Ira Throne

UID: 862217380

email: ithro001@ucr.edu

CS 236

06/11/2023

Project Report

Since I couldn't find any groupmates, I did the project by myself. Therefore, my contribution is 100%. The project was executed using Hadoop, Python 3.10.6, MRJob library, and some bash scripting.

I decided to break the problem into 4 MapReduce jobs: one mapper, one combiner, and two reducers.

The mapper's job is to extract the data from the source files in chunks and process those chunks in parallel. Each record is extracted and transformed in some way and then sent further as a key-value pair. In this case, the mapper's job is to extract the data, compute the amount of money made in each month from each reservation, and then pass the result in the form of (month/year) as the key and the money made from that date form the reservation as the value.

To extract the data, I use .split(',') command, which extracts comma-separated values from each record into a separate list. In order to further process the data, there are some issues that need to be fixed:

1. there are two datasets with different schemas
2. some reservations were canceled and therefore did not bring any profit

3. some reservations were taking place in two (or maybe more) months. For example, if the move-in date is June 27, and the guest stayed for seven days, we have to split the reservation into two parts: four days in June and three days in July.

To handle the first issue, I wrote up two different cases. Since the two datasets have different numbers of columns, I decided that the easiest way to figure out which is what would be to count the number of elements in the list that resulted from splitting the entry by the comma:

```
# if in hotel-booking file
if len(cells) == 13:
    # checks for cancellation
    if cells[1] == '0':
    # extracts data into variables
        day = int(cells[6])
        month_name = cells[4]
        month = int(month_map.get(month_name))
        year = int(cells[3])
        stay_length = int(cells[7]) + int(cells[8])
        avg_price = float(cells[11])
```

```
# if customer-reservations file
elif len(cells) == 10:
    # checks for cancellation
    if cells[9] == 'Not_Canceled':
        # extracts data into variables
        day = int(cells[6])
        month = int(cells[5])
        year = int(cells[4])
        stay_length = int(cells[1]) + int(cells[2])
        avg_price = float(cells[8])
```

Once the dataset has been determined, the first thing that needs to be done is to check whether the reservation was canceled. If it was canceled, there would be no need to waste resources and assign variables for it. To implement that, I introduce a Boolean variable called canceled, which is by default set to False. However, if there is an indication that the reservation was canceled (the second row equals 1 in the hotel-booking database or the tenth row equals

‘Cancelled” in the customer-reservations database), the variable switches to True and prevents the program from further executing the entry.

Another thing worth pointing out is that the hotel-booking dataset has months written in words, whereas the customer-reservations database has them in numeric format. For the sake of uniformity and ease of calculation, I convert the months from the hotel-booking dataset into the numeric format as well. To do that, I mapped every month to a number, and then when the program runs, it matches the name to its number using the .get() command:

```
month_map = {
    'January': 1,
    'February': 2,
    'March': 3,
    'April': 4,
    'May': 5,
    'June': 6,
    'July': 7,
    'August': 8,
    'September': 9,
    'October': 10,
    'November': 11,
    'December': 12
}
```

All useful variables then need to be cast into numeric format (float for the average price, integer for all others).

Now, to the main processing part. If the reservation was not canceled, we have to determine how many days are in the reservation month to see if the reservation was split between two (or maybe more) months. First, we check for months that have 31 days (there are seven of them); then we check for 30-day months (there are four of them); and then we get to February, check for non-leap year case, and if none of those cases apply, it’s February of a leap year. Since we are going to use subtraction, we have to add 1 to the number of days to make the math work (for example, January has 31 days; if someone checked in on January 1 and stayed there exactly

the whole month, we need 32 to subtract from 1 to get the maximum length of stay that would fit in January alone). Therefore, if the length of stay plus the move-in day is greater or equal to that number, the entry has to be split in (at least) two to accommodate for the part that goes to another month. If that happens, we need to find the number of days of the stay that count for the current month and then roll the rest over to the next month. We need to reset the day to 1, increment the month, and if the month was December, we need to increment the year and reset to January:

```
# if the reservation length rolls over to next month
if (day + stay_length) >= max_days_plus_one:
    # count how many days of the reservation are in current month
    days_in_month = max_days_plus_one - day
    total_price = days_in_month * avg_price
    # update the stay length to not account for the month that we already printed
    stay_length = stay_length - days_in_month
    # print a separate entry for current month
    form_date = str(month)+ '/' + str(year)
    yield form_date, total_price

    # reset the day, increment the month
    day = 1
    month += 1
    # if the month was December, increment year, reset month to January
    if month > 12:
        month = 1
        year += 1
```

If the whole reservation fits in one month, we just proceed to the last step of the mapper, which is formatting the date and calculating the total from that reservation. Once that is done, we pass the date as the key and the total as the value.

