Big Data Analytics Programming
Assignment 3 -Taxi trip analysis Report
Ramaravind Kommiya Mothilal (r0605648)

1. Trip Length Distribution:

The distribution of trip length is determined by constructing a **histogram**. The range of values in each uniform interval of the histogram is determined by the **number of bins** (passed as an input argument) and the **maximum trip length** in the dataset. An additional argument is passed to find and remove outlier lengths (trips with speed > 200 km/hr) in both the implementations as the histogram bin width depends on the maximum trip length given the number of bins.

1.1. Description:

Scala is used for both the non-park and the spark implementations. For the non-spark implementation, the lengths of each trip is calculated on the fly using the **Elliptical Flat Earth distance formula** in the first pass of the data. In the next pass, each length is mapped to its corresponding bin and a counter is incremented. The same template of code is followed in the spark implementation. But the computed **lengths of trips** is **cached** *after* the *'max'* action so that when the *'saveAsTextFile'* action is called, the cached data is used instead of loading the file and doing all the operations once again.

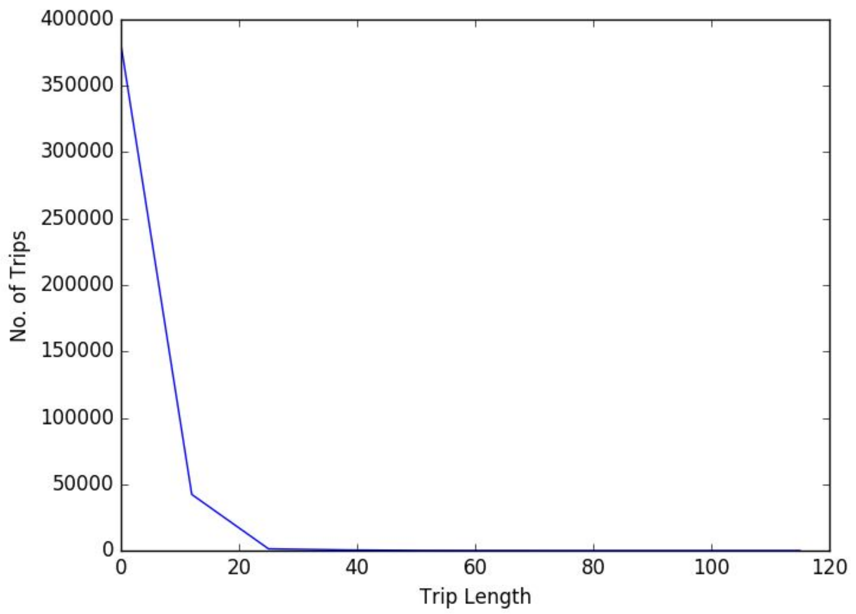
After removing outliers, the plot and tables below show that about **90% of trips are below 13 km and almost all trips are within 25 km**. But without removing outliers, the distribution does not provide much information as is dominated by the maximum distance, which is around 1000km.

1.2. Efficiency:

For the spark implementation, as the output data is very small, the **number of partition** is set to **1** to reduce the read/write overhead. The total time taken is as follows:
Non-spark : ~2.8 s **Spark :** ~7 s

As expected, the non-spark implementation is somewhat faster than the single node spark execution as there is **no overhead to organize the data for shuffle** (since the shuffle operation by spark is expensive as it involves disk I/O and data serialization) given the small size of the dataset and a single node for implementation. However, in case of large files, the non-spark implementation is limited by disk I/O bottleneck as it can run on only 1 node in which case the spark version will be much faster because of parallelization.

Length Distribution Plot - outliers removed



Length Distribution Frequency table (No of Bins = 10)

Trip Length	Frequency
0.0	381302
12.886314	42297
25.772629	1322
38.658943	478
51.545258	99
64.43157	23
77.31789	3
90.2042	3
103.090515	2
115.97683	4

