CLEANING DATA IN SQL

THE SELECT QUERY: SYNTAX REVIEW

- ✓ **SELECT** picks the columns.
- ✓ **FROM** points to the table.
- ✓ WHERE puts filters on rows.

FROM & JOINs determine & filter rows
WHERE more filters on the rows
GROUP BY combines those rows into groups
HAVING filters groups
ORDER BY arranges the remaining rows/groups

- ✓ GROUP BY aggregates across values of a variable.
- **✓ HAVING** filters aggregated values.
- ✓ **ORDER BY** sorts the results.
- ✓ **LIMIT** limits result to the first **n** rows.

LEARNING OBJECTIVES

- Explain the differences between NULL and zero.
- Explore the issues of math equations and NULLs.
- Use SQL **NULL** to create Boolean functions and handle zeros.
- Use **CASE** statements to add "**IF THEN ELSE**" logic to SQL.

INTRODUCTION: WORKING WITH NULLs

WORKING WITH NULLS

A **NULL** represents missing data, but a **NULL** is different than a zero or a blank.

- A blank cell could have been left blank on purpose.
- In some cases, a blank represents data.

When **NULLs** exist in a data set that was the result of joined tables, the presence of a **NULL** often has meaningful implications.

What is your company's best practice or policy for handling **NULLs**? If your company doesn't have one, what do you think you should do?



NULLS AND MATH

It can be tricky working with **NULLs** and zeros if the analysis requires addition/subtraction or division.

- No dividing by zero.
- Can't add or subtract a **NULL**.

Various SQL tools can solve either situation. For example:

- Using a CASE statement, you can change zeros to NULLs.
- **NULLIF** can substitute a **NULL** for a zero value, allowing division equations to execute without an error condition.

EXAMPLE: NULLIF

NULLIF(field, testing_value)

- Returns NULL if expressions are equal.
- Otherwise returns the first expression.
- ✓ **NULLIF** can test for zero values.

Example: Using the sample table on the right, divide Field 1 by Field 2. You must ensure there are no zeros in Field 2:

Field1	Field2
15	1
20	0
25	5
30	3

WORKING WITH NULLS

IFNULL is the opposite of **NULLIF**. This function takes a **NULL** value and turns it into a zero or another appropriate value.

When adding and subtracting values, SQL will not allow you to add or subtract **NULLs**, but will allow you to add and subtract numbers.

You can use this function to fill in averages where there are **NULLs**.

For example, IFNULL(Field1, AVG(Field2)).

Pro Tip: PostgreSQL does not have **IFNULL**. Instead, use **COALESCE** or CASE.

WORKING WITH NULLS

COALESCE returns the first of its arguments that is **not null**. This is implemented within PostgreSQL to provide capabilities similar to **IFNULL**.

Example syntax:

```
SELECT COALESCE(description, short_description, '(none)')
SELECT COALESCE(field2,alternate_value, 0)
```

GUIDED PRACTICE: NULLS AND JOINS

NULLS AND JOINS

You can find **NULL** data using the **WHERE** clause.

Run:

```
SELECT vendor_name, item_description FROM products WHERE
upc = 'NULL';
```

Do you expect problems with this statement? Why or why not?

NULLS AND JOINS

To find **NULL** values, we use the SQL function "**IS NULL**."

Run:

SELECT vendor_name, item_description, upc FROM products WHERE upc IS NULL;

This will return all of the rows where the UPC field is unknown.

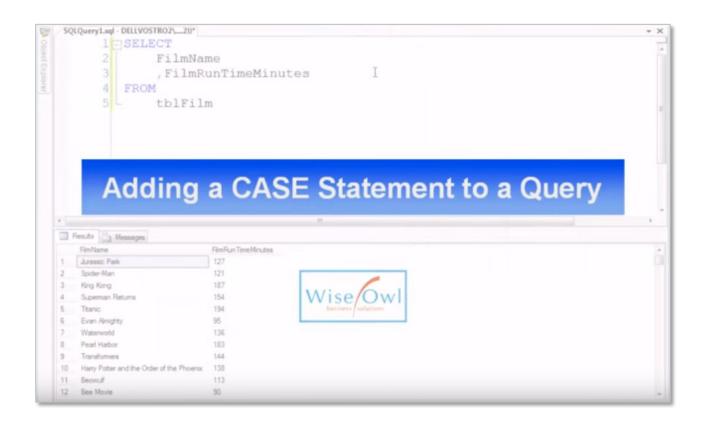
NULLS AND JOINS

We can also find non-NULL values by using **IS NOT NULL** – the opposite of **IS NULL**.

