Joshua Harrison
Cloud Computing and Big Data
25/08/20

Part 1.

Prerequisite - Setup

For section one an 11 node spark cluster was set up on AWS. The cluster was configured using Flinkrock (see Appendix 2), and each node was a m5.large machine in aws. In order to create/destroy the cluster using Flintrock a makefile was created with several commands to automate launching, connecting to, sending data to, and destroying the cluster (see appendix 2). This proved invaluable as it meant clusters could be constructed and torn down with ease with no requirement to manually configure any of the properties of the nodes. The ephemeral nature of being able to take down clusters at ease is powerful and inline with 'the pets vs cattle' devops philosophy.

```
NODES=4

# CLUSTER=clo-spark-cluster-lt-${NODES}-node
CLUSTER=clo-spark-cluster-lt-q2-${NODES}-node

flintrock_launch:
   flintrock launch ${CLUSTER} --num-slaves ${NODES}
   flintrock run-command ${CLUSTER} 'sudo yum update -y'
   flintrock run-command ${CLUSTER} 'sudo yum -y install python36 python36-pip'
   flintrock run-command ${CLUSTER} 'sudo pip install virtualenvwrapper'
   flintrock copy-file ${CLUSTER} clo_bashrc /home/ec2-user/.bashrc
   flintrock copy-file ${CLUSTER} part_1.ipynb /home/ec2-user/part_1.ipynb
   flintrock copy-file ${CLUSTER} part_2.py /home/ec2-user/part_2.py
   flintrock copy-file ${CLUSTER} requirements.txt /home/ec2-user/requirements.txt
   flintrock copy-file ${CLUSTER} chicago.shp /home/ec2-user/chicago.shp
   flintrock copy-file ${CLUSTER} chicago.dbf /home/ec2-user/chicago.dbf
   flintrock copy-file ${CLUSTER} chicago.prj /home/ec2-user/chicago.prj
   flintrock copy-file ${CLUSTER} chicago.shx /home/ec2-user/chicago.shx
   flintrock run-command ${CLUSTER} 'source .bashrc'
   flintrock run-command ${CLUSTER} 'mkvirtualenv clo --python=`which python3`'
   flintrock run-command ${CLUSTER} 'workon clo'
   flintrock run-command ${CLUSTER} 'sudo yum install gcc gcc-c++ -y'
   # https://docs.aws.amazon.com/elasticbeanstalk/latest/dg/eb-cli3-install-linux.html
   flintrock run-command ${CLUSTER} 'curl -O https://bootstrap.pypa.io/get-pip.py &&
python3 get-pip.py --user'
   flintrock run-command ${CLUSTER} 'workon clo && pip3.6 install pip install
ipykernel'
```

```
   flintrock run-command ${CLUSTER} 'workon clo && pip3.6 install jupyter'
   flintrock run-command ${CLUSTER} 'workon clo && pip3.6 install -r requirements.txt'
   # create jupyter python 3 kernel -
https://ipython.readthedocs.io/en/6.5.0/install/kernel_install.html#kernels-for-differ
ent-environments
   flintrock run-command ${CLUSTER}  'workon clo && sudo `which python` -m ipykernel
install --name "python3"'
   flintrock run-command ${CLUSTER} 'workon clo &&  export PYSPARK_PYTHON=`which
python`'
   flintrock run-command ${CLUSTER} 'echo "pyspark --master spark://0.0.0.0:7077
--packages  org.apache.hadoop:hadoop-aws:2.7.4" > run.sh && chmod +x run.sh'
```

FIgure 1: extract of makefile used to launch new clusters. Notice the NODES variable set at the top of the file - this can be changed to create a cluster with an arbitrary number of nodes. (see appendix 2 for full makefile).

clo-spark-cluster-lt-master	i-094d520abfb4972ac	Stopped	m5.large	–	No alarms +
clo-spark-cluster-lt-slave	i-0a5b09f31cf48b318	Stopped	m5.large	–	No alarms +
clo-spark-cluster-lt-slave	i-0ccd7a7df3b1763d8	Stopped	m5.large	–	No alarms +
clo-spark-cluster-lt-slave	i-0b05a6fc1d962e301	Stopped	m5.large	–	No alarms +
clo-spark-cluster-lt-slave	i-033ded6a8e80fa102	Stopped	m5.large	–	No alarms +
clo-spark-cluster-lt_test-master	i-0859692b389f91df2	Stopped	m5.large	–	No alarms +
clo-spark-cluster-lt_test-slave	i-02ee563ddf74e8178	Stopped	m5.large	–	No alarms +
clo-spark-cluster-lt-2-node-slave	i-06371752d5076b402	Stopped	m5.large	–	No alarms +
clo-spark-cluster-lt-2-node-master	i-02724eb2046eafe39	Stopped	m5.large	–	No alarms +
clo-spark-cluster-lt-2-node-slave	i-0021ae40855b0f005	Stopped	m5.large	–	No alarms +
clo-spark-cluster-lt-10-node-slave	i-0509f266b7dec9a1d	Running	m5.large	2/2 checks ...	No alarms +
clo-spark-cluster-lt-10-node-slave	i-0a36dcf1feef8b372	Running	m5.large	2/2 checks ...	No alarms +
clo-spark-cluster-lt-10-node-slave	i-0220fb9d66e417eec	Running	m5.large	2/2 checks ...	No alarms +
clo-spark-cluster-lt-10-node-slave	i-06cf98e53226a67f5	Running	m5.large	2/2 checks ...	No alarms +
clo-spark-cluster-lt-10-node-slave	i-06ef47bbf997c9811	Running	m5.large	2/2 checks ...	No alarms +
clo-spark-cluster-lt-10-node-master	i-00b2179de51b46c07	Running	m5.large	2/2 checks ...	No alarms +
clo-spark-cluster-lt-10-node-slave	i-09002a3f129faa03a	Running	m5.large	2/2 checks ...	No alarms +
clo-spark-cluster-lt-10-node-slave	i-06273feab5fc4e757	Running	m5.large	2/2 checks ...	No alarms +
clo-spark-cluster-lt-10-node-slave	i-0f4e0b637e1b90a7e	Running	m5.large	2/2 checks ...	No alarms +
clo-spark-cluster-lt-10-node-slave	i-07d2a90b58defb18d	Running	m5.large	2/2 checks ...	No alarms +

Figure 2: list of EC2 instances created via the flintrock_launch script shown in Figure 1. Note, the mix of live and dead instances.

▾ Workers (10)

Worker Id	Address	State	Cores	Memory
worker-20200823223342-172.31.13.173-45411	172.31.13.173:45411	ALIVE	2 (2 Used)	6.6 GB (1024.0 MB Used)
worker-20200823223342-172.31.14.107-46757	172.31.14.107:46757	ALIVE	2 (2 Used)	6.6 GB (1024.0 MB Used)
worker-20200823223342-172.31.2.189-46453	172.31.2.189:46453	ALIVE	2 (2 Used)	6.6 GB (1024.0 MB Used)
worker-20200823223342-172.31.3.215-42985	172.31.3.215:42985	ALIVE	2 (2 Used)	6.5 GB (1024.0 MB Used)
worker-20200823223342-172.31.4.12-41995	172.31.4.12:41995	ALIVE	2 (2 Used)	6.5 GB (1024.0 MB Used)
worker-20200823223342-172.31.5.53-42685	172.31.5.53:42685	ALIVE	2 (2 Used)	6.6 GB (1024.0 MB Used)
worker-20200823223342-172.31.5.91-41563	172.31.5.91:41563	ALIVE	2 (2 Used)	6.6 GB (1024.0 MB Used)
worker-20200823223342-172.31.6.87-33679	172.31.6.87:33679	ALIVE	2 (2 Used)	6.6 GB (1024.0 MB Used)
worker-20200823223342-172.31.7.245-46779	172.31.7.245:46779	ALIVE	2 (2 Used)	6.6 GB (1024.0 MB Used)
worker-20200823223342-172.31.9.23-35077	172.31.9.23:35077	ALIVE	2 (2 Used)	6.5 GB (1024.0 MB Used)

Figure 3: list of running workers shown in SparkUI.

URL: spark://ec2-34-255-29-19.eu-west-1.compute.amazonaws.com:7077
Alive Workers: 10
Cores in use: 20 Total, 0 Used
Memory in use: 65.2 GB Total, 0.0 B Used
Applications: 0 Running, 4 Completed
Drivers: 0 Running, 0 Completed
Status: ALIVE

Figure 4: Summary view of cluster in SparkUI.

Introduction

This phase of the document is on the form, *rationale, code, evidence*, where rationale provides a brief explanation of the approach and any problems encountered. Both ‘vanilla’ spark, i.e. RDD based computation following a map-reduce pattern and spark-sql using DataFrames were used to solve the problems, in general, the majority of the data analysis was performed via map-reduce, with spark-sql preferred for some aggregation tasks. The reason behind this decision was that It was felt that using RDDs forces one to proactively consider the locality of operations (i.e. narrow vs wide computation), whereas sometimes this logic can be abstracted when using spark-sql.

1.1

Rationale: The count was achieved using `rdd.count()`. For the yearly counts, a map function was used to perform string manipulation on the ‘Trip Start Timestamp’ to extract just the year value from the timestamp string, before a reducing by key operation was used to count the instances of each year. Note, it is not necessary at this stage to convert the Timestamp from a string.

```
def q1(rdd):
    """How many taxi records are there?
    How many taxi records for each year of the dataset?
    """
    count = rdd.count()
    yearly_counts = rdd.map(lambda x: (getattr(x, 'Trip Start Timestamp'
                                          ).split('/')[-1].split('
')[0], 1)).reduceByKey(lambda a,b: a+b)

    return count, yearly_counts
```

Figure 5: q1 function extract.

Evidence:

```
In [149]: # q1 answer
          total_records, yearly_counts = q1(taxi_rdd)
          yearly_counts.collect()

Out[149]: [('2016', 31759339),
           ('2013', 27217716),
           ('2015', 32385875),
           ('2018', 20732088),
           ('2020', 2880334),
           ('2019', 16477365),
           ('2017', 24988003),
           ('2014', 37395436)]

In [150]: # q1 answer
          total_records

Out[150]: 193836156
```

Figure 6: Q1 results

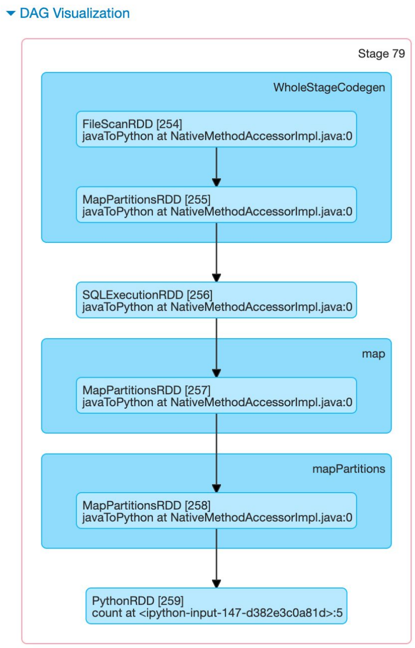

Figure 7: DAG schedule of q1 function innovation in the cluster

collect at <ipython-input-149-543127fb662e>:3	+details	2020/08/23 20:32:08	7 s	590/590			109.3 KB	
reduceByKey at <ipython-input-147-d382e3c0a81d>:7	+details	2020/08/23 19:57:53	34 min	590/590	73.6 GB			109.3 KB
count at <ipython-input-147-d382e3c0a81d>:5	+details	2020/08/23 19:28:36	29 min	590/590	73.6 GB			
csv at NativeMethodAccessorImpl.java:0	+details	2020/08/23 19:15:55	11 min	590/590	73.7 GB			

Figure 8: The completed stage for running q1 in the cluster

1.2.

This method uses a filter function on the given RDD data to filter all of the given conditions. N.b. the function makes use of two helper functions (see appendix 1), `get` and `avg_speed`, which are explained below.

Firstly `*get*` this provides a simple wrapper to the builtin python function `getattr`, in which 0 is returned by default if the attribute doesn't exist e.g. `get(x, 'y')` is equivalent to `getattr(x, 'y', 0)`. Its purpose is two-fold - one it leads to less verbose code, and two it also acts as a method for handling data with missing values, in that 0 will be returned. It is important to note that this doesn't actually update the None in the data, i.e. this isn't equivalent to a `pandas.DataFrame.fillnas` operation (https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.fillna.html), as the value is just changed in the context of the call. This was a deliberate choice as replacing None values on disk as it creates the need for a system-wide write operation across the cluster. However, it is important to note the tradeoff here, in that a new RDD/DF is created from the data on disk at a later point these None values still persist.

The `*avg_speed*` function is largely self-explanatory, but it is noteworthy that if a ZeroDivisionError occurs (e.g. before Trip Seconds = 0), 0 is returned.

N.b. for trips under 1 mile, the ave. speed = 0.0 (as miles = 0), therefore this is a decent approximation without guaranteeing total accuracy as it doesn't take into account the precise coordinates of the journey when calculating average speed.

```
def q2(rdd, total_records):
    good_trips = rdd.filter(lambda x: (get(x, 'Trip Seconds') > 60 )
                    & (get(x, 'Trip Miles') < 1000)
                    & (get(x, 'Fare') < 2000)
                    & (avg_speed(x) < 100))

    return good_trips, total_records - good_trips.count()
```

Fig 9: q2 function extract. Note, total_records is obtained from q1().

```
print(num_bad)
print(num_bad / total_records *100)
```

```
167633
8.38165
```

```
good_trips_by_year.collect()
```

```
[('2013', 250767),
 ('2014', 344979),
 ('2018', 204488),
 ('2015', 297365),
 ('2017', 243462),
 ('2016', 299756),
 ('2020', 28431),
 ('2019', 163119)]
```

Figure 10: Number of good trips per year(bottom), and the total number of bad records and overall percentage of bad records (top).