(i) outliers removed

Trip Length	Frequency
0.0	441925
9755.997	8

(i) with outliers

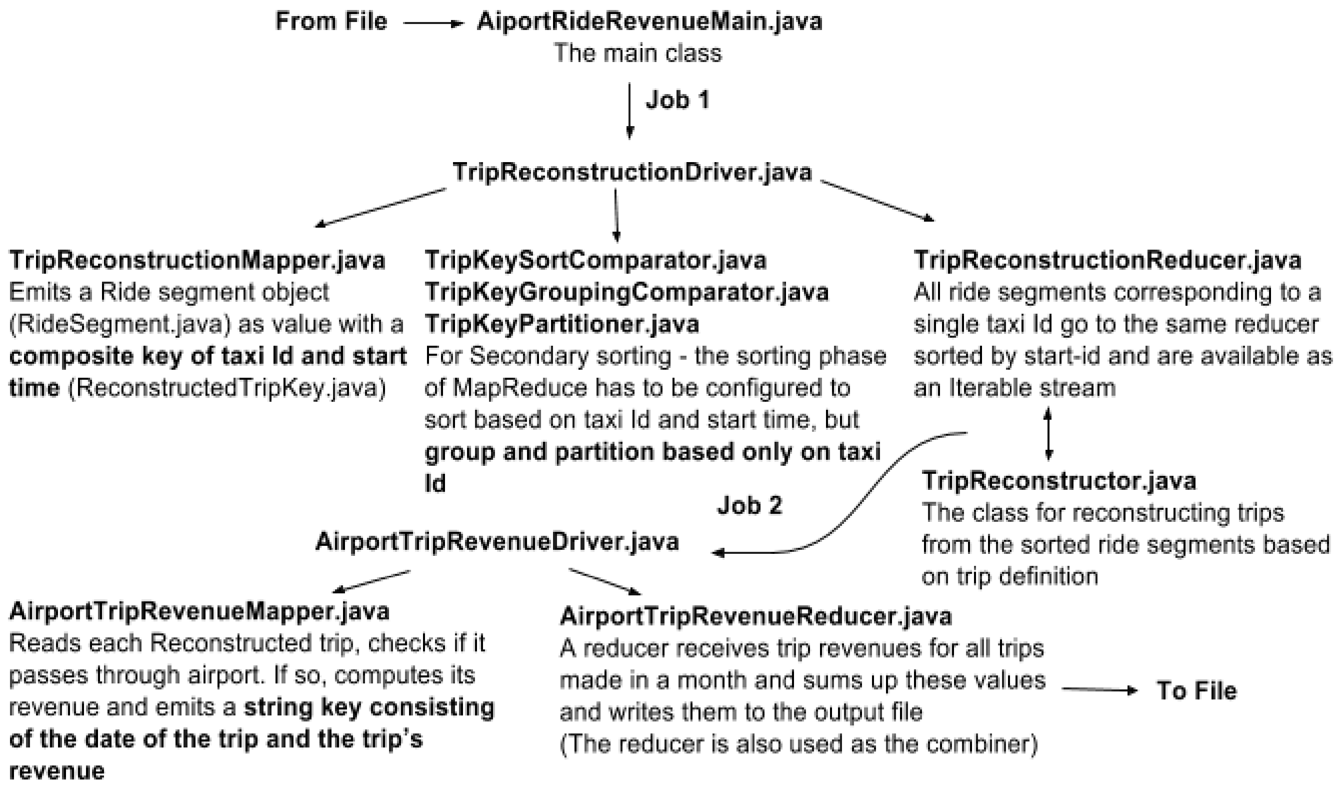
2. Computing Airport Ride Revenue:

The task consist of two parts - to reconstruct trips from the ride segments and compute the revenue based on distance of those trips that passes through the airport. As the final goal is to compute revenue, it is of interest to consider only **those ride segments which are accountable, that is when at least one of the start state or end state is full.** Further since the input data is more likely to be unordered, it is wise to **sort** the ride segments **first by taxiID and then by start time** as any trip is characterised by a set of *temporally overlapping ride segments with a common taxiID*. Once the segments are sorted, a ride segment can be considered as a part of a trip if

- It has the same taxiID as the previous segment (or)
- It's start time is equal to the end time of the previous segment (or)
- It overlaps temporally with the previous segment.

Also a new trip begins if the start state is empty but end state is full for the current ride segment or if the start state is full but the end state is empty for the previous ride segment of the running trip. While the three conditions mentioned before need to be followed strictly in order for a segments to be a part of a trip, the last one based on taxi states need not be strictly followed considering the fact that ride segments are prone to errors. Hence it is taken as a **weak condition** along with the aforementioned conditions based on taxiID and time.

