CS839 NoSQL Systems - Assignment 3

Group 21

Malay Agarwal (MT2022059)
Umair Ahmad Beig (MT2022126)
Sami Ullah Naikoo (MT2022099)

Contents

Part A: Pig and PigLatin

Problem 1: Processing YAGO Dataset Using Pig

Problem-1: Processing YAGO dataset using PIG

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)

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)

General Instructions

The steps to execute the Pig scripts are as follows:

- Make sure all Hadoop services are running:

```
start-all.sh
```

- Add the datasets to the Hadoop file system.

```
hdfs dfs -mkdir /input
hdfs dfs -put /path/to/dataset /input
```

- Create the output directory:

```
hdfs dfs -mkdir /output
```

- Run the script:

```
pig /path/to/script/ --verbose
```

Solution 1 - Script

The script is as shown:

```
yago = LOAD '/input/yago_full_clean.tsv'
    USING PigStorage(' ')
    AS (subject:chararray,
        predicate:chararray,
        object:chararray
    );

yago1 = FILTER yago BY predicate != '';

yago2 = FOREACH yago1
    GENERATE subject, TRIM(predicate) AS predicate, object;
```

```
grouped_data = GROUP yago2 BY predicate;

count_data = FOREACH grouped_data
    GENERATE groupAS predicate, COUNT(yago2) AS count;

sorted_data = ORDER count_data BY count DESC;

STORE sorted_data
INTO '/path/to/output/file'
USING PigStorage();

DUMP sorted_data;
```

We first load the dataset into `yago` from the file `/input/yago_full_clean.tsv` on HDFS, remove any rows where the `predicate` is empty and remove trailing whitespace characters are removed from `predicate` to obtain `yago2`. Then, we group `yago2` by `predicate` and take the count of each group to obtain `count_data`.

Finally, we sort the data by `count` and store the result in `/path/to/output/file`, which is path in HDFS.

Using `limit` or `rank` while creating `sorted_data` results in a deserialization error. Thus, after executing the script, we can obtain the top three predicates as follows:

```
hdfs dfs -cat /path/to/output/file | head -n 3
```

The output is shown below:

```
<isCitizenOf> 2141725
<hasFamilyName> 2002574
<hasGivenName> 1984813
```

Solution 2 - Script

```
yago_data = LOAD '/input/yago_full_clean.tsv'
    USING PigStorage(' ')
    AS (subject:chararray,
        predicate:chararray,
        object:chararray
    );

livesIn_data = FILTER yago_data BY predicate == '<livesIn>';

livesIn_count = FOREACH (
        GROUP livesIn_data
```

```
        BY subject
    )
    GENERATE group AS subject,
    COUNT(livesIn_data.predicate) AS count;

multiple_livesIn = FILTER livesIn_count BY count > 1;

person_givenName = JOIN multiple_livesIn
    BY subject, yago_data BY subject;

person_givenName = FILTER person_givenName
    BY yago_data::predicate == '<hasGivenName>';

person_givenName = FOREACH person_givenName
    GENERATE yago_data::object AS given_name;

person_givenName_distinct = DISTINCT person_givenName;

STORE person_givenName_distinct
INTO '/path/to/output/file'
USING PigStorage();

DUMP person_givenName_distinct;
```

We start by loading the dataset into `yago_data`. We then filter the data to obtain those those tuples which have `livesIn` as their `predicate` (`livesIn_data`).

Using this, we group the data by `subject`, calculate the number of tuples in each group and then filter by the count to obtain those tuples which are associated with more than one instance of the `livesOn` predicate (`multiple_livesIn`).

This is joined with the original dataset using `subject` and those tuples which have `hasGivenName` as their predicate are filtered out (`person_givenName`). Finally, we extract each distinct `object` to obtain the desired result.

The result is stored in `/path/to/output/file`, a filepath on HDFS.

