Spark SQL / HiveQL

Fundamentals

1. Table of Contents

2. Accessing Spark SQL / Hive With Beeline

The goal of this lab is to demonstrate the usage of Beeline to connect to Spark SQL ThriftServer / Hive

1. Start a Jupyter terminal session and run:

```
/opt/spark/bin/beeline -u jdbc:hive2://127.0.0.1:10000/default
```

2. On the Beeline shell, lets list all databases

```
SHOW databases;
```

3. Accessing Spark SQL / Hive With Squirrel SQL

The goal of this lab is to install Squirrel SQL and use it to connect to Hive database

1. Dependency:
   - Download JRE : https://www.oracle.com/technetwork/java/javase/downloads/jre8-downloads-2133155.html
   - Download SquirrelSQL : http://squirrel-sql.sourceforge.net/#installation
   - Download Hive JDBC 4.1 driver : https://hortonworks.com/downloads/
   - Ensure the VM have port forwarding configured for port 10000 and 10500

Lets install Squirrel SQL and Hive2 driver:

1. On your main host (outside the VM), install JRE and then run Squirrel SQL jar
2. Extract JDBC driver JAR from the zip file, and save it in SquirrelSQL's lib directory
3. Launch SquirrelSQL, and click "Drivers" tab on the left
4. Click + button on the left bar to register a new driver
   - Name: HiveServer2
   - Example URL: jdbc:hive2://localhost:10000/default
   - Extra Class Path: add the HiveJDBC4.1.jar into the path list, select it, then click List Drivers to update list of Class Name
   - Class Name: com.simba.hive.jdbc41.HS2Driver
5. Click OK to Save

Lets create a new connection alias to Hive

1. Launch SquirrelSQL and click Aliases tab on the left
2. Click + button to add new alias
   - Name: Sandbox Hive
   - Driver: HiveServer2
   - URL: jdbc:hive2://127.0.0.1:10000/default
   - User Name: admin
3. Click Test to test the driver for connectivity and if it function correctly, click OK to save.

Lets connect to Hive

1. Launch SquirrelSQL and click Aliases tab on the left
2. Select Sandbox Hive, right click, connect
3. On the SQL tab, enter:

```
SHOW databases;
```

4. Select the statement and Ctrl+Enter to execute

4. Creating Tables With Hive DDL

The goal of this lab is to demonstrate the DDL for different types of tables on Hive

4.1. Basic Hive Managed Table

1. Connect to Spark SQL / Hive through beeline in Jupyter terminal session

```
/opt/spark/bin/beeline -u jdbc:hive2://127.0.0.1:10000/default
```

2. Lets create a simple demo table, stored as TextFile

```
CREATE TABLE demo_basic (
    key int,
    value string )
ROW FORMAT DELIMITED
FIELDS TERMINATED BY ','
STORED AS TEXTFILE;
```

3. Lets insert some values

```
INSERT INTO TABLE demo_basic VALUES (1, 'hello'), (2, 'world');
SELECT * FROM demo_basic;
```

4. Press CTRL+d to exit beeline.
5. Lets check the contents of /warehouse

```
hdfs dfs -ls /warehouse/
hdfs dfs -ls /warehouse/demo_basic/
```

6. Lets see the content of the first file

```
hdfs dfs -cat /warehouse/demo_basic/*
```

4.2. Partitioned Table

1. Connect to Spark SQL / Hive through beeline in Jupyter terminal session

```
/opt/spark/bin/beeline -u jdbc:hive2://127.0.0.1:10000/default
```

2. Lets create a simple partitioned demo table, stored as TextFile

```
CREATE TABLE demo_partition (
    key int,
    value string )
PARTITIONED BY (dt string)
ROW FORMAT DELIMITED
FIELDS TERMINATED BY ','
STORED AS TEXTFILE;
```

3. Lets insert some values

```
INSERT INTO TABLE demo_partition PARTITION (dt='2018-01-01')
                                VALUES    (1,    'hello'),    (2,    'world');
INSERT INTO TABLE demo_partition PARTITION (dt='2018-01-02')
       VALUES (3, 'foo'), (4, 'bar');
SELECT * FROM demo_partition;
```

