**Basic Sql Command**

**Select and From Command:**

Every query will have at least a **SELECT** and **FROM** statement.

The **SELECT** statement is where you put the **columns** for which you would like to show the data.

The **FROM** statement is where you put the **tables** from which you would like to pull data.

**Example:**

select only the id, account\_id, and occurred\_atcolumns for all orders in the **orders** table.

Answer:

select id, account\_id ,occurred\_at from orders

**Limit**

The limiting functionality is built into Mode to prevent you from accidentally returning millions of rows without meaning to (we’ve all done it). However,

if you’re ever using SQL outside of Mode, you can manually add a limit with a SQL command.

Example:

writing a query that limits the response to only the first 15 rows and includes the occurred\_at, account\_id, and channel fields in the **web\_events** table.

**select account\_id,occurred\_at,channel from web\_events limit 15;**

**Order By**

The ORDER BY keyword is used to sort the result-set in ascending or descending order.

The ORDER BY keyword sorts the records in ascending order by default. To sort the records in descending order, use the DESC keyword.

Example:

Write a query to return the 10 earliest orders in the **orders** table. Include the id, occurred\_at, and total\_amt\_usd.

select id,occurred\_at,total\_amt\_usd from orders order by total asc limit 20;

**Where**

The WHERE clause is used to filter records.

The WHERE clause is used to extract only those records that fulfill a specified condition.

Example

Pull the first 10 rows and all columns from the **orders** table that have a total\_amt\_usd less than 500.

select \* from orders where total\_amt\_usd<500 limit 10

**Arithmetic Operations :+,-,\*,/ can be used in sql we can add two columns create new column**

Example:

**SELECT** **id**, account\_id, standard\_amt\_usd/standard\_qty **AS** unit\_price

**FROM** orders

**LIMIT** 10;

**LIKE**

This operator is used in a WHERE clause to search for a specified pattern in a column.

There are two wildcards used in conjunction with the LIKE operator:

* % - The percent sign represents zero, one, or multiple characters
* \_ - The underscore represents a single character

EXample :All the companies whose names start with 'C'.

select \* from accounts where name like 'C%'

**IN**

This operator allows you to specify multiple values in a WHERE clause.

The IN operator is a shorthand for multiple OR conditions.

Example :Use the **accounts** table to find the account name, primary\_poc, and sales\_rep\_id for Walmart, Target, and Nordstrom.

select name,primary\_poc,sales\_rep\_id from accounts

where name in ('Walmart','Target','Nordstrom')

## The SQL AND, OR and NOT Operators

The WHERE clause can be combined with AND, OR, and NOT operators.

The AND and OR operators are used to filter records based on more than one condition:

* The AND operator displays a record if all the conditions separated by AND is TRUE.
* The OR operator displays a record if any of the conditions separated by OR is TRUE.
* The NOT operator displays a record if the condition(s) is NOT TRUE.

Not Example

Use the **accounts** table to find the account name, primary poc, and sales rep id for all stores except Walmart, Target, and Nordstrom.

select name,primary\_poc,sales\_rep\_id from accounts

where name not in('Walmart', 'Target','Nordstrom')

And Example

Write a query that returns all the **orders** where the standard\_qty is over 1000, the poster\_qty is 0, and the gloss\_qty is 0.

select \* From orders where standard\_qty >1000 and poster\_qty=0 and gloss\_qty=0

**BETWEEN** is tricky for dates! While **BETWEEN** is generally inclusive of endpoints, it assumes the time is at 00:00:00 (i.e. midnight) for dates.

Or Example

Find all the company names that start with a 'C' or 'W', and the primary contact **contains** 'ana' or 'Ana'

Select \* from accounts where name like 'C%' or name like 'W%' and primary\_poc like '%ana%' or primary\_poc like '%Ana'

Over view of all commands given above:

| **Statement** | **How to Use It** | **Other Details** |
| --- | --- | --- |
| SELECT | SELECT **Col1**, **Col2**, ... | Provide the columns you want |
| FROM | FROM **Table** | Provide the table where the columns exist |
| LIMIT | LIMIT **10** | Limits based number of rows returned |
| ORDER BY | ORDER BY **Col** | Orders table based on the column. Used with **DESC**. |
| WHERE | WHERE **Col > 5** | A conditional statement to filter your results |
| LIKE | WHERE **Col LIKE '%me%'** | Only pulls rows where column has 'me' within the text |
| IN | WHERE **Col IN ('Y', 'N')** | A filter for only rows with column of 'Y' or 'N' |
| NOT | WHERE **Col NOT IN ('Y', 'N')** | **NOT** is frequently used with **LIKE** and **IN** |
| AND | WHERE **Col1 > 5 AND Col2 < 3** | Filter rows where two or more conditions must be true |
| OR | WHERE **Col1 > 5 OR Col2 < 3** | Filter rows where at least one condition must be true |
| BETWEEN | WHERE **Col BETWEEN 3 AND 5** | Often easier syntax than using an **AND** |

