Data Analytics

Organizing Data with SQL

GENERAL ASSEMBLY

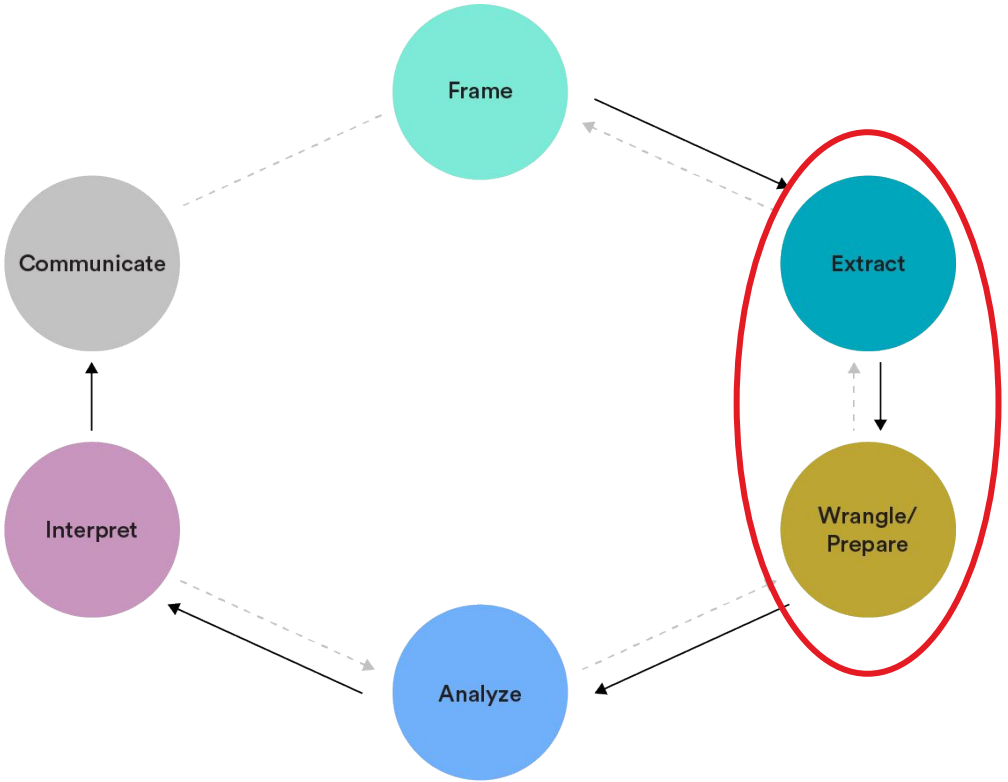
Our Learning Goals

- Navigate a relational database.
- Practice writing and executing SQL queries, including **SELECT**, **FROM**, **WHERE**, and **DISTINCT SELECT**.
- Work with **CASE** to handle if/then logic and apply multiple conditions.
- Practice writing aggregate functions: **MIN**, **MAX**, **SUM**, **AVG**, and **COUNT**.
- Use SQL commands such as **GROUP BY** and **HAVING** to group and filter data.
- Combine data from multiple sources using **JOINs** and **UNIONs**.

Where We Are in the DA Workflow

Extract: Select, import, and clean relevant data.

Wrangle/prepare: Clean and prepare relevant data.



Before We Begin...

The SQL database tool we'll use in this class is **PostgreSQL** because:

- It can be configured as an **object-oriented** or a **relational database** management system (RDBMS).
- It's powerful and standardized, used for enterprise databases in both public and private environments.
- It's free and open source!

Organizing Data with SQL

Getting Started With SQL

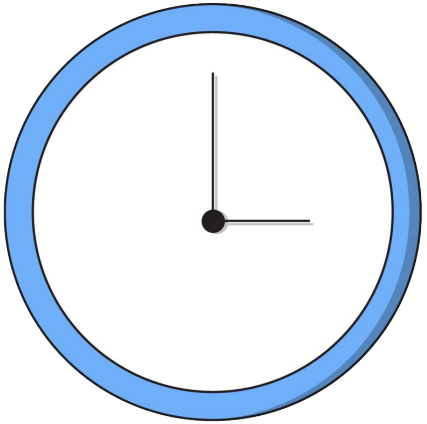
Discussion:

What Happens in an Internet Minute?

Within an **internet minute**, there are:

- 4.1 million Google searches
- 59 million instant messages sent
- 400,000 apps downloaded
- $1.1 million spent online

All of this (and more) happens within 60 seconds!



What can we do with all that data?

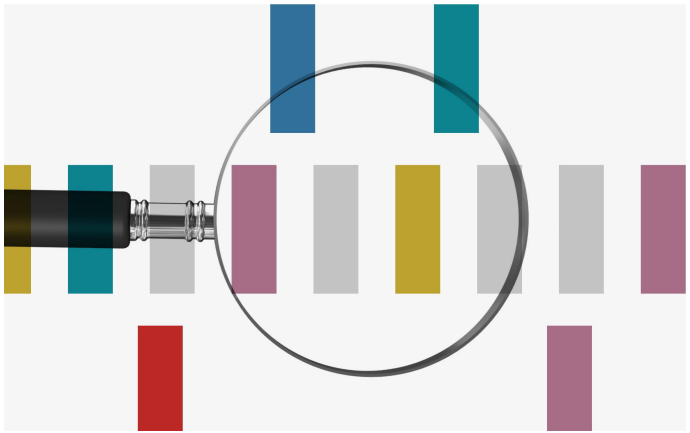
Image Source: Allaccess.com

What Is SQL?

SQL is short for **Structured Query Language**.

It's a language you can use to *query* information from your data. You can use it to ask questions and make requests such as:

- "How many users logged in this week?"
- "Show me the posts by this user from the past month."



SQL

- Can query, retrieve, and aggregate **millions** of records.
- Cloud-based data query and retrieval is not limited to your local computer system.
- Can organize data tables.
- Allows users to remotely interact with large data sets in production environments.

Excel

- Has a fixed upper **limit** of **1,048,576 rows** and **16,384 columns**.
- Is limited by your computer's available memory and system resources.
- Can analyze and visualize small amounts of data.

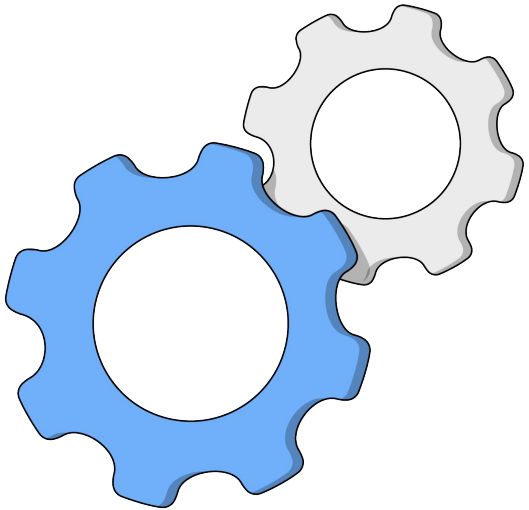
Benefits of SQL

- Simple to use.
- Industry standard for nearly all relational databases.
- Approachable syntax.
- Supportive community.
- Available in free and open versions.

Limitations of SQL

Although SQL has quite a few impressive superpowers, there are also limitations...

- SQL is **not** a data visualization tool.
- SQL can query, manipulate, organize, and analyze data.
- SQL is not a complete replacement for Excel given its lack of visualization functionality.
    - It's normally used **in conjunction with Excel, Tableau, and other tools**.



Navigating a SQL Database

PgAdmin PostgreSQL

- **PostgreSQL** is an open-source SQL standard compliant RDBMS (relational database management system).
- **PgAdmin** is a popular and feature-rich open-source administration and development platform for PostgreSQL.
- **PostgreSQL** is extensible and has strong online community support.

Guided Walk-Through:
Navigating the Superstore Data Set

We'll continue working with the Superstore data set in this unit. Let's go through the following steps to get started:

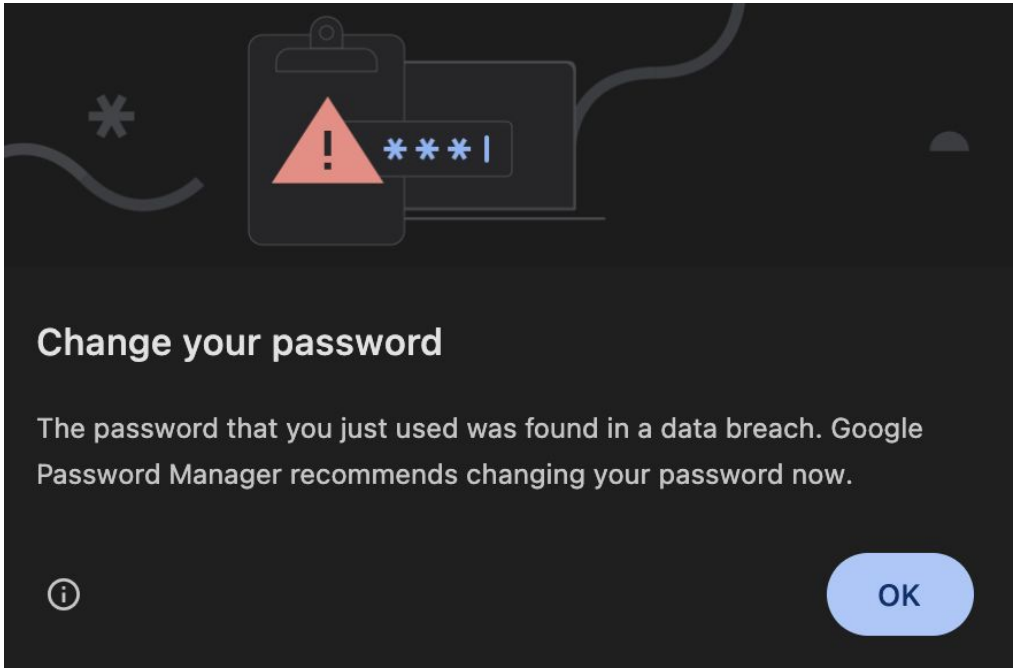
1. Connect to the SQL database that we'll be using for this unit: https://analyticsga-global.generalassemb.ly/

   Username: analytics_student@generalassemb.ly
   Password: analyticsga

2. Explore the functions of the client software (execute, stop, save, new query).
3. Look at the first and last 100 rows of the data from the tables using the menus.
4. Review how the column properties are defined using the menus.

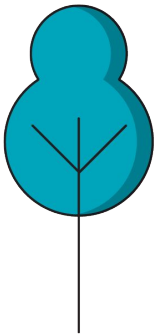
Guided Walk-Through:

Don’t worry if you get this message...



Guided Walk-Through:
Superstore Data Directory Tree

- Servers (*pgAdmin can be configured for multiple server connections.*)
  - Databases ➜ Superstore (our database for today)
    - Schemas ➜ Public (our collection of tables)
      - Tables (This would be the Superstore Table Names.)
        - Columns (for each table)



Telling the Story of One Row

To understand our data, we need to know what's in the stored data. One way to do this is by *telling the story of one row*:

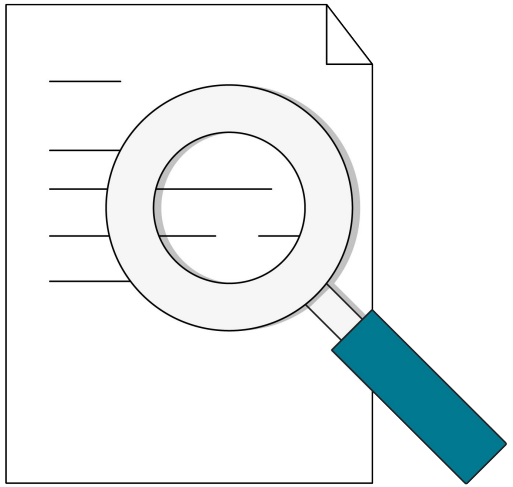
- **Read across one row of your data to really see and understand the values contained in every column.**

This will help you get to know what's in your data set or propel you to investigate further if you don't understand a particular column.

Discussion:

Previewing Your Data Set

Examining 100 rows of data may help us better understand if this is the data set we need to answer our research questions. This subset allows us to preview the data.



How does “previewing” the data connect to the data analytics workflow?