1.3

This is again achieved with RDDs alone in a four-step map function:

1. Map a new tuple of (taxi_id, fare + tips, 1), where the final bit shall later be used for a count in the reduction phase.
2. Perform a reduce by key operation, to add all of the collected fares and accumulator bits

for each key. (`*(x[0]+y[0], x[1]+y[1])*` in figure 11 below.)
3. Perform a map values operation to divide the earnings value by the number of rides.
4. Finally, perform a `take ordered` operation to sort the data descendingly and return the top 6 values.

Assumptions: Average revenue for the day excluding tolls was taken to mean Fare + Tips, as opposed to Trip Total - Tolls, the difference here is that the latter takes into account Extras.
In hindsight, this implementation returns the fares per trip, not the fares per day as the accumulator simply adds an additional count for each trip.

```
def q3(rdd):
    """For each taxi, calculate the average revenue per day excluding tolls
(i.e. Fare + Tips).
    Identify the most successful taxi in 2018 in terms of total revenue
(Fare + Tips).


https://stackoverflow.com/questions/29930110/calculating-the-averages-for-e
ach-key-in-a-pairwise-k-v-rdd-in-spark-with-pyth

    """
    return rdd.map(lambda x: (get(x, 'Taxi ID'),
                              [get(x, 'Fare') + get(x, 'Tips'), 1])
                      ).reduceByKey(lambda x,y:
                                    (x[0]+y[0], x[1]+y[1])
                                   ).mapValues(
                       lambda v: v[0]/v[1]).takeOrdered(6,
                                            key=lambda x: -x[1])
```

Figure 11: q3 implementation.

Regarding the limitation discussion above the following implementation should handle per day with minimal changes to the logic, however, note this was not tested on the cluster.

```
def q3_per_day(rdd):
    """For each taxi, calculate the average revenue per day excluding tolls
(i.e. Fare + Tips).
    Identify the most successful taxi in 2018 in terms of total revenue

(Fare + Tips).

```

```
    """
    return rdd.map(lambda x: (get(x, 'Taxi ID'),
                              [get(x, 'Fare') + get(x, 'Tips'), get(x, Trip
Start Timestamp)])
                        ).reduceByKey(lambda x,y:
                                    (x[0]+y[0], x[1]+y[1])
                                   ).mapValues(
                        lambda v: v[0]/
aggregate_timestamps(v[1])).takeOrdered(6,
                                          key=lambda x: -x[1])
```

Figure 12: alternative implementation for q3, the difference here is the accumulator bit is replaced with the row timestamp and in the mapValues call, these timestamps are then aggregated to return the count of distinct days. This assumes the function `aggregate_timestamps` which returns an integer from a row RDD of timestamped data.

Figure 13 (above): Event timeline for running q3() in the 10 worker cluster. N.b. the majority of each stage is spent on executor computation time, indicating leveraging narrow computations.

```
In [23]: sorted_fares

Out[23]: [('574b161e75a7dda9dc1f63751af8b83c982a3a76f5fc889d41c802a4c4ffd1c81c9752bb64b9973d27e92d27e609cec80339ee5d4868a77784
49c4cc54bbee52',
  65.81),
 ('bd8e56507781dbc93fbb8a08e8d441b8d64f78a52affec71f14ad3aad6e0d044352be167df78a16637bcc7d4a29e58905047af5434b269c8b2
509dd297477ad7',
  65.39999999999999),
 ('2ab652af79401a1e9f3a16bfa63845ec3d1266d0515d9dc38f9387271a7114514a992f7f8634c7ec3712586667451c92b2d65cb93870cc894a
faced4792c79b2',
  64.24555555555554),
 ('64848e03709152785ab1f99953fe601c44acad3e49fc5a515fa5b448b2d1206bdcce453c32fad109c77c45194b793a37c085116a11014ef3f0
6eeb37647977c4',
  64.0),
 ('26b43fecf9e9479444973797e89a74f559183f1cc1abf04cf05c829a33c229a8c1a5f797e493a02b589056ab0c0c1483255b2e059a463eef4d
cd39f852c5d5de',
  61.92),
 ('4d1dbd80c3b4c74b6441906c81e74c1150c532a88fd32572dfa9b23506ad6c388f45c453b7ddc2506042ec16187a698cd3c5f160a968286633
c7bfebc8f2553f',
  60.375)]
```

Figure 14: sorted fares output from Jupyter notebooks, note the top-performing taxi had an average of $65.81.

It should be noted that there was no qualification boundaries implemented, e.g. setting barriers to entry such as a minimum of 20 rides.

1.4

This was the first solution which was broken into two functions - one for preparing the data (`prepare_q4`), and a second for computing the results (`q4`).

This question involved a three-stage map function, with the following key actions:

1. Find the create a tuple of start and end-times (S, E)
2. Convert each of these into DateTime instances
3. Find the midpoint, that is the start, + (the time delta of the end - start),

and then set the hour in scope to this hour.

It is also noteworthy that the result of the lambda function is to construct a `namedtuple` (https://docs.python.org/3/library/collections.html) instance named `Prepared`. Named tuples act like standard tuples with the exception of the fact that the elements are named. This provides convenience as it means we can a) subset the data to only the columns we're interested in, and b) we don't have to rely on accessing the members of the output by index. E.g:

x = (1, 2)
x[1]

Is equivalent to:

X = namedtupe('X', ['foo', 'bar'])
X.bar

This makes subsequent operations more convenient and robust as we don't have to rely on accessing collection items by index.

Various permutations of this implementation were theorised, such as doing as:

1. A *pure sparksql implementation as below:*

speedy = hourly_avgs.agg(max('avg_speed'))
tips = hourly_avgs.agg(max('tips'))
fares = hourly_avgs.agg(max('fare'))
return hourly_avgs, tips, fares

However, this leads to greater complexity as each .agg call returns just the datapoint, not the row i.e. critically the midpoint is lost. For this reason, this approach was rejected.

2. A simple sort for each feature and then take the first item of each rdd.

This was rejected as it is impossible to avoid three sort operations which are not only expense operations in general but require wide dependencies reduction, so even more expensive here.

3. A transformation-based approach (chosen approach):

Transform the row data into k, v tuples for the relevant features and then find the max of each of these e.g. transform a given row rdd of format: *[m, s, f, t] to: [(m,s), (m,f), (m,t)].* where m=midpoint, f=fare, s=speed, and t=tips.

This enables calling performing a filter on each position to gain the maximum averages for each parameter. Whilst this detail wasn't implemented, optimisation would be to re-partition the data following the transform so that each (m,s), (m,f), and (m,t) pair were localised to make use of narrow computation in the reduction phase when finding the max value from each.

```
def midpoint(x):
        """Midpoint:  lambda x: (x[0] + (x[1] - x[0]) /2).hour) , where x =
(start, end) """
        start = string_to_time(get(x, 'Trip Start Timestamp'))
        end = string_to_time(get(x, 'Trip End Timestamp'))
        return start, end, (start + (start - end) /2).hour
```

```
    Prepared = namedtuple('Prepared', ['fare', 'tips', 'avg_speed',
'start', 'end', 'midpoint', 'miles'])
    return rdd.map(lambda x: Prepared(get(x, 'Fare'), get(x, 'Tips'),
avg_speed(x),
                                       *midpoint(x), get(x, 'Trip Miles')))
```

Figure 15: Midpoint function used to extract midpoint from raw string timestamp data, see appendix 1 for the `string_to_time` function implementation.

```
def q4(df):
    hourly_avgs = df.groupBy('midpoint').agg(avg('avg_speed'), avg('fare'),
avg('tips'))
    rdd = hourly_avgs.rdd


    Results = namedtuple('Results', ['midpoint', 'fare', 'tips',
'avg_speed'])
    results = rdd.map(lambda x: Results(x.midpoint, x[1], x[2], x[3]))

#   N.b. `or 0` handles comparison of NoneTypes as part of the max function
    max_fare = results.max(key=lambda x: x.fare or 0)
    max_tips = results.max(key=lambda x: x.tips or 0)
    max_avg_speed = results.max(key=lambda x: x.avg_speed or 0 )

    return max_fare, max_tips, max_avg_speed
```

Figure 16: q4() implementation

```
# answer q4
prepared_rdd = prepare_q4(rides_2018)
# prepared_rdd.first()

# prepared_rdd.take(5)
prepared_df = prepared_rdd.toDF()
max_fare, max_tips, max_avg_speed = q4(prepared_df)
```

```
print(f'max_fare: {max_fare}')
print(f'max_tips: {max_tips}')
print(f'max_avg_speed: {max_avg_speed}')
```

```
max_fare: Results(midpoint=5, fare=23.462893054843466, tips=23.221218860433687, avg_speed=6.305159716758459)
max_tips: Results(midpoint=5, fare=23.46289305484345, tips=23.221218860433687, avg_speed=6.3051597167584585)
max_avg_speed: Results(midpoint=5, fare=23.462893054843466, tips=23.221218860433687, avg_speed=6.305159716758455)
```

```
prepared_rdd.count()
```

```
19791796
```

Figure 17: Output from running Q4 in the cluster, not the however the max average speed of 6.3 mph for the day suggests that there may have been an bug in the logic, with more time this would be investigated further.

Figure 18: execution map for running q4() on the 10 node cluster

```
INFO Executor: Finished task 462.0 in stage 52.0 (TID 3246). 2964 bytes result sent to driver
INFO CoarseGrainedExecutorBackend: Got assigned task 3299
INFO Executor: Running task 515.0 in stage 52.0 (TID 3299)
INFO FileScanRDD: Reading File path: s3a://chictaxi/chictaxi.csv, range: 69122129920-69256347648, partition values: [empty row]
INFO PythonRunner: Times: total = 18040, boot = 7, init = 121, finish = 17912
INFO Executor: Finished task 510.0 in stage 52.0 (TID 3294). 2835 bytes result sent to driver
INFO CoarseGrainedExecutorBackend: Got assigned task 3306
INFO Executor: Running task 522.0 in stage 52.0 (TID 3306)
INFO FileScanRDD: Reading File path: s3a://chictaxi/chictaxi.csv, range: 70061654016-70195871744, partition values: [empty row]
INFO PythonRunner: Times: total = 20510, boot = -39, init = 171, finish = 20378
INFO Executor: Finished task 515.0 in stage 52.0 (TID 3299). 2835 bytes result sent to driver
INFO CoarseGrainedExecutorBackend: Got assigned task 3312
INFO Executor: Running task 528.0 in stage 52.0 (TID 3312)
INFO FileScanRDD: Reading File path: s3a://chictaxi/chictaxi.csv, range: 70866960384-71001178112, partition values: [empty row]
INFO PythonRunner: Times: total = 25285, boot = -7, init = 141, finish = 25151
INFO Executor: Finished task 522.0 in stage 52.0 (TID 3306). 2964 bytes result sent to driver
INFO CoarseGrainedExecutorBackend: Got assigned task 3322
INFO Executor: Running task 538.0 in stage 52.0 (TID 3322)
INFO FileScanRDD: Reading File path: s3a://chictaxi/chictaxi.csv, range: 72209137664-72343355392, partition values: [empty row]
INFO PythonRunner: Times: total = 18577, boot = 4, init = 117, finish = 18456
INFO Executor: Finished task 528.0 in stage 52.0 (TID 3312). 2835 bytes result sent to driver
INFO CoarseGrainedExecutorBackend: Got assigned task 3328
INFO Executor: Running task 544.0 in stage 52.0 (TID 3328)
INFO FileScanRDD: Reading File path: s3a://chictaxi/chictaxi.csv, range: 73014444032-73148661760, partition values: [empty row]
INFO PythonRunner: Times: total = 17656, boot = -57, init = 196, finish = 17517
INFO Executor: Finished task 538.0 in stage 52.0 (TID 3322). 2835 bytes result sent to driver
INFO CoarseGrainedExecutorBackend: Got assigned task 3341
INFO Executor: Running task 557.0 in stage 52.0 (TID 3341)
INFO FileScanRDD: Reading File path: s3a://chictaxi/chictaxi.csv, range: 74759274496-74893492224, partition values: [empty row]
INFO PythonRunner: Times: total = 17122, boot = 5, init = 93, finish = 17024
INFO Executor: Finished task 557.0 in stage 52.0 (TID 3341). 2835 bytes result sent to driver
INFO CoarseGrainedExecutorBackend: Got assigned task 3353
INFO Executor: Running task 569.0 in stage 52.0 (TID 3353)
INFO FileScanRDD: Reading File path: s3a://chictaxi/chictaxi.csv, range: 76369887232-76504104960, partition values: [empty row]
```

Figure 19: spark logs in cluster

1.5

The first point worth mentioning is that the input to q5 is the output from q4. It would be possible to run the functions independently, however as the output of q4 is already a well-formed named tuple with common features to q5, it felt sensible to re-use this data. It should be noted there is a trade-off here, by reusing the output of q4 we're essentially caching - that is that this method is not well suited to real time analytics as any new data points added to the raw set since the invocation of q4 would be ignored. That this, this clearly isn't the use case here, so for our purposes reuse is both appropriate and more efficient.

The logic for q5 is fairly simple, a named tuple construct is again used as an output format for the mapped RDD data. The RDD is first mapped to calculate the overall tip percentage and the tips per mile ratio. Having calculated these values, the RDD is then sorted by the highest tips per mile to yield the highest tripped rides by distance. Finally, cast to a DF and grouped by month to enable plotting of tips by month.