Run:

```
SELECT vendor_name, item_description, upc
FROM products WHERE upc IS NOT NULL;
```

INTRODUCTION: CASE STATEMENTS



CASE statements group data into **categories** or **classifications**.

In SQL, CASE is part of the columns list in the SELECT statement.

CASE syntax looks like this:

```
SELECT columns,

CASE

WHEN condition THEN result

WHEN condition THEN result

ELSE result

END AS output_name

FROM table;
```

GUIDED PRACTICE: CASE STATEMENTS

Let's say we want to classify Iowa's counties as either:

- Small (less than 100,000 people).
- Medium (100,000-to-400,000 people).
- Large (more than 400,000 people).

How would we go about this?

```
SELECT county, population,
   CASE
   WHEN population >= 400000 THEN 'large'
   WHEN population >= 100000 AND population < 400000
THEN 'medium'
   WHEN population < 100000 THEN 'small'
   END AS county_size
FROM counties;</pre>
```

Note that we define each size category. SQL will only return values that meet these conditions.

We can also add an "other" category if we want to include an **ELSE** 'other' before the **END** AS county_size line. This will capture any missing data.

Pro Tip: while the use of indentation is helpful, it's not necessary in SQL.

Next we'll look at an example of **CASE** being used within a **SELECT** statement to generate an **aggregate**.

Calculate the percentage of items in the liquor product offerings that are whiskey products.

```
SELECT AVG(
CASE
WHEN category_name LIKE '%WHISK%' THEN 1
ELSE 0
END)
AS AverageWhisky
FROM products;
```

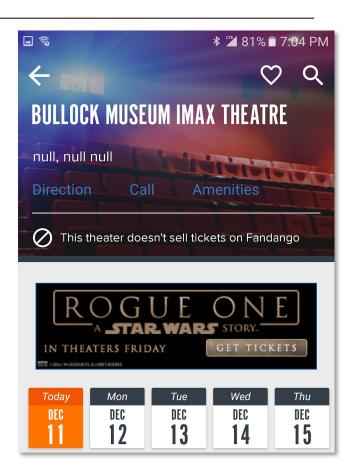
If we want a checksum for this formula, we can use:

```
select sum(case when category_name LIKE '%WHISK%' THEN 1
ELSE 0 end) as whiskey_product_sales
, count(1) as total_sales
, 1.00*(sum(case WHEN category_name LIKE '%WHISK%' THEN
1 ELSE 0 end))/count(1) as pct_whiskey_sales
from products
```

REAL-WORLD NULLS

What happens when we don't deal with **Nulls** effectively?

What could they have done differently?



INDEPENDENT PRACTICE: PUTTING IT ALL TOGETHER

ACTIVITY: PUTTING IT ALL TOGETHER

DIRECTIONS



Create one query that:

- 1. Identifies the items that were sold (by item number) that don't have matching reference information in the Products table (based on item_no NULLs).
 - a. In the output, include a count of how many distinct stores sold these items.

Create another query that:

- 2. Lists the distinct store numbers with recorded sales but no listing in the Stores table.
 - b. Use a CASE and COALESCE combination to add a column that labels these records as either a liquor store or convenience store.

SAMPLE SOLUTION

1. Identify the items that were sold (by item number) that do not have matching reference information in the Products table (based on item_no NULLs). In your output, create a count of how many distinct stores sold these items.

Solution: 88 rows.

```
SELECT a.item, COUNT(DISTINCT(a.store)) AS number_of_stores
FROM sales a
LEFT OUTER JOIN products b
ON a.item = b.item_no
WHERE b.item_no IS NULL
GROUP BY a.item
ORDER BY 1
```

SAMPLE SOLUTION

2. Create a list of the distinct store numbers with recorded sales but no listing in the Stores table. Practice using a **CASE** and **COALESCE** combination to add a column that labels them as either a liquor store or convenience store.

Solution: 31 rows.

SELECT DISTINCT ON(store) store, CASE
WHEN a.convenience_store ilike 'Y' THEN 'Convenience'
ELSE COALESCE(a.convenience_store, 'Liquor store')
END AS Store_Type FROM sales a LEFT JOIN stores b
USING(store) WHERE b.store IS NULL GROUP BY store,
a.convenience_store;

ACTIVITY: PUTTING IT ALL TOGETHER



DIRECTIONS

Let's apply our SQL skills to this problem:

- Categorize all of the items by age:
 - Based on list date.
 - Ranges of zero-10 years, 11-20 years, 21-30 years, 31-40 years, and 41+ years.
- Then bring in the sum of total sales.

