SQL Training

Source: 10 SQL Tricks Every Data Scientist Should Know - Part 1 | Data Talks with Kat

```
CREATE TABLE toys (
	ID_VAR VARCHAR(10),
	SEQ_VAR	INT,
	EMPTY_STR_VAR VARCHAR(20),
	NULL_VAR VARCHAR(20),
	NA_STR_VAR  VARCHAR(20),
	NUM_VAR	Numeric,
	DATE_VAR1 DATE,
	DATE_VAR2 DATE
)

COPY toys
FROM 'D:\Toys.csv'
DELIMITER ','
CSV HEADER


SELECT * FROM toys
```

1. **– handling null values**

```
SELECT COALESCE(null, null, 101, null, 102)

-- handling null, empty or na values in sql
SELECT
	ID_VAR,
	SEQ_VAR,
	COALESCE(EMPTY_STR_VAR, 'MISSING') AS RECODED_EMPTY_STR_VAR,
	COALESCE(NULL_VAR, 'MISSING') AS RECODED_NULL_VAR,
	COALESCE(NA_STR_VAR, 'MISSING') AS RECODED_NA_STR_VAR
FROM toys

-- limitation of COALESCE, it will not work with missing (empty strings) or NA values
-- so we will use standard CASE STATEMENT

SELECT
	ID_VAR,
	SEQ_VAR,
	CASE WHEN EMPTY_STR_VAR = ' ' THEN 'EMPTY_MISSING' ELSE EMPTY_STR_VAR END AS
CE_EMPTY_STR_VAR,
	CASE WHEN NULL_VAR IS NULL THEN 'MISSING' ELSE NULL_VAR END AS CE_NULL_VAR,
	CASE WHEN NA_STR_VAR = 'NA' THEN 'NA_MISSING' ELSE NA_STR_VAR END AS CE_NA_STR_VAR
FROM toys
```

2. **–cumulative sum and cumulative sum/frequency**

```
SELECT
        id_var, num_var, date_var1,
        SUM(num_var) OVER (ORDER BY date_var1 ROWS BETWEEN 2 PRECEDING AND CURRENT ROW) AS
cum_sum
FROM toys
WHERE id_var = '19064'


SELECT
  DAT.NUM_VAR, cum_sum,
  SUM(NUM_VAR) OVER (PARTITION BY JOIN_ID) AS TOTAL_SUM,
  ROUND(CUM_SUM / SUM(NUM_VAR) OVER (PARTITION BY JOIN_ID), 4) AS CUM_FREQ
FROM
(
  SELECT
        T.*,
        SUM(NUM_VAR) OVER (ORDER BY NUM_VAR ROWS UNBOUNDED PRECEDING) AS CUM_SUM,
        CASE WHEN ID_VAR IS NOT NULL THEN '1' END AS JOIN_ID
  FROM toys    T
) DAT
--WHERE  ID_VAR = '19228'
ORDER BY CUM_FREQ;




SELECT
        ID_VAR,
        NUM_VAR,
        CUM_SUM,
        (select SUM(NUM_VAR) FROM toys) AS TOTAL_SUM,
        ROUND(cum_sum / (SELECT SUM(NUM_VAR) FROM toys),4) as freq
FROM
(
        SELECT
                T.*,
                SUM(NUM_VAR) OVER (ORDER BY NUM_VAR ROWS BETWEEN UNBOUNDED PRECEDING
AND CURRENT ROW) AS cum_sum
        FROM toys T ) AS t1
        --where id_var = '19228'
        order by freq
```

3. **–find records with extreme values without self join**

   --max value for each user id and it’s respective values

--with join:

```
        select *
        from toys t1
        inner join
```

```
        (
        SELECT
                id_var,
                MAX(num_VAR) as max_value
        from toys
        group by id_var
) as t2
        on t1.id_var = t2.id_var
        and t1.num_var = t2.max_value
```

--without self join:

```
SELECT *
FROM
(
        SELECT *,
        CASE WHEN (num_var = MAX(num_var) OVER (PARTITION BY Id_var)) THEN 'Y' ELSE 'N' END AS max_col
        FROM Toys) AS T1
WHERE T1.max_col = 'Y'
```

