CS 839 NoSQL Systems: Assignment 3

Chinthan Chandra
IMT2020109
IIIT Bangalore
chinthan.chandra@iiitb.ac.in

Kaushik Mishra
IMT2020137
IIIT Bangalore
kaushik.mishra@iiitb.ac.in

Vivek Tangudu
IMT2020110
IIIT Bangalore
vivek.tangudu@iiitb.ac.in

INTRODUCTION

We were given the YAGO dataset and two tasks to be done using `Pig`, `Latin`, `Hive` and `HiveQL`.

PART A: PIG AND LATIN

12 points

Problem 1. PROCESSING YAGO DATASET USING PIG.

7 points

1 **Load the `YAGO dataset` and find out the top three frequently occurring predicates in the YAGO dataset using operators available in the Pig Latin. (3 points)**

Ans. The task is to find the top three predicates. To do this first we load the yago dataset into a bag datatype with each tuple in the form of (subject: chararray, predicate: chararray, object: chararray). Since the file is tsv but on looking at the file we found it was actually space separated, so we used this as a separator to load the file into the triples bag. The command for the same is:

```
triples= load '../yago_full_clean.tsv'
using PigStorage(' ') AS (subject:
chararray, predicate: chararray, object:
chararray);
```

Since we need only predicates from this we generate a new bag using the triples bag.

```
predicates= FOREACH triples GENERATE
predicate;
```

Next we group the predicates bag based on the predicate.

```
predicates_grouped= GROUP predicates BY
predicate;
```

Now we check the count of predicates.

```
predicate_counts = FOREACH
predicates_grouped GENERATE group as
predicate, COUNT(predicates) AS count;
```

Order them in descending order.

```
predicate_counts_desc= ORDER
predicate_counts BY count DESC;
```

Since we need only the top 3, we limit the output and store it in a new bag.

```
top_three_predicates = LIMIT
predicate_counts_desc 3;
```

Lastly, we store them into a tsv file.

```
store top_three_predicates
INTO './top_three' USING
PigStorage('\t','-schema');
```

All these commands are stored in `A3-AP11.pig`. The output is shown in Fig 1.

```
cat top_three/.pig_header top_three/part-r-00000
predicate       count
<isCitizenOf>   2141725
<hasFamilyName> 2002574
<hasGivenName>  1984813
```

Fig. 1: Top three predicates

2 **Identify all the given-names (i.e., object values of the `<hasGivenName>` predicate) of persons who are associated with more than on `<livesIn>` predicates from the YAGO dataset using the relational operators (join, grouping, etc.) in the Pig Latin. (4 points)**

Ans. For this task, we do the same loading as earlier. We create two new bags by filtering based on the predicates `<hasGivenName>` and `<livesIn>`.

```
hasGiven= FILTER triples BY
predicate=='<hasGivenName>';
livesIn= FILTER triples BY
predicate=='<livesIn>';
```

Since we need to take only those which have a count of more than one in the `livesIn` bag, we perform a grouping and then get the counts and filter based on this.

```
livesInGrouped = Group livesIn BY subject;
livesInGroupedCounts = FILTER ( FOREACH
livesInGrouped GENERATE group AS Name,
COUNT(livesIn) AS counts ) BY counts>1 ;
```

Next we perform a join on `hasGiven` and `livesInGroupedCounts` on the subject field of tuples of both of the bags. It is Name in the bag `livesInGroupedCounts` as it is renamed in the previous command.

```
joined = JOIN hasGiven BY subject,
livesInGroupedCounts BY Name;
```

Next we take only the objects and then store them into a tsv file.

```
givenNames = FOREACH joined GENERATE
object;
STORE givenNames INTO './givenNames' USING
PigStorage('\t','-schema');
```

All the commands for this are stored in `A3-AP12.pig`. The output is shown in Fig 2.

```
cat givenNames/.pig_header givenNames/part-r-00000
<Ilze>
<Imtiaz>
<Inge>
<Irawati>
<Irene>
<Irina>
<Irving>
<Isaac>
<Isha>
<Ishita>
<Ismail>
<Ivan>
<Ivan>
```

Fig. 2: Given-names

Problem-2 UDF. **5 points**

1 **Use the sample.txt data discussed in the lecture and do the following:**

a **Group the students based on the group-id.**

b **For each group create a new attribute named "project tool" and assign randomly a value from the list [MR, Pig, Hive, MongoDB]. Define a user-defined function for this purpose.**

Ans. We load the file into a bag with the schema `(name:chararray,rollno:chararray, emailid:chararray,groupid:int)`.
Then we group using the groupid.
`GroupedRecords = GROUP MyRecords By groupid;`
We register our UDF using `Register A3-AP2.jar;`.
The UDF is an eval function type that takes in a tuple and returns a string for each tuple. The string returned is the tool assigned to the group.
The code is:

```
public class AssignRandomUDF extends
    EvalFunc<String>{
  public String[]
      tool={"MR","Pig","Hive","MongoDB"};
  public Random random= new Random();
  public String exec(Tuple input) throws
      IOException {
      try{
        int index=random.nextInt(4);
        return tool[index];
      }catch(Exception e){
         throw new IOException("Caught
             exception processing input row
             ", e);
      }
  }
}
```