Is this useful? Why or why not?

Guided Walk-Through:
Viewing the Tables

To view the contents of a table, let's navigate to the tables in the Superstore database's public schema.

- There, you should see five tables: **Customers, Orders, Products, Regions, and Returns.**
- **To view the data, right click on the table, go to "View Data," and select the top 100 rows.**

Revealing contents of the tables and keeping the column names in view will assist you as you author queries.

Discussion:

Transactional vs. Reference Tables

Transactional tables are large and updated frequently, whereas **reference** tables are rarely modified.

Now we've seen the five tables — Customers, Orders, Products, Regions, and Returns.

Which of these do you think are transactional and which ones are reference?

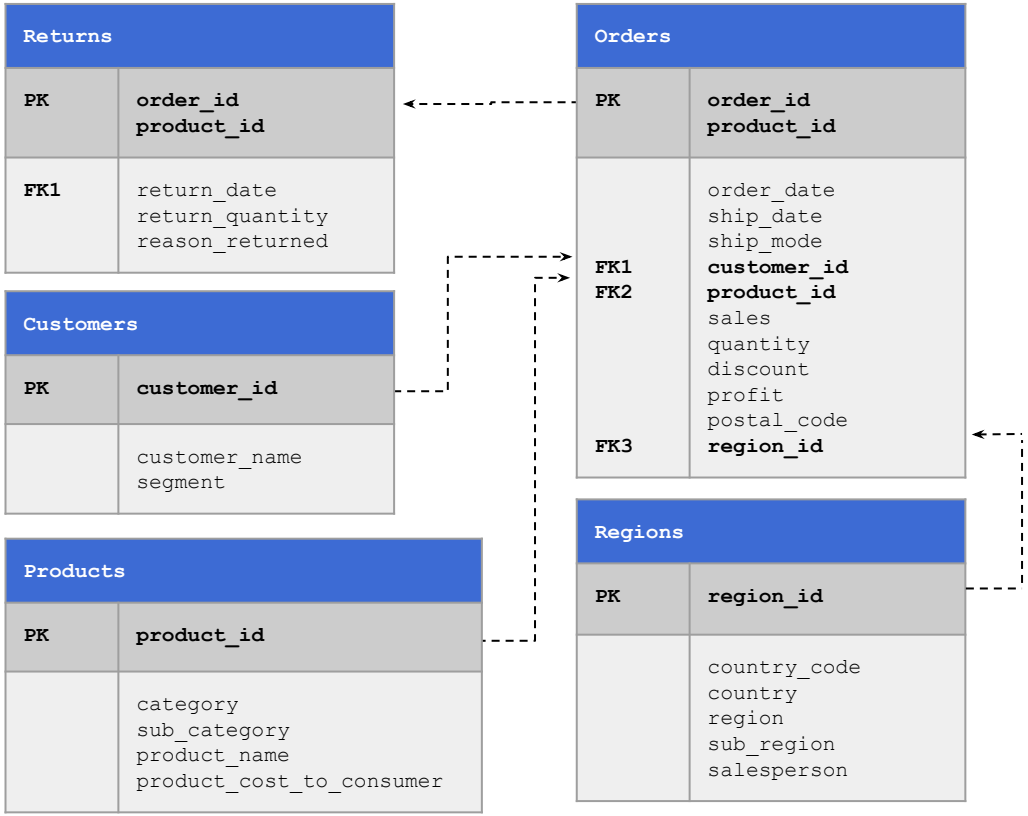
Partner Exercise:

Describing Stored Data

Take a closer look at the tables in the Superstore data set and explore the data they contain with your partner.

- Characterize each table as either **transactional** or **reference**.
- Take five minutes and make notes about the database:
  - Write a few sentences describing the data stored in each table.
  - Note the data types assigned to each column.
  - Which columns could serve as links between tables later in our data exploration?
- Discuss with your partner.

Entity Relationship Diagram (ERD)



An **entity** is a **component of data.**

An **entity relationship diagram** (ERD) shows the relationships of **entity sets** stored in a database.

ERDs help us illustrate the **logical structure** of databases.

PK: Primary key
FK: Foreign key, all unique identifiers

Adapted from: Guru99.com

Writing SQL Queries

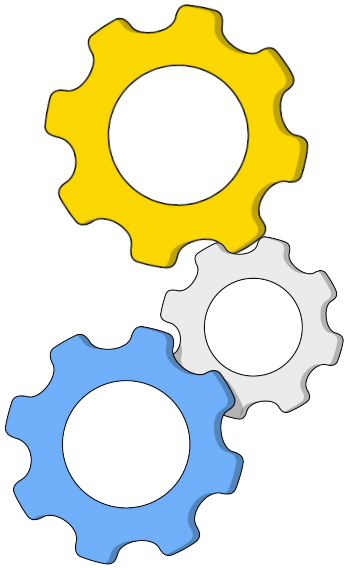
The Two Main Clauses of a SQL Query

SELECT

- Allows you to select certain ***columns*** from a table.
- Determines which ***columns*** of information are downloaded.

FROM

- Specifies the ***tables*** from which the query extracts data.



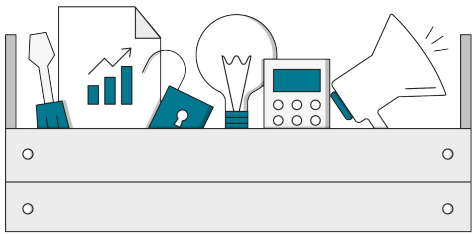
Did You Know...?

You ran a query already! When we selected the top 100 rows of a table using the “View Data” menu or “Query Tool” menu selections, this SQL statement ran in the background:

```
SELECT * FROM products LIMIT 100;
```

Take a closer look at the syntax above and try answering these questions:

- ❑ What does * mean?
- ❑ What does “**FROM products**” mean?
- ❑ What does the **LIMIT** do?



The SQL Query Order of Construction

SELECT picks the columns.

FROM points to the table.

WHERE puts filters on rows.

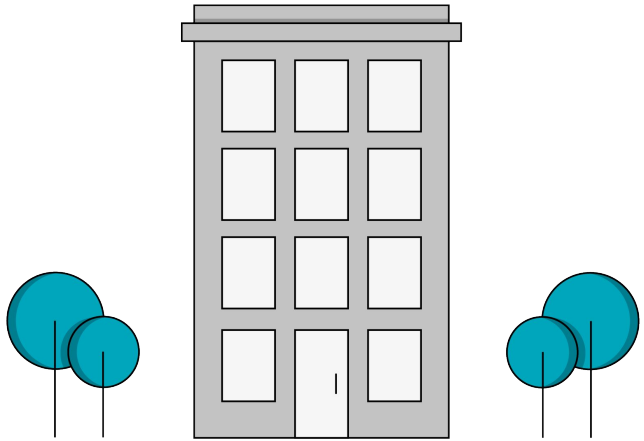
GROUP BY aggregates across values of a variable.

HAVING filters aggregated values ***after*** they have been grouped.

ORDER BY sorts the results.

LIMIT limits results to the first **n** rows.

The Query Building



Practicing Good SQL Grammar

Common Punctuation

- Signal the end of your SQL Query with a semicolon (;)
- Commas separate column names in an output list (,)
- Use single quotations around text/strings ('Nokia')

```
SELECT column1, column2
FROM table -- important note
WHERE column1 = 'Some Text';
```

Query Code Spacing

- SQL only requires a single white space to separate elements
- Carriage returns are often used to enhance readability

Notes Within Queries

- Always provide comments (source, revision, author, etc.)
- Comments are made after typing double dashes; or the pair /* */

Guided Walk-Through:
Modifying a Query

All queries can be run in the pgAdmin SQL window. For the remainder of this session, we'll practice modifying queries.

If we wanted to return the product ID and product name, which table would we use?

First, we can tell **SELECT** which columns or variables we want:

```
SELECT product_id, product_name
FROM ???;
```

Group Exercise:

Modifying a Query

Work with your group to return **all countries, regions, and salespeople**.

Which table would you use and which columns do you need to return?

Group Exercise:
Modifying a Query | Solution

Your query should look like this:

```sql
SELECT country, region, salesperson

FROM regions;
```


Introducing SELECT DISTINCT

- **SELECT DISTINCT** returns a unique combination of values for all columns selected.

- Every row, therefore, is a unique combination of values and results in a de-duplicated table.

```
SELECT DISTINCT column1, column2
FROM table
WHERE column1 = 'Some Text';
```


Guided Walk-Through:

SELECT DISTINCT

We can add **DISTINCT** to the query statement to eliminate duplicates of categories and subcategories:

```
SELECT DISTINCT category, sub_category
FROM products;
```

The SQL Query Order of Construction

SELECT picks the columns.

FROM points to the table.

WHERE puts filters on rows.

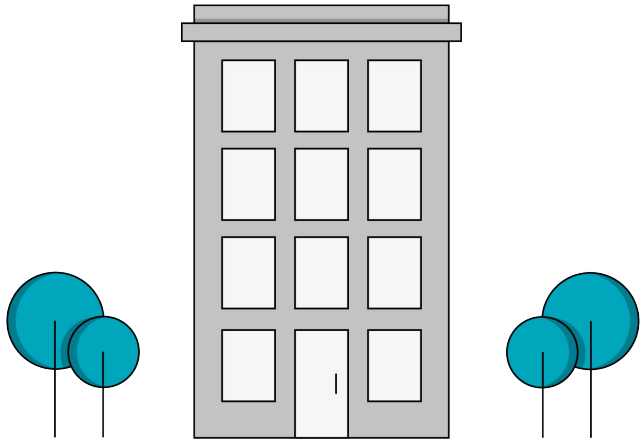
GROUP BY aggregates across values of a variable.

HAVING filters aggregated values ***after*** they have been grouped.

ORDER BY sorts the results.

LIMIT limits results to the first **n** rows.

The Query Building



Guided Walk-Through:

Sorting Results With **ORDER BY**

ORDER BY sorts results in ascending or descending order for the column specified. **ORDER BY** can also use a number that indicates the column by which you're sorting; this is the order of the columns listed in the **SELECT**.

The default sort order is ascending, but you can specify ascending (**ASC**) or descending (**DESC**) to determine the sort order. Because the default is ascending, using ASC is optional.

```
SELECT
product_id, product_name
FROM products
ORDER BY product_id ASC;
```

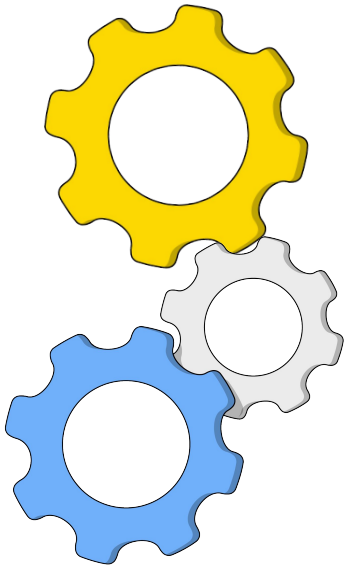
```
SELECT
product_id, product_name
FROM products
ORDER BY 1 ASC;
```


WHERE Conditions

Introducing the WHERE Clause

WHERE

- Allows you to select certain *rows* from a table based on a single condition or multiple conditions.



The SQL Query Order of Construction

SELECT picks the columns.

FROM points to the table.

WHERE puts filters on rows.

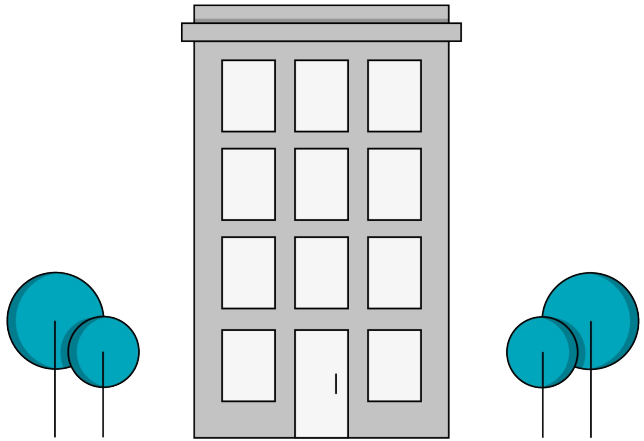
GROUP BY aggregates across values of a variable.

HAVING filters aggregated values ***after*** they have been grouped.

