CS 6240: Project Final Report

Explore Digital Footprints of London's Bike-sharing System

Team Members

Name	Email
Chuan-Ti Chen	chen.chuan@northeastern.edu
Chia-Hui Lin	lin.chiahu@northeastern.edu

Problem Statement

City of London introduced a bike-sharing program in 2010, which now spreads across 40 square miles (100 square kilometers) of London, and operates over 11,500 bikes and 800+ stations, thus making it the second largest bike-sharing system in Europe and a global top ten. With the rise of Internet of Things (IoT), bike-sharing systems can now track bike rentals and their digital footprints that provide information on what bike is picked up from which station, returned to which station, by whom, when, and for how long. The collection of digital footprints, if used correctly, can generate valuable insights on the pattern of urban commute in London which not only benefits users of the bike-sharing system but also facilitates the government to make decisions on how to maintain and improve the system.

Introduction to Project Goals

A bike rental record contains a start and end station, which can be easily associated with an edge (u, v). The data set we found contains millions of bike records which is essentially a large and complex directed graph complemented with time-series data (the start and end time of each rental). Hence, the goal of our project is to gather interesting patterns or insights from these records, and there are two major problems that we believe would be challenging and rewarding to work on.

Problem I: Top K Pairs

- Analysis task: Since it is important to ensure a consistent supply of bikes available for rental and empty docks for bike returns at each station, we want to find out which (start station, end station) pairs are the most frequently visited every month in the year of 2019.
- Major task: Identify the top K pairs of stations where bikes are picked up and returned every month in the year of 2019.

Problem II: Average Speed

- Analysis task: We want to provide insights on the average speed of commute between all pairs of stations by calculating the average speed of bike rides from and to these stations in peak hours.
- Major task: Calculate the average speed of commute by bike using HBase and MapReduce, and compare the performance and scalability with using plain MapReduce.
- Using HBase & MapReduce:
    - Helper task 1: Calculate the distance between every two stations from the longitude and latitude data provided in the dataset and store them in an auxiliary file.
    - Helper task 2: Populate a HTable with duration data.
    - Helper task 3: Compute the average duration of each pair in peak hours, and store the output in an auxiliary file.

    - Helper task 4: Use reduce-side join to join the distance and average duration data on common keys, and calculate the average speed.
- Using plain MapReduce:
    - Helper task 1: Same as Helper task 1 using HBase & MapReduce.
    - Helper task 2: Read duration data from bike usage dataset, filter by peak hours, calculate the average duration between each pair of start and end stations, and store the output in an auxiliary file.
    - Helper task 3: Same as Helper task 4 using HBase & MapReduce.

Dataset Description

We are going to work with Bike-Share Usage in London Network data set which was obtained from Kaggle(https://www.kaggle.com/ajohrn/bikeshare-usage-in-london-and-taipei-network?select=london.csv). There are two data sets - bike usage data in London and London stations data. The London stations data contains more than 800 different station's names and their geographical location (longitude and latitude). On the other hand, the bike usage data contains over 38 million usage records. Each record covers duration time, pickup location, return location, and timestamp precise to minutes. The data was collected from the end of 2016 to July 2020 and reflects the distribution of all usage records received during this period.

The London stations data set contains the unique station id followed by station name, longitude, and latitude.

- station_id range from 1 to 839 sequentially
- station_name has unique 802 values
- longitude range from -0.24 to 0
- latitude range from 51.3 to 51.5

The bike usage data set contains the unique rental_id followed by these headers:

- rental_id
- duration (seconds)
- bike_id
- start_rental_date_time (yyyy-mm-dd HH:mm:ss)
- start_station_id
- start_station_name
- End_rental_date_time (yyyy-mm-dd HH:mm:ss)
- end_station_id
- end_station_name

London stations data excerpt:

station_id	station_name	longitude	latitude
1	River Street, Clerkenwell	-0.109971	51.5292
2	Phillimore Gardens, Kensington	-0.197574	51.4996
3	Christopher Street, Liverpool Street	-0.0846057	51.5213