A snippet of the output is shown below:

```
<Victor-Mugurel>
<Balasubramanian>
<Claire-Hélène>
<François-Henri>
<Alexis-François>
<Charles-Édouard>
<Florin-Alexandru>
```

Problem 2: UDF

Problem-2: UDF

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

a. Group the students based on the group-id

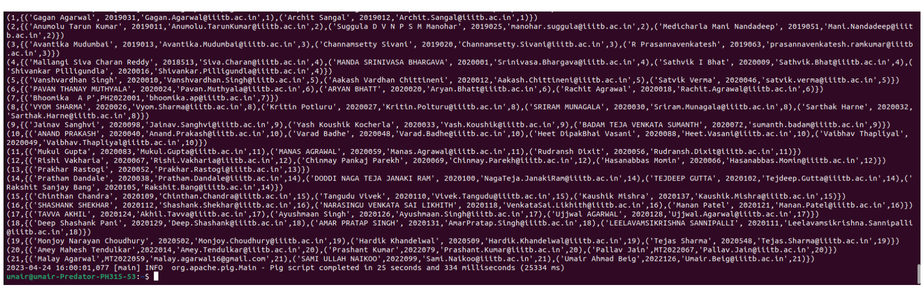
b. For each group create a new attribute named "project tool" and assign randomly a value from the list [MR, Pig, Hive, MongoDB]. Define a user defined function for this purpose. (5 marks)

Solution (a) - Script

```
students = LOAD '/input/sample.txt'
    USING PigStorage('\t')
    AS (name:chararray,
        rollnumber:chararray,
        email:chararray,
        groupid:chararray
    );

grouped = GROUP students BY groupid;

STORE grouped
INTO '/path/to/output/file'
USING PigStorage();

DUMP grouped;
```

We load the dataset and then group it by `groupid`. The result is stored in `/path/to/output/file`, a filepath on the HDFS.

