HandOut.8: Delving into Hive (2%)

Installation

Access your docker working directory

Get the git for docker hive: git clone https://github.com/big-data-europe/docker-hive.git

Execute: docker-compose up –d

Check all containers: docker ps

Check your IP: ipconfig /all

[ToDo: Snapshot of Presto at 8080] (Presto is part of Hive)

Bash into hive: docker-compose exec hive-server bash

Execute Beeline: /opt/hive/bin/beeline -u jdbc:hive2://localhost:10000

Basics

Check Beeline help for all available commands: !help

List all current connections: !list

Set variables in Beeline for session-specific configurations: !set [ToDo: Try setting 2 variables and check their output]

Modify how query results are displayed:!set outputformat=<format> (replace format by “table”, “vertical”, “csv”)

Quit the session: !quit

Login again into Beehive

Create Database:

CREATE DATABASE [IF NOT EXISTS] userdb;

CREATE SCHEMA userdb;

SHOW DATABASES;

Drop Database:

```
DROP DATABASE IF EXISTS userdb;

DROP DATABASE IF EXISTS userdb CASCADE; //drop tables then database

DROP SCHEMA userdb;

SHOW DATABASES;
```

Create Table:

```
CREATE TABLE IF NOT EXISTS employee ( eid int, name String, salary String, destination String)

COMMENT ‘Employee details’

ROW FORMAT DELIMITED

FIELDS TERMINATED BY ‘\t’

LINES TERMINATED BY ‘\n’

STORED AS TEXTFILE;
```

Alter Table:

```
ALTER TABLE name RENAME TO new_name

ALTER TABLE name ADD COLUMNS (col_spec[, col_spec ...])

ALTER TABLE name DROP [COLUMN] column_name

ALTER TABLE name CHANGE column_name new_name new_type ALTER TABLE name REPLACE COLUMNS (col_spec[, col_spec ...])

ALTER TABLE employee RENAME TO emp;
```

Drop Table

```
DROP TABLE IF EXISTS employee;
```

Exercises with Student DB:

DDL:

- CREATE DATABASE IF NOT EXISTS student_db;
- USE student_db;
- SHOW DATABASES;
- DROP DATABASE IF EXISTS student_db CASCADE;

-
  ```
  CREATE TABLE students (
          student_id INT,
          name STRING,
          age INT,
          major STRING
  )
  ROW FORMAT DELIMITED
  FIELDS TERMINATED BY ','
  STORED AS TEXTFILE;
  ```
- SHOW TABLES;
- DESCRIBE students;
- ALTER TABLE students ADD COLUMNS (gpa FLOAT);
- ALTER TABLE students RENAME TO student_info;
- DROP TABLE IF EXISTS student_info;

DDL:

- [ToDo: Create a CSV file with some fictitious student data – should have column names at top and data below, e.g., name, CGPA, degree program etc.]
- Load student CSV [ToDo: paste output]: LOAD DATA LOCAL INPATH '/path/to/students_data.csv' INTO TABLE students;
- [ToDo: Modify the data in the CSV file]
- Replace existing data in the table [ToDo: paste output]: LOAD DATA LOCAL INPATH '/path/to/students_data.csv' OVERWRITE INTO TABLE students;

- Inserting data – this should add data after your CSV data [ToDo: paste output]: INSERT INTO TABLE students VALUES (1, 'John Doe', 20, 'Computer Science', 3.5);
- Insert data from another table – you need to create another table for this – try for your own knowledge [ToDo: paste output]: INSERT INTO TABLE students SELECT * FROM student_backup;
- Update GPA: UPDATE students SET gpa = 3.8 WHERE student_id = 1;
- DELETE FROM students WHERE age < 18;

Queries:

- SELECT * FROM students;
- SELECT * FROM students WHERE major = 'Computer Science';
- SELECT * FROM students ORDER BY gpa DESC;
- SELECT major, COUNT(*) AS student_count FROM students GROUP BY major;
- [ToDo: paste output – you need to create a departments table first and then fill it up with some dummy data] SELECT s.student_id, s.name, d.department_name
  FROM students s
  JOIN departments d ON s.major = d.major;

Partition and Bucket:

Suppose that a table named Tab1 contains employee data such as id, name, dept, and yoj (i.e., year of joining). Suppose you need to retrieve the details of all employees who joined in 2012. A query searches the whole table for the required information. However, if you partition the employee data with the year and store it in a separate file, it reduces the query processing time. The following example shows how to partition a file and its data:

The following file contains employeedata table.

