CISC 5950 Project 1

Chih-You Chen, William Cheung, Nicola Damian, Debarshi Dutta, Qiangwen Xu March 27^{th}

1 Introduction

In this project we seek to apply a three node cluster with Hadoop Distributed File System (HDFS) to solve real world questions. We will be exploring parking violation and National Basketball Association (NBA) data.

2 NY Parking Violations

2.1 Overview

The New York parking violation database contains tickets issued for vehicle parking infractions. The data is organized by fiscal year (FY) streching back from FY2020 - FY 2014. At the time of this project 8.52 million records have been issued. The data is organized in table format with the following 43 features provided: *Summons Number, Plate ID, Registration State, Plate Type, Issue Date, Violation Code, Vehicle Body Type, Vehicle Expiration Date, Violation Location, Violation Prescient, Issuer Precinct, Issuer Code, Issuer Command, Issuer Squad, Violation Time, Time First Observed, Violation County, Violation In Front Of Or Opposite, House Number, Street Name, Intersecting Street, Date First Observed, Law Section, Sub Division, Violation Legal Code, Days Parking in Effect, From Hours In Effect, To Hours In Effect, Vehicle Color, Unregistered Vehicle?, Vehicle Year, Meter Number, Feet From Curb, Violation Post Code, Violation Description, No Standing or Stopping Violation, Hydrant Violation, Double Parking Violation*

This database format downloads in csv format making calling the respectively needed columns convent. However, this database is most likely used by NY officers and human errors were apparent in the database. Although we face issues with missing data there are enough in most categories where it is safe to drop tickets with blanks in needed fields. Misspellings and different uses of acronyms also rose some issues and question specific solutions were developed and detailed in the following sections with the greatest difficulty in 2.5.

2.2 Time of Highest Ticket Issuance

We liked to know what time of day are the tickets most likely to be issued. The data base has a *Violation Time* feature that signifies the time of day of the violation in a 12-hour-clock format. Our strategy in interpreting this is to convert the given time to military time. Additionally to better make sense of the data we generalize the time granularity to the hour.

Fig. 1 shows the mapper code which help output the counts of the tickets against the military hour in which it occurred. To save time and cut down on traffic a combiner was used to summarize the total count. Mapper output can be summarizes as <hour, ticket count>.

```
#!/usr/bin/python
# --*-- coding:utf-8 --*--
import re
import sys

pat = re.compile('(?P<hour>\d{4}[A|P])')
matched = {}
for line in sys.stdin:
    match = pat.search(line)
    if match:
        time = match.group('hour')
        if ('A' in time):
            time = int(time[:2])
        elif ('P' in time):
            time = int(time[:2])+12
        else:
            time = int(time[:2])
        if time not in matched.keys():
            matched[time] = 1
        else:
            matched[time] += 1

for i in matched.items():
    print '%s\t%s' % (i[0], i[1])
```

Figure 1: Time of Highest Ticket issuer mapper

In the given format of our cluster this map-reduce is designed to funnel to one reducer. This generally is a safe assumption because we have already combined most of the information in a way that even with many mappers each is at most outputting 24 key-value pairs and is difficult to overwhelm one reducer. With this design of one reducer we can safely sum the ticket counts by hour and output the sorted list by ticket count as the local maximum and global maximum are the same. Fig. 2 shows the reducer code with the output of <total ticket count, hour>.

```
#!/usr/bin/python
from operator import itemgetter
import sys

dict_ip_count = {}

for line in sys.stdin:
    line = line.strip()
    time, num = line.split('\t')
    try:
        num = int(num)
        dict_ip_count[time] = dict_ip_count.get(time, 0) + num

    except ValueError:
        pass

sorted_dict_ip_count = sorted(dict_ip_count.items(), key=itemgetter(1), reverse=True)
for time, count in sorted_dict_ip_count:
    print '%s\t%s' % (time, count)
```

Figure 2: Time of Highest Ticket Issuance reducer

The project has found as seen in Fig. 3 that **8am, 9am, and 11am** is the most likely time for tickets. They have ticket counts of **76,682, 76,620, 75,758** respectively. This is not surprising because that is when people are rushing to go to work to those who are late to work. It is plausible to believe that they did not remember or skipped paying to meter to get to a meeting.