4. Press CTRL+d to exit beeline.

5. Lets check the contents of /apps/hive/warehouse. Notice the partition based directory structure

```
hdfs dfs -ls /warehouse/
hdfs dfs -ls /warehouse/demo_partition/
hdfs dfs -ls /warehouse/demo_partition/dt=2018-01-01/
```

4.3. Basic Table with ORC Storage

1. Connect to Spark SQL / Hive through beeline in Jupyter terminal session

```
/opt/spark/bin/beeline -u jdbc:hive2://127.0.0.1:10000/default
```

2. Lets create a table, using ORC store

```
CREATE TABLE demo_basic_orc (
    key int,
    value string )
STORED AS ORC;
```

3. Lets insert some values

```
INSERT INTO TABLE demo_basic_orc VALUES (1, 'hello'), (2, 'world');
SELECT * FROM demo_basic_orc;
```

4. Press CTRL+d to exit beeline.
5. Lets check the contents of /apps/hive/warehouse

```
hdfs dfs -ls /warehouse/
hdfs dfs -ls /warehouse/demo_basic_orc/
```

6. Lets see the content of the first file

```
hdfs dfs -cat /warehouse/demo_basic_orc/* | hexdump -C
```

4.4. Create Table As Select

1. Connect to Spark SQL / Hive through beeline in Jupyter terminal session

```
/opt/spark/bin/beeline -u jdbc:hive2://127.0.0.1:10000/default
```

2. Lets create a copy of demo_basic, but convert it to ORC using CTAS

```
CREATE TABLE demo_basic_orcconvert STORED AS ORC AS SELECT * FROM demo_basic;
SELECT * FROM demo_basic_orcconvert;
```

5. Accessing Spark SQL / Hive Table Using Spark

The goal of this lab is to demonstrate accessing a table in Hive / Catalog using PySpark.

1. In a Jupyter PySpark - KDP Notebook, run the following code (this may take a while):

```
from pyspark.sql import SparkSession
from IPython.core.display import display

spark = (SparkSession
    .builder
    .master("local")
    .enableHiveSupport()
    .appName("Python Spark SQL basic example")
    .getOrCreate())
display(spark.sql('show tables').toPandas())
display(spark.sql('select * from demo_basic').toPandas())
```

6. Window Based Analysis Using Spark SQL / Hive

The goal of this lab is to demonstrate the usage of windowing functions in Hive

Lets create a sample dataset for demonstrating windowing functions

1. Connect to Spark SQL / Hive through beeline in Jupyter terminal session

```
/opt/spark/bin/beeline -u jdbc:hive2://127.0.0.1:10000/default
```

2. Run following query to create a sample data

```
CREATE TABLE products (
 product_id int,
 product_name VARCHAR (255),
 group_name VARCHAR (255),
 price int
) STORED AS ORC;

INSERT INTO products
VALUES
 (1, 'Sony VAIO', 'Laptop', 600),
 (2, 'Dell Vostro', 'Laptop', 600),
 (3, 'HP Elite', 'Laptop', 900),
 (4, 'Lenovo Thinkpad', 'Laptop', 1100),
 (5, 'Microsoft Lumia', 'Smartphone', 300),
 (6, 'HTC One', 'Smartphone', 400),
 (7, 'Nexus', 'Smartphone', 500),
 (8, 'iPhone', 'Smartphone', 800),
 (9, 'Kindle Fire', 'Tablet', 300),
 (10, 'iPad', 'Tablet', 700),
 (11, 'Samsung Galaxy Tab', 'Tablet', 800);
```

6.1. ROW_NUMBER()

1. Connect to Spark SQL / Hive through beeline in Jupyter terminal session

```
/opt/spark/bin/beeline -u jdbc:hive2://127.0.0.1:10000/default
```

2. Run following query

```
SELECT
 product_name,
 price,
 group_name,
ROW_NUMBER () OVER (
       PARTITION BY group_name
       ORDER BY price) AS rownum
FROM products;
```

6.2. DENSE_RANK() and RANK()

1. Connect to Spark SQL / Hive through beeline in Jupyter terminal session

```
/opt/spark/bin/beeline -u jdbc:hive2://127.0.0.1:10000/default
```