Code

```
def prepare_q5(prepared_rdd):
    """ What is the overall percentage of tips that drivers get?
    Find the top ten trips with the best tip per distance travelled.

    Create a graph of average tip percentage by month for the whole period.

    """
    def tips_percentage(row):
        try:
            return (row.tips / row.fare) * 100
        except ZeroDivisionError:
            return 0


    def tip_per_mile(row):
        try:
            return row.tips / row.miles
        # In the case of a trip of 0 miles, just use the tip amount
        except ZeroDivisionError:
            return row.tips


    Q5Results = namedtuple('Q5Results', ['start', 'month', 'fare', 'tips',
```

```
'tip_per_mile', 'tip_percentage_of_fare'])
    return prepared_rdd.map(lambda x: Q5Results(x.start,
calendar.month_name[x.start.month], x.fare, x.tips,

tip_per_mile(x), tips_percentage(x))
                                                ).sortBy(lambda x:
-x.tip_per_mile)



def get_overall_tips_percentage(prepared_rdd):
        try:
            return prepared_rdd.map(lambda x: x.tips.sum() /
prepared_rdd.map(lambda x: x.fare).sum() * 100)
        except ZeroDivisionError:
            return 0


def get_tips_percentage_per_month(prepared_rdd):
       return prepared_rdd.sortBy(lambda x: x.start)
```

Figure 20: preparing Q5 data (above)

```
# Q5 answer
prepared_q5_rdd = prepare_q5(prepared_rdd)
generous_tippers = prepared_q5_rdd.take(10)
tippers_by_month = prepared_q5_rdd.sortBy(lambda x: x.start).take(10)

df =  prepared_q5_rdd.toDF().toPandas()
avg = df.groupby('month').mean()

# Q5 Plot
figure, axes = plt.subplots(1,1)
avg_plt = axes.bar(avg.index, avg['tip_percentage_of_fare'])

axes.set_title('Tip Percentage of total fare per month', fontsize=20)
axes.set_xlabel('Month')
axes.set_ylabel('Tip Percentage')
plt.grid()
plt.show()
```

Figure 21: Running q5, including graphs. Note, matplotlib is used as the graphing library.

```
In [45]: generous_tippers.sort(key=lambda x: -x.tips)
         generous_tippers

Out[45]: [Q5Results(start=datetime.datetime(2018, 7, 29, 1, 30), month='July', fare=4.0, tips=150.0, tip_per_mile=1500.0, tip_
         percentage_of_fare=3750.0),
          Q5Results(start=datetime.datetime(2018, 2, 16, 18, 0), month='February', fare=3.5, tips=75.0, tip_per_mile=1875.0, t
         ip_percentage_of_fare=2142.8571428571427),
          Q5Results(start=datetime.datetime(2018, 1, 26, 11, 30), month='January', fare=4.5, tips=25.35, tip_per_mile=1267.5,
         tip_percentage_of_fare=563.3333333333334),
          Q5Results(start=datetime.datetime(2018, 8, 14, 5, 45), month='August', fare=3.25, tips=24.0, tip_per_mile=2400.0, ti
         p_percentage_of_fare=738.4615384615385),
          Q5Results(start=datetime.datetime(2018, 2, 14, 20, 0), month='February', fare=3.5, tips=20.0, tip_per_mile=2000.0, t
         ip_percentage_of_fare=571.4285714285714),
          Q5Results(start=datetime.datetime(2018, 10, 11, 15, 45), month='October', fare=4.25, tips=15.75, tip_per_mile=1575.
         0, tip_percentage_of_fare=370.5882352941177),
          Q5Results(start=datetime.datetime(2018, 1, 26, 11, 30), month='January', fare=3.75, tips=14.78, tip_per_mile=1478.0,
         tip_percentage_of_fare=394.1333333333333),
          Q5Results(start=datetime.datetime(2018, 1, 24, 10, 0), month='January', fare=4.0, tips=12.88, tip_per_mile=1288.0, t
         ip_percentage_of_fare=322.0),
          Q5Results(start=datetime.datetime(2018, 9, 10, 17, 45), month='September', fare=3.25, tips=12.15, tip_per_mile=1215.
         0, tip_percentage_of_fare=373.84615384615387),
          Q5Results(start=datetime.datetime(2018, 2, 10, 15, 0), month='February', fare=5.25, tips=12.05, tip_per_mile=1205.0,
         tip_percentage_of_fare=229.52380952380955)]
```

Figure 22: The highest tips of 2018 - one lucky driver received a $150 tip.

```
avg.show()
```

month	avg(fare)	avg(tips)	avg(tip_per_mile)
July	14.24935282157495	3.8741833947264466	1.0196895752901305
November	14.398324532913524	3.901858131037569	1.1744691525802238
February	13.124509576132832	3.610912494084176	1.0017781538991144
January	12.827626281768856	3.56878032280292	1.025528435098136
March	13.343275860302803	3.639173683027003	0.963288158301273
October	14.918783149173684	4.010853796905106	1.1458443393229907
May	14.583230743171315	3.98551809935468	1.0211047356793157
August	14.362112295666265	3.8751314236885483	1.0554075943867258
April	13.984622979688657	3.847713132269758	0.9945019613526083
June	14.603612335664936	3.9631562175560213	1.0872622351860073
December	12.90149756067258	3.566174164706671	1.1162678686976772
September	14.775316380121243	4.00243631904812	1.0836002614721254

Figure 23: Average tips per month for 2018.

The above code was run via jupyter notebook in a single node instance with a subset of the data without issue, however when running on the full 2018 dataset in the cluster I encountered a runtime error which unfortunately could not be diagnosed in the timeframe. See Appendix E for the stack track from the cluster.

1.6

The approach here was again quite simple, filter out the trips which are not the correct time window, before clustering them by latitude and longitude, finally collect the most desirable locations from the size of the clusters.

Kmeans was used as the clustering algorithm to group the points. Often Haversine distance is used for tasks involving lat/long coordinates, however, as this task wasn't looking to plot the distance between any points, k-means was an appropriate choice.

The 'pyspark.ml.clustering' module contains an inbuilt Kmeans algorithm, and is well documented - this documentation was followed in constructing the results.

Code:

```
def prepare_geo_data(rdd):
    Q6Results = namedtuple('Q6Results', ['start_lat', 'start_long',
                                         'end_lat', 'end_long',
                                         'start_timestamp',
'end_timestamp'])

    filtered = prepared_geo_data.filter(lambda x: ((x.start_timestamp.hour
>= 17) & (x.end_timestamp.hour <= 19)))
    return filtered.map(lambda x: Q6Results(get(x, 'Pickup Centroid
Latitude'),
                   get(x, 'Pickup Centroid Longitude'),
                   get(x, 'Dropoff Centroid Latitude'),
                   get(x, 'Dropoff Centroid Longitude'),
                   string_to_time(get(x, 'Trip Start Timestamp')),
                   string_to_time(get(x, 'Trip End Timestamp'))))

# https://spark.apache.org/docs/latest/ml-clustering.html
https://spark.apache.org/docs/2.2.0/api/python/pyspark.ml.html#pyspark.ml.c
lustering.KMeans
https://spark.apache.org/docs/2.2.0/api/python/pyspark.ml.html#pyspark.ml.f
eature.VectorAssembler
def q6(df):
    import pandas as pd
    from pyspark.ml.clustering import KMeans
    from pyspark.ml.feature import VectorAssembler
```

```python
    from pyspark.ml.evaluation import ClusteringEvaluator

    vectors = VectorAssembler(inputCols=['start_lat', 'start_long'],
                              outputCol='features', handleInvalid='skip')

    df_ = df.where((col("foo") > 0) & (col("bar") < 0))
    df_ = vectors.transform(df_)

    kmeans = KMeans(k=308, seed=1)
    model = kmeans.fit(df_.select('features'))
    predictions = model.transform(df_)
    centers = model.clusterCenters()

    predictions.centers = pd.Series(centers)

    return predictions, centers
```

Figure 24: implementing Kmeans via pyspark.ml. Note that K is initialised to 308 this was due to the fact that there are 77 district in Chicago and so this allows for four clusters per distinct, aiming to providing a greater granularly of the result. This method is clearly an approximation for an appropriate K initialization, however, performing any automated K initialisation was out of scope.

For the plotting, the data, the 'geopandas' library was used to plot the lat/long data against a downloaded map of Chicago. A combination of guides were followed from (https://www.kaggle.com/threadid/geopandas-mapping-chicago-crimes/notebook?select=geo_export_33ca7ae0-c469-46ed-84da-cc7587ccbfe6.shx) and the geopandas documentation. However unfortunately I was unable to get the graph to render in the cluster the lat/long points overlaid on the Chicago map.

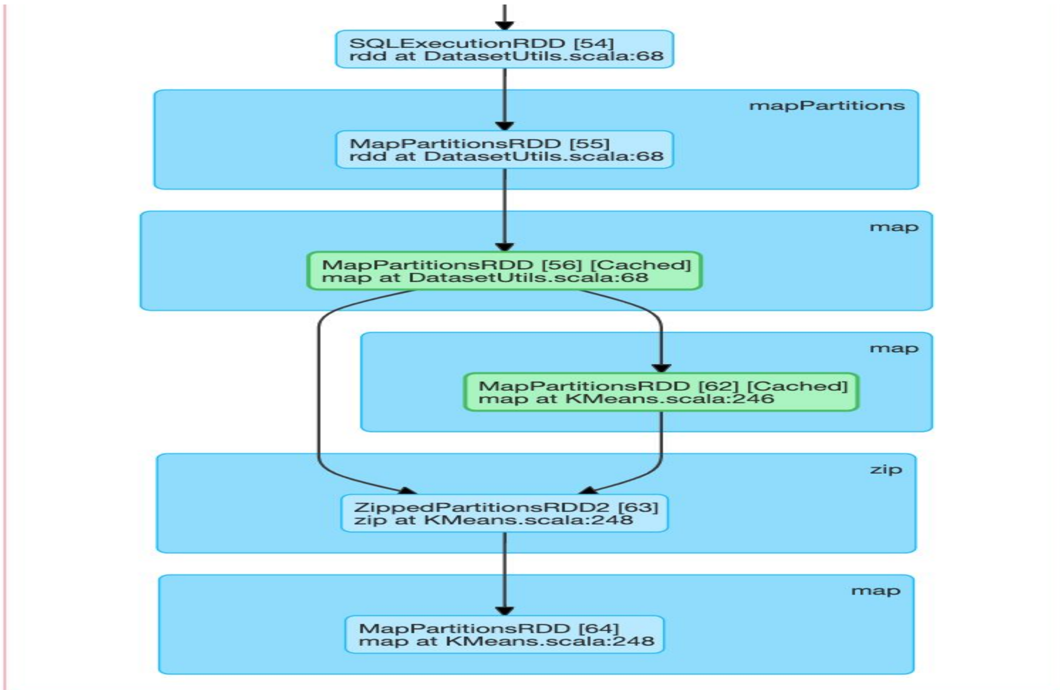

FIgure 25: DAG visualisation of running Kmeans on the cluster

1.7

The approach once again was simple, find all pairs of start/end postcodes, make them agnostic of start end, group the pairs and then return the largest sets.

For example given the data *[(A, B), (A, C), (B, A)]*, where the first index of each tuple represents the start and the second the end postcodes, the pairs (A, B), (B, A) would be genericised to (A,B), (A, B) and then the group by operation would take place, i.e. group by start and then by end. Resulting in the following groups and counts: (A,B) = 2, (A, C) =1.

The decision to make the task agnostic of direction was taken as if we consider a start, stop pairing as a single datapoint then when evaluating which data points are most common, the direction is irrelevant. One could reasonably argue that (A, B) and (B, A) are different journeys and should, therefore, be considered separately, however it was felt in this implementation that commonality of occurrence was the desired outcome.

The `uszipcode` was used to analyse the zipcodes. N.b. a coordinate can map to >1 postcode, (by default uszipcode returns 5 addresses per lat/long search), however, sometimes these results return duplicates. In order to dedupe the results the following method was used:

0. construct a list of tuples of zipcodes pairs
   0.a query the start_lat, start_long and end_lat, end_long coordinates
   0.b assemble (start, end) pairs by zipping the results back (using the python built-in function zip).

1. Cast zipcodes to integers. At this point the data resembles:

    [(60640, 60660),
    (60660, 60640),
    (60613, 60626),
    (60657, 60659),
    (60625, 60645)]

n.b. here indexes 0,1 are essentially duplicated with the start/end positions switched.

2. sort each zipcode tuple
3. cast the output to a set to remove duplicate entries:

    {(60613, 60626), (60625, 60645), (60640, 60660), (60657, 60659)}

```
from uszipcode import SearchEngine as ZipCodeEngine

zip_seacher = ZipCodeEngine(simple_zipcode=True)

def q7(row):
        starts = [res.zipcode for res in
zip_seacher.by_coordinates(get(row, 'start_lat'), get(row, 'start_long'))]
        ends = [res.zipcode for res in zip_seacher.by_coordinates(get(row,
'end_lat'), get(row, 'end_long'))]

        res = list(set([tuple(sorted([int(s), int(e)])) for s,e in
zip(starts, ends)]))
        return res
```

Figure 23: q7() function implementation.

1.8

This answer used a similar technique to question six, in first assembling a RDD, casting that to a DF and groupby the output by timestamp (to ensure that the various weather station events are aligned) and then getting the mean rainfall from these.

Code:

```
def prepare_weather_data(rdd):
    Q8Data = namedtuple('Q8Data', ['meaurement_id','rain_interval',
'intensity','station',  'timestamp', ])
    return rdd.map(lambda x: Q8Data( get(x, 'Measurement ID'), get(x,
'Interval Rain'),
                    get(x, 'Rain Intensity'),
                    get(x, 'Station Name').replace(' ', ''),
                    string_to_time(get(x, 'Measurement Timestamp')
                                  )))
                    # gladly TS format matches so this helper can be
reused...

def q8(q8_data):
    """Uses a similar approach to question 6, group the data then recast to
RDD and the max values. """

    grouped = q8_data.toDF().groupby('timestamp').mean()
    rdd = grouped.rdd


    Results = namedtuple('Results', ['timestamp', 'avg_interval',
'avg_intensity'])
    results = rdd.map(lambda x: Results(x.timestamp, x[1], x[2]))

    #   N.b. `or 0` handles comparison of NoneTypes as part of the max
function
    max_interval = results.max(key=lambda x: x.avg_interval or 0)
    max_intensity = results.max(key=lambda x: x.avg_intensity or 0)

    return max_interval, max_intensity, rdd
```

Figure 24: q8() function implementation.

```
In [57]: max_interval
Out[57]: Results(timestamp=datetime.datetime(2018, 7, 5, 15, 0), avg_interval=50.4, avg_intensity=None)
```

```
max_intensity
Results(timestamp=datetime.datetime(2018, 7, 20, 11, 0), avg_interval=3.7, avg_intensity=119.4)
```

Figure 25: max intensity and max interval values for 2018

1.9

The method here was to use join the weather data and taxi data so that the values could be correlated. This was achieved using the pyspark dataframe join method and joining on the rows timestamp of both datapoints. (https://spark.apache.org/docs/latest/api/python/pyspark.sql.html?highlight=join).