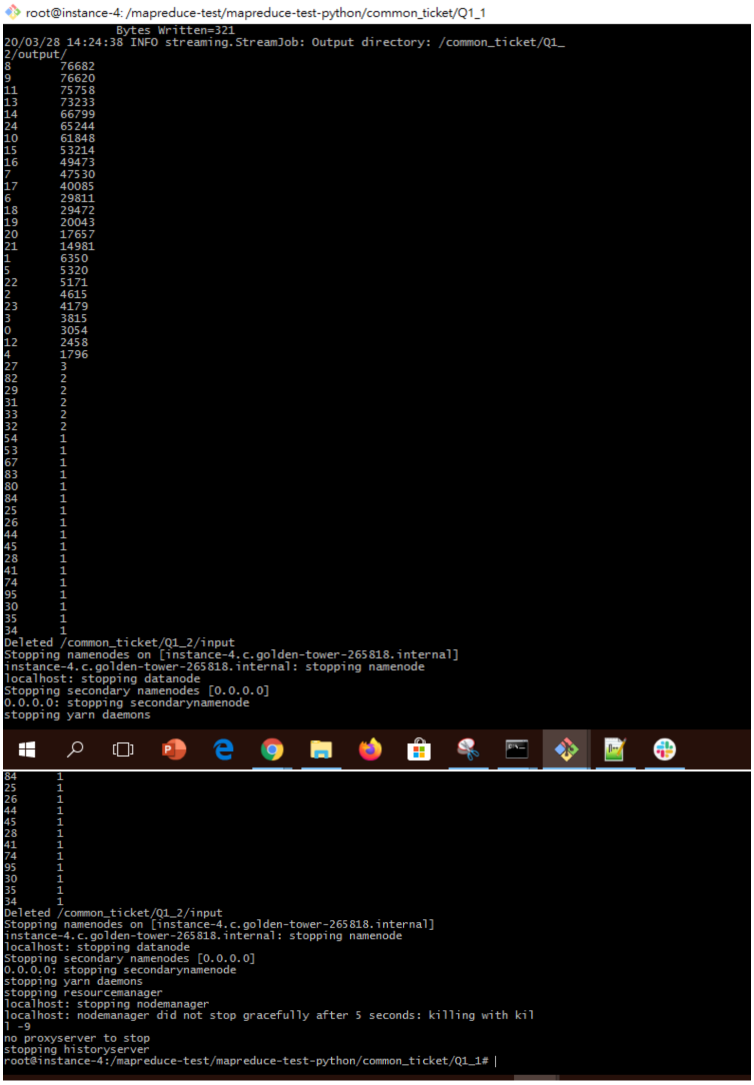


Figure 3: Time of Highest Ticket Issuance output

2.3 Most Common Year and Types of Cars Ticketed

In the pursuit of solving the most common year and types of cars to be ticketed we must first carefully define its premise. With respect to the year of the car that can be directly defined as model year of the car which can be found in the feature *Vehicle Year*. When considering type we must want to understand what would be the most interesting and insightful option. Although brands names can appeal to certain demographics in industries such as clothing, the vehicle body type is more applicable to car industry. Vehicle body type can be a greater link between type of car and of type of driver. With this in mind we used the *Vehicle Body Type* feature to define type of car.

Drawing directly on the table format of the data set we can get both the color and car type and combine them as a key. A combiner was used to reduce the output by summing the the counts if repeated keys. The output of the mapper is <(color+body type), ticket count> as seen in 4.

```
#!/usr/bin/python
# --*-- coding:utf-8 --*--
import csv
import sys


reader = csv.reader(sys.stdin)
matched = {}

for row in reader:
    try:
        match = row[6]
        match2 = row[35]
        if((len(match)>=2&len(match)<=4)&(len(match2)==4)):
            key = str(match)+str(match2)
            if key not in matched.keys():
                matched[key] = 1
            else:
                matched[key] += 1
        else:
            continue
    except:
        continue

for i in matched.items():
    print '%s\t%s' % (i[0], i[1])
```