Path: /tab1/employeedata/file1

id, name, dept, yoj

1, gopal, TP, 2012

2, kiran, HR, 2012

3, kaleel,SC, 2013

4, Prasanth, SC, 2013

The above data is partitioned into two files using year.

Path: /tab1/employeedata/2012/file2

1, gopal, TP, 2012

2, kiran, HR, 2012

Path: /tab1/employeedata/2013/file3

3, kaleel,SC, 2013

4, Prasanth, SC, 2013

[ToDo: paste output – you can create a partition of your own choice] **Partition by year of enrollment:**

```
CREATE TABLE students_by_year (
  student_id INT,
  name STRING,
  age INT,
  major STRING
)
PARTITIONED BY (year INT)
STORED AS TEXTFILE;
```

[ToDo: paste output – you can create clusters/buckets of your own choice] Clustered by student_id into 4 buckets:

```
CREATE TABLE students_bucketed (
  student_id INT,
  name STRING,
  age INT,
  major STRING
)
CLUSTERED BY (student_id) INTO 4 BUCKETS
STORED AS TEXTFILE;
```

[ToDo: paste output – you can explain plan of any query of your own choice – describe the output] Explain the query plan:

```
EXPLAIN SELECT * FROM students WHERE age > 20;
```

Built-in Functions:

round, floor, ceil, rand, concat, substr, upper, ucase, lower, lcase, trim, rtrim, regex_replace, size, cast, to_date, year, month, day, get_json_object

[ToDo: try 2 functions and paste output]

Aggregate Functions:

count(*), count(expr), sum, avg, min, max

Views:

The usage of view in Hive is same as that of the view in SQL. It is a standard RDBMS concept. We can execute all DML operations on a view.

Example: CREATE VIEW emp_30000 AS SELECT * FROM employee WHERE salary>30000;

[ToDo: create a view on the student's database and show output]

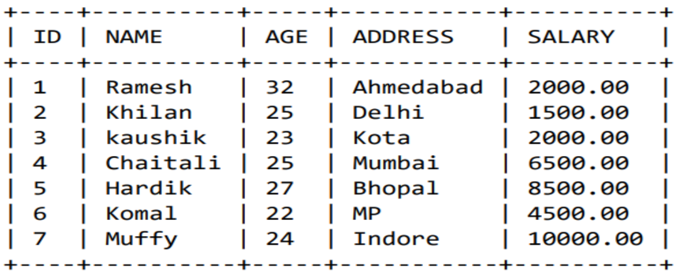
Indexes

CREATE INDEX inedx_salary ON TABLE employee(salary) AS 'org.apache.hadoop.hive.ql.index.compact.CompactIndexHandler';

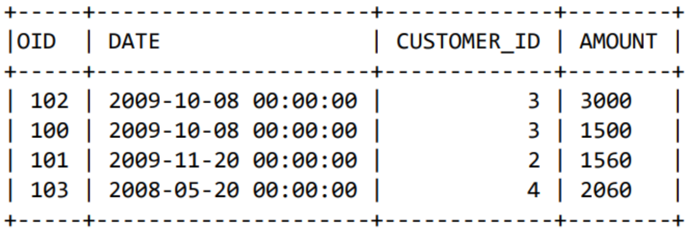
[ToDo: create an index on the student's database and execute queries to show difference in performance after indexing]

Joins:

Some basic examples are given below.