**JOIN**s.

This is the real secret (well not really a secret) behind the success of SQL as a language. **JOIN**s allow us to combine multiple tables together. All of the operations we learned here will still be important moving forward, but we will be able to answer much more complex questions by combining information from multiple tables!

we have introduced two new parts to our regular queries: **JOIN** and **ON**. The **JOIN** introduces the second table from which you would like to pull data, and the **ON** tells you how you would like to merge the tables in the **FROM** and **JOIN** statements together.

**SELECT** orders.\*

**FROM** orders

**JOIN** accounts

**ON** orders.account\_id = accounts.**id**;

### Primary Key (PK)

A **primary key** is a unique column in a particular table. This is the first column in each of our tables. Here, those columns are all called **id**, but that doesn't necessarily have to be the name. **It is common that the primary key is the first column in our tables in most databases.**

### Foreign Key (FK)

A **foreign key** is when we see a primary key in another table. Which used to link two table together

If we wanted to join all three of these tables, we could use the same logic. The code below pulls all of the data from all of the joined tables.

**SELECT** \*

**FROM** web\_events

**JOIN** accounts

**ON** web\_events.account\_id = accounts.**id**

**JOIN** orders

**ON** accounts.**id** = orders.account\_id

When we **JOIN** tables together, it is nice to give each table an **alias**. Frequently an alias is just the first letter of the table name. You actually saw something similar for column names in the **Arithmetic Operators** concept.

Example:

**FROM** tablename AS t1

**JOIN** tablename2 AS t2

Join Example

Provide a table for all **web\_events** associated with account **name** of Walmart. There should be three columns. Be sure to include the primary\_poc, time of the event, and the channel for each event. Additionally, you might choose to add a fourth column to assure only Walmart events were chosen.

**SELECT** a.primary\_poc, w.occurred\_at, w.channel, a.**name**

**FROM** web\_events w

**JOIN** accounts a

**ON** w.account\_id = a.**id**

**WHERE** a.**name** = 'Walmart';

Provide a table that provides the **region** for each **sales\_rep** along with their associated **accounts**. Your final table should include three columns: the region **name**, the sales rep **name**, and the account **name**. Sort the accounts alphabetically (A-Z) according to account name.

**SELECT** r.**name** region, s.**name** rep, a.**name** **account**

**FROM** sales\_reps s

**JOIN** region r

**ON** s.region\_id = r.**id**

**JOIN** accounts a

**ON** a.sales\_rep\_id = s.**id**

**ORDER** **BY** a.**name**;

**INNER** JOIN

This keyword selects records that have matching values in both tables.

SELECT column\_name(s)  
FROM table1  
INNER JOIN table2 ON table1.column\_name = table2.column\_name;



**LEFT JOIN**

This keyword returns all records from the left table (table1), and the matched records from the right table (table2). The result is NULL from the right side, if there is no match.

### **LEFT JOIN Syntax**

SELECT column\_name(s)  
FROM table1  
LEFT JOIN table2 ON table1.column\_name = table2.column\_name;

**Note:** In some databases LEFT JOIN is called LEFT OUTER JOIN.



**RIGHT JOIN**

This keyword returns all records from the right table (table2), and the matched records from the left table (table1). The result is NULL from the left side, when there is no match.

### **RIGHT JOIN Syntax**

SELECT column\_name(s)  
FROM table1  
RIGHT JOIN table2 ON table1.column\_name = table2.column\_name;

**Note:** In some databases RIGHT JOIN is called RIGHT OUTER JOIN.



**FULL OUTER JOIN**

This keyword return all records when there is a match in either left (table1) or right (table2) table records.

**Note:** FULL OUTER JOIN can potentially return very large result-sets!

### **FULL OUTER JOIN Syntax**

SELECT column\_name(s)  
FROM table1  
FULL OUTER JOIN table2 ON table1.column\_name = table2.column\_name;



Sql Aggregation

## What is a NULL Value?

A field with a NULL value is a field with no value.

