DAWD 02-5-2 - Demo - Basic SQL

The two fields below are used to customize queries used in this course. Enter your schema (database) name and username, and press "Enter" to populate necessary information in the queries on this page.

Schema Name:	hive_metastore.class_013_odg7_da_dawd
Username:	class+013@databricks.com

Lesson Objective

At the end of this lesson, you will be able to:

- Describe how to write basic SQL queries to subset tables using Databricks SQL Queries

Retrieving Data

SELECT

1. Run the code below.

This simple command retrieves data from a table. The "*" represents "Select All," so the command is selecting all data from the table

However, note that only 1,000 rows were retrieved. Databricks SQL defaults to only retrieving 1,000 rows from a table. If you wish to retrieve more, deselect the checkbox "LIMIT 1000".

```
USE hive_metastore.class_013_odg7_da_dawd;
SELECT * FROM customers;
```

Copy

SELECT ... AS

By adding the `AS` keyword, we can change the name of the column in the results.

2. Run the code below.

Note that the column `customer_name` has been renamed `Customer`

```
USE hive_metastore.class_013_odg7_da_dawd;
SELECT customer_name AS Customer FROM customers;
```

Copy

DISTINCT

If we add the `DISTINCT` keyword, we can ensure that we do not repeat data in the table.

3. Run the code below.

There are more than 1,000 records that have a state in the `state` field. But, we only see 51 results because there are only 51 distinct state names.

```
USE hive_metastore.class_013_odg7_da_dawd;
SELECT DISTINCT state FROM customers;
```

Copy

WHERE

The `WHERE` keyword allows us to filter the data.

4. Run the code below.

We are selecting from the `customers` table, but we are limiting the results to those customers who have a `loyalty_segment` of 3.

```
USE hive_metastore.class_013_odg7_da_dawd;
SELECT * FROM customers WHERE loyalty_segment = 3;
```

Copy

GROUP BY

We can run a simple `COUNT` aggregation by adding `count()` and `GROUP BY` to our query.

5. Run the code below.

`GROUP BY` requires an aggregating function. We will discuss more aggregations later on.

```
USE hive_metastore.class_013_odg7_da_dawd;
SELECT loyalty_segment, count(loyalty_segment) FROM customers GROUP BY loyalty_segment;
```

Copy

ORDER BY

By adding `ORDER BY` to the query we just ran, we can place the results in a specific order.

6. Run the code below.

`ORDER BY` defaults to ordering in ascending order. We can change the order to descending by adding `DESC` after the `ORDER BY` clause.

```
USE hive_metastore.class_013_odg7_da_dawd;
SELECT loyalty_segment, count(loyalty_segment) FROM customers GROUP BY loyalty_segment ORDER BY
loyalty_segment;
```

Copy

Column Expressions

Mathematical Expressions of Two Columns

In our queries, we can run calculations on the data in our tables. This can range from simple mathematical calculations to more complex computations involving built-in functions.

7. Run the code below.

This code displays three columns from the `silver_promo_prices` table. If the calculation in the `discounted_price` column was done correctly, it should equal the `sales_price` minus the `promo_disc`.

```
USE hive_metastore.class_013_odg7_da_dawd;
SELECT sales_price, promo_disc, discounted_price FROM promo_prices;
```

Copy

We can check the proper calculation of the discounted price by performing a calculation.

8. Run the code below.

The results show that the Calculated Discount, the one we generated using Column Expressions, matches the Discounted Price.

```
USE hive_metastore.class_013_odg7_da_dawd;
SELECT sales_price - sales_price * promo_disc AS Calculated_Discount, discounted_price AS
Discounted_Price FROM promo_prices;
```

Copy

Built-In Functions -- String Column Manipulation

There are many, many Built-In Functions. We are going to talk about just a handful, so you can get a feel for how they work.

9. Run the code below.

Take a look at the `city` column. The city names are mostly in all capital letters, and some are mixed case. We want to get them all into mixed case format.

```
USE hive_metastore.class_013_odg7_da_dawd;
SELECT * FROM customers;
```

Copy

We are going to use a built-in function called `lower()`. This function takes a string expression and returns the same expression with all characters changed to lowercase. Let's have a look.