2.1 Outline of the process:



From File → **AiportRideRevenueMain.java**
The main class

Job 1

TripReconstructionDriver.java

TripReconstructionMapper.java
Emits a Ride segment object (RideSegment.java) as value with a **composite key of taxi Id and start time** (ReconstructedTripKey.java)

TripKeySortComparator.java
TripKeyGroupingComparator.java
TripKeyPartitioner.java
For Secondary sorting - the sorting phase of MapReduce has to be configured to sort based on taxi Id and start time, but **group and partition based only on taxi Id**

TripReconstructionReducer.java
All ride segments corresponding to a single taxi Id go to the same reducer sorted by start-id and are available as an Iterable stream

TripReconstructor.java
The class for reconstructing trips from the sorted ride segments based on trip definition

Job 2

AirportTripRevenueDriver.java

AirportTripRevenueMapper.java
Reads each Reconstructed trip, checks if it passes through airport. If so, computes its revenue and emits a **string key consisting of the date of the trip and the trip's revenue**

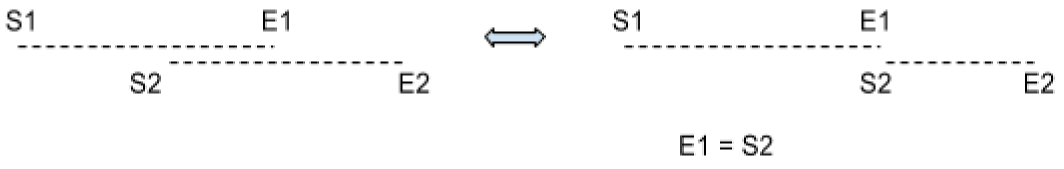
AirportTripRevenueReducer.java
A reducer receives trip revenues for all trips made in a month and sums up these values and writes them to the output file
(The reducer is also used as the combiner)

→ **To File**

2.2 Data Cleaning:

- **Average Speed:** Trips containing a single ride segment with **average speed > 200 km/hr** will distort the revenue computation and hence are deemed as invalid and are dropped from calculation.

- **Overlapping Segments:** When two segments overlap, the distance computation gets complicated as there are lots of ways to reconsider the distances in overlapping intervals, such as - If the current segment is lengthier by a larger factor than the previous segment, then the previous segment can be dropped or vice versa or, distance can be averaged over the overlapping intervals or, latest segment can transformed into a new segment with exactly the same parameters except **replacing its start time (S2) with end time (E1) of the overlapping previous segment.**
  Hence an argument is passed as input to consider the overlapping distance or ignore all trips with overlapping segments. For simplicity, the last mentioned method is used in the current implementation (if the flag is true) as it is difficult to chose any particular method without some kind of test data. Note that this method also have a *chance to escalate* the actual revenue!



- The segments are ordered by taxi ID and start time. While checking if the current segment overlaps temporally with the previous segment, if the previous segment (with respect to start time) is not actually the **previous segment with latest end time,** then the revenue computation will be distorted. Hence, while iterating through the segments, a variable (**latestEndTimeSegment in TripReconstructor.java**) is maintained to store the latest ride segment and the current segment is checked for temporal overlapping with both the previous segment (by start time) and the latest end time segment.

2.3 Number of Mappers and Reducers:

The number of mappers is determined by default by the hadoop framework depending on the input file size and split block size. Although it can't be set below that which Hadoop determines via splitting the input data, there's an option to increase the split size[1]. So after considering the given input file size and total running time of the task, it was decided **not to change the default split size** as the current setting itself gives desirable results.

Increasing the number of reducer beyond the number of nodes in the cluster (nine) did not significantly improve the overall performance as expected. Hence, the number of reducers for the trip reconstruction (TripReconstructionDriver.java) stage is set to **9** while the number of reducers for the airport ride revenue calculation in stage 2 (AirportTripRevenueDriver.java) is set to **1** because of the low number of keys (only 33 from 2008-05-01 - 2011-01-01).

2.4 Efficiency and scalability:

As the objective of the assignment is only to compute the revenue, the work of **checking if a trip is a valid airport trip** can be done during the trip reconstruction task itself. The reducer (TripReconstructionReducer.java) is appended with codes such that to **write only the airport trips** needed to compute the revenue over time. This vastly **reduces** the amount of data written in the trip reconstruction task and reduces the input to the revenue computation task. The runtime on the all.segments dataset is as follows: (time for reconstructing trips + time for revenue computation)

	Write all reconstructed trips	**Write only reconstructed airport trips**
Consider overlapping segments	~ 15 min + ~ 4 min	~11 min + ~ 20 sec
Ignore overlapping segments	~ 15 min + ~ 3 min	~ 9 min + ~ 20 sec

In all cases, the bulk of the time is taken by the trip reconstruction task. Clearly when only the reconstructed airport trip is written to file, the total time gets reduced from **~20 mins** to only **~10 mins**. The time for revenue computation is much lesser in the later case as almost all of the computation has taken place in the trip reconstruction stage itself. From the above runs it was observed that **minimizing writing file to disk** leads to a **significant decrease** in the time taken by mapreduce framework.

