**SELECT, WHERE, DISTINCT, LIKE, ORDER BY**

**SELECT** \* FROM moviesdb.movies;

**SELECT** title, industry FROM movies;

**SELECT** \* FROM movies **WHERE** industry="bollywood";

**SELECT** **count**(\*) FROM movies **WHERE** industry="bollywood";

**SELECT** **distinct** industry FROM movies;

https://www.imdb.com/

**SELECT** \* FROM movies **WHERE** title **LIKE** "%THOR%";

**SELECT** \* FROM movies **WHERE** title **LIKE** "america%";

**SELECT** \* FROM movies **WHERE** studio="";

**Takeaways**

- **SELECT**, **FROM** and **WHERE** are the basic SQL functions

- **'\*'** means all columns. Using **'\*'** after database after the SELECT query will select all columns of

- With the help of the **USE function**, you can indicate the query to use a particular database, especially when there are multiple databases

- The **COUNT function** will provide the numerical count of rows

-The **DISTINCT function** will help you see the unique values present in a given column

- **'%'** is a **wild card** search

- Use **LIKE function** and **'%'** to filter the rows based on a text value

**SELECT** \* FROM movies **WHERE** imdb\_rating>=9

**SELECT** \* FROM movies **WHERE** imdb\_rating<=5

**SELECT** \* FROM movies **WHERE** imdb\_rating>=6 **AND** imdb\_rating<=8

**SELECT** \* FROM movies **WHERE** imdb\_rating **BETWEEN** 6 **AND** 8

**SELECT** \* FROM movies **WHERE** release\_year=2022 **OR** release\_year=2019 **OR** release\_year=2018

**SELECT** \* FROM movies **WHERE** release\_year **IN** (2022,2019,2018)

**SELECT** \* FROM movies **WHERE** studio **IN** ("Marvel Studios","Zee Studios")

**SELECT** \* FROM movies **WHERE** imdb\_rating **IS** **NULL**

**SELECT** \* FROM movies **WHERE** imdb\_rating **IS NOT NULL**

**SELECT** \*FROM movies **WHERE** industry="bollywood" **ORDER BY** imdb\_rating

**SELECT \*** FROM movies **WHERE** industry="bollywood" **ORDER BY** imdb\_rating **desc**

**SELECT \*** FROM movies **WHERE** industry="bollywood" **ORDER BY** imdb\_rating **asc**

**SELECT \*** FROM movies **WHERE** industry="bollywood" **ORDER BY** imdb\_rating **desc LIMIT** 5

Export the data

**SELECT \*** FROM movies **WHERE** industry="bollywood" **ORDER BY** imdb\_rating **desc LIMIT** 5 **OFFSET** 1

**Takeaways**

• **<, <=, >, >=** are the basic numerical operators used in SQL.

• You can also use **AND, OR, BETWEEN, IN** to perform numerical queries.

• You can sort the table by using **'ORDER BY'** clause.

• By default, it sorts the data in ascending order but you can specify the sort order.

• **LIMIT** clause can be used to fetch the top **'N'** or bottom **'N'** amount of records. **'N'** can be any numerical value.

queries

• **OFFSET** clause help you to skip a certain number of rows in your final result.

