

Hands-On Experiment 2-2: Data Warehousing with Hive

Instructor: Dr. Peng Kang

Introduction

We are using the Truck sensor data to understand better the risk associated with every driver. This hands-on exercise will teach you how to analyze data to find risks using Apache Hive.

Objectives

In this Hands-on exercise, you will learn

1. Practice PySpark SQL for data analytics.
2. Use enhanced aggregation to emulate SQL concepts like GROUPING SETS, ROLLUP, and CUBE in PySpark.
3. Analyzing Driver Risk factor
4. Analyzing data using Data Warehousing/OLAP functions in Hive

Prerequisites:

Before you begin, Upload "geolocation.csv", "trucks_extended.csv", and "apache-hive-3.1.3-bin.tar" to Google Colab. Make sure you have access to Hive and PySpark in your Google Colab environment:

```
!apt-get install openjdk-8-jdk-headless -qq > /dev/null

!wget https://downloads.apache.org/hadoop/common/hadoop-3.3.6/hadoop-3.3.6.tar.gz
!tar -xzf hadoop-3.3.6.tar.gz

!tar xf apache-hive-3.1.3-bin.tar

import os
os.environ["JAVA_HOME"] = "/usr/lib/jvm/java-8-openjdk-amd64"
os.environ["HADOOP_HOME"] = "/content/hadoop-3.3.6"
os.environ["HIVE_HOME"] = "/content/apache-hive-3.1.3-bin"
```

1 Preparation

We will use several tables, such as geolocation or trucks which were created in an earlier hands-on exercise.

If we were using a permanent installation (e.g., a physical cluster) of Hive, we could still access our database and its tables, even after we restarted our session. However, we are using google colab for

educational purposes, and for this reason, our session gets restarted after a while, with all of its data being automatically purged by google. This means that the next time we want to work on this test data set, we need to recreate the data tables.

If it's been a while since you last executed the previous hands-on exercise, you may have lost your hive database instance, along with the tables that we created in it. Fortunately, it is a rather easy process to recreate these tables, once we have our notebooks created.

Before we continue, let's check if our data tables are purged.

1.1 Check if your tables are available

1. Check for Hive databases:

```
!$HIVE_HOME/bin/hive -e 'show databases;'
```

You should see the *default* database

2. Let's select the default database:

```
!$HIVE_HOME/bin/hive -e 'use default;'
```

3. Check if your tables still exist:

```
!$HIVE_HOME/bin/hive -e 'show tables;'
```

You should see a list of the tables in the default database:

```
<your-NetID>\_geolocation  
<your-NetID>\_trucks
```

In the case you don't see these tables, we need to recreate them. Otherwise, skip the following section

1.1.1 OPTIONAL: Let's re-create the tables from the previous hands-on exercise

Let us create a copy of our previous exercise (EX-2-1) and start from the last cell of that exercise: In colab, go to File and choose Save a copy in Drive. You can rename this new notebook to reflect Exercise 2-2 Data warehousing with Hive.

You can comment out (add a # at the start of each line of code) cells that do not perform essential tasks. These may include cells that used to "show tables" or "describe" the tables. **Keep in mind that creation of the tables, and loading data into them are our essential tasks, and you need to keep those cells active.**

Go to the last cell in your new notebook, select the *Runtime menu* and *Restart runtime*. Next, select the *Runtime menu* and *Run all*. Once all cells are executed, you should have a fresh copy of your database populated with your tables.

Attention: if you were uploading your input csv files to colab using the file upload command, you will need to re-upload them, or else the execution will be stuck there until you upload the files.

Check if your tables are there:

```
!$HIVE_HOME/bin/hive -e 'show tables;'
```

You should see a list of the tables in the default database:

```
<your-NetID>\_geolocation  
<your-NetID>\_trucks
```

2 Enhanced Aggregation with Grouping Sets (0.5 points)

We want to analyze the Geolocation table to show how a trucking company can analyze geolocation data to reduce fuel costs and improve driver safety.

