COMMANDS ON SPARK SQL

Spark SQL

Spark SQL Syntax:

1. Spark SQL is one of the spark module used to work with structured data.
2. We will be looking at DML, DDL, Data retrieval and auxiliary sql statements using Spark.

Data Definition Language (DDL):

1. We will be mainly looking at creating, altering and dropping of tables, views.

==========

DATA DEFINITION LANGUAGE

==========

ALTER STATEMENT:

1. It is used to change the schema or properties of the table/view
2. Here it is used to
    - 2.1 Rename table names and partitions
    - 2.2 Add new columns, alter existing columns
    - 2.3 Add/drop partitions
    - 2.4 Set/Unset table properties

1. Add/Alter columns

    1.1 Add new column

    -->DESC db.students; (It describes about the table)

```
DESC student;
+----------------------+---------+-------+
|              col_name|data_type|comment|
+----------------------+---------+-------+
|                  name|   string|   NULL|
|                rollno|      int|   NULL|
|                   age|      int|   NULL|
|# Partition Information|         |       |
|             # col_name|data_type|comment|
|                   age|      int|   NULL|
+----------------------+---------+-------+
```

--> ALTER TABLE Students ADD columns (LastName string, DOB timestamp)

```
ALTER TABLE StudentInfo ADD columns (LastName string, DOB timestamp);

-- After Adding New columns to the table
DESC StudentInfo;
+----------------------+---------+-------+
|              col_name|data_type|comment|
+----------------------+---------+-------+
|                  name|   string|   NULL|
|                rollno|      int|   NULL|
|              LastName|   string|   NULL|
|                   DOB|timestamp|   NULL|
|                   age|      int|   NULL|
|# Partition Information|         |       |
|             # col_name|data_type|comment|
|                   age|      int|   NULL|
+----------------------+---------+-------+
```

--> ALTER TABLE Students ALTER columns (name varchar, age longint)

2. ADD/DROP PARTITIONS

   2.1 Add new partitions

-->SHOW PARTITIONS students (To show all the partitions for the table students)

-->ALTER TABLE StudentInfo ADD IF NOT EXISTS PARTITION (age=10) PARITION (age=20)

```
-- Add a new partition to a table
SHOW PARTITIONS StudentInfo;
+---------+
|partition|
+---------+
|   age=11|
|   age=12|
|   age=15|
+---------+

ALTER TABLE StudentInfo ADD IF NOT EXISTS PARTITION (age=18);

-- After adding a new partition to the table
SHOW PARTITIONS StudentInfo;
+---------+
|partition|
+---------+
|   age=11|
|   age=12|
|   age=15|
|   age=18|
+---------+
```

2.2 DROP existing partitions

-->ALTER TABLE StudentInfo DROP IF EXISTS PARTITION (age=10)

```
ALTER TABLE StudentInfo DROP IF EXISTS PARTITION (age=18);

-- After dropping the partition of the table
SHOW PARTITIONS StudentInfo;
+--------+
|partition|
+--------+
|   age=11|
|   age=12|
|   age=15|
+--------+
```

3. SET/UNSET Table properties

3.1 To add file format and location to the table

-->ALTER TABLE students SET fileformat=parquet;

-->ALTER TABLE students SET

location='/users/documents/table/'

3.2 To set properties for a given partition

-->ALTER TABLE students PARITION (age=10) SET fileformat=parquet;

-->ALTER TABLE students PARITION (age=10) SET location='/users/documents/table/'

CREATE TABLE USING DATA SOURCE:

1. It is used to create a table structure along with various properties that can be set along the way.

-->**CREATE TABLE** db.students (id int, name string, age int, school string)

USING csv (input format used to create a table)

PARTITIONED BY (age) (partition the table based on columns)

CLUSTERED BY (id) INTO 4 buckets (bucket the column values)

TBLPROPERTIES (is-cache=true)

LOCATION '/users/downloads/table/student' (location where the table needs to be stored)

AS select * from student_blr (data from this select statement will populate into this table)

2. CREATE TABLE USING HIVE FORMAT

-->**CREATE EXTERNAL TABLE** db.orders (order_id longint, customer_id longint, item string, cost float, state string)

PARTITIONED BY (state)
CLUSTERED BY (customer_id) **INTO** 8 buckets

ROW FORMAT DELIMITED

FIELDS TERMINATED BY ','

LINES TERMINATED BY '\n'

NULL DEFINED AS '-1'

STORED AS parquet
LOCATION 's3://table/orders'

