Cloud Computing Capstone Task 1

Álvaro Castro-Castilla

All code described here can be found in Github. **The VIDEO can be found here: https://youtu.be/luFAHk8KdUo**.

1 System setup and integration

The task required integrating a Hadoop cluster for data processing, and a Cassandra cluster for storing the processed data, making it available to an end-user API or manual querying.

- **Hadoop cluster**: The Hadoop cluster was set up using *Ambari*, for provisioning, management and monitoring of the Hadoop cluster. For development, the initial cluster consisted of a single Ambari manager and web server, and 3 Hadoop nodes (AWS m4.large instances). Once the algorithms were developed and tested, the cluster was scaled to 6 nodes of the same type. Scaling and stopping/starting instances is seamless and straightforward thanks to Ambari.

  OS: Ubuntu Server 14.04LTS. Hadoop version: 2.3.

- **Cassandra cluster**: The Cassandra cluster was set up using the same AWS instance types, on 3 Ubuntu machines. Only one was established as seed. The setup process is almost completely automated: a base AWS AMI was created with all required components, including the seed node configured. The only manual tweak necessary per instance was the local IP and RPC entries in *cassandra.yaml* config file. Generally this works well as long as the seed node doesn't change IP, without any required updates after reboots. Scaling is also simple, requiring the creation of new instances in AWS and following the same process described.

  OS: Ubuntu Server 14.04LTS. Cassandra version: 2.2.4

- **Sytems integration**: Pig was chosen for Hadoop development. Therefore, it was necessary to connect Pig with Cassandra. However, there seems to be incompatibilities between several versions of Pig and Cassandra. For versions 2.x of Cassandra, Datastax developed the *CqlNativeStorage* and *CqlStorage* Pig UDFs for storing computation results in a Cassandra DB directly. This has been discontinued for more recent versions of Cassandra (3.x), and current status is limited to local setups.

  The practical solution achieved was the use of HDFS as storage for the data in CVS format, which was then transferred into Cassandra. This worked well even for the larger dataset.

  Finally, data plotting was accomplished through Python, so it could be integrated as well as part of the process.

  All questions can be computed running a single Bash script that integrates all necessary files, given all services are up and running.

2 Data extraction and cleaning

The first step for data extraction was the use of an extra volume, attached to an HDFS instance, so data could be unzipped in place and transferred into the distributed filesystem. Once this is done, the original data drive can be unmounted or discarded.

Data cleaning has been done with *Pig* scripting, which has direct access to HDFS, where we had stored the raw unzipped data. The raw files were processed to keep only the relevant fields for computation, which drastically reduced: Original on-time data 34Gb, cleaned 7.4Gb; original origin-destination data 74Gb, cleaned 5.4Gb.

Regarding field merging, the most notorious operation was the merging of date and time for departures, which simplifies some later operations on the data.

Empty delay fields have been considered with a value of zero.

3 Questions: Approach and Solution

Both Hadoop development in Java and Pig have been tested and considered. Pig proved a very interesting option for quickly allowing the prototyping of ideas. Through the use of *UDF*s, Pig becomes a flexible language for developing algorithms in Hadoop, when used in combination with Java. Optimizations are also possible using this system, so all solutions could be computed and optimized with it.

As a note, during the development process, data loaded was limited to a few months, and generally truncated within Pig's scripts to further reduce the amount of data processed. The use of *Tez* as execution engine was also sometimes effective in the reduction of running time during development.

3.1 Q 1-1: Rank the top 10 most popular airports by numbers of flights to/from the airport.

1. Group all flights per origin and per destination separately.
2. Count flights in each one of the groups.
3. Join both per-origin and per-destination lists.
4. Add the counts of flights with each airport as origin and as destination.
5. Sort and then select the first 10 the result to obtain the 10 most popular airports.

Solution: ATL (58187766 flights), ORD (49596357), DFW (44373231), DEN (31219499), LAX (25452018), MSP (24668293), CLT (24276984), DTW (22927673), PHX (21864094), IAH (21501371)

3.2 Q 1-3: Rank the days of the week by on-time arrival performance.

1. Group all flights by day of week.
2. Per each group (one per day of the week), compute the average of all arrival delays.
3. Sort the results.

Solution: Saturday (4.22 mins), Tuesday (5.85 mins), Sunday (6.5 mins), Monday (6.57 mins), Wednesday (7.04 mins), Thursday (8.89 mins), Friday (9.51 mins).