One of the main complexity in this problem was properly formatting the timestamp data to enable Pearson correlation. Pyspark facilitates Pearson correlation by simply calling `df.stat.corr(x, y)` on a given dataframe (https://people.eecs.berkeley.edu/~jegonzal/pyspark/pyspark.sql.html#pyspark.sql.DataFrame.corr), however, this function only accepts numeric values to be correlated. In order to handle this, the timestamp data was converted to UNIX timestamp integers, using the python function `mktime` form the built-in time module. In order to do this, the weather data timestamps had to first be converted to isoformat time strings.

```
def prepare_q9(rain_data_df, taxi_data_df):
    """N.b. this function isn't generic - requires DF in correct format to
be able to join.
    rain_data_df - e.g. as output from `q8`
    taxi_data_df - e.g. as output from `q4`

    """
    def to_unix_timestamp(isoformat):
        """Correlation not supported on timestamp data, so need to convert
timestamps to ints.
        """
        import time
        return int(time.mktime(time.strptime(isoformat,
'%Y-%m-%dT%H:%M:%S')))
```

```
    # 
https://spark.apache.org/docs/latest/api/python/pyspark.sql.html?highlight=
join
    joined = rain_data_df.join(taxi_data_df, taxi_data_df.start == 
rain_data_df.timestamp)

    Q9Results = namedtuple('Q9Results', ['timestamp', 'avg_interval', 
'avg_intensity', 'fare', 'tips'])
    rdd = joined.rdd.map(lambda x: Q9Results(
        to_unix_timestamp(x.timestamp.isoformat()),
        get(x, 'avg(rain_interval)'), get(x, 'avg(intensity)'),
        x.fare, x.tips))
    return rdd

def q9(df):
    # 
https://people.eecs.berkeley.edu/~jegonzal/pyspark/pyspark.sql.html#pyspark
.sql.DataFrame.corr
    fare_corr = df.stat.corr("timestamp", "fare")
    tip_corr = df.stat.corr("timestamp", "tips")
    return fare_corr, tip_corr
```

Figure 25: q9() implementation.

```
: q9_data = prepare_q9(rain_rdd.toDF(), prepared_rdd.toDF())
  fare_corr, tip_corr = q9(q9_data.toDF())

: fare_corr

: 0.024260759999924934

: tip_corr

: 0.010381871867977703
```

Figure 26: output from running q9() - note the small correlations which seem somewhat counterintuitive. In a real world scenario, more time would be spent validating these outputs.

Stage Id	Description	
280	corr at NativeMethodAccessorImpl.java:0	+details
279	corr at NativeMethodAccessorImpl.java:0	+details
275	corr at NativeMethodAccessorImpl.java:0	+details
274	corr at NativeMethodAccessorImpl.java:0	+details
270	runJob at PythonRDD.scala:153	+details
266	runJob at PythonRDD.scala:153	+details
265	javaToPython at NativeMethodAccessorImpl.java:0	+details
263	javaToPython at NativeMethodAccessorImpl.java:0	+details
262	runJob at PythonRDD.scala:153	+details
261	runJob at PythonRDD.scala:153	+details
260	runJob at PythonRDD.scala:153	+details

Figure 27: stage breakdown of running correlations.

Part 2

For this section the Jupter notebook created in part one was exported to a normal python file, and adapted to meet the requirements of the task. (See appendix F). A ‘make’ command was created to iteratively send a spark submit job to the cluster utilising between 1-8 of cores.The `make flintrock_launch’ command was reused to setup a fresh four node cluster, to ensure that the runtime were not affected by any ongoing processing on the existing cluster. Running the make command I was able to send the task to the cluster via spark submit, however the executors completed almost instantly and with no meaningful logging in the system, meaning that unfortunately the Karp-Flatt metric evaluation of the system could not be achieved.

```
CORES = 1 2 4 8
run_q2:
   # $(foreach var,$(CORES), echo $(var);)
   # Set pyspark executor to the system python, not jupyter
   $(foreach core,$(CORES), flintrock run-command ${CLUSTER} 'workon clo && export
PYSPARK_DRIVER_PYTHON=`which python` \
   && export PYSPARK_DRIVER_PYTHON_OPTS=`which python` \
   && spark-submit --master
spark://ec2-34-244-39-108.eu-west-1.compute.amazonaws.com:7077 part_2.py \
   --packages org.apache.hadoop:hadoop-aws:2.7.4 part_2.py --executor-cores $(core)';)
```

FIgure 28: Make file command to iteratively send spark submit jobs to the cluster with an increasing number of executor-cores. The script iterates the number of cores (in the ‘CORES’ variable), and foreach call sources the python env on the cluster, exports the pyspark driver variables (so that the cluster doesn’t attempt to run the spark submit job in a notebook), and

finally sends the spark submit job to the master node.

```
(clo) ➜  CLO git:(master) ✗ make run_q2
#  echo 1;  echo 2;  echo 4;  echo 8;
# Set pyspark executor to the system python, not jupyter
flintrock run-command clo-spark-cluster-lt-10-node 'workon clo && export PYSPARK_DRIVER_PYT
r spark://ec2-34-244-39-108.eu-west-1.compute.amazonaws.com:7077 part_2.py --packages org.a
cluster-lt-10-node 'workon clo && export PYSPARK_DRIVER_PYTHON=`which python` && export PYS
est-1.compute.amazonaws.com:7077 part_2.py --packages org.apache.hadoop:hadoop-aws:2.7.4 pa
& export PYSPARK_DRIVER_PYTHON=`which python` && export PYSPARK_DRIVER_PYTHON_OPTS=`which p
 part_2.py --packages org.apache.hadoop:hadoop-aws:2.7.4 part_2.py --executor-cores 4';  fl
which python` && export PYSPARK_DRIVER_PYTHON_OPTS=`which python` && spark-submit --master 
.hadoop:hadoop-aws:2.7.4 part_2.py --executor-cores 8';
Running command on cluster...
[3.250.137.203] Running command...
[54.216.23.211] Running command...
[18.203.99.6] Running command...
[63.32.71.111] Running command...
[34.255.29.19] Running command...
[34.240.48.54] Running command...
[54.155.226.21] Running command...
[34.244.120.203] Running command...
```

Figure 29: Shell output resulting in issue the above `make run_q2` command. Note the increasing executor cores with each call.

Appendix F demonstrates that the jobs were picked up in the cluster, but unfortunately I was unable to diagnose why the jobs either didn’t run fully or error.

Part 3

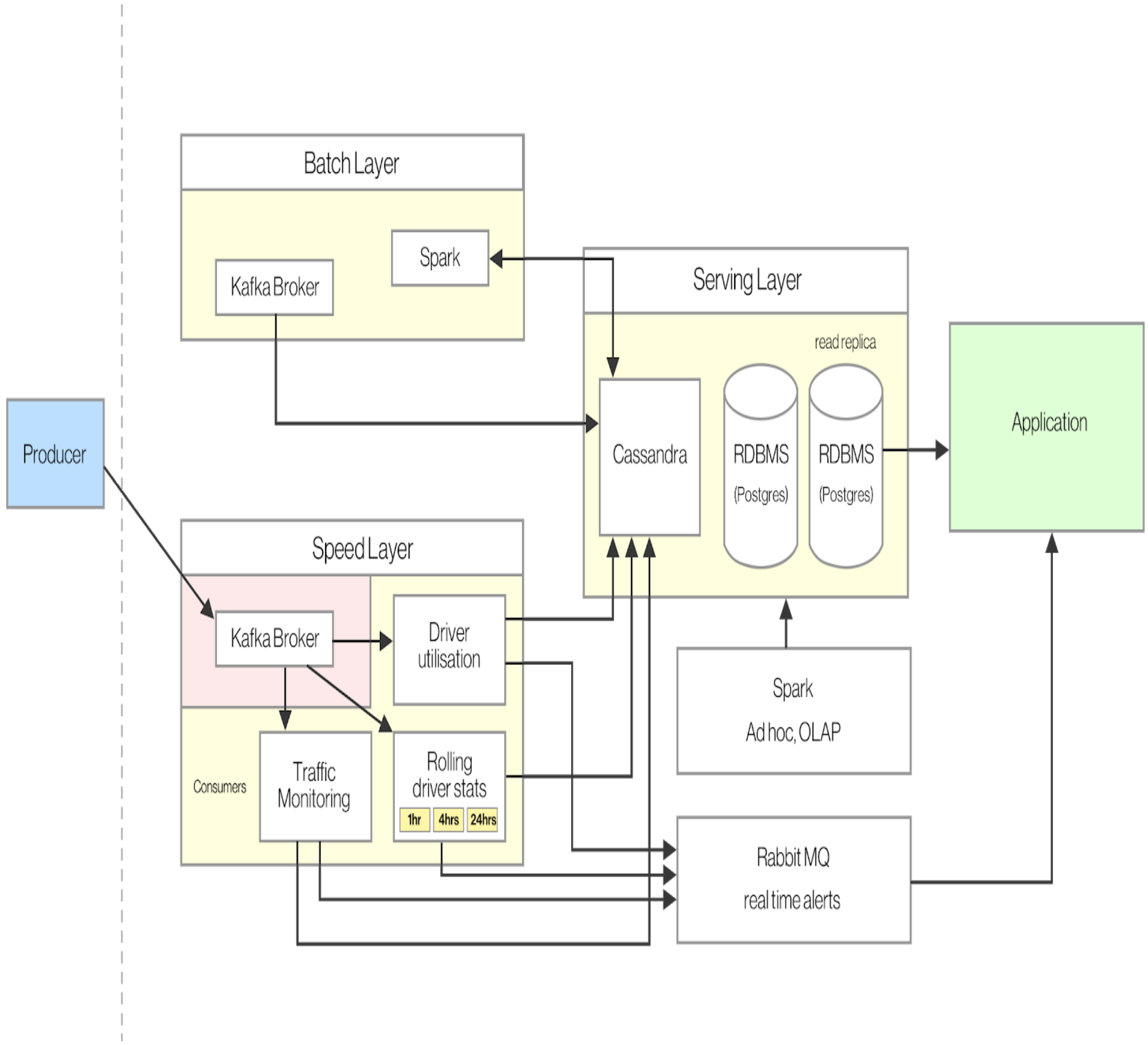

Figure 26: architecture diagram

- **How the data is ingested into the system and stored?**

The data is ingested via Kafka using a mini batch architecture. Mini bath felt appropriate as the velocity of the data is unlikely to be truly real time. On the speed side of the architecture there is one Kafka broker which connects to multiple consumers. No business logic is implemented in the broker, rather, simple data preprocessing enables feeding data to the consumers. This leads to a flexible architecture wherein the consumers are loosely coupled with the broker and further brokers and consumers can be added/removed with no effect on the existing consumers.

The consumers will output data to a Cassandra database in the serving layer. The batch side is also connected to a Cassandra cluster to store output of regularly scheduled batch processes which are run periodically throughout the day and therefore produce a large volume of output suitable for an append only event store db. Additionally in the serving layer, a relational database instance eg. postgres is used as a source of aggregate data and system metadata. All of this serving layer can be connected to by the data scientist to perform ad hoc queries via Spark submit tasks.

Which big data framework or algorithms are you going to choose? Why?

For both batch processing and ad hoc querying from the data scientist Spark will be used. This is because it is widely adopted, well maintained, highly flexible, plays well with Cassandra and supports embarrassingly parallel computation across a cluster. In general, the architecture can be categorised as a Lambda architecture as there is a clear separation of batch and speed processing by design. For the traffic analysis and driver utilisation tasks, clustering algorithms would be used as a means of anomaly detection. This is discussed in greater detail below.

- **How does your chosen approach efficiently process the data?**

By utilising Spark we can ensure that data processes are optimised for narrow computations across the cluster via the map reduce pattern. The architecture is inherently parallel, with multiple consumers in the speed layer, each with a clear separation of concerns providing loose coupling between the speed tasks eg. if any one consumer fails the failing is isolated to that consumer meaning that not all of the disparate tasks have to be repeated. In this sense the consumers can be seen to be implementing a lazy evaluation model, in line with Spark in general and the notion of directed acyclic graphs. The same principles can be applied to the batch side. Separate Spark jobs for each reporting function would be implemented.

- **Did you utilise a well-known pattern such as the lambda or kappa**

architectures? Justify your approach.

As previously stated, this is a Lambda architecture. Kappa was considered, however, the use case was a natural choice for Lambda. Given that the use case includes batch, real time and ad hoc querying. It felt prudent to choose an architecture that mirrored these requirements. In the same way that architecture is said to mirror team structure in

Conway's law. If architecture can closely match the business use case, this will surely lead to better adoption and maintainability of a project. That is not to say that this set of requirements could not be fulfilled with a Kappa architecture however, the implementation would be inherently more complex due to the requirement of long running batch processes, resulting in a greater cost of implementation.

- **Which cloud infrastructure and why?**

AWS would be chosen as the cloud infrastructure, the primary reason for this, is that Spark is a key component of the architecture and of the available mainstream providers, AWS has the lowest barrier to entry for Spark implementations. This is evidenced by the integration with Databricks, providing a managed Spark service, AWS' own EMR service or the ability to roll your own.

- **How are you going to scale this?**

For the exercises Flintrock was used to provide a script for building and destroying clusters automatically with no need for customisation on any of the nodes. Whilst Flintrock advise against using the tool in production, in general this would be the approach, ie. to utilise build scripts to automate as much as possible of the spinning up of infrastructure. This has the added benefit of implementing *gitops* as standard (i.e. all actions to create the cluster are recorded and version controlled). In terms of scaling of services, all of the components of the architecture are easily horizontally scalable, perhaps with the exception of the RDBMS which has a lower requirement for this as it will be the slowest growing component of the data storage architecture.