2.1 Dimensional Aggregation

ATTENTION: if you haven't already populated your geolocation table with data, you may receive errors, or no output. Make sure that you have uploaded your data first:

```
!$HIVE_HOME/bin/hive -e "LOAD DATA LOCAL INPATH 'geolocation.csv' INTO TABLE default.<your-NetID>_geolocation;"
```

In the previous hands-on exercise, we used the GROUP BY operation to perform aggregations in our queries. Now we want to calculate how many events occurred according to the event types, e.g., normal, overspeed, lane departure, etc., to each driver, as well as the total number of events that occurred per driver. This will not be possible in a single GROUP BY statement. One easy way to do it is by connecting several GROUP BY result sets with UNION ALL.

Let's do it one by one. First, run below HiveQL query to calculate how many events occurred according to the event types of each driver.

```
# Calculate how many events occurred according to the event types, e.g., normal, overspeed, lane departure, etc., to each driver

%%bash
$HIVE_HOME/bin/hive -e "
SELECT driverid, event, count(*)
FROM default.<your-NetID>_geolocation
GROUP BY driverid, event;
"
```

It shows the driver's id, event type, and a total number of events. It took about 10 sec. to be calculated.

Next, run below HiveQL query to calculate the total number of events that occurred per driver.

```
# Calculate total events that occurred per drivers

%%bash
$HIVE_HOME/bin/hive -e "
SELECT driverid, count(*)
FROM default.<your-NetID>_geolocation
GROUP BY driverid;
"
```

It shows the driver's id and the total number of events. It took about 10 sec. to be calculated.

Finally, let's try to connect both GROUP BY result sets with UNION ALL.

ATTENTION: For a conventional installation (i.e., not colab), we could run a simple command as below:

```
# Connecting both GROUP BY result sets with UNION ALL

%%bash
$HIVE_HOME/bin/hive -e "
SELECT driverid, event, count(*)
FROM default.<your-NetID>_geolocation
GROUP BY driverid, event

UNION ALL

SELECT driverid, null as event, count(*)
FROM default.<your-NetID>_geolocation
GROUP BY driverid;
"
```

However, *you may experience errors if you wanted to do this operation in colab*, due to the system limitations. Instead, we can create two temporary tables, store the query results in them, and then union them. This is a useful little hack that would help you, when you are dealing with even larger datasets.

```
# Connecting both GROUP BY result sets with UNION ALL

%%bash
```

```
$HIVE_HOME/bin/hive -e "
-- Create temporary table for the first query
CREATE TEMPORARY TABLE temp1 AS
SELECT driverid, event, count(*)
FROM default.<Your-NetID>_geolocation
GROUP BY driverid, event;

-- Create temporary table for the second query
CREATE TEMPORARY TABLE temp2 AS
SELECT driverid, CAST(null AS STRING) as event, count(*)
FROM default.<Your-NetID>_geolocation
GROUP BY driverid;

SELECT * FROM temp1
UNION ALL
SELECT * FROM temp2;
"
```

It shows the driver's ID, event type, and the total number of events from both query result sets. It takes longer than previous queries to run, because it reads the datasets twice. If we need to analyze Big data, it could be a problem. This is where GROUPING SETS come in.

2.2 GROUPING SETS

Hive has offered the GROUPING SETS keywords to implement advanced multiple GROUP BY operations against the same set of data. GROUPING SETS is a shorthand way of connecting several GROUP BY result sets with UNION ALL.

The GROUPING SETS keyword completes all processes in one stage of jobs, which is more efficient than GROUP BY and UNION ALL having multiple stages. The GROUPING SETS clause in GROUP BY allows us to specify more than one GROUP BY option in the same record set. All GROUPING SET clauses can be logically expressed in terms of several GROUP BY queries connected by UNION.

Let's run the same query with the GROUPING SETS clause.

```
#Using GROUPING SETS
%%bash
$HIVE_HOME/bin/hive -e "
SELECT driverid, event, count(*) as occurrence
FROM default.<your-NetID>_geolocation
GROUP BY driverid, event
GROUPING SETS ((driverid, event), driverid);
"
```

It took about 10 seconds to calculate it, which is much faster than the above one, about 20 sec, when we use multiple GROUP BY and UNION ALL queries.