Once processed in the mapper, the data goes to the combiner:

```
# pre-combines entries with the same date
# makes reducer more efficient
def combiner(self, date, totals):
    yield date, sum(totals)
```

The combiner is designed to take input from the mapper and aggregate it before passing it to the reducer. In my particular case, I didn't notice much of a difference since I was using a pseudo-distributed mode on a single computer, but I'd imagine that the combiner really shines in truly distributed systems.

The first reducer takes input from the combiner, adds up totals for each month/year combination, and sends the result to another reducer for sorting. It doesn't send a key because the other reducer is going to sort the results by totals.

```
# sums up totals for each date
# doesn't yield a key, but yields value with the totals as the first column
def sum_reducer(self, date, totals):
    yield None, (sum(totals), date)
```

Once the first reducer has done its job, the second reducer sorts the input in reverse order with respect to revenue totals. It then yields the date and the total rounded by two decimal places.

```
# takes input form the first reducer, sorts it in reverse order by total
# yields date as key, total (rounded to two decimals) as value
def sort_reducer(self, date, totals):
    for total, date in sorted(totals, reverse=True):
        yield date, round(total,2)
```

This program is then wrapped into a bash script that prompts the user to input the location of the input files, the location for the output, and runs hadoop with those parameters as variables:

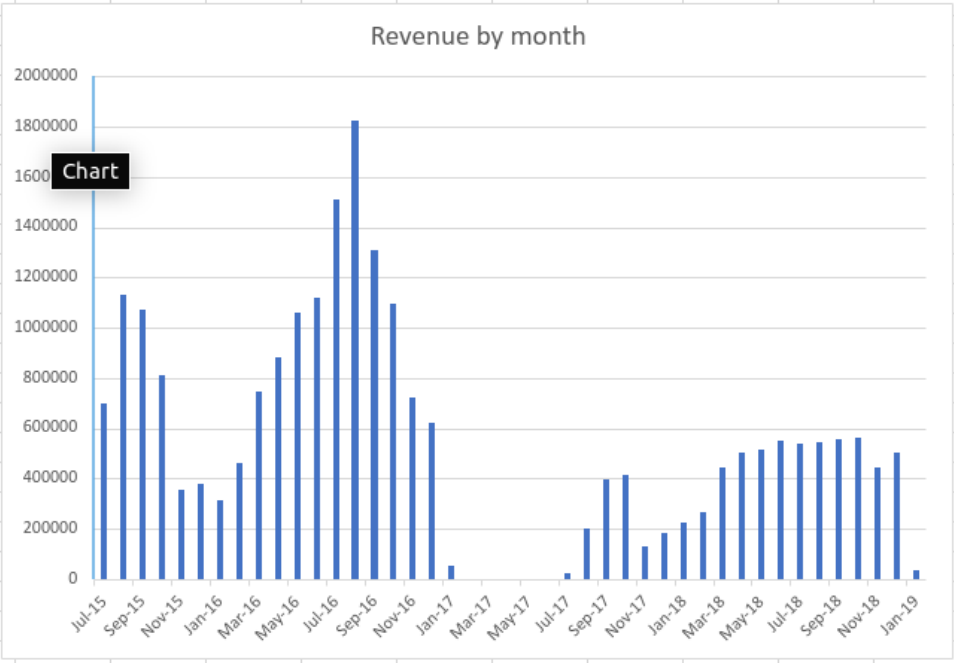
```
# Taking input for input and putput folders
read -p "Input the location of the customer-reservations.csv file: " cust_res_dir
read -p "Input the location of the hotel-booking.csv file: " hot_book_dir
read -p "Input the path to the output directory: " output_dir

# Executing the mapreduce
python3 mrjob_script.py -r hadoop "${cust_res_dir}/customer-reservations.csv"
"${hot_book_dir}/hotel-booking.csv" -output="${output_dir}"
```