ACTIVITY: PUTTING IT ALL TOGETHER

DIRECTIONS



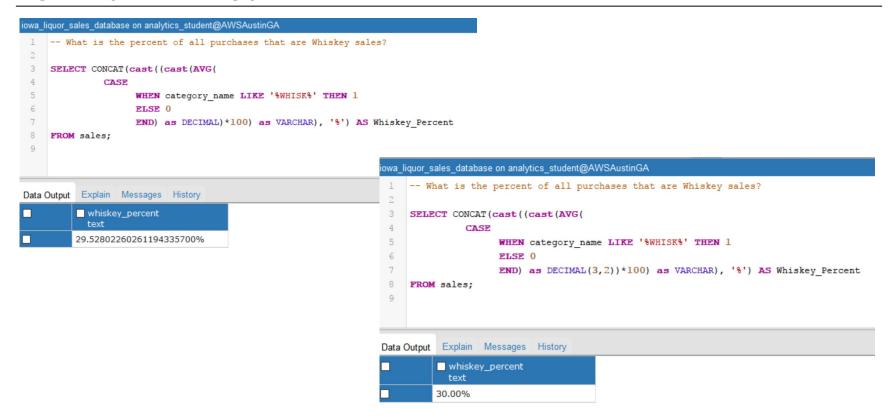
Combine your query tools (CONCAT, CASE, and other SQL formatting functions) to answer the following question from Iowa's liquor licensing board. Create a query with an output properly formatted as a percent with two decimal places and a percent notation.

What percentage of all purchases are whisky sales?

PUTTING IT ALL TOGETHER

```
iowa liquor sales database on analytics student@AWSAustinGA
     SELECT DISTINCT b.item no, b.item description, b.list date,
         CASE
             WHEN CAST(LEFT(CAST(AGE(CURRENT DATE, b.list date) as varchar), 2) as integer) BETWEEN 0 AND 10 THEN '0-10 YEARS'
             WHEN CAST(LEFT(CAST(AGE(CURRENT DATE, b.list date) as varchar),2) as integer) BETWEEN 11 AND 20 THEN '11-20 YEARS'
             WHEN CAST(LEFT(CAST(AGE(CURRENT DATE, b.list date) as varchar),2) as integer) BETWEEN 21 AND 30 THEN '21-30 YEARS'
             WHEN CAST(LEFT(CAST(AGE(CURRENT DATE, b.list date) as varchar),2) as integer) BETWEEN 0 AND 10 THEN '31-40 YEARS'
             ELSE '41+ YEARS'
         END AS Age Groupings,
         SUM(a.total) AS Sum Total
     FROM sales a
    JOIN products b
12 ON a.item = b.item no
    GROUP BY b.item no, b.item description
     ORDER BY 3
    LIMIT 1000;
16
Data Output Explain Messages History
          item_no  item_description
                                                    list date
                                                                                   age_groupings
                                                                                                                  sum tota
                                                      timestamp without time zone
            integer
                      text
                                                                                     text
                                                                                                                    numeric
              22156 Wild Turkey 101
1963-06-01 00:00:00
                                                                                   41+ YEARS
                                                                                                                   409684.22
             13036 Canadian Reserve Whisky
                                                                                   41+ YEARS
                                                                                                                    62570.88
                                                   1969-06-26 00:00:00
80576 Arrow Peppermint Schnapps
                                                   1969-07-01 00:00:00
                                                                                   41+ YEARS
                                                                                                                   54716.58
              39866 Smirnoff Vodka 100 Prf
                                                   1969-07-01 00:00:00
                                                                                   41+ YEARS
                                                                                                                   286299.65
             37416 Popov Vodka 80 Prf
                                                                                                                   97780.50
                                                   1969-07-01 00:00:00
                                                                                   41+ YEARS
```

PUTTING IT ALL TOGETHER



CONCLUSION

REVIEW: DATA AGGREGATION IN SQL

In this lesson, we:

- Explored methods of dealing with NULLs in tables, including: NULLIF, IS NULL, IS NOT NULL, and COALESCE.
- 2. Reviewed **CASE** statements for adding categories and aggregates, using Boolean logic structures.
- 3. Practiced evaluating and interpreting the presence of **NULLs** that occur as a result of joining tables together.

Q&A

RESOURCES



RESOURCES

- "Strategies for Approaching NULL Values with SQL Server:" https://goo.gl/wALy62
- "Handling Null Values," Microsoft SQL: https://goo.gl/4frfgz
- CASE Statement YouTube Video From WiseOwl: https://youtu.be/zlgrhj2D63E
- TechOnTheNet.com for SQL Server NULLIF Reference: <u>https://goo.gl/AFeKuV</u>