3. CREATE TABLE LIKE

3.1 Here table is created by making use of the definitions and schema of an existing table/view

-->**CREATE TABLE** IF NOT EXISTS db.salary_dup **LIKE**

db_new.salary_college **ROW FORMAT**
DELIMITED

FIELDS TERMINATED BY ','

LINES TERMINATED BY '\n'

NULL DEFINED AS '-1'

STORED AS parquet
LOCATION 's3://table/orders'

DROP STATEMENT:

1. DROP TABLE
   - 1.1 It drops the table by deleting the tables and removes it from file system
   - 1.2 If it is external table then only metadata is dropped but data still exists
   - 1.3 If table is cached, then it gets uncached and all definitions will be erased

-->DROP TABLE IF EXISTS db1.students

DESCRIBE TABLE:

-->Used to describe the table structure, metadata --
>**DESCRIBE TABLE table_name; DESCRIBE DATABASE db_name DESCRIBE VIEW view_name**

-->desc formatted table_name;

```
DESCRIBE TABLE customer;
+----------------------+---------+----------+
|              col_name|data_type|   comment|
+----------------------+---------+----------+
|               cust_id|      int|      null|
|                  name|   string|Short name|
|                 state|   string|      null|
|# Partition Information|         |          |
|             # col_name|data_type|   comment|
|                 state|   string|      null|
+----------------------+---------+----------+
```

==========

VIEWS

==========

1. CREATE VIEW

1.1 Views are based on result set from a sql query

1.2 They are virtual tables created on top of actual db tables so we can restrict access to db tables

1.3 Alter/drop of views will only affect the metadata

```
-->CREATE OR REPLACE VIEW IF NOT EXISTS db1.studentView
   AS select s1.*, s2.subject, s2.school     from studentInfo s1
    left join studentSchool s2 on s2.id=s1.id
```

2. ALTER VIEW

2.1 Alter views can be used to rename views, set/unset metadata by changing TBLPROPERTIES, change definition of view

```
-->ALTER VIEW IF EXISTS db1.studentView
   AS select * from studentsinfo
```

3. DROP VIEW

3.1 It drops the existing view and doesn't affect the underlying table

```
-->DROP VIEW IF EXISTS db1.studentView
```

==========

DATA MANIPULATION STATEMENT

==========

Topics covered under this section are:

- INSERT INTO
- INSERT OVERWRITE
- INSERT OVERWRITE DIRECTORY
- INSERT OVERWRITE DIRECTORY with Hive format
- LOAD

1. INSERT INTO

1.1 Used to insert rows into a table through various what we will be exploring shortly

Generic Statement: INSERT INTO table_name PARTITION

[(col_name='col_value)] (col_list) VALUES (val1,val2)

1.1 INSERT multiple rows into a table

1.1.1 INSERT INTO students PARTITION (school='navodaya')

VALUES(100,'vaishnavi','navodaya')

1.1.2 column list is optional, if we want to insert data into specific columns only then we can specify the columns

1.2 INSERT USING A SELECT STATEMENT

1.2.1 INSERT INTO db1.students

select stu_id,name,school,city from db1.studentDetails where city='banglore'

1.2.2Make sure the schema of both the tables is same

1.3 INSERT USING TABLE STATEMENT

1.3.1It is used to insert data into a table from another table

1.3.2INSERT INTO db1.students TABLE db1.studentDetails

1.4 INSERT USING FROM STATEMENT

1.4.1 It is used to insert data through a FROM statement 1.4.2 INSERT INTO db1.students

FROM db1.studentBlr select stuId,stuName,stuCity where city='blr'

2. INSERT OVERWRITE

-->It is used to overwrite the existing data using the INSERT

statement by new values

-->All the above statements are same, just append OVERWRITE in the INSERT statements

==========

DATA RETRIEVAL QUERIES

==========

1. COMMON TABLE EXPRESSION

1.1 It gives us the feasibility to have a temporary result set which can be referred to and re-used within the scope of a sql statement Example:

```
With student_cte as
(
Select id, name,school from details where name='vaishnavi'
)

Select * from student_cte
```