10. Run the code below.

Although the letters are now all lowercase, they are not the way the need to be. We want to have the first letter of each word capitalized.

```
USE hive_metastore.class_013_odg7_da_dawd;
SELECT lower(city) AS City FROM customers;
```

Copy

Let's add a second built-in function, `initcap()`. This function also takes a string expression and returns the same expression with the first character in each word changed to uppercase. We are going to nest the two functions so that the string expression returned by `lower()` is used as the input to `initcap()`.

11. Run the code below.

Now, the city names are correctly capitalized.

```
USE hive_metastore.class_013_odg7_da_dawd;
SELECT initcap(lower(city)) AS City FROM customers;
```

Copy

Date Functions

In the `promo_prices` table, there is a column called `promo_began`.

12. Run the code below.

We want to use a function to make the date more human-readable.

```
USE hive_metastore.class_013_odg7_da_dawd;
SELECT * FROM promo_prices;
```

Copy

Let's use `from_unixtime()`.

13. Run the code below.

The date looks better, but let's adjust the formatting. Formatting options for many of the date and time functions are available here.

```
USE hive_metastore.class_013_odg7_da_dawd;
SELECT from_unixtime(promo_began, 'd MMM, y') AS Beginning_Date FROM promo_prices;
```

Copy

Date Calculations

Let's determine how long a specific promotion has been going by running a date calculation on the `promo_began` column.

14. Run the code below.

In this code, we are using the function `current_date()` to get today's date. We are then nesting `from_unixtime()` inside `to_date` in order to convert `promo_began` to a date object. We can then run the calculation.

```
USE hive_metastore.class_013_odg7_da_dawd;
SELECT current_date() - to_date(from_unixtime(promo_began)) FROM promo_prices;
```

Copy

CASE ... WHEN

Often, it is important for us to use conditional logic in our queries. `CASE ... WHEN` provides us this ability.

15. Run the code below.

This statement allows us to change numeric values that represent loyalty segments into human-readable strings. It is certainly true that this association would more-likely occur using a join on two tables, but we can still see the logic behind `CASE ... WHEN`

```
USE hive_metastore.class_013_odg7_da_dawd;
SELECT customer_name, loyalty_segment,
    CASE
        WHEN loyalty_segment = 0 THEN 'Rare'
        WHEN loyalty_segment = 1 THEN 'Occasional'
        WHEN loyalty_segment = 2 THEN 'Frequent'
        WHEN loyalty_segment = 3 THEN 'Daily'
    END AS Loyalty
FROM customers;
```

Copy

Updating Data

So far, all of the changes we have made to our data have only been in the results set. We haven't made any actual changes to our tables. In this section, we are going to run some command that will make changes to the data in the tables.

16. Run the code below.

Recall that earlier we looked at capitalization in the city names in our `customers` table. Note that those changes were not implemented in the table itself.

```
USE hive_metastore.class_013_odg7_da_dawd;
SELECT city FROM customers;
```

Copy

UPDATE

Let's make those changes.

17. Run the code below.

The `UPDATE` does exactly what it sounds like: It updates the table based on the criteria specified.

```
USE hive_metastore.class_013_odg7_da_dawd;
UPDATE customers SET city = initcap(lower(city));
SELECT city FROM customers;
```

Copy

INSERT INTO

In addition to updating data, we can insert new data into the table.

18. Run the code below.

We can see that there are four loyalty segments in our table.

```
USE hive_metastore.class_013_odg7_da_dawd;
SELECT * FROM loyalty_segments;
```

Copy

`INSERT INTO` is a command for inserting data into a table.

19. Run the code below.

We run the `INSERT INTO` command, and `SELECT` from the table and see the newly inserted data.

```
USE hive_metastore.class_013_odg7_da_dawd;
INSERT INTO loyalty_segments
    (loyalty_segment_id, loyalty_segment_description, unit_threshold, valid_from, valid_to)
    VALUES
    (4, 'level_4', 100, current_date(), Null);
SELECT * FROM loyalty_segments;
```

Copy

INSERT TABLE

`INSERT TABLE` is a command for inserting entire tables into other tables. There are two tables `suppliers` and `source_suppliers` that currently have the exact same data. Let's run a `SELECT` and take a look at the data.

20. Run the code below.

```
USE hive_metastore.class_013_odg7_da_dawd;
SELECT * FROM suppliers;
```

Copy

Note the number of rows. Now, let's insert the `source_suppliers` table.

