15,6252001,6252001,,0,0
141.130.0.0/16,2921044,2921044,,0,0
141.131.0.0/16,6252001,6252001,,0,0
```



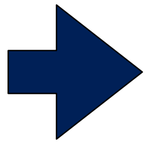
```
10001101 01111111 01000000 00000000
10001101 01111111 10000000 00000000
10001101 10000000 00000000 00000000
10001101 10000010 00000000 00000000
10001101 10000011 00000000 00000000
```



```
2373926912
2373943296
2373976064
2374107136
2374172672
```

The maxmind table will store Network, CIDR, and Location

```
141.127.64.0/18,660013,6252001,,0,0
141.127.128.0/17,660013,6252001,,0,0
141.128.0.0/15,6252001,6252001,,0,0
141.130.0.0/16,2921044,2921044,,0,0
141.131.0.0/16,6252001,6252001,,0,0
```



```
2373926912, 18, Finland
2373943296, 17, Finland
2373976064, 15, "United States"
2374107136, 16, Germany
2374172672, 16, "United States"
```

networks_and_locations

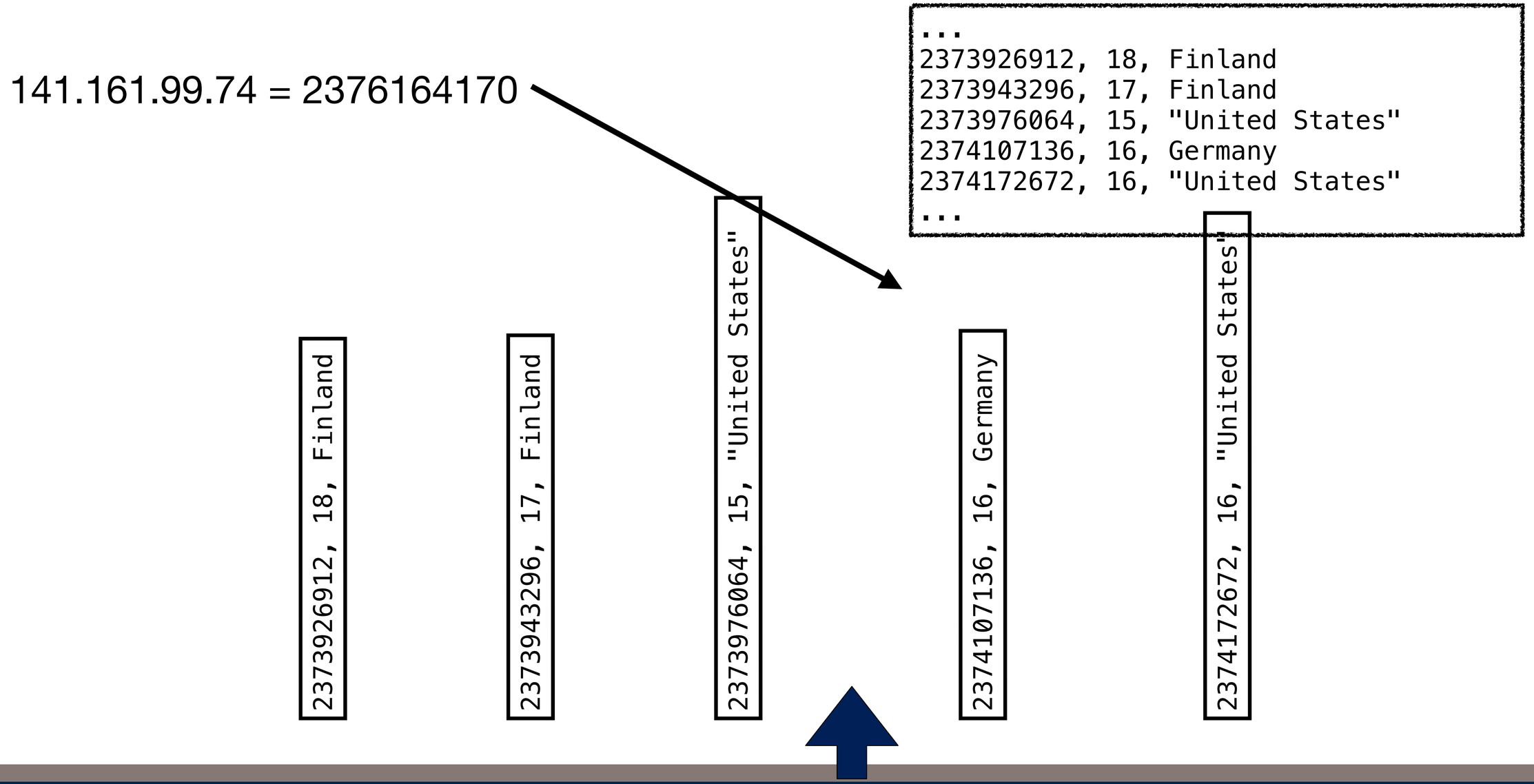
For the geolocation problem, we need to:

- Create the database — *Created as an RDD with a join (not really needed)*
- Distribute the database to the nodes
- Search the database

Options for searching:

- #1: You can do a lookup IP/32, IP/31, IP/30 — RDD supports lookup but not search
- #2: You can distribute a sorted array to the nodes, and use **bisect.bisect**

bisect — python built-in module for efficiently searching a sorted list with binary search.



My database prep and bisect search code looks like this:

We had this:

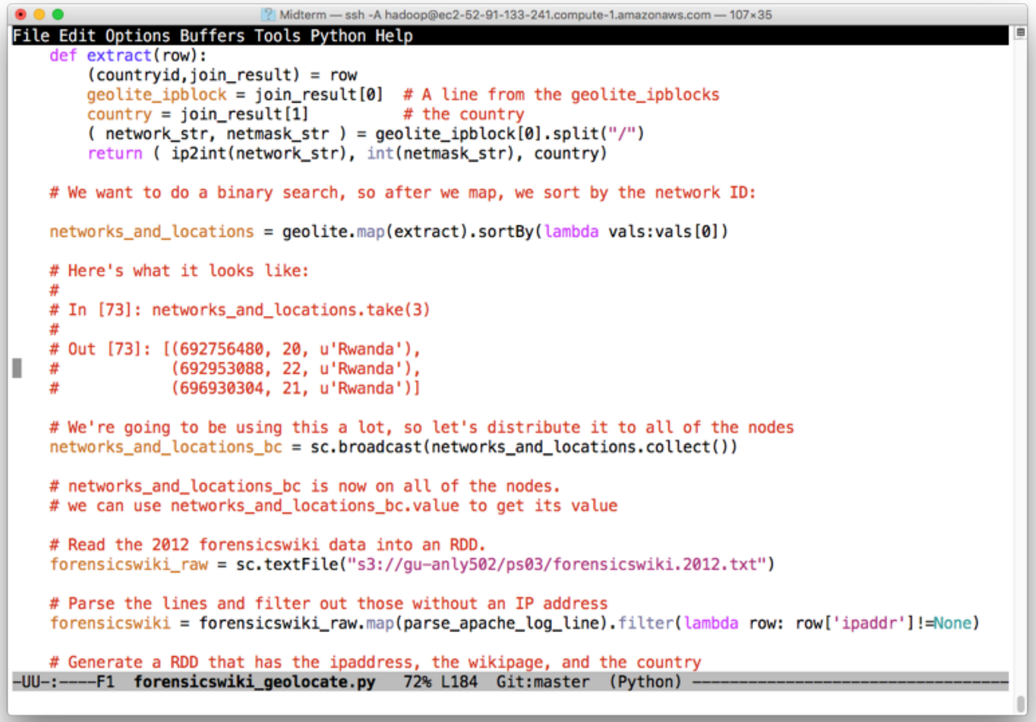
```
In [23]: geolite = geolite_ipblocks_by_countryid.join(geolite_locations_by_countryid)

