Hands-On Experiment 2-1: Introduction to Apache Spark

Instructor: Dr. Peng Kang

In this exercise, we will look at using Apache Spark and Apache Hive. The exercise is in two main sections. In Section 1, you will create an instance of Spark on Google Colab, and perform some basic operations with it. In Section 2, you will perform OLAP operations with Hive.

Section 1 (1 point): Step-by-Step Instructions on Setting up Spark on Google Colab

Spark provides a simple way to learn the API and a powerful tool to analyze data interactively. It is available in either Scala (which runs on the Java VM and is thus an excellent way to use existing Java libraries) or Python.

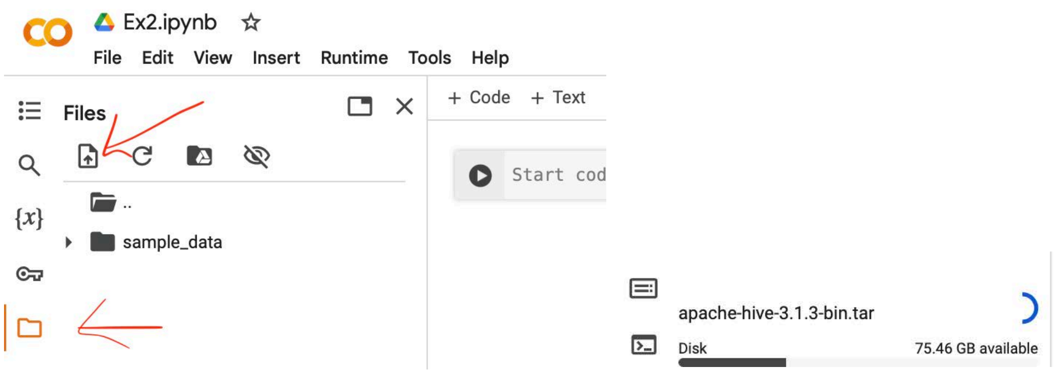
In this section, we will setup Apache Spark on Google Colab and perform a few simple exercises with it.

Step 1: Setting up Apache Spark 3.5.0 on Google Colab

1. Open Google Colab: `https://colab.research.google.com/`
2. Create a new notebook. Upload "geolocation.csv", "trucks_extended.csv", "spark-3.5.2-bin-hadoop3.tar", and "apache-hive-3.1.3-bin.tar" to Google Colab. You can follow the figure on the next page.

   Some files are large. Please be patient and wait for the upload to complete, as indicated by the blue circle.
3. Install the necessary dependencies for Apache Spark:
```
!apt-get install openjdk-8-jdk-headless -qq > /dev/null
!tar xf spark-3.5.2-bin-hadoop3.tgz
!pip install -q findspark
```
4. Set up the environment for Spark:
```
import os
os.environ["JAVA_HOME"] = "/usr/lib/jvm/java-8-openjdk-amd64"
os.environ["SPARK_HOME"] = "/content/spark-3.5.2-bin-hadoop3"
```
5. Initialize Spark:
```
import findspark
findspark.init()
from pyspark.sql import SparkSession
spark = SparkSession.builder.master("local[*]").getOrCreate()
```

Loading the Data



Step 2: Interactive Analysis with Spark

For our hands-on exercises, we will use the Spark shell for Python, commonly known as pyspark, through Google Colab. However, you may use any other languages supported by Spark: Scala, Python, Java, and R programming languages. Unfortunately, Spark doesn't provide any Spark shell for Java and R; only two Spark shells exist: Spark shell for Scala and Python. One more significant advantage of choosing Python is that it is the most popular language for Spark, Big data analytics, machine learning, and data mining. See this article

Now let's read the **geolocation.csv** file with spark. Importing data into a DataFrame using the DataFrame API provides easier access to data since it looks conceptually like a Table and many developers from Python/R/Pandas are familiar with it.

We can use the 'spark.read.text' command to create a DataFrame. This would allow us to see the dataset, but it would not be perfect, and all data will show up in one column. To get a better-formatted DataFrame, "CSV format" is a better option. Try the following command.

```
# Read the dataset into a Spark DataFrame named df
df = spark.read.csv("geolocation.csv", header=True, inferSchema=True)
```

Exploring the Data

For this hands-on exercise, you'll use the Trucking IoT Data. Note, of course, that this is 'small' data and that using Spark in this context might be overkill.