Bike usage data excerpt:

rental_id	duration	bike_id	end_rental_date_time	end_station_id	end_station_name	start_rental_date_time	start_station_id	start_station_name
61343322	60.0	12871.0	2016-12-28 00:01:00	660.0	West Kensington Station, West Kensington	2016-12-28 00:00:00	633	Vereker Road North, West Kensington

Technical Discussion

Problem I: Top K Pairs

This problem is essentially a pair frequency-counting task combined with reducing the results to Top K. The techniques that we plan on using to optimize the performance of the frequency counting part are as follows:

- Tally counts per map task: To reduce the number of records emitted to the reducer, we will aggregate the count of each pair in each map task. In order to leverage the advantage of range partitioning, the output key format is designed to be (month, start_station, end_station)
- Range partitioning: For each reduce task to only process the records of the same month, we will utilize range partitioning. To be more specific, the records will be partitioned by month, and 12 reduce jobs will be spawned to calculate the top K pairs in each month.

As for how we are going to calculate the top K, we decided to use heap instead of secondary sort. Reasons for this decision are discussed in Major Problems section in this report.

- Use heap/tree map in reducer: in each reduce task, we will create a min-heap of size K; for each record, if the count is less than the top element in the min-heap, we will pop the top element and add the new record to the heap.

Pseudo Code

- CustomMapper: Tally count of each pair per map task

```
setup():
        counter = new HashMap()
map(inKey, inValue):
        for each line:
                counterKey = month + "," startStationId + ","  + endStationId
                counter[counterKey] += 1
cleanup():
        for each (key, count) in counter:
                emit(month, {startStationId, endStationId, count})
```

- RangePartitioner: Partition each record by month

```
getPartition(key, value, numReduceTasks):
        return (month-1) % numReduceTasks
```

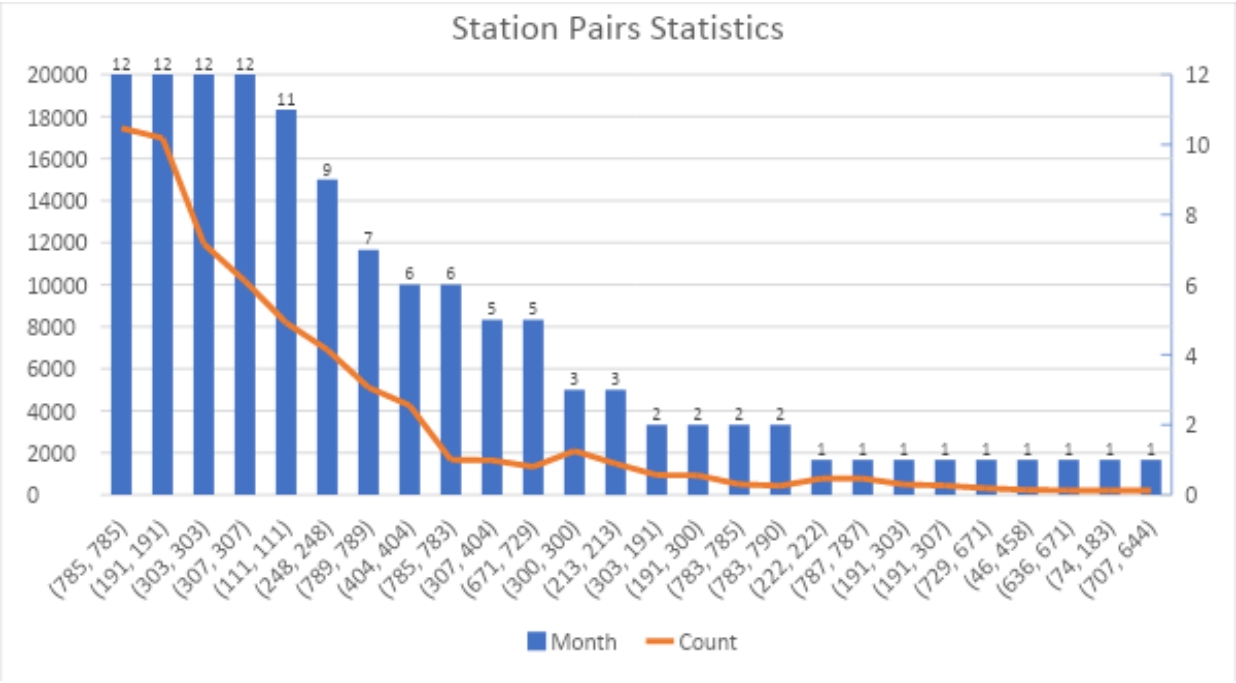
- TopKReducer: Aggregates the intermediary count of each pair from mapper, use min heap to find top K count

```
reduce(key, values):
        minHeap = new PriorityQueue()
        counter = new HashMap()
        // aggregate again since one mapper can't read and aggregate all entries in the same month
        for each value in values:
                counterKey = value.getStartStationId() + ","  + value.getEndStationId()
                counter[counterKey] += value.getCount()
        // populate minHeap
        for each key in counter.keySet():
                startStationId , endStationId = key.split(",")
                count = counter[key]
                pair = new Pair(startStationId, endStationId, count);
                if  minHeap.size() < K:
                        minHeap.add(pair)
                else if minHeap.peek().getCount() < count:
                        minHeap.poll()
                        minHeap.add(pair)
        // get top K
```

```
topK = new Pair[K]
i = K-1
while minHeap.size() > 0:
        topK[i--] = minHeap.poll()
// emit results
for each pair in topK
        emit({month, startStationId, endStationId}, count)
```