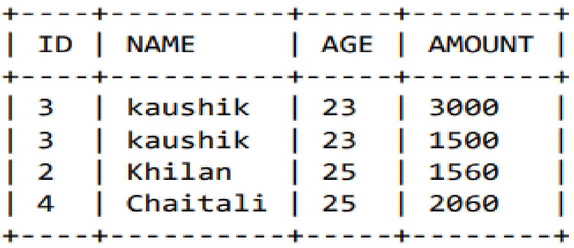
```
+----+----------+-----+-----------+----------+
| ID | NAME     | AGE | ADDRESS   | SALARY   |
+----+----------+-----+-----------+----------+
| 1  | Ramesh   | 32  | Ahmedabad | 2000.00  |
| 2  | Khilan   | 25  | Delhi     | 1500.00  |
| 3  | kaushik  | 23  | Kota      | 2000.00  |
| 4  | Chaitali | 25  | Mumbai    | 6500.00  |
| 5  | Hardik   | 27  | Bhopal    | 8500.00  |
| 6  | Komal    | 22  | MP        | 4500.00  |
| 7  | Muffy    | 24  | Indore    | 10000.00 |
+----+----------+-----+-----------+----------+
```



```
+-----+---------------------+-------------+--------+
|OID  | DATE                | CUSTOMER_ID | AMOUNT |
+-----+---------------------+-------------+--------+
| 102 | 2009-10-08 00:00:00 |           3 | 3000   |
| 100 | 2009-10-08 00:00:00 |           3 | 1500   |
| 101 | 2009-11-20 00:00:00 |           2 | 1560   |
| 103 | 2008-05-20 00:00:00 |           4 | 2060   |
+-----+---------------------+-------------+--------+
```

Inner Join

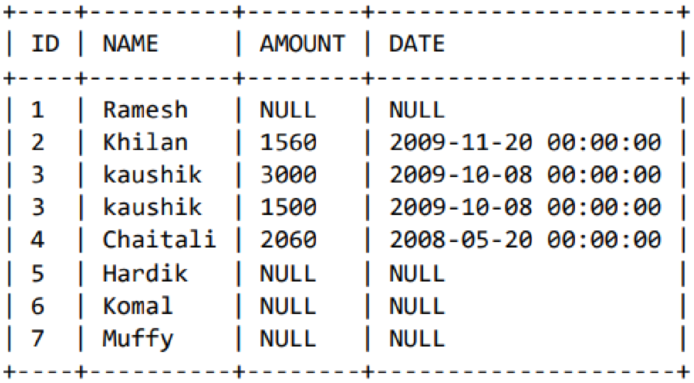
SELECT c.ID, c.NAME, c.AGE, o.AMOUNT FROM CUSTOMERS c JOIN ORDERS o ON (c.ID = o.CUSTOMER_ID);



```
+----+----------+-----+--------+
| ID | NAME     | AGE | AMOUNT |
+----+----------+-----+--------+
| 3  | kaushik  | 23  | 3000   |
| 3  | kaushik  | 23  | 1500   |
| 2  | Khilan   | 25  | 1560   |
| 4  | Chaitali | 25  | 2060   |
+----+----------+-----+--------+
```

Left Outer Join

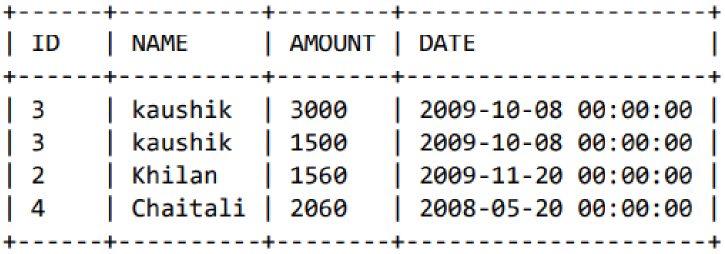
SELECT c.ID, c.NAME, o.AMOUNT, o.DATE FROM CUSTOMERS c LEFT OUTER JOIN ORDERS o ON (c.ID = o.CUSTOMER_ID);



```
+----+----------+--------+---------------------+
| ID | NAME     | AMOUNT | DATE                |
+----+----------+--------+---------------------+
| 1  | Ramesh   | NULL   | NULL                |
| 2  | Khilan   | 1560   | 2009-11-20 00:00:00 |
| 3  | kaushik  | 3000   | 2009-10-08 00:00:00 |
| 3  | kaushik  | 1500   | 2009-10-08 00:00:00 |
| 4  | Chaitali | 2060   | 2008-05-20 00:00:00 |
| 5  | Hardik   | NULL   | NULL                |
| 6  | Komal    | NULL   | NULL                |
| 7  | Muffy    | NULL   | NULL                |
+----+----------+--------+---------------------+
```