We are looking at a use case where we have a truck fleet. Each truck has been equipped to log location and event data. These events are streamed back to a data center where we will process the data. The company wants to use this data to understand risk better.

The dataset that we used in our previous exercises, Geolocation, contains the following files:

- geolocation.csv: This is the collected geolocation data from the trucks. It contains records showing the truck location, date, time, type of event, speed, etc.
- trucks.csv: This data was exported from a relational database and shows the information on truck models, driverid, truckid, and aggregated mileage info.

Now that the dataset is loaded, let's explore it. We can see the rows stored in the DataFrame by typing a show() action.

1. View the first 5 rows of the dataframe `df` to understand its structure:

   ```
   df.show(5)
   ```

2. To understand the data types of the columns and ensure they are correct:

   ```
   df.printSchema()
   ```

Filtering Data

For data analysis, it's common to filter out rows that are not relevant to the questions at hand. For example, let's repeat our previous exercise and filter out 'normal' events to only show the unsafe events:

```
df.select("truckid", "driverid", "event", "city").filter("event != 'normal'").show()
```

Grouping and Aggregating Data

Aggregating data helps in summarizing it for a clearer understanding. For instance, to count specific events:

```
df.groupBy("event").count().show()
```

Data Distribution

Understanding how the data is distributed is vital for insights. We calculated the five number summary in our previous exercises. Spark provides similar function as the one we saw in pandas, describe(). The function returns a DataFrame containing information such as the number of non-null entries (count), mean, standard deviation, and minimum and maximum values for each numerical column.

Spark gives standard deviation rather than showing quantiles and median. The reason is that median and quantiles are costly to compute on large data. Both values need data to be in sorted order and result in skewed calculations. However, we can calculate quartiles using the approxQuantile() method introduced in spark 2.0. This allows us to find 25%, median, and 75% values. (This will not be covered in this exercise)

Let's use Spark's `describe()` function to look at the five number summary for **velocity**:

```
df.describe("velocity").show()
```

Note: If there's a need to convert string columns to numeric for correct calculations, use the `cast` function as shown in the provided StackOverflow link.

Section 2 (1.5 points):
Data Warehousing with Hive

Apache Hive is a data warehouse software project built on top of Apache Hadoop for providing data queries and analysis. It provides a SQL interface to query data stored in various databases and file systems that integrate with Hadoop. Hive enables analysts familiar with SQL to run queries on large volumes of data. Hive has three main functions: data summarizing, query, and analysis. Hive provides tools that enable easy data extraction, transformation, and loading (ETL).

This exercise will use Hive as a data warehouse tool instead of Spark. In this section of the exercise, you will learn

- Create Hive tables and loading datasets
- Transformation (ETL) with Hive
- Simple Analysis with Hive

Setting up the Environment

Install Java: Hive relies on Java. You'll need to install it in your Colab environment, similar to what we did previously for Spark. This step is redundant, if you are continuing with your Spark exercise, but we are including it here for completeness, and for your future reference.

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

Download and Set Up Apache Hadoop and Hive: Hadoop is needed for Hive to operate.

```
!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
```

Setup Environment Variables: This helps in making Java and Hive commands accessible from the command line in the Colab notebook.

```
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"
```

Relational Database Management System (RDBMS)

When you initialize Hive for the first time, it needs a metastore, which is a place where it stores metadata about the tables, databases, and other Hive entities. Hive can use various RDBMS as its metastore. By default, Hive uses ***Derby*** as its metastore database in embedded mode. Derby, also known as Apache Derby, is an RDBMS that offers a small footprint and is suitable for testing and development; however, for production environments or for concurrent access, we need an external RDBMS (such as MySQL, PostgreSQL, Oracle, etc.).

Clean the Existing Metastore (If Any) Let's clear out any existing metastore data. Hive, by default with Derby, creates a metastore_db directory in the location from which you run the command. Let's delete this directory and then run the schema initialization command:

```
!rm -rf metastore_db
```

Initialize Derby the schema

```
!$HIVE_HOME/bin/schematool -initSchema -dbType derby
```

Check for Existing Databases

Let's take a look at the databases that are available in our Hive instance on Google Colab. Since we are accessing Hive CLI through colab, we need to use the "!" to inform the system that we are passing a command to the system. *Don't forget semi-colon, ';'*

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

We can see that we only have a *"default"* database.

