Project Capstore Task 1 Report

Introduction

Project Capstore performs queries on the transportation datasets[1] from the US Bureau of Transportation Statistics (BTS) hosted on an Amazon EBC Volume Snapshot (snap-e1608d88). The dataset contains a lot of data and statistics, but in this project we will focus on the aviation part up and to 2008. The queries will be executed on Apache Hadoop and Apache Spark system. This report shows how the cloud environment was setup and how data was processed and analyzed.

This report, code sources, results and videos can be found at the following location: https://gitlab.com/ddjohn/cloudcourse

System Integration

Since the AWS promotion credit code had expired only a smaller environment has been set up. All Hadoop (name node, data node) and Yarn nodes (node manager, resource manager) have all been deployed on the same physical EC2 instance. The free tier policy at AWS allows a small node to be setup with very limited CPU, memory and disk.

Name	Instance ID	Instance Type	Availability Zone	Instance State	Status Checks	Alarm Status	Public DNS (IPv4)	IPv4 Public IP
cloud	i-0273054a411d2068d	t2.micro	us-east-1a	running	2/2 checks ...	None	ec2-54-209-122-151.co...	54.209.122.151

Figure 1: A free tier EC2 instance

The EBS snapshot[2] volume with the Aviation sets where added to the EC2 instance as a volume mounted on /cloud. The EC2 instance was created in zone us-east-1a so that it can access the volume.

Name	Volume ID	Size	Volume Type	IOPS	Snapshot	Created	Availability Zone	State	Alarm Status
Root	vol-0fe8496f...	8 GiB	gp2	100	snap-0f08fe61...	December 3, 2018 ...	us-east-1a	in-use	None
Transport	vol-0706a9e...	15 GiB	gp2	100	snap-e1608d88	December 3, 2018 ...	us-east-1a	in-use	None

Figure 2: The root disk and the volume containing aviation datasets

For the data processing the stack below has been used for processing the queries:

[1] http://www.transtats.bts.gov/DataIndex.asp
[2] https://aws.amazon.com/datasets/transportation-databases/

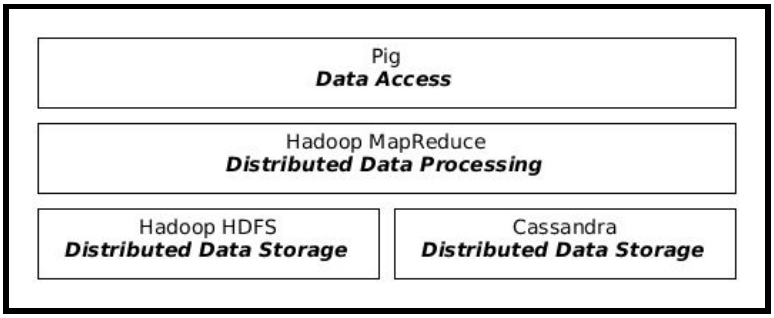

Figure 3: The data processing stack

Apache Pig

Apache Pig is used for the data access layer. It provides a simple way to process the big datasets and to describe and select the columns that are of interest. It is also very powerful to combine data and filter in an sql way.

Hadoop MapReduce

Hadoop MapReduce is used for the distributed data processing layer. To be able to distribute the work on several data nodes the map- and reduce-functions must be defined in a correct way.

Hadoop HDFS

Hadoop HDFS is used for distributed data storage. This is needed since the problem is distributed to map/reduce functions running on different nodes and needs to access common data. HDFS is perfect for handling big data files and splitting it to pieces.

Cassandra

Cassandra is used for distributed data storage. This is needed since the problem is distributed to map/reduce functions running on different nodes and needs to access common data. In contrast to HDFS Cassandra is perfect for managing many small files in parallel.

General Processing

There are several steps in the processing.

- Data cleaning
- Deploy data on HDFS
- Run MapReduce on the data
- Produce a report