Elastic scaling can be configured on the Kafka brokers and consumers to ensure even traffic distribution. Note that the Cassandra cluster also supports horizontal scaling, however it's not a good candidate for auto scaling in AWS due to its own inbuilt clustering mechanism for node discovery. An ancillary batch monitoring job could monitor the capacity of the Cassandra nodes to ensure the cluster wasn't nearing capacity before we had a chance to scale it.

- **Which language(s) are you going to use to process the data and why?**

The language choices would be somewhat open depending on the team we hired. Spark is a superbly flexible framework in that it is offered fully featured in java, python and scala. Therefore, if I were managing this team I would be somewhat led by the pool of developers available to me. Python is perhaps preferable due to the wide number of open source data science and machine learning libraries which can all integrate with pyspark. Further, java would be the least favoured choice as it's strong OO leanings feel slightly misplaced in this heavily functional domain. However, for the kafka framework java would be the preference, this is because although a kafka python library is available, it is not as widely adopted as the java implementation.

- **How can the system handle real-time analysis and alerting?**

The traffic alerting and driver monitoring systems are implemented as distinct consumers in the speed layer. Each of these subscribe to the main raw data broker and implement anomaly detection algorithms to identify outliers worth alerting. For example, clustering algorithms such as KNN or Kmeans can be used to analyse the behaviour of a particular car or driver. The same is true for road data in identifying traffic jams. For road data we can use a comparison to historic averages to identify anomalies eg. looking at specific events for a given window on a given day of the week and comparing this to the historic average for that same window on the same iso week day. More concretely, when comparing traffic events, the time, day and season are all relevant. For example we would expect traffic at 1pm on a Saturday to be different to 1pm on a Tuesday.

Therefore it is not enough to consider the hour timestamp alone when looking for anomalies, we must first filter historic data by the day and then the hour in order to compare it to live samples to determine whether that sample is an anomaly. Furthermore, seasonal data may also be relevant for example, the traffic activity for public holidays will most likely display different throughputs vs equivalent windows on non-holiday dates (eg. 1pm Tuesday 24th December vs 1pm Tuesday 5th June).

Finally, external data sources may want to be integrated to enrich the anomaly detection. An obvious example here is real time weather data, as we may expect different thresholds of activity on rainy vs dry days. Events detected as anomalies would be pushed to a queue to alert the end user application.

- **How would new datasets and queries be added to the system?**

This really depends on the nature of the data sets. For real time streaming data, adding more brokers and consumers would be simple due to the shared nothing architecture currently implemented by the kafka consumers. For slower moving data ingestion, eg. pulling data from an external third party API, separate micro services could be spun up and connected to the serving layer independent of the batch and speed processing layers. This again fits the shared nothing approach as adding and removing new API services has no impact on the existing architecture.

For adding queries data scientists are able to write their jobs in Spark and send these directly to Cassandra via spark submit jobs. If queries need to be made to the RBDMS, ideally a read replica would be used to avoid the risk of overloading a live production database with heavy read workloads.

To conclude, the architecture implements a Lambda architecture, utilising principles of horizon scaling, shared nothing and loose couple to provide a robust, highly available and elastic data service all both ad-hoc, batch querying and anomaly reporting.

Appendix A: Makefill including all Flintrock customisation commands. N.b. the `run_q2` which provides an iterative script for sending spark submit jobs to the cluster with an increasing number of nodes.

```
notebook_start:
   docker run -p 8888:8888 jupyter/pyspark-notebook

exec_pyspark_container:
    docker exec -it `docker container ls  --format='{{json .}}' | jq -r .ID` bash

MASTER_IP=ec2-54-171-74-236.eu-west-1.compute.amazonaws.com

NODES=10

CLUSTER=clo-spark-cluster-lt-${NODES}-node
# CLUSTER=clo-spark-cluster-lt-q2-${NODES}-node

flintrock_launch:
   flintrock launch ${CLUSTER} --num-slaves ${NODES}
   flintrock run-command ${CLUSTER} 'sudo yum update -y'
   flintrock run-command ${CLUSTER} 'sudo yum -y install python36 python36-pip'
   flintrock run-command ${CLUSTER} 'sudo pip install virtualenvwrapper'
   flintrock copy-file ${CLUSTER} clo_bashrc /home/ec2-user/.bashrc
   flintrock copy-file ${CLUSTER} part_1.ipynb /home/ec2-user/part_1.ipynb
   flintrock copy-file ${CLUSTER} part_2.py /home/ec2-user/part_2.py
   flintrock copy-file ${CLUSTER} requirements.txt /home/ec2-user/requirements.txt
   flintrock copy-file ${CLUSTER} chicago.shp /home/ec2-user/chicago.shp
   flintrock copy-file ${CLUSTER} chicago.dbf /home/ec2-user/chicago.dbf
   flintrock copy-file ${CLUSTER} chicago.prj /home/ec2-user/chicago.prj
   flintrock copy-file ${CLUSTER} chicago.shx /home/ec2-user/chicago.shx
   flintrock run-command ${CLUSTER} 'source .bashrc'
   flintrock run-command ${CLUSTER} 'mkvirtualenv clo --python=`which python3`'
   flintrock run-command ${CLUSTER} 'workon clo'
   flintrock run-command ${CLUSTER} 'sudo yum install gcc gcc-c++ -y'
   # https://docs.aws.amazon.com/elasticbeanstalk/latest/dg/eb-cli3-install-linux.html
   flintrock run-command ${CLUSTER} 'curl -O https://bootstrap.pypa.io/get-pip.py &&
python3 get-pip.py --user'
   flintrock run-command ${CLUSTER} 'workon clo && pip3.6 install pip install
```

```
ipykernel'
   flintrock run-command ${CLUSTER} 'workon clo && pip3.6 install jupyter'
   flintrock run-command ${CLUSTER} 'workon clo && pip3.6 install -r requirements.txt'
   # create jupyter python 3 kernel -
https://ipython.readthedocs.io/en/6.5.0/install/kernel_install.html#kernels-for-differ
ent-environments
   flintrock run-command ${CLUSTER}  'workon clo && sudo `which python` -m ipykernel
install --name "python3"'
   flintrock run-command ${CLUSTER} 'workon clo &&  export PYSPARK_PYTHON=`which
python`'
   flintrock run-command ${CLUSTER} 'echo "pyspark --master spark://0.0.0.0:7077
--packages  org.apache.hadoop:hadoop-aws:2.7.4" > run.sh && chmod +x run.sh'

CORES = 1 2 3 4
run_q2:
   # $(foreach var,$(CORES), echo $(var);)
   # Set pyspark executor to the system python, not jupyter
   $(foreach core,$(CORES), flintrock run-command ${CLUSTER} 'workon clo && export
PYSPARK_DRIVER_PYTHON=`which python` \
   && export PYSPARK_DRIVER_PYTHON_OPTS=`which python` \
   && spark-submit --master
spark://ec2-34-244-39-108.eu-west-1.compute.amazonaws.com:7077 part_2.py \
   --packages org.apache.hadoop:hadoop-aws:2.7.4 part_2.py --executor-cores $(core)';)

flintrock_login:
   flintrock login ${CLUSTER}

flintrock_destroy:
   flintrock destroy ${CLUSTER}

make tunnel:
   lsof -ti:8888 | xargs kill -9
   ssh -i spark_cluster.pem -4 -fN -L 8888:localhost:8888
ec2-user@ec2-34-255-29-19.eu-west-1.compute.amazonaws.com
```