In [24]: geolite.take(5)
Out[24]:
[(u'49518',
  ([u'196.12.140.0/22', u'49518', u'934292', u'', u'0', u'0'], u'Rwanda')),
 (u'49518',
  ([u'196.12.144.0/22', u'49518', u'934292', u'', u'0', u'0'], u'Rwanda')),
 (u'49518',
  ([u'196.12.153.0/24', u'49518', u'934292', u'', u'0', u'0'], u'Rwanda')),
 (u'49518',
  ([u'196.44.240.0/20', u'49518', u'49518', u'', u'0', u'0'], u'Rwanda')),
 (u'49518',
  ([u'196.49.7.0/24', u'49518', u'49518', u'', u'0', u'0'], u'Rwanda'))]
```

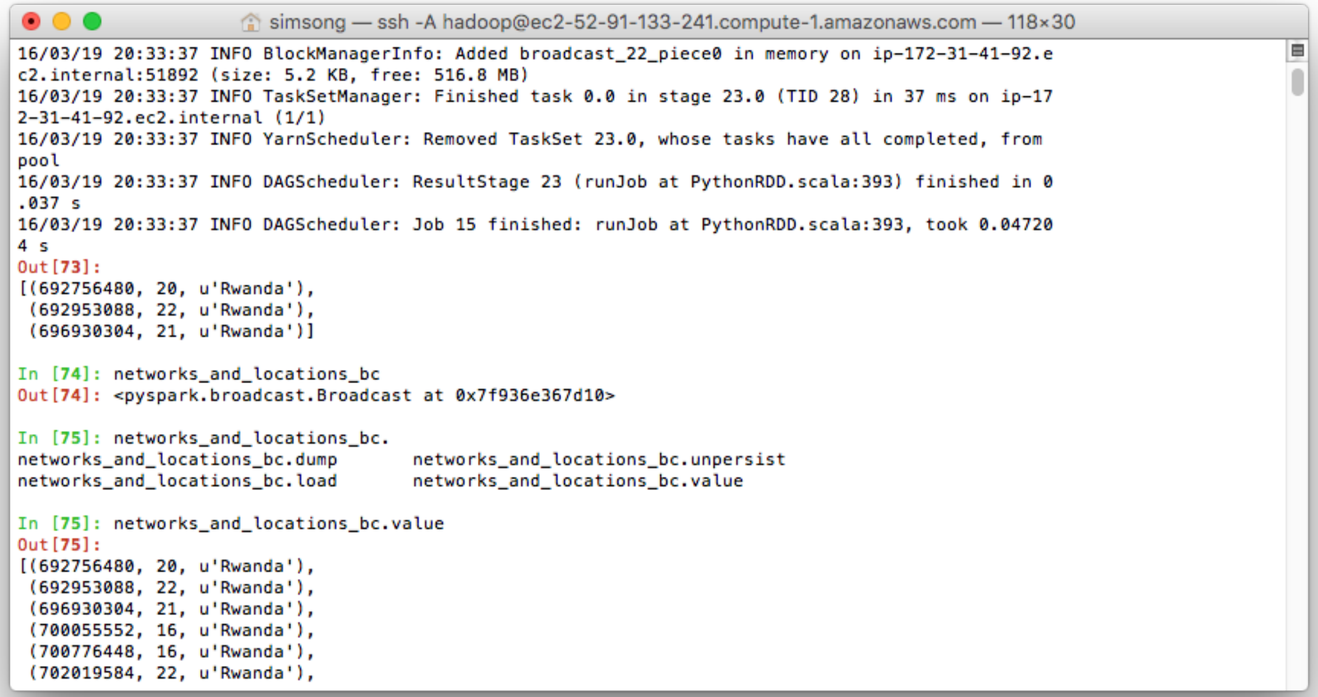
So let's try this:

```
    # geolite now looks like (countryid, ([geolite locations], countryname)]
    # Use a function that extracts the network CIDR block and the country name and returns as a list
    def extract(row):
        (countryid,join_result) = row
        geolite_ipblock = join_result[0]  # A line from the geolite_ipblocks
        country = join_result[1]          # the country
        ( network_str, netmask_str ) = geolite_ipblock[0].split("/")
        return ( ip2int(network_str), int(netmask_str), country)

    # We want to do a binary search, so after we map, we sort by the network ID:

    networks_and_locations = geolite.map(extract).sortBy(lambda vals:vals[0])
```

Remember — I'm switching back-and-forth between two windows...

```
In [73]: networks_and_locations.take(3)

Out[73]:
[(692756480, 20, u'Rwanda'),
 (692953088, 22, u'Rwanda'),
 (696930304, 21, u'Rwanda')]
```





Spark broadcast variables — read-only, transmitted from the

Broadcast Variables

Broadcast variables allow the programmer to keep a read-only variable cached on each machine rather than shipping a copy of it with tasks. They can be used, for example, to give every node a copy of a large input dataset in an efficient manner. Spark also attempts to distribute broadcast variables using efficient broadcast algorithms to reduce communication cost.

Spark actions are executed through a set of stages, separated by distributed "shuffle" operations. Spark automatically broadcasts the common data needed by tasks within each stage. The data broadcasted this way is cached in serialized form and deserialized before running each task. This means that explicitly creating broadcast variables is only useful when tasks across multiple stages need the same data or when caching the data in deserialized form is important.

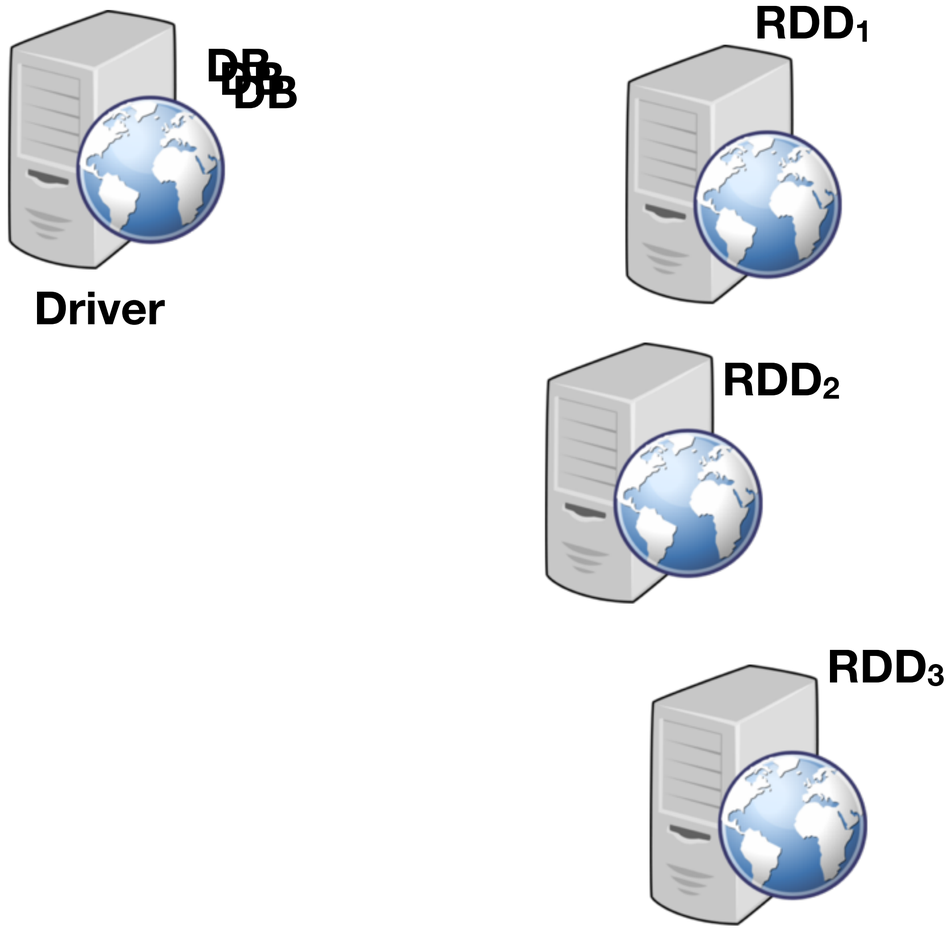
Broadcast variables are created from a variable `v` by calling `SparkContext.broadcast(v)`. The broadcast variable is a wrapper around `v`, and its value can be accessed by calling the `value` method. The code below shows this:

Scala | Java | Python

```
>>> broadcastVar = sc.broadcast([1, 2, 3])
<pyspark.broadcast.Broadcast object at 0x102789f10>

>>> broadcastVar.value
[1, 2, 3]
```

After the broadcast variable is created, it should be used instead of the value `v` in any functions run on the cluster so that `v` is not shipped to the nodes more than once. In addition, the object `v` should not be modified after it is broadcast in order to ensure that all nodes get the same value of the broadcast variable (e.g. if the variable is shipped to a new node later).

- http://spark.apache.org/docs/latest/programming-guide.html#broadcast-variables



Broadcast code

```
# We're going to be using this a lot, so let's distribute it to all of the nodes
networks_and_locations_bc = sc.broadcast(networks_and_locations.collect())

