Report Project 2: BDAP [B-KUL-H00Y4A]

Andreas Hinderyckx

April 2022

Part 1: Trip Length Distribution

The pure Java implementation was implemented by reading each line of the `.trips` file, calculating the distance between the start and end coordinates of that trip using a flat-surface formula[1] and writing the result to the output file. The Spark implementation reads the `.trips` file as a csv using the `SparkSession`, and creates an RDD of it using the `SparkContext`. Next, the distances are calculated by applying a `Map` transformation to the read data. Finally, the data is written to an output file.

The run time of the pure Java-implementation is 0.927 seconds, whereas the Spark implementation took 5.114 seconds. The pure Java-implementation is about 5 times as fast: this is as expected, as the Spark implementation introduces quite some overhead by setting up the `SparkContext`, converting the data to an RDD, dividing the RDD into logical partitions, creating the Directed Acyclic Graph (DAG) to schedule tasks and orchestration of worker nodes... [2] The performance gain achieved by parallelizing the task in-memory do not outweigh the introduced overhead; thus, the pure Java implementation is indeed faster for small files such as `2010_03.trips`.

The normalized histogram of the trip durations is shown in figure 1, where trips with a length ≥ 20 km are collected in the last bin. When seeking to minimize the sum of squared errors, the data is best approximated by the `sciPy` Folded Cauchy distribution[3], with parameters $\approx$ (c: 1.28, location: -4.58e-10, scale: 1.28), which achieves a sum of squared errors ≈ 0.037. These results are shown in figure 1

Part 2: Computing Airport Revenue

Identifying erroneous GPS points The first technique used to identify erroneous GPS points is to verify whether the calculated speed of a segment is smaller than 200 km/h. A lower speed limit was not chosen, as GPS results may be imprecise at times, which could cause valid but slightly inaccurate trips to be invalidated as well. Segments that don't contain 9 comma separated fields, or contain NULL-values are rejected as well, for example:
`2876,'2010-03-12 14:14:54',37.75116,-122.39468,'E',NULL,NULL,NULL,NULL`. Furthermore, it was apparent that quite some GPS points were located in the sea, to the West of San Francisco. One can reject these points either in the mapper, or in the reducer.

To implement the former, two coordinates were handpicked to create a rough approximation of the coastline. Based on the cross product's properties, points to the west of this coast line were rejected. This check detected roughly 1500 points, but did hurt performance, however (see below). Therefore, the latter option was chosen: points were rejected in the reducer by checking whether they formed an airport trip, and by checking the speed of each segment as mentioned. This approach rejects trips containing sudden large jumps to erroneous coordinates, and thus eliminates the majority of erroneous trips. Extra checks (e.g.verifying the elapsed time between two subsequent records, or verifying whether their coordinates properly chain up) were unnecessary as these detected segments are already invalidated by the previous checks. Other examples of erroneous segments, are those that take place during another segment's time-span:

```
1007,'2010-03-10 09:01:07',38.08841,-121.27446,'E','2010-03-10 09:03:07',38.08192,-121.2613,'M'
1007,'2010-03-10 09:02:07',38.08463,-121.26412,'M','2010-03-10 09:03:07',38.08192,-121.2613,'M'
```

and those that are part of a trip which doesn't end during the period within which the data was recorded. Segments pertaining to the first case are skipped in the reducer to prevent doubly counted distances. The corresponding trip is still counted as a valid trip. Trips belonging to the second case are detected by comparing the `TaxiID` to that of the previous segment. As we do not have access to the complete data of these trips, it was chosen not to count these trips as valid ones.

[1] https://en.wikipedia.org/wiki/Geographical_distance#Flat-surface_formulae
[2] https://www.ibm.com/cloud/learn/apache-spark#toc-how-apache-VZl4w8Yx
[3] https://docs.scipy.org/doc/scipy/reference/generated/scipy.stats.foldcauchy.html