Appendix B: Full download of part_1 ipnyb notebook file.

```
#!/usr/bin/env python
# coding: utf-8

# In[1]:



import calendar
from pyspark import SparkContext
from pyspark import SparkConf
from pyspark.sql import SQLContext, SparkSession
from pyspark.sql.functions import avg, max, sum
from datetime import datetime
from collections import namedtuple
import matplotlib.pyplot as plt
from uszipcode import SearchEngine as ZipCodeEngine
get_ipython().run_line_magic('matplotlib', 'inline')


BIG_TAXI = 's3a://chictaxi/chictaxi.csv'
SMALL_TAXI = 's3a://chictaxi/small.csv'
WEATHER = 's3a://chictaxi/weather.csv'

# sc = SparkContext()
sql_context = SQLContext(sc)


# In[2]:



# Helper functions

def get_data(sql_context, path=BIG_TAXI):
   df = sql_context.read.csv(path, header='true', inferSchema='true')
   return (df, df.rdd)


def get(x, key, default=0):
```

```
    return getattr(x, key) or default


def string_to_time(date):
    """E.g. turns '04/13/2017 07:30:00 AM' into datetime.time(6, 15).

    N.b. extra complexity here as time format isn't a simple 24hr clock;
    first convert to PM times to 24 hr format by manipulating the string,
    then convert to DateTime.
    """
    try:
        if 'PM' in date:
            time = date.split(' ')[1]
            hour =int(date.split(':')[0].split(' ')[-1])
            # Don't turn 12.30 into 24.30!
            if hour != 12:
                hour += 12
            hour = str(hour)
            _time = hour + time[2:]
            _date = date.replace(time, _time)[:-3]
        else:
            _date = date[:-3]
        # https://www.journaldev.com/23365/python-string-to-datetime-strptime
        return datetime.strptime(_date, '%m/%d/%Y %H:%M:%S')
    except ValueError:
        date

def avg_speed(x):
    try:
        return ((get(x, 'Trip Miles') / (get(x, 'Trip Seconds'))) * 60) * 60
    except ZeroDivisionError:
        return 0

    Prepared = namedtuple('Prepared', ['fare', 'tips', 'avg_speed', 'start', 'end',
'midpoint', 'miles'])
    return rdd.map(lambda x: Prepared(get(x, 'Fare'), get(x, 'Tips'), avg_speed(x),
                                      *midpoint(x), get(x, 'Trip Miles')))
```

```
def test_string_to_time():
   assert string_to_time('04/13/2017 07:30:00 PM') == datetime.datetime(2017, 4, 13,
19, 30)
   assert string_to_time('04/13/2017 07:30:00 AM') == datetime.datetime(2017, 4, 13,
7, 30)


# In[7]:


taxi_df, taxi_rdd = get_data(sql_context)


# In[4]:


#
https://spark.apache.org/docs/latest/api/python/pyspark.html?highlight=rdd#pyspark.RDD
.sample
sampled_rdd = taxi_rdd.sample(False, 0.0001, 81)


# In[5]:


def q1(rdd):
   """How many taxi records are there?
   How many taxi records for each year of the dataset?
   """
   count = rdd.count()
   yearly_counts = rdd.map(lambda x: (getattr(x, 'Trip Start Timestamp'
                                             ).split('/')[-1].split(' ')[0],
1)).reduceByKey(lambda a,b: a+b)

   return count, yearly_counts



# In[8]:
```

```
# q1 answer
# total_records, yearly_counts = q1(sampled_rdd)
total_records, yearly_counts = q1(taxi_rdd)
# yearly_counts.collect()


# In[11]:


def q2(rdd, total_records):
   """"How many records in total would you classify as bad?
       Consider a bad record to be one where the Trip Seconds are less than 60,
       but also if the average speed is over 100 mph, the distance is more
       than 1000 miles or the fare is over $2000 (excluding tips, tolls, etc).

       Once you have defined this, ensure that all further answers are based only on
good data.
       How many records are "good" by year


       N.b. for trips under 1 mile, the ave. speed = 0.0 (as miles = 0), therefore
this is a decent approximation
       without guaranteeing total accuracy as it doesn't take into account the precise
coordinates of the journey
       when calculating average speed.



   """
   good_trips = rdd.filter(lambda x: (get(x, 'Trip Seconds') > 60 )
                   & (get(x, 'Trip Miles') < 1000)
                   & (get(x, 'Fare') < 2000)
                   & (avg_speed(x) < 100))

   return good_trips, total_records - good_trips.count()
```

```
# In[12]:


# q2 answer
# good_trips, num_bad = q2(sampled_rdd, total_records)
good_trips, num_bad = q2(taxi_rdd, total_records)


# In[13]:


# num_bad / total_records
_, good_trips_by_year = q1(good_trips)


# In[13]:


print(num_bad)
print(num_bad / total_records *100)


# In[15]:


good_trips_by_year.collect()


# In[14]:


def get_2018_rides(rdd):
   return rdd.filter(lambda x: getattr(x, 'Trip Start
Timestamp').split('/')[-1].split(' ')[0] == '2018')


# In[4]:
```

```
# rides_2018 = get_2018_rides(good_trips)

# https://spark.apache.org/docs/latest/rdd-programming-guide.html#external-datasets
# Save/ Load 2018 data to avoid having to recreate from scratch
# rides_2018.saveAsPickleFile('2018_rdd')
# rides_2018_df = rides_2018.toDF()
rides_2018 = sc.pickleFile('2018_rdd')


# In[20]:


# tricky because of custom aggregation required by excluding tolls
# def q3(df):
#     df.groupBy('Taxi Id').agg({'Total Price':"avg"}).orderBy('column_name',
ascending=False)


# MapReduce approach - not complete
def q3(rdd):
   """For each taxi, calculate the average revenue per day excluding tolls (i.e. Fare
+ Tips).
   Identify the most successful taxi in 2018 in terms of total revenue (Fare + Tips).


https://stackoverflow.com/questions/29930110/calculating-the-averages-for-each-key-in-
a-pairwise-k-v-rdd-in-spark-with-pyth

   """
   return rdd.map(lambda x: (get(x, 'Taxi ID'), [get(x, 'Fare') + get(x, 'Tips'), 1])
                    ).reduceByKey(lambda x,y: (x[0]+y[0], x[1]+y[1])).mapValues(
       lambda v: v[0]/v[1]).takeOrdered(6, key=lambda x: -x[1])



# In[21]:
```

```
# q3 answer
sorted_fares = q3(rides_2018)


# In[26]:


rides_2018.count()


# In[ ]:


def test_q3_aggregation(sorted_fares,
id_='50b668c005b90b8a98cb429f7ad632b913158b885e8c0a2948c4ed8a39801ca3027d4b0e3ee313f82
046c085dd7ae8b044666fbd612e0ef663700efbf1dcc54a'):
    """Test to verify that aggregation logic is correct for q3 is correct using simple
python"""
    fares = rides_2018.filter(lambda x: get(x, 'Taxi ID') == id_).map(lambda x: [get(x,
'Fare') + get(x, 'Tips'), 1]).collect()

    acc = 0
    for x in fares:
        acc += x[0]
    assert acc / len(ble) == sorted_fares.filter(lambda x: x[0] == id_).collect()[0][1]


# In[19]:


# q4

def prepare_q4(rdd):
    """ Taking 1 hour periods throughout the day (from midnight to midnight)
    across the complete dataset, answer the following.
    Where a trip crosses a boundary (where the drop off is in a different period to the
pickup),
    assign that trip to the period where the midpoint of the journey happened.
```

```
   a. What is the average speed of taxis during each period?
   b. Which is the period where drivers in total earn the most money in
   terms of fares?
   c. Which is the period of the day where drivers in total earn the most
   in tips?


   Approach:

   - Find the create a tuple of start and end-times (S, E)
   - Convert each of these into DateTime instances
   - Find the midpoint, that is the start, + the time delta of the end - start,
   and then set the hour in scope to this hour.


   N.b. due to not being able to assign and therefore reuse variables in the context
of the lambda func,
   the start time has to be computed twice in this implementation. Whilst the code is
very concise and expressive,
   however it is slightly inefficient. A possible refactor is to use a function which
takes a row rather than the
   whole RDD and map to this.

   """
   def midpoint(x):
       """Midpoint:  lambda x: (x[0] + (x[1] - x[0]) /2).hour) , where x = (start,
end) """
       start = string_to_time(get(x, 'Trip Start Timestamp'))
       end = string_to_time(get(x, 'Trip End Timestamp'))
       return start, end, (start + (start - end) /2).hour

   Prepared = namedtuple('Prepared', ['fare', 'tips', 'avg_speed', 'start', 'end',
'midpoint', 'miles'])


   return rdd.map(lambda x: Prepared(get(x, 'Fare'), get(x, 'Tips'), avg_speed(x),
                                      *midpoint(x), get(x, 'Trip Miles')))
```

```
def q4(df):
   """ Find the max values for the prepared df

   """
   #  Various permutations of this implementation were theorised, such as doing as:

   # 1. pure spark sql implementation as below:
   # speedy = hourly_avgs.agg(max('avg_speed'))
   #     tips = hourly_avgs.agg(max('tips'))
   #     fares = hourly_avgs.agg(max('fare'))
   #     return hourly_avgs, tips, fares

   # however, this leads to greater complexity as each each .agg call returns just the
datapoint, not the row
   # i.e. crucially the midpoint is lost

   # 2. A simple sort for each feature and then take the first item of each rdd.
   # This was rejected as it is impossible to avoid three sort operations which are
not only expense operations
   # in general, but requires wide dependencies reduction, so even more expensive
here.


   # 3. A transformation based approach:

   # Transfor the row data into k, v tuples for the relevant features
   # and then find the max of each of these

   #  e.g. transform a given row rdd of format [m, s, f, t] to:
   # [(m,s), (m,f), (m,t)]
   # This would enable calling performing a filter on each position to gain the
maximum averages for each parameter.

   hourly_avgs = df.groupBy('midpoint').agg(avg('avg_speed'), avg('fare'),
avg('tips'))
#     hourly_avgs.fillna( { 'avg_speed':0, 'fare':0, 'tips':0 } )
   rdd = hourly_avgs.rdd
```

```
    Results = namedtuple('Results', ['midpoint', 'fare', 'tips', 'avg_speed'])
    results = rdd.map(lambda x: Results(x.midpoint, x[1], x[2], x[3]))

#   N.b. `or 0` handles comparision of NoneTypes as part of the max function

    max_fare = results.max(key=lambda x: x.fare or 0)
    max_tips = results.max(key=lambda x: x.tips or 0)
    max_avg_speed = results.max(key=lambda x: x.avg_speed or 0 )

    return results, max_fare, max_tips, max_avg_speed


# In[157]:


def test_q4_mid_points():

    start = string_to_time('04/13/2017 07:30:00 AM')
    end = string_to_time('04/13/2017 07:37:00 AM')
    assert (start + (end - start)/2).hour == 7

    start = string_to_time('04/13/2017 09:30:00 AM')
    end = string_to_time('04/13/2017 07:37:00 AM')
    assert (start + (end - start)/2).hour == 14

    start = string_to_time('04/13/2017 11:30:00 PM')
    end = string_to_time('04/18/2017 01:00:00 AM')
    assert (start + (end - start)/2).hour == 0


# In[20]:


# answer q4
prepared_rdd = prepare_q4(rides_2018)
# prepared_rdd.first()

# prepared_rdd.take(5)
prepared_df = prepared_rdd.toDF()
```

```
results, max_fare, max_tips, max_avg_speed = q4(prepared_df)


# In[35]:


# print(f'max_fare: {max_fare}')
# print(f'max_tips: {max_tips}')
# print(f'max_avg_speed: {max_avg_speed}')

df = results.toDF()
df.agg({"fare": "max"}).collect()[0]
df.agg({"tips": "max"}).collect()[0]
df.agg({"avg_speed": "max"}).collect()[0]

# df.show()


# In[17]:


prepared_rdd.count()


# In[36]:


def prepare_q5(prepared_rdd):
    """ What is the overall percentage of tips that drivers get?
    Find the top ten trips with the best tip per distance travelled.

    Create a graph of average tip percentage by month for the whole period.

    """
    def tips_percentage(row):
        try:
            return (row.tips / row.fare) * 100
        except ZeroDivisionError:
            return 0
```

```
   def tip_per_mile(row):
       try:
           return row.tips / row.miles
       # In the case of a trip of 0 miles, just use the tip amount
       except ZeroDivisionError:
           return row.tips



   Q5Results = namedtuple('Q5Results', ['start', 'month', 'fare', 'tips',
'tip_per_mile', 'tip_percentage_of_fare'])
   return prepared_rdd.map(lambda x: Q5Results(x.start,
calendar.month_name[x.start.month], x.fare, x.tips,
                                                                  tip_per_mile(x),
tips_percentage(x))
                                              ).sortBy(lambda x: -x.tip_per_mile)



def get_overall_tips_percentage(prepared_rdd):
       try:
           return prepared_rdd.map(lambda x: x.tips.sum() / prepared_rdd.map(lambda x:
x.fare).sum() * 100)
       except ZeroDivisionError:
           return 0


def get_tips_percentage_per_month(prepared_rdd):
      return prepared_rdd.sortBy(lambda x: x.start)




# In[ ]:


# Q5 answer
```

```
prepared_q5_rdd = prepare_q5(prepared_rdd)
generous_tippers = prepared_q5_rdd.take(10)
tippers_by_month = prepared_q5_rdd.sortBy(lambda x: x.start).take(10)


df =  prepared_q5_rdd.sortBy(lambda x: x.start).toDF().toPandas()
avg = df.groupby('month').mean()



# In[ ]:



# Q5 Plot
figure, axes = plt.subplots(1,1)
# raw_plt = axes.bar(df['start'], df['tip_percentage_of_fare'])
avg_plt = axes.bar(avg.index, avg['tip_percentage_of_fare'])


axes.set_title('Tip Percentge of total fare per month', fontsize=20)
axes.set_xlabel('Month')
axes.set_ylabel('Tip Percentge')
plt.grid()
plt.show()



# In[8]:



# Q6

# N.b. Lat and Long chosen over the Location data as this would reqiure further
parsing

def prepare_geo_data(rdd):


   Q6Results = namedtuple('Q6Results', ['start_lat', 'start_long',
                                        'end_lat', 'end_long',
                                        'start_timestamp'])
```

```
    return rdd.map(lambda x: Q6Results(get(x, 'Pickup Centroid Latitude'),
                  get(x, 'Pickup Centroid Longitude'),
                  get(x, 'Dropoff Centroid Latitude'),
                  get(x, 'Dropoff Centroid Longitude'),
                  string_to_time(get(x, 'Trip Start Timestamp'))))


# https://spark.apache.org/docs/latest/ml-clustering.html
#
https://spark.apache.org/docs/2.2.0/api/python/pyspark.ml.html#pyspark.ml.clustering.K
Means
#
https://spark.apache.org/docs/2.2.0/api/python/pyspark.ml.html#pyspark.ml.feature.Vect
orAssembler
def q6(df):
    import pandas as pd
    from pyspark.ml.clustering import KMeans
    from pyspark.ml.feature import VectorAssembler
    from pyspark.ml.evaluation import ClusteringEvaluator

    vectors = VectorAssembler(inputCols=['start_lat', 'start_long'],
                              outputCol='features', handleInvalid='skip')
    df_ = vectors.transform(df)

    kmeans = KMeans(k=308, seed=1)
    model = kmeans.fit(df_.select('features'))
    predictions = model.transform(df_)
    centers = model.clusterCenters()

    predictions.centers = pd.Series(centers)

#     evaluator = ClusteringEvaluator()
#     silhouette = evaluator.evaluate(predictions)
#     print(f'Silhouette with squared euclidean distance = {str(silhouette)}')

    print('Cluster Centers: ')
    for center in centers:
        print(center)
```

```
   return predictions, centers


# In[11]:


# answer q6
# prepared_geo_data = prepare_geo_data(rides_2018)
# predictions, centers = q6(prepared_geo_data.toDF())
# q6_answer = predictions.groupBy('prediction', 'start_lat',
'start_long').count().orderBy(
#      'count', ascending=False)
q6_answer.take(10)


# In[4]:


# prepared_geo_data.saveAsPickleFile('prepared_geo_data')
prepared_geo_data = sc.pickleFile('prepared_geo_data')


# In[5]:


# plot q6

# https://www.bigendiandata.com/2017-06-27-Mapping_in_Jupyter/

import numpy as np
import matplotlib.image as mpimg

df = q6_answer.toPandas()

df.plot(kind='scatter', x='start_long', y='start_lat', alpha=0.4)
# plt.show()

chicago_img=mpimg.imread('/home/ec2-user/chicago.png')
```

```
axes = df.plot(kind="scatter", x="start_long", y="start_lat",
   s=df['count'] *100, label="count",
              cmap=plt.get_cmap("jet"),
              colorbar=True,
              alpha=0.4, figsize=(10,7)
)

plt.imshow(chicago_img, alpha=0.5,
          interpolation='nearest' )
plt.ylabel("Latitude", fontsize=14)
plt.xlabel("Longitude", fontsize=14)

cbar = plt.colorbar()
cbar.set_label('samples in cluster', fontsize=16)

plt.legend(fontsize=8)
plt.show()




# Q7
zip_seacher = ZipCodeEngine(simple_zipcode=True)



def map_postcodes(row):
   """Return a list of tuples of zipcodes for a given trip's start end coords.

   N.b. a coordinate can map to >1 postcode, (by default uszipcode returns 5 addresses
per lat/long search),
   however sometimes these results return duplicates. In order to dedupe the results
the following method is used:

   0. construct a list of tuples of list of zipcodes pairs
       0.a query the start_lat, start_long and end_lat, end_long coordinates
       0.b assemble (start, end) pairs by zipping the results back

   1. Cast zipcodes to ints, e.g at this point the data resembles:
```

```
   [(60640, 60660),
    (60660, 60640),
    (60613, 60626),
    (60657, 60659),
    (60625, 60645)]
    # - n.b. index 0,1 are essentially dupes with the start/end positions switched

   2. sort the zipcode pairs
   3. cast the output to a set to remove dupicate entries:

   {(60613, 60626), (60625, 60645), (60640, 60660), (60657, 60659)}


   """
   # https://pypi.org/project/uszipcode/
   starts = [res.zipcode for res in zip_seacher.by_coordinates(get(row, 'start_lat'),
get(row, 'start_long'))]
   ends = [res.zipcode for res in zip_seacher.by_coordinates(get(row, 'start_lat'),
get(row, 'start_long'))]

#     starts = [res.zipcode for res in zip_seacher.by_coordinates(start_lat,
start_long)]
#     ends = [res.zipcode for res in zip_seacher.by_coordinates(start_lat,
start_long)]

   return set([tuple(sorted([int(s), int(e)])) for s,e in zip(starts, ends)])


def q7(rdd):
   Q7Results = namedtuple('Q7Results', ['zipcodes'])
   # TO DO finish this - currently errors, not sure why
   return rdd.map(map_postcodes(x))


# In[72]:


weather_df, weather_rdd = get_data(sql_context, WEATHER)
```

```
# In[196]:



# Q8
def prepare_weather_data(rdd):
   Q8Data = namedtuple('Q8Data', ['meaurement_id','rain_interval',
'intensity','station',  'timestamp', ])
   return rdd.map(lambda x: Q8Data( get(x, 'Measurement ID'), get(x, 'Interval Rain'),
                   get(x, 'Rain Intensity'),
                   get(x, 'Station Name').replace(' ', ''),
                   string_to_time(get(x, 'Measurement Timestamp')
                                 )))
                   # gladly TS format matches so this helper can be reused...

def q8(q8_data):
   """Uses a similar approach to question 6, group the data then recast to RDD and
   the max values. """

   grouped = q8_data.toDF().groupby('timestamp').mean()
   rdd = grouped.rdd



   Results = namedtuple('Results', ['timestamp', 'avg_interval', 'avg_intensity'])
   results = rdd.map(lambda x: Results(x.timestamp, x[1], x[2]))

   #   N.b. `or 0` handles comparision of NoneTypes as part of the max function
   max_interval = results.max(key=lambda x: x.avg_interval or 0)
   max_intensity = results.max(key=lambda x: x.avg_intensity or 0)

   return max_interval, max_intensity, rdd



# In[153]:



# answer q8
q8_data = prepare_weather_data(weather_rdd)
```

```
max_interval, max_intensity, rain_rdd = q8(q8_data)


# In[275]:


# Q9

def prepare_q9(rain_data_df, taxi_data_df):
   """N.b. this function isn't generic - reqiures DF in correct format to be able to
join.
   rain_data_df - e.g. as output from `q8`
   taxi_data_df - e.g. as output from `q4`

   """
   def to_unix_timestamp(isoformat):
       """Correlation not supported on timestamp data, so need to convert timestamps
to ints.
       """
       import time
       return int(time.mktime(time.strptime(isoformat, '%Y-%m-%dT%H:%M:%S')))

   # https://spark.apache.org/docs/latest/api/python/pyspark.sql.html?highlight=join
   joined = rain_data_df.join(taxi_data_df, taxi_data_df.start ==
rain_data_df.timestamp)

   Q9Results = namedtuple('Q9Results', ['timestamp', 'avg_interval', 'avg_intensity',
'fare', 'tips'])
   rdd = joined.rdd.map(lambda x: Q9Results(
       to_unix_timestamp(x.timestamp.isoformat()),
       get(x, 'avg(rain_interval)'), get(x, 'avg(intensity)'),
       x.fare, x.tips))
   return rdd

def q9(df):

#     # https://spark.apache.org/docs/2.2.0/ml-statistics.html#correlation
#     from pyspark.ml.stat import Correlation
#     from pyspark.ml.feature import VectorAssembler
```

```
#     # convert to vector column first
#     assembler = VectorAssembler(inputCols=df.columns, outputCol=vector_col,
handleInvalid='skip')
#     df_vector = assembler.transform(df).select('features')

   #
https://people.eecs.berkeley.edu/~jegonzal/pyspark/pyspark.sql.html#pyspark.sql.DataFr
ame.corr
   fare_corr = df.stat.corr("timestamp", "fare")
   tip_corr = df.stat.corr("timestamp", "tips")
   return fare_corr, tip_corr


# In[277]:


# q9_data = prepare_q9(rain_rdd.toDF(), prepared_rdd.toDF())
fare_corr, tip_corr = q9(q9_data.toDF())


# In[280]:


fare_corr


# In[ ]:
```