This basically chooses one number from 0 to 4 randomly and assigns the corresponding tool from the tool array.
We use this to generate one tool per group and take only the groupid and tool.
`AssignedProjectTool = FOREACH GroupedRecords GENERATE group AS GroupID, AssignRandomUDF() AS ProjectTool;`
We store these into a tsv file.
`store AssignedProjectTool into './projectTool' using PigStorage('\t','-schema');`
All the commands can be found in `A3-AP2.pig` and the java code is in `AssignRandomUDF.java`. Output is shown in Fig. 3.

```
cat projectTool/.pig_header projectTool/part-r-00000
GroupID ProjectTool
1       Pig
2       MongoDB
3       Pig
4       Hive
5       Hive
6       Pig
7       MR
8       Hive
9       MR
10      Hive
11      Pig
12      Hive
13      MongoDB
14      Pig
15      Pig
16      MR
17      Pig
18      MR
19      Hive
20      MR
21      MongoDB
```

Fig. 3: Project Tools Assigned

PART-B: HIVE AND HIVEQL

Problem 1. PROCESSING YAGO DATASET USING HIVE

1 **Load the YAGO dataset and find out the top three frequently occurring predicates in the YAGO dataset using operators available in HiveQL. (3 points)**
Ans. We have to do the same problem given above but using HiveQL this time. We start by creating an external table with the schema that follows `subject STRING, predicate STRING, object STRING`. We specify that the fields are delimited by ' ' for correct loading from the tsv file. Then we load it using a load query into the yago table. The full query is:

```
CREATE TABLE yago(
subject STRING,
predicate STRING,
object STRING)
ROW FORMAT DELIMITED
FIELDS TERMINATED BY ' ';

LOAD DATA LOCAL INPATH
'/pathTo/yago_full_clean.tsv' INTO TABLE
yago;
```

This is the initial table that will be used for all the tasks ahead. The corresponding code is in `init.hiveql`.
After this has been done, we need to find the top 3 most frequently occurring predicates. This is done by a simple query that counts the number of rows when grouped by predicates and the number of output rows is limited to 3. The query is:

```
SELECT predicate, COUNT(*) AS count
FROM yago
GROUP BY predicate
SORT BY count DESC LIMIT 3;
```

The queries can be found in `A3-BP11.hiveql`. The output for the same is shown in Fig. 4.

```
OK
+------------------+----------+
|    predicate     |  count   |
+------------------+----------+
| <isCitizenOf>    | 2141725  |
| <hasFamilyName>  | 2002574  |
| <hasGivenName>   | 1984813  |
+------------------+----------+
3 rows selected (11.602 seconds)
```

Fig. 4: Top three predicates

2 **Identify all the given-names (i.e., object values of the hasGivenName predicate) of persons who are associated**

with more than one LivesIn predicates from the YAGO dataset using the relational operators (join, grouping, etc.) in HiveQL. (4 points)

Ans. Now, we need to find all the object values associated with the <hasGivenName> predicate and also associated with more than one <livesIn> predicate. For this, we use two select subqueries, one that gets all rows containing the <hasGivenName> predicate and another one with all rows containing the <livesIn> predicate which is grouped by the subject and has count¿1. Then we perform a join on these two subqueries on their subject column. The query is:

```
SELECT hasgiven.object AS Object
FROM (
SELECT subject,object
FROM yago
WHERE predicate = '<hasGivenName>'
) AS hasgiven
JOIN (
SELECT subject, COUNT(*) AS count
FROM yago
WHERE predicate = '<livesIn>' GROUP BY
subject HAVING count>1
) AS livesin
ON (hasgiven.subject = livesin.subject);
```

The queries can be found in A3-BP12.hiveql. The output for the same is shown in Fig. 5

Fig. 5: Given Names

Problem 2. USING PARTITIONS AND BUCKETS

1 **Write a HiveQL query to find all the subjects (x) and objects (y and z) matching the pattern: ?x <hasGivenName> ?y. ?x <livesIn> ?z., from the Yago dataset . Implement this problem by**

1 **by considering partitioning and bucketing;**

2 **by considering partitioning but not bucketing;**

3 **by considering neither partitioning nor bucketing.**

Ans. In this problem, we have to implement it in 3 different ways. For the first two, we need to create the same yago table that we created 2 more times, once using partitions and once using partitions and buckets. We use the existing yago table to insert data into the new tables. We create partitions in both tables on predicates statically and we create buckets that are clustered on the subject column since our queries are primarily focused on these two columns.

For creating table with partitions only:

```
CREATE TABLE yago_part_only(
subject STRING,
predicate STRING,
object STRING)
PARTITIONED BY (pred STRING);
```

For creating table with buckets and partitions:

```
CREATE TABLE yago_buck_part(
subject STRING,
predicate STRING,
object STRING)
PARTITIONED BY (pred STRING)
CLUSTERED BY (subject)
INTO 4 BUCKETS;
```

To load data into these tables, we need to manually load each into each predicate partition, buckets are made for each partition automatically.