If a field in a table is optional, it is possible to insert a new record or update a record without adding a value to this field. Then, the field will be saved with a NULL value.

## How to Test for NULL Values?

It is not possible to test for NULL values with comparison operators, such as =, <, or <>.

We will have to use the IS NULL and IS NOT NULL operators instead.

### **IS NULL Syntax**

SELECT column\_namesFROM table\_name  
WHERE column\_name IS NULL;

### **IS NOT NULL Syntax**

SELECT column\_namesFROM table\_name  
WHERE column\_name IS NOT NULL;

**COUNT ( ) this function returns the number of rows that matches a specified criteria.**

**AVG( ) this function returns the average value of a numeric column.**

**SUM( ) this function returns the total sum of a numeric column.**

### **COUNT() Syntax**

SELECT COUNT(column\_name)  
FROM table\_name  
WHERE condition;

### **AVG() Syntax**

SELECT AVG(column\_name)  
FROM table\_name  
WHERE condition;

### **SUM() Syntax**

SELECT SUM(column\_name)  
FROM table\_name  
WHERE condition;

The MIN() function returns the smallest value of the selected column.

The MAX() function returns the largest value of the selected column.

### **MIN() Syntax**

SELECT MIN(column\_name)  
FROM table\_name  
WHERE condition;

### **MAX() Syntax**

SELECT MAX(column\_name)  
FROM table\_name  
WHERE condition;

### GROUP BY

Now that you have been introduced to **JOIN**s, **GROUP BY**, and aggregate functions, the real power of **SQL** starts to come to life. Try some of the below to put your skills to the test!

The GROUP BY statement is often used with aggregate functions (COUNT, MAX, MIN, SUM, AVG) to group the result-set by one or more columns.

### **GROUP BY Syntax**

SELECT column\_name(s)  
FROM table\_name  
WHERE condition  
GROUP BY column\_name(s)ORDER BY column\_name(s)

Find the total sales in **usd** for each account. You should include two columns - the total sales for each company's orders in **usd** and the company **name**.

**SELECT** a.**name**, **SUM**(total\_amt\_usd) total\_sales

**FROM** orders o

**JOIN** accounts a

**ON** a.**id** = o.account\_id

**GROUP** **BY** a.**name**;

For each account, determine the average amount of each type of paper they purchased across their orders. Your result should have four columns - one for the account **name** and one for the average quantity purchased for each of the paper types for each account.

**SELECT** a.**name**, **AVG**(o.standard\_qty) avg\_stand, **AVG**(o.gloss\_qty) avg\_gloss, **AVG**(o.poster\_qty) avg\_post

**FROM** accounts a

**JOIN** orders o

**ON** a.**id** = o.account\_id

**GROUP** **BY** a.**name**;

**DISTINCT**

**This**  is always used in **SELECT** statements, and it provides the unique rows for all columns written in the **SELECT** statement. Therefore, you only use **DISTINCT** once in any particular **SELECT** statement.

You could write:

**SELECT** **DISTINCT** column1, column2, column3

**FROM** table1;

which would return the unique (or **DISTINCT**) rows across all three columns.

### HAVING

**HAVING** is the “clean” way to filter a query that has been aggregated, but this is also commonly done using a **[subquery](https://community.modeanalytics.com/sql/tutorial/sql-subqueries/" \t "_blank)**. Essentially, any time you want to perform a **WHERE** on an element of your query that was created by an aggregate, you need to use **HAVING** syntax

SELECT column\_name(s)  
FROM table\_name  
WHERE condition  
GROUP BY column\_name(s)HAVING conditionORDER BY column\_name(s);

Which account has the most orders?

**SELECT** a.**id**, a.**name**, **COUNT**(\*) num\_orders

**FROM** accounts a

**JOIN** orders o

**ON** a.**id** = o.account\_id

**GROUP** **BY** a.**id**, a.**name**

**ORDER** **BY** num\_orders **DESC**

**LIMIT** 1

How many accounts spent more than 30,000 usd total across all orders?

**SELECT** a.**id**, a.**name**, **SUM**(o.total\_amt\_usd) total\_spent

**FROM** accounts a

**JOIN** orders o

**ON** a.**id** = o.account\_id

**GROUP** **BY** a.**id**, a.**name**

**HAVING** **SUM**(o.total\_amt\_usd) > 30000

**ORDER** **BY** total\_spent;

**DATE\_TRUNC**.

**DATE\_TRUNC** allows you to truncate your date to a particular part of your date-time column. Common trunctions are day, month, and year. [**Here**](https://blog.modeanalytics.com/date-trunc-sql-timestamp-function-count-on/) is a great blog post by Mode Analytics on the power of this function.