# networks_and_locations_bc is now on all of the nodes.
# we can use networks_and_locations_bc.value to get its value
```

So where are we:

- ✓Create the database — *Created as an RDD with a join (not really needed)*
- ✓Distribute the database to the nodes — *Broadcast Variables*
- Search the database

Improved forensicswiki parser

```
## Instead of throwing an error, it return a Row() object
## with all NULLs
from pyspark.sql import Row
import dateutil, dateutil.parser, re

APPACHE_COMBINED_LOG_REGEX = '([(\d\.)]+) [^ ]+ [^ ]+ \[(.*)\] "(.*)" (\d+) [^ ]+ ("(.*)")? ("(.*)")?'
WIKIPAGE_PATTERN = "(index.php\?title=|/wiki/)([^ &]*)"

appache_re  = re.compile(APPACHE_COMBINED_LOG_REGEX)
wikipage_re = re.compile(WIKIPAGE_PATTERN)

def parse_apache_log_line(logline):
    from dateutil import parser
    m = appache_re.match(logline)
    if m==None: return Row(ipaddr=None, timestamp = None, request = None, result = None,
                           user=None, referrer = None, agent = None, url = None, datetime = None,
                           date = None, time = None, wikipage = None)

    timestamp = m.group(2)
    request   = m.group(3)
    agent     = m.group(7).replace('"','') if m.group(7) else ''

    request_fields = request.split(" ")
    url         = request_fields[1] if len(request_fields)>2 else ""
    datetime    = parser.parse(timestamp.replace(":", " ", 1)).isoformat()
    (date,time) = (datetime[0:10],datetime[11:])

    n = wikipage_re.search(url)
    wikipage = n.group(2) if n else ""

    return Row( ipaddr = m.group(1), timestamp = timestamp, request = request,
        result = int(m.group(4)), user = m.group(5), referrer = m.group(6),
        agent = agent, url = url, datetime = datetime, date = date,
        time = time, wikipage = wikipage)
```

Make a RDD that has [IP address, WikiPage, Country]

```
# We will have a database of (network, netmask, country) tuples.
# The database is a sorted array.
# This function searches the database and returns the country or "" if there is no match
# The search is done with the IP address as a dotted-quad
def geo_search( db, ipaddr_str ):
    import bisect
    ipaddr_int = ip2int( ipaddr_str )
    idx = bisect.bisect_left( db, (ipaddr_int,0,""))
    idx -= 1                    # because the one to the left is the match!
    return db[idx][2] if netmatch( db[idx][0], db[idx][1], ipaddr_int ) else ""


# Read the 2012 forensicswiki data into an RDD.
forensicswiki_raw = sc.textFile("s3://gu-anly502/ps03/forensicswiki.2012.txt")

# Parse the lines and filter out those without an IP address
forensicswiki = forensicswiki_raw.map(parse_apache_log_line).filter(lambda row: row['ipaddr']!=None)


# Generate a RDD that has the ipaddress, the wikipage, and the country
forensicswiki_res = forensicswiki.map(
    lambda row: (row['ipaddr'],row['wikipage'],
                 geo_search(networks_and_locations_bc.value,row['ipaddr'])))
```

```
...
2373926912, 18, Finland
2373943296, 17, Finland
2373976064, 15, "United States"
2374107136, 16, Germany
2374172672, 16, "United States"
...
```

RDD now has [ipaddr, wikipage, country]

```
In [243]: forensicswiki_res.take(3)

Out[243]:
[(u'77.21.0.59', u'Write_Blockers', u'Germany'),
 (u'77.21.0.59', '', u'Germany'),
 (u'77.21.0.59', '', u'Germany')]
```

Use the wordcount pattern!

```
In [244]: forensicswiki_counts = forensicswiki_res.map(lambda x: ((x[2],x[1]),1))

In [245]: forensicswiki_counts.take(3)
Out[245]:
[((u'Germany', u'Write_Blockers'), 1),
 ((u'Germany', ''), 1),
 ((u'Germany', ''), 1)]
In [248]: import operator
In [249]: forensicswiki_sums = forensicswiki_counts.reduceByKey(operator.add)

In [250]: forensicswiki_sums.take(3)
...
```

(my code had a bug)

This is what I got:

```
---------------------------------------------------------------------------
Py4JJavaError                             Traceback (most recent call last)
<ipython-input-250-08bb410b8961> in <module>()
----> 1 forensicswiki_sums.take(3)

/usr/lib/spark/python/pyspark/rdd.py in take(self, num)
   1295
   1296             p = range(partsScanned, min(partsScanned + numPartsToTry, totalParts))
-> 1297             res = self.context.runJob(self, takeUpToNumLeft, p)
   1298
   1299             items += res

/usr/lib/spark/python/pyspark/context.py in runJob(self, rdd, partitionFunc, partitions, allowLocal)
    937         # SparkContext#runJob.
    938         mappedRDD = rdd.mapPartitions(partitionFunc)
--> 939         port = self._jvm.PythonRDD.runJob(self._jsc.sc(), mappedRDD._jrdd, partitions)
    940         return list(_load_from_socket(port, mappedRDD._jrdd_deserializer))
    941

/usr/lib/spark/python/lib/py4j-0.9-src.zip/py4j/java_gateway.py in __call__(self, *args)
    811         answer = self.gateway_client.send_command(command)
    812         return_value = get_return_value(
--> 813             answer, self.gateway_client, self.target_id, self.name)
    814
    815         for temp_arg in temp_args:

/usr/lib/spark/python/pyspark/sql/utils.py in deco(*a, **kw)
     43     def deco(*a, **kw):
     44         try:
---> 45             return f(*a, **kw)
     46         except py4j.protocol.Py4JJavaError as e:
     47             s = e.java_exception.toString()

/usr/lib/spark/python/lib/py4j-0.9-src.zip/py4j/protocol.py in get_return_value(answer, gateway_client, target_id, name)
    306                 raise Py4JJavaError(
    307                     "An error occurred while calling {0}{1}{2}.\n".
--> 308                     format(target_id, ".", name), value)
    309             else:
    310                 raise Py4JError(
```

I had bugs in both parse_apache_log_line and in geo_search

I developed these modules in forensicswiki_geolocate.py and tested them on non-RDD data.

Cycle:

1. `import forensicswiki_geolocate`
2. Develop code.
3. Save forensicswiki_geolocate.py
4. `reload(forensicswiki_geolocate.py)`
5. Try the code
6. Repeat

Example:

```
In [271]: reload(forensicswiki_geolocate)
Out[271]: <module 'forensicswiki_geolocate' from 'forensicswiki_geolocate.py'>

In [272]: forensicswiki_geolocate.print_most_popular_pages_per_country(results)
```

After a bit more ~~trial-and-error~~ iterative programming:

```
In [281]: forensicswiki_geolocate.print_most_popular_pages_per_country(results)

In [282]: reload(forensicswiki_geolocate)
Out[282]: <module 'forensicswiki_geolocate' from 'forensicswiki_geolocate.py'>

In [283]: forensicswiki_geolocate.print_most_popular_pages_per_country(results)
"American Samoa":
  Dd_rescue                          1


"Antigua and Barbuda":
  Tools:Network_Forensics           73
  MediaWiki:Print.css               16
  MediaWiki:Monobook.css            16


"Bonaire:
  Ddrescue                           3
  TCP_timestamps                     1
  Mozilla_Firefox_3_History_File_Format     1


"Bosnia and Herzegovina":
  MediaWiki:Common.css              39
  MediaWiki:Print.css               38
  MediaWiki:Monobook.css            38
...
```

Here is the code:

```
#
# This function expects an array of results where each result
# has the format ((country,wikipage), count)
def print_most_popular_pages_per_country(results):
    # This version prints out a format that is easier to undersatnd
    def print_top_3(counts_for_country):
        if not current_country: return
        import unicodedata
        print("{}:".format(unicodedata.normalize('NFKD',current_country).encode('ascii','ignore')))
        i = 0
        for (count,wikipage) in sorted(counts_for_country,reverse=True):
            if wikipage=="": continue # don't print empty pages
            if wikipage=="-": continue
            print("  {:30} {:-5}".format(wikipage,count))
            i += 1
            if i==3: break
        print("\n")

    current_country = None
    counts_for_country = []
    for result in results:
        country  = result[0][0]; wikipage = result[0][1]; count    = result[1]
        if country != current_country:
            print_top_3(counts_for_country)
            counts_for_country = []
            current_country = country
        counts_for_country.append((count,wikipage))
    print_top_3(counts_for_country)