2. SET OPERATORS

2.1 Set operators are used to combine 2 or more sql statements into single one

2.2 Make sure to have same no. of columns and same/related schema to perform these operations

2.3 spark SQL supports the below SET operators

EXCEPT/MINUS

INTERSECT

UNION

2.4 EXCEPT/MINUS

2.4.1 It is used to filter out rows which are present in one table but not the other

2.4.2 EXCEPT ALL includes duplicates as well

2.4.3 EXCEPT DISTINCT includes only distinct values

Select name from students EXCEPT select name from details;

```
-- Use number1 and number2 tables to demonstrate set operators in this page.
SELECT * FROM number1;
+---+
|  c|
+---+
|  3|
|  1|
|  2|
|  2|
|  3|
|  4|
+---+

SELECT * FROM number2;
+---+
|  c|
+---+
|  5|
|  1|
|  2|
|  2|
+---+

SELECT c FROM number1 EXCEPT SELECT c FROM number2;
+---+
|  c|
+---+
|  3|
|  4|
+---+

SELECT c FROM number1 MINUS SELECT c FROM number2;
+---+
|  c|
+---+
|  3|
|  4|
+---+

SELECT c FROM number1 EXCEPT ALL (SELECT c FROM number2);
+---+
|  c|
+---+
|  3|
|  3|
|  4|
+---+
```

2.5 INTERSECT

2.5.1 It is used to show the common rows between the two tables

2.5.2 INTERSECT ALL includes duplicate rows as well

2.5.3 INTERSECT DISTINCT includes only distinct rows

```
(SELECT c FROM number1) INTERSECT (SELECT c FROM number2);
+---+
|  c|
+---+
|  1|
|  2|
+---+

(SELECT c FROM number1) INTERSECT DISTINCT (SELECT c FROM number2);
+---+
|  c|
+---+
|  1|
|  2|
+---+

(SELECT c FROM number1) INTERSECT ALL (SELECT c FROM number2);
+---+
|  c|
+---+
|  1|
|  2|
|  2|
+---+
```

2.6 UNION

2.6.1 It is used to print the combined output from the sql queries if they have same no. of columns and schema gets matched

```
(SELECT c FROM number1) UNION (SELECT c FROM number2);
+---+
|  c|
+---+
|  1|
|  3|
|  5|
|  4|
|  2|
+---+

(SELECT c FROM number1) UNION DISTINCT (SELECT c FROM number2);
+---+
|  c|
+---+
|  1|
|  3|
|  5|
|  4|
|  2|
+---+

SELECT c FROM number1 UNION ALL (SELECT c FROM number2);
+---+
|  c|
+---+
|  3|
|  1|
|  2|
|  2|
|  3|
|  4|
|  5|
|  1|
|  2|
|  2|
+---+
```

3. ORDER BY vs SORT BY vs DISTRIBUTE BY vs CLUSTER BY

ORDER BY	SORT BY	DISTRIBUTE BY	CLUSTER BY
1. It is used to order the entire data is either ASC or DESC order. It exists as a single partition.	1. It is used to sort the data in individual partitions	1. It just distributes the entire data into n partitions	1. Distribute By + Sort By 2. It is used to distribute the data and sort the data in each partition

```
-- Produces rows clustered by age. Persons with same age are clustered together.
-- In the query below, persons with age 18 and 25 are in first partition and the
-- persons with age 16 are in the second partition. The rows are sorted based
-- on age within each partition.
SELECT age, name FROM person CLUSTER BY age;
+---+-------+
|age|   name|
+---+-------+
| 18| John A|
| 18| Anil B|
| 25|Zen Hui|
| 25| Mike A|
| 16|Shone S|
| 16| Jack N|
+---+-------+
```

4. CASE CLAUSE vs PIVOT CLAUSE

CASE CLAUSE	PIVOT CLAUSE
1. It is used to assign a value to a column based on the conditions set	1. It is used to get the aggregated values for specific columns mentioned

Case Clause:

Select id, name,

Case

When city='blr' then 'banglore'

When city='dlh' then 'delhi'

Else 'NA'

End

As 'City name'

From cityDetails

Pivot Clause:

```
CREATE TABLE person (id INT, name STRING, age INT, class INT, address STRING);
INSERT INTO person VALUES
    (100, 'John', 30, 1, 'Street 1'),
    (200, 'Mary', NULL, 1, 'Street 2'),
    (300, 'Mike', 80, 3, 'Street 3'),
    (400, 'Dan', 50, 4, 'Street 4');

SELECT * FROM person
    PIVOT (
        SUM(age) AS a, AVG(class) AS c
        FOR name IN ('John' AS john, 'Mike' AS mike)
    );
```

id	address	john_a	john_c	mike_a	mike_c
200	Street 2	NULL	NULL	NULL	NULL
100	Street 1	30	1.0	NULL	NULL
300	Street 3	NULL	NULL	80	3.0
400	Street 4	NULL	NULL	NULL	NULL

5. GROUP BY CLAUSE vs HAVING CLAUSE

GROUP BY	HAVING
1. It is used to group data rows in a table based on the specific aggregate function set. 2. Sum, avg, count	1. It is used to filter/set conditions for the data produced by the grouping aggregates. 2. It is always used along with GROUP BY