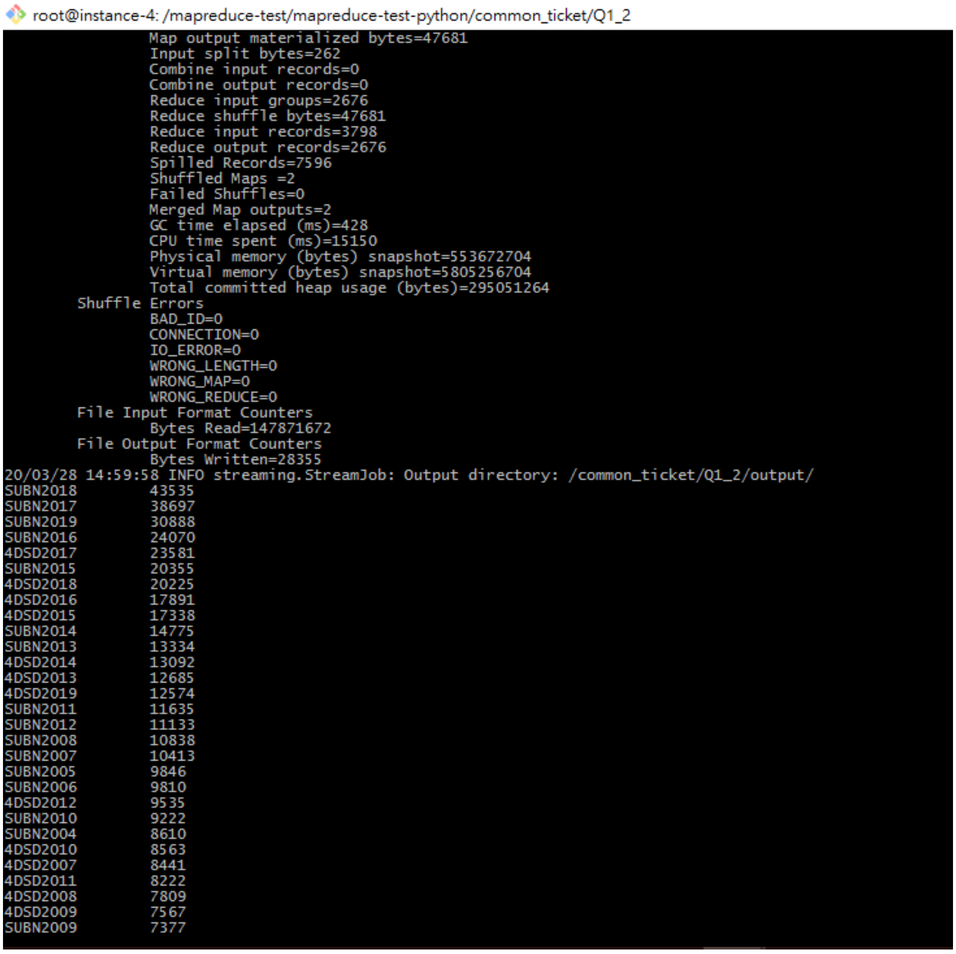
Figure 4: Most Common Year and Types of Cars Ticketed mapper

Given the key value pair and the same assumptions on single reducer system in 2.2, we deploy the same sorting. The output is in the format of <total ticket count, color+body type > as seen below in fig. 5.

```
#!/usr/bin/python
from operator import itemgetter
import sys

dict_ip_count = {}

for line in sys.stdin:
    line = line.strip()
    vehicle, num = line.split('\t')
    try:
        num = int(num)
        dict_ip_count[vehicle] = dict_ip_count.get(vehicle, 0) + num

    except ValueError:
        pass

sorted_dict_ip_count = sorted(dict_ip_count.items(), key=itemgetter(1), reverse=True)

for vehicle, count in sorted_dict_ip_count:
    print '%s\t%s' % (vehicle, count)
```

Figure 5: Most Common Year and Types of Cars Ticketed reducer

The output fig.6 below shows that many of the tickets are classified suburban (SUBN). According to the NY State Department of Motor Vehicle (NYDMV) SUBN is defined as that has windows on the side in the rear and has seats in the rear that can be folded or removed so the vehicle can carry cargo. Is is a very borad definition that includes but not limited to Sedans (SEDN), two-door sedans (2DSD) and four-door sedans (4DSD). Because of this its better to look past the SUBN category and the top 3 afterward are: **4DSD2017 (43,535 tickets), 4DSD2018 (38,697 tickets), 4DSD2016 (30,888 tickets)**. This means more recent versions of four door sedans are the most likely to be ticketed.



```
root@instance-4: /mapreduce-test/mapreduce-test-python/common_ticket/Q1_2
                Map output materialized bytes=47681
                Input split bytes=262
                Combine input records=0
                Combine output records=0
                Reduce input groups=2676
                Reduce shuffle bytes=47681
                Reduce input records=3798
                Reduce output records=2676
                Spilled Records=7596
                Shuffled Maps =2
                Failed Shuffles=0
                Merged Map outputs=2
                GC time elapsed (ms)=428
                CPU time spent (ms)=15150
                Physical memory (bytes) snapshot=553672704
                Virtual memory (bytes) snapshot=5805256704
                Total committed heap usage (bytes)=295051264
        Shuffle Errors
                BAD_ID=0
                CONNECTION=0
                IO_ERROR=0
                WRONG_LENGTH=0
                WRONG_MAP=0
                WRONG_REDUCE=0
        File Input Format Counters
                Bytes Read=147871672
        File Output Format Counters
                Bytes Written=28355
20/03/28 14:59:58 INFO streaming.StreamJob: Output directory: /common_ticket/Q1_2/output/
SUBN2018        43535
SUBN2017        38697
SUBN2019        30888
SUBN2016        24070
4DSD2017        23581
SUBN2015        20355
4DSD2018        20225
4DSD2016        17891
4DSD2015        17338
SUBN2014        14775
SUBN2013        13334
4DSD2014        13092
4DSD2013        12685
4DSD2019        12574
SUBN2011        11635
SUBN2012        11133
SUBN2008        10838
SUBN2007        10413
SUBN2005        9846
SUBN2006        9810
4DSD2012        9535
SUBN2010        9222
SUBN2004        8610
4DSD2010        8563
4DSD2007        8441
4DSD2011        8222
4DSD2008        7809
4DSD2009        7567
SUBN2009        7377
```