# In [257]: forensicswiki_sums.take(2)
# Out[257]:
# [((u'Germany',
#   u'How+Pennsylvania+businesses+can+shop+for+a+lower+rate+for+their+organization+'), 4),
# ((u'Germany', u'Dealing+With+Bankruptcy'), 1)]

results = forensicswiki_sums.sortByKey(lambda x:x[0]).collect()

# We can actually process this in a single loop:
print_most_popular_pages_per_country(results)
```

note unicode issue

Output format is wrong....

Submit:

- **forensicswiki_geolocate.py** (your pyspark program)
- **forensicswiki_bycountry.txt** (output in the format "date \t count")
- **forensicswiki_popular.txt**
  (output in the format "country \t mostpopular1 \t mostpopular2 \t mostpopular3")

Ugh! What a lousy format...

You can just change the output function…

```
def print_most_popular_pages_per_country(results):
    # This format prints out what was requested, which is not a good format
    def print_top_3_requested(counts_for_country):
        if not current_country: return
        import unicodedata
        country = unicodedata.normalize('NFKD',current_country).encode('ascii','ignore')
        counts  = sorted(counts_for_country,reverse=True)
        if len(counts)<3:
            counts.append("")       # make sure that there are at least 3 entries
            counts.append("")
            counts.append("")
        out_bycountry.write("{}\t{}\n".format(country,total_for_country))
        out_popular.write("{}\t{}\t{}\t{}\n".format(country,counts[0],counts[1],counts[2]))

    out_bycountry = open("forensicswiki_bycountry.txt","w")
    out_popular   = open("forensicswiki_popular.txt","w")
    current_country = None
    counts_for_country = []
    total_for_country  = 0
    for result in results:
        country  = result[0][0]
        wikipage = result[0][1]
        count    = result[1]
        total_for_country += count
        if country != current_country:
            print_top_3_requested(counts_for_country)
            counts_for_country = []
            current_country = country
        counts_for_country.append((count,wikipage))
    print_top_3_requested(counts_for_country)
```

Here is the output

Time: 2818 seconds

```
$ ls -alt|head -2
total 116
-rw-rw-r--  1 hadoop hadoop  4133 Mar 20 20:30 forensicswiki_bycountry.txt
-rw-rw-r--  1 hadoop hadoop 14585 Mar 20 20:30 forensicswiki_popular.txt

$ head forensicswiki_bycountry.txt
"American Samoa"           84310
"Antigua and Barbuda"      84633
"Bonaire        89393
"Bosnia and Herzegovina"   90411
"British Virgin Islands"   90525
"Burkina Faso"  90805
"Cape Verde"    91212
"Cayman Islands"           95230
"Costa Rica"    146066
"Czech Republic"           170052

$ head forensicswiki_popular.txt
"American Samoa"           (8, '')    (1, u'Dd_rescue')
"Antigua and Barbuda"      (667, '')  (73, u'Tools:Network_Forensics') (32, u'-')
"Bonaire        (38, '')   (3, u'Ddrescue')      (1, u'TCP_timestamps')
"Bosnia and Herzegovina"   (4753, '') (77, u'-') (39, u'MediaWiki:Common.css')
"British Virgin Islands"   (73, '')   (2, u'Word_Document_(DOCX)')     (2, u'Talk:Word_Document_(DOCX)')
"Burkina Faso"  (105, '')  (19, u'AT_Commands')  (2, u'Bulk_extractor')
"Cape Verde"    (256, '')  (6, u'Ddrescue')      (4, u'DNA')
"Cayman Islands"           (345, '')  (18, u'Tools')        (3, u'Write_Blockers')
"Costa Rica"    (3959, '') (32, u'Tools')        (32, u'Ddrescue')
"Czech Republic"           (50151, '')          (1387, u'Main_Page')  (562, u'-')
```

This was a super-hard problem. How could you break it into smaller pieces?

Break the problem into pieces and get each working by itself WITHOUT SPARK.

Examples:

- Manually geolocate a few IP addresses (to see how it works)
- Write single-threaded code running locally to geolocate an IP address.

```
$ wc GeoLite2-Country-*
 183888  183888 6666094 GeoLite2-Country-Blocks-IPv4.csv
    250     422    9275 GeoLite2-Country-Locations-en.csv
 184138  184310 6675369 total
```

—*The database is only 183,888 lines long!*

- Get the JOIN working with Spark at the command line (ipyspark), then copy the code into your program.
- Create an RDD with the Forensicswiki logfile lines
- Create an RDD with just the IP addresses and the Wikipage
- Create a function that you can use to MAP the IP address to a geographical location
- Run the function on LOCAL DATA (with collect)
- Run the function with a BROADCAST VARIABLE

My solution took ≈ 45 minutes to run...

How could we make it go faster? Measure and experiment...

Which is faster....

Local variable	Broadcast Variable	RDD
Geolocate each IP address	Make a SET of all IP addresses & Geolocate	Cache IP address geolocate
Search with bisect	Linear search	
Converting IP addresses to integers	Working with IP addresses as strings	
1 m3.xlarge node	2 m3.xlarge nodes	1 m3.x2large node

Or is it something else entirely?

Add local caching.

Before:

```
def geo_search( db, ipaddr_str ):
    import bisect
    ipaddr_int = ip2int( ipaddr_str )
    idx = bisect.bisect_left( db, (ipaddr_int,0,""))
    idx -= 1                    # because the one to the left is the match!
    return db[idx][2] if netmatch( db[idx][0], db[idx][1], ipaddr_int ) else ""
```

After:

```
geocache = {}
def geo_search( db, ipaddr_str ):
    import bisect
    if ipaddr_str not in geocache:
        ipaddr_int = ip2int( ipaddr_str )
        idx = bisect.bisect_left( db, (ipaddr_int,0,""))
        idx -= 1                    # because the one to the left is the match!
        geocache[ipaddr_str] = db[idx][2] if netmatch( db[idx][0], db[idx][1], ipaddr_int ) else ""
    return geocache[ipaddr_str]
```

Note: Each node will maintain its own cache

Caching results: 2818 seconds → 2682 seconds (5% improvement)

Turn off broadcast: 2682 seconds → 2675 seconds (within error measurement).

- How do we improve this more?

Look at the stages … stage 9 does take the longest. What's Stage 9?

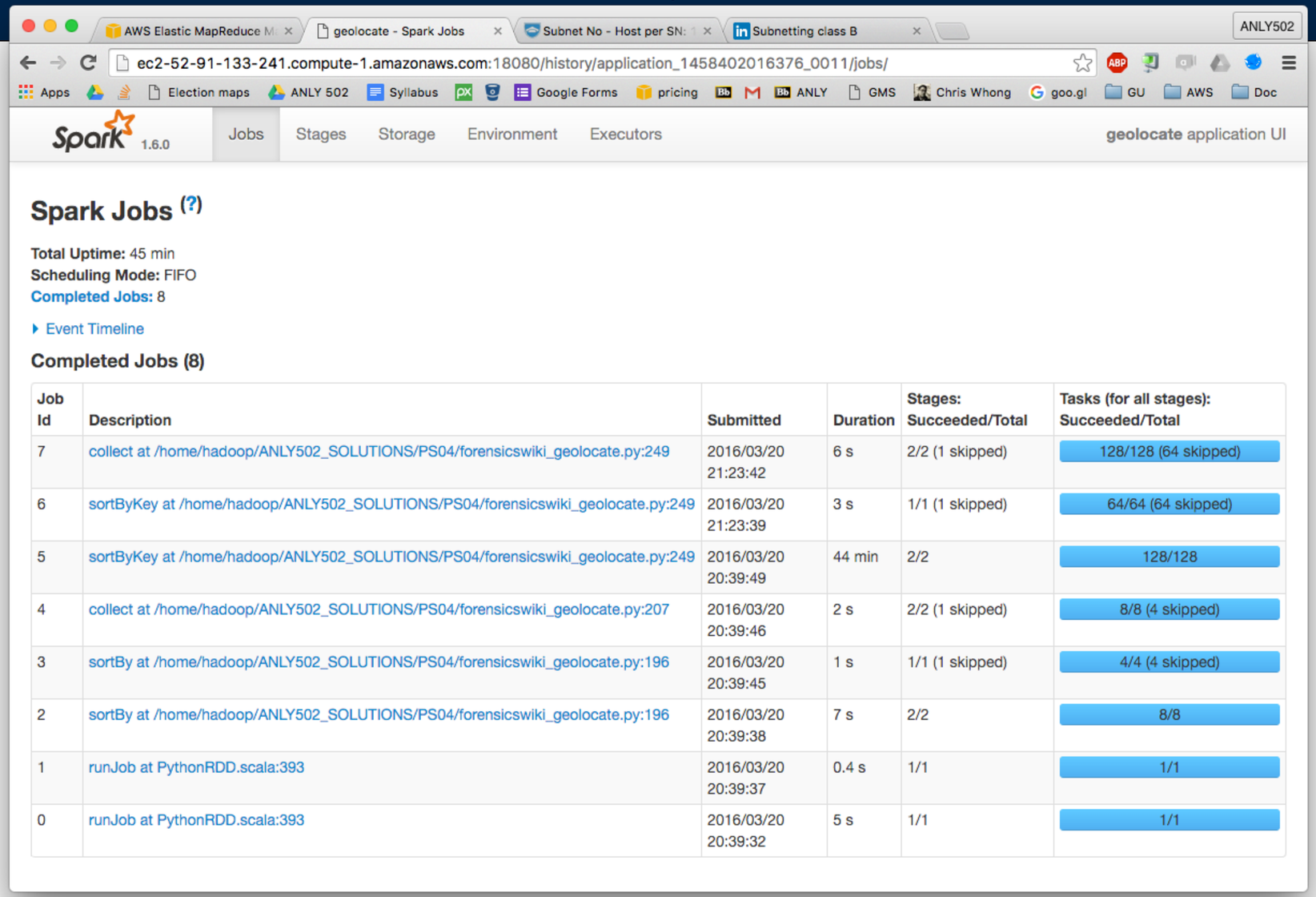
Stage 9:

```
16/03/20 21:38:05 INFO TaskSetManager: Finished task 6.0 in stage 9.0 (TID 28) in 101438 ms on ip-172-31-41-92.ec2.internal (7/64)
16/03/20 21:38:12 INFO TaskSetManager: Starting task 9.0 in stage 9.0 (TID 31, ip-172-31-41-92.ec2.internal, partition 9,RACK_LOCAL, 2134 bytes)
16/03/20 21:38:12 INFO TaskSetManager: Finished task 7.0 in stage 9.0 (TID 29) in 97163 ms on ip-172-31-41-92.ec2.internal (8/64)
16/03/20 21:39:27 INFO TaskSetManager: Starting task 10.0 in stage 9.0 (TID 32, ip-172-31-41-92.ec2.internal, partition 10,RACK_LOCAL, 2134 bytes)
16/03/20 21:39:27 INFO TaskSetManager: Finished task 8.0 in stage 9.0 (TID 30) in 81940 ms on ip-172-31-41-92.ec2.internal (9/64)
16/03/20 21:39:33 INFO TaskSetManager: Starting task 11.0 in stage 9.0 (TID 33, ip-172-31-41-92.ec2.internal, partition 11,RACK_LOCAL, 2134 bytes)
16/03/20 21:39:33 INFO TaskSetManager: Finished task 9.0 in stage 9.0 (TID 31) in 81375 ms on ip-172-31-41-92.ec2.internal (10/64)
16/03/20 21:40:37 INFO TaskSetManager: Starting task 12.0 in stage 9.0 (TID 34, ip-172-31-41-92.ec2.internal, partition 12,RACK_LOCAL, 2134 bytes)
16/03/20 21:40:37 INFO TaskSetManager: Finished task 10.0 in stage 9.0 (TID 32) in 70533 ms on ip-172-31-41-92.ec2.internal (11/64)
16/03/20 21:40:42 INFO TaskSetManager: Starting task 13.0 in stage 9.0 (TID 35, ip-172-31-41-92.ec2.internal, partition 13,RACK_LOCAL, 2134 bytes)
16/03/20 21:40:42 INFO TaskSetManager: Finished task 11.0 in stage 9.0 (TID 33) in 68849 ms on ip-172-31-41-92.ec2.internal (12/64)
16/03/20 21:41:44 INFO TaskSetManager: Starting task 14.0 in stage 9.0 (TID 36, ip-172-31-41-92.ec2.internal, partition 14,RACK_LOCAL, 2134 bytes)
16/03/20 21:41:44 INFO TaskSetManager: Finished task 12.0 in stage 9.0 (TID 34) in 66984 ms on ip-172-31-41-92.ec2.internal (13/64)
16/03/20 21:41:48 INFO TaskSetManager: Starting task 15.0 in stage 9.0 (TID 37, ip-172-31-41-92.ec2.internal, partition 15,RACK_LOCAL, 2134 bytes)
```

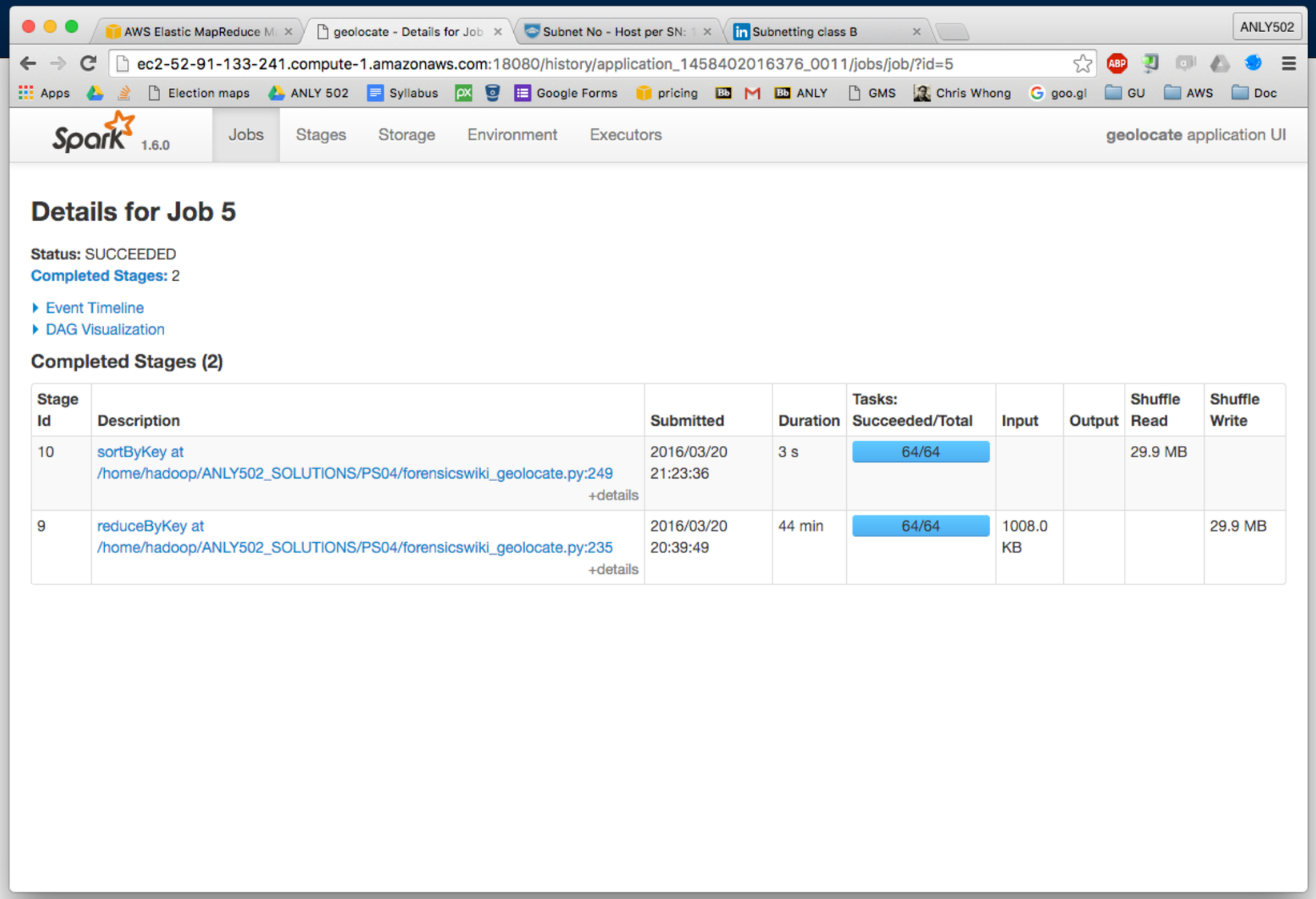
Stage 15:

```
16/03/20 21:23:48 INFO TaskSetManager: Finished task 51.0 in stage 15.0 (TID 329) in 53 ms on ip-172-31-41-92.ec2.internal (52/64)
16/03/20 21:23:48 INFO TaskSetManager: Starting task 54.0 in stage 15.0 (TID 332, ip-172-31-41-92.ec2.internal, partition 54,NODE_LOCAL, 1894 bytes)
16/03/20 21:23:48 INFO TaskSetManager: Finished task 52.0 in stage 15.0 (TID 330) in 45 ms on ip-172-31-41-92.ec2.internal (53/64)
16/03/20 21:23:48 INFO TaskSetManager: Starting task 55.0 in stage 15.0 (TID 333, ip-172-31-41-92.ec2.internal, partition 55,NODE_LOCAL, 1894 bytes)
16/03/20 21:23:48 INFO TaskSetManager: Finished task 53.0 in stage 15.0 (TID 331) in 80 ms on ip-172-31-41-92.ec2.internal (54/64)
16/03/20 21:23:48 INFO TaskSetManager: Starting task 56.0 in stage 15.0 (TID 334, ip-172-31-41-92.ec2.internal, partition 56,NODE_LOCAL, 1894 bytes)
16/03/20 21:23:48 INFO TaskSetManager: Finished task 54.0 in stage 15.0 (TID 332) in 73 ms on ip-172-31-41-92.ec2.internal (55/64)
16/03/20 21:23:48 INFO TaskSetManager: Starting task 57.0 in stage 15.0 (TID 335, ip-172-31-41-92.ec2.internal, partition 57,NODE_LOCAL, 1894 bytes)
16/03/20 21:23:48 INFO TaskSetManager: Finished task 56.0 in stage 15.0 (TID 334) in 35 ms on ip-172-31-41-92.ec2.internal (56/64)
16/03/20 21:23:48 INFO TaskSetManager: Starting task 58.0 in stage 15.0 (TID 336, ip-172-31-41-92.ec2.internal, partition 58,NODE_LOCAL, 1894 bytes)
16/03/20 21:23:48 INFO TaskSetManager: Finished task 55.0 in stage 15.0 (TID 333) in 70 ms on ip-172-31-41-92.ec2.internal (57/64)
16/03/20 21:23:48 INFO TaskSetManager: Starting task 59.0 in stage 15.0 (TID 337, ip-172-31-41-92.ec2.internal, partition 59,NODE_LOCAL, 1894 bytes)
16/03/20 21:23:48 INFO TaskSetManager: Finished task 57.0 in stage 15.0 (TID 335) in 86 ms on ip-172-31-41-92.ec2.internal (58/64)
16/03/20 21:23:48 INFO TaskSetManager: Starting task 60.0 in stage 15.0 (TID 338, ip-172-31-41-92.ec2.internal, partition 60,NODE_LOCAL, 1894 bytes)
16/03/20 21:23:48 INFO TaskSetManager: Finished task 58.0 in stage 15.0 (TID 336) in 88 ms on ip-172-31-41-92.ec2.internal (59/64)
```

Use Spark's built-in tools to see where the delay is.



Job Id	Description	Submitted	Duration	Stages: Succeeded/Total	Tasks (for all stages): Succeeded/Total
7	collect at /home/hadoop/ANLY502_SOLUTIONS/PS04/forensicswiki_geolocate.py:249	2016/03/20 21:23:42	6 s	2/2 (1 skipped)	128/128 (64 skipped)
6	sortByKey at /home/hadoop/ANLY502_SOLUTIONS/PS04/forensicswiki_geolocate.py:249	2016/03/20 21:23:39	3 s	1/1 (1 skipped)	64/64 (64 skipped)
5	sortByKey at /home/hadoop/ANLY502_SOLUTIONS/PS04/forensicswiki_geolocate.py:249	2016/03/20 20:39:49	44 min	2/2	128/128
4	collect at /home/hadoop/ANLY502_SOLUTIONS/PS04/forensicswiki_geolocate.py:207	2016/03/20 20:39:46	2 s	2/2 (1 skipped)	8/8 (4 skipped)
3	sortBy at /home/hadoop/ANLY502_SOLUTIONS/PS04/forensicswiki_geolocate.py:196	2016/03/20 20:39:45	1 s	1/1 (1 skipped)	4/4 (4 skipped)
2	sortBy at /home/hadoop/ANLY502_SOLUTIONS/PS04/forensicswiki_geolocate.py:196	2016/03/20 20:39:38	7 s	2/2	8/8
1	runJob at PythonRDD.scala:393	2016/03/20 20:39:37	0.4 s	1/1	1/1
0	runJob at PythonRDD.scala:393	2016/03/20 20:39:32	5 s	1/1	1/1



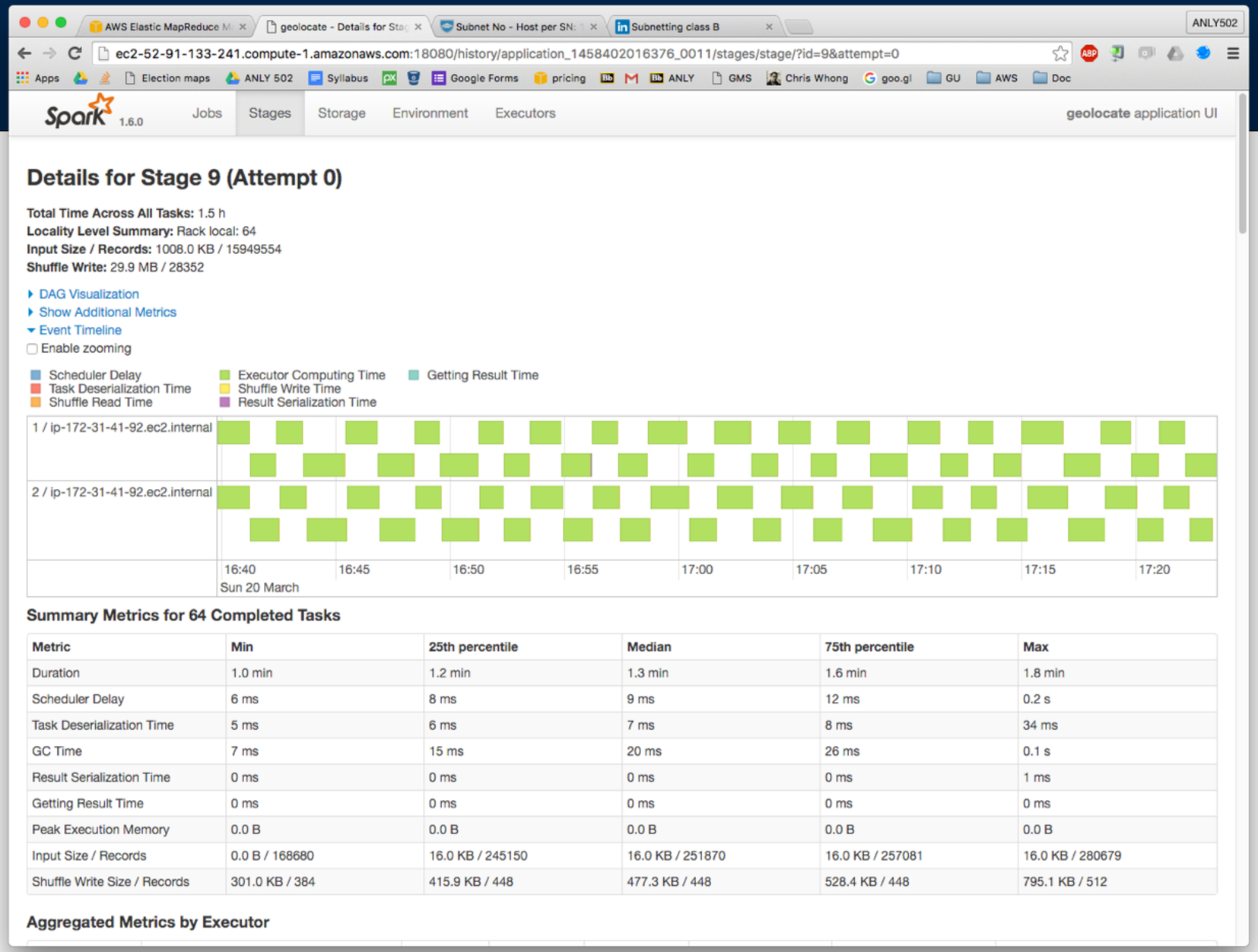
Don't optimize before you measure! Reduce by key is the main time consumer!

Moral:

- Global sorting is SLOW.

Question:

- Could we run faster with more than two executors?



Metric	Min	25th percentile	Median	75th percentile	Max
Duration	1.0 min	1.2 min	1.3 min	1.6 min	1.8 min
Scheduler Delay	6 ms	8 ms	9 ms	12 ms	0.2 s
Task Deserialization Time	5 ms	6 ms	7 ms	8 ms	34 ms
GC Time	7 ms	15 ms	20 ms	26 ms	0.1 s
Result Serialization Time	0 ms	0 ms	0 ms	0 ms	1 ms
Getting Result Time	0 ms	0 ms	0 ms	0 ms	0 ms
Peak Execution Memory	0.0 B	0.0 B	0.0 B	0.0 B	0.0 B
Input Size / Records	0.0 B / 168680	16.0 KB / 245150	16.0 KB / 251870	16.0 KB / 257081	16.0 KB / 280679
Shuffle Write Size / Records	301.0 KB / 384	415.9 KB / 448	477.3 KB / 448	528.4 KB / 448	795.1 KB / 512

More on tuning:

- http://blog.cloudera.com/blog/2015/03/how-to-tune-your-apache-spark-jobs-part-2/

Four executors...

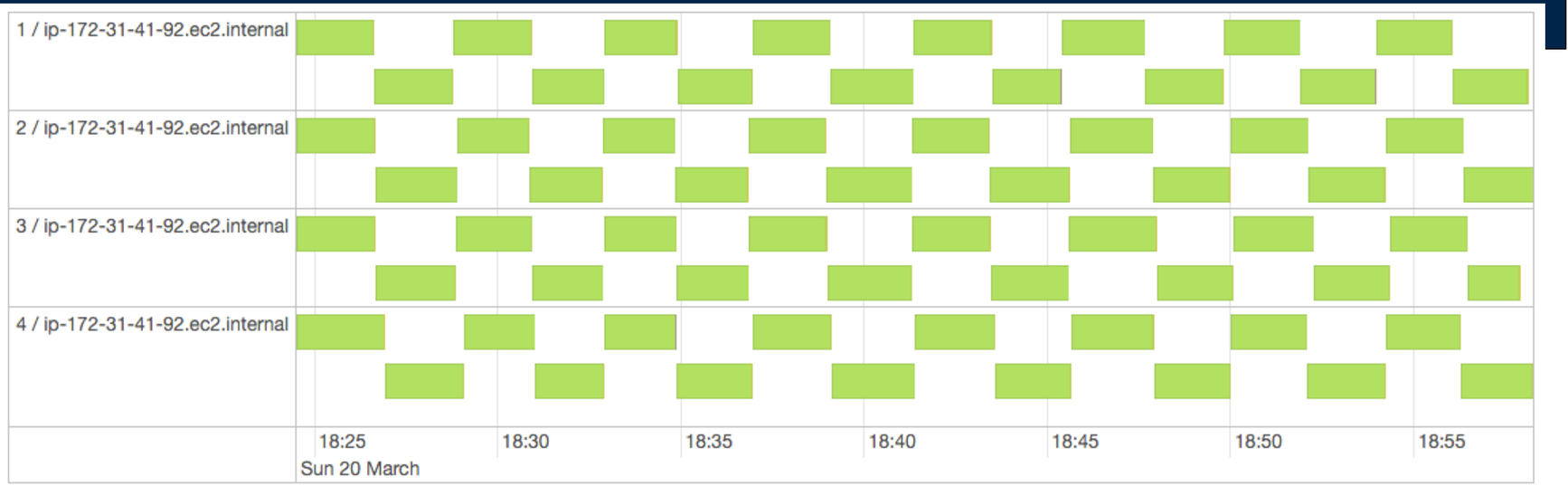
```
$ spark-submit --num-executors 4 forensicswiki_geolocate.py

