Project Capstore Task 2 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.

In this task the queries will be executed on an Apache Spark system. This report shows how the cloud environment is setup and how data was processed and analyzed. This report, code sources, results and videos can be found at the following location: https://github.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

The following components are installed on the EC2 instance:

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

Apache Pig

Apache Pig 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 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.

Zookeeper

Zookeeper is used to coordinate task in the cloud for Kafka and Spark for instance for election and membership.

Apache Kafka

Kafka provide a unified, high-throughput, low-latency platform for handling real-time data feeds. It is used for streaming the cleaned data from HDFS to Spark. Topics are used between producer and consumers of the stream.

Apache Spark

Spark provides an interface for programming entire clusters with implicit data parallelism and fault tolerance. It does that wit Data Frame/Resilient Distributed Datasets processing before sending it to output sink, e.g. Cassandra or console.

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 and distribute data for processing
- Process data
- Store data in database and query the result

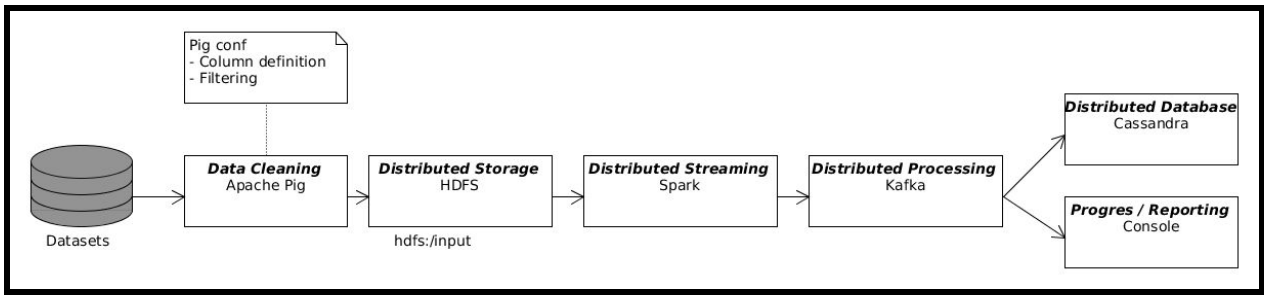

Figure 3: The process used in this report

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 and distribute data for processing

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

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

The data is then distributed by Apache Kafka to consumers on a specific topic:

```
$ hdfs dfs -cat /input | /opt/kafka/bin/kafka-console-producer.sh \
           --broker-list localhost:9092 --topic cloudcourse
```

Process data

Firstly the data coming in is parsed by removing the first and last characters (‘(‘ and ‘)’). The string is tokenized so that each field can be used in the stream.

```
String[] tokens =  x.value().substring(1, x.value().length() - 1).split(",");
```

Individual fields can then be used as below:

```
String origin = tokens[DataSet.ORIGIN];
```

The data is consumed by Apache Spark by creating a stream in Java by providing filters, mapings an update-functions on the data. How the data is streamed is described i the result chapter.

Store data and query the result

The Datastax Spark Cassandra Connector has been used to save a Apache Spark stream directly to Cassandra through the following code:

```
.foreachRDD(rdd -> {
       CassandraJavaUtil.javaFunctions(rdd).writerBuilder("cloudcourse", "g3q2",
              CassandraJavaUtil.mapToRow(Database.class))
       .saveToCassandra();
});
```

In the example above the Database.class needs to map the actual columns in the table in Cassandra.

```
public class Database {
       private String carrier;
       private float delay;

       public Database(String carrier, float delay) {
              this.carrier = carrier;
```

```
            this.delay = delay;
        }

        public String getCarrier() {return carrier;}

        public float getDelay() {return delay;}
}
```

The database can then be queried.

The two latter steps are written in java and can be submitted to the cloud with a jarfile and an entry point:

```
$ /opt/spark/bin/spark-submit --class sparg1q1.G1Q1Main cloudcourse.jar
```

Queries and Results[3]

G1Q1--G2Q4

SPark Streaming and the connection to Cassandra have been written in Java.
The streaming functions written are described in the table below.

Question	Applied functions (pseudo code)	Notes
G1Q1 Popular airports	.flatMapToPair({(ORIGIN,1),(DEST, 1)}) .updateStateByKey(SUM) .mapToPair(x -> x.swap()) .transformToPair(x -> x.sortByKey(DESC)) .mapToPair(x -> x.swap()) .print(10)	The SUM function sums the values per key and returns it. The combination swap->sort->swap was used to sort on the second column. The number of flights will be doubled since we add flights both from origin and dest.
G1Q2 Arrival performance (airlines)	.flatMapToPair(CARRIER, DELAY) .updateStateByKey(AVERAGE)) .mapToPair(x -> x.swap()) .transformToPair(x -> x.sortByKey(ASC)) .mapToPair(x -> x.swap()) .print(10)	The AVERAGE function uses a Java class that help us to handle all corner cases that can end up in division by zero. Average is calculated by sum the value of a key divided by the number of samples.
G1Q3 Arrival performance (day of week)	.flatMapToPair(DAY, DELAY) .updateStateByKey(AVERAGE)) .mapToPair(x -> x.swap()) .transformToPair(x -> x.sortByKey(ASC)) .mapToPair(x -> x.swap()) .print(10)	As G1Q2.