Figure 6: Most Common Year and Types of Cars Ticketed reducer

2.4 Location with Most Tickets

As we try to look into the features there were even more options of features to look into for location than car type. From various *street code* types, *intersecting street* to *street name* all give different view of the same locations. However two of the three types listed has severe complications. In trying to interpret the three *street code* types no documentation on the NYDMV website were found. When trying to input into google maps many codes lead to locations outside NY or did not exist at all. Because of this the *street code* types were not usable also given key translation may have given greater insight then the other two features. *Intersecting street* has its own issues. Over 5.8 million of tickets have no value for feature. Also we could simply dump tickets with missing values in other questions this is more than 60% tickets that need to be thrown out. This level of null values is understandable as most parking are along streets and most likely not at an intersection. And for those reasons the *street name* was chosen.

Similar map-reduce format is used from previous questions after accessing the feature complications. The output has been provided below (Fig.8). The top two streets **Broadway** with **9,519** tickets and **WB Seagrit Boulevard at CR 3rd avenue** with **8,441** tickets almost double the tickets of the third place street.

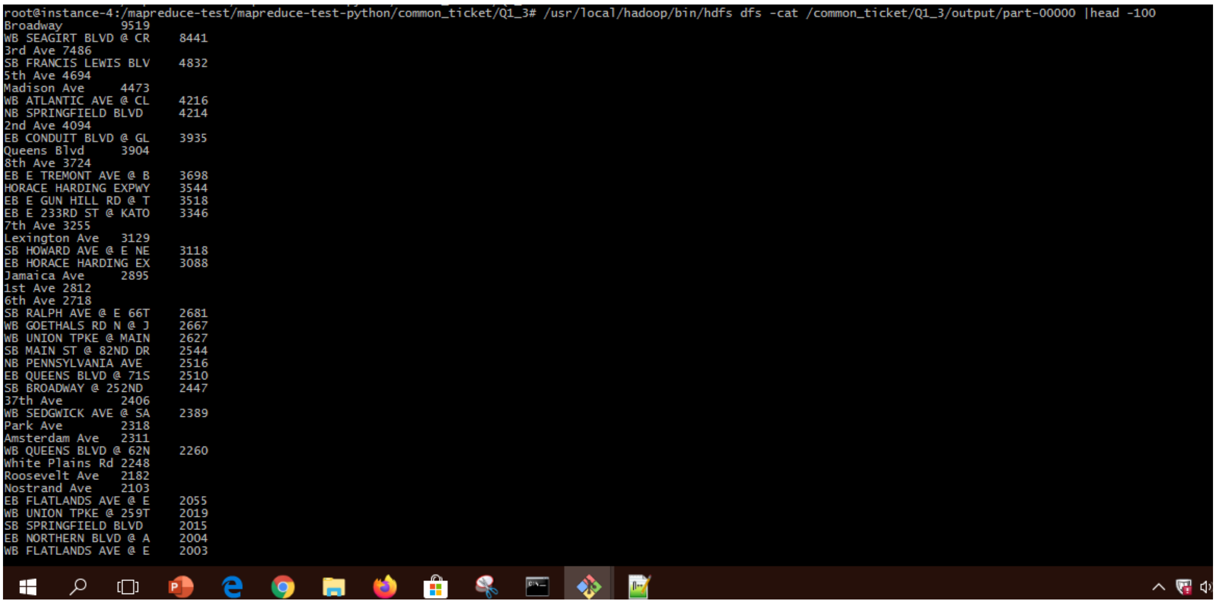


Figure 7: Location with most tickets output

2.5 Color of Vehicle That is Most Likely to be Ticketed

Lots of studies show that color has an effect on people's perception of a driver. Red is commonly associated with speed and black cars resembling elegance. One aspect we seek to analyse is the association of color with being ticketed. At first the feature *vehicle color* seem to give us the only and best option. However after viewing the color types at a high level we see that there are a lot of misspellings and mixed use of abbreviations.

As the output in Fig.9 suggests- **black** cars are in fact the most likely to be ticketed. The top three are by far much grater than other colors. Black cars ticketed are in fact ticketed **260,884** times from the data available. The second color on the list is white- with **218,799** white cars ticketed. The next is **gray** cars being ticketed **211,019** times.

