Project Intermediate Report

Team: Sparklings
Team Members:
Shubham Rastogi (MW Section)
Abhishek Ahuja (TR Section)
Ritika Nair (MW Section)
Link to GitHub Repo: https://github.ccs.neu.edu/cs6240f18/Sparklings

Input Dataset

Name: On-time performance of flights in the US, published by the Bureau of Transportation Statistics
Link: https:/www.transtats.bts.gov/DL_SelectFields.asp?Table_ID=236&DB_Short_Name=On-Time

Description: The data contains scheduled and actual departure and arrival times reported by certified U.S. air carriers that account for at least one percent of domestic scheduled passenger revenues. The data is collected by the Office of Airline Information, Bureau of Transportation Statistics (BTS). The above link allows us to create a custom dataset in terms of number of records (we can select the number of years and months) and attributes. We are planning to use the data for 10 years from 2008 to 2018 which will amount to approximately 60 million records with 17 attributes each (including origin airport, destination airport, flight data, reason of delay, departure delay etc.). The space required to store this dataset will be slightly over 1 GB. It will be stored in CSV format.

Sample Input Record:

FL_DATE	ORIGIN_AIRPORT_ID	ORIGIN_CITY_NAME	DEST_AIRPORT_ID	DEST_CITY_NAME	DEP_TIME	DEP_DELAY	ARR_TIME	ARR_DELAY	CANCELLED
6/1/2018	12953	New York, NY	14492	Raleigh/Durham, NC	1114	104	1245	80	0

CRS_ELAPSED_TIME	ACTUAL_ELAPSED_TIME	DISTANCE	CARRIER_DELAY	WEATHER_DELAY	NAS_DELAY	SECURITY_DELAY	LATE_AIRCRAFT_DELAY	ACTUAL_ELAPSED_TIME
115	91	431	0	0	0	0	80	91

TASKS

1. Join implementation using HBase

Goal

To find a 3-hop itinerary with a constraint of spending at least 10 hours in each city, by performing reduce side joins on the flight data.

Hbase and Joins

Two of the fundamental techniques use for performing joins are :

- Hash + Shuffle (Reduce Side Join)
- Partition + Broadcast (Replication join)

One of the biggest disadvantages of reduce side join on huge data sets is its tendency to duplicate data while transferring it from mapper to reducer. This becomes a huge bottleneck in terms of memory. Also, this algorithm is limited to only equi joins and cannot perform/extend well to theta joins.

Replication join can perform theta joins but however they also have a huge bottleneck of transferring one of the data sets to all mappers. One advantage of replicated join is it can perform entire join computation in map phase itself without the need of shuffling. However, to accomplish map only computation it has to broadcast one of the data sets to all the map tasks. Unless one of the data sets is extremely small, replication join can lead to higher memory consumption.

Why data duplication/transfer is needed in the above algorithms?

It is needed so that it is possible to apply join logic to the entire data sets containing common attributes. If only a sample of data is transferred to avoid memory costs, then it will not lead to efficient join computations and join results will have missing tuples.

Solution from Hbase

Because of its ability to access random data stored on hdfs in the form of regions, Hbase acts as a hash index on the data store. Hence, whenever a join needs to perform on a common attribute, Hbase requests for that row key from ZooKeeper (performs coordination among multiple region servers and contains meta data to handle incoming requests). Zookeeper directs the request to the region server from which it can fetch all the matching keys as per the join attribute.

PseudoCode

The implementation consist of three Map reduce jobs :

- **Insert data into Hbase Table : Map only job that inserts data to Hbase Table “airlines”.**

Driver

//set input format to path to the data source

//set output format to TableOutputFormat.OUTPUT_TABLE

jobConfiguration.set(TableOutputFormat.OUTPUT_TABLE, “airlines”) //airlines name of the table

Map (TextFile line)

//convert the line into line array with each element of the array corresponding to one input cell

//create a hbase put object and set rowkey as line.source_city_id

Put put = new Put(line.source_city_id);

//add other columns to put object using the array created above

Emit(null, put) //this will write the data to hbase table “airlines” as per the output format in driver.

- Fetch the two hop path

Driver

//using mapreduceutils set the input and output job configs

TableMapReduceUtil.initTableMapperJob("**airlines**", s, HbaseJoinMapper.class, Text.class, Text.class,job);

TableMapReduceUtil.initTableReducerJob("**AirLinesTwoHop**", HbaseJoinReducer.class,job);

Map(a row of “airlines” table)

Emit(source_city_id, destination_City_id)

Reduce(city_id, a list of all the destinations can be reached)

for a destination d, find all the rows where it acts as a source :

for a destination s from d:

newrowkey = source_city_id + d + s

add the newrowkey and rest of the columns into a put object of table AirLinesTwoHop

emit(null, put object) //the put object will have the row key of the form a -> b -> c

- **Fetch the three hop path**

This job is similar to two hop path job with two input tables : “airlines” and “AirLinesTwoHop” and one output table AirLinesThreeHop. The row key is of the form : source_id > d1 > d2 > d3, where d* corresponds to destinations.