A snippet of the output is shown below:



```
(1,{('Gagan Agarwal', 2019031,'Gagan.Agarwal@iiitb.ac.in',1),('Archit Sangal', 2019012,'Archit.Sangal@iiitb.ac.in',1)})
(2,{('Anumolu Tarun Kumar', 2019011,'Anumolu.TarunKumar@iiitb.ac.in',2),('Suggula D V N P S M Manohar', 2019025,'manohar.suggula@iiitb.ac.in',2),('Medicharla Mani Nandadeep', 2019051,'Mani.Nandadeep@iiit
b.ac.in',2)})
(3,{('Avantika Mudumbai', 2019013,'Avantika.Mudumbai@iiitb.ac.in',3),('Channamsetty Sivani', 2019020,'Channamsetty.Sivani@iiitb.ac.in',3),('R Prasannavenkatesh', 2019063,'prasannavenkatesh.ramkumar@iiitb
.ac.in',3)})
(4,{('Mallangi Siva Charan Reddy', 2018513,'Siva.Charan@iiitb.ac.in',4),('MANDA SRINIVASA BHARGAVA', 2020001,'Srinivasa.Bhargava@iiitb.ac.in',4),('Sathvik I Bhat', 2020009,'Sathvik.Bhat@iiitb.ac.in',4),(
'Shivankar Pilligundla', 2020016,'Shivankar.Pilligundla@iiitb.ac.in',4)})
(5,{('Vanshvardhan Singh', 2020010,'Vanshvardhan.Singh@iiitb.ac.in',5),('Aakash Vardhan Chittineni', 2020012,'Aakash.Chittineni@iiitb.ac.in',5),('Satvik Verma', 2020046,'satvik.verma@iiitb.ac.in',5)})
(6,{('PAVAN THANAY MUTHYALA', 2020024,'Pavan.Muthyala@iiitb.ac.in',6),('ARYAN BHATT', 2020020,'Aryan.Bhatt@iiitb.ac.in',6),('Rachit Agrawal', 2020018,'Rachit.Agrawal@iiitb.ac.in',6)})
(7,{('Bhoomika  A P',PH2022001,'bhoomika.ap@iiitb.ac.in',7)})
(8,{('VYOM SHARMA', 2020026,'Vyom.Sharma@iiitb.ac.in',8),('Kritin Potluru', 2020027,'Kritin.Polturu@iiitb.ac.in',8),('SRIRAM MUNAGALA', 2020030,'Sriram.Munagala@iiitb.ac.in',8),('Sarthak Harne', 2020032,
'Sarthak.Harne@iiitb.ac.in',8)})
(9,{('Jainav Sanghvi', 2020098,'Jainav.Sanghvi@iiitb.ac.in',9),('Yash Koushik Kocherla', 2020033,'Yash.Koushik@iiitb.ac.in',9),('BADAM TEJA VENKATA SUMANTH', 2020072,'sumanth.badam@iiitb.ac.in',9)})
(10,{('ANAND PRAKASH', 2020040,'Anand.Prakash@iiitb.ac.in',10),('Varad Badhe', 2020048,'Varad.Badhe@iiitb.ac.in',10),('Heet DipakBhai Vasani', 2020088,'Heet.Vasani@iiitb.ac.in',10),('Vaibhav Thapliyal',
2020049,'Vaibhav.Thapliyal@iiitb.ac.in',10)})
(11,{('Mukul Gupta', 2020083,'Mukul.Gupta@iiitb.ac.in',11),('MANAS AGRAWAL', 2020059,'Manas.Agrawal@iiitb.ac.in',11),('Rudransh Dixit', 2020056,'Rudransh.Dixit@iiitb.ac.in',11)})
(12,{('Rishi Vakharia', 2020067,'Rishi.Vakharia@iiitb.ac.in',12),('Chinmay Pankaj Parekh', 2020069,'Chinmay.Parekh@iiitb.ac.in',12),('Hasanabbas Momin', 2020066,'Hasanabbas.Momin@iiitb.ac.in',12)})
(13,{('Prakhar Rastogi', 2020052,'Prakhar.Rastogi@iiitb.ac.in',13)})
(14,{('Pratham Dandale', 2020038,'Pratham.Dandale@iiitb.ac.in',14),('DODDI NAGA TEJA JANAKI RAM', 2020100,'NagaTeja.JanakiRam@iiitb.ac.in',14),('TEJDEEP GUTTA', 2020102,'Tejdeep.Gutta@iiitb.ac.in',14),('
Rakshit Sanjay Bang', 2020105,'Rakshit.Bang@iiitb.ac.in',14)})
(15,{('Chinthan Chandra', 2020109,'Chinthan.Chandra@iiitb.ac.in',15),('Tangudu Vivek', 2020110,'Vivek.Tangudu@iiitb.ac.in',15),('Kaushik Mishra', 2020137,'Kaushik.Mishra@iiitb.ac.in',15)})
(16,{('SHASHANK SHEKHAR', 2020112,'Shashank.Shekhar@iiitb.ac.in',16),('NARASINGU VENKATA SAI LIKHITH', 2020118,'VenkataSai.Likhith@iiitb.ac.in',16),('Manan Patel', 2020121,'Manan.Patel@iiitb.ac.in',16)})
(17,{('TAVVA AKHIL', 2020124,'Akhil.Tavva@iiitb.ac.in',17),('Ayushmaan Singh', 2020126,'Ayushmaan.Singh@iiitb.ac.in',17),('Ujjwal AGARWAL', 2020128,'Ujjwal.Agarwal@iiitb.ac.in',17)})
(18,{('Deep Shashank Pani', 2020129,'Deep.Shashank@iiitb.ac.in',18),('AMAR PRATAP SINGH', 2020131,'AmarPratap.Singh@iiitb.ac.in',18),('LEELAVAMSIKRISHNA SANNIPALLI', 2020111,'Leelavamsikrishna.Sannipalli
@iiitb.ac.in',18)})
(19,{('Monjoy Narayan Choudhury', 2020502,'Monjoy.Choudhury@iiitb.ac.in',19),('Hardik Khandelwal', 2020509,'Hardik.Khandelwal@iiitb.ac.in',19),('Tejas Sharma', 2020548,'Tejas.Sharma@iiitb.ac.in',19)})
(20,{('Amey Mahesh Tendulkar',2022014,'Amey.Tendulkar@iiitb.ac.in',20),('Prashant Kumar',2022079,'Prashant.Kumar@iiitb.ac.in',20),('Pallav Jain',MT2022067,'Pallav.Jain@iiitb.ac.in',20)})
(21,{('Malay Agarwal',MT2022059,'malay.agarwal16@gmail.com',21),('SAMI ULLAH NAIKOO',2022099,'Sami.Naikoo@iiitb.ac.in',21),('Umair Ahmad Beig',2022126,'Umair.Beig@iiitb.ac.in',21)})
2023-04-24 16:00:01,077 [main] INFO  org.apache.pig.Main - Pig script completed in 25 seconds and 334 milliseconds (25334 ms)
umair@umair-Predator-PH315-53:~$
```