The result is this text file that contains 38 month/year combinations sorted by the revenue from the largest to the smallest:

```
 1 "8/2016"        1822416.64
 2 "7/2016"        1512810.62
 3 "9/2016"        1309432.06
 4 "8/2015"        1129153.72
 5 "6/2016"        1122601.3
 6 "10/2016"       1094761.78
 7 "9/2015"        1075243.53
 8 "5/2016"        1058766.78
 9 "4/2016"        885728.58
10 "10/2015"       813698.33
11 "3/2016"        748241.12
12 "11/2016"       724951.13
13 "7/2015"        698419.85
14 "12/2016"       622785.25
15 "10/2018"       565639.63
16 "9/2018"        558188.48
17 "6/2018"        549671.88
18 "8/2018"        544999.3
19 "7/2018"        538957.35
20 "5/2018"        517699.98
21 "12/2018"       505522.13
22 "4/2018"        502916.85
23 "2/2016"        461773.48
24 "11/2018"       446645.4
25 "3/2018"        444640.46
26 "10/2017"       416008.51
27 "9/2017"        397942.2
28 "12/2015"       376706.95
29 "11/2015"       356657.99
30 "1/2016"        314770.39
31 "2/2018"        263750.94
32 "1/2018"        227903.78
33 "8/2017"        199818.87
34 "12/2017"       182490.66
35 "11/2017"       128265.25
36 "1/2017"        56141.04
37 "1/2019"        35643.8
38 "7/2017"        24399.2
```

Here is the graph with the month/year combinations and revenues plotted:



As we can see, the second dataset doesn't start until July 2017, thus there is a gap. The revenues from Jan 2017 and Jan 2019 are actually roll-overs from December of the previous year.

The main problem I had with this project was trying to run datanode after the first run of hadoop. First, I was not allowed to rewrite the existing output files, so I formatted the namenode. After the formatting, neither by start-all.sh nor hadoop datanode –start would start the datanode. It took me awhile to find a way to find out that I have to remove temporary data folders (dfsdata and tmpdata), then format the namenode, and then restart hadoop. That deletes all files from the HDFS, including the previously uploaded input files, so it's necessary to re-upload them.

Sources Used:

Apache Software Foundation. “Apache Hadoop 3.3.5 – MapReduce Tutorial.” *Hadoop.apache.org*, hadoop.apache.org/docs/stable/hadoop-mapreduce-client/hadoop-mapreduce-client-core/MapReduceTutorial.html.

Barakzai, Mohammad Rafi. “Lesson 8 MapReduce with Python Wordcount Program.” *Www.youtube.com*, 24 Aug. 2020, www.youtube.com/watch?v=mhisPejz-6c&list=PLylBsOS1de9igxKhZJNpB32sxLTKY43_j&index=14.

---. “Lesson 9 Hadoop Streaming and Options.” *Www.youtube.com*, 24 Aug. 2020, www.youtube.com/watch?v=9CW8rD_MF_o&list=PLylBsOS1de9igxKhZJNpB32sxLTKY43_j&index=13.

---. “Lesson 11 Python MRJob Installing and Testing.” *Www.youtube.com*, 24 Aug. 2020, www.youtube.com/watch?v=zp1w8iM0UR4&list=PLylBsOS1de9igxKhZJNpB32sxLTKY43_j&index=11.

Brandeis University. “Hadoop Troubleshooting.” *Www.cs.brandeis.edu*, www.cs.brandeis.edu/~rshaull/cs147a-fall-2008/hadoop-troubleshooting/.

NeuralNine. “MapReduce Jobs for Distributed Hadoop Clusters in Python.” *Www.youtube.com*, 12 May 2023, www.youtube.com/watch?v=NAGu_oBCtwI&t=903s.

Python Software Foundation. “9. Classes — Python 3.10 Documentation.” *Docs.python.org*, docs.python.org/3.10/tutorial/classes.html.

Yelp, and Contributors. “Mrjob — Mrjob V0.7.4 Documentation.” *Mrjob.readthedocs.io*, mrjob.readthedocs.io/en/latest/.