**Summary Analytics (MIN, MAX, AVG, GroupBy)**

**SELECT** **MAX** (imdb\_rating) FROM movies **WHERE** industry="bollywood";

**SELECT** **MIN** (imdb\_rating) FROM movies **WHERE** industry="bollywood";

**SELECT** **AVG** (imdb\_rating) FROM movies **WHERE** studio="Marvel studios";

**SELECT** **ROUND** (**AVG** (imdb\_rating),2) FROM movies **WHERE** studio="Marvel studios";

**SELECT** **ROUND** (**AVG** (imdb\_rating),2) **as** avg\_rating FROM movies **WHERE** studio="Marvel studios";

**SELECT** **MIN**(imdb\_rating) **as** min\_rating, **MAX**(imdb\_rating) **as** max\_rating,

**ROUND**(**AVG** (imdb\_rating),2) **as** avg\_rating FROM movies **WHERE** studio="Marvel studios";

**SELECT** **COUNT** (\*) FROM movies **WHERE** industry="bollywood";

**SELECT** industry, **COUNT** (\*) FROM movies **GROUP BY** industry;

**SELECT** studio, **COUNT** (\*) FROM movies **GROUP BY** studio;

**SELECT** studio, **COUNT**(\*) **as** cnt FROM movies **GROUP BY** studio **ORDER BY cnt DESC**;

**SELECT** industry, **COUNT**(industry) **as** cnt, **avg**(imdb\_rating) **as** avg\_rating FROM movies **GROUP BY** industry;

**SELECT** industry, **COUNT**(industry) **as** cnt, **ROUND(avg**(imdb\_rating),1) **as** avg\_rating FROM movies **GROUP BY** industry;

**SELECT** studio, **COUNT**(studio) **as** cnt, **ROUND**(**avg**(imdb\_rating),1) **as** avg\_rating FROM movies **GROUP BY** studio **ORDER BY** avg\_rating **DESC**;

**HAVING clause, Calculated Columns**

#Print all the years where more than 2 movies were released

**SELECT** release\_year, **COUNT**(\*) FROM movies **GROUP BY** release\_year;

**SELECT** release\_year, **COUNT**(\*) **as** movies\_count FROM movies **GROUP BY** release\_year **ORDER BY** movies\_count **DESC**;

**SELECT** release\_year, **COUNT**(\*) **as** movies\_count FROM movies **WHERE** movies\_count>2 **GROUP BY** release\_year **ORDER BY** movies\_count **DESC**;

# Error Code: 1054. Unknown column 'movies\_count' in 'where clause'

FROM ---> WHERE ---> GROUP BY ---> HAVING ---> ORDER BY

**SELECT** release\_year, **COUNT**(\*) **as** movies\_count FROM movies **GROUP BY** release\_year **HAVING** movies\_count>2 **ORDER BY** movies\_count **DESC**;

**Takeaways**

• The order of query execution in - **SQL** is **FROM** **WHERE → GROUP BY HAVING ORDER BY**

• **GROUP BY** and **HAVING** clauses are often used together

• The Column you use in **HAVING** should be present in **SELECT** clause whereas **WHERE** can use columns that is not present in select clause as well

**SELECT** \* FROM moviesdb.actors;

**SELECT** **CURDATE()**;

**SELECT** **YEAR**(**CURDATE())**;

**SELECT** \*, **YEAR**(**CURDATE())-birth\_year as age** FROM actors;

**SELECT** \* FROM moviesdb.financials;

**SELECT** \*, (revenue-budget) **as** profit **FROM** financials;

**SELECT** \*, **if**(currency='USD',revenue\*77,revenue) **as** revenue\_inr **FROM** financials;

**SELECT distinct** unit **FROM** financials;

**SELECT** \*,

**CASE**

**WHEN** unit="thousands" **THEN** revenue/1000

**WHEN** unit="billions" **THEN** revenue\*1000

**ELSE** revenue

**END** **as** revenue\_mln

**FROM** financials;

**Takeaways**

* You can derive new columns from the existing columns in a table
* As a data analyst, **Revenue** and **Profit** are the most common metrics that you will calculate in any industry
* **Currency conversion** and **unit conversion** are important business use cases of SQL
* **IF function** is often used in SQL queries
* When you have more than two conditions, you need to use **CASE** and **END** function instead of **IF Function**

**Takeaways**

* Companies use **multiple tables** to store data
* To save space by **avoiding repetition**
* **Organize** data better
* Make **updates easily**
* In SQL you will be able to **map multiple tables** using JOIN clause