Create Hive tables

Let's create and load tables for the geolocation and trucks data. In this part, we will learn how to use Hive to create two tables: geolocation and trucks.

We need to define the database that we would like to use for this purpose

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

We will create a 'trucks' table in our default database in Hive. We already know the schema of the trucks and geolocation. We will add some options, including (case insensitive)

-
  ```
  ROW FORMAT DELIMITED FIELDS TERMINATED BY ‘’, LINES TERMINATED BY \n # Define delimiters
  ```

-
  ```
  STORED AS TEXTFILE \# Specify a file format for a Hive table (e.g., textfile, Avro,
      parquet, etc.)
  ```

-
  ```
  TBLPROPERTIES('skip.header.line.count'='1') \# The dataset has a header in the file, so we
       'dont want to store the header as a row.
  ```

See details on various commands here. Continue on the next page!

1. Create trucks table Note: For this exercise, you are going to work with an extended version of the trucks table. In previous exercises you used a smaller set, with only seven columns. We are going to use the more complete version of that table here.

We are going to use a combination of *cell magic* commands and string encapsulation, so that we can pass multi-line commands to Hive, through Colab. The **%%bash** cell magic is used to run shell commands.

Please pay close attention to the format of the bash commands.

Replace <your-NetID> with your NetID.

For example, my third line would look like: **CREATE TABLE default.pkang8_trucks**

```
%%bash
$HIVE_HOME/bin/hive -e "
CREATE TABLE default.<your-NetID>_trucks
(driverid String,truckid String,model String,
jun13_miles int,jun13_gas int,may13_miles int,may13_gas int,apr13_miles int,
apr13_gas int,mar13_miles int,mar13_gas int,feb13_miles int,feb13_gas int,
jan13_miles int,jan13_gas int,dec12_miles int,dec12_gas int,nov12_miles int,
nov12_gas int,oct12_miles int,oct12_gas int,sep12_miles int,sep12_gas int,
aug12_miles int,aug12_gas int,jul12_miles int,jul12_gas int,jun12_miles int,
jun12_gas int,may12_miles int,may12_gas int,apr12_miles int,apr12_gas int,
mar12_miles int,mar12_gas int,feb12_miles int,feb12_gas int,jan12_miles int,
jan12_gas int,dec11_miles int,dec11_gas int,nov11_miles int,nov11_gas int,
oct11_miles int,oct11_gas int,sep11_miles int,sep11_gas int,aug11_miles int,
aug11_gas int,jul11_miles int,jul11_gas int,jun11_miles int,jun11_gas int,
may11_miles int,may11_gas int,apr11_miles int,apr11_gas int,mar11_miles int,
mar11_gas int,feb11_miles int,feb11_gas int,jan11_miles int,jan11_gas int,
dec10_miles int,dec10_gas int,nov10_miles int,nov10_gas int,oct10_miles int,
oct10_gas int,sep10_miles int,sep10_gas int,aug10_miles int,aug10_gas int,
jul10_miles int,jul10_gas int,jun10_miles int,jun10_gas int,may10_miles int,
may10_gas int,apr10_miles int,apr10_gas int,mar10_miles int,mar10_gas int,
feb10_miles int,feb10_gas int,jan10_miles int,jan10_gas int,dec09_miles int,
dec09_gas int,nov09_miles int,nov09_gas int,oct09_miles int,oct09_gas int,
sep09_miles int,sep09_gas int,aug09_miles int,aug09_gas int,jul09_miles int,
jul09_gas int,jun09_miles int,jun09_gas int,may09_miles int,may09_gas int,
apr09_miles int,apr09_gas int,mar09_miles int,mar09_gas int,feb09_miles int,
feb09_gas int,jan09_miles int,jan09_gas int)
COMMENT 'trucks'
ROW FORMAT DELIMITED
FIELDS TERMINATED BY ','
LINES TERMINATED BY '\n'
STORED AS TEXTFILE
TBLPROPERTIES('skip.header.line.count'='1');
"
```

2. Create a 'geolocation' Hive table

```
%%bash
$HIVE_HOME/bin/hive -e "
CREATE TABLE default.<your-NetID>_geolocation (
truckid String,driverid String,event String,latitude String,longitude String,
city String,state String,velocity int,event_ind int,idling_ind int)
COMMENT 'geolocation'
ROW FORMAT DELIMITED
FIELDS TERMINATED BY ','
LINES TERMINATED BY '\n'
STORED AS TEXTFILE
TBLPROPERTIES('skip.header.line.count'='1');
"
```