4. **conditional WHERE clause:**

You have a table named Toys with columns ID_VAR, SEQ_VAR, date_var1, and date_var2. Write a SQL query that calculates the difference in days between date_var2 and date_var1 (as d_diff) for each row. The query should only include rows where the difference (d_diff) meets the following conditions based on the SEQ_VAR value:

If SEQ_VAR is 1, 2, or 3, then d_diff must be greater than or equal to 0
If SEQ_VAR is 4, 5, or 6, then d_diff must be greater than or equal to 1
For all other values of SEQ_VAR, d_diff must be greater than or equal to 2

```
SELECT *,
(date_var2 - date_var1) as d_diff
FROM Toys
WHERE (date_var2 - date_var1) >= CASE WHEN SEQ_VAR IN (1,2,3) THEN 0
                                      WHEN SEQ_VAR IN (4,5,6) THEN 1
                                     ELSE 2 END
ORDER BY ID_VAR, SEQ_VAR
```

--ranking functions:

```
SELECT ID_VAR,  DATE_VAR1,
RANK() OVER (PARTITION BY ID_VAR ORDER BY DATE_VAR1) AS RANKK,
DENSE_RANK() OVER (PARTITION BY ID_VAR ORDER BY DATE_VAR1) AS DENSERANKK,
ROW_NUMBER() OVER (PARTITION BY ID_VAR ORDER BY DATE_VAR1) AS ROWNUM
FROM Toys
WHERE ID_VAR = '19064'
```

Source: 10 SQL Tricks Every Data Scientist Should Know - Part 2 | Data Talks with Kat

```
CREATE TABLE toys2 (
        ID_Var VARCHAR(10),
        Num_Var Numeric,
        Gen_Var VARCHAR(10),
        Date_Var DATE
)

COPY toys2
FROM 'D:\Toys2.csv'
DELIMITER ','
CSV HEADER


SELECT * FROM toys2
```

5. Difference in money spent comparing two consecutive logins for each customer

```
SELECT *,
        num_var - prev_var AS Num_Diff
FROM
(
        SELECT *,
        LAG(num_var,1,0) OVER (PARTITION BY id_var ORDER BY date_var) as Prev_var
        FROM toys2
) AS t2
ORDER BY id_var, date_var
```

--5b. Find the list of customers who have ordered in consecutive days (eg- order made on 1 Jan and 2 Jan)

```
    SELECT *,
    date_var - date2 AS date_count
    FROM
    (
        SELECT * ,
                LAG(date_var) OVER (PARTITION BY Id_var ORDER BY date_var) AS date2
        FROM toys2
    ) AS t2
    WHERE date2 IS NOT NULL
    AND (date_var - date2) = 1
    ORDER BY id_var, date_var
```

6. **find the count of products purchased by each customer in the below table:**

CustID	Prod 1	Prod 2	Prod 3	Prod 4	Prod 5	Prod 6	Prod 7
1	boat	book	cat	cat	dog		boat
2	cat	dog	book	cat	bat	book	
3	dog		boat	book		cat	
4	pen	cat	dog	book		pen	book

```
CREATE TABLE Prod (
   CustID INT,
   Prod1 VARCHAR(50),
   Prod2 VARCHAR(50),
   Prod3 VARCHAR(50),
   Prod4 VARCHAR(50),
   Prod5 VARCHAR(50),
   Prod6 VARCHAR(50),
   Prod7 VARCHAR(50)
);


INSERT INTO Prod (CustID, Prod1, Prod2, Prod3, Prod4, Prod5, Prod6, Prod7) VALUES
(1, 'boat', 'book', 'cat', 'cat', 'dog', NULL, 'boat'),
(2, 'cat', 'dog', 'book', 'cat', 'bat', 'book', NULL),
(3, 'dog', NULL, 'boat', 'book', NULL, 'cat', NULL),
(4, 'pen', 'cat', 'dog', 'book', NULL, 'pen', 'book');


SELECT * FROM Prod


SELECT
        CustID,
        ProductName,
        COUNT(*) AS TotalProduct
FROM
(
        SELECT CustID, prod1 AS ProductName FROM prod WHERE prod1 IS NOT null
        UNION ALL
        SELECT CustID, prod2 AS ProductName FROM prod WHERE prod2 IS NOT null
        UNION ALL
        SELECT CustID, prod3 AS ProductName FROM prod WHERE prod3 IS NOT null
        UNION ALL
        SELECT CustID, prod4 AS ProductName FROM prod WHERE prod4 IS NOT null
        UNION ALL
        SELECT CustID, prod5 AS ProductName FROM prod WHERE prod5 IS NOT null
        UNION ALL
        SELECT CustID, prod6 AS ProductName FROM prod WHERE prod6 IS NOT null
        UNION ALL
        SELECT CustID, prod7 AS ProductName FROM prod WHERE prod7 IS NOT null
) AS AllProduct
GROUP BY CustID, ProductName
ORDER BY CustID
```