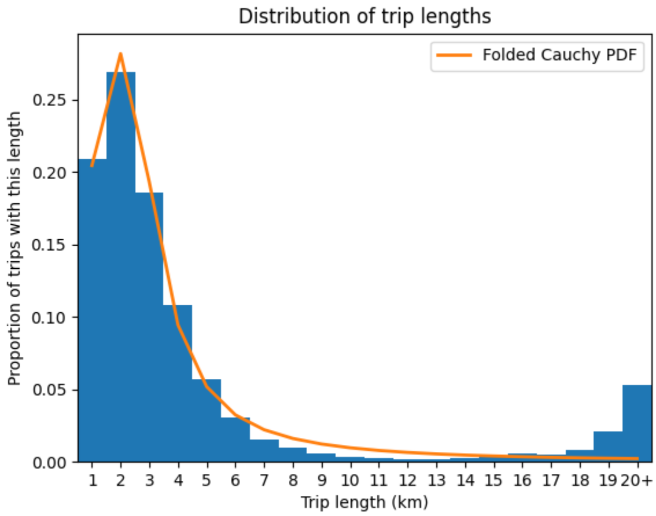

Figure 1: Distribution of trip lengths

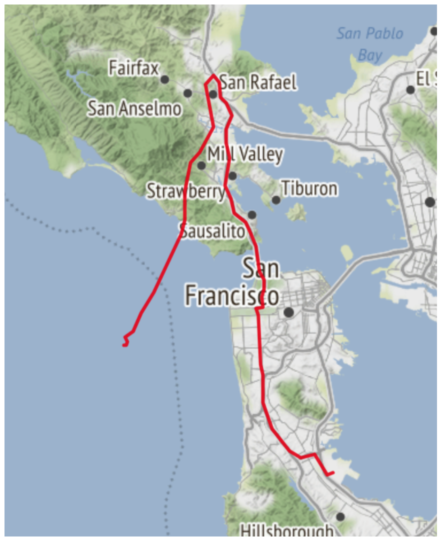

Figure 2: An airport trip (red line) that ends up in an erroneous location

Finally, there are also trips which still pass the checks mentioned above, but start in the airport and end up in an erroneous location (the ocean): figure 2 shows an example of such a trip. There are only three trips that satisfy these conditions, which together make up about 0.002% of the total revenue. Roughly speaking, at most 50% of the distance traveled in each of these trips is effectively erroneous (i.e. is located in the ocean). Thus, these edge cases account for less than 0.0009% of the total revenue, which is why it was decided to count these trips as valid airport trips as well. The alternative would be to eliminate these erroneous segments in the map-phase, but we don't opt for this approach as it would: (1) be subject to overfitting to the test data, as a precise line must be drawn which acts as a classifier for valid coordinates, and (2) incur a significant performance penalty, as these checks would need to be conducted on every segment.

Reconstructing the trips

NB: in what follows, a composite `key` or `value` will be respectively written as `{key}` or `{value}` to avoid confusion.

The trips are reconstructed in the first of the two map-reduce-processes. In the `mapper`, each segment is mapped onto a key-value pair with as key: `{TaxiID,StartDate}`. The custom `Comparator` implementation makes sure that the segments are primarily sorted on `TaxiID` and secondarily sorted on `StartDate`, when they are sorted by the MapReduce framework when transitioning from the `mapper` to the `reducer`. The `mapper` also rejects malformed records (see above) and doesn't map `E`-`E`-segments to limit network overhead, as these segments don't contribute to any trip. This already eliminates more than half of the segments. Additionally, a `Partitioner` is added to ensure that segments with the same `TaxiID` end up at the same reduce-task, and therefore end up at the same physical node to limit network overhead. Finally, a custom `GroupingComparator` implementation was added to specify which records end up in which `key, Iterable<value>` pair, received by the reducer.

Next, given the sorted order of the records, trips can be constructed during the reduce-phase. The reducer receives as input `{TaxiID,StartDate},Iterable<segment>` pairs. When iterating over an `Iterable<segment>`, the `reducer` looks for records that transition from the `E` state to the `M` state to initiate a trip. Subsequent `M`-`M`-records are added to the trip, until a record is encountered that transitions from the `M` state to the `E` state. During this process, the reducer rejects trips that don't satisfy the conditions explained above.

After the first map-reduce-job, a second one is chained which processes the output of the first. This job's mapper maps each record of the output of the previous job of the form `{StartDate ,StartCoordinates}, {EndCoordinates - TripRevenue}` to a `{Year-Month}, TripRevenue` key-value pair. This way, the reducer receives as input `{Year-Month},Iterator<TripRevenue>` pairs, which allows it to aggregate all trips' revenues which took place during the same month to obtain the final