We calculated how many events occurred according to the event types of each driver and the total events that occurred per driver. Where the event column is null, we have the total sum of events that occurred to a driver across all event types. For example, 80 events occurred to drive A1, and most (77) of them are normal. However, two lane departure events and one unsafe tail distance event occurred. Figure 1 shows the results in Hive Editor of the HUE interface for better presentation.

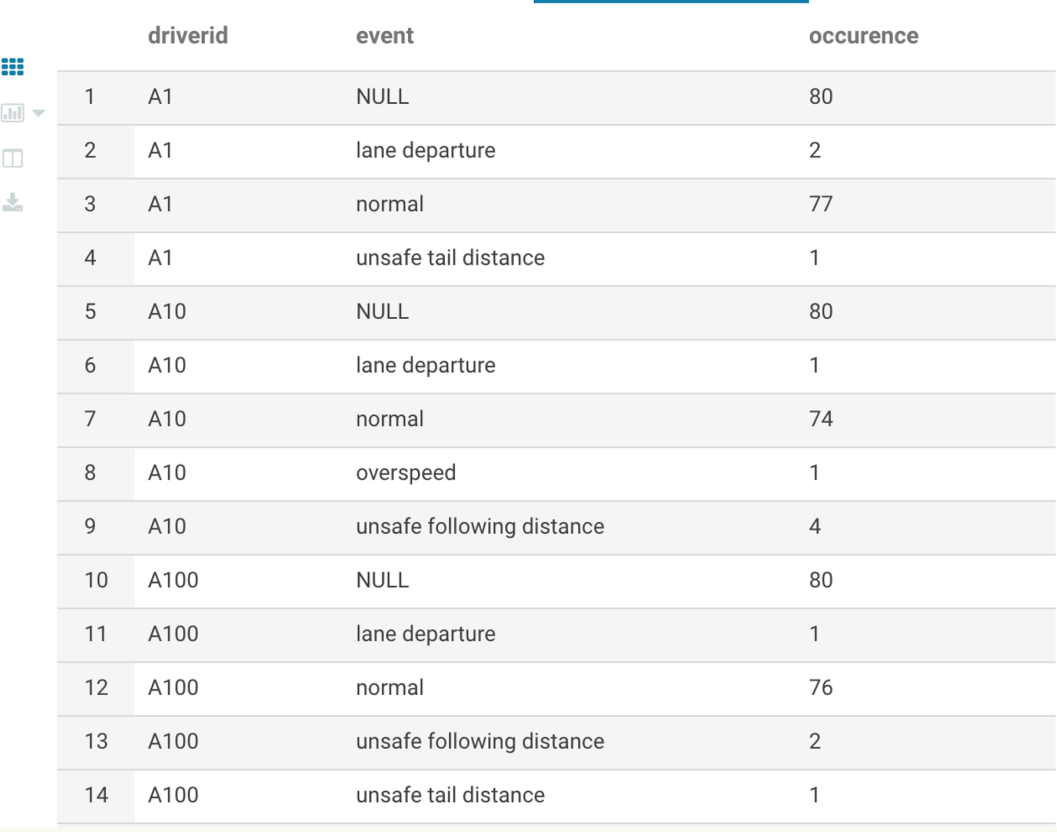
3 Analyzing Driver Risk factor (0.5 points)

We also want to analyze risks associated with drivers in detail. We will use event and mileage records to calculate the driver's risk factor.

3.1 Non-normal events

First, we will filter normal records from the events records (geolocation table), then create the unusual_event table. Run the below query.