In another window:
$ top
```

```
top - 22:44:12 up 1 day,  7:06,  3 users,  load average: 4.26, 4.47, 3.76
Tasks: 189 total,   5 running, 182 sleeping,   2 stopped,   0 zombie
Cpu(s): 95.6%us,  3.4%sy,  0.0%ni,  0.0%id,  0.0%wa,  0.0%hi,  1.0%si,  0.0%st
Mem:  15407244k total, 14979340k used,   427904k free,    74572k buffers
Swap:        0k total,        0k used,        0k free,  1383908k cached

  PID USER      PR  NI  VIRT  RES  SHR S %CPU %MEM    TIME+  COMMAND
27018 yarn      20   0  316m  69m 3768 R 96.6  0.5  17:42.51 python
27004 yarn      20   0  318m  71m 3712 R 95.6  0.5  17:47.58 python
27111 yarn      20   0  317m  70m 3872 R 89.3  0.5  17:41.74 python
27132 yarn      20   0  316m  69m 3812 R 88.3  0.5  17:44.77 python
 4908 yarn      20   0 2980m 400m  28m S  6.0  2.7  33:07.86 java
26672 yarn      20   0 1946m 697m  40m S  5.6  4.6   0:51.10 java
26762 yarn      20   0 1919m 645m  40m S  4.0  4.3   0:43.55 java
26671 yarn      20   0 1941m 687m  40m S  2.6  4.6   0:46.20 java
 9271 oozie     20   0 3511m 215m  24m S  2.3  1.4  38:00.15 java
26622 yarn      20   0 1943m 695m  40m S  2.0  4.6   0:44.33 java
 4479 yarn      20   0 3373m 420m  28m S  1.0  2.8  14:21.31 java
 3810 hdfs      20   0 2530m 565m  27m S  0.7  3.8   7:07.61 java
 3941 hdfs      20   0 1508m 186m  28m S  0.7  1.2   4:08.51 java
```

Results with 4 executors

Time with 2 executors: 2675

Time with 4 executors: 2097



Summary Metrics for 64 Completed Tasks

2.1 min * 64 tasks / 4 processors = 33.6 min

Metric	Min	25th percentile	Median	75th percentile	Max
Duration	1.4 min	2.0 min	2.1 min	2.2 min	2.4 min
Scheduler Delay	8 ms	13 ms	16 ms	18 ms	31 ms
Task Deserialization Time	5 ms	10 ms	12 ms	16 ms	96 ms
GC Time	8 ms	16 ms	24 ms	37 ms	0.4 s
Result Serialization Time	0 ms	0 ms	0 ms	0 ms	1 ms
Getting Result Time	0 ms	0 ms	0 ms	0 ms	0 ms
Peak Execution Memory	0.0 B	0.0 B	0.0 B	0.0 B	0.0 B
Input Size / Records	0.0 B / 168680	16.0 KB / 245150	16.0 KB / 251870	16.0 KB / 257081	16.0 KB / 280679