Results

Due to limited space in the report, we will show a summary statistic of the top K pairs instead of the actual computed results. The complete results of each month can be found in 12 output files submitted along with this report. The summary statistic below shows the number of occurrences of the top K pairs in 2019 (e.g. month = 9 denotes that this pair has made into the top K pairs 9 out of 12 months in 2019) as well as the total counts in 2019 (e.g. Count = 17450 denotes that this pair has a total count of 17450 in 2019):



One of the interesting patterns we have observed from the statistics is that the start and end station pairs that topped the charts have the same ids, which means the majority of users of London's bike sharing system chose to rent and return the bikes to the same station.

Performance & Runtime

We believe our version of TopKPairs program scales reasonably well as using 10 workers almost sped up the runtime twice than the runtime when using 5 workers. A naive version would be without in-mapper aggregation, and the aggregation will be performed in the combiner and reducer. Our version not only greatly decreases the number of records emitted from the mapper, since it continually aggregates the data in the input block processed by a map task (i.e. as soon as it receives two values with the same key it combines them and stores the key-value pair in a HashMap); but also it guarantees local aggregation will be executed because it is applied inside the code, and it provides more control for the users as in where the aggregation happens, whereas the execution of default combiner is not guaranteed by MapReduce and we have no control over when and where the aggregation should take place.

Machines/Runtime	Start time	End time	Runtime
6 instances of m4.xlarge	02:04:13	02:06:42	149s
11 instances of m4.xlarge	02:11:39	02:13:05	86s

Problem II: Average Speed

To tackle this problem we will perform an equi-join between the data of distance and the data of average duration stored in auxiliary files to calculate the average speed of commute between every two stations. Below are the steps broken down for each task:

- Pre-process the distance between each pair of the locations across all permutations of two stations: The results in an auxiliary file. The definition of distance is shown below, where φ is latitude, λ is longitude, R is earth's radius (mean radius = 6,371km). We have found a snippet of code that's equivalent to Haversine formula on StackOverflow and the source can be found in citations:

  Haversine formula: $a = sin^2(\Delta\varphi/2) + cos\,\varphi 1 \cdot cos\ \phi 2 \cdot sin^2(\Delta\ \lambda/2)$