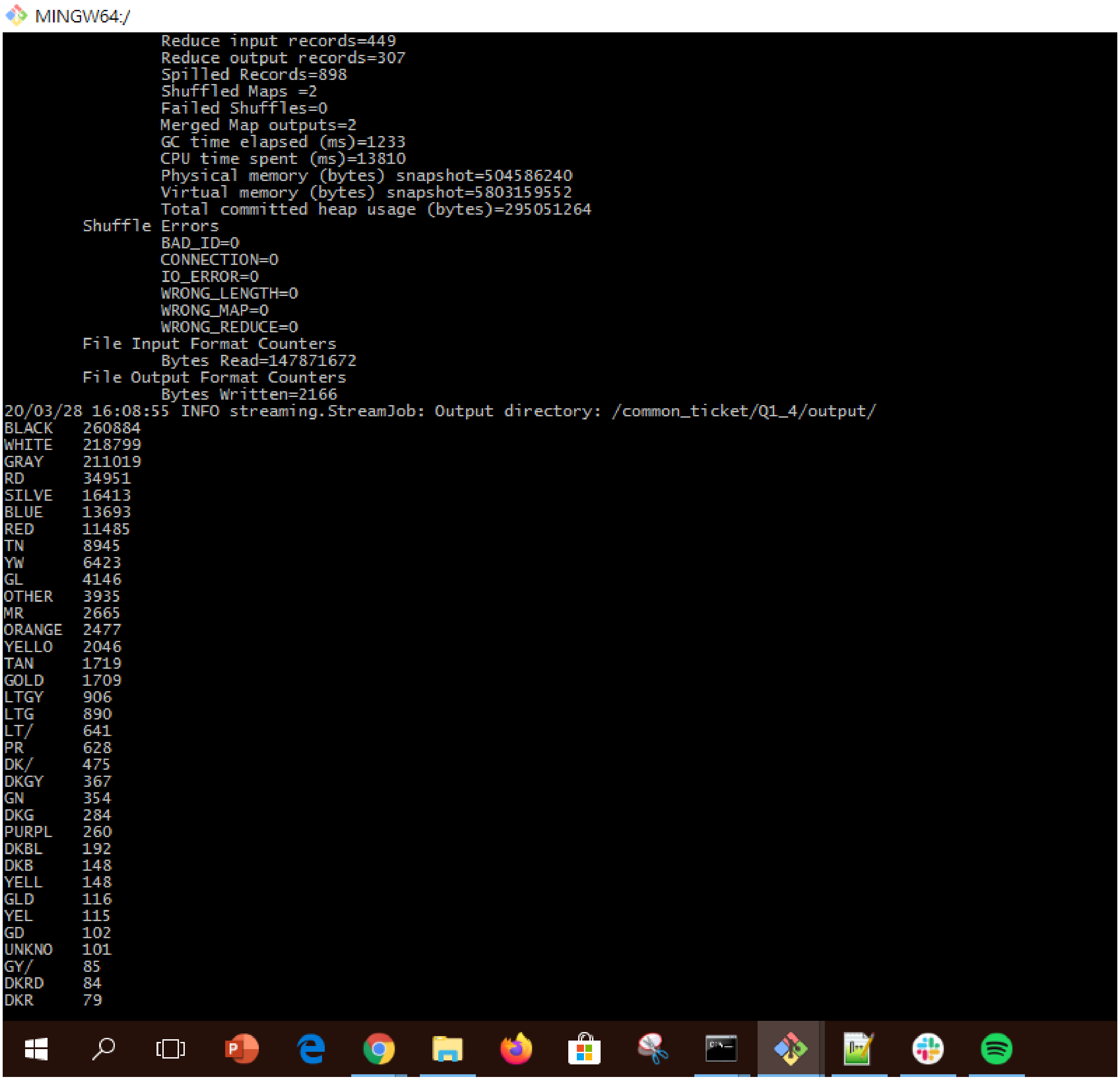


Figure 8: Color of Vehicle That is Most Likely to be Ticketed

3 NBA Shot Logs

3.1 Overview

In a recent trend the NBA has now incorporated analytics into its game-play. This has evolved the game dramatically and drown teams to play in a way that optimize points per play. One extreme example is the Huston Rockets, who have predominately shoot three-pointers and drive in for lay-ups as analytic has seen those plays are the highest yielding. We seek to use analytics to find similar insights but on a player basis to figure their strengths and weaknesses.

The data set is from Kaggle and consists of 21 features: *Game ID, Matchup, Location, W, Final Margin, Shot Number, Period, Game Clock, Shot Clock, Dribbles, Touch Time, Shot Dist, PTS Type, Shot Result, Closest Defender, Closest Defender Player ID, Close Def Dist, FGM, PTS, Player Name, Player ID* The data set was uploaded 2016 with 128 thousand datapoints.

3.2 Most Feared Defenders

Although there are great all-around defenders, with many types of players certain defenders are better limiting certain players in a match-up. Therefore, we define fear score as the defender in which the player struggles scoring on the most. Because the the data's granularity at per shot level we have to get the total information per player in one reducer. This project entails a three node cluster as mentioned in 1, and so it is a guarantee that all info will be in one reducer node.