3.3 Q 2-1: For each airport X, rank the top-10 carriers in decreasing order of on-time departure performance from X. See Task 1 Queries for specific queries.

1. Group all flights by origin airport AND carrier.
2. Per each group, compute the average departure delay. This will produce the average departure delay per airport and carrier.
3. Group these results, but this time only by origin airport.
4. Within each group (that is, per airport), sort the carriers according to their average departure delay in that airport, and limit the output to the first 10 carriers.

```
CMI | OH, US, TW, PI, DH, EV, MQ
BWI | F9, PA  1, CO, YV, NW, AA, 9E, US, DL, UA
MIA | 9E, EV, TZ, XE, PA 1, NW, US, UA, ML 1, FL
LAX | MQ, OO, FL, TZ, PS, NW, F9, HA, YV, US
IAH | NW, PI, PA 1, US, F9, AA, TW, WN, MQ, OO
SFO | TZ, MQ, F9, PA 1, NW, PS, DL, CO, US, AA
```

3.4 Q 2-2: For each airport X, rank the top-10 airports in decreasing order of on-time departure performance from X. See Task 1 Queries for specific queries.

Idem than previous, but the first grouping is done by origin AND destination.

```
CMI | ABI, PIT, CVG, DAY, STL, PIA, DFW, ATL, ORD
BWI | SAV, MLB, DAB, SRQ, IAD, UCA, CHO, GSP, OAJ, SJU
MIA | SHV, BUF, SAN, SLC, HOU, ISP, MEM, PSE, TLH, MCI
LAX | SDF, IDA, DRO, RSW, LAX, BZN, MAF, PIH, IYK, MFE
IAH | MSN, AGS, MLI, EFD, HOU, JAC, MTJ, RNO, BPT, VCT
SFO | SDF, MSO, PIH, LGA, PIE, OAK, FAR, BNA, MEM, SCK
```

3.5 Q 2-3: For each source-destination pair X-Y, determine the mean arrival delay (in minutes) for a flight from X to Y. See Task 1 Queries for specific queries.

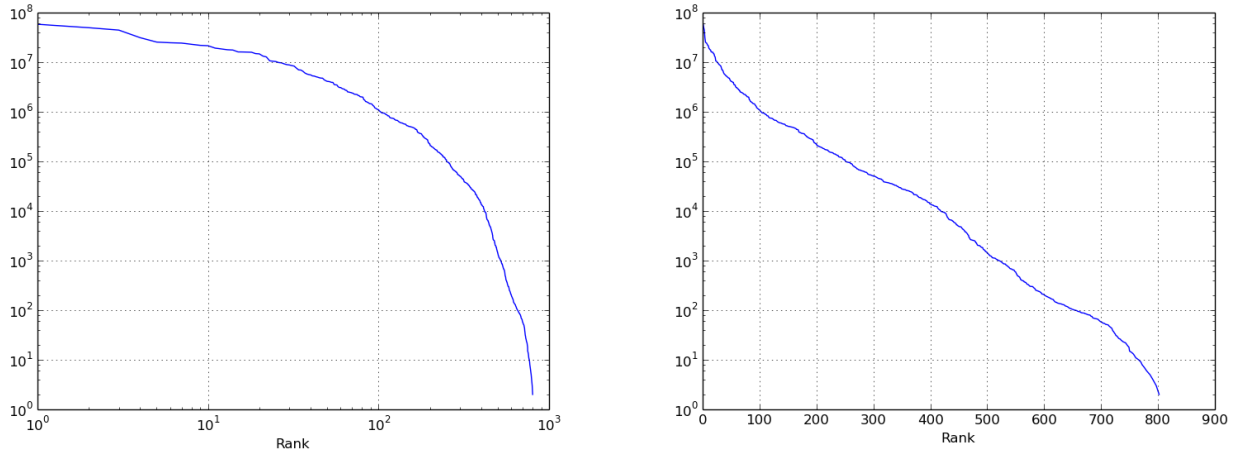
1. Group all flights by origin airport AND destination.
2. Compute the mean arrival delay for the flights in each group.

```
 origin | destination | mean_delay
--------+-------------+------------
    CMI |         ORD |    9.55227
    IND |         CMH |    2.87141
    DFW |         IAH |    7.49696
    LAX |         SFO |    9.15427
    JFK |         LAX |     6.4971
    ATL |         PHX |    8.89073
```

3.6 Q 3-1: Does the popularity distribution of airports follow a Zipf distribution? If not, what distribution does it follow?