2.5 Final Results:

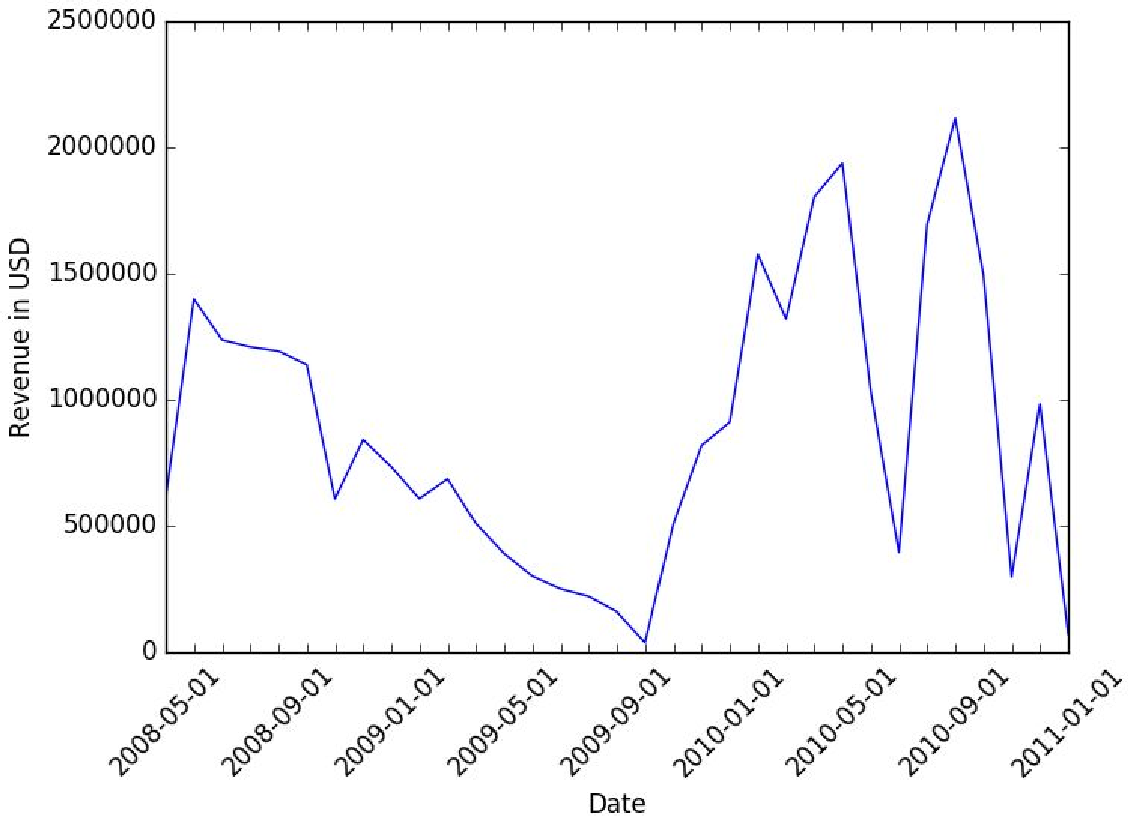
[1] https://wiki.apache.org/hadoop/HowManyMapsAndReduces

The total revenue by considering the overlapping segments is **29,093,411.69 USD** while by ignoring the overlapping segments from computation, it is **28,523,023.82 USD**. The revenue estimation has blown up by **570,388 USD** by taking the overlapping distance in account as explained in the Data Cleaning section.

The statistics for trip reconstruction and revenue computation

TripReconstructorMapper input records **(number of ride segments)**	306111264
TripReconstructorMapper output records **(number of segments considered for trip reconstruction)**	135778658
TripReconstructorReducer input groups **(total number of taxis)**	684
TripReconstructorReducer output records - write all trips **(number of reconstructed trips)**	11711145
TripReconstructorReducer output records - write only airport trips and *ignore overlapping segments* **(number of reconstructed airport trips)**	701120
TripReconstructorReducer output records - write only airport trips and *consider overlapping segments* **(number of reconstructed airport trips)**	738823

Revenue over time for all.segments dataset (considering overlapping segments)[2]



[2] The difference is not so obvious between the plot considering overlapping distances and the plot ignoring them.