Shuffle Write Size / Records	300.9 KB / 384	415.9 KB / 448	477.7 KB / 448	528.5 KB / 448	795.5 KB / 512

Metrics by Executor

Aggregated Metrics by Executor

Executor ID ▲	Address	Task Time	Total Tasks	Failed Tasks	Succeeded Tasks	Input Size / Records	Shuffle Write Size / Records
1	ip-172-31-41-92.ec2.internal:50961	34 min	16	0	16	256.0 KB / 3997311	7.3 MB / 7040
2	ip-172-31-41-92.ec2.internal:53089	34 min	16	0	16	240.0 KB / 4007248	7.5 MB / 7168
3	ip-172-31-41-92.ec2.internal:47739	33 min	16	0	16	256.0 KB / 3949871	7.4 MB / 6976
4	ip-172-31-41-92.ec2.internal:53497	34 min	16	0	16	256.0 KB / 3995124	7.7 MB / 7168

Final Projects

What makes a good final project?

Reinforce something you've learned.

Learn something new.

Matched to your skill level.

Prepares you for a job that you want.

Use:

- Publicly available data source.
- Technology that we've learned in class — or a similar technology elsewhere. (Try Google's Cloud!)

Don't forget your deliverables:

- Project proposals (2)
- Project team (1)
- Presentation (15 minutes) (May 2)
- Write-up (5-10 pages) (May 13)

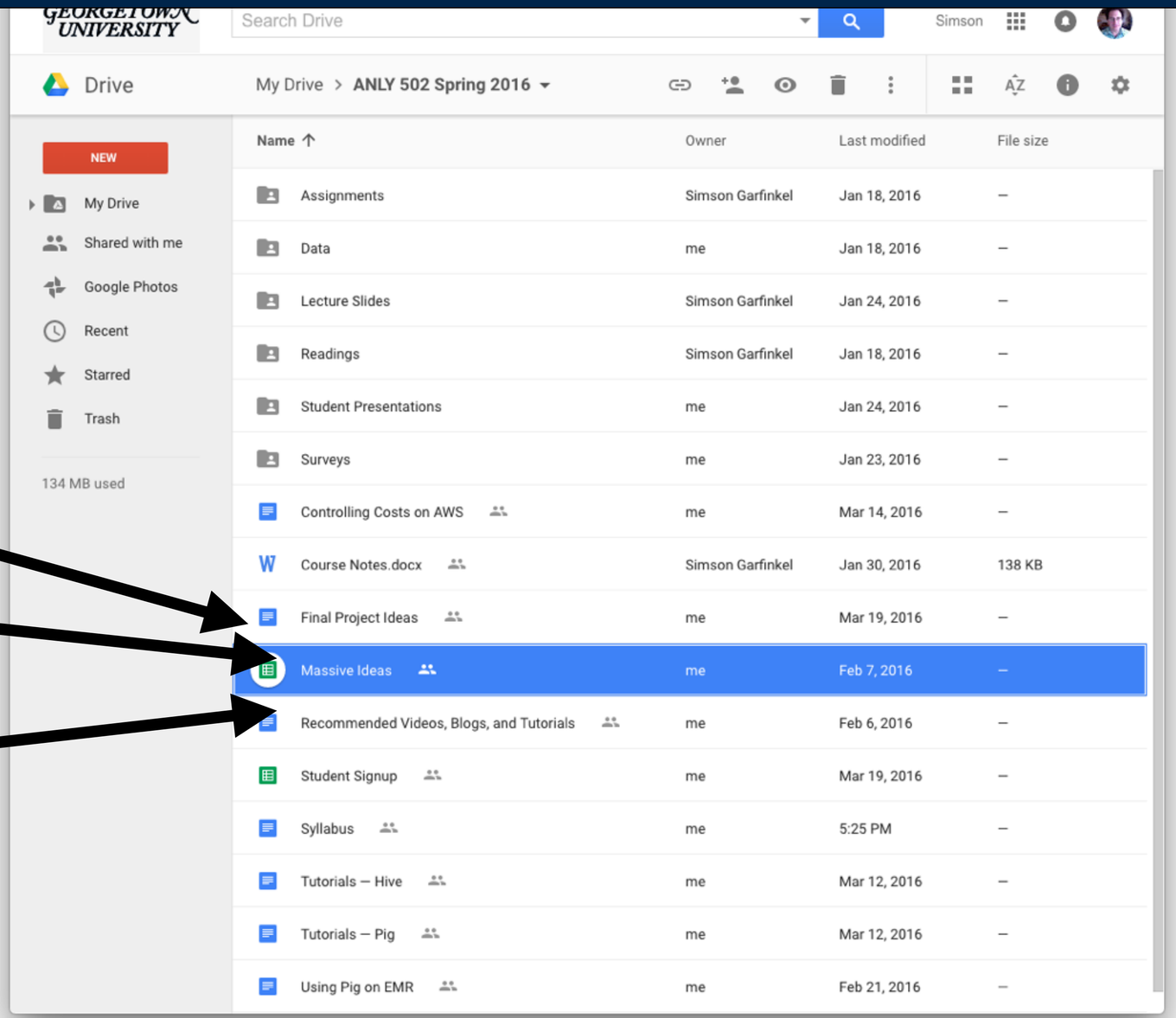
  —I will be available for in-person meetings after May 2nd.

Final project @ Google Drive:

Ideas Here!

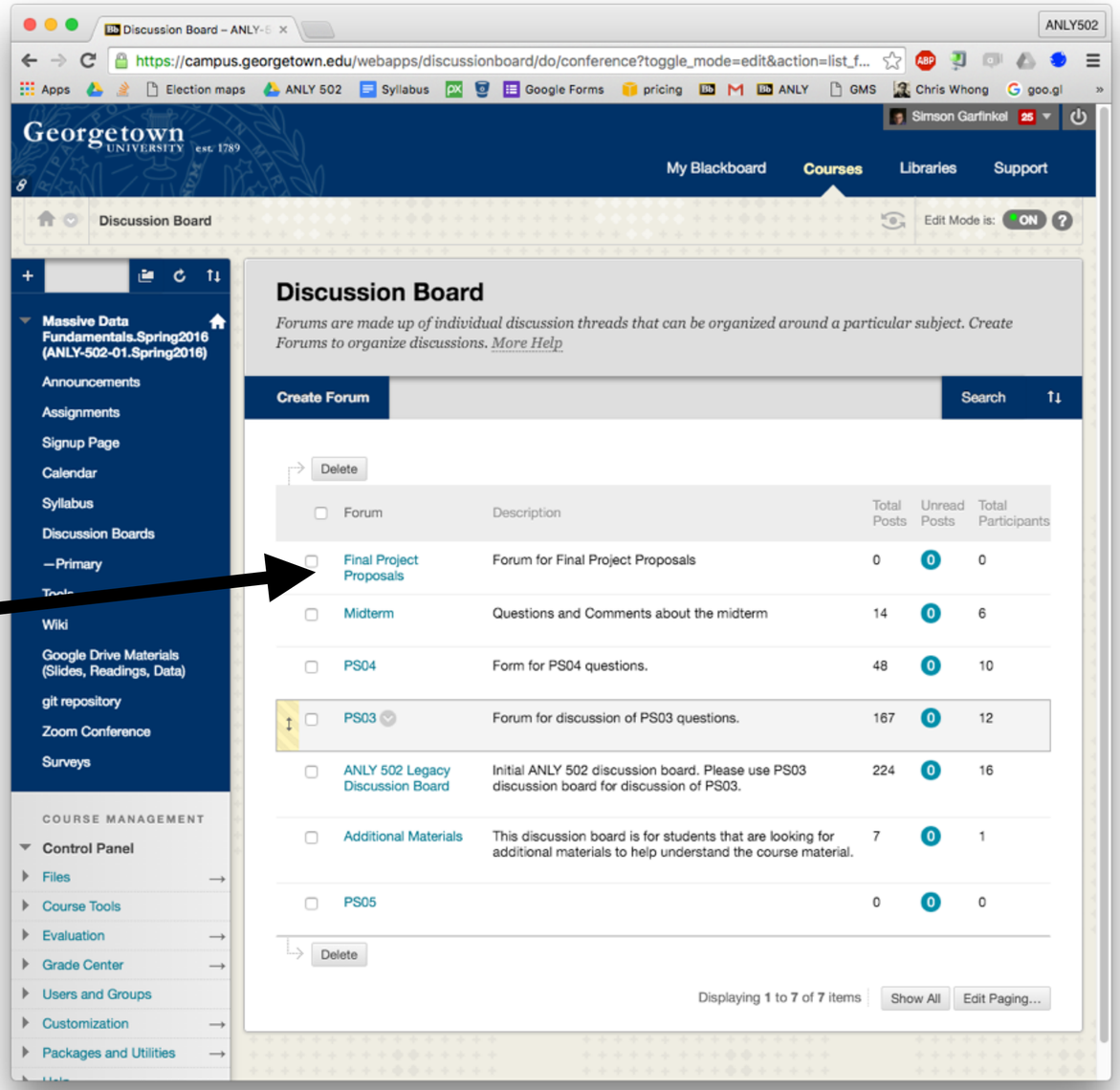
More ideas here!

Even more ideas!



Put your two proposals in the discussion board

Here!



Upcoming

March 21 — PS05 Assigned

March 22 — Final Project Proposals Due (2)

April 1 — Final Project Group Proposal

- Each group member must submit the same proposal on Blackboard!
- Blackboard groups will be created.

April 22 — PS05 Due

Mon, April 4 — L09: LLNL #1: Big Data in High Performance Computing

Mon, April 11 — L10: LLNL #2: Data Visualization

Mon, April 18 — L11: LLNL #3: Deep Learning

Mon, April 25 — L12: LLNL #4: Scientific Data Analysis Approaches, Architectures, and Workflow Systems

Mon, May 2 — L13: Final Project Presentations.

Fri, May 13 — Final Projects Due

We will continue to have student presentations during April

Simson Garfinkel will stay after class to answer questions