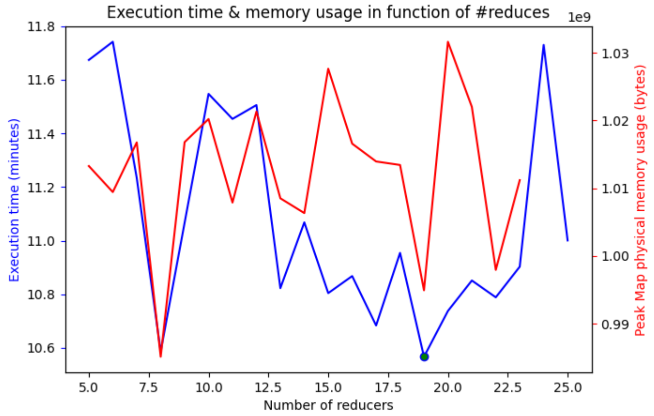

Figure 3: Execution time & physical memory usage in function of nb. of reducers

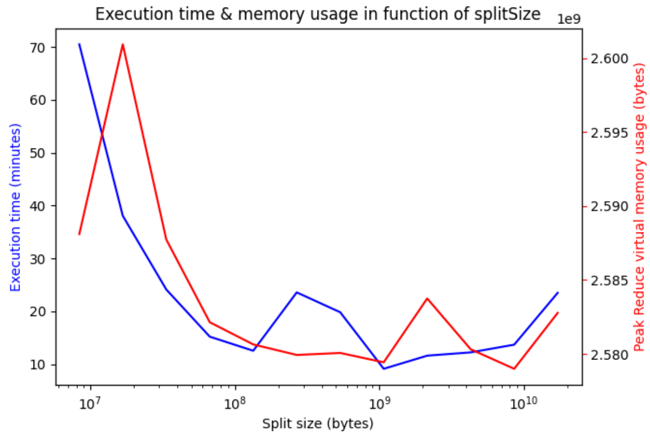

Figure 4: Execution time and virtual memory usage in function of split size

output containing the total monthly revenues.

Efficiency of the solution As the number of mappers can't be changed directly, one must adjust the `splitSize` instead. MapReduce determines the `splitSize` (which directly determines the number of Map tasks) based on the following formula [1]:

```
splitSize = Math.max(minSize, Math.min(maxSize, blockSize))
```

Thus, in order to increase the number of map tasks, one must decrease the `splitSize` by decreasing the `maxSize`. To decrease the number of map tasks, one must increase the `splitSize` by increasing the `minSize` beyond the default value of `blockSize` (128Mb). The results of these actions are depicted in figure 4, which clearly shows that extremely low or high `splitSize`s lead to higher execution times and memory usage (in this case: virtual memory). A balanced option is to take the `splitSize` around 1024Mb. This is substantially higher than the default block size (128Mb), but this choice yields better efficiency as the mapper is mainly IO-bound, rather than CPU-bound. Thus, the optimal number of maps tasks is $28Gb \times 1024\frac{Mb}{Gb} \div 1024\frac{Mb}{\text{map task}} = 28$ map tasks. The effect of the number of reducers is shown in figure 3. Here it is apparent that the number of reducers is chosen best to be slightly smaller than a multiple of 10 (which is the number of nodes in the cluster) to minimize execution time and memory usage. A possible explanation is that if a multiple of 10 is chosen and a reduce task fails, it has to wait until at least one of the other tasks is finished, whereas choosing a value slightly lower than 10 provides some slack in case of failure. Running the job with these tuned parameters (8 reducers, 28 mappers) results in a total execution time of 9.5 - 10.5 minutes, depending on the load on the cluster.

Total revenue & revenue over time As described above, the total revenue is aggregated per month by the map-reduce-job. These results are shown in figure 5. It is remarkable that there are months during which almost no revenue is made; this could be due to maintenance breaks of the taxis, or due to yearly leave of the taxi drivers... The approximate total revenue collected during the time-span of `all.segments` is equal to $ $23,852,063.65$.

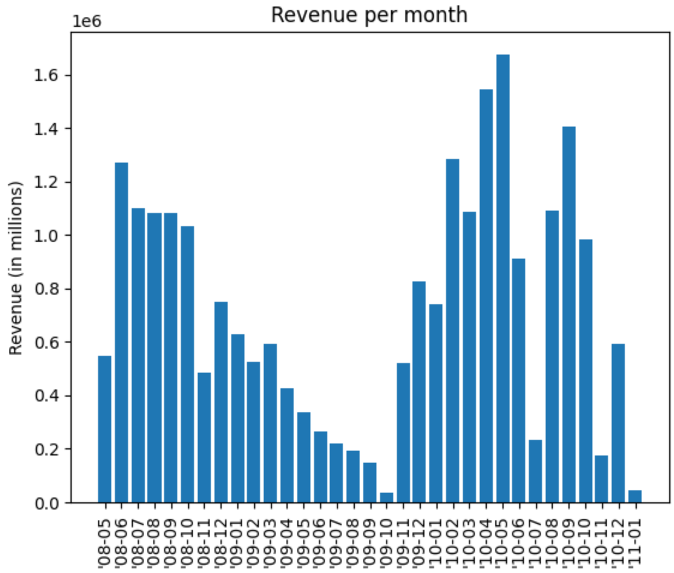

Figure 5: Trip revenue per month

References

[1] White, T. *Hadoop: The Definitive Guide.* O'Reilly Media, Inc, 2015.