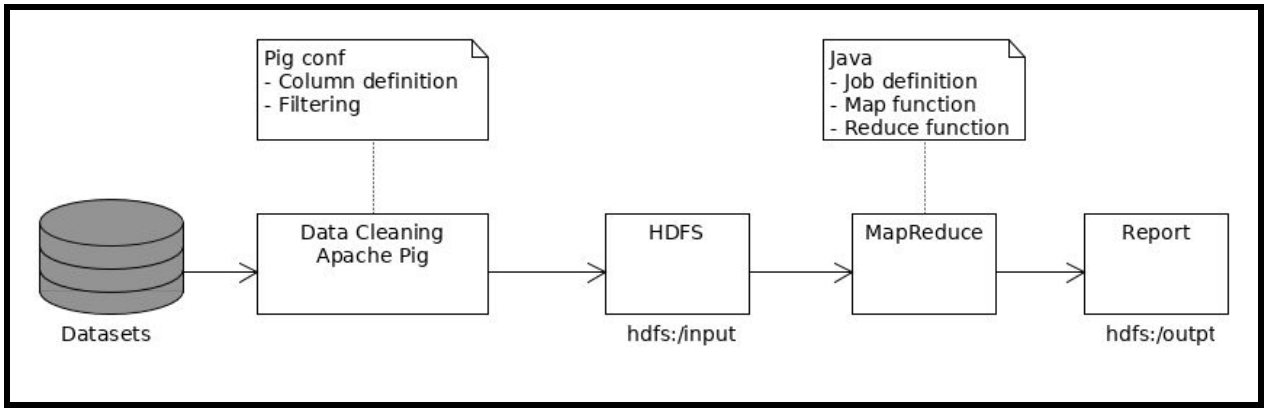

Figure 4 : The process used in this report

Data Cleaning

The datasets provided needs to be cleaned. These are some of the necessary things to consider:

- Dataset contains header rows
- Strings are grouped by “”
- The data may contain commas, just as the field separators
- Two datasets are corrupted: On_Time_On_Time_Performance_2008_11.zip and On_Time_On_Time_Performance_2008_12.zip

Apache Pig has been used to clean up the data. The provided `CSVLoader` has been used to handle the comma separated values. For removing the header row the Linux `tail` command has been used.

```
$ pig -x local -f pigfile -param FILE=datafile | tail +2

DEFINE CSVLoader org.apache.pig.piggybank.storage.CSVLoader();
a = LOAD '$FILE' USING CSVLoader(',') AS (
     Column definitions
);

b = foreach a generate column 1, column 2, column n;
DUMP b;
```

The Pig file is divided into two parts - one describing the format of the dataset and a part that filter out exactly those columns that are of interest for us. The columns that have been used are:

- DAYOFWEEK (int) Day of week
- FLIGHTDATE (chararray) Flight Date (yyyy-mm-dd)
- FLIGHTNUM Flight Number
- UNIQCARRIER (chararray) Unique Carrier

- ORIGIN (chararray) Origin Airport
- DEST (chararray) Destination Airport
- DEPTIME (chararray) Actual Departure Time (local time: hhmm)
- DEPDELAY (float) Difference in minutes between scheduled and actual departure time.
- ARRDELAY (float) Difference in minutes between scheduled and actual arrival time.

Deploy data on HDFS

The data is is copied to HDFS with the following command:

```
$ hdfs dfs -copyFromLocal localfile /hdfsfile
```

Process data with MapReduce

The map() and reduce() functions have been implemented in Java extending Hadoop classes as described in the picture below.

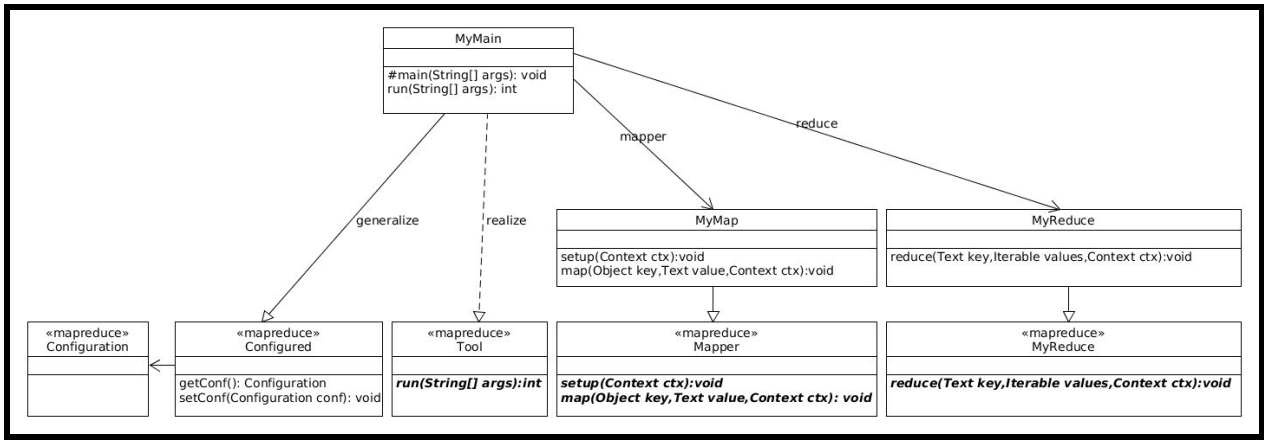

Figure 5: Simplified implementation of MapReduce in Java

To process the datasets the MapReduce functions have been exported to a Java Archive that can be run as:

```
$ hadoop jar cloudcourse.jar SomeMain-function
```

Produce report

Most reports needs to be sorted and report a list of *n* entries. Standard Linux commands have been used for this. In the example below the report is sorted by number on column 2 in reversed order and a limit of 10 is displayed.

```
$ hdfs dfs -cat /output/part-r-00000 | sort -n -k2 -r | head -10
```

The results have also been extracted from HDFS and inserted into Cassandra through `cqlsh` as SQL queries.