Zipf distribution can be observed by plotting frequency against rank, in a log-log graph. If the result is a straight line, the data can be said as following that distribution. To answer this question, Hadoop was used first to determine the popularity of each airport, and its rank in the table. This data is exported and handed to a Python script that plots airport popularity against its rank.

The plot at the left shows the airport popularity (y) and its rank (x), both in logarithmic scale. If it were a Zipf distribution, the graph on the left should approach a straight line. **The data better approaches a linear relation when plotted directly against the rank, so we can draw a much clearer linear relation between the airport rank and the *log* of popularity.**



(a) Airport popularity (log) - Rank (log)

(b) Airport popularity (log) - Rank

Figure 1: Graphs plotting the relationshipt of airport popularity and its rank, with rank plotted in logarithmic (left) and linear (right) scales.

3.7 Q 3-2: Traveler problem

This is the most complex question posed. The algorithm for solving it can be summarized as follows:

1. Generate two lists: one for the first leg of the trip and one for the second. Filter those flights before and after 12pm accordingly.
2. In each of those two lists, group origin, destination and flight date as uniqueness keys.
3. Select the best flight in all those groups, for both lists. That would result in the best performing flight for each leg, separated in the two legs of the trip.
4. Join these two lists, using the destination for the first leg and the origin for the second leg.
5. Filter the combinations that follow the 2 days of difference rule.

```
orig | conn | dest | d_car | origin_date_time | o_fli| c_car| conn_date_time   | c_fli|
-----+------+------+-------+------------------+------+------+------------------+------+
 CMI |  ORD |  LAX |    MQ | 2008-03-04 07:10 | 4278 |   AA | 2008-03-06 19:50 |  607 |
 JAX |  DFW |  CRP |    AA | 2008-09-09 07:25 |  845 |   MQ | 2008-09-11 16:45 | 3627 |
 SLC |  BFL |  LAX |    OO | 2008-04-01 11:00 | 3755 |   OO | 2008-04-03 14:55 | 5429 |
 LAX |  SFO |  PHX |    WN | 2008-07-12 06:50 | 3534 |   US | 2008-07-14 19:25 |  412 |
 DFW |  ORD |  DFW |    UA | 2008-06-10 07:00 | 1104 |   AA | 2008-06-12 16:45 | 2341 |
 LAX |  ORD |  JFK |    UA | 2008-01-01 06:00 |  618 |   B6 | 2008-01-03 19:00 |  918 |
```

Full solution details **here**.

4 Optimizations

Probably the most relevant problem concerning the work on optimizations has been the last. In other questions, similar approaches to optimization have been applied. This is a list of some of the optimizations that have been implemented:

1. Since JOIN is an expensive operation, it is worth pruning the data as much as possible beforehand, to reduce the effects of the resulting combinatorial explosion. All constraints that didn't require the join to be already performed in order to be computed, need to been applied before the JOIN.
2. This allows the use of 'replicated' strategies for joins, an optimization that Pig offers for small enough datasets. The replicated join of Pig actually loads the right-hand side table into memory on each mapper, allowing us to compute the join without reducers.
3. Merging two or more operations in Pig can be done thanks to *UDFs* (user defined functions). An example of this can be found in the 3.2 question script, where and external Java function *ExtremalTupleByNthField* allows us to avoid computing ORDER and LIMIT as two separate operations.
4. Loading only the data that is going to be used saves an intermediate step filtering the appropriate rows, such as in the last exercise loading only data for 2008.
5. A recommendation of Pig's optimization documentation is to prune fields and rows as much as possible and as soon as possible, to lower the amount of data moved in later steps. This is basically a similar idea than the first optimization mentioned, but applied to other operations than JOINs.
6. An easy optimization is the use of Tez as Hadoop Pig executer. It often cuts time by 30%, although not always.

5 Conclusion and opinion on the results

This task has been challenging, but I consider that I've gained a lot of practical knowledge. Most of my issues have been related to systems integration, while I particularly enjoyed the algorithm design and optimization parts.

The generated data seems useful, and you could imagine it being used in real-world research and/or business scenarios. Particularly the last question, while apparently convoluted, seems like a realistic use case of the original dataset for practical purposes.