21. Run the code below.

After selecting from the table again, we note that the number of rows has doubled. This is because `INSERT TABLE` inserts all data in the source table, whether or not there are duplicates.

```
USE hive_metastore.class_013_odg7_da_dawd;
INSERT INTO suppliers TABLE source_suppliers;
SELECT * FROM suppliers;
```

Copy

INSERT OVERWRITE

If we want to completely replace the contents of a table, we can use `INSERT OVERWRITE`.

22. Run the code below.

After running `INSERT OVERWRITE` and then retrieving a `count(*)` from the table, we see that we are back to the original count of rows in the table. `INSERT OVERWRITE` has replaced all the rows.

```
USE hive_metastore.class_013_odg7_da_dawd;
INSERT OVERWRITE suppliers TABLE source_suppliers;
SELECT * FROM suppliers;
```

Copy

Subqueries

Let's create two new tables.

23. Run the code below.

These two command use subqueries to `SELECT` from the `customers` table using specific criteria. The results are then fed into `CREATE OR REPLACE TABLE` and `CREATE OR REPLACE TABLE` statements. Incidentally, this type of statement is often called a `CTAS` statement for `CREATE OR REPLACE TABLE ... AS`.

```
USE hive_metastore.class_013_odg7_da_dawd;
CREATE OR REPLACE TABLE high_loyalty_customers AS
    SELECT * FROM customers WHERE loyalty_segment = 3;
CREATE OR REPLACE TABLE low_loyalty_customers AS
    SELECT * FROM customers WHERE loyalty_segment = 1;
```

Copy

Joins

We are now going to run a couple of `JOIN` queries. The first is the most common `JOIN`, an `INNER JOIN`. Since `INNER JOIN` is the default, we can just write `JOIN`.

24. Run the code below.

In this statement, we are joining the `customers` table and the `loyalty_segments` tables. When the `loyalty_segment` from the `customers` table matches the `loyalty_segment_id` from the

`loyalty_segments` table, the rows are combined. We are then able to view the `customer_name`, `loyalty_segment_description`, and `unit_threshold` from both tables.

```
USE hive_metastore.class_013_odg7_da_dawd;
SELECT
        customer_name,
        loyalty_segment_description,
        unit_threshold
    FROM
        customers
    INNER JOIN loyalty_segments
        ON customers.loyalty_segment = loyalty_segments.loyalty_segment_id;
```

Copy

CROSS JOIN

Even though the `CROSS JOIN` isn't used very often, I wanted to demonstrate it.

25. Run the code below.

First of all, note the use of `UNION ALL`. All this does is combine the results of all three queries, so we can view them all in one results set. The `Customers` row shows the count of rows in the `customers` table. Likewise, the `Sales` row shows the count of the `sales` table. `Crossed` shows the number of rows after performing the `CROSS JOIN`.

```
USE hive_metastore.class_013_odg7_da_dawd;
SELECT
  "Sales", count(*)
FROM
  sales
UNION ALL
SELECT
  "Customers", count(*)
FROM
  customers
UNION ALL
SELECT
  "Crossed", count(*)
FROM
  customers
  CROSS JOIN sales;
```

Copy

Aggregations

Now, let's move into aggregations. There are many aggregating functions you can use in your queries. Here are just a handful.

26. Run the code below.

Again, we are viewing the results of a handful of queries using a `UNION ALL`.

```
USE hive_metastore.class_013_odg7_da_dawd;
SELECT
        "Sum" Function_Name, sum(units_purchased) AS Value
    FROM customers
    WHERE state = 'CA'
UNION ALL
SELECT
        "Min", min(discounted_price) AS Lowest_Discounted_Price
    FROM promo_prices
UNION ALL
SELECT
        "Max", max(discounted_price) AS Highest_Discounted_Price
    FROM promo_prices
UNION ALL
SELECT
        "Avg", avg(total_price) AS Mean_Total_Price
    FROM sales
UNION ALL
SELECT
        "Standard Deviation", std(total_price) AS SD_Total_Price
    FROM sales
UNION ALL
SELECT
        "Variance", variance(total_price) AS Variance_Total_Price
        FROM sales;
```

Copy

1.2.13