GENERAL ASSEMBLY

FILTERING AND AGGREGATION IN SQL

LEARNING OBJECTIVES

1. Apply commenting to code using `--` and `/* comment */`.
2. Use SQL conditional operators `=`, `!=`, `>`, `<`, `IN`, `NOT IN`, and `BETWEEN`.
3. Use the SQL Boolean operator `OR` to include only the desired data.
4. Introduce advanced SQL commands `GROUP BY` and `HAVING` to filter data.
5. Use aggregate functions `MIN`, `MAX`, `SUM`, `AVG`, and `COUNT`.
6. Apply calculations to fields using the order of operations.

LIGHTNING ROUND REVIEW

- Name two ways to access a quick view of the first 100 rows.
- What’s the SQL command used to designate data sources?
- What’s important about upper- and lowercase for stored data?
- How do you solve for comparing a number to text value?
- What punctuation is used to encapsulate a text string in SQL?
- Name two ways to see how many rows a query returns.

INTRODUCTION: COMMENTING CODE

HEADER COMMENTS

We’ll briefly walk through three types of comments you are likely to encounter :

1. Header Comments: `--`
2. In-Line Comments: `/* comment */`
3. Multi-Line Comments `/* comment, new line */`

HEADER COMMENTS

Header comments are a great way to keep a history of why a SQL query was built and who requested any changes that were made.

When commenting in the header of a file, we recommend following a standardized format, like this one:

```
/**********************
** NAME: Name of report.
** DESC: Description of report.
** AUTH: Name of author.
** DATE: Date report published.
***********************
**Change History
***********************
** Version | Date | Author | Description
***************************/
```

HEADER COMMENTS

In-line comments are usually used to explain calculations.

```
SELECT item_no /* This is an in-line comment about this query.*/
```

Multi-line comments are a good way to document more complicated processes and the reasoning behind them, but should be used sparingly.

```
SELECT item_no, description /*Here is an example multi-line
comment about this query.*/
```

WHERE CONDITIONS

WHERE CONDITIONS

- ✓ `SELECT`: *Selects* the columns.
- ✓ `FROM`: *Points* to the table.
- ✓ `WHERE:` *Filters* on rows
- ✓ `GROUP BY:` *Aggregates* across values of a variable.
- ✓ `HAVING:` *Filters* groups.
- ✓ `ORDER BY:` *Sorts or arranges* the results.
- ✓ `LIMIT:` *Limits* result to the first `n` rows.

WHERE CONDITIONS

WHERE statements filter and focus resulting information, and can be used to filter different types of data.

Remember, when combining **logical operators**, be aware of the **order of operations!** Use (parentheses) to create groupings or priorities.

WHERE CONDITIONS: ORDER OF OPERATIONS

Order of Operations = **BIMDAS**

B	**Brackets:** these go first.
I	**Indices:** powers and square roots.
MD	**Multiplication & Division:** left to right.
AS	**Addition & Subtraction:** left to right.

WHERE CONDITIONS

Ok, let’s learn about a few new operators and apply them to our Iowa data set:

`!=`	Not equal to.
`>, >=`	Greater than, greater than or equal to.
`<, <=`	Less than, less than or equal to.
`IN ( )`	Found in list of items.
`NOT`	Negates a condition.
`LIKE,ILIKE`	Contains item.
`BETWEEN`	Within the range of.
`%`	Wildcard.
`_`	Wildcard, single character.

GUIDED PRACTICE: WHERE CONDITIONS

WHERE CONDITIONS: GUIDED PRACTICE

Follow along with these queries for each of the new predicate operators:

!=

1. Which products are not from vendor ‘Jim Beam Brands’?

```
SELECT * FROM products WHERE vendor_name!='Jim Beam Brands’;
```

>, >=

1. Which products are higher than 90 proof?

```
SELECT * FROM products WHERE CAST(proof as INT) > 90;
```

<, <=

1. Which products have a case cost of less than $60?

```
SELECT * FROM products WHERE case_cost < 60;
```

WHERE CONDITIONS: GUIDED PRACTICE

Follow along with these queries for each of the new predicate operators:

IN

1. Which products are either single malt scotches or Canadian whiskies (based on category name)?

```
SELECT * FROM products WHERE category_name IN ('SINGLE MALT
SCOTCH', 'CANADIAN WHISKIES’);
```