Showing the above results in the Hive Editor of the HUE interface.



	driverid	event	occurence
1	A1	NULL	80
2	A1	lane departure	2
3	A1	normal	77
4	A1	unsafe tail distance	1
5	A10	NULL	80
6	A10	lane departure	1
7	A10	normal	74
8	A10	overspeed	1
9	A10	unsafe following distance	4
10	A100	NULL	80
11	A100	lane departure	1
12	A100	normal	76
13	A100	unsafe following distance	2
14	A100	unsafe tail distance	1

Figure 1: GROUPING SETS query results

```
%%bash
$HIVE_HOME/bin/hive -e "
CREATE TABLE default.<your-NetID>_unusual_events
STORED AS ORC
AS
SELECT driverid, count(*) as occurrence
FROM default.<your-NetID>_geolocation
WHERE event != 'normal'
GROUP BY driverid;
"
```

The resulting table will have a count of total unusual or non-normal events associated with each driver.

```
%%bash
$HIVE_HOME/bin/hive -e "
SELECT *
FROM default.<your-NetID>_unusual_events
LIMIT 5;
"
```

3.2 Perform JOIN operation

Let's create the driver_total_mileage table. The driver_total_mileage table has the total miles traveled by each driver:

```
# Creating driver_total_mileage table
```

```
%%bash
$HIVE_HOME/bin/hive -e "
CREATE TABLE default.<your-NetID>_driver_total_mileage
STORED AS ORC
AS
SELECT driverid, sum(miles) totalmiles
FROM default.<your-NetID>_truck_mileage
GROUP BY driverid
DISTRIBUTE BY totalmiles
SORT BY totalmiles DESC;
"
```

NOTE: As you remember from the previous hands-on exercise, we used a trick to reduce the load on our colab instance, by using **DISTRIBUTE BY ...** and **SORT BY ...**. In a normal setting, we only needed to use **ORDER BY ...**.

Next, we will perform a JOIN operation. The `unusual_events` table has the driver's ID and count of their respective non-normal events.

```
# Creating joined table
%%bash
$HIVE_HOME/bin/hive -e "
CREATE TABLE default.<your-NetID>_joined
STORED AS ORC
AS
SELECT a.driverid, a.occurence, b.totalmiles
FROM <your-NetID>_unusual_events a
JOIN <your-NetID>_driver_total_mileage b
ON (a.driverid = b.driverid);
"
```

The resulting data set will give us a driver's total miles and the number of non-normal events.

```
#Display results
%%bash
$HIVE_HOME/bin/hive -e "
SELECT *
FROM default.<your-NetID>_joined
LIMIT 5;
"
```

3.3 Compute driver risk factor

We will associate a driver risk factor with every driver. The risk factor for each driver is the number of abnormal occurrences over the total number of miles per driver. Simply put, a high number of abnormal occurrences over a short number of miles driven is an indicator of high risk.

Let's translate this intuition into an HiveQL query. **NOTE:** I believe by now, you should have a good handle on this, but please remember that you need to add the necessary **bash** and **HIVE_HOME** directives before the following commands, just like previous prompts!