ORDER BY sorts the results.

LIMIT limits results to the first **n** rows.

The Query Building



Guided Walk-Through:
Filtering With **WHERE**

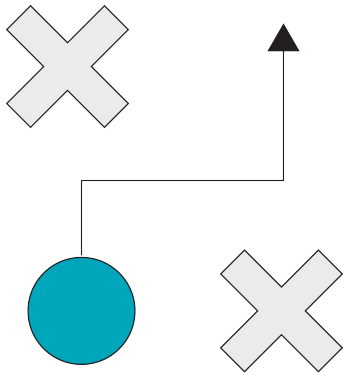
The **WHERE** clause filters rows by setting a criteria:

```
SELECT *
FROM products
WHERE sub_category = 'Furnishings';
```

A Little Filtering Goes a Long Way

SQL has a variety of commands and operators to ask filtering questions. To make the most out of filtering with **WHERE**:

- Carefully consider the data collection process for **potential irregularities** in your data.
- Select your command tools to make sure no data is left behind.
- **Validate your results** by including the testing columns in your output.
- Consider how to **reconcile your results** with the rest of the table to ensure accurate insights.



Guided Walk-Through:

Validating Data With **COUNT**

COUNT is a basic aggregation function that counts the number of rows returned.

Here are two popular use cases:

- **COUNT(*)** - Counts all rows returned by the query.
- **COUNT(field)** - Counts all rows where the field is not **NULL**.

```
SELECT COUNT(*)
FROM orders
WHERE sales > 100;
```

Counts all rows of orders over $100.

Guided Walk-Through:
Basic Operators for WHERE

`<>, !=` Not equal to.

`>, >=` Greater than; greater than or equal to.

`<, <=` Less than; less than or equal to.

Which countries are **not** in the “EMEA” region?

```
SELECT DISTINCT country
FROM regions
WHERE region <> 'EMEA';
```

Guided Walk-Through:
Basic Operators for WHERE (Cont.)

Which orders have more than $90 in profit?

```sql
SELECT *
FROM orders
WHERE profit > 90;
```

Guided Walk-Through:
Basic Operators for WHERE (Cont.)

Which returns include more than one quantity of an item?

```
SELECT *
FROM returns
WHERE return_quantity > 1;
```


Guided Walk-Through:
Filtering With **AND**, **OR**, and **()**

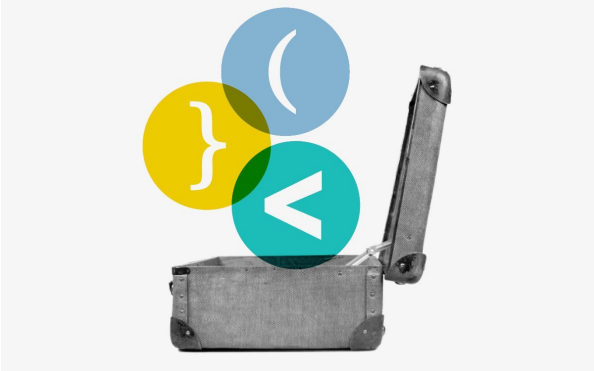
Some additional query methods to try in the WHERE clause:

- **AND**: Returns if both conditions are true.
- **OR**: Returns if either condition is true (FALSE if neither is true).
- **( )**: Parentheses group conditions to ensure your desired logic is followed.

```
SELECT *
FROM regions
WHERE (region = 'Americas'
    OR region = 'APAC')
    AND salesperson <> 'Anna Andreadi';
```

Comparison Operators

IN ( )	Found in list of items.
BETWEEN	Within the range of, including boundaries.
NOT	Negates a condition.
LIKE, ILIKE	Contains item. ILIKE disregards case.
%	Wildcard, none to many characters.
_	Wildcard, single character.



Guided Walk-Through:
Filtering With **IN**

`IN ( )` allows you to specify multiple values in WHERE, while `NOT IN ( )` allows you to negate the list of values.

Which products are **either** furnishings or technology (based on category name)?

```sql
SELECT *
FROM products
WHERE category IN ('Furniture', 'Technology');
```

Which products are **neither** furnishings nor technology?

```sql
SELECT *
FROM products
WHERE category NOT IN ('Furniture', 'Technology');
```

Guided Walk-Through:
Filtering With **BETWEEN**

BETWEEN ( ) allows you to select values within a given range.

1. Which products have a cost to consumers between $25 and $100?

```
SELECT *
FROM products
WHERE product_cost_to_consumer BETWEEN 25 AND 100;
```

2. Which orders have sales in 2019?

```
SELECT *
FROM orders
WHERE order_date BETWEEN '2019-01-01' AND '2019-12-31';
```


Guided Walk-Through:
Filtering **LIKE** and **ILIKE**

`LIKE` is used for pattern matching in SQL, while `NOT LIKE` negates the match. `ILIKE` matches with case insensitivity.

1. Which products have ‘Calculator’ in the product name?

```sql
SELECT *
FROM products
WHERE product_name LIKE '%Calculator%';
```

2. Which products have ‘Printer’ in the product name?

```sql
SELECT *
FROM products
WHERE product_name ILIKE '%PRINTER%';
```

Guided Walk-Through:

Filtering With Wildcards

PostgreSQL provides two wildcard characters to work with **LIKE**:

- **percent (%)** for matching any sequence of characters.
- **underscore (_)** for matching any single character.

1. Which products have ‘Clock’ in the product name?

```
SELECT *
FROM products
WHERE product_name LIKE '%Clock%';
```

2. Which customers have names that start with “A” and third letter of “r”?

```
SELECT *
FROM customers
WHERE customer_name LIKE 'A_r%';
```


Quick Re-cap

So far we have covered a number of SQL statements:

```
SELECT                                  SELECT DISTINCT
FROM                                    COUNT
WHERE (<>, >, >=, AND, OR, IN, BETWEEN, NOT, LIKE, ILIKE, NOT LIKE, %, _)
ORDER BY
LIMIT
```


Solo Exercise:

Now You Try

To help a sales director better understand the data, write queries that answer the questions below:

1. Who are the salespeople in the United States? *Get a list of **only** the salespeople, with no duplicates*
2. What were the top five sales in 2019? *Get the Order ID, Product ID and Revenue of each sale*
3. What was the highest discount we gave in 2019? *Get just the discount amount, and nothing else*

Solo Exercise:

Now You Try - Solutions (1 of 3)

Who are the salespeople in the United States?

Resulting SQL query:

```sql
SELECT DISTINCT salesperson
FROM regions
WHERE country ILIKE '%United States%';
```

Solo Exercise:

Now You Try - Solutions (2 of 3)

What were the top five sales in 2019?

Resulting SQL query:

```sql
SELECT DISTINCT order_id, product_id, sales
FROM orders
WHERE order_date BETWEEN '2019-01-01' AND '2019-12-31'
ORDER BY sales DESC
LIMIT 5;
```

Solo Exercise:

Now You Try - Solutions (3 of 3)

What was the highest discount we gave in 2019?

Resulting SQL query:

```sql
SELECT DISTINCT discount
FROM orders
WHERE order_date BETWEEN '2019-01-01' AND '2019-12-31'
ORDER BY discount DESC
LIMIT 1;
```

Organizing Data with SQL

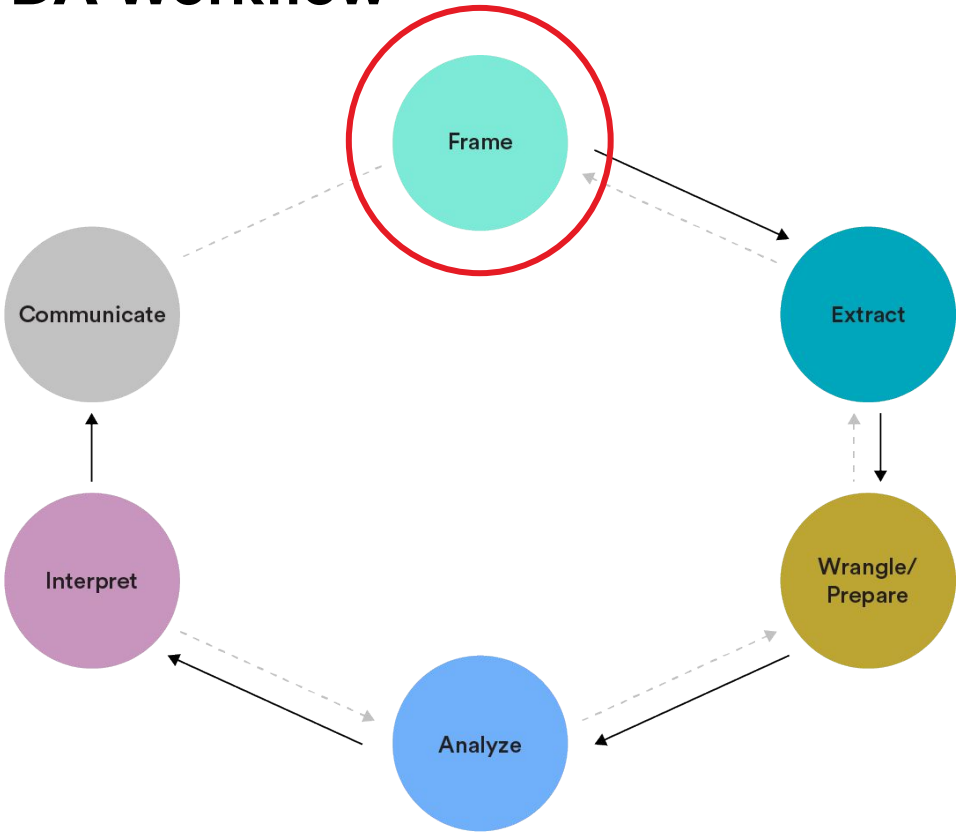
From Stakeholder Questions to Efficient Queries

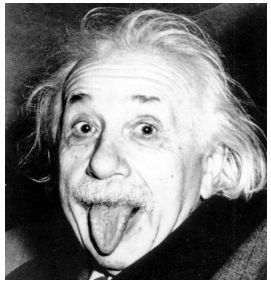
We are moving in the DA Workflow

Frame

What makes a **good** and **robust** data question?

Framing our hypothesis is a crucial first step in any data analysis, so it's important that we get this right.





If I were given one hour to save the planet, I would spend 59 minutes defining the problem and one minute resolving it.

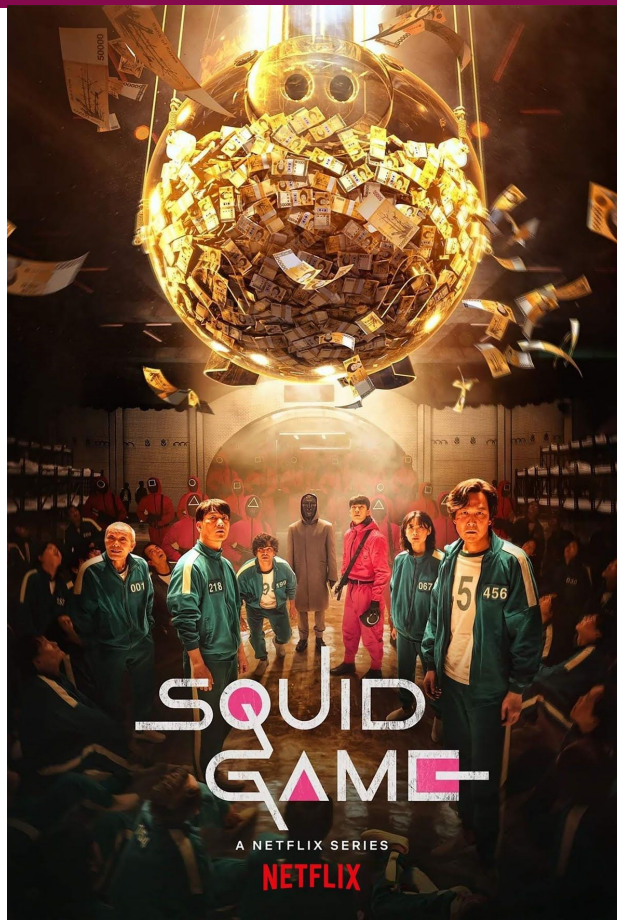
Albert Einstein

Discussion:
Scenario

2 minutes

You’re meeting up with friends one evening and start discussing what you’ve been watching on TV. One friend starts talking about “Squid Game.” You’ve heard a lot about it, and want to know whether it’s as good as the hype.