WHERE CONDITIONS: GUIDED PRACTICE

Follow along with these queries for each of the new predicate operators:

LIKE, NOT LIKE

1. Which products have ‘whiskies’ in the category name?
```
SELECT * FROM products WHERE category_name LIKE '%WHISKIES';
```

2. Which products don’t have ‘whiskies’ in the category name?
```
SELECT * FROM products WHERE category_name NOT LIKE '%WHISKIES';
```

WHERE CONDITIONS: GUIDED PRACTICE

Follow along with these queries for each of the new predicate operators:

BETWEEN (boundary values)

1. Which products have a shelf_price between $4 and $10?
   ```
   SELECT * FROM products WHERE shelf_price   BETWEEN  4  AND  10;
   (2701 rows)
   ```

2. Which products have a bottle_price between $4 and $10?
   ```
   SELECT * FROM products WHERE CAST(bottle_price as DEC) BETWEEN 4 AND 10;
   (4011 rows)
   ```

INDEPENDENT PRACTICE: WHERE CONDITIONS

WHERE CONDITIONS: INDEPENDENT PRACTICE

EXERCISE

DIRECTIONS

For your new project at Deloitte, your boss has asked you a few more questions about the Iowa liquor data set.

Please write queries that answer the following questions:

1. Which products have a case cost of more than $100?
2. Which tequilas have a case cost of more than $100?
3. Which tequilas or scotch whiskies have a case cost of more than $100?
4. Which tequilas or scotch whiskies have a case cost between $100 and $120?
5. Which whiskies of any kind cost more than $100?
6. Which whiskies of any kind cost between $100 and $150?
7. Which products, excluding tequilas, cost between $100 and $120?

COMBINING LOGICAL OPERATORS

COMBINING LOGICAL OPERATORS

In some cases, **OR** is an effective way to include additional data. However, it may sometimes include more data than intended.

We want to see any Washington State sales or any sales in the United States greater than 500.

```
WHERE Country = ‘US’
AND State = ‘Washington’
OR cost > 500
```

This query will not properly answer the question. Why?

COMBINING LOGICAL OPERATORS

The presumption is you will only get results from the United States and everything in Washington State **OR** any states for which cost is greater than 500.

However, because the **OR** is not grouped, it will instead override the Country = US and bring back **ANY** country with sales greater than 500.

SOLUTION:

```
WHERE Country = ‘US’
AND (State = ‘Washington’
OR cost > 500)
```

INDEPENDENT PRACTICE

COMBINING LOGICAL OPERATORS

EXERCISE

DIRECTIONS

Your boss at Deloitte has another question for you to research:

> "From the Iowa Liquor Sales Database, I only want information about Vendor 305. Can you get me the bottle price and proof? The price should be less than 5 OR the proof should be greater than 100, either is fine."

COMBINING LOGICAL OPERATORS: SAMPLE SOLUTION

EXERCISE

DIRECTIONS

Your boss at Deloitte has another question for you to research:

> “From the Iowa Liquor Sales Database, I only want information about Vendor 305. Can you get me the bottle price and proof? The price should be less than 5 OR the proof should be greater than 100, either is fine.”

```
SELECT vendor, bottle_price, proof
FROM products
WHERE vendor = 305
  AND (cast(bottle_price AS decimal) <5
  OR cast(proof AS integer)>100);
```

AGGREGATIONS INTRODUCTION

AGGREGATIONS

There are many SQL commands that can be used to aggregate your data, such as: **`MIN`**, **`MAX`**, **`SUM`**, **`COUNT`**, and **`AVG`**.

A few valuable SQL commands that include aggregations are **`GROUP BY`** and **`HAVING`**:

- **`GROUP BY`** indicates the dimensions you want to group your data by (e.g. a category that you wish to sort into subgroups).
- **`HAVING`** is used to filter measures you've aggregated (e.g., to filter a `SUM` over a certain value).

AGGREGATIONS

In many cases, fields need to be **CAST** before an aggregation can be performed:

```
Count(CAST(field1 as INTEGER))
```

COUNT can be used as a checksum, evaluating scope, or for quantifying data that meet a specified criteria.

Aggregate functions often need a **GROUP BY**.