Right Outer Join

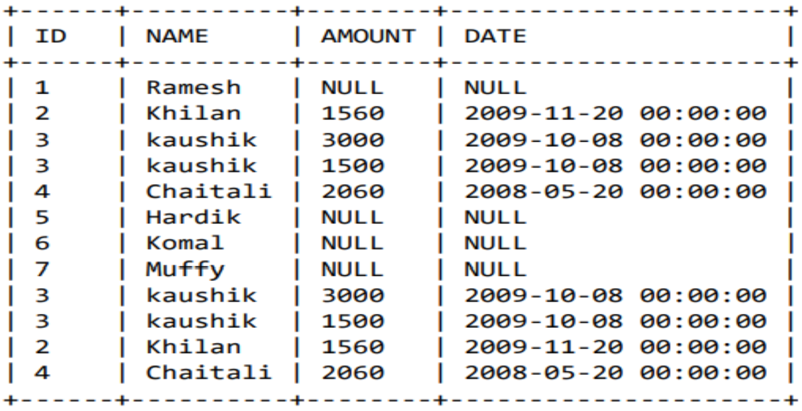
SELECT c.ID, c.NAME, o.AMOUNT, o.DATE FROM CUSTOMERS c RIGHT OUTER JOIN ORDERS o ON (c.ID = o.CUSTOMER_ID);



```
+------+----------+--------+---------------------+
| ID   | NAME     | AMOUNT | DATE                |
+------+----------+--------+---------------------+
| 3    | kaushik  | 3000   | 2009-10-08 00:00:00 |
| 3    | kaushik  | 1500   | 2009-10-08 00:00:00 |
| 2    | Khilan   | 1560   | 2009-11-20 00:00:00 |
| 4    | Chaitali | 2060   | 2008-05-20 00:00:00 |
+------+----------+--------+---------------------+
```

Full Outer Join:

SELECT c.ID, c.NAME, o.AMOUNT, o.DATE FROM CUSTOMERS c FULL OUTER JOIN ORDERS o ON (c.ID = o.CUSTOMER_ID);



```
+------+----------+--------+---------------------+
| ID   | NAME     | AMOUNT | DATE                |
+------+----------+--------+---------------------+
| 1    | Ramesh   | NULL   | NULL                |
| 2    | Khilan   | 1560   | 2009-11-20 00:00:00 |
| 3    | kaushik  | 3000   | 2009-10-08 00:00:00 |
| 3    | kaushik  | 1500   | 2009-10-08 00:00:00 |
| 4    | Chaitali | 2060   | 2008-05-20 00:00:00 |
| 5    | Hardik   | NULL   | NULL                |
| 6    | Komal    | NULL   | NULL                |
| 7    | Muffy    | NULL   | NULL                |
| 3    | kaushik  | 3000   | 2009-10-08 00:00:00 |
| 3    | kaushik  | 1500   | 2009-10-08 00:00:00 |
| 2    | Khilan   | 1560   | 2009-11-20 00:00:00 |
| 4    | Chaitali | 2060   | 2008-05-20 00:00:00 |
+------+----------+--------+---------------------+
```

Experiment with geolocation file:

Make directories for data files in container (geolocation data, available as geolocation.csv)

```
docker-compose exec hive-server bash

cd ..

whereis hive

cd home

mkdir geolocationmkdir trucks

exit
```

Copy from local to container

Access hive folder (where yml file is located)

```
docker cp geolocations.csv docker-hive-master_hive-server_1:/home/geolocation
```

```
docker-compose exec hive-server bash
```

check the copied file

Copy from container to hdfs

```
hadoop fs -mkdir /user/data/
```

```
hadoop fs -put -f /home/geolocation /user/data/
```

```
hadoop fs -ls /user/data/geolocation
```

Access hive command prompt

```
/opt/hive/bin/beeline -u jdbc:hive2://localhost:10000
```

First, we need to create the table according to the schema of geolocation.csv

```
CREATE TABLE IF NOT EXISTS geos
(
truckid string,
driverid string,
event string,
latitude decimal(5,0),
longitude decimal(5,0),
city string,
state string,
velocity int,
event_ind int,

idling_ind int

)

COMMENT 'Geo Table' ROW FORMAT DELIMITED FIELDS TERMINATED BY ',';


LOAD DATA INPATH '/user/data/geolocation/geolocation.csv' INTO TABLE geos;
```