What might you ask them so you can find out?



Business Questions vs. Analytics Questions

When people ask us to analyze data, they often ask us **qualitative** business questions:

“How good is X?”

As data analysts, we need to make questions **quantitative** in order to be able to answer them.

This involves either clarifying exactly what someone wants, or making assumptions about what they really want and then answering that question. Usually, it’s a combination of the two.

Side note: *When people ask us to analyze data, they are giving us* ***business requirements****. These can be formal, such as by using a specific template, or they can be informal (chat, DM, email).*

What's a Good Data Analysis Question?

Specific Asking a single question

Measurable Uses metrics to quantify the result

Achievable Can be answered using the available data

Characteristics of a Good Question

A **good question** is a statement which:

- Might be wrong.
- Could be rejected with a certain confidence.

This is what a **HYPOTHESIS** is: A proposed explanation for something that is then tested.

e.g. Hyperactivity is unrelated to eating sugar.

Source: giphy.com

What's a Good Data Analysis Question?

Instead of asking:

How good is healthcare in the UK?

You could ask:

Are hospitals meeting government targets on waiting times for cancer treatment, mental health treatment, and emergency services?

Partner Exercise:

Well Defined Questions...?

With your partner, look at these questions and discuss whether you think they are well written or not. How can you improve them?

1. Who is the best soccer player in the world?
2. What effect does social media have on people's minds?
3. Who is the best salesperson in the company?

Common Questions Asked in Data Analysis

From a data analysis perspective:

1. Does X predict Y?
2. Are there any distinct groups in our data?
3. Is one of our observations “weird”?

Common Questions Asked in Data Analysis

From a business perspective:

1. What is the likelihood that a customer will buy this product?
2. How much demand will there be for my service tomorrow?
3. What groups of products are customers purchasing together?

Group Exercise:
Framing the Question

You work for a global public health agency and it has asked you to produce a piece of data journalism on **“Where was the coronavirus situation the worst?”**

In groups, discuss better and more specific versions of this question, and be ready to share with the rest of the group. Think about:

- How you’d quantify “worst” and the pros and cons of your choice.
- What level of geography you’d looking at, and the limitations of your choice.
- The timeframe you’re interested in.
- Do you think the data needed to answer your question is available?

Be ready to share your thoughts when you’re done.

“

A problem well stated is half solved.

Charles Kettering

Solo Exercise:

Now You Try

To help a sales director better understand the data, write queries that answer the questions below:

1. Who are the salespeople in the United States? *Get a list of **only** the salespeople, with no duplicates*

2. What were the top five sales in 2019? *Get the Order ID, Product ID and Revenue of each sale*

3. What was the highest discount we gave in 2019? *Get just the discount amount, and nothing else*

Discussion:

Are these good questions?

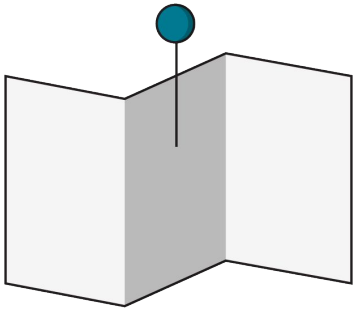
1. Who are the salespeople in the United States?
2. What were the top five sales in 2019?
3. What was the highest discount we gave in 2019?

Partner Exercise:

From Stakeholder Questions to Queries

Together with a partner, write SQL queries for the following questions.

1. How many countries are in the “Americas” region?
2. Which technology products are sold to consumers for more than $1,000?
3. How many product items were returned in 2019 for unknown reasons?



Partner Exercise:

Writing and Executing SQL Queries - Stretch

Here are the questions that we want to ask of our data. With your partner, practice writing and executing queries that'll help us answer these questions.

1. How many countries are in the APAC sales region?
2. Which furniture products are sold for more than $50 and less than $500?
3. Which products are recycled material and are sold for less than $25?
4. How many customers are in our consumer segment with a first name that starts with the letter "A"?
5. What's the total profit (dollars) for orders made in 2019 with 2–3 product items in the order?

Validating Your Output

How do you know your results are correct?

- Check that the table and column names are correct.
- Check that your quotation marks are closed.
- Check operators such as >, <, =, <>, != etc.
- Review error messages when your query does not run.
- Apply business context or industry knowledge (of similar data) to check the order of magnitude.

Partner Exercise:

How Did It Go? | **Solutions**

1. How many countries are in the “Americas” region?

```
SELECT COUNT(DISTINCT country) FROM regions
WHERE region = 'Americas';
```
Answer: 26

2. Which technology products are sold to consumers for more than $1,000?

```
SELECT * FROM products WHERE category = 'Technology'
AND product_cost_to_consumer > 1000;
```
Answer: 9 rows

3. How many product items were returned in 2019 for unknown reasons?

```
SELECT COUNT(order_id) FROM returns
WHERE return_date BETWEEN '2019-01-01' AND '2019-12-31'
AND reason_returned = 'Not Given';
```
Answer: 10,076

Partner Exercise:

Writing and Executing SQL Queries - Stretch | **Solutions**

1	`SELECT COUNT(country) FROM regions WHERE region = 'APAC';` 29 countries
2	`SELECT * FROM products WHERE category = 'Furniture' AND product_cost_to_consumer BETWEEN 50 AND 500;` 1461 rows
3	`SELECT * FROM products WHERE product_name LIKE '%Recycled%' AND product_cost_to_consumer < 25;` 309 rows
4	`SELECT COUNT(customer_id) FROM customers WHERE segment = 'Consumer' AND customer_name LIKE 'A%';` 66 customers
5	`SELECT SUM(profit) FROM orders WHERE DATE_PART('year', order_date) = 2019 AND quantity BETWEEN 2 and 3;` $181,944.85

Solo Exercise:

Optional Practice

To help Superstore better understand their data, write queries that answer the questions below:

1. Which products are made by Xerox?
2. How many countries are in our Western Europe sub-region?
3. How many customers in our consumer segment have names that start with the letter “S”?
4. What is the total number of items (not orders) returned with some reason given?
5. How many orders used Standard Shipping and a discount code?

When you’re done, check your responses using the answers provided in the next slide.

Solo Exercise:

Optional Practice | Solutions

1. `SELECT * FROM products WHERE product_name LIKE '%Xerox%';`
Returns 260 rows

2. `SELECT COUNT(country) FROM regions WHERE sub_region = 'Western Europe';`
Returns count of 6

3. `SELECT COUNT(customer_name) FROM customers WHERE segment = 'Consumer' AND customer_name LIKE 'S%';`
Returns count of 70

4. `SELECT SUM(return_quantity) FROM returns WHERE reason_returned != 'Not Given';`
Returns count of 31,781

5. `SELECT * FROM orders WHERE ship_mode = 'Standard Class' AND discount > 0;`
Returns 397,723 rows

Handling Multiple Conditions With CASE

What If...

Let's take a look at this query from earlier:

```
SELECT *
FROM products
LIMIT 100;
```

What if, from the list of 100 products, we want to **break our product prices into groups (free, cheap, affordable, expensive)**?

To get the result, your query must include ***multiple*** conditions: free, cheap, affordable, and expensive.

What Are CASE Statements?

CASE statements group data into *categories* or *classifications*. They **go through multiple conditions and return a value when the *first* condition is met**.

- When a condition is true, CASE will stop reading and return the result.
- If no conditions are true, it will return the value in the ELSE clause.

CASE Syntax

```
SELECT column,
    CASE
        WHEN condition THEN result
        WHEN condition THEN result
        ELSE condition
    END AS output_name
FROM table;
```

CASE syntax in plain words:

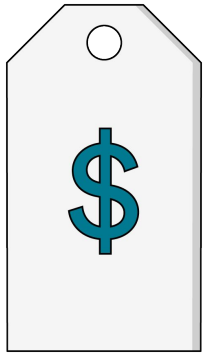
- SELECT takes the column on which you want to run **CASE**.
- WHEN **<condition is true>**.
- THEN **<what to return as a value for that row>**.
- ELSE (optional condition).
- **END AS** <header title for the new column you just made>.
- FROM <table>.

Guided Walk-Through:

Handling Multiple Conditions With CASE

Let’s try classifying the discounts in the Superstore data set into these groups:

- Free (100%)
- High (25–99%)
- Low (1–25%)
- None (0%)



Using CASE statements, let’s write this out.

Guided Walk-Through:

Handling Multiple Conditions With CASE (Cont.)

Below is our query with a CASE statement. **Remember**: SQL will *only* return values that meet these conditions, but we can also add an “other” category using ELSE.

```
SELECT DISTINCT
  discount,
     CASE
          WHEN discount = 1 THEN 'Free'
          WHEN discount = 0 THEN 'None'
          WHEN discount BETWEEN .25 AND 1 THEN 'High'
          WHEN discount < .25 THEN 'Low'
     END AS discount_level
FROM orders
ORDER BY discount;
```

Group Exercise:

Handling Multiple Conditions With CASE

Work with your group to create a CASE statement that groups **orders by whether or not they had a positive profit.**

Group Exercise:

How Did It Go? | Solution

How did it go? Your query should look like the following:

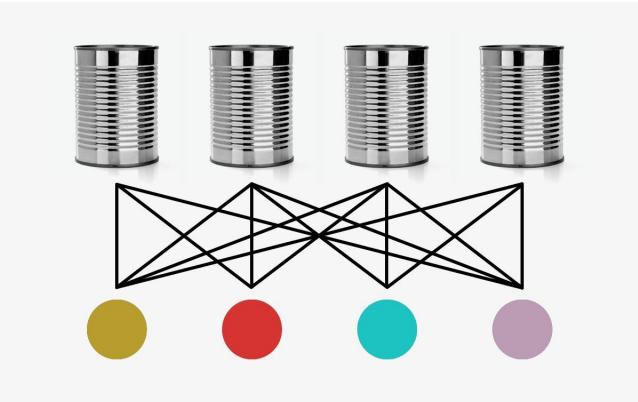
```sql
SELECT profit,
    CASE WHEN profit > 0 THEN 'Positive'
         WHEN profit = 0 THEN 'Breakeven'
         ELSE 'Negative'
    END AS profit_level
FROM orders
LIMIT 100;
```

Aggregate Functions in SQL

Aggregate Functions

In SQL, aggregate functions help summarize large quantities of data and...

- Produce a single value from a defined group.
- Operate on sets of rows and return results based on **groups of data**.



The most commonly used aggregate functions are **MIN**, **MAX**, **SUM**, **AVG**, and **COUNT**.

Guided Walk-Through:
Adding Aggregates to SELECT

Aggregate functions fit into the **SELECT** statement just like unaggregated columns:

```
SELECT SUM(col1)
FROM table;
```

Group Exercise:

Aggregate Functions Matchmaking

Refer to the scenarios on the left and connect them to the aggregate functions on the right that will return the results you need.

1. The number of orders placed on March 31, 2019.
2. The highest profit for orders shipped via Standard Class.
3. The typical quantity of product id TEC-BRO-10000381 sold.
4. The number of customers placing orders with zip code 94591.
5. The lowest discount given for orders placed using zip code 94591.

COUNT: Counts how many values are in a particular column.

COUNT DISTINCT: Counts how many unique values are in a particular column.

SUM: Adds together all the values in a particular column.

AVG: Calculates the average of a group of selected values.

MIN/MAX: Return the lowest and highest values in a particular column, respectively.

Partner Exercise:

Let's Try It!

Now that you have a good sense of what each aggregate function does, select 2–3 prompts from the previous slide and try writing out the query for each.

At a minimum, your queries should include **SELECT** and **FROM** and an aggregate function. For example:

```
SELECT SUM(sales)
FROM orders;
```

Ready, set, go!