7. **Find the spending top customers in each month for the below Customer table:**

```
CREATE TABLE Customer (
  CustID INT,
  Purchase_Date DATE,
  Purchase_Price DECIMAL(10, 2) -- Allows for prices up to 99999.99
);

INSERT INTO Customer (CustID, Purchase_Date, Purchase_Price)
VALUES (1, '2024-01-24', 150),
      (1, '2024-01-26', 120),
      (2, '2024-01-10', 200),
      (3, '2024-01-30', 50),
      (1, '2024-02-10', 250),
      (2, '2024-02-12', 55),
      (2, '2024-02-20', 85),
      (3, '2024-02-15', 120),
      (1, '2024-03-10', 45),
      (2, '2024-03-20', 75),
      (3, '2024-03-30', 175),
      (3, '2024-03-21', 300);


      WITH CTE AS
      (
          SELECT
                  CustID,
                  EXTRACT(MONTH FROM Purchase_Date) AS Months,
                  EXTRACT(YEAR FROM Purchase_Date) AS Years,
                  SUM(Purchase_Price) AS Total_Price
          FROM Customer c1
          GROUP BY CustID,Months, Years
      ),
      Rank_CTE AS
      (
          SELECT *,
          RANK() OVER (PARTITION BY Months, Years ORDER BY Total_Price DESC) AS Top_Cust
          FROM CTE
      )
          SELECT *
          FROM Rank_CTE
          WHERE Top_Cust = 1
```

8. What is difference between JOIN VS UNION VS INTERSECT

Key Differences

Feature	JOINs	UNION	INTERSECT
Purpose	Combine rows from two or more tables based on a related column (primary or foreign key)	Combine result sets of two or more queries into a single set	Return common rows from two or more queries
Result Set	Combines columns horizontally	Combines rows vertically	Returns rows common to both queries
Duplicate Handling	Includes all matched rows (depends on join type)	Removes duplicates (default)	Removes duplicates
Column Requirements	Related columns must have compatible data types	Same number of columns with compatible data types	Same number of columns with compatible data types
Types	INNER JOIN, LEFT JOIN, RIGHT JOIN, FULL JOIN	UNION, UNION ALL	INTERSECT
Use Case	Combine related data from different tables	Combine results from similar tables or queries	Find common data between queries
Order of Columns	Can differ; based on the SELECT clause	Must be the same in each query	Must be the same in each query
Performance	Depends on join type and indexing	UNION ALL is faster than UNION (no deduplication)	Can be slower due to duplicate removal
Example Use Case	Fetching orders with corresponding customer information	Merging customer lists from different regions	Identifying customers who made purchases in multiple years

9. Check data type of a column

One of the easiest ways will be t just query SELECT statement and use LIMIT 5, and then from the output we can see the data types for each column in the header. If you want you can **SELECT** specific columns also.

custid integer	purchase_date date	purchase_price numeric (10,2)
1	2024-01-24	150.00
1	2024-01-26	120.00
2	2024-01-10	200.00

Query to check data type:

You can use the **information_schema.columns** view to check the data type of a column.

```
SELECT column_name, data_type
FROM information_schema.columns
WHERE table_name = 'your_table_name'
  AND column_name = 'your_column_name';    -- can also use IN operator for multiple column names
```

