March 3rd, 2018

**CS417: Data Mining**

Wassim Gharbi, Lafayette College

# Frequent Item Pairs in the Instacart Dataset

## Introduction

This assignment aims to use the Hadoop/MapReduce framework to extract insight on products that are frequently bought together from the "Instacart Online Grocery Shopping Dataset 2017", a publicly available dataset assembled by the Instacart company. The program aims to leverage the properties of the MapReduce framework in order to automate the processing of a large amount of data collected from Instacart user profiles (most importantly orders they have placed on the platform) in the order of 3.4 million rows/hundreds of megabytes of data.

## Dataset Overview

The “Instacart Online Grocery Shopping Dataset” is composed of a big set of CSV (Comma Separated Values) files spanning orders, product names, departments and aisles. However, for the purpose of this assignment we will only be using the files named **`order_products__SET.csv`** and **`products.csv`**, the former contains information about orders made by users (most notably which product IDs belong to which order IDs) while the latter makes the correspondence between product IDs and their names.

For the purpose of faster development, we used a smaller split of product orders (pre-split by Instacart) named **`order_products__train.csv`** and containing only about 130k orders which we used to test the correctness of the algorithm on a single-node server provided by the professor.

Once the algorithm was validated, we ran it on the full dataset (both **`order_products__train.csv`** and **`order_products__prior.csv`**) on an Amazon Web Services EMR cluster in order to process ~4m orders.

## MapReduce Algorithm Architecture

In order to devise the MapReduce algorithm, we took an iterative approach, working on one step at a time and verifying that it fully works before moving on to the next step. We started

by trying to obtain a set of item pairs which we then proceed to count, sort, get the top 100 items and finally match with their product names (obtained from the **`products.csv`** file).

We found the last few steps to be particularly harder than we anticipated it to be: Getting the top 100 items was interestingly counterintuitive using the MapReduce platform, mostly because of the lack of a communication channel between the reducer workers which makes it impossible to determine how many entries each reducer will contribute to the total of 100 items. We solved this problem by simply limiting the number of reducer tasks to one reducer (after consulting with the professor).

Matching the product IDs to their names was also unexpectedly complicated given that it differs substantially from the usual MapReduce Join problem given that we are joining names to a pair of IDs rather than a single ID. We opted to solve this problem in a less optimized way by simply reading the product names from a public URL and joining them in memory inside the last reducer. A better approach to this problem could have been to either create a three-step pipeline that first replaces the first product ID in the pair with its name, then the second product ID with its name. A more optimized approach could make use of a custom partitioner sending the necessary product names corresponding to the given pairs of IDs to the appropriate reducer using a custom hashing function. However, since the product names could easily fit in memory, we chose to employ the easier solution.

The structure of the algorithm is detailed in the following table:

| Job | Step | Goal | Mappers/ Reducers | Functionality | Output Location |
|---|---|---|---|---|---|
| 1 | orders | Associates each product with another product that was bought in the same order (regardless of frequency) | OrderMapper | Emits a key-value pair where the key is the order and the value is a product ID | ./intermediate_1 |
| | | | ProductReducer | For each order, emits every pair of products bought together in that order (making sure not to double-count) | |
| 2 | products | Counts the number of times each product pair was bought together. | IntermediateMapper | Simply reads the last intermediate result and emits the pairs of products with a value of 1 | ./intermediate_2 |
| | | | IntSumReducer | Sums up the 1s to get a frequency of the each product pair | |
| 3 | sortncrop | Sorts the product pairs in descending order and only outputs the first 100 entries combined with their product names | Inversion Mapper | Inverses the order of keys and values in order to allow the data to be sorted by value instead of by key (combined with a custom comparator class that sorts in descending order instead of the default ascending order) | ./output |
| | | | Inversion Reducer | Re-inverses the order of keys and values in order to get back the original structure of the data after sorting by value. Also associates product IDs to their product names and limits the output to 100 values. | |

# Elastic MapReduce Architecture

We ran this algorithm on the entire dataset using two structures: first we used 3 x m4.large instances which led to a run-time of 18 minutes then we used 3 x m4.xlarge instances to see if we could speed up the algorithm which effectively led to a runtime of ~10 minutes in total.