The mapper, as seen in Fig. 9 takes into account all shots taken from one player to another defender. We we split the *Shot_Result* into 2 binary features made shot and shot (shot always 1). To reduce traffic, we use a combiner to sum these items in the format of <player/defender pair, sum of made shots, sum of shots taken>

```
#!/usr/bin/python
# --*-- coding:utf-8 --*--
import csv
import sys

reader = csv.reader(sys.stdin)

matched = {}


for row in reader:
    pair = str(row[19])+'@'+str(row[14]).replace(',', '').lower() # construct the "pair" as the key for Mapper
    result = row[13] # shot result
    if result == 'missed':
        shot = 0
    else:
        shot = 1
    if pair not in matched.keys():
        # construct a tuple (scored count, shot count) as the value
        matched[pair] = (shot, 1) #(1,1) or (0,1), means one scored in one shot / one missed in one shot
    else:
        matched[pair] = (matched[pair][0]+shot, matched[pair][1]+1) # accumulate the score count and the shot count


for i in matched.items():
    print '%s\t%sm%s' % (i[0], i[1][0], i[1][1])

'''
# Column Number in the csv, as reference
0 GAME_ID
1 MATCHUP
2 LOCATION
3 W
4 FINAL_MARGIN
5 SHOT_NUMBER
6 PERIOD
7 GAME_CLOCK
8 SHOT_CLOCK
9 DRIBBLES
10 TOUCH_TIME
11 SHOT_DIST
12 PTS_TYPE
13 SHOT_RESULT
14 CLOSEST_DEFENDER
15 CLOSEST_DEFENDER_PLAYER_ID
16 CLOSE_DEF_DIST
17 FGM
18 PTS
19 player_name
20 player_id
'''
```

Figure 9: The mapper of the Most Feared Defender

The reducer , in Fig. 10, then splits the player/defender pair so that we can create a sub dictionary to sort the defender's performance against the player. To judge performance we take into account a two tiered metric. We first calculate the rate of scoring (made/all shots) that the player has when defended by the defender. Given the case of a tie between defenders we check the total shots against the defender. Given the same rate the defender with the higher shot count is the more feared defender. The most feared defender of any player is then deemed as the defender in which the play's rate of scoring is lowest with the highest shots taken against.

```
#!/usr/bin/python
import sys


stat = {}
# get the keys and values
for line in sys.stdin:
    line = line.strip()
    pair, shot_raw = line.split('\t')
    a, b = shot_raw.split('m')
    shot = (int(a), int(b))
    # combaine them
    if pair not in stat.keys():
        stat[pair] = shot
    else:
        stat[pair] = tuple((stat[pair][0]+shot[0], stat[pair][1]+shot[1]))


dict_player = {}
for pair, shot in stat.items():
    # cut the pair into shooter and defender, and calculate the hit rate
    shooter, defender = pair.split('@')
    rate = int(shot[0])/int(shot[1])
    # store the (key: defender, value: hit rate) by shooter
    if shooter not in dict_player.keys():
        dict_player[shooter] = {defender: (rate, shot[1])}
    else:
        dict_player[shooter][defender] = (rate, shot[1])


# sort the data for each shooter
for player, records in dict_player.items():
    rate_rank = sorted(records.items(), key = lambda item:item[1][0])
    good_defender = []
    try:
        for record in rate_rank:
            if record[1][0] == 0.0:
                good_defender.append(record)
        shot_rank = sorted(good_defender, key = lambda item:item[1][1], reverse=True)
        print '%s\t%s' % (player, shot_rank[0]) # print the most unwanted defender for each shooter
    except:
        print '%s\t%s' % (player, rate_rank[0])  # print the most unwanted defender for each shooter
```