Solution (b) - Script

Creating a UDF requires creating a Java class which extends the `org.apache.pig.EvalFunc` class and overrides the `exec(Tuple input)` method of the class:

```java
import java.io.IOException;
import java.util.Random;
import org.apache.pig.EvalFunc;
import org.apache.pig.data.Tuple;

public class GroupId extends EvalFunc<String> {
    private static final String[] projectTools = {
        "MR", "Pig", "Hive", "MongoDB"
    };
    private static final Random random = new Random();

    public String exec(Tuple input) throws IOException {
        if (input == null || input.size() == 0) {
            return null;
        }
        try {
            return projectTools[
                random.nextInt(projectTools.length)
            ];
        }
        catch (Exception e) {
            throw new IOException(
                "Caught exception processing input row ", e
            );
        }
    }
}
```

We extend the `EvalFunc` class and implement the `exec()` method to return a random project tool from `projectTools`. To access this UDF in a `pig` script, we need to create a JAR file, which can be accomplished using tools like Maven or Gradle.

The `pig` script that uses this UDF is shown below:

```
REGISTER '/path/to/jar/file';

DEFINE RandomProjectTool org.example.GroupId();

students = LOAD '/input/sample.txt'
    USING PigStorage('\t')
    AS (name:chararray,
        rollnumber:chararray,
        email:chararray,
        groupid:chararray
```

```
    );
grouped_students = GROUP students BY groupid;

students_with_project_tool = FOREACH grouped_students {
    project_tool = RandomProjectTool();
    GENERATE FLATTEN(students), project_tool AS project_tool;
}

STORE students_with_project_tool
INTO '/path/to/output/file'
USING PigStorage();

DUMP students_with_project_tool;
```

We start by registering the JAR file so that `pig` can load it appropriately when the script is executed. Next, we assign an instance of the `GroupId` class to `RandomProjectTool`. We then load the data and group it by `groupid`. To assign a random project tool, we then go over each group, generate a random project tool for the group using `RandomProjectTool`, unpack the `students` in the group and combine them with the random project tool.

The result is stored in `/path/to/output/file`, a filepath on HDFS.

A snippet of the output is shown below:

```
'Gagan Agarwal' 2019031 'Gagan.Agarwal@iiitb.ac.in' 1 MongoDB
'Archit Sangal' 2019012 'Archit.Sangal@iiitb.ac.in' 1 MongoDB
'Satvik Verma'  2020046 'satvik.verma@iiitb.ac.in' 5 Pig
'VYOM SHARMA'  2020026 'Vyom.Sharma@iiitb.ac.in' 8 MR
'Pallav Jain' MT2022067 'Pallav.Jain@iiitb.ac.in' 20 Hive
```

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)

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 HiveQL. (4 points)

Loading the data

The steps to load the dataset in Hive are as follows:

- Make sure all Hadoop services are running:

```
start-all.sh
```

- Add the dataset to the Hadoop file system.

```
hdfs dfs -mkdir /input
hdfs dfs -put /path/to/dataset /input
```

- Run the Hive server:

```
hiveserver2
```

- In another terminal window, login to `beeline`:

```
beeline -n user -u jdbc:hive2://
```

  Here, `<user>` is the user set up to run Hadoop.

- Create an external table using the following query:

```
CREATE EXTERNAL TABLE IF NOT EXISTS yago_file(
    subject STRING,
    predicate STRING,
    object STRING
)
ROW FORMAT DELIMITED FIELDS TERMINATED BY ' '
STORED AS TEXTFILE LOCATION '/input';
```

Here, we specify that we want to create an external table called `yago_file` which has three string columns `subject`, `predicate` and `object`. The table should reference the file(s) located in the `/input` directory in Hadoop and each field in each row of the file(s) is separated by a space character.

Solution 1

The query is as follows:

```
SELECT predicate,
    COUNT(predicate) as count
FROM yago_file
GROUP BY predicate
ORDER BY COUNT(predicate) DESC
LIMIT 3;
```

We group the data by `predicate` and order it in descending order of the counts of the `predicate`. Finally, we limit the result to 3 using `limit` so that we get the top three predicates.