**DATE\_PART** can be useful for pulling a specific portion of a date, but notice pulling month or day of the week (dow) means that you are no longer keeping the years in order. Rather you are grouping for certain components regardless of which year they belonged in.

For additional functions you can use with dates, check out the documentation [**here**](https://www.postgresql.org/docs/9.1/static/functions-datetime.html), but the **DATE\_TRUNC** and **DATE\_PART** functions definitely give you a great start!

You can reference the columns in your select statement in **GROUP BY** and **ORDER BY**clauses with numbers that follow the order they appear in the select statement.

For example

SELECT standard\_qty, COUNT(\*)

FROM orders

GROUP BY 1 (this 1 refers to standard\_qty since it is the first of the columns included in the select statement)

ORDER BY 1 (this 1 refers to standard\_qty since it is the first of the columns included in the select statement)

Example

Find the sales in terms of total dollars for all orders in each year, ordered from greatest to least. Do you notice any trends in the yearly sales totals?

**SELECT** DATE\_PART('year', occurred\_at) ord\_year, **SUM**(total\_amt\_usd) total\_spent

**FROM** orders

**GROUP** **BY** 1

**ORDER** **BY** 2 **DESC**;

When we look at the yearly totals, you might notice that 2013 and 2017 have much smaller totals than all other years. If we look further at the monthly data, we see that for 2013 and 2017 there is only one month of sales for each of these years (12 for 2013 and 1 for 2017). Therefore, neither of these are evenly represented. Sales have been increasing year over year, with 2016 being the largest sales to date. At this rate, we might expect 2017 to have the largest sales.

### **CASE**

* The CASE statement always goes in the SELECT clause.
* CASE must include the following components: WHEN, THEN, and END. ELSE is an optional component to catch cases that didn’t meet any of the other previous CASE conditions.
* You can make any conditional statement using any conditional operator (like [**WHERE**](https://community.modeanalytics.com/sql/tutorial/sql-where/)) between WHEN and THEN. This includes stringing together multiple conditional statements using AND and OR.
* You can include multiple WHEN statements, as well as an ELSE statement again, to deal with any unaddressed conditions.

### Example

In a quiz question in the previous Basic SQL lesson, you saw this question:

1. Create a column that divides the standard\_amt\_usd by the standard\_qty to find the unit price for standard paper for each order. Limit the results to the first 10 orders, and include the id and account\_id fields. **NOTE - you will be thrown an error with the correct solution to this question. This is for a division by zero. You will learn how to get a solution without an error to this query when you learn about CASE statements in a later section.**

Let's see how we can use the **CASE** statement to get around this error.

**SELECT** **id**, account\_id, standard\_amt\_usd/standard\_qty **AS** unit\_price

**FROM** orders

**LIMIT** 10;

Now, let's use a **CASE** statement. This way any time the **standard\_qty** is zero, we will return 0, and otherwise we will return the **unit\_price**.

**SELECT** account\_id, **CASE** **WHEN** standard\_qty = 0 **OR** standard\_qty **IS** NULL **THEN** 0

**ELSE** standard\_amt\_usd/standard\_qty **END** **AS** unit\_price

**FROM** orders

**LIMIT** 10;

We would like to understand 3 different branches of customers based on the amount associated with their purchases. The top branch includes anyone with a Lifetime Value (total sales of all orders) greater than 200,000 usd. The second branch is between 200,000 and 100,000 usd. The lowest branch is anyone under 100,000 usd. Provide a table that includes the **level** associated with each **account**. You should provide the **account name**, the **total sales of all orders** for the customer, and the **level**. Order with the top spending customers listed first.

**SELECT** a.**name**, **SUM**(total\_amt\_usd) total\_spent,

**CASE** **WHEN** **SUM**(total\_amt\_usd) > 200000 **THEN** 'top'

**WHEN** **SUM**(total\_amt\_usd) > 100000 **THEN** 'middle'

**ELSE** 'low' **END** **AS** customer\_level

**FROM** orders o

**JOIN** accounts a

**ON** o.account\_id = a.**id**

**GROUP** **BY** a.**name**

**ORDER** **BY** 2 **DESC**;