  $c = 2 \cdot atan2(\sqrt{a}, \sqrt{(1-a)})$

  $d = R \cdot c$
- HBasePopulate - Populate HTable with duration data: We will store the startStationId, endStationdId as well as duration of commute retrieved from the bike usage dataset, using the start time of bike rental record as index prefix for efficient range querying.
- HBaseCompute - Compute the average duration of each pair from the bike usage data: Query the duration data from HTable that are within the peak hours, and calculate the average duration between each pair of start and end stations, and store the output in an auxiliary file.
- Perform equi-join (in the reducer) on the distance data and the average duration data stored in auxiliary files.

The plain MapReduce program is trivial and the steps are generally the same with those described above except for HPopulate and HCompute.

<u>Pseudo Code</u>

HBase

1. HBasePopulate
   - HPopulateMapper

     map():

     Parse each line to get BikeData object.

     Create rowKey:

     startTime(format: "hh:mm:ss"), system time nanoseconds

     Create rowData:

     startTime + startStationId + endStationId + duration

     Populate the Table with rowKey and rowData
2. HBaseCompute (Compute average duration)
   - HComputeMapper

     setup():

     Initialize hashmap counter to store key value pairs

counter = new HashMap()
Key: (startStationId, endStationId)
Value: list of durations

map():
Create scanner with start row:
startTime 16:00:00
and end row:
endTime 19:00:01
Each row from the table, parse the row to get counterKey and duration
counterKey = "(" + startStationId + "," + endStationId + ")"
counter[counterKey].add(duration)

cleanup():
for each (key, list(duration)) in counter:
emit({startStationId, endStationId}, list(duration))

- HComputeReducer

reduce():
float durationSum = 0
int durationCount = 0

for each duration list in values:
for each duration in duration list:
durationSum += duration
durationCount += 1
float avgDuration = durationSum / durationCount;
emit(key, avgDuration);

Plain MapReduce

1. AvgDuration
   - DurationMapper

setup():
START_TIME = "19:00:00", END_TIME = "19:00:00"
// counter = {(startStationId, endStationId): [d1, d2, d3...]}
counter = new HashMap()

map(inKey, inValue):
for each line:
if startRentalTime >= START_TIME && startRentalTime <= END_TIME:
counterKey = "(" + startStationId + "," + endStationId + ")"
counter[counterKey].add(duration)

cleanup():
for each (key, list(duration)) in counter:
emit({startStationId, endStationId}, list(duration))

   - DurationReducer

reduce(key, values):
durationSum = 0, durationCount = 0
for each list(duration) in values:
for each duration in list(duration):
durationSum += duration

durationCount += 1
avgDuration = durationSum / durationCount
emit({startStationId, endStationId}, avgDuration)