Query for loading in one predicate partition(repeat for each of the 29 predicates):

```
INSERT INTO yago_part_only PARTITION
(pred='<actedIn>')
SELECT subject, predicate, object
FROM yago
WHERE predicate='<actedIn>';
```

The queries to create the tables can be found in A3-BP21.hiveql for buckets and partitions and in A3-BP22.hiveql for partitions only.

The query for obtaining the given pattern is the same for all three with some changes for part i and ii.

The query format is

```
SELECT hasgiven.subject AS Subject,
hasgiven.object, livesin.object
FROM (
SELECT *
FROM tablename
WHERE predicate = '<hasGivenName>'
) AS hasgiven
JOIN (
SELECT *
FROM tablename
WHERE predicate = '<livesIn>'
) AS livesin
ON (hasgiven.subject = livesin.subject);
```

i **by considering partitioning and bucketing;**

For this we just need run the above query with the table name yago_part_buck. The query is:

```
SELECT hasgiven.subject AS Subject,
hasgiven.object, livesin.object
FROM (
SELECT *
FROM yago_part_buck
WHERE pred = '<hasGivenName>'
) AS hasgiven
JOIN (
```

```
SELECT *
FROM yago_part_buck
WHERE pred = '<livesIn>'
) AS livesin
ON (hasgiven.subject =
livesin.subject);
```
This query can be found in `B1.hql`. The output is shown in Fig. 7.

Fig. 6: With bucketing and partitioning

ii **by considering partitioning but no bucketing;**
For running the query on the partitioned table, we do a similar join by using the pred column that is created in the partitioned table. This means that we are using the partitions and end up saving a lot of time in searching.
The query is:
```
SELECT hasgiven.subject AS Subject,
hasgiven.object, livesin.object
FROM (
SELECT *
FROM yago_part
WHERE pred = '<hasGivenName>'
) AS hasgiven
JOIN (
SELECT *
FROM yago_part
WHERE pred = '<livesIn>'
) AS livesin
ON (hasgiven.subject =
livesin.subject);
```
This query can be found in `B2.hql`. The output is shown in Fig. 7.

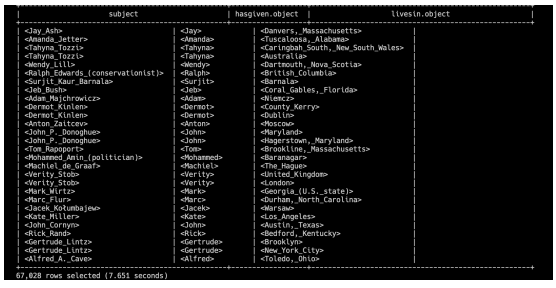

Fig. 7: With partitioning and no bucketing

iii Now, for running the queries with no partition and no bucketing. We take the two copies of the yago table, join them on the subject, and check for each containing the hasGivenName predicate on the first table and the livesIn predicate on the second table.
The query is:
```
SELECT hasgiven.subject AS Subject,
hasgiven.object, livesin.object
FROM (
SELECT *
FROM yago
WHERE predicate = '<hasGivenName>'
) AS hasgiven
JOIN (
SELECT *
FROM yago
WHERE predicate = '<livesIn>'
) AS livesin
ON (hasgiven.subject =
livesin.subject);
```
The output is shown in Fig. 8.

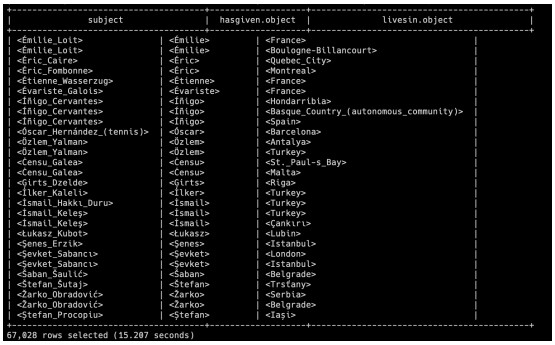

Fig. 8: Without bucketing and partitioning

Comparing runtime of the 3 cases:
The third case is the slowest as we are performing two filters on the same table using the predicate names and then applying a join on these two tables. It takes approximately 15-16s on an M1 MacBook.
Both query 1 which uses bucketing and partitioning and query which uses only partitioning are much faster than the third query. This is due to the fact that the partitions which are made based on predicates are used in both cases.
When run on an M1 MacBook they have similar runtimes. This might be due to the fact that the subject buckets aren't very necessary as we are using predicates to get the two subqueries and perform the joins.
However, we tested on a Virtual Machine which is a much slower system and the performance gain of using buckets is evident, performance went from 36s using only partitions to 31s with buckets.
Also, changing the number of buckets seems to have an effect on performance for this dataset and this query.
The runtimes can be observed in the attached images.