The output is as follows:

predicate	**count**
<isCitizenOf>	2141725
<hasFamilyName>	2002574
<hasGivenName>	1984813

Solution 2

The query is as follows:

```
SELECT b.object AS given_name
FROM (
        SELECT y.subject
        FROM yago_file y
        WHERE y.predicate = '<livesIn>'
        GROUP BY y.subject
        HAVING COUNT(*) > 1
    ) a
    JOIN yago_file b ON b.subject = a.subject
    AND b.predicate = '<hasGivenName>'
GROUP BY b.object;
```

Using the sub-query, we first obtain all the subjects that are associated with at least two instances of the `livesIn` predicate.

Using that as the table in the `FROM` clause, we join it with `yago_file` on the `subject` column and with the additional condition that `predicate` is `hasGivenName`.

This gives us all subjects that are associated with at least two instances of the `livesIn` predicate and are also associated with the `hasGivenName` predicate. Finally, we group by `object` so that only unique objects are shown.

A snippet of the output is shown below:

given_name
<Émile>
<Émilie>
<Éric>
<Érik>
<Íñigo>
<Ömer>

There are 6,020 rows in total in the result.

Problem 2: Partitioning and Bucketing

Problem-2: 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:

(i) by considering partitioning and bucketing;

(ii) by considering partitioning but not bucketing;

(iii) by considering neither partitioning nor bucketing.

- For the first case alone perform a Bucketized Merge-Join by enabling the necessary parameters (see the note below).
- Compare the run time of the three cases by performing your experiments on your local system.

Creating the yago Table

To create and insert data into the `yago` table, we can use the `yago_file` external table created for Problem 1. The steps are as follows:

- Create the `yago` table:

```
CREATE TABLE yago(
    subject STRING,
    predicate STRING,
    object STRING
);
```

- Insert data into `yago` using `yago_file`:

```
INSERT OVERWRITE TABLE yago SELECT * FROM yago_file;
```

Query

The query for the desired result is as follows:

```
SELECT a.subject AS x,
    a.object AS y,
    b.object AS z
FROM (
        SELECT subject,
            object
        FROM table_name
```

```sql
        WHERE predicate = "<hasGivenName>"
    ) a
    JOIN (
        SELECT subject,
            object
        FROM table_name
        WHERE predicate = "<livesIn>"
    ) b ON a.subject = b.subject;
```

Here, `table_name` is the name of the table the query should be run on.

In the first sub-query after the `FROM` clause, we obtain all `(subject, object)` tuples which have `hasGivenName` as their predicate. In the second sub-query, we obtain all tuples which have `livesIn` as their predicate. Finally, we join the two resultant tables on the `subject` column.

This returns 67,028 rows.

A snippet of the output is shown below:

x	y	z
<Đorđe_Branković_(count)>	<Đorđe>	<Vienna>
<Đorđe_Branković_(count)>	<Đorđe>	<Cheb>
<Ġużè_Ellul_Mercer>	<Ġużè>	<Msida>
<Ľudmila_Cervanová>	<Ľudmila>	<Slovakia>
<Ľudmila_Cervanová>	<Ľudmila>	<Piešťany>

By Considering Partitioning and Bucketing

The steps to create the table and execute the query are as follows:

- Create the table `yago_buck_part` as follows:

  ```sql
  CREATE TABLE yago_buck_part(
      subject STRING,
      object STRING
  )
  PARTITIONED BY (predicate STRING)
  CLUSTERED BY (subject)
  SORTED BY (subject ASC) INTO 10 BUCKETS;
  ```

- Insert the data (statically) into the table for each predicate. Use an `.hql` file with all the insert statements and use the `run` command on `beeline`:

  ```
  !run /path/to/hql/file
  ```

  A generic insert statement is shown below:

```sql
INSERT OVERWRITE TABLE yago_buck_part
PARTITION(predicate = "<predicate>")
SELECT subject,
    object
FROM yago
WHERE predicate = "<predicate>";
```

Here, `<predicate>` is the predicate which we want to insert into the specified partition.