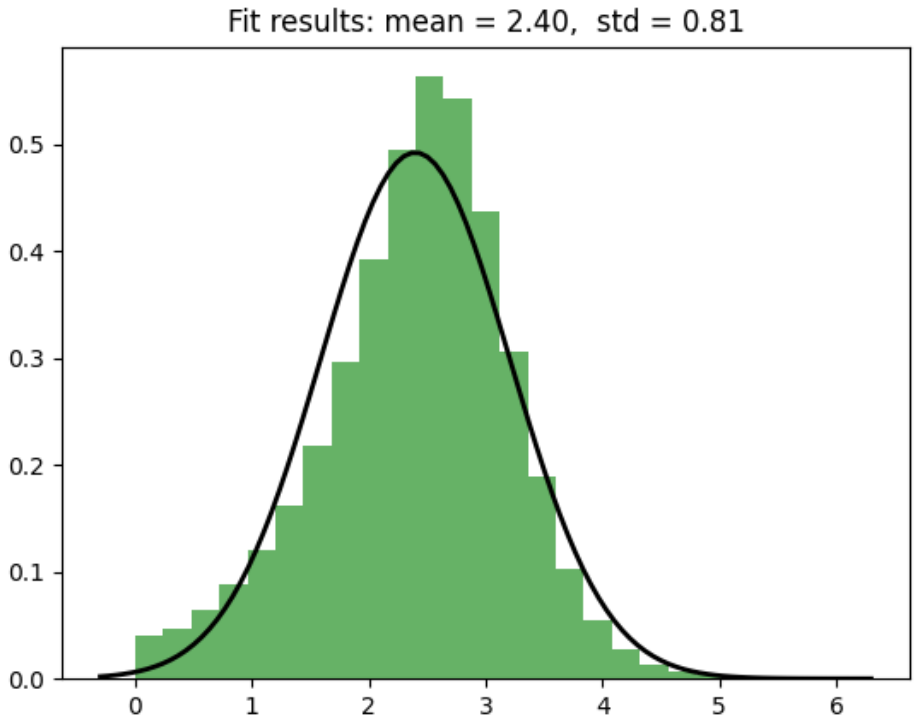
2. AvgSpeed
    - DurationMapper

      map(inKey, inValue):
        emit({startStationId, endStationId}, “0 ” + duration.toString())
    - DistanceMapper

      map(inKey, inValue):
        emit({startStationId, endStationId}, “1 ” + distance.toString())
    - JoinReducer

      reduce(key, values):
        duration = null, distance = null
        for each value in values:
          String[] data = value.toString().split(“ ”)
          if data[0] == “0”:
            duration = data[1]
          else:
            distance = data[1]
          if duration != null && distance != null:
            speed = Float.parseFloat(distance) / Float.parseFloat(duration)
            emit({startStationId, endStationId}, speed)
            duration = null
            distance = null

Results



Again, due to limited space in the report, we will only show a summary statistics of the average speed dataset. The complete results of computed speed data can be found in the output file submitted along with this report. After removing several outliers (data points > 30 meter/second) from the results, we generated a normal distribution graph. From this graph, we can see that the speed dataset has a mean of 2.40 a standard deviation of 0.81m/s, and about 80% of the data points fall in the range of 1.0m/s ~ 4.0m/s.

Performance & Runtime

Since the major difference between HBase + MapReduce v.s Pain MapReduce is how average duration is computed, we will compare the performance and runtime of this task.

Runtime

	HBase + MapReduce(HBaseCompute)	Plain MapReduce (AvgDuration)
6 instances	89s	138s
11 instances	80s	83s

No. of Mapper Output Records

HBase + MapReduce(HBaseCompute)	Plain MapReduce (AvgDuration)
906,474	4,636,039

Comparatively, the Plain MapReduce version scales better than HBaseCompute, since the Plain MapReduce program has a 30% boost in runtime when 5 more machines are used, whereas HBaseCompute only has 10%. We believe it is because the size of Plain MapReduce's input, which is the original bike usage data set, is so large that over 70 map tasks are spawned duration computation which can be well distributed over 5 - 10 workers in parallel, whereas the input of HBaseCompute is of much smaller size because it is only consisted of the filtered rows from the HTable, and hence only 3-4 map tasks were spawned, resulting in low utilization rate of the workers.

However, the performance of HBaseCompute clearly surpassed that of its counterpart. Not only are the runtimes of both 6 and 11 instances shorter, but also the number of mapper output records of HBaseCompute is only 20% of that of Plain MapReduce. This result is aligned with our expectations since in Plain MapReduce our program has to read the entire dataset which is sorted in chronological order, meaning rows with the same (startStationId, endStationId) within the peak hours on different days are scattered throughout the dataset, resulting in more mapper output records after in-mapper aggregation; whereas the rows queried from the HTable within the peak hours are much more centralized, and hence resulting in fewer mapper output records.