**Inner join**

**SELECT** movies.movie\_id,title,budget,revenue,currency,unit

from movies **join** financials

**on** movies.movie\_id=financials.movie\_id;

**(OR)**

**SELECT** m.movie\_id,title,budget,revenue,currency,unit

from movies m **Inner join** financials f

**on** m.movie\_id=f.movie\_id;

**Left Join**

**SELECT** m.movie\_id,title,budget,revenue,currency,unit

from movies m **Left join** financials f **on** m.movie\_id=f.movie\_id;

**Right Join**

**SELECT** f.movie\_id,title,budget,revenue,currency,unit

from movies m **Right join** financials f **on** m.movie\_id=f.movie\_id;

**Full Join**

**SELECT** m.movie\_id,title,budget,revenue,currency,unit

from movies m **Left join** financials f **on** m.movie\_id=f.movie\_id

**union**

**SELECT** f.movie\_id,title,budget,revenue,currency,unit

from movies m **Right join** financials f **on** m.movie\_id=f.movie\_id

**Using**

**SELECT** movie\_id,title,budget,revenue,currency,unit

from movies  **Left join** financials  **Using(**movie\_id)

**SELECT** movie\_id,title,budget,revenue,currency,unit

from movies  **Right join** financials  **Using(**movie\_id)

In SQL, when writing a SELECT query, understanding when and where to use WHERE, GROUP BY, HAVING, and ORDER BY is essential to ensure that the query is logically correct and optimized. Each clause serves a different purpose, and they should be applied in a specific sequence in the query.

**1. WHERE Clause**

* **Purpose**: The WHERE clause is used to filter rows before any grouping or aggregation occurs. It limits the rows that are selected based on a given condition.
* **When to Use**:
  + Use it to filter rows that you don't want in the result set.
  + It can apply to columns that are not part of an aggregation or grouping.

**2. GROUP BY Clause**

* **Purpose**: The GROUP BY clause groups rows that have the same values into summary rows, like "total sales per region" or "average salary per department."
* **When to Use**:
  + Use it when you need to aggregate data (using functions like SUM(), COUNT(), AVG(), MAX(), MIN()) and group by one or more columns.
  + It comes **after** the WHERE clause but before the HAVING and ORDER BY clauses.

**3. HAVING Clause**

* **Purpose**: The HAVING clause is used to filter groups after they have been formed by the GROUP BY clause. It allows you to filter the results of aggregated data.
* **When to Use**:
  + Use it when you need to filter based on the result of aggregate functions, for example, filtering groups with a total greater than a certain value.
  + It is used **after** the GROUP BY clause and **before** ORDER BY.

**4. ORDER BY Clause**

* **Purpose**: The ORDER BY clause is used to sort the result set based on one or more columns, either in ascending (default) or descending order.
* **When to Use**:
  + Use it when you want to sort the result data in a specific order.
  + It is applied **after** the WHERE, GROUP BY, and HAVING clauses.

**Sequence of Execution in SQL Query**

When SQL processes a query, it follows a specific order of operations, which is as follows:

1. **FROM**: The query starts by identifying the source table(s) for the data.
2. **WHERE**: Filters rows before any grouping takes place.
3. **GROUP BY**: Groups the rows according to the specified columns.
4. **HAVING**: Filters the groups formed by GROUP BY (only used if aggregation is involved).
5. **SELECT**: Selects the columns to return in the result set (including aggregated columns).
6. **ORDER BY**: Sorts the result set.

**Example Query**

**SELECT** department, **COUNT**(employee\_id) **AS** employee\_count, **AVG**(salary) **AS** avg\_salary

**FROM** employees

**WHERE** hire\_date > '2020-01-01' -- Filter rows based on condition

**GROUP BY** department -- Group by department

**HAVING** **COUNT**(employee\_id) > 5 -- Filter groups having more than 5 employees

**ORDER BY** avg\_salary **DESC**; -- Sort by average salary in descending order

**In summary:**

* **WHERE:** Filters rows before grouping.
* **GROUP BY:** Groups rows for aggregation.
* **HAVING:** Filters groups after they are formed (only if there's aggregation).
* **ORDER BY:** Sorts the final result set.