10. Write a query to return the users having logins in both NYC and Illinois

```
-- Create the table
CREATE TABLE logins (
   UserId VARCHAR(10),
   PurchaseAmt DECIMAL(10, 2),
   UserName VARCHAR(50),
```

```
   LoginDate DATE,
   LoginLocation VARCHAR(50)
);

-- Insert data into the table
INSERT INTO logins (UserId, PurchaseAmt, UserName, LoginDate, LoginLocation) VALUES
('0155', 14.35, 'ABC', '2018-06-03', 'California'),
('0108', 282.89, 'XYZ', '2016-07-31', 'Illinois'),
('0155', 60.79, 'ABC', '2018-06-01', 'Massachusetts'),
('0155', 0.00, 'ABC', '2018-06-02', 'Nevada'),
('0155', 79.77, 'ABC', '2018-09-29', NULL),
('0155', 122.82, 'ABC', '2018-09-30', 'Arizona'),
('0180', 1810.47, 'DEF', '2016-06-30', 'NYC'),
('0188', 732.62, 'GHI', '2016-06-30', 'Massachusetts'),
('0188', 2782.89, 'GHI', '2016-07-31', 'Illinois'),
('0188', 2989.38, 'GHI', '2016-08-31', 'NYC'),
('0792', 721.36, 'JKL', '2016-06-30', 'California'),
('0792', 817.94, 'JKL', '2016-07-01', 'NYC'),
('0792', 886.17, 'JKL', '2016-08-31', 'NYC'),
('0792', 954.71, 'JKL', '2016-09-30', 'Illinois'),
('0792', 1048.69, 'JKL', '2016-10-31', 'NYC');


SELECT * from logins

--Solution:A

SELECT userid, username
FROM logins
WHERE loginlocation IN ('NYC', 'Illinois')
GROUP BY userid, username
HAVING COUNT(DISTINCT loginlocation) = 2


--Solution:B

SELECT *
FROM
(
        select userid, username,
        SUM(CASE WHEN loginlocation = 'NYC' THEN 1 ELSE 0 END) AS nyc_loc,
        SUM(CASE WHEN loginlocation = 'Illinois' THEN 1 ELSE 0 END) AS ill_loc
        FROM logins
        GROUP BY userid, username
) AS l1
WHERE nyc_loc > 0
AND ill_loc > 0
```

11. Histogram bucket creation using WIDTH_BUCKET

```
-- Step 1: Create the table with random values

CREATE TABLE sal_data (
   customer_id SERIAL PRIMARY KEY,
   salary INT
);

-- Step 2: Insert random salaries into the table

INSERT INTO sal_data (salary)
SELECT ROUND((RANDOM() * 10000)::INT)
FROM generate_series(1, 108);


-- Optional: View the inserted data
SELECT * FROM sal_data
order by salary

--Width bucket basic query:
```

WIDTH_BUCKET

Constructs equi-width histograms, in which the histogram range is divided into intervals of identical size, and returns the bucket number into which the value of an expression falls, after it has been evaluated. The function returns an integer value or null (if any input is null).

-- **WIDTH_BUCKET(<expr> , <min_value> , <max_value> , <num_buckets>)**

```
SELECT
        salary,
        WIDTH_BUCKET(salary, 0, 10000, 5) AS bucket
FROM sal_data
order by salary


SELECT
        WIDTH_BUCKET(salary, 0, 10000, 5) AS bucket,
        COUNT(*) sal_count
FROM sal_data
GROUP BY bucket
ORDER BY bucket


SELECT
        salary,
        WIDTH_BUCKET(salary, 0, 10000, 5) AS bucket,
        NTILE(5) OVER (Order by Salary) AS Tile  --divides data into equal number of groups
FROM sal_data
order by salary
```

Automate the minimum and maximum values in WIDTH_BUCKET function and determine the optimized number of buckets, using CTEs.

```
WITH salary_stats AS (
   SELECT
      MIN(salary) AS min_salary,
      MAX(salary) AS max_salary,
      COUNT(*) AS total_count
   FROM sal_data
),
optimized_bucket AS (
   SELECT
      min_salary,
      max_salary,
      CEIL((max_salary - min_salary) / 1000.0) AS bucket_count
   FROM salary_stats
)
SELECT
   WIDTH_BUCKET(salary::int, min_salary::int, max_salary::int, bucket_count::int) AS bucket,
   COUNT(*) AS sal_count
FROM sal_data, optimized_bucket
GROUP BY bucket
ORDER BY bucket;
```