Missed Connection Analysis on OTP Flights Dataset (Spark version)

Jun Cai, Vikas Boddu

February 25, 2016

This document describes an implementation of Mapreduce instance with Hadoop doing missed connection analysis on the OTP (On-Time Performance) dataset. The purpose of the application is to find out how many missed connections happened for each carrier for each year.

High-level Design

To find out all the missed connections, a join between flights arrived and departed from the same airport. For two reasons, we are using the Reducer-side join approach. The first reason is that since we can't garantee that all the records for the same carrier for the same airport and the same year will be processed in one Mapper instance. So by doing a Mapper-only join we might get an incomplete join. The second reason is to reduce the data trasfer between the Mapper and the Reducer. The size of the joined data may be as large as O(N^2) where N is the number of flights. So in the Mapper, we just extract related information about each flights from the input data. Then in Reducer, we compare (arrived flight, departed flight) from the same carrier, for the same airpot to compute the number of missed connections, in a very careful way (will be discussed in the Implementation chapter).

For the cross year missed connections, we include them in the previous year's result (i.e. a connection across 2012 and 2013 will belong to year 2012). The concern is that the flight schduled in the previous year is most likely be responsible for that missed connection.

Implementation: MapReduce

Mapper

The output of the Mapper is a (Text, Text) pair for each flight seen, where the key contains carrier ID and year of the flight. The value contains airport ID of both origin and destination airports, the scheduled and actual timestamps of both arrival and departure.

Partitioner

To calculate cross year missed connections, we need to ensure that all records for the same carrier are processed by the same Reducer instance. As mentioned, the key of the Mapper output is a string contains not only the carrier but also the year when the flight is scheduled. A custom Partitioner is implement in our solution to send the key-value pair to a certain Reducer based only on the hash code of the carrier ID (ignore the year).

Reducer

In Reducer, we first create a collection for each airport where all the arrived and departed flights for that airport reside. For each collection, there are two lists. One for arrived flights and one for departed flights. In order to reduce the computation, we split each list into 8760 (or 8748 depends on if it was a leap year; this is the number of hours in that year) sublists. Since a connection is always within 6 hours, so comparing each flight withup to 7 sublists of flights is enough for missed connection calculation. In this way, we optimize the performance of our solution.

The first and last 6 hours' flights for each year are also stored in a static map structure in the Reducer class for cross year missed connection calculation in the cleanup phase.

Implementation: Spark

Parsing and Sanity Check

In our Spark implementation, the each record is first map to an array of strings then filtered by the sanity check function. After that, another map operation will only keep the carrier ID, airport IDs, scheduled departure/arrival timestamps, actual departure/arrival timestamps and departure depay for all the records.

Joining the arrival and departure flights

Then the records will be map into two RDDs: one for departure flights and one for arrival flights. These two RDDs will contains key value pairs where the key contains carrier ID, airport iD and departure/arrvial timestamp (with hour precision). For all the arrival flights, we do a flatmap operation to map one original record into seven records with different timestamps in the key. These seven timestamps will match all the possible timestamps (up to six hours after the arrival) of the departure flights in a connection.

A join is done with this flatmapped arrival flights records and the original departure flights records. The join results will contains all of the possible connections. Then a map operation is performaned to check if each joined record is a connection or missed connection. Finally, a reduce operation is used to sum up the number of missed connections for each carrier for each year.

Performance Discussion

Following is the performance of all the four configurations for missed connection analysis. Note that, all EMR jobs are using 4 worker nodes. In the local configurations, the application is running on a Ubuntu virtual machine with 4g of RAM and 2 processors. The virtual machine is hosted by a laptop with 8g of RAM and a i5-6300U dual-core CPU.

Configuration	Running Time(s)
Spark (Local)	1268
Spark (EMR)	1080
MapReduce (Local)	433
MapReduce (EMR)	480

From the results, we can see that there is no significant performance difference between the local mode and the EMR mode. For Spark, running the job on EMR is slightly faster than on the local virtual machine. But for MapReduce jobs, it's the other way around. Our guess is that there is more computation in our Spark implementation than the Mapreduce implementation (this can be easily inferred from both the code and the running time). When there is more computation, the advantage of parallel data processing will outweight the overhead of the communication and interaction between nodes.

The other interesting observation we just mentioned is that the Spark implementation needs more time to finish than the MapReduce implementation. This might be caused by we having more control on lower level operations in the MapReduce so we don't need to duplicate the reocrds (the flatmap phase of arrival flight records) before join. Instead, the details of how the flight records are joined and reduced can be manipulated in the MapReduce with the help of global variables, the setup and cleanup phases in every Mapper and Reducer. For the purpose of missed connection analysis, MapReduce can achieve the goal in only one map-reduce job. However, we need multiple map-reduce cycles in the Spark implementation (we know there must be some more decent solutions to this problem with Spark, but within the time limit, we can't optimize the implementation further).