The sequence for writing a SELECT query should follow this order:

1. **FROM**
2. **WHERE**
3. **GROUP BY**
4. **HAVING**
5. **SELECT**
6. **ORDER BY**

**What are joins in sql and its types**

**SQL Joins** are used to combine records from two or more tables in a database based on a related column between them. Joins help retrieve meaningful data by linking different tables that store related information. There are several types of joins in SQL, each serving a specific purpose based on how they combine the tables.

Here’s an overview of the types of joins and their syntax:

### ****1. INNER JOIN****

* **Purpose**: An INNER JOIN returns only the rows where there is a match in both tables.
* **When to Use**: Use INNER JOIN when you want to retrieve data that has matching rows in both tables.

### ****2. LEFT JOIN (or LEFT OUTER JOIN)****

* **Purpose**: A LEFT JOIN returns all rows from the left table and the matching rows from the right table. If no match is found in the right table, NULL is returned for the right table's columns.
* **When to Use**: Use LEFT JOIN when you want to include all rows from the left table and match them with rows from the right table.

**3. RIGHT JOIN (or RIGHT OUTER JOIN)**

* **Purpose**: A RIGHT JOIN returns all rows from the right table and the matching rows from the left table. If no match is found in the left table, NULL is returned for the left table's columns.
* **When to Use**: Use RIGHT JOIN when you want to include all rows from the right table and match them with rows from the left table.

**4. FULL OUTER JOIN**

* **Purpose**: A FULL OUTER JOIN returns all rows from both the left and right tables. If there is no match, NULL is returned for the missing side.
* **When to Use**: Use FULL OUTER JOIN when you want to return all rows from both tables, whether or not they have a match.

**5. CROSS JOIN**

* **Purpose**: A CROSS JOIN returns the Cartesian product of the two tables. This means it combines each row from the first table with each row from the second table.
* **When to Use**: Use CROSS JOIN when you need all possible combinations of rows from both tables.

**6. SELF JOIN**

* **Purpose**: A SELF JOIN is a join where a table is joined with itself. It is often used to compare rows within the same table.
* **When to Use**: Use a SELF JOIN when you need to compare rows within the same table, like finding employees who work for the same department.

| **Join Type** | **Description** | **Returns** |
| --- | --- | --- |

|  |  |  |
| --- | --- | --- |
| **INNER JOIN** | Returns only matching rows between two tables. | Only matching rows |

|  |  |  |
| --- | --- | --- |
| **LEFT JOIN** | Returns all rows from the left table, with matching rows from the right table (or NULL if no match). | All rows from left table + matching rows from right |

|  |  |  |
| --- | --- | --- |
| **RIGHT JOIN** | Returns all rows from the right table, with matching rows from the left table (or NULL if no match). | All rows from right table + matching rows from left |

|  |  |  |
| --- | --- | --- |
| **FULL OUTER JOIN** | Returns all rows from both tables, with NULL for unmatched rows. | All rows from both tables |

|  |  |  |
| --- | --- | --- |
| **CROSS JOIN** | Returns the Cartesian product of both tables (every row from the first table combined with every row of the second). | All combinations of rows |

|  |  |  |
| --- | --- | --- |
| **SELF JOIN** | Joins a table with itself. | Compares rows within the same table |