Figure 10: The mapper of the Most Feared Defender

The results sample can be seen in 11. One of the results is that player Kevin Garnett has never scored on Nene in 16 shot attempts.

```
20/03/30 03:08:29 INFO streaming.StreamJob: Output directory: /CISC5950P1Q2.1/output/
kevin garnett   ('nene', (0, 16))
nick young      ('miles cj', (0, 11))
kentavious caldwell-pope        ('temple garrett', (0, 16))
anthony morrow  ('iguodala andre', (0, 9))
jerome jordan   ('gasol pau', (0, 7))
roy hibbert     ('drummond andre', (0, 21))
reggie jackson  ('wall john', (0, 17))
jordan hill     ('kaman chris', (0, 19))
derrick favors  ('duncan tim', (0, 22))
lou williams    ('larkin shane', (0, 16))
demarre carroll ('oladipo victor', (0, 11))
pau gasol       ('mozgov timofey', (0, 37))
andre iguodala  ('harden james', (0, 7))
elfrid payton   ('walker kemba', (0, 19))
chris copeland  ('pierce paul', (0, 11))
paul millsap    ('monroe greg', (0, 18))
jeff teague     ('walker kemba', (0, 20))
kyle lowry      ('teague jeff', (0, 18))
blake griffin   ('kanter enes', (0, 38))
joe harris      ('mcdaniels kj', (0, 4))
steve adams     ('asik omer', (0, 10))
zach randolph   ('chandler tyson', (0, 30))
trey burke      ('nelson jameer', (0, 20))
jason terry     ('conley mike', (0, 9))
pj tucker       ('harden james', (0, 15))
deron williams  ('rose derrick', (0, 16))
greivis vasquez ('schroder dennis', (0, 13))
rasual butler   ('muhammad shabazz', (0, 11))
luol deng       ('johnson joe', (0, 13))
nick collison   ('humphries kris', (0, 6))
jonas jerebko   ('collison nick', (0, 9))
damjan rudez    ('gallinari danilo', (0, 8))
lance stephenson        ('hill solomon', (0, 12))
glen davis      ('baynes aron', (0, 6))
andrew bogut    ('hill jordan', (0, 12))
kawhi leonard   ('gay rudy', (0, 19))
cole aldrich    ('drummond andre', (0, 9))
manu ginobili   ('mclemore ben', (0, 15))
kenneth faried  ('ibaka serge', (0, 14))
shaun livingston        ('brewer corey', (0, 7))
matt barnes     ('hayward gordon', (0, 16))
mario chalmers  ('calderon jose', (0, 10))
mnta ellis      ('butler jimmy', (0, 23))
jarrett jack    ('jennings brandon', (0, 16))
danny green     ('thompson klay', (0, 11))
cody zeller     ('smith jason', (0, 7))
courtney lee    ('mclemore ben', (0, 12))
jared dudley    ('millsap paul', (0, 9))
jeremy lamb     ('butler caron', (0, 10))
robert covington        ('harris tobias', (0, 10))
james johnson   ('james lebron', (0, 12))
john henson     ('davis ed', (0, 9))
patrick patterson       ('nowitzki dirk', (0, 11))
bismack biyombo ('smith jason', (0, 4))
aaron gordon    ('ibaka serge', (0, 5))
enes kanter     ('griffin blake', (0, 28))
darrell arthur  ('collison nick', (0, 10))
carl landry     ('plumlee miles', (0, 13))
chris kaman     ('koufos kosta', (0, 13))
kyle korver     ('ross terrence', (0, 14))
darren collison ('curry stephen', (0, 23))
wesley matthews ('roberson andre', (0, 14))
lebron james    ('wiggins andrew', (0, 25))
norris cole     ('williams deron', (0, 12))
```

Figure 11: Player's Most Feared Defender

3.3 Comfort Zones

In this questions we try to find the comfort zone of four players: James Harden, Chris Paul, Steph Curry, and Lebron James. Comfort zone is defined as the highest shot percentage from a set of {*Shot_Dist*, *Close_Def_Dist*, *Shot_Clock*}. This is solved through K-means clustering on data pertaining to each of the four selected players