Appendix C: Adapted code from Appendix B to run question 1-2 as part of a spark submit job.

```
# coding: utf-8
import time
import calendar
from pyspark import SparkConf, SparkContext
from pyspark.sql import SQLContext
from pyspark.sql.functions import avg
from datetime import datetime
from collections import namedtuple

BIG_TAXI = 's3a://chictaxi/chictaxi.csv'
SMALL_TAXI = 's3a://chictaxi/small.csv'
WEATHER = 's3a://chictaxi/weather.csv'


def get_data(sql_context, path=SMALL_TAXI):
   df = sql_context.read.csv(path, header='true', inferSchema='true')
   return (df, df.rdd)


def get(x, key, default=0):
   return getattr(x, key) or default


def string_to_time(date):
   """E.g. turns '04/13/2017 07:30:00 AM' into datetime.time(6, 15).

   N.b. extra complexity here as time format isn't a simple 24hr clock;
   first convert to PM times to 24 hr format by manipulating the string,
   then convert to DateTime.
   """
   try:
       if 'PM' in date:
           time = date.split(' ')[1]
           hour = int(date.split(':')[0].split(' ')[-1])
           # Don't turn 12.30 into 24.30!
           if hour != 12:
```

```
            hour += 12
        hour = str(hour)
        _time = hour + time[2:]
        _date = date.replace(time, _time)[:-3]
    else:
        _date = date[:-3]
    # https://www.journaldev.com/23365/python-string-to-datetime-strptime
    return datetime.strptime(_date, '%m/%d/%Y %H:%M:%S')
except ValueError:
    date


def test_string_to_time():
   assert string_to_time('04/13/2017 07:30:00 PM') == datetime.datetime(
       2017, 4, 13, 19, 30)
   assert string_to_time('04/13/2017 07:30:00 AM') == datetime.datetime(
       2017, 4, 13, 7, 30)


#
https://spark.apache.org/docs/latest/api/python/pyspark.html?highlight=rdd#pyspark.RDD
.sample
# sampled_rdd = taxi_rdd.sample(False, 0.0001, 81)


def q1(rdd):
   """How many taxi records are there?
   How many taxi records for each year of the dataset?
   """
   count = rdd.count()
   yearly_counts = rdd.map(
       lambda x: (getattr(x, 'Trip Start Timestamp').split('/')[-1].split(' ')
                  [0], 1)).reduceByKey(lambda a, b: a + b)

   return count, yearly_counts


def answer_q1(rdd):
   total_records, yearly_counts = q1(rdd)
```

```
    counts = yearly_counts.collect()
    return total_records, counts


def q2(rdd, total_records):
    """"How many records in total would you classify as bad?
        Consider a bad record to be one where the Trip Seconds are less than 60,
        but also if the average speed is over 100 mph, the distance is more
        than 1000 miles or the fare is over $2000 (excluding tips, tolls, etc).

        Once you have defined this, ensure that all further answers are based
        only on good data. How many records are “good” by year.


        N.b. for trips under 1 mile, the ave. speed = 0.0 (as miles = 0),
        therefore this is a decent approximation without guarenteeing total
        accuracy as it doesn't take into account the precise
        coordinates of the journey when calculating average speed.



    """
    good_trips = rdd.filter(lambda x: (get(x, 'Trip Seconds') > 60)
                            & (get(x, 'Trip Miles') < 1000)
                            & (get(x, 'Fare') < 2000))
    #                      & ((get(x, 'Trip Miles') / (get(x, 'Trip Seconds') / 60)) <
100)

    return good_trips, total_records - good_trips.count()


def answer_q2(taxi_rdd, total_records):
    good_trips, num_bad = q2(taxi_rdd, total_records)
    num_bad / total_records
    _, good_trips_by_year = q1(good_trips)
    print(f'Bad records: {num_bad} ')
    print(f'Bad percentage: {num_bad / total_records * 100}')
    return good_trips, num_bad, good_trips_by_year
```

```
def get_2018_rides(rdd):
    return rdd.filter(lambda x: getattr(x, 'Trip Start Timestamp').split('/')[
        -1].split(' ')[0] == '2018')


def q3(rdd):
    """For each taxi, calculate the average revenue per day excluding tolls (i.e. Fare
+ Tips).
    Identify the most successful taxi in 2018 in terms of total revenue (Fare + Tips).


https://stackoverflow.com/questions/29930110/calculating-the-averages-for-each-key-in-
a-pairwise-k-v-rdd-in-spark-with-pyth

    """
    return rdd.map(lambda x: (get(
        x, 'Taxi ID'), [get(x, 'Fare') + get(x, 'Tips'), 1])).reduceByKey(
            lambda x, y: (x[0] + y[0], x[1] + y[1])).mapValues(
                lambda v: v[0] / v[1]).takeOrdered(6, key=lambda x: -x[1])


# q3 answer
def answer_q3(rides_2018):
    return q3(rides_2018)


def prepare_q4(rdd):
    """ Taking 1 hour periods throughout the day (from midnight to midnight)
    across the complete dataset, answer the following.
    Where a trip crosses a boundary (where the drop off is in a different period to the
pickup),
    assign that trip to the period where the midpoint of the journey happened.

    a. What is the average speed of taxis during each period?
    b. Which is the period where drivers in total earn the most money in
    terms of fares?
    c. Which is the period of the day where drivers in total earn the most
    in tips?
```

```
Approach:

- Find the create a tuple of start and end-times (S, E)
- Convert each of these into DateTime instances
- Find the midpoint, that is the start, + the time delta of the end - start,
and then set the hour in scope to this hour.


N.b. due to not being able to assign and therefore reuse variables in the
context of the lambda func, the start time hs to to computed twice in this
implementation. Whilst the code is very concise and expressive, however it
is slightly inefficient. A possible refactor is to use a function which
takes a row rather than the whole RDD and map to this.

"""
def midpoint(x):
    """Midpoint:  lambda x: (x[0] + (x[1] - x[0]) /2).hour) ,
    where x = (start, end)
    """
    start = string_to_time(get(x, 'Trip Start Timestamp'))
    end = string_to_time(get(x, 'Trip End Timestamp'))
    return start, end, (start + (start - end) / 2).hour

def avg_speed(x):
    try:
        return ((get(x, 'Trip Miles') /
                 (get(x, 'Trip Seconds'))) * 60) * 60
    except ZeroDivisionError:
        return 0

    Prepared = namedtuple(
        'Prepared',
        ['fare', 'tips', 'avg_speed', 'start', 'end', 'midpoint', 'miles'])
    return rdd.map(lambda x: Prepared(get(x, 'Fare'), get(
        x, 'Tips'), avg_speed(x), *midpoint(x), get(x, 'Trip Miles')))
```

```
def q4(df):
    """ Find the max values for the prepared df

    """
    hourly_avgs = df.groupBy('midpoint').agg(avg('avg_speed'), avg('fare'),
                                              avg('tips'))
    rdd = hourly_avgs.rdd

    Results = namedtuple('Results', ['midpoint', 'fare', 'tips', 'avg_speed'])
    results = rdd.map(lambda x: Results(x.midpoint, x[1], x[2], x[3]))

    #   N.b. `or 0` handles comparison of NoneTypes as part of the max function
    max_fare = results.max(key=lambda x: x.fare or 0)
    max_tips = results.max(key=lambda x: x.tips or 0)
    max_avg_speed = results.max(key=lambda x: x.avg_speed or 0)

    return max_fare, max_tips, max_avg_speed



def answer_q4(rides_2018):
    prepared_rdd = prepare_q4(rides_2018)
    prepared_df = prepared_rdd.toDF()
    max_fare, max_tips, max_avg_speed = q4(prepared_df)
    return prepared_rdd, max_fare, max_tips, max_avg_speed



def prepare_q5(prepared_rdd):
    """ What is the overall percentage of tips that drivers get?
    Find the top ten trips with the best tip per distance travelled.

    Create a graph of average tip percentage by month for the whole period.

    """
    def tips_percentage(row):
        try:
            return (row.tips / row.fare) * 100
        except ZeroDivisionError:
            return 0
```

```
    def tip_per_mile(row):
        try:
            return row.tips / row.miles
        # In the case of a trip of 0 miles, just use the tip amount
        except ZeroDivisionError:
            return row.tips

    Q5Results = namedtuple('Q5Results', [
        'start', 'month', 'fare', 'tips', 'tip_per_mile',
        'tip_percentage_of_fare'
    ])
    return prepared_rdd.map(lambda x: Q5Results(
        x.start, calendar.month_name[x.start.month], x.fare, x.tips,
        tip_per_mile(x), tips_percentage(x))).sortBy(lambda x: -x.tip_per_mile)


def get_overall_tips_percentage(prepared_rdd):
    try:
        return prepared_rdd.map(lambda x: x.tips.sum() / prepared_rdd.map(
            lambda x: x.fare).sum() * 100)
    except ZeroDivisionError:
        return 0


def get_tips_percentage_per_month(prepared_rdd):
    return prepared_rdd.sortBy(lambda x: x.start)


def answer_q5(prepared_rdd):
    prepared_q5_rdd = prepare_q5(prepared_rdd)
    generous_tippers = prepared_q5_rdd.take(10)
    tippers_by_month = prepared_q5_rdd.sortBy(lambda x: x.start).take(10)
    return prepared_q5_rdd, generous_tippers, tippers_by_month


def prepare_geo_data(rdd):
    Q6Results = namedtuple(
        'Q6Results',
        ['start_lat', 'start_long', 'end_lat', 'end_long', 'start_timestamp'])
```

```
    return rdd.map(
        lambda x: Q6Results(get(x, 'Pickup Centroid Latitude'),
                            get(x, 'Pickup Centroid Longitude'),
                            get(x, 'Dropoff Centroid Latitude'),
                            get(x, 'Dropoff Centroid Longitude'),
                            string_to_time(get(x, 'Trip Start Timestamp'))))


# https://spark.apache.org/docs/latest/ml-clustering.html
#
https://spark.apache.org/docs/2.2.0/api/python/pyspark.ml.html#pyspark.ml.clustering.K
Means
#
https://spark.apache.org/docs/2.2.0/api/python/pyspark.ml.html#pyspark.ml.feature.Vect
orAssembler
def q6(df):
    import pandas as pd
    from pyspark.ml.clustering import KMeans
    from pyspark.ml.feature import VectorAssembler

    vectors = VectorAssembler(inputCols=['start_lat', 'start_long'],
                              outputCol='features',
                              handleInvalid='skip')
    df_ = vectors.transform(df)

    kmeans = KMeans(k=308, seed=1)
    model = kmeans.fit(df_.select('features'))
    predictions = model.transform(df_)
    centers = model.clusterCenters()

    predictions.centers = pd.Series(centers)

    print('Cluster Centers: ')
    for center in centers:
        print(center)

    return predictions, centers
```

```python
def answer_q6(rides_2018):
   prepared_geo_data = prepare_geo_data(rides_2018)
   predictions, centers = q6(prepared_geo_data.toDF())
   q6_answer = predictions.groupBy('prediction', 'start_lat',
                                   'start_long').count().orderBy(
                                       'count', ascending=False)
   q6_answer.take(5)


def run(sc, sql_context):
   total_records, counts = taxi_rdd = get_data(sql_context)
   answer_q1(taxi_rdd)

   good_trips, num_bad, good_trips_by_year = answer_q2(
       taxi_rdd, total_records)
   rides_2018 = get_2018_rides(good_trips)

   answer_q3(rides_2018)

   prepared_rdd, max_fare, max_tips, max_avg_speed = answer_q4(rides_2018)

   q5_rdd, generous_tippers, tippers_by_month = answer_q5(prepared_rdd)

   answer_q6(rides_2018)


if __name__ == 'main':
   print('hi')
   # conf = SparkConf().setAppName("Q2").setMaster(
   #     "spark://ec2-34-244-39-108.eu-west-1.compute.amazonaws.com:7077")
   sc = SparkContext()
   sql_context = SQLContext(sc)
   start = time.time()
   run(sc, sql_context)
   end = time.time()
   print(f'jobs runtime: {int(end - start)}')
   sc.stop()
```