The configuration for the EMR cluster (m4.xlarge) and its steps is detailed below:

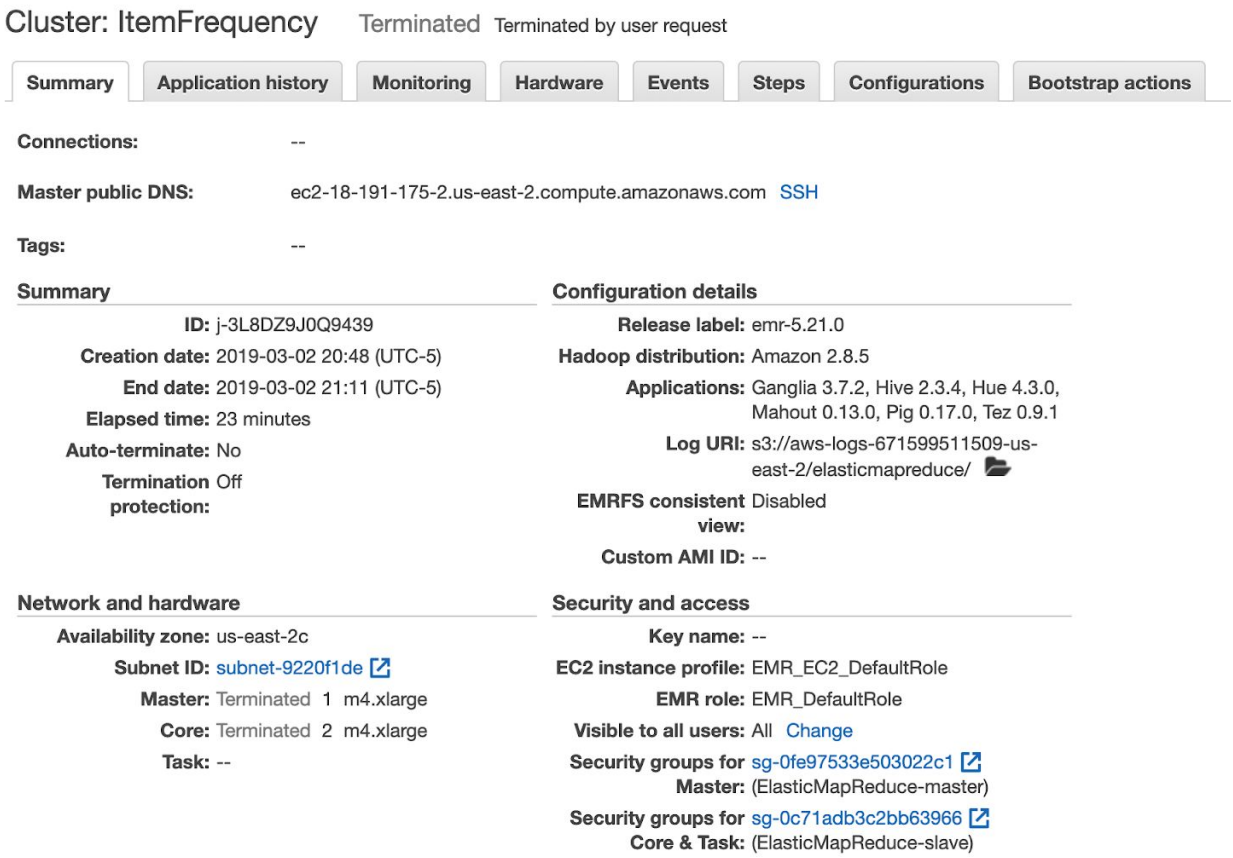
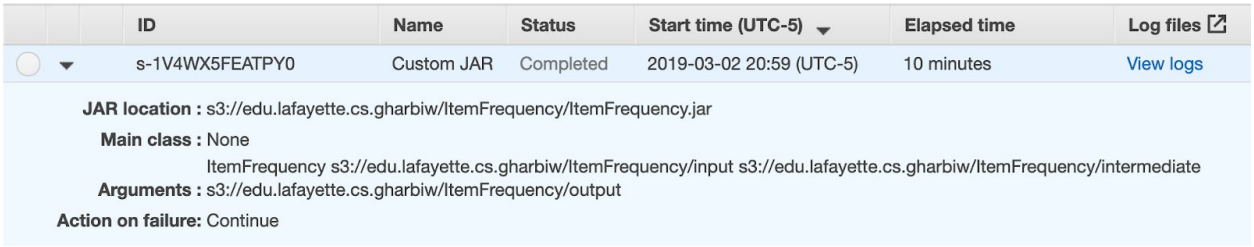