The mapper, Fig. 12, takes the given centriods and computes the distance and resulting closest centriod the data point as defined by the set. The information is then sent to the reducer seen in Fig. 13. The reducer receiving the total information can recompute the centriods and checks if there was a change from starting centriods and ending centriods. If there is then another iteraltion will be run, otherwise the centriods will be the comfort zone. The output for the comfort zones for LeBron James, Chris Paul, James Harden and Stephen Curry are provided in fig.14. In Fig. 15 , Lebron James for example, we can see that as we increase the iterations of the loop for the K-means the results improve.

```
#!/usr/bin/python
"""mapper.py"""

import sys
from math import sqrt
from random import sample
import csv

# get shot record data of target player
def get_data(name):
    reader = csv.reader(sys.stdin)
    PLAYER_NAME = name # The player you want to analyze
    shots_record = {}
    for row in reader:
        # loop the data set to find all records of target player
        if row[19].replace(',', '').lower() == PLAYER_NAME:
            try:
                SHOT_DIST = float(row[11])
                CLOSE_DEF_DIST = float(row[16])
                SHOT_CLOCK = float(row[8])
            except:
                # There always be some errors, because some "SHOT_CLOCK" values can not be converted to float data type
                continue
            # construct the key and value, similar to that in 1st question
            key = (SHOT_DIST, CLOSE_DEF_DIST, SHOT_CLOCK)
            result = row[13]
            if result == 'missed':
                shot = 0
            else:
                shot = 1
            # store the raw data record of comfortable zone shoots
            if key not in shots_record.keys():
                shots_record[key] = (shot, 1)
            else:
                shots_record[key] = (shots_record[key][0]+shot, shots_record[key][1]+1)
    return shots_record

# get initial centroids from a txt file and add them in an array
def getCentroids(filepath):
    centroids = []
    with open(filepath) as fp:
        line = fp.readline()
        while line:
            if line:
                try:
                    line = line.strip()
                    cord = line.split(', ')
                    centroids.append([float(cord[0]), float(cord[1]), float(cord[2])])
                except:
                    break
            else:
                break
            line = fp.readline()

    fp.close()
    return centroids


def euclDistance(x, y):
    distance = sqrt(sum([(a - b) ** 2 for a, b in zip(x, y)]))
    return distance

# create clusters based on initial centroids
def createClusters(centroids, dataPoints):
    for cord in dataPoints:
        min_dist = 100000000000000
        index = -1

        for centroid in centroids:
            # euclidian distance from every point of dataset
            # to every centroid
            cur_dist = euclDistance(centroid, cord)

            # find the centroid which is closer to the point
            if cur_dist <= min_dist:
                min_dist = cur_dist
                index = centroids.index(centroid)

        var = "%s\t%s\t%s\t%s" % (index, cord[0], cord[1], cord[2])
        print var

if __name__ == "__main__":
    K = 4
    PLAYER_NAME = sys.argv[1] # stephen curry
    player_shots_record = get_data(PLAYER_NAME)
    LOCATIONS = list(PLAYER_SHOT_RECORDS.keys())
    centroids = getCentroids('centroids.txt')
    if not centroids:
        centroids = sample(LOCATIONS, K)
    createClusters(centroids, LOCATIONS)
```

Figure 12: The mapper of the comfort zone question

```
#!/usr/bin/python
"""reducer.py"""

import sys

def updateCentroids():

    # create the cluster information dictionary
    dict_centroids_points = {}
    for i in range(4):
        dict_centroids_points[i] = []

    # input comes from STDIN
    for line in sys.stdin:
        # parse the input of mapper.py
        index, SHOT_DIST, CLOSE_DEF_DIST, SHOT_CLOCK = line.split('\t')
        # convert x and y (currently a string) to float
        try:
            SHOT_DIST = float(SHOT_DIST)
            CLOSE_DEF_DIST = float(CLOSE_DEF_DIST)
            SHOT_CLOCK = float(SHOT_CLOCK)
        except ValueError:
            continue
        dict_centroids_points[index].append((SHOT_DIST, CLOSE_DEF_DIST, SHOT_CLOCK))

    # this lambda function is used to get the new "central point" of input points
    meanTupleList = lambda t : (sum([a[0] for a in t])/len(t), sum([a[1] for a in t])/len(t), sum([a[2] for a in t])/len(t))
    for j in range(4):
        print meanTupleList(dict_centroids_points[j])


if __name__ == "__main__":
    updateCentroids()
```

Figure 13: The reducer of the comfort zone question

- James Harden:
  (4.123795180722891, 2.7102409638554197, 17.31837349397591) 0.5701219512195121
- Chris Paul:
  (16.38548387096775, 4.5064516129032235, 16.041532258064517) 0.5474452554744526
- Stephen Curry:
  (5.913793103448278, 3.3577586206896552, 17.859051724137945) 0.6277056277056277
- Lebron James:
  (5.118215613382902, 4.0271375464684, 18.625650557620816) 0.6617100371747212

Figure 14: Comfort Zones of 4 NBA players

```
>>>>>>> round 10 <<<<<<<
Current updated centroids:
(5.594956140350877, 2.686622807017541, 13.361403508771918) (23.36777777777778, 5.6272222222222
24, 15.676111111111124) (22.06079136690649, 4.639928057553957, 5.729856115107916) (5.60625, 13
.603125000000002, 22.178125)

>>>>>>> round 11 <<<<<<<
Current updated centroids:
(5.5975824175824185, 2.6802197802197774, 13.344395604395594) (23.36777777777778, 5.62722222222
2224, 15.676111111111124) (22.06079136690649, 4.639928057553957, 5.729856115107916) (5.5696969
69696971, 13.360606060606063, 22.145454545454548)

>>>>>>> round 12 <<<<<<<
Current updated centroids:
(5.609713024282561, 2.670640176600439, 13.30728476821191) (23.36777777777778, 5.62722222222222
4, 15.676111111111124) (22.06079136690649, 4.639928057553957, 5.729856115107916) (5.4142857142
857155, 12.874285714285717, 22.122857142857143)

>>>>>>> round 13 <<<<<<<
final zone and correspondent hit rate for <<<lebron james>>>:
(5.4142857142857155, 12.874285714285717, 22.122857142857143) 0.9142857142857143
Okay now we all know where to stop this guy!
```

Figure 15: Comfort Zones of 4 NBA players