Appendix D: 'make flintock_launch && make flintock_launch' command output showing destroying and recreating an 11 node cluster.

```
Are you sure you want to destroy this cluster? [y/N]: y
Destroying clo-spark-cluster-lt-10-node...
(clo) ➜  CLO git:(master) ✗ make flintrock_launch
flintrock launch clo-spark-cluster-lt-10-node --num-slaves 10
Warning: Downloading Hadoop from an Apache mirror. Apache mirrors are often slow and unreliable, and typically only serve the most recent releases.
ustom download source. For more background on this issue, please see: https://github.com/nchammas/flintrock/issues/238
Warning: Downloading Spark from an Apache mirror. Apache mirrors are often slow and unreliable, and typically only serve the most recent releases.
stom download source. For more background on this issue, please see: https://github.com/nchammas/flintrock/issues/238
Launching 11 instances...
[34.240.48.54] SSH online.
[34.240.48.54] Configuring ephemeral storage...
[34.240.48.54] Installing Java 1.8...
[3.250.137.203] SSH online.
[3.250.137.203] Configuring ephemeral storage...
[3.250.137.203] Installing Java 1.8...
[18.203.99.6] SSH online.
[18.203.99.6] Configuring ephemeral storage...
[18.203.99.6] Installing Java 1.8...
[54.155.226.21] SSH online.
[54.78.16.183] SSH online.
[54.78.16.183] Configuring ephemeral storage...
[54.155.226.21] Configuring ephemeral storage...
[54.78.16.183] Installing Java 1.8...
[54.155.226.21] Installing Java 1.8...
[63.32.71.111] SSH online.
```

Appendix E: stack trace failure from part 1. q5. Cluster innovation.

```
Spark Executor Command: "/usr/lib/jvm/jre/bin/java" "-cp"
"/home/ec2-user/spark/conf/:/home/ec2-user/spark/jars/*:/home/ec2-user/hadoop/conf/"
"-Xmx1024M" "-Dspark.driver.port=41907"
"org.apache.spark.executor.CoarseGrainedExecutorBackend" "--driver-url"
"spark://CoarseGrainedScheduler@ip-172-31-14-54.eu-west-1.compute.internal:41907"
"--executor-id" "4" "--hostname" "172.31.4.242" "--cores" "2" "--app-id"
"app-20200824192005-0000" "--worker-url" "spark://Worker@172.31.4.242:38109"
========================================

Using Spark's default log4j profile: org/apache/spark/log4j-defaults.properties
20/08/24 19:20:06 INFO CoarseGrainedExecutorBackend: Started daemon with process name:
15578@ip-172-31-4-242
20/08/24 19:20:06 INFO SignalUtils: Registered signal handler for TERM
20/08/24 19:20:06 INFO SignalUtils: Registered signal handler for HUP
20/08/24 19:20:06 INFO SignalUtils: Registered signal handler for INT
20/08/24 19:20:06 WARN NativeCodeLoader: Unable to load native-hadoop library for your
platform... using builtin-java classes where applicable
20/08/24 19:20:06 INFO SecurityManager: Changing view acls to: ec2-user
20/08/24 19:20:06 INFO SecurityManager: Changing modify acls to: ec2-user
20/08/24 19:20:06 INFO SecurityManager: Changing view acls groups to:
20/08/24 19:20:06 INFO SecurityManager: Changing modify acls groups to:
20/08/24 19:20:06 INFO SecurityManager: SecurityManager: authentication disabled; ui acls
disabled; users  with view permissions: Set(ec2-user); groups with view permissions: Set();
users  with modify permissions: Set(ec2-user); groups with modify permissions: Set()
Exception in thread "main" java.lang.reflect.UndeclaredThrowableException
	at org.apache.hadoop.security.UserGroupInformation.doAs(UserGroupInformation.java:1713)
	at org.apache.spark.deploy.SparkHadoopUtil.runAsSparkUser(SparkHadoopUtil.scala:64)
	at
org.apache.spark.executor.CoarseGrainedExecutorBackend$.run(CoarseGrainedExecutorBackend.scala
```

```
:188)
        at 
org.apache.spark.executor.CoarseGrainedExecutorBackend$.main(CoarseGrainedExecutorBackend.scal
a:285)
        at 
org.apache.spark.executor.CoarseGrainedExecutorBackend.main(CoarseGrainedExecutorBackend.scala
)
Caused by: org.apache.spark.rpc.RpcTimeoutException: Cannot receive any reply from 
ip-172-31-14-54.eu-west-1.compute.internal:41907 in 120 seconds. This timeout is controlled by 
spark.rpc.askTimeout
        at 
org.apache.spark.rpc.RpcTimeout.org$apache$spark$rpc$RpcTimeout$$createRpcTimeoutException(Rpc
Timeout.scala:47)
        at 
org.apache.spark.rpc.RpcTimeout$$anonfun$addMessageIfTimeout$1.applyOrElse(RpcTimeout.scala:62
)
        at 
org.apache.spark.rpc.RpcTimeout$$anonfun$addMessageIfTimeout$1.applyOrElse(RpcTimeout.scala:58
)
        at scala.runtime.AbstractPartialFunction.apply(AbstractPartialFunction.scala:36)
        at scala.util.Failure$$anonfun$recover$1.apply(Try.scala:216)
        at scala.util.Try$.apply(Try.scala:192)
        at scala.util.Failure.recover(Try.scala:216)
        at scala.concurrent.Future$$anonfun$recover$1.apply(Future.scala:326)
        at scala.concurrent.Future$$anonfun$recover$1.apply(Future.scala:326)
        at scala.concurrent.impl.CallbackRunnable.run(Promise.scala:36)
        at 
org.spark_project.guava.util.concurrent.MoreExecutors$SameThreadExecutorService.execute(MoreEx
ecutors.java:293)
        at 
scala.concurrent.impl.ExecutionContextImpl$$anon$1.execute(ExecutionContextImpl.scala:136)
        at scala.concurrent.impl.CallbackRunnable.executeWithValue(Promise.scala:44)
        at scala.concurrent.impl.Promise$DefaultPromise.tryComplete(Promise.scala:252)
        at scala.concurrent.Promise$class.complete(Promise.scala:55)
        at scala.concurrent.impl.Promise$DefaultPromise.complete(Promise.scala:157)
        at scala.concurrent.Future$$anonfun$map$1.apply(Future.scala:237)
        at scala.concurrent.Future$$anonfun$map$1.apply(Future.scala:237)
        at scala.concurrent.impl.CallbackRunnable.run(Promise.scala:36)
        at 
scala.concurrent.BatchingExecutor$Batch$$anonfun$run$1.processBatch$1(BatchingExecutor.scala:6
3)
        at 
scala.concurrent.BatchingExecutor$Batch$$anonfun$run$1.apply$mcV$sp(BatchingExecutor.scala:78)
        at 
scala.concurrent.BatchingExecutor$Batch$$anonfun$run$1.apply(BatchingExecutor.scala:55)
        at 
scala.concurrent.BatchingExecutor$Batch$$anonfun$run$1.apply(BatchingExecutor.scala:55)
        at scala.concurrent.BlockContext$.withBlockContext(BlockContext.scala:72)
        at scala.concurrent.BatchingExecutor$Batch.run(BatchingExecutor.scala:54)
        at scala.concurrent.Future$InternalCallbackExecutor$.unbatchedExecute(Future.scala:601)
```

```
        at scala.concurrent.BatchingExecutor$class.execute(BatchingExecutor.scala:106)
        at scala.concurrent.Future$InternalCallbackExecutor$.execute(Future.scala:599)
        at scala.concurrent.impl.CallbackRunnable.executeWithValue(Promise.scala:44)
        at scala.concurrent.impl.Promise$DefaultPromise.tryComplete(Promise.scala:252)
        at scala.concurrent.Promise$class.tryFailure(Promise.scala:112)
        at scala.concurrent.impl.Promise$DefaultPromise.tryFailure(Promise.scala:157)
        at
org.apache.spark.rpc.netty.NettyRpcEnv.org$apache$spark$rpc$netty$NettyRpcEnv$$onFailure$1(NettyRpcEnv.scala:206)
        at org.apache.spark.rpc.netty.NettyRpcEnv$$anon$1.run(NettyRpcEnv.scala:243)
        at java.util.concurrent.Executors$RunnableAdapter.call(Executors.java:511)
        at java.util.concurrent.FutureTask.run(FutureTask.java:266)
        at
java.util.concurrent.ScheduledThreadPoolExecutor$ScheduledFutureTask.access$201(ScheduledThreadPoolExecutor.java:180)
        at
java.util.concurrent.ScheduledThreadPoolExecutor$ScheduledFutureTask.run(ScheduledThreadPoolExecutor.java:293)
        at java.util.concurrent.ThreadPoolExecutor.runWorker(ThreadPoolExecutor.java:1149)
        at java.util.concurrent.ThreadPoolExecutor$Worker.run(ThreadPoolExecutor.java:624)
        at java.lang.Thread.run(Thread.java:748)
Caused by: java.util.concurrent.TimeoutException: Cannot receive any reply from
ip-172-31-14-54.eu-west-1.compute.internal:41907 in 120 seconds
        ... 8 more
```

Appendix F: bashrc file - note this is sent and loaded onto the cluster as part of the `make flintrock_launch` command. Inspired by https://raw.githubusercontent.com/pzfreo/ox-clo/master/code/flintrock-jupyter.sh

```
# .bashrc

# Source global definitions
if [ -f /etc/bashrc ]; then
        . /etc/bashrc
fi

# User specific aliases and functions
export HADOOP_LIBEXEC_DIR='/home/ec2-user/hadoop/libexec'
export SPARK_HOME='/home/ec2-user/spark'
export PYSPARK_PYTHON=/usr/bin/python3
export WORKON_HOME=$HOME/.virtualenvs
export AWS_ACCESS_KEY_ID=AKIAT6VGICDBXRWLOQMT
export AWS_SECRET_ACCESS_KEY=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
```

```
export PYTHONPATH=$SPARK_HOME/python:$PYTHONPATH
#export PYSPARK_PYTHON=~/.virtualenvs/clo/bin/python
export PYSPARK_DRIVER_PYTHON=jupyter
export PYSPARK_DRIVER_PYTHON_OPTS='notebook --no-browser'
source /usr/local/bin/virtualenvwrapper.sh
```

Appendix F: question 2 clutter, showing 4 running nodes and a range of jobs which successfully executed, but wit runtime of ~1 second and no meaning output in stderr.

Workers (4)

Worker Id	Address	State	Cores	Memory
worker-20200825010121-172.31.1.97-44443	172.31.1.97:44443	ALIVE	2 (0 Used)	6.5 GB (0.0 B Used)
worker-20200825010121-172.31.12.151-45389	172.31.12.151:45389	ALIVE	2 (0 Used)	6.5 GB (0.0 B Used)
worker-20200825010121-172.31.4.204-46631	172.31.4.204:46631	ALIVE	2 (0 Used)	6.6 GB (0.0 B Used)
worker-20200825010121-172.31.9.189-38831	172.31.9.189:38831	ALIVE	2 (0 Used)	6.5 GB (0.0 B Used)

Running Applications (0)

Application ID	Name	Cores	Memory per Executor	Submitted Time	User	State	Duration

Completed Applications (25)

Application ID	Name	Cores	Memory per Executor	Submitted Time	User	State	Duration
app-20200825023340-0024	part_2.py	8	1024.0 MB	2020/08/25 02:33:40	ec2-user	FINISHED	1 s
app-20200825023340-0023	part_2.py	8	1024.0 MB	2020/08/25 02:33:40	ec2-user	FINISHED	1 s
app-20200825023340-0020	part_2.py	8	1024.0 MB	2020/08/25 02:33:40	ec2-user	FINISHED	0.9 s
app-20200825023340-0022	part_2.py	0	1024.0 MB	2020/08/25 02:33:40	ec2-user	FINISHED	0.5 s
app-20200825023340-0021	part_2.py	0	1024.0 MB	2020/08/25 02:33:40	ec2-user	FINISHED	0.6 s
app-20200825023332-0019	part_2.py	8	1024.0 MB	2020/08/25 02:33:32	ec2-user	FINISHED	1 s
app-20200825023332-0018	part_2.py	8	1024.0 MB	2020/08/25 02:33:32	ec2-user	FINISHED	1 s
app-20200825023331-0015	part_2.py	8	1024.0 MB	2020/08/25 02:33:31	ec2-user	FINISHED	1 s
app-20200825023332-0016	part_2.py	0	1024.0 MB	2020/08/25 02:33:32	ec2-user	FINISHED	0.8 s
app-20200825023332-0017	part_2.py	0	1024.0 MB	2020/08/25 02:33:32	ec2-user	FINISHED	0.5 s
app-20200825023325-0014	part_2.py	8	1024.0 MB	2020/08/25 02:33:25	ec2-user	FINISHED	1 s
app-20200825023325-0011	part_2.py	8	1024.0 MB	2020/08/25 02:33:25	ec2-user	FINISHED	1 s

Bibliography -

(N.b. comments are given in code (Appendix A) where a resource has been used to inform the approach implemented)

https://spark.apache.org/docs/latest/rdd-programming-guide.html#external-datasets
https://spark.apache.org/docs/latest/api/python/pyspark.html?highlight=rdd#pyspark.RDD.sample
https://www.journaldev.com/23365/python-string-to-datetime-strptime
https://stackoverflow.com/questions/29930110/calculating-the-averages-for-each-key-in-a-pairwise-k-v-rdd-in-spark-with-pyth
https://spark.apache.org/docs/latest/api/python/pyspark.html#pyspark.RDD.reduceByKeyLocally
https://spark.apache.org/docs/latest/ml-clustering.html
https://spark.apache.org/docs/2.2.0/api/python/pyspark.ml.html#pyspark.ml.clustering.KMeans
https://spark.apache.org/docs/2.2.0/api/python/pyspark.ml.html#pyspark.ml.feature.VectorAssembler
https://www.bigendiandata.com/2017-06-27-Mapping_in_Jupyter/
https://pypi.org/project/uszipcode/
https://spark.apache.org/docs/latest/api/python/pyspark.sql.html?highlight=join

https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.fillna.html