Major Problems

Problem I: Top K Pairs

- In the initial design of our program, we did not consider adding an additional count aggregation step in Reducer as we do in the final version. The reason being we believed a mapper could read all of the bike usage data of the same month in a year and thus the records emitted from the mapper accurately reflects the total count of each pair. However, we realized there could be edge cases that the data of the same month spans across two mappers or the block size of the default input split is simply not large enough to cover all that data, hence our original approach which only tally counts

in mapper would be insufficient. Since only in reducer can we guarantee the data of the same month would be present (with the help of range partitioner by month), we added another aggregation step in Reducer before calculating the top K. We have tested our program with and without the additional aggregation, the results proved our speculation correct as the total counts without the additional aggregation are lower than the ones where the additional aggregation is applied. To corroborate our speculation, we also tried commenting out the partitioner and reducer, and only let the mapper emit the partially aggregated counts. The results contain multiple entries of the same (startStationId, endStationId) pair with different counts, which again proves that a single mapper could not process all data of the same month.

- Initially, we proposed to compare the performance of using heap and using MapReduce's built-in sort to get the top K pairs. However, when we were running the built-in sort program, we encountered the same issues as using heap, which is that the block size of a mapper is not large enough to cover all data of the same month and hence generating partially aggregated mapper outputs. There are two workarounds for this issue: first, store the partially aggregated outputs in an auxiliary file, and create another MapReduce job to continue aggregate on the auxiliary file; second, add an additional aggregation step in reducer and use ArrayList's sort functions. However, it is evident that both of these workarounds will necessarily increase the running time and coding efforts. Hence, we did not proceed with this option and decided to go with using heap.

Problem II: Average Speed

- Originally, we wanted to calculate the average duration and join the average duration with distance in the same job. Theoretically it is achievable by prepending each value in a record with a "duration tag" or a "distance tag" so that we can tell the types of data apart in the reducer. Once we know how to separate the duration data and distance data in the reducer, we can simply 1) aggregate the duration, 2) calculate average duration, 3) calculate distance / duration. However, due to the limitation that a MapReduce job can only read multiple inputs from HDFS system, meaning that it can't read from an HTable and a file stored in HDFS system at the same time, our idea couldn't be implemented, so we had to compute the average duration and speed in two separate steps as explained in the technical discussion above.

Conclusions & Extensions

In this project we explored the digital footprints of London's bike-share network with focus on two major topics stated below.

Problem I: Top K Pairs

We completed a pair frequency-counting task and reduced the results to Top K using several MapReduce techniques we learned in class, such as in-mapper aggregation and range partitioning. The results have shown that most of the start and end station pairs that made into the top K have the same ids. This interesting pattern led us to a potential extension on this problem: since most bike renters choose to return the bikes to where they rent them from, we are interested to find out whether these renters changed their mind about renting bikes immediately after they picked up the bikes and eventually returned them without ever riding them, or they actually did ride the bikes and made around trip back to the start stations.

Problem II: Average Speed

We performed an equi-join on distance and duration data to calculate the speed of commute between every pair of start and end stations within peak hours using two approaches, and compared their scalability and performance. The major difference between the first and second approach is that in the first one, we populated duration data in a HTable so that we can take advantage of efficient range-query on sorted keys. One potential expansion we can work on for this problem is to get a more accurate measure of route length

(in meters) between one station and another, as currently the formula we used to calculate the distance is a variation of Euclidean distance factoring in the curvature of Earth. However, the real distance a user biked from one station to another will be much longer than that since they can't possibly always ride in a straight line - we must factor in the geographical information of roads and transportation in London, which can be potentially provided by the London's GIS Model.

Citations

- Data source: https://www.kaggle.com/ajohrn/bikeshare-usage-in-london-and-taipei-network
- Calculate distance between two locations in the format of longitude and longitude: https://stackoverflow.com/questions/27928/calculate-distance-between-two-latitude-longitude-points-haversine-formula/12600225#12600225

Source code & log files

All the source codes we wrote for this project and log files generated on AWS EMR are submitted in a zip file along with this report.