[3] Results are available here: https://github.com/ddjohn/capstore/tree/master/solutions/task2

G2Q1 Departure performance (airport, carrier)	.flatMapToPair(ORIGIN_CARRIER, DELAY) .updateStateByKey(AVERAGE)) .filter(x -> ORIGIN exists in <ist>) .transformToPair(x -> x.sortByKey()) .filter(x -> ORIGIN exists in <ist>) .map(x -> new DatabaseClass(origin, carrier, delay)) .saveToCassandra()	The key contains of two values. For simplicity the fields are concatenated with an underscore. The filter list is {"SRQ", "CMH", "JFK", "SEA", "BOS"};
G2Q2 Departure performance (airport, dest)	.flatMapToPair(ORIGIN_DEST, DELAY) .updateStateByKey(AVERAGE)) .filter(x -> ORIGIN exists in <ist>) .transformToPair(x -> x.sortByKey()) .map(x -> new DatabaseClass(origin, dest, delay)) .saveToCassandra()	As G2Q1.
G2Q3 Arrival Performance (airport→airport, carrier)	.flatMapToPairCARRIER_ORIGIN_DEST, DELAY) .updateStateByKey(AVERAGE)) .transformToPair(x -> x.sortByKey()) .map(x -> new DatabaseClass(carrier, origin, dest, delay)) .saveToCassandra()	Contains tree fields in the key, concatenated with underscore. The filter list is {"LGA_BOS", "BOS_LGA", "OKC_DFW", "MSP_ATL"}
G2Q4 Arrival performance (airport→airport)	.flatMapToPair(ORIGIN_DELAY, DELAY) .updateStateByKey(AVERAGE)) .transformToPair(x -> x.sortByKey()) .print(10) and saveToCassandra()	As G2Q3.

Table 1: Questions G1Q1--G2Q4

G3Q2 Tom's Flight from X to Y and Z

This sound a bit like a relationship query so I will fill Cassandra with relevant data and query from the Database.

Initially I parse the stream building up a Java class with available flights with the following fields:
`Origin, Dest, FlightDate, DepTime, concat(Carrier, FlightNum), Delay`

To limit the list stored in Cassandra i add three filters:

- Filter out all flights for the year 2008
- Filter out flights with origin {"BOS", "PHX", "DFW", "LAX"} and dest {"ATL", "JFK", "STL", "MIA"} that departures prior to 1200
- Filter out flights with origin {"ATL", "JFK", "STL", "MIA"} and dest {"LAX", "MSP", "ORD", "LAX"} that departures after to 1200

The interesting flights should now be in the database and can now be queried with CQL:

```
cqlsh> select origin,dest,flight,depdate,deptime,MIN(delay)
   ... from cloudcourse.g3q2
   ... where origin = 'BOS' and dest = 'ATL' and depdate = '2008-04-03'
   ... and deptime <= '1200' allow filtering;
```

```
 origin | dest | flight | depdate    | deptime | system.min(delay)
--------+------+--------+------------+---------+-------------------
    BOS |  ATL | FL 273 | 2008-04-03 |    0853 |                 7

cqlsh> select origin,dest,flight,depdate,deptime,MIN(delay)
   ... from cloudcourse.g3q2
   ... where origin = 'ATL' and dest = 'LAX' and depdate = '2008-04-05'
   ... and deptime >= '1200' allow filtering;

 origin | dest | flight | depdate    | deptime | system.min(delay)
--------+------+--------+------------+---------+-------------------
    ATL |  LAX |  DL 75 | 2008-04-05 |    1704 |                -2
```

Further Optimizations

There are several performance improvements to be made:

- Bigger cloud
    - More computing power
    - More parallelism
    - A promotion code :-)
- Data cleaning
    - Removing unused columns in the processing
    - Filter out unnecessary rows directly, e.g. cancelled flights
- Processing
    - Currently Spark cannot handle the data being streamed by Kafka, so adding executors should do the trick.
    - Functions like reduce(), group() and update() can do the same job, but seems to be implemented very differently and knowing the underlying mechanism would help to select the most efficient one for what processing that is done.

Sum

Spark seems to be very easy to program and much quicker than MapReduce. It also uses memory in favour of disk which leads to smaller latency in the cloud. While MapReduce produces a result for a fixed problem, Spark is streaming continuously.

I can however see that MapReduce might be the solution for bigger problems that will not fit into physical memory.