Verify the tables you just created Verify the tables if they were created successfully

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

Let's use the 'DESCRIBE table' to show the list of columns, including partition columns for the given table.

```
!$HIVE_HOME/bin/hive -e "DESCRIBE default.<your-NetID>_trucks;"
```

```
!$HIVE_HOME/bin/hive -e "DESCRIBE default.<your-NetID>_geolocation;"
```

Verify storage of the tables in HDFS By default, when you create a table in Hive, a directory with the same name gets created in the /user/hive/warehouse folder in HDFS. You should see both a geolocation and trucks directory.

```
!$HADOOP_HOME/bin/hadoop fs -ls /user/hive/warehouse
```

Populate Hive tables

loading datasets (files) into Hive tables is done through the Load operation. This is used to move the data into the corresponding Hive table. If the keyword local is specified, we need to give the local file system path. If the keyword local is not specified, we need to use the HDFS path of the file.

Before proceeding, make sure you've uploaded `geolocation.csv` and `trucks_extended.csv` to Google Colab.

Note: For this exercise, you are using the extended version of *trucks.csv*, called *trucks_extended.csv*. In previous exercises you used a smaller copy of this file. Please make sure you use the ***trucks_extended.csv*** file.

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

Load geolocation dataset.

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

Sample data from the trucks table Type the following query

```
!$HIVE_HOME/bin/hive -e "SELECT * FROM default.<your-NetID>_trucks LIMIT 1;"
```

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

Hive - Transformation

Next, we will be using Hive to analyze derived data from the geolocation and trucks tables. The business objective is to better understand the risk the company is under from drivers' fatigue, over-used trucks, and the impact of various trucking events on risk. In order to accomplish this, we will apply a series of transformations to the source data.

Let's get started with the first transformation. We want to calculate the miles per gallon for each truck. We will begin with our truck data table. We need to sum up all the miles and gas columns on a per truck basis. Hive has a series of functions that can be used to reformat a table. The keyword LATERAL VIEW is how we invoke things. The stack function allows us to restructure the data into 3 columns labeled rdate, mile, and gas, e.g., 'june13', june13_miles, june13_gas, that make up a maximum of 54 rows. We pick truckid, driverid, rdate, miles, gas from our original table and add a calculated column for mpg (miles/gas). And then we will calculate the average mileage.

For more detail on Lateral View, see Hive Language Manual.

Create table 'truck_mileage' from existing trucking data

Execute the following query

```
%%bash
$HIVE_HOME/bin/hive -e "
CREATE TABLE default.<your-NetID>_truck_mileage STORED AS ORC AS
    SELECT truckid, driverid, rdate, miles, gas, miles / gas mpg
    FROM default.<your-NetID>_trucks
    LATERAL VIEW stack(54,
        'jun13',jun13_miles,jun13_gas,'may13',may13_miles,may13_gas,
        'apr13',apr13_miles,apr13_gas,'mar13',mar13_miles,mar13_gas,
        'feb13',feb13_miles,feb13_gas,'jan13',jan13_miles,jan13_gas,
        'dec12',dec12_miles,dec12_gas,'nov12',nov12_miles,nov12_gas,
        'oct12',oct12_miles,oct12_gas,'sep12',sep12_miles,sep12_gas,
        'aug12',aug12_miles,aug12_gas,'jul12',jul12_miles,jul12_gas,
        'jun12',jun12_miles,jun12_gas,'may12',may12_miles,may12_gas,
        'apr12',apr12_miles,apr12_gas,'mar12',mar12_miles,mar12_gas,
        'feb12',feb12_miles,feb12_gas,'jan12',jan12_miles,jan12_gas,
        'dec11',dec11_miles,dec11_gas,'nov11',nov11_miles,nov11_gas,
        'oct11',oct11_miles,oct11_gas,'sep11',sep11_miles,sep11_gas,
        'aug11',aug11_miles,aug11_gas,'jul11',jul11_miles,jul11_gas,
        'jun11',jun11_miles,jun11_gas,'may11',may11_miles,may11_gas,
        'apr11',apr11_miles,apr11_gas,'mar11',mar11_miles,mar11_gas,
        'feb11',feb11_miles,feb11_gas,'jan11',jan11_miles,jan11_gas,
        'dec10',dec10_miles,dec10_gas,'nov10',nov10_miles,nov10_gas,
        'oct10',oct10_miles,oct10_gas,'sep10',sep10_miles,sep10_gas,
        'aug10',aug10_miles,aug10_gas,'jul10',jul10_miles,jul10_gas,
        'jun10',jun10_miles,jun10_gas,'may10',may10_miles,may10_gas,
        'apr10',apr10_miles,apr10_gas,'mar10',mar10_miles,mar10_gas,
        'feb10',feb10_miles,feb10_gas,'jan10',jan10_miles,jan10_gas,
        'dec09',dec09_miles,dec09_gas,'nov09',nov09_miles,nov09_gas,
        'oct09',oct09_miles,oct09_gas,'sep09',sep09_miles,sep09_gas,
        'aug09',aug09_miles,aug09_gas,'jul09',jul09_miles,jul09_gas,
        'jun09',jun09_miles,jun09_gas,'may09',may09_miles,may09_gas,
        'apr09',apr09_miles,apr09_gas,'mar09',mar09_miles,mar09_gas,
        'feb09',feb09_miles,feb09_gas,'jan09',jan09_miles,jan09_gas )
        dummyalias AS rdate, miles, gas;
"
```