Partner Exercise:

Solutions 1 - 3

```sql
-- The number of orders placed on March 31, 2019.
SELECT COUNT(*)
FROM orders
WHERE order_date = '2019-03-31';
```
Answer: 922

```sql
-- The highest profit for orders shipped via Standard Class.
SELECT MAX(profit)
FROM orders
WHERE ship_mode = 'Standard Class';
```
Answer: $8,399.98

```sql
-- The typical quantity of product id sold.
SELECT AVG(quantity)
FROM orders
WHERE product_id = 'TEC-BRO-10000381';
```
Answer: 1.41

Partner Exercise:

Solutions 4 - 5

```sql
-- The number of customers placing orders zip code 94591.
SELECT COUNT(DISTINCT customer_id)
FROM orders
WHERE postal_code = '94591';
```
Answer: 52

```sql
-- The lowest discount given for orders placed using zip code 94591.
SELECT MIN(discount)
FROM orders
WHERE postal_code = '94591';
```
Answer: 0.02 (2%)

Guided Walk-Through:

Using CASE Within a SELECT Statement

Say you need to **calculate the percentage of orders that were considered a high discount item.** Here is the query that would allow us to categorize the orders:

```
SELECT CASE  WHEN discount > .25  THEN 1
        ELSE 0  END as discount_level
FROM orders;
```


Guided Walk-Through:
Using CASE Within a SELECT Statement (Cont.)

Say you need to **calculate the percentage of orders that were considered a high discount item.** Then, we can average the output to get the percentage:

```
SELECT AVG(CASE  WHEN discount > .25  THEN 1
        ELSE 0 END) as discount_level
FROM orders;
```

GROUP BY and HAVING

Clauses for Aggregate Functions

Aggregate functions are also used in these clauses:

- **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).

Where They Live in a Query

SELECT picks the columns.

FROM points to the table.

WHERE puts filters on rows.

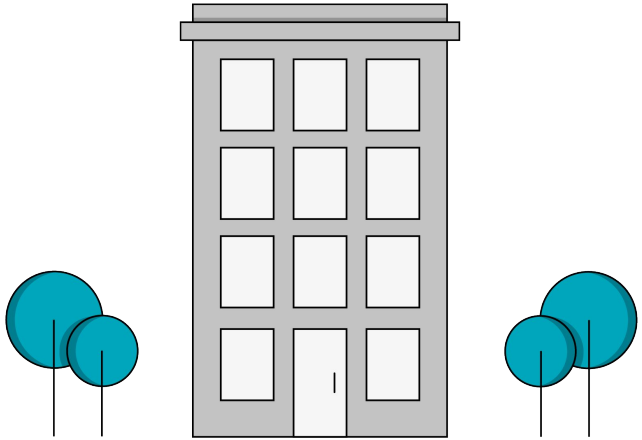
GROUP BY aggregates multiple rows, based on one or more aggregate functions (MIN, AVG, etc.).

HAVING filters aggregated values ***after*** they have been grouped.

ORDER BY sorts the results.

LIMIT limits results to the first **n** rows.

The Query Building



Guided Walk-Through:
What Will the Code Look Like?

Let's look at an example using a table called "People" with the columns "Gender" and "Height." From this table, we want to:

- Find the average height of people by gender.

What will our query look like?

```
SELECT gender, AVG(height) AS avg_height
FROM people
GROUP BY gender;
```

People	
Gender	**Height (Inches)**
Female	67
Male	67
Non-binary	62
Female	63
Male	68
Female	36
Non-binary	24

Guided Walk-Through:
What Will the Code Look Like? (Cont.)

Now, we want to limit our analysis to only those people taller than three feet:

- Find the average height of people taller than three feet by gender.

What will our query look like?

```
SELECT gender, AVG(height) AS avg_height
FROM people
WHERE height > 36
GROUP BY gender;
```

People	
Gender	**Height (Inches)**
Female	67
Male	67
Non-binary	62
Female	63
Male	68
~~Female~~	~~36~~
~~Non-binary~~	~~24~~

Guided Walk-Through:

What Will the Code Look Like? (Cont.)

Finally, we want to only return genders that have an average height of more than 5.5 feet (66 inches).

- Determine which of those people have an average height greater than 5.5 feet tall, sorted by gender.

What will our query look like?

```
SELECT gender, AVG(height) AS avg_height
FROM people
WHERE height > 36
GROUP BY gender
HAVING AVG(height)>65;
```

People	
Gender	**Avg Height (Inches)**
~~Female~~	~~65~~
Male	**67.5**
~~Non-binary~~	~~62~~

Guided Walk-Through:

What Will The Code Look Like? | **Solution**

```
SELECT gender, AVG(height)
FROM people
WHERE height >36
GROUP BY gender
HAVING AVG(height) >65
ORDER BY gender;
```

What each part of the query does:

- SELECT picks the columns.
- FROM determines and filters rows.
- WHERE adds more filters on those rows.
- **GROUP BY combines those rows into groups.**
- **HAVING filters groups.**
- ORDER BY arranges the remaining rows/groups.

Partner Exercise:

Let's Build a Few More Together!

With your partner, build the following queries with GROUP BY and HAVING:

```sql
SELECT segment,
    COUNT(*) AS num_customers
FROM customers
GROUP BY segment;
```

```sql
SELECT segment,
    COUNT(*) AS num_customers
FROM customers
GROUP BY segment
HAVING COUNT(*) > 300;
```

- What is each query showing you?
- How many results do you get with the **GROUP BY** statement?
- How many results do you get with the **HAVING** statement included?

Guided Walk-Through:
Aggregating Data With GROUP BY and HAVING

Superstore wants an order discount analysis to identify average order qty and amount by discount level for orders that have a discount higher than 15% and where the discount level has more than $500 in sales.

Let’s break this down and build up a query to get the data.

Guided Walk-Through:
Aggregating Data With GROUP BY and HAVING

Firstly, let's look at the main part of the request: Superstore wants an **order discount analysis to identify average order qty and amount by discount level for orders** ~~that have a discount higher than 15% and where the discount level has more than $500 in sales~~.

To write our query, we'll use **GROUP BY** to aggregate qty and sales.

```
SELECT discount, AVG(quantity), AVG(sales)
FROM orders
GROUP BY discount;
```

Guided Walk-Through:
Aggregating Data With GROUP BY and HAVING

Secondly, let's look at the second part of the request: Superstore wants an order discount analysis to identify average order qty and amount by discount level for orders **that have a discount higher than 15%** ~~and where the discount level has more than $500 in sales~~.

For this part, we'll add a **WHERE** clause to filter discount levels greater than 15%.

```
SELECT discount, AVG(quantity), AVG(sales)
FROM orders
WHERE discount > 0.15
GROUP BY discount;
```

Guided Walk-Through:
Aggregating Data With GROUP BY and HAVING

Thirdly, let's look at the second part of the request: Superstore wants an order discount analysis to identify average order qty and amount by discount level for orders that have a discount higher than 15% **and where the discount level has more than $500 in sales**.

Let's add a **HAVING** clause to filter discount levels above a $500 sales threshold

```
SELECT discount, AVG(quantity), AVG(sales)
FROM orders
WHERE discount > 0.15
GROUP BY discount
HAVING AVG(sales) > 500;
```

Guided Walk-Through:
Aggregating Data With GROUP BY and HAVING

Finally, let's format the query so the output is easy to read:

- Give the aggregate columns aliases
- Round the average quantity
- Convert the data type for the average sales output to be money
- Order the data by average sales

```
SELECT discount, ROUND(AVG(quantity), 2) AS qty, AVG(sales)::money AS
sales
FROM orders
WHERE discount > 0.15
GROUP BY discount
HAVING AVG(sales) > 500
ORDER BY 3 DESC;
```

Solo Exercise:

Over to You

Use the starter code below to find **the number of products by category** in the Superstore data set.

```
SELECT category, (aggregate of rows)
FROM products
GROUP BY category;
```

Solo Exercise:

Over to You | An Extra Challenge

Once you have filled in the parts of the starter code, try the following:

- Include only products with “computer” or “color” (case-insensitive) in the name.
- Further refine to those that *have* an aggregate 100 or more products.
- Alias your aggregate column to **count_of_products**
- Sort the results by the **count_of_products** column.

Ask for help if you have questions or need a hint.

Solo Exercise:

How Did It Go? | **Solution**

Here is what your end query might look like:

```sql
SELECT category, COUNT(*) AS count_of_products
FROM products
WHERE
  product_name ILIKE '%computer%'
  OR product_name ILIKE '%color%'
GROUP BY 1
HAVING COUNT(*) > 100
ORDER BY 2 DESC;
```

Guided Walk-Through:
Using CASE and GROUP BY

Now, we're able to look at the total number of sales made in each of our discount categories from earlier:

```
SELECT CASE
    WHEN discount = 1 THEN 'Free'
    WHEN discount BETWEEN .25 AND 1 THEN 'High'
    WHEN discount = 0 THEN 'None'
    WHEN discount < .25 THEN 'Low'
  END AS discount_level, COUNT(DISTINCT order_id) AS num_sales
FROM orders
GROUP BY 1;
```

Solo Exercise:

Over to You

Use the starter code below to find **the number of orders by profit category** in the Superstore data set.

```
SELECT CASE
        WHEN profit > 0 THEN 'Positive'
        WHEN profit = 0 THEN 'Breakeven'
        ELSE 'Negative'
        END AS profit_level,
        (aggregate of rows)
FROM orders;
```


Solo Exercise:

How Did It Go? | Solution

```sql
SELECT CASE
        WHEN profit > 0 THEN 'Positive'
        WHEN profit = 0 THEN 'Breakeven'
        ELSE 'Negative'
        END AS profit_level,
        COUNT(DISTINCT order_id) AS num_sales
FROM orders
GROUP BY profit_level;
```

Solo Exercise:

Over to You

We want to dig into **how many of our sales have been for more than 1 *of the same* item in a single sale** (i.e. quantity 2+).

First, create a **CASE** statement that will group orders by their quantity, then count up the number of orders associated with your groupings.

Solo Exercise:

How Did It Go? | Solution

Here is what your end query might look like:

```
SELECT CASE
    WHEN quantity>1 THEN 'Multiple'
    ELSE 'Single' END AS quantity,
    COUNT(DISTINCT order_id) AS num_sales
FROM orders
GROUP BY 1;
```

Solo Exercise:

Optional Practice

Use your tables to answer the following questions:

1. Do our customers prefer a certain type of shipping class? Find the number of orders per ship mode.
2. How many unique salespeople do we have employed in each region? How does that compare to the number of unique countries in each region?
3. Find the most popular reason for returns.
4. **Bonus:** Create a query that groups the total number of products available by vendor. The vendors we want to focus on are 3M, Apple, Avery, Cisco, Epson, Hewlett-Packard (HP, Hewlett Packard), Logitech, Panasonic, Samsung, and Xerox.

Solo Exercise:

Optional Practice | Solution 1

```sql
SELECT
    ship_mode,
    COUNT(DISTINCT order_id) as order_count
FROM orders
GROUP BY 1
ORDER BY 2 DESC;
```

Returns 4 Rows; *Standard Class* has the highest order count

Solo Exercise:

Optional Practice | Solution 2

```sql
SELECT
    region,
    COUNT(DISTINCT salesperson) as sales_person_count
FROM regions
GROUP BY 1
ORDER BY 2 DESC;
```

***Americas* has the highest sales person count**

Solo Exercise:

Optional Practice | Solution 2 (Cont.)

```sql
SELECT
    region,
    COUNT(DISTINCT country) as country_count
FROM regions
GROUP BY 1
ORDER BY 2 DESC;
```

***EMEA* has the highest country count**

Solo Exercise:

Optional Practice | Solution 3

```sql
SELECT
    reason_returned,
    COUNT(DISTINCT order_id) as return_reason_count
FROM returns
GROUP BY 1
ORDER BY 2 DESC;
```

Not Given **has the highest count; Wrong Items has the highest count for orders where a reason was given.**

Solo Exercise:

Optional Practice | Solution 4 **Bonus**

```
SELECT CASE WHEN product_name ILIKE '%3M%' THEN '3M'
            WHEN product_name ILIKE '%Apple%' THEN 'Apple'
            WHEN product_name ILIKE '%Avery%' THEN 'Avery'
            WHEN product_name ILIKE '%Epson%' THEN 'Epson'
            WHEN product_name ILIKE '%HP%' OR product_name ILIKE '%Hewlett%' THEN 'Hewlett-Packard'
            WHEN product_name ILIKE '%Logitech%' THEN 'Logitech'
            WHEN product_name ILIKE '%Panasonic%' THEN 'Panasonic'
            WHEN product_name ILIKE '%Samsung%' THEN 'Samsung'
            WHEN product_name ILIKE '%Xerox%' THEN 'Xerox'
      END as vendor,  COUNT(*) as product_count
FROM products GROUP BY 1 ORDER BY 1;
```

Re-cap

What we've covered in this section:

`CASE (WHEN THEN ELSE END)`

AGGREGATE FUNCTIONS:

- `COUNT`
- `COUNT DISTINCT`
- `SUM`
- `AVG`
- `MIN`
- `MAX`
- % of orders (use CASE and 0 / 1)

- `GROUP BY`
- `HAVING`

Organizing Data with SQL

Combining Data with JOINs and UNIONs

Celebrating Table Togetherness

One 2019 study found that most companies with 1,000 employees or more are pulling from 400+ data sources for business intelligence.