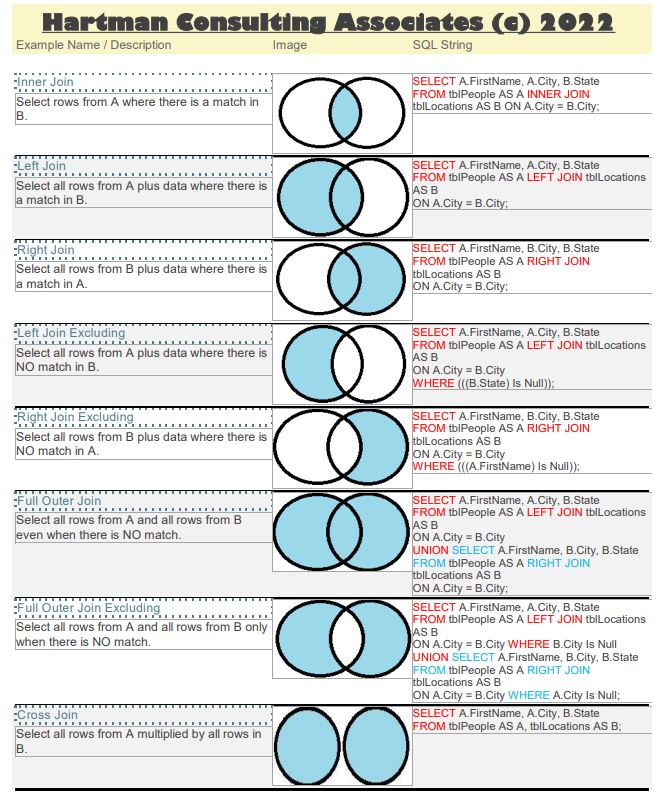
**References:**

mysql in built functions

<https://www.w3schools.com/mysql/mysql_ref_functions.asp>

mysql standard deviation in built functions

<https://dev.mysql.com/doc/refman/9.1/en/built-in-function-reference.html>



<https://www.access-programmers.co.uk/forums/threads/sql-join-types.324662/>

<https://www.google.com/imgres?q=joins%20in%20sql%20venn%20diagram&imgurl=https%3A%2F%2Fapi.datalemur.com%2Fassets%2Fe829c7ea-7a0b-42df-87c5-65a16fef12ff&imgrefurl=https%3A%2F%2Fdatalemur.com%2Fblog%2Fsql-joins-infographics&docid=2OVqCRpEzcxkZM&tbnid=M1gEUOsgQEGdaM&vet=12ahUKEwitpPa1kcKKAxWdzTgGHX_1KBQQM3oECH0QAA..i&w=692&h=928&hcb=2&ved=2ahUKEwitpPa1kcKKAxWdzTgGHX_1KBQQM3oECH0QAA>

<https://www.google.com/imgres?q=joins%20in%20sql%20venn%20diagram&imgurl=https%3A%2F%2Fi.pinimg.com%2Foriginals%2Ff5%2F0d%2Fb4%2Ff50db4ceb625c1185daf4d975298cb50.png&imgrefurl=https%3A%2F%2Fwww.pinterest.com%2Fpin%2F2-stepik--299911656445967738%2F&docid=YAVycRqkwRvIyM&tbnid=t7-kaEdtLll74M&vet=12ahUKEwitpPa1kcKKAxWdzTgGHX_1KBQQM3oECGcQAA..i&w=960&h=612&hcb=2&ved=2ahUKEwitpPa1kcKKAxWdzTgGHX_1KBQQM3oECGcQAA>

<https://www.google.com/imgres?q=joins%20in%20sql%20venn%20diagram&imgurl=https%3A%2F%2Fi.pinimg.com%2Foriginals%2Ff5%2F0d%2Fb4%2Ff50db4ceb625c1185daf4d975298cb50.png&imgrefurl=https%3A%2F%2Fwww.pinterest.com%2Fpin%2F2-stepik--299911656445967738%2F&docid=YAVycRqkwRvIyM&tbnid=t7-kaEdtLll74M&vet=12ahUKEwitpPa1kcKKAxWdzTgGHX_1KBQQM3oECGcQAA..i&w=960&h=612&hcb=2&ved=2ahUKEwitpPa1kcKKAxWdzTgGHX_1KBQQM3oECGcQAA#imgrc=MPsyHVCLOEZrKM&imgdii=nLbMIJL6DhfNeM>