Algorithm and Program Analysis

The algorithm above can be used to determine best 3 hop paths between a source and a destination and more deeper insights by performing early filtration and projection which will be implemented in the final phase of the project.

The rows can be further refined to output more relevant column data to be used in the intermediate jobs.

Experiments

The program is executed on the data of January month of the year 2008. The data has more than 56k records. The sample output data is as follows :

Sample Output

A row of Two hop path : the row key corresponds to <source -> destination1 -> destination2

```
101351039715016     column=info:ARR_DELAY, timestamp=1543628156867, value=64.0
            0
 101351039715016     column=info:ARR_TIME, timestamp=1543628156867, value="2302
            "
 101351039715016     column=info:CANCELLED, timestamp=1543628156867, value=0.00
 101351039715016     column=info:CARRIER_DELAY, timestamp=1543628156867, value=
            0.00
 101351039715016     column=info:CRS_ELAPSED_TIME, timestamp=1543628156867, val
            ue=93.00
 101351039715016     column=info:DEP_DELAY, timestamp=1543628156867, value=33.0
            0
 101351039715016     column=info:DEP_TIME, timestamp=1543628156867, value="2158
            "
 101351039715016     column=info:DEST_CITY_NAME, timestamp=1543628156867, value
            ="St. Louis
 101351039715016     column=info:DISTANCE, timestamp=1543628156867, value=483.0
            0
 101351039715016     column=info:LATE_AIRCRAFT_DELAY, timestamp=1543628156867,
            value=33.00
^C 101351039715016     column=info:NAS_DELAY, timestamp=1543628156867, value=31.0
            0
 101351039715016     column=info:ORIGIN_CITY_NAME, timestamp=1543628156867, val
            ue="Atlanta
 101351039715016     column=info:SECURITY_DELAY, timestamp=1543628156867, value
            =0.00
 101351039715016     column=info:WEATHER_DELAY, timestamp=1543628156867, value=
            0.00
```

2. Join implementation using Reduce Side Join

Goal

To find a 3-hop itinerary with a constraint of spending at least 10 hours in each city, by performing reduce side joins on the flight data.

Pseudocode

Job 1: Preprocessing data by performing selections and projections. Columns related to delay causes and cancellation were removed and rows for cancelled flight data are ignored.

Job 2: Performs First Hop to get to the first two cities

```
map(FlightData airlineRow) {
```

```
        emit(airlineRow.getDestAirportId(),airlineRow.withTag("in") );
        emit(airlineRow.getOriginAirportId(), airlineRow.withTag("out"));

}

reduce(Text airportId, Iterable<AirlineRowWritable> values) {

        outFlights = new ArrayList
        inFlights = new ArrayList

        //Segregate into leaving and arriving flights
        For each flight in values {
                if(flight.getTag()=="out") outFlights.add(flight )
                if(flight.getTag()==in") inFlights.add(flight)

        }

        //Join all possible pairs in both lists
        For each outflight in outFlights {
                For each inflight in inFlights) {

                        if(reachTime - leaveTime) >=10{
                                emit(inflight, outflight)
                        }
                }
        }
    }
```

Job 3: Performs Second Hop to get to the third city

One mapper reads the first hop joined result and the second mapper reads the original flight data. Job2's reducer was reused.

```
map(FlightData firstHopPath) {
        emit(firstHopPath.getDestAirportId(),airlineRow.withTag("in") )
}


map(FlightData airlineRow) {
        emit(airlineRow.getOriginAirportId(), airlineRow.withTag("out"))
}
```

Job 4: Performs Third Hop to get back to source

This is pending.

Algorithm/Program analysis

To optimize our program and reduce volume of data transferred we preprocessed the data by performing projection and selection to remove irrelevant columns (such as delay causes, cancellation code etc) and rows(cancelled flights).

We further plan to extract out 'aiportId-airportName' mapping to a separate file so that we can avoid sending originAirportName and destAirportName within each job. This will further reduce data transfer.

Experiments

Speedup:

Machine type: m4 large

Data	**5 machine cluster**	**10 machine cluster**
6 months data	15 mins	13 mins

Indicates average speedup for this kind of high volume data.

Scalability:

These were run on local machine:

Data	Run time
1 month data	1 min 16 secs
6 months data	16 min 45 sec

Sample Output:

The following is one row of output. It represents a 2 hop itinerary: starting from Jacksonville on 11th September to travel to Charlotte, then going to Washington DC on 19th September, then to Miami on 23rd September.

2018-09-11,13795,Jacksonville/Camp Lejeune NC,11057,Charlotte NC,0552,0659,67.0----
2018-09-19,11057,Charlotte NC,11278,Washington DC,0913,1031,78.0----
2018-09-23,11278,Washington DC,13303,Miami FL,1350,1634,164.0

3. Comparison and analysis of results

Pending. This task is dependent on the completion of above Tasks 1 and 2.

RISKS

The volume of data transfer is still huge. This is because we require at least 6 columns (flight date, origin airport id, destination airport id, departure time, arrival time) for each flight on each join. This may result in a long run time.

References: https://www.transtats.bts.gov/DatabaseInfo.asp?DB_ID=120&DB_URL=