2. Run following queries, and compare their behaviors

```
SELECT
 product_name,
 price,
 group_name,
DENSE_RANK () OVER (
       PARTITION BY group_name
       ORDER BY price) AS group_rank
FROM products;

SELECT
 product_name,
 price,
 group_name,
RANK () OVER (
       PARTITION BY group_name
       ORDER BY price) AS group_rank
FROM products;
```

6.3. LAG() and LEAD()

1. Connect to Spark SQL / Hive through beeline in Jupyter terminal session

```
/opt/spark/bin/beeline -u jdbc:hive2://127.0.0.1:10000/default
```

2. Run following queries, and compare their behaviors

```
SELECT
 product_name,
 price,
 group_name,
LAG (price, 1) OVER (
       PARTITION BY group_name
       ORDER BY price) AS prev_price
FROM products;

SELECT
 product_name,
 price,
 group_name,
LEAD (price, 1) OVER (
       PARTITION BY group_name
       ORDER BY price) AS next_price
FROM products;

SELECT
 product_name,
 price,
 group_name,
LAG (price, 3) OVER (
       PARTITION BY group_name
       ORDER BY price) AS prev_price
FROM products;

SELECT
```

```
    product_name,
    price,
    group_name,
   LEAD (price, 3) OVER (
          PARTITION BY group_name
          ORDER BY price) AS next_price
   FROM products;
```

6.4. FIRST_VALUE() and LAST_VALUE()

1. Connect to Spark SQL / Hive through beeline in Jupyter terminal session

```
/opt/spark/bin/beeline -u jdbc:hive2://127.0.0.1:10000/default
```

2. Run following queries, and compare their behaviors

```
SELECT
 product_name,
 price,
 group_name,
FIRST_VALUE (price) OVER (
       PARTITION BY group_name
       ORDER BY price) AS firstval
FROM products;

SELECT
 product_name,
 price,
 group_name,
LAST_VALUE (price) OVER (
       PARTITION BY group_name
       ORDER BY price) AS lastval
FROM products;
```

6.5. CUME_DIST() and PERCENT_RANK()

1. Connect to Spark SQL / Hive through beeline in Jupyter terminal session

```
/opt/spark/bin/beeline -u jdbc:hive2://127.0.0.1:10000/default
```

2. Run following queries, and compare their behaviors

```
SELECT
 product_name,
 price,
 group_name,
CUME_DIST () OVER (
       PARTITION BY group_name
       ORDER BY price) AS cumedist
FROM products;

SELECT
 product_name,
 price,
 group_name,
PERCENT_RANK () OVER (
       PARTITION BY group_name
       ORDER BY price) AS prcrank
FROM products;
```

6.6. Aggregate Function With Windowing

1. Connect to Spark SQL / Hive through beeline in Jupyter terminal session

```
/opt/spark/bin/beeline -u jdbc:hive2://127.0.0.1:10000/default
```

2. Run following queries, and compare their behaviors

```
SELECT
 product_name,
 price,
 group_name,
SUM (price) OVER (
       PARTITION BY group_name ORDER BY price
       ROWS BETWEEN 1 PRECEDING AND CURRENT ROW) AS sum_prev
FROM products;

SELECT
 product_name,
 price,
 group_name,
SUM (price) OVER (
       PARTITION BY group_name ORDER BY price
       ROWS BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW) AS sum_prev
FROM products;

SELECT
 product_name,
 price,
 group_name,
SUM (price) OVER (
       PARTITION BY group_name ORDER BY price
       ROWS BETWEEN CURRENT ROW AND UNBOUNDED FOLLOWING) AS sum_next
FROM products;

SELECT
```

```
 product_name,
 price,
 group_name,
SUM (price) OVER (
       PARTITION BY group_name ORDER BY price
       RANGE BETWEEN 400 PRECEDING AND CURRENT ROW) AS sum_range
FROM products;

SELECT
 product_name,
 price,
 group_name,
SUM (price) OVER (
       PARTITION BY group_name ORDER BY price
       RANGE BETWEEN CURRENT ROW AND 400 FOLLOWING) AS sum_range
FROM products;
```