In fact, more than 20% of the organizations reported drawing from a whopping 1,000 or more data sources.

So, let's get comfortable bringing that data together!

Image source: The A Team Meme Generator

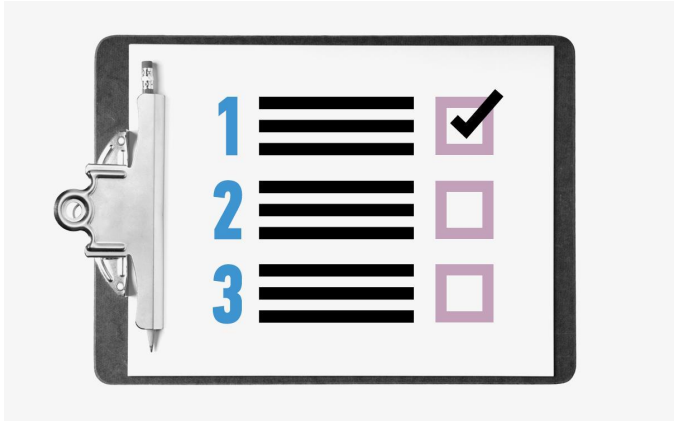
Discussion:

What Could Possibly Go Wrong?

You've handled a data set or two before.

Let's make a list that addresses the following:

1. **What could go wrong when combining two or more data sets?**
2. **What might you want to have control over?**



JOINs and UNIONs

In SQL, there are two primary methods for bringing data together:

A **JOIN** combines **columns** from tables using common unique identifiers (keys).

A **UNION** combines **rows** of *similar* data.

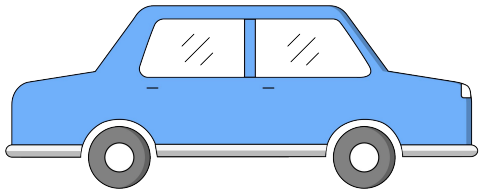
JOINs

A **JOIN** combines columns from multiple tables using a common unique identifier or “key.”

drivers		
id	name	vehicle_id
1	Janet	3
2	Emily	4
3	Yoko	4
4	Ali	5

vehicles	
id	vehicle_name
1	Explorer
2	Civic
3	Corolla
4	Impala

id	name	vehicle_id	vehicle_name
1	Janet	3	Corolla
2	Emily	4	Impala
3	Yoko	4	Impala



UNIONs

A **UNION** combines rows from multiple tables with similar data to create a new set. Using “UNION” removes duplicates when combining the two tables.

carpoolers		
id	name	vehicle_id
1	Janet	3
2	Emily	4
3	Yoko	4

monthly_parkers		
id	name	vehicle_id
2	Emily	4
4	Ali	5
5	Ray	1

id	name	vehicle_id
1	Janet	3
2	Emily	4
3	Yoko	4
4	Ali	5
5	Ray	1

Where They Live in a Query

SELECT picks the columns.

FROM points to the table.

WHERE puts filters on rows.

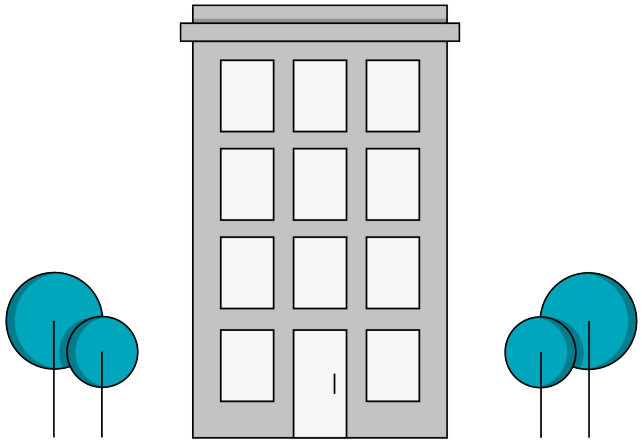
GROUP BY aggregates across values of a variable.

HAVING filters aggregated values ***after*** they have been grouped.

ORDER BY sorts the results.

LIMIT limits results to the first **n** rows.

The Query Building



JOINs

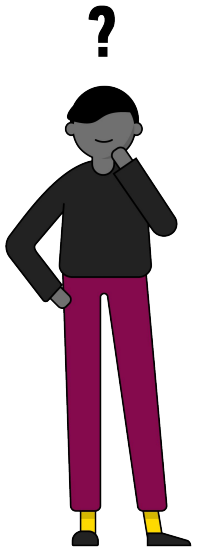
Partner Exercise:

JOINs vs. UNIONs

With your partner, Google "**database normalization**" and discuss:

1. The concept of normalization.
2. Why JOINs are needed for normalized data stores for a transactional database.

Be prepared to share your answers with the class.



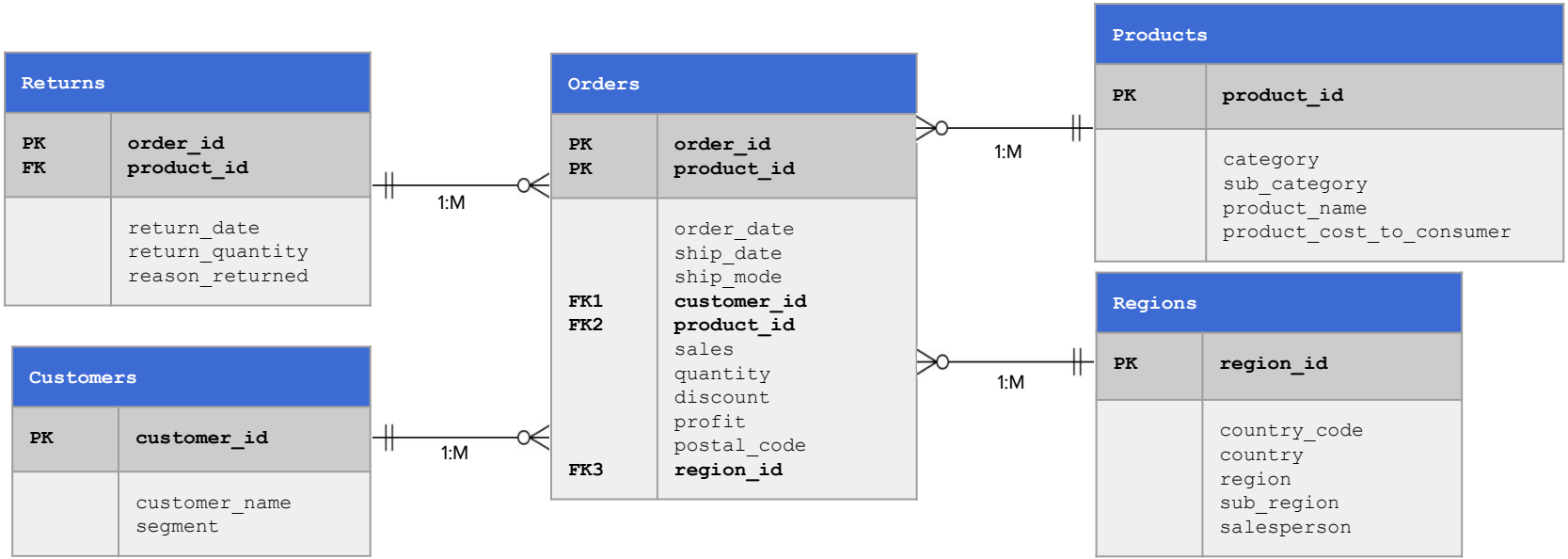
It’s Because...

- A normalized database will seek to **separate data across multiple tables**, related to each other by keys.
- This reduces redundancy, memory footprint, and **improves speed for transactional databases**.
- These databases are typically tied to an interface where it is important for the interface application to be able to **update quickly** as data is being entered, etc.

Source: TechDifferences

Discussion: Where Are Our Keys?

Take a look through our five tables. **Which columns would we use to connect these tables?**

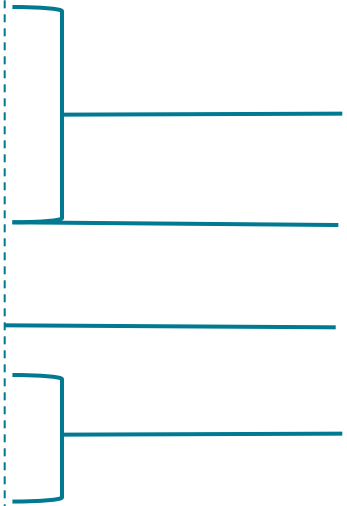


In Orders and Returns, order_id and product_id are used together to relate to the returns table; this combination of columns to create a unique row is commonly referred to as a ***concatenated*** *or* ***composite primary key***.

Sources: Relational DBD Design: Concatenated Primary Keys
Crow's Foot Notation

JOIN Syntax

```
SELECT
     orders.sales,
     regions.region
FROM
     orders
JOIN
     regions
ON
     orders.region_id =
     regions.region_id
```



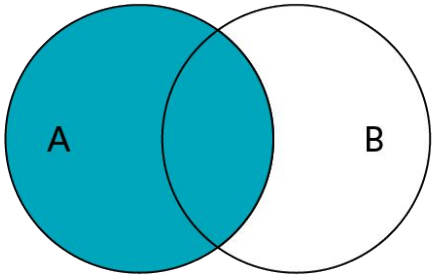
1. Designate columns we want returned, specifying the table from which they came.

2. Name the **primary table** from which we're pulling data. `orders`

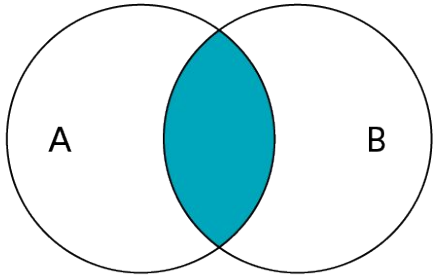
3. Name the **secondary table** from which we're pulling data. `regions`

4. Specify the **keys** to JOIN these two tables.

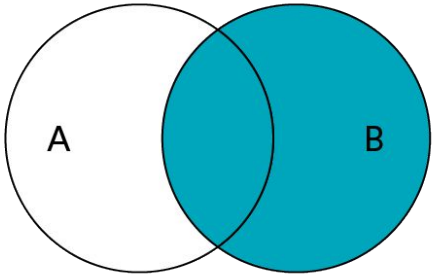
Types of JOINs



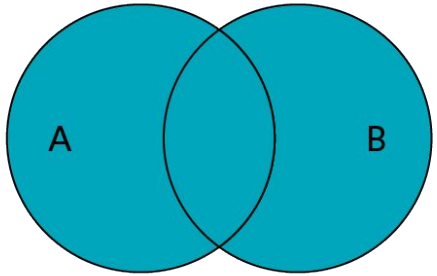
Left Join



Inner Join



Right Join



Full Outer Join

INNER JOIN is the same thing as JOIN

Image source: GA Instruction

Different types of merge

To understand different ways of merging datasets, imagine we want to merge the two datasets shown here. We'll refer to them as the **left table** and **right table.**

To perform the merge, we need to:

- Identify a column that both datasets have in common, which can be used to match the rows
- Make sure the column used for merging contains the same **type** of data in both tables (e.g. text, numbers)

Can you anticipate any problems with this merge?

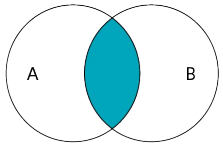
Left table

Company	NumberOfEmployees
Bloogle	10,000
Bookface	15,000
Flamazon	30,000
Webflix	1,550
Banana	3,500

Right table

Company	PayGap
Bloogle	20
Bookface	34
Flamazon	40
Witter	50
Banana	32

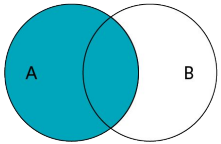
Inner merge / join



An **inner join** or **inner merge** **only merges the rows that exist in both tables**.
Any rows that exist in one table but not the other are excluded.