- Enable bucketized merge-join:

```sql
set hive.auto.convert.sortedmerge.join=true;
set hive.optimize.bucketmapjoin=true;
set hive.optimize.bucketmapjoin.sortedmerge=true;
```

- Execute the query:

```sql
SELECT a.subject AS x,
a.object AS y,
b.object AS z
FROM (
    SELECT subject,
        object
    FROM yago_buck_part
    WHERE predicate = "<hasGivenName>"
) a
JOIN (
    SELECT subject,
        object
    FROM yago_buck_part
    WHERE predicate = "<livesIn>"
) b ON a.subject = b.subject;
```

By Considering Partitioning but not Bucketing

- Create the table `yago_part` as follows:

```sql
CREATE TABLE yago_part(
    subject STRING,
    object STRING
)
PARTITIONED BY (predicate STRING);
```

- Insert the data, similar to the step above but changing the table name from `yago_buck_part` to `yago_part`.
- Turn off bucketized merge-join:

```
set hive.auto.convert.sortedmerge.join=false;
set hive.optimize.bucketmapjoin=false;
set hive.optimize.bucketmapjoin.sortedmerge=false;
```

- Execute the query:

```
SELECT a.subject AS x,
a.object AS y,
b.object AS z
FROM (
    SELECT subject,
        object
    FROM yago_part
    WHERE predicate = "<hasGivenName>"
) a
JOIN (
    SELECT subject,
        object
    FROM yago_part
    WHERE predicate = "<livesIn>"
) b ON a.subject = b.subject;
```

By Considering Neither Partitioning nor Bucketing

- Turn off bucketized merge-join:

```
set hive.auto.convert.sortedmerge.join=false;
set hive.optimize.bucketmapjoin=false;
set hive.optimize.bucketmapjoin.sortedmerge=false;
```

- The query is executed directly on `yago`:

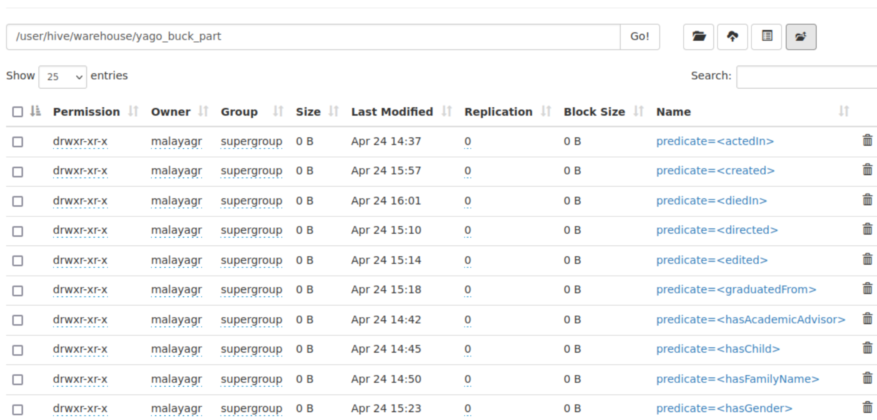
```
SELECT a.subject AS x,
a.object AS y,
b.object AS z
FROM (
    SELECT subject,
        object
    FROM yago
    WHERE predicate = "<hasGivenName>"
) a
JOIN (
    SELECT subject,
        object
    FROM yago
    WHERE predicate = "<livesIn>"
) b ON a.subject = b.subject;
```

Performance Comparison

The time taken by each of the methods is shown below:

Method	Time (s)
By Considering Partitioning and Bucketing	80.026
By Considering Partitioning but not Bucketing	**76.061**
By Considering Neither Partitioning nor Bucketing	175.336

When we use partitions, Hive automatically creates different directories in HDFS for each partition. In this case, the file structure generated is as shown:



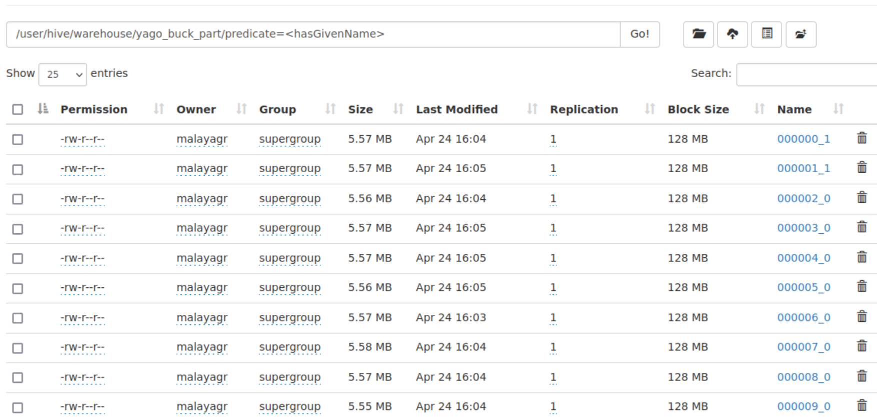
/user/hive/warehouse/yago_buck_part Go!