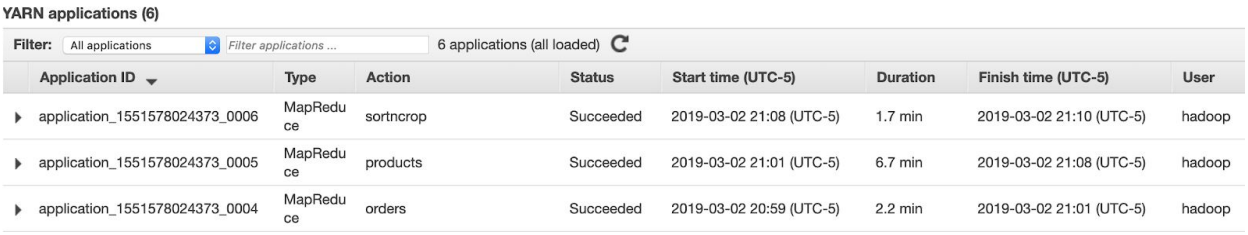

*Figure 1.* Cluster configuration (three instances of m4.xlarge machines)



| ID | Name | Status | Start time (UTC-5) | Elapsed time | Log files |
|---|---|---|---|---|---|
| s-1V4WX5FEATPY0 | Custom JAR | Completed | 2019-03-02 20:59 (UTC-5) | 10 minutes | View logs |

**JAR location :** s3://edu.lafayette.cs.gharbiw/ItemFrequency/ItemFrequency.jar
**Main class :** None
**Arguments :** ItemFrequency s3://edu.lafayette.cs.gharbiw/ItemFrequency/input s3://edu.lafayette.cs.gharbiw/ItemFrequency/intermediate s3://edu.lafayette.cs.gharbiw/ItemFrequency/output
**Action on failure:** Continue

*Figure 2.* Step configuration with specified input, intermediate and output files



YARN applications (6)

Filter: All applications | Filter applications ... | 6 applications (all loaded)

| Application ID | Type | Action | Status | Start time (UTC-5) | Duration | Finish time (UTC-5) | User |
|---|---|---|---|---|---|---|---|
| application_1551578024373_0006 | MapReduce | sortncrop | Succeeded | 2019-03-02 21:08 (UTC-5) | 1.7 min | 2019-03-02 21:10 (UTC-5) | hadoop |
| application_1551578024373_0005 | MapReduce | products | Succeeded | 2019-03-02 21:01 (UTC-5) | 6.7 min | 2019-03-02 21:08 (UTC-5) | hadoop |
| application_1551578024373_0004 | MapReduce | orders | Succeeded | 2019-03-02 20:59 (UTC-5) | 2.2 min | 2019-03-02 21:01 (UTC-5) | hadoop |

*Figure 3.* Application analytics showing run-times for each of the steps in the algorithm

# Results

The results are provided in the **`final_output`** folder detailing the product IDs, names and frequency count for the 100 most frequently bought together items in the dataset. The first few entries are provided below:

```
1   13176,47209,Bag of Organic Bananas,Organic Hass Avocado 64761
2   13176,21137,Bag of Organic Bananas,Organic Strawberries 64702
3   21137,24852,Organic Strawberries,Banana 58330
4   24852,47766,Banana,Organic Avocado  55611
5   21903,24852,Organic Baby Spinach,Banana 53395
6   13176,21903,Bag of Organic Bananas,Organic Baby Spinach 52608
7   16797,24852,Strawberries,Banana 43180
8   24852,47626,Banana,Large Lemon  43038
9   21137,47209,Organic Strawberries,Organic Hass Avocado 42333
10  13176,27966,Bag of Organic Bananas,Organic Raspberries  42283
```

*Figure 4.* First 10 most frequently bought-together products