Queries and Results[3]

G1Q1--G2Q4

MapReduce functionalities is written in Java. The hadoop-common and hadoop-mapreduce-client-core packages provides the needed framework. The map- and reduce functions written are described in the table below.

Question	Map function	Reduce Function	Note
G1Q1 Popular airports	`add(ORIGIN, 1)` `add(DEST, 1)`	`foreach(key)` `→ sum(values)`	The number of flights will be doubled since we add flights both from origin and dest.

G1Q2 Arrival performance (airlines)	`add(UNIQUECARRIER, ARRDELAY)`	`foreach(key)`
		`→ sum(values)`
		`/numberof(values)`
			Average is calculated by sum the value of a key divided by the number of samples.

G1Q3 Arrival performance (day of week)	`add(DAYOFWEEK, ARRDELAY)`	`foreach(key)`
		`→ sum(values)`
		`/numberof(values)`
			As G1Q2.

G2Q1 Departure performance (airport, carrier)	`add(ORIGIN_UNIQUECARRIER, DEPDELAY)`	`foreach(key)`
		`→ sum(values)`
		`/numerof(values)`
			The key contains of two values. For simplicity the fields are separated with an underscore.

G2Q2 Departure performance (airport, dest)	`add(ORIGIN_DEST, DEPDELAY)`	`foreach(key)`
		`→ sum(values)`
		`/numerof(values)`
			As G2Q1.

G2Q3 Arrival Performance (airport→airport, carrier)	`add (ORIGIN_DEST_UNIQUECARRIER, ARRDELAY)`	`foreach(key)`
		`→ sum(values)`
		`/numerof(values)`
			Contains tree fields in the key.

G2Q4 Arrival performance (airport→`airport`)	`add (ORIGIN_DEST, ARRDELAY)`	`foreach(key)`
		`→ sum(values)`
		`/numerof(values)`
			As G2Q1.

Table 1: Questions G1Q1--G2Q4

G3Q1 Does the popularity distribution of airports follow a Zipf distribution?

Zipf's law states that given some corpus of natural language utterances, the frequency of any word is inversely proportional to its rank in the frequency table. Thus the most frequent word will

[3] Results are available here: https://gitlab.com/ddjohn/cloudcourse/tree/master/solutions

occur approximately twice as often as the second most frequent word, three times as often as the third most frequent word, …[4]

We plot the data from question G1Q1 in normal and logarithmic form. One characteristic aspect with Zipf distribution is that it tends to be linear in logarithmic form. The logarithmic graph seems to not have needed linear aspects and the data do no seem to comply with Zipf distribution.

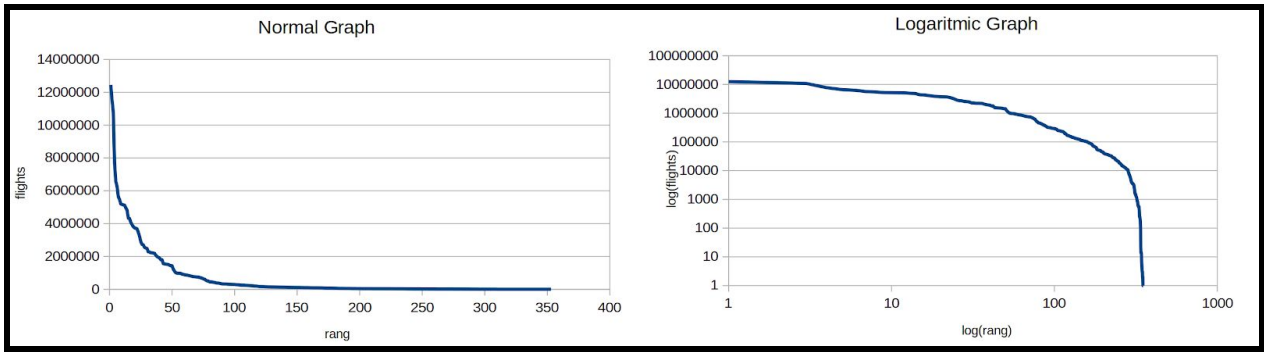

Figure 6: Flight by ranking graphs

G3Q2 Tom's travel from X to Y and Z

This sound a bit like a relationship query that I run entirely with Pig. I join two lists - early_flights that contains flights departing prior to noon and late_flights that contains departing after noon two days after.

The join, sorting and top result is run as below (somewat simplified):

```
j = JOIN early_flights BY (DEST), late_flights BY (ORIGIN);
b = ORDER j BY (early_flights::ARRDELAY+late_flights::ARRDELAY);
LIMIT b 1;
```

Further Optimizations

There are several performance improvements to be made:

- Bigger cloud
  - More computing power
  - More parallelism
- Data cleaning
  - Removing unused columns in the processing
  - Filter out unnecessary rows directly, e.g. cancelled flights
  - Stream directly to HDFS without using filesystem
- MapReduce functions
  - Do not create new Java objects if needed as this takes time
  - Use combiner functions to e.g. limit network traffic

[4] https://en.wikipedia.org/wiki/Zipf%27s_law