Company	NumberOfEmployees	PayGap
Bloogle	10,000	20
Bookface	15,000	34
Flamazon	30,000	40
Banana	3,500	32

Left merge / join

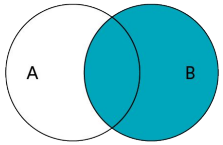


A **left join** or **left merge** includes **all rows in the left table** plus **any matches that exist in the right table**.

Any rows that exist in the left table but not in the right table are filled in as missing values.

Company	NumberOfEmployees	PayGap
Bloogle	10,000	20
Bookface	15,000	34
Flamazon	30,000	40
Webflix	1,550	NULL
Banana	3,500	32

Right merge / join

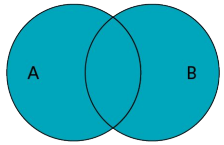


A **right join** or **right merge** includes **all rows in the right table** plus **any matches that exist in the left table**.

Any rows that exist in the right table but not in the left table are filled in as missing values.

Company	NumberOfEmployees	PayGap
Bloogle	10,000	20
Bookface	15,000	34
Flamazon	30,000	40
Witter	NULL	50
Banana	3,500	32

Outer merge / join



An **outer join** or **outer merge** **will merge all the rows from each table.**
Any rows that exist in one table but not the other are filled in with missing values.

Company	NumberOfEmployees	PayGap
Bloogle	10,000	20
Bookface	15,000	34
Flamazon	30,000	40
Webflix	1,550	NULL
Witter	NULL	50
Banana	3,500	32

One way to remember them...

Image source: https://aprogrammerlife.com/top-rated/types-of-sql-joins-1734

Guided Walk-Through:
Let's Create a JOIN!

With the global expansion of Superstore, your sources of reliable data are also growing. That's good news, right?

Yes, for the most part, but... The high volume of data can also make referencing tricky and error-prone. Just in time, you got a request from your *super* boss asking you to **identify returns by reason and order date**. This requires you to pull and combine data from these two tables:

Orders		
order_id	**order_date**	**product_id**
AE-2019-1711936	2019-11-27	OFF-BIC-10002270
AE-2019-2092798	2019-11-23	OFF-BIC-10002270

Returns		
order_id	**product_id**	**reason_returned**
AE-2019-1711936	OFF-BIC-10002270	Not Given
AE-2019-2092798	OFF-BIC-10002270	Not Given

Guided Walk-Through:
And So, a JOIN Is Born

Orders		
order_id	**order_date**	**product_id**
AE-2019-1711936	2019-11-27	OFF-BIC-10002270
AE-2019-2092798	2019-11-23	OFF-BIC-10002270

Returns		
order_id	**product_id**	**reason_returned**
AE-2019-1711936	OFF-BIC-10002270	Not Given
AE-2019-2092798	OFF-BIC-10002270	Not Given

```
SELECT
      orders.order_id,
      orders.order_date,
      orders.product_id,
      returns.return_date,
      returns.reason_returned
FROM orders
INNER JOIN returns
ON orders.order_id = returns.order_id
AND orders.product_id =
returns.product_id
LIMIT 2;
```

JOIN Result			
order_id	**order_date**	**product_id**	**reason_returned**
AE-2019-1711936	2019-11-27	OFF-BIC-10002270	Not Given
AE-2019-2092798	2019-11-23	OFF-BIC-10002270	Not Given

Working With Long Table Names

What if you're frequently referring tables with names like this one in your query?

Sales_With_Discount_Transaction_History

Imagine adding that to a column name twice as long! The solution?

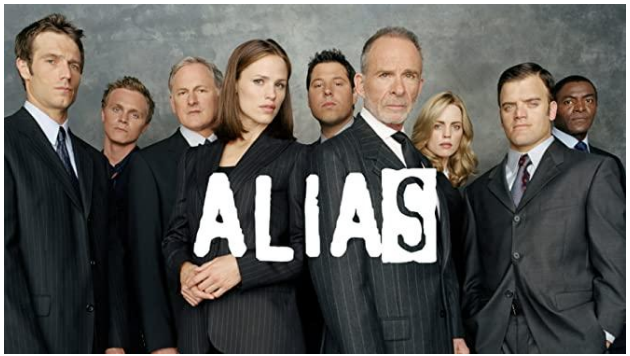


Image source: Imgflip

Shortcuts | Using an Alias

An **alias** is a shorthand name given to tables (or columns in a table) that you intend to reference repeatedly.

When creating a JOIN, each table or column can have an alias. Each column is then connected to the table by the alias. An alias usually cannot start with a number.

table1 a ➔ table1 uses the alias a.

a.column4 ➔ column4 is connected to table1 by the alias a.

Alias Syntax

Aliases are user-defined and designated:

- in the FROM statement, immediately following a table name, or
- in the SELECT statement, immediately following a column name.

Take a look at the syntaxes below. Notice that AS is in square brackets because it is optional — you don't need it to designate an alias.

Alias for tables:

```
table_name [AS] alias_name
```

Alias for columns:

```
column_name [AS] alias_name
```

Guided Walk-Through:
Aliases in a Query

```
SELECT
     orders.order_id,
     orders.order_date,
     returns.return_date
FROM
     orders
INNER JOIN returns
ON orders.order_id =
returns.order_id
AND orders.product_id =
returns.product_id;
```

Let's use an alias in this query from earlier. First, designate the aliases in **FROM**.

- Orders table will be a.
- Returns table will be b.

Next, specify the connection, by column name, on which you want to link tables:

- **ON** a.column_name = b.column_name with alias for source table.
- **USING**(column_name) only if the columns have same name in each table.

Guided Walk-Through:

Aliases in a Query | **Solution**

```
SELECT
     a.order_id,
     a.order_date,
     b.return_date

FROM
     orders a

INNER JOIN returns b
ON a.order_id = b.order_id
AND a.product_id = b.product_id;
```

This is what your query should look like with an alias for each table. Keep in mind that:

- The renaming is only temporary, and that table name does not change in the original database.
- Aliases work well when there are multiple tables in a query.

Wireframing JOINs | Single Tables

You may find drawing out tables (like below) can help you conceptualize how you plan to JOIN them. Remember, wireframes do not have to be super detailed.

Primary Table **ON** **Secondary Table**

orders o		customers c	
order_id	**customer_id**	**customer_id**	**customer_name**
AE-2016-1308551	JR-16210	JR-16210	Justin Ritter
AE-2016-1522857	KM-16375	KM-16375	Katherine Murray
....	....	....	....

JOINing Multiple Tables

You can also JOIN multiple tables together. Here is an example — notice that we have *two* JOIN statements.

Syntax: JOIN syntax restarts when you add on a new table:

```
SELECT a.field3, a.field4, b.field1, c.field4
FROM table1 a
    JOIN table2 b ON a.field1 = b.field1
    JOIN table3 c ON a.field2 = c.field1
ORDER BY b.field1
```


Wireframing JOINs | Multiple Tables

Secondary Table 1

returns r

order_id	reason_returned
AE-2016-1308551	Not Given
AE-2016-1522857	Not Needed

ON

Primary Table

orders o

order_id	customer_id
AE-2016-1308551	JR-16210
AE-2016-1522857	KM-16375
....	....

Secondary Table 2

customers c

customer_id	customer_name
JR-16210	Justin Ritter
KM-16375	Katherine Murray

Partner Exercise:

JOINing Single Tables

Working with your partner, use **Orders** as the primary table and JOIN the **Customers** table. Your query should:

1. Include **order_id** from the Orders table, and **customer_name** from the Customers table.
2. Use aliases for the tables.
3. Limit the results to 100 rows.

Before going into SQL, practice wireframing your JOINs on a piece of paper.

Partner Exercise:

JOINing Single Tables | **Solution**

Solution Query

```sql
SELECT
  o.order_id,
  c.customer_name
FROM orders o
INNER JOIN customers c ON o.customer_id = c.customer_id
LIMIT 100;
```

JOINing Multiple Tables

You can also JOIN multiple tables together. Here is an example — notice that we have *two* JOIN statements.

Syntax: JOIN syntax restarts when you add on a new table:

```
SELECT a.field3, a.field4, b.field1, c.field4
FROM table1 a
    JOIN table2 b ON a.field1 = b.field1
    JOIN table3 c ON a.field2 = c.field1
ORDER BY b.field1;
```

Wireframing JOINs | Multiple Tables

Secondary Table 1

returns r	
order_id	**reason_returned**
AE-2016-1308551	Not Given
AE-2016-1522857	Not Needed

ON

Primary Table

orders o	
order_id	**customer_id**
AE-2016-1308551	JR-16210
AE-2016-1522857	KM-16375
....	....

Secondary Table 2

customers c	
customer_id	**customer_name**
JR-16210	Justin Ritter
KM-16375	Katherine Murray

Partner Exercise:

JOINing Multiple Tables

Using **Orders** as our primary table, JOIN *both* the **Returns** *and* the **Customers** tables.

Before going into SQL, practice wireframing your JOINs on a piece of paper.

Your query should:

1. Include **order_id** from the orders table, **customer_name** from the Customers table, and **reason_returned** from the Returns table.
2. Limit results to 100 rows.

Partner Exercise:

JOINing Multiple Tables | **Solution**

Solution Query

```sql
SELECT
    o.order_id,
    r.reason_returned,
    c.customer_name
FROM orders o
INNER JOIN returns r
    ON o.order_id = r.order_id
    AND o.product_id = r.product_id
INNER JOIN customers c ON o.customer_id = c.customer_id
LIMIT 100;
```

Partner Exercise:

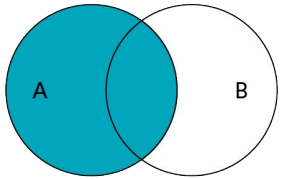
JOINing Multiple Tables | Data Output

Desired Data Output

(including the join predicates)

order_id text	product_id text	reason_returned text	customer_id text	customer_name text
AE-2019-1711936	OFF-BIC-10002270	Not Given	GH-14665	Greg Hansen
AE-2019-2092798	OFF-BIC-10002270	Not Given	GH-14665	Greg Hansen
AE-2019-2170363	OFF-BIC-10002270	Not Given	GH-14665	Greg Hansen
AE-2019-2262642	OFF-BIC-10002270	Not Given	GH-14665	Greg Hansen
AE-2019-2343602	OFF-BIC-10002270	Not Given	GH-14665	Greg Hansen
AE-2019-288592	OFF-BIC-10002270	Not Given	GH-14665	Greg Hansen
AE-2019-2952905	OFF-BIC-10002270	Not Given	GH-14665	Greg Hansen
AE-2019-3001630	OFF-BIC-10002270	Not Given	GH-14665	Greg Hansen
AE-2019-3369522	OFF-BIC-10002270	Not Given	GH-14665	Greg Hansen
AE-2019-3800683	OFF-BIC-10002270	Not Given	GH-14665	Greg Hansen
AE-2019-3959747	OFF-BIC-10002270	Not Given	GH-14665	Greg Hansen

Left JOINs



LEFT JOIN loads all entries that appear in the first table, with NULLs where there is no match.

people

id	name	vehicle_id
1	Janet	3
2	Emily	4
3	Yoko	5

vehicles

id	vehicle_name
1	Explorer
2	Civic
3	Corolla
4	Impala

id	name	vehicle_id	vehicle_name
1	Janet	3	Corolla
2	Emily	4	Impala
3	Yoko	5	NULL

Guided Walk-Through:
Let's Create a LEFT JOIN!

Let's revisit the query we wrote earlier that JOINs the Orders and Returns tables. We want to find all orders and **return information if it exists**. How should we JOIN these two tables?