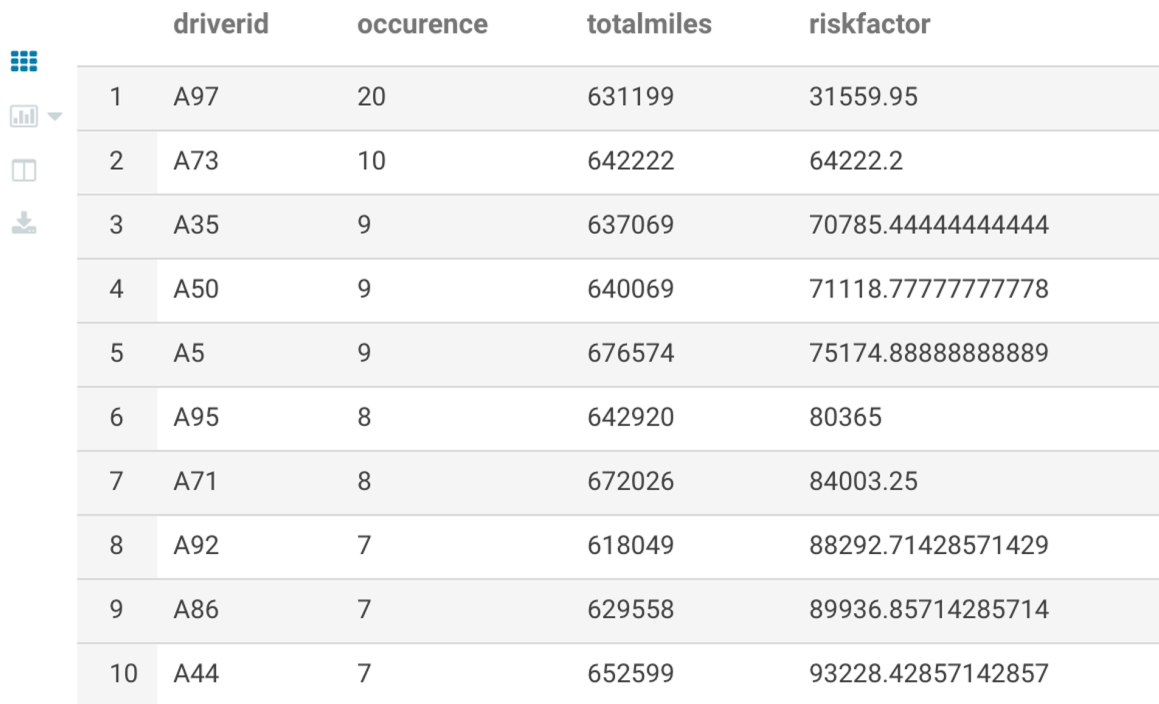
```
SELECT driverid, occurence, totalmiles, totalmiles/occurence AS riskfactor
FROM default.<your-NetID>_joined
ORDER BY riskfactor ASC;
```

Figure 2 shows the above results in the Hive Editor of the HUE interface.

4 Rollup and CUBE (3 points)

Answer all the questions and write scripts for each question. Submit your ipynb with explanations and all the intermediate results of running your queries.

We exercised GROUPING SETS in the exercise. There are additional ways to perform these aggregations, such as using CUBE or ROLLUP. These are almost like shortcuts. While CUBE returns all possible aggregation combinations, ROLLUP does it in a more hierarchical fashion. You will replace GROUPING SETS in the below query with ROLLUP, and then similarly replace GROUPING SETS with CUBE and see the differences.



	driverid	occurrence	totalmiles	riskfactor
1	A97	20	631199	31559.95
2	A73	10	642222	64222.2
3	A35	9	637069	70785.444444444444
4	A50	9	640069	71118.777777777778
5	A5	9	676574	75174.888888888889
6	A95	8	642920	80365
7	A71	8	672026	84003.25
8	A92	7	618049	88292.71428571429
9	A86	7	629558	89936.85714285714
10	A44	7	652599	93228.42857142857

Figure 2: Driver Risk Factor results: the inverse of abnormal occurrences over the total number of miles per driver: the smaller, the riskier!

HiveQL query, Let us call it grouping-set-query. Remember that you need to add the necessary bash and HIVE_HOME directives before the following commands,

```
SELECT driverid, event, city, count(*) as occurrence
FROM default.<your-NetID>_geolocation
GROUP BY driverid, event, city
GROUPING SETS ((driverid, event, city), (driverid, event), driverid)
```

- Q1. (1pt) Modify/rewrite the grouping-set-query in the example with ROLLUP (Let's call it rollup-query). Run it, check the results, and explain the differences.
 - Replace the GROUPING SETS part with ROLLUP:
 - * Delete GROUPING SETS line
 - * Add 'WITH ROLLUP' at the end of the GROUP BY line.
 For syntax, see this Apache Hive cwiki link
 - Show query results. The first line of the result should be like below
 - * NULL NULL NULL 8000
 - Explain the differences in the results between the grouping-set-query and the roll-up-query with your query results.
- Q2. (1pt) Modify/rewrite the rollup-query (or grouping-set-query) with GROUPING SETS (Let's call it rollup-like-query) to produce identical results of rollup-query
 - The query result should be the same.
 - Show query results
- Q3. (1pt) Modify/rewrite the grouping-set-query with CUBE. Run it, check the results, and explain the differences and similarities.
 - Explain the differences and similarities of the query results among GROUPING SETS, ROLLUP, and CUBE
 - Show query results