Select Statement :-

→ select statement used to query and fetch data from one or more tables.

- To select a table ⇒ Select *
  * means select everything all rows all columns.
- MySql follows PEMDAS order.
  ↳ Paranthesis, exponent, multiplication, Division, Addition, Subtraction.
- DISTINCT :- To get the unique values from the columns.

Where Clause :-

Used to help filter our records or our rows of data, whereas the select statement is used to help filter or select our actual columns.

[We only return a row that fulfills the condition]

[Default date format in MySQL = yyyy/mm/dd]

```
Select *
From employee_demographics
where gender NOT 'male';
```

↳ is incorrect because 'Not' by itself doesn't work directly with a value like 'male'. It must be used with a comparison or logical operator.

like → where Not gender = 'male';

Like Statement :-

It is super unique because we can look for specific patterns, not necessarily looking for an exact match.

two special characters used with like statement :-

% and _ → a specific value.

↳ means anything

Group by :-

Is used to group rows that have same values in specified columns.
It helps summarizing data. especially when used with aggregate functions.

=> Syntax :-

```
Select column1, Aggregate_function(column2)
from table_name
Group by column1;
```

- The Group by Clause comes after the Where clause and before the Order by clause.

- Common Aggregate functions with Group by :-
  - COUNT(column) → counts entries
  - SUM(column) → Adds values
  - AVG(column) → Average of values
  - MIN(column) → Minimum value.
  - MAX(column) → Maximum value.

Order By :-

Is used to sort the result set of a query by one or more columns, either in ascending (ASC) or descending (DESC) order.

- Comes at the end of the Sql query.

Having Vs Where Clause :-

📘 WHERE vs HAVING in SQL – Notes

Feature	WHERE	HAVING
Use With	Filters **rows** before grouping	Filters **groups** after grouping
Used With	Normal columns	Aggregate functions like `AVG()`, `SUM()`
Position in SQL	Comes **before** `GROUP BY`	Comes **after** `GROUP BY`

- Key rule :-
  - Use WHERE to filter individual rows before grouping.
  - Use HAVING to filter aggregated groups results after GROUP BY.

[Both can be used in the same query].

Limit :

Is used to restrict the number of rows returned by a query. It is commonly used when you only want to see the top results.

Query:

```sql
SELECT name, marks
FROM students
ORDER BY marks DESC
LIMIT 2;
```

→ only top 2 students are shown.

- Limit is usually used with order by to control which top or bottom rows are shown.

```sql
LIMIT offset, count;
```

Where:

- `offset` = number of rows to skip
- `count` = number of rows to return

e.g :-

LIMIT 2,3 ;

⇒ skips first 2 rows, then returns the next 3 rows.

Aliasing in SQL:

is used to assign a temporary name to a table or column.

📊 Column Alias Example:

```sql
SELECT name AS student_name, marks AS score
FROM students;
```

🔷 Output will show column headers as `student_name` and `score` instead of `name` and `marks`.

** JOINS **

* Inner Join :-

Is used to combine rows from two or more tables based on a related column between them.

It only returns rows with matching values in both tables.

- Combines rows from both tables only when there's match.
- If there's no match, that row is excluded from the result.
- Inner join is the default join if you just write JOIN.

✅ **Query:**

```sql
SELECT e.emp_name, d.dept_name
FROM employees AS e
INNER JOIN departments AS d
ON e.dept_id = d.dept_id;
```

← syntax

* Use case of inner join.

→ want only related data.

→ Need to work with foreign keys.

→ Combining data from multiple tables that share a common column.

using aliasing in joins →

Aliases with JOIN:

Aliases make queries **shorter and more readable.**

```sql
FROM employees AS e
JOIN departments AS d
ON e.dept_id = d.dept_id;
```

* Outer Joins :-

Outer joins return matched rows as well as unmatched rows from one or both tables.

There are two types :-

→ Left Join
→ Right Join

◆ Left Join :-

- Returns all rows from the left table
- If there's no match in the right table, NULLs are shown for the right table columns.

◆ Right Join :-

- Returns all rows from the right table.
- If there's no match in the left table, NULLs are shown for the left table columns.

* Self Join :-

- self join is a regular join where a table is joined with itself.

[must use aliases to differentiate between the two instances of the same table]

Joining Multiple tables :-

You can join more than two tables using multiple join clauses.

each join links two tables.

✅ **Syntax:**

```sql
SELECT t1.column, t2.column, t3.column
FROM table1 AS t1
JOIN table2 AS t2 ON t1.col = t2.col
JOIN table3 AS t3 ON t2.col = t3.col;
```

Summary

✅ 🟦 SQL JOINs Summary Table (Updated)

Join Type	Definition	Returns	Use When...
`INNER JOIN`	Combines rows with matching values in both tables	Only matching rows	You want **only related** records from both tables
`LEFT JOIN`	All rows from the left table + matched rows from the right	All left table rows, right side NULL if no match	You want **all left-side** data even if there's no match
`RIGHT JOIN`	All rows from the right table + matched rows from the left	All right table rows, left side NULL if no match	You want **all right-side** data even if there's no match
`SELF JOIN`	Table joined with itself	Rows from the same table related to each other	You want to compare rows **within the same table** (e.g., employee hierarchy)
Multi-table Join	Joining more than two tables using multiple `JOIN` clauses	Depends on type of