**Day 1-2: Deep Dive into Filtering & Sorting**

**SELECT**

The SELECT statement is how you **ask a database for information**. You use it to pick which columns (like categories of data) you want to see from a table. It's the most common command in SQL because it's how you read data.

For example, if you have a table of employees, you can use SELECT to only see their names and not all the other information like their salary or start date.

Here is a basic example:

**SELECT** first\_name, last\_name **FROM** employees;

This query asks the database to show you the first\_name and last\_name columns from the employees table.

**WHERE**

The WHERE clause is how you **filter your results**. You use it to specify conditions that must be true for a row to be included in your query's output. It's like telling the database, "I only want to see the data that meets this specific rule."

For example, if you're looking at a table of products, you can use WHERE to only show products that cost more than $50.

Here is a basic example:

**SELECT** product\_name **FROM** products **WHERE** price > 50;

This query asks the database to show you the product\_name for all rows in the products table where the price is greater than 50.

**ORDER BY**

The ORDER BY clause is how you **sort your results**. It lets you arrange the rows in your query's output based on the values in one or more columns. You can sort the data in **ascending order** (A to Z, 1 to 10) or **descending order** (Z to A, 10 to 1).

For example, if you have a list of students, you can use ORDER BY to sort them alphabetically by their last name.

Here is a basic example:

**SELECT** first\_name, last\_name **FROM** students **ORDER** **BY** last\_name;

This query asks the database to show you the first\_name and last\_name from the students table, sorted in ascending alphabetical order by last\_name.

**Tip to know**

By default, ORDER BY sorts in **ascending order**. If you want to sort in descending order, you must add the DESC keyword after the column name. For example: ORDER BY last\_name DESC.

**IN and NOT IN Operators**

**Definition:** The IN operator is a handy way to check if a value is **one of the items in a list**. It's a cleaner and shorter way to write many OR conditions. The NOT IN operator does the exact opposite—it finds all the values that are **not in the list**.

**Use/Importance:** These operators make your code much easier to read and write, especially when you need to filter for several specific values. Instead of writing WHERE country = 'USA' OR country = 'Canada' OR country = 'Mexico', you can use a simple list with IN.

**Syntax/Structure:**

WHERE column\_name IN (value1, value2, ...);

**Example:** To find all customers who are from the USA or Canada.

**SELECT** customer\_id, first\_name, country

**FROM** customers

**WHERE** country IN ('USA', 'Canada');

**Tip to know**

The IN operator can also be used with a subquery (a query inside another query) instead of a static list. This is useful when the list of values you want to check against is the result of another SELECT statement. This makes the IN operator very flexible and powerful.

**The BETWEEN Operator**

**Definition:** The BETWEEN operator helps you **select values that fall within a specific range**. Think of it as a convenient shortcut for WHERE clauses that use >= (greater than or equal to) and <= (less than or equal to) with the AND operator. The range you set is **inclusive**, meaning the starting and ending values are both part of the result.

**Use/Importance:** This operator is very useful for filtering numbers, text, or dates that are in a continuous sequence. It makes your queries shorter and easier to read. For instance, you can use it to find all products with a price between $10 and $20, or to find all records from a specific month.

**Syntax/Structure:**

**WHERE** column\_name **BETWEEN** start\_value **AND** end\_value;

**Example:** To find all products with a price between $10 and $20.

**SELECT** product\_name, price

**FROM** products

**WHERE** price **BETWEEN** 10 **AND** 20;

**Tip to know**

When using BETWEEN with dates, be aware of the time component. If your date column includes time, BETWEEN '2025-01-01' AND '2025-01-31' will only include records up to the very beginning of January 31. To include all of January 31, you might need to use '2025-01-31 23:59:59' as your end value or use CAST to ignore the time. A simpler method is to use < '2025-02-01' as your end condition.

**The LIKE Operator and Wildcards (%, \_)**

**Definition:** The LIKE operator is used to search for specific patterns within a column of text. It's a "fuzzy" search, meaning you don't need the exact word to find what you're looking for. It works with two special characters called **wildcards**:

* % (percent sign): This wildcard matches **any number of characters**, including zero.
* \_ (underscore): This wildcard matches **exactly one single character**.

**Use/Importance:** This is crucial for building search features, such as finding all customer names that start with a certain letter or finding products with a particular word in their description. It gives you flexibility when the data isn't perfectly structured.

**Syntax/Structure:**

**WHERE** column\_name **LIKE** 'pattern';

**Example:** To find all customers whose last name contains the letters 'son'.

**SELECT** first\_name, last\_name

**FROM** customers

**WHERE** last\_name **LIKE** '%son%';

**Tip to know**

When using LIKE with the underscore (\_) wildcard, each underscore represents one character, so \_ \_ \_ \_ is very different from \_. For example, LIKE 'P\_ne' would match 'Pine' but not 'Pne'. This can be a common point of confusion for beginners.

**Handling NULL Values**

**Definition:** In a database, NULL is a special marker that represents a **missing, unknown, or undefined value**. It's important to remember that NULL is not the same as zero (0) or an empty string (''). Because of its unique nature, you cannot use standard comparison operators like = or != with NULL. Instead, you must use the special operators IS NULL to find missing values or IS NOT NULL to find rows that have a value.

**Use/Importance:** Dealing with NULL values is a critical part of working with real-world data, which is rarely perfect. Knowing how to find or exclude these records is essential for data cleaning, reporting, and ensuring your calculations are accurate. For example, if you're calculating an average salary, including NULL values could lead to errors.

**Syntax/Structure:**

**WHERE** column\_name **IS** **NULL**;

**Example:** To find all customers who do not have an email address on file.

**SELECT** customer\_id, first\_name

**FROM** customers

**WHERE** email IS **NULL**;

**Tip to know**

When you perform mathematical operations or use aggregate functions (like SUM or AVG) on columns that contain NULL values, the NULLs are typically ignored. This means the result of SUM(column) will be the sum of all the non-NULL values in that column, and the count used for AVG will only include non-NULL values. This is an important behavior to remember when analyzing data.