Verify the table. You should have your own truck_mileage table

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

Explore a sampling of the data in the truck_mileage table To view the data, execute the following query.

```
    !$HIVE_HOME/bin/hive -e "DESCRIBE default.<your-NetID>_truck_mileage;"
```

You should see a table that lists each trip made by a truck and driver.

If you want to show the header in the output, use the set command.

```
!$HIVE_HOME/bin/hive -e "SET hive.cli.print.header=true;"
!$HIVE_HOME/bin/hive -e "SELECT * FROM default.<your-NetID>_truck_mileage LIMIT 10;"
```

Save Results of Query: create table avg_mileage from existing trucks_mileage data.

It is common to save the results of the query into a table, so the result set becomes persistent. This is known as Create Table As Select (CTAS).

Let's try to find the top 10 truck (mpg). To do that, create a table avg_mileage using 'group by' and 'order by'. We will store the table as an ORC file.

Since we are using Google Colab, we are limited in terms of resources available to us. If we had Hive installed in a more traditional environment or using cloud-based big data solutions, then we would have been able to try more advanced features (such as MapReduce's ORDER BY). Instead, we will be using less memory intensive operations that yield the same results. For example, instead of **ORDER BY**, we will be using the **DISTRIBUTE BY** and **SORT BY** as an alternative for a more scalable approach.

Let's learn a little more about Hive file formats. Apache ORC is a fast columnar storage file format for Hadoop workloads. The Optimized Row Columnar (Apache ORC project) file format provides a highly efficient way to store Hive data. It was designed to overcome the limitations of the other Hive file formats. Using ORC files improves performance when Hive is reading, writing, and processing data.

To find the top 10 truck (MPG)

Running the following code will result in the execution of two jobs and finally data will be stored in a directory.

```
%%bash
$HIVE_HOME/bin/hive -e "
CREATE TABLE default.<your-NetID>_avg_mileage
STORED AS ORC
AS
SELECT truckid, avg(mpg) avgmpg
FROM default.<your-NetID>_truck_mileage
GROUP BY truckid
DISTRIBUTE BY avgmpg
SORT BY avgmpg DESC;
"
```

Now you can find the top 10 truck/drivers (mpg) using a select statement. This provides a list of average miles per gallon for each truck.

```
!$HIVE_HOME/bin/hive -e "SELECT * FROM default.<your-NetID>_avg_mileage LIMIT 10;"
```

Section 3 (1.5 points)

Answer all the questions and write codes for each question in **Separate Cells**.

Q1. (1 point) Create a Hive table using the **trucks_extended.csv** file, 'default.<your-NetID>_driver_mileage', to find top drivers.

From default.<your-NetID>_truck_mileage table:

- Group the records by driverid and aggregate (sum()) their miles
- Sort the records by sums of miles (descending order)
- Store the table as an ORC file format

Q2. (0.2 points) Show the detailed information of the table:

- Should include Column names and data_type
- Should include Detailed Table Info (Owner, createTime, and so on)
- Should include Storage Information if it's stored as ORC file format
- See this resource.

Q3. (0.3 points) Show top 10 drivers and their aggregated miles

- Using the driver_mileage table you created in Q1
- Note: Check the output values/numbers are correct
- Show 10 rows only