- Aggregates are typically performed on measurement fields: sales, miles, heights, etc.
- `GROUP BY` is performed on the dimensions that are being measured.

AGGREGATIONS

1. Let’s look at an example using a table called “people” with the fields “gender” and “height.”
2. You want to find the average height of people taller than 3 feet by gender.
3. Next, you want to determine which of those people have an average height greater than 5.5 feet tall, divided by gender.

The code should look something like this:

```
SELECT gender, AVG(height)
FROM people
WHERE height >3
GROUP BY gender
HAVING AVG(height) >5.5
ORDER BY gender;
```

AGGREGATIONS

The **ROUND** function returns the given number, rounded to **n** places to the right of the decimal point. If **n** is negative, it will be rounded to **n** places to the left of the decimal.

Syntax: `ROUND(numeric_expression, n)`

Example (assume x = 177.3589):

```
    ROUND(x, 2) would return 177.3600

    ROUND(x, -2) would return 200
```

GUIDED PRACTICE: AGGREGATIONS

AGGREGATIONS: GUIDED PRACTICE

Let’s practice building these together:

```
SELECT MAX(total) FROM sales;
SELECT AVG(state_btl_cost) FROM sales;
```

1. Is **CAST** necessary for successful execution? Why or why not?
2. Use **ROUND** to control the number of digits beyond the decimal.
3. Use **ROUND** to limit the average bottle cost to two decimals.

AGGREGATIONS: GUIDED PRACTICE

Let’s build one together that groups the results:

```
SELECT vendor,vendor_name,AVG(bottle_price)
FROM products
GROUP BY vendor_name
ORDER BY 3 DESC
```

1. Is **CAST** necessary for successful execution? Why or why not?
2. Use **ROUND** to control the number of digits beyond the decimal.
3. Use **ROUND** to limit the average bottle cost to two decimals.

INDEPENDENT PRACTICE: AGGREGATIONS

AGGREGATIONS: INDEPENDENT PRACTICE

EXERCISE

DIRECTIONS

Using our aggregation tools, find the largest and smallest bottle price offered by each vendor.

Starter Code:

```
SELECT vendor_name, (aggregate of large bottle_price),
       (aggregate of smallest bottle_price)
FROM products
GROUP BY vendor_name.
```

- Further refine the results to those that *have* a minimum bottle_price of $10.
- Sort the results by maximum bottle_price column.
- Limit the report data to the top 20.
- Keep in mind that the data type of bottle_price is *money*.

AGGREGATIONS: INDEPENDENT PRACTICE SOLUTION

EXERCISE

```
SELECT vendor_name,
   ROUND(MAX(CAST(bottle_price AS DEC)),2) AS
   max_cost,
   ROUND(MIN(CAST(bottle_price AS DEC)),2) AS
   min_cost

FROM products
GROUP BY vendor_name
HAVING ROUND(MIN(CAST(bottle_price AS DEC)),2) > 10
ORDER BY 2 DESC
LIMIT 20;
```

FILTERING AND AGGREGATION

MISSION IMPOSQL

MISSION IMPOSQL

Project description:

- Work through as many questions in the handout as you can before class ends
- With 5 minutes left, we'll review answers to each section

CONCLUSION

RECAP: FILTERING AND AGGREGATION IN SQL

Here's a quick review of what we covered today:

1. How to comment our code using `--` and `/* comment */`.
2. SQL conditional operators `=`, `!=`, `>`, `<`, **`IN`**, **`NOT IN`**, and **`BETWEEN`**.
3. How to use the SQL Boolean operator **`OR`** to include only the desired data.
4. Introducing SQL commands **`GROUP BY`** and **`HAVING`** to filter data.
5. Using aggregate functions **`MIN`**, **`MAX`**, **`SUM`**, **`AVG`**, and **`COUNT`**.
6. Appling calculations to fields using the order of operations.

Q&A

RESOURCES

RESOURCES

- SQL server logical operators: https://goo.gl/2Q9gmH
- "Query Results Using Boolean Logic" from essentialSQL: https://www.essentialsql.com/get-ready-to-learn-sql-server-query-results-using-boolean-logic/
- SQL `HAVING` clause overview: https://goo.gl/Je3M85
- "Difference between `WHERE`, `GROUP BY`, and `HAVING` clauses," by Manoj Pandey: https://goo.gl/cNCtBa