Show 25 entries Search:

Permission	Owner	Group	Size	Last Modified	Replication	Block Size	Name
drwxr-xr-x	malayagr	supergroup	0 B	Apr 24 14:37	0	0 B	predicate=<actedIn>
drwxr-xr-x	malayagr	supergroup	0 B	Apr 24 15:57	0	0 B	predicate=<created>
drwxr-xr-x	malayagr	supergroup	0 B	Apr 24 16:01	0	0 B	predicate=<diedIn>
drwxr-xr-x	malayagr	supergroup	0 B	Apr 24 15:10	0	0 B	predicate=<directed>
drwxr-xr-x	malayagr	supergroup	0 B	Apr 24 15:14	0	0 B	predicate=<edited>
drwxr-xr-x	malayagr	supergroup	0 B	Apr 24 15:18	0	0 B	predicate=<graduatedFrom>
drwxr-xr-x	malayagr	supergroup	0 B	Apr 24 14:42	0	0 B	predicate=<hasAcademicAdvisor>
drwxr-xr-x	malayagr	supergroup	0 B	Apr 24 14:45	0	0 B	predicate=<hasChild>
drwxr-xr-x	malayagr	supergroup	0 B	Apr 24 14:50	0	0 B	predicate=<hasFamilyName>
drwxr-xr-x	malayagr	supergroup	0 B	Apr 24 15:23	0	0 B	predicate=<hasGender>

Due to each partition being in a separate directory, Hive only has to only look at a subset of the data when a query refers to the partition. In this case, we refer to the two partitions `hasGivenName` and `livesIn`. Thus, Hive directly accesses the data stored in the directories for these two partitions instead of scanning the entire data twice to find all matching tuples. This greatly speeds up processing, as is evident from the table above. Both the partitioning-based methods are ~57% faster than the non-partitioning based solutions.

On the other hand, partitioning without bucketing is ~5% faster than partitioning with bucketing. In bucketing, each partition is further divided into a fixed number of buckets (here 10). For example:



/user/hive/warehouse/yago_buck_part/predicate=<hasGivenName> Go!

Show 25 entries Search:

Permission	Owner	Group	Size	Last Modified	Replication	Block Size	Name
-rw-r--r--	malayagr	supergroup	5.57 MB	Apr 24 16:04	1	128 MB	000000_1
-rw-r--r--	malayagr	supergroup	5.57 MB	Apr 24 16:05	1	128 MB	000001_1
-rw-r--r--	malayagr	supergroup	5.56 MB	Apr 24 16:04	1	128 MB	000002_0
-rw-r--r--	malayagr	supergroup	5.57 MB	Apr 24 16:05	1	128 MB	000003_0
-rw-r--r--	malayagr	supergroup	5.57 MB	Apr 24 16:05	1	128 MB	000004_0
-rw-r--r--	malayagr	supergroup	5.56 MB	Apr 24 16:05	1	128 MB	000005_0
-rw-r--r--	malayagr	supergroup	5.57 MB	Apr 24 16:03	1	128 MB	000006_0
-rw-r--r--	malayagr	supergroup	5.58 MB	Apr 24 16:04	1	128 MB	000007_0
-rw-r--r--	malayagr	supergroup	5.57 MB	Apr 24 16:04	1	128 MB	000008_0
-rw-r--r--	malayagr	supergroup	5.55 MB	Apr 24 16:04	1	128 MB	000009_0

This allows Hive to use a hash function to quickly figure out which bucket to access for some piece of data. Since each bucket is even smaller than the overall partition, processing is even faster. It is most beneficial when the data is evenly distributed among buckets. In this case, it is unlikely that the data is evenly distributed since subjects are not evenly distributed. This might explain why it is slower than only partitioning.