Orders		
order_id	**order_date**	**product_id**
AE-2019-1711936	2019-11-27	OFF-BIC-10002270
AE-2019-2092798	2019-11-23	OFF-BIC-10002270

+

Returns		
order_id	**return_date**	**reason_returned**
AE-2019-1711936	2019-11-27	Not Given
AE-2019-2092798	2019-11-23	Not Given

Guided Walk-Through:
Creating a LEFT JOIN

Knowing that we want to keep all entries that appear in the Orders table, we'll add a LEFT JOIN that designates the Orders table as the first table. Here is our query:

```
SELECT
    o.order_id,
    o.product_id,
    r.reason_returned
FROM orders o
LEFT JOIN returns r
    ON o.order_id = r.order_id
    AND o.product_id = r.product_id
LIMIT 100;
```

Partner Exercise:

How Do We JOIN This? | Challenge

Superstore is developing a training program to help salespeople reduce the likelihood of returns. To do so, Superstore wants to interview salespeople (each salesperson has a region) who've processed higher volumes of returns in the past. You're generating a list of salespeople and return reasons (including NULL returns!). With your partner, discuss what type of JOIN(s) will you use. Be ready to explain why.

Orders	
order_id	**product_id**
AE-2019-1711936	OFF-BIC-10002270
AE-2019-2092798	OFF-BIC-10002270

Returns	
order_id	**reason_returned**
AE-2019-1711936	Not Given
AE-2019-2092798	Not Given

Region		
country	**region**	**salesperson**
United States	Americas	Kelly Williams
Mexico	Americas	Beatrice Top

Partner Exercise:

How Do We JOIN This? | Solution

```
SELECT
     rg.salesperson,
     r.reason_returned,
     COUNT(o.order_id) as count_of_returns
FROM orders o
LEFT JOIN returns r
     ON o.order_id = r.order_id
     AND o.product_id = r.product_id
INNER JOIN regions rg
     ON o.region_id = rg.region_id
GROUP BY 1,2
ORDER BY 3;
```

This aggregate is run ***after*** the JOIN on the **`returns`** and **`regions`** tables is complete.

UNIONs

Discussion:
Why UNIONs?

As we learned earlier, UNIONs combine rows from multiple tables with the same columns. In what scenarios will we use a UNION instead of a JOIN?

carpoolers		
id	name	vehicle_id
1	Janet	3
2	Emily	4
3	Yoko	4

monthly_parkers		
id	name	vehicle_id
2	Emily	4
4	Ali	5
5	Ray	1

id	name	vehicle_id
1	Janet	3
2	Emily	4
3	Yoko	4
4	Ali	5
5	Ray	1

UNION Syntax

Let's look at some simple mock syntax for a **UNION**:

```
SELECT field1
    FROM table1
UNION
SELECT field1
    FROM table2
```

Guided Walk-Through:
Exploring Examples of UNIONs

A UNION takes a single column or collection of columns and “stacks” them on top of each other. A common use case is if we have similar data between two tables and want to UNION those two tables together.

For illustration purposes, we’ll be using the following sample HR tables:

current_employees			
id	**first_name**	**last_name**	**salary**
2	Gabe	Moore	50000
3	Doreen	Mandeville	60000
5	Simone	MacDonald	55000

retired_employees			
id	**first_name**	**last_name**	**salary**
7	Madisen	Flateman	75000
11	Ian	Paasche	120000
13	Mimi	St. Felix	70000

Guided Walk-Through:

Creating a UNION for Two Tables

When you want to combine the two tables, and both tables have the same columns, you can use a UNION with a SELECT *:

```
SELECT *
FROM current_employees
UNION
SELECT *
FROM retired_employees
```

id	first_name	last_name	salary
2	Gabe	Moore	50000
3	Doreen	Mandeville	60000
5	Simone	MacDonald	55000
7	Madisen	Flateman	75000
11	Ian	Paasche	120000
13	Mimi	St. Felix	70000

Guided Walk-Through:

Creating a UNION for Two Tables (Cont.)

You can also UNION tables on only columns. These columns must match data types but do not have to represent the same data. What happened in the table below? And where do the resulting headers come from?

```
SELECT first_name,
last_name
FROM current_employees
UNION
SELECT last_name,
first_name
FROM retired_employees
```

first_name	last_name
Gabe	Moore
Doreen	Mandeville
Simone	MacDonald
Flateman	Madisen
Paasche	Ian
St. Felix	Mimi

Guided Walk-Through:
Creating a UNION

UNIONs can help organize tables into logical groups, making your SQL code more reusable and easier to debug. Let's see how this works by applying UNION to the regions table to combine region and sub-regions.

```
SELECT region, sub_region
FROM regions
WHERE sub_region = 'Central
United States'
UNION
SELECT region, sub_region
FROM regions
WHERE sub_region = 'Caribbean'
```

*	region	sub_region
1	Americas	Central United States
2	Americas	Caribbean

Guided Walk-Through:
Creating a UNION ALL

Let's rework the same example with a UNION ALL. What changed?

```
SELECT region, sub_region
FROM regions
WHERE sub_region = 'Central
United States'
UNION ALL
SELECT region, sub_region
FROM regions
WHERE sub_region = 'Caribbean'
```

*	region	sub_region
1	Americas	Central United States
2	Americas	Caribbean
3	Americas	Caribbean
4	Americas	Caribbean
5	Americas	Caribbean
6	Americas	Caribbean
7	Americas	Caribbean
8	Americas	Caribbean
9	Americas	Caribbean

Discussion: UNION ALL

We know that UNIONs remove duplicates, whereas UNION ALL allows duplicates.

Looking at the UNION ALL syntax for Superstore, what are some of the reasons why we'd want to keep duplicate values?

```
SELECT region, sub_region
FROM regions
WHERE sub_region = 'Central
United States'
UNION ALL
SELECT region, sub_region
FROM regions
WHERE sub_region = 'Caribbean'
```

Rules for Using UNIONs

Remember these rules when using UNIONs:

- You *must* match the number of columns, and they *must* be of compatible data types.
- You can only have one **ORDER BY** at the bottom of your full SELECT statement.
- UNION removes *composite* duplicates.
- UNION ALL allows duplicates.

Solo Exercise:

Optional Practice

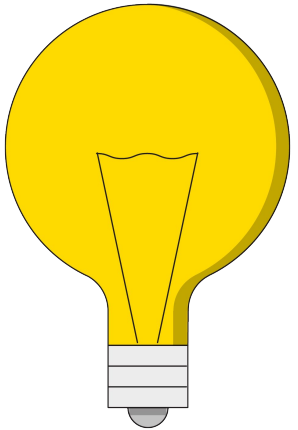
Use JOINs and UNIONs to answer the following questions:

1. Which region saw the most returned items? For what reasons?
2. What product was returned most often?
3. Which of our “top vendors” (3M, Apple, Avery, Cisco, Epson, Hewlett-Packard (HP, Hewlett Packard), Logitech, Panasonic, Samsung, and Xerox) saw the most returns?
4. Which product is most profitable with the consumer segment?

Solo Exercise:

Optional Practice | Solution 1

```sql
SELECT
     reg.region,
     ret.reason_returned,
     COUNT(DISTINCT ret.order_id) as returns_count
FROM regions reg
JOIN orders ord ON reg.region_id = ord.region_id
JOIN returns ret ON ret.order_id = ord.order_id
                    AND ret.product_id = ord.product_id
GROUP BY 1,2
ORDER BY 3 DESC;
```

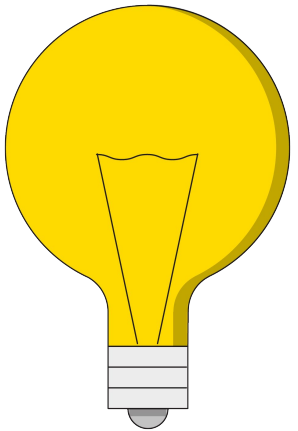


Americas and EMEA has the most returns for Not Given and Wrong Item

Solo Exercise:

Optional Practice | Solution 2

```sql
SELECT
      prd.product_id,
      prd.product_name,
      COUNT(DISTINCT ret.order_id) as returns_count
FROM products prd
JOIN orders ord ON prd.product_id = ord.product_id
JOIN returns ret ON ret.order_id = ord.order_id
                  AND ret.product_id = ord.product_id
GROUP BY 1,2
ORDER BY 3 DESC;
```

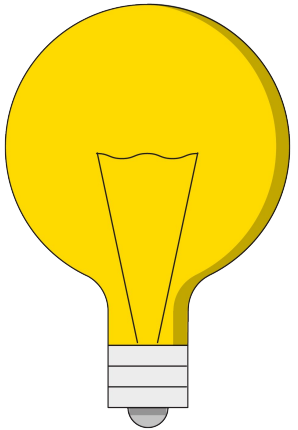


Cardinal Binding Machine, Clear and Sanford Pencil Sharpener, Easy-Erase have the most returns

Solo Exercise:

Optional Practice | Solution 3

```sql
SELECT
	CASE
		WHEN product_name ILIKE '%3M%' THEN '3M'
		WHEN product_name ILIKE '%Apple%' THEN 'Apple'
		WHEN product_name ILIKE '%Avery%' THEN 'Avery'
		WHEN product_name ILIKE '%Epson%' THEN 'Epson'
		WHEN product_name ILIKE '%HP%' OR product_name ILIKE '%Hewlett%' THEN 'Hewlett-Packard'
		WHEN product_name ILIKE '%Logitech%' THEN 'Logitech'
		WHEN product_name ILIKE '%Panasonic%' THEN 'Panasonic'
		WHEN product_name ILIKE '%Samsung%' THEN 'Samsung'
		WHEN product_name ILIKE '%Xerox%' THEN 'Xerox'
	END as vendor,
	COUNT(DISTINCT ret.order_id) as returns_count
FROM products prd
JOIN orders ord ON prd.product_id = ord.product_id
JOIN returns ret ON ret.order_id = ord.order_id AND ret.product_id = ord.product_id
WHERE product_name ILIKE '%3M%' OR product_name ILIKE '%Apple%'
	OR product_name ILIKE '%Avery%' OR product_name ILIKE '%Epson%'
	OR product_name ILIKE '%HP%' OR product_name ILIKE '%Hewlett%'
	OR product_name ILIKE '%Logitech%' OR product_name ILIKE '%Panasonic%'
	OR product_name ILIKE '%Samsung%' OR product_name ILIKE '%Xerox%'
GROUP BY 1  ORDER BY 2 DESC;
```

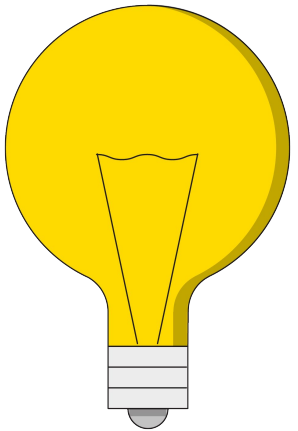


Avery and Xerox have the most returns.

Solo Exercise:

Optional Practice | Solution 4

```sql
SELECT
    prd.product_id,
    prd.product_name,
    SUM(ord.profit) as total_profit
FROM orders ord
JOIN products prd ON ord.product_id = prd.product_id
JOIN customers cst ON ord.customer_id = cst.customer_id
WHERE cst.segment = 'Consumer'
GROUP BY 1,2
ORDER BY 3 DESC;
```



Canon imageCLASS 2200 Advanced Copier is most profitable product for Consumer

Organizing Data with SQL

Wrapping Up

Recap

In today's class, we...

- Navigated a relational database.
- Practiced writing and executing SQL queries, including SELECT, FROM, WHERE, and DISTINCT SELECT.
- Worked with **CASE** to handle if/then logic and apply multiple conditions.
- Practiced writing aggregate functions: **MIN**, **MAX**, **SUM**, **AVG**, and **COUNT**.
- Used SQL commands such as **GROUP BY** and **HAVING** to group and filter data.
- Combined data from multiple sources using **JOINs** and **UNIONs**.

Looking Ahead

Optional Homework:

- Optional myGA lessons: **Beginner SQL** (unit)
- Optional Homework

Up Next: Querying Data with SQL

Additional Resources

- What's the Difference Between a Primary and Unique Key?
- Combining the AND and OR Conditions
- Logical Operators (Transact-SQL)
- Get Ready to Learn SQL Server: 4. Query Results Using Boolean Logic
- SQL **HAVING** Clause Overview
- Difference Between WHERE, GROUP BY, And HAVING clauses
- Microsoft reference material on UNIONs
- INNER JOIN Tutorial
- Common Table Expressions (where UNIONS are used frequently) — this is a more advanced concept, out of the scope of this course.
- Differences Between Normalization and Denormalization


GA