Having Clause:

-->As seen, we filter the data after the grouping is done and based on the grouping done

```
-- `HAVING` clause referring to column in `GROUP BY`.
SELECT city, sum(quantity) AS sum FROM dealer GROUP BY city HAVING city = 'Fremont';
+-------+---+
|   city|sum|
+-------+---+
|Fremont| 32|
+-------+---+

-- `HAVING` clause referring to aggregate function.
SELECT city, sum(quantity) AS sum FROM dealer GROUP BY city HAVING sum(quantity) > 15;
+-------+---+
|   city|sum|
+-------+---+
| Dublin| 33|
|Fremont| 32|
+-------+---+

-- `HAVING` clause referring to aggregate function by its alias.
SELECT city, sum(quantity) AS sum FROM dealer GROUP BY city HAVING sum > 15;
+-------+---+
|   city|sum|
+-------+---+
| Dublin| 33|
|Fremont| 32|
+-------+---+

-- `HAVING` clause referring to a different aggregate function than what is present in
-- `SELECT` list.
SELECT city, sum(quantity) AS sum FROM dealer GROUP BY city HAVING max(quantity) > 15;
+------+---+
|  city|sum|
+------+---+
|Dublin| 33|
+------+---+
```

5. WINDOW FUNCTION

5.1 Window functions operate on a group of rows, referred to as a window, and calculate a return value for each row based on the group of rows

5.2 They are mainly used in performing analysis of data to operate on huge amount of data

5.3 Ex. Sum(), lag(), lead(), count(), dense_rank()

Syntax

```
cwindow_function[nulls_option]OVER([{PARTITION|DISTRIBUTE}BYpartition_col_name=partition_
  ol_val([,...])]{ORDER|SORT}BYexpression[ASC|DESC][NULLS{FIRST|LAST}][,...][window_frame])
```

Various Window Function:

- Ranking Functions
  Syntax: RANK | DENSE_RANK | PERCENT_RANK | NTILE | ROW_NUMBER
- Analytic Functions
  Syntax: CUME_DIST | LAG | LEAD | NTH_VALUE | FIRST_VALUE | LAST_VALUE
- Aggregate Functions
  Syntax: MAX | MIN | COUNT | SUM | AVG |

Window Size:

Specifies which row to start the window on and where to end it.

Syntax:

{ RANGE | ROWS } { frame_start | BETWEEN frame_start AND frame_end }

- frame_start and frame_end have the following syntax:

  Syntax:

  UNBOUNDED PRECEDING | offset PRECEDING | CURRENT ROW | offset FOLLOWING | UNBOUNDED FOLLOWING

Sample Queries:

```
SELECT name, dept, DENSE_RANK() OVER (PARTITION BY dept ORDER BY salary ROWS BETWEEN
    UNBOUNDED PRECEDING AND CURRENT ROW) AS dense_rank FROM employees;
+-----+-----------+------+----------+
| name|       dept|salary|dense_rank|
+-----+-----------+------+----------+
| Lisa|      Sales| 10000|         1|
| Alex|      Sales| 30000|         2|
| Evan|      Sales| 32000|         3|
| Fred|Engineering| 21000|         1|
|  Tom|Engineering| 23000|         2|
|Chloe|Engineering| 23000|         2|
| Paul|Engineering| 29000|         3|
|Helen|  Marketing| 29000|         1|
| Jane|  Marketing| 29000|         1|
| Jeff|  Marketing| 35000|         2|
+-----+-----------+------+----------+
```

```
SELECT name, salary,
    LAG(salary) OVER (PARTITION BY dept ORDER BY salary) AS lag,
    LEAD(salary, 1, 0) OVER (PARTITION BY dept ORDER BY salary) AS lead
    FROM employees;
+-----+-----------+------+-----+-----+
| name|       dept|salary|  lag| lead|
+-----+-----------+------+-----+-----+
| Lisa|      Sales| 10000| NULL|30000|
| Alex|      Sales| 30000|10000|32000|
| Evan|      Sales| 32000|30000|    0|
| Fred|Engineering| 21000| NULL|23000|
|Chloe|Engineering| 23000|21000|23000|
|  Tom|Engineering| 23000|23000|29000|
| Paul|Engineering| 29000|23000|    0|
|Helen|  Marketing| 29000| NULL|29000|
| Jane|  Marketing| 29000|29000|35000|
| Jeff|  Marketing| 35000|29000|    0|
+-----+-----------+------+-----+-----+
```

Source Docs:

https://spark.apache.org/docs/latest/sql-ref-syntax.html