Now, run the following queries:

- select * from geos;
- select * from geos where event = "overspeed";
- select * from geos where velocity < 40;
- select distinct event from geos;
- select avg(velocity) from geos where event ="lane departure" group by event;
- select * from geos order by velocity desc limit 5;

[ToDo: Show queries and their output]

Exercises

- Get all the cities for driver id A54
- Get driver who have visited the least amount of cities
- Get driver who on average drives the slowest
- Using the like statement get all cities with name starting from A

Importing trucks file

We need to import the trucks data file now.

Make a “trucks” directory, copy from local to container, copy from container to hdfs, and access hive command prompt (as for the above geolocation file)

```
CREATE EXTERNAL TABLE IF NOT EXISTS 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) ROW FORMAT DELIMITED FIELDS
TERMINATED BY ',' STORED AS TEXTFILE LOCATION '/user/data/trucks';
```

[ToDo: Show output of above – what advantage do we have for an external table]

[ToDo: What is each query doing – explain in a single sentence only in your own words:]

- select count(model) as modelCount, model from trucks group by model order by modelCount desc;
- select * from geos ORDER BY truckid;
- select * from geos SORT BY driverid ASC;
- select driverid, city from geos DISTRIBUTE BY driverid; (what is distribute by and its advantage?)
- select driverid, city from geos CLUSTER BY driverid; (what is cluster by and its advantage?)
- UPDATE geolocation SET driverid = null WHERE event = “normal”;
- MERGE INTO geos USING
  (SELECT * FROM trucks) sub ON sub.driverid=geos.driverid
  WHEN MATCHED THEN UPDATE SET truckid = sub.truckid, driverid = sub.driverid
  WHEN NOT MATCHED THEN INSERT VALUES (sub.truckid, sub.driverid);
- Joins:

    - select geos.driverid,trucks.model,geos.city from geos join trucks where geos.driverid=trucks.driverid;
    - select geos.driverid,trucks.model,geos.city from geos left outer join trucks where geos.driverid=trucks.driverid;
    - select geos.driverid,trucks.model, geos.city from geos right outer join trucks where geos.driverid=trucks.driverid;
    - select geos.driverid,trucks.model, geos.city from geos full outer join trucks where geos.driverid=trucks.driverid;
- [ToDo: Write queries for the following]:
    - Display truckid, driverid and model for every abnormal event
    - Display truckid, driverid, model and city for velocity > 25.
    - Display complete record of the trucks for events where they were on unsafe following distance.

Hive Transaction Manager:

One of the important properties that you need to know is hive.txn.manager which is used to set Hive Transaction manager, by default hive uses DummyTxnManager, to enable ACID, we need to set it to DbTxnManager.

SET hive.support.concurrency=true;

SET hive.txn.manager=org.apache.hadoop.hive.ql.lockmgr.DbTxnManager;

# The following are not required if you are using Hive 2.0

SET hive.enforce.bucketing=true;

SET hive.exec.dynamic.partition.mode=nostrict;

# The following parameters are required for standalone hive metastore:

SET hive.compactor.initiator.on=true;

SET hive.compactor.worker.threads=1

[ToDo: execute several hive commands from truck data and show the difference in performance between ACID and non-ACID transaction managers]

Below are some of the limitations of using Hive ACID transactions:

- To support ACID, Hive tables should be created with TRANSACTIONAL table property. Currently, Hive supports ACID transactions on tables that store ORC file format.
- Enable ACID support by setting transaction manager to DbTxnManager
- Transaction tables cannot be accessed from the non-ACID Transaction Manager (DummyTxnManager) session.
- External tables cannot be created to support ACID since the changes on external tables are beyond Hive control.
- LOAD is not supported on ACID transactional Tables. hence use INSERT INTO.
- On Transactional session, all operations are auto commit as BEGIN, COMMIT and ROLLBACK are not yet supported.

Bonus Question (1%):

Experiment with Tez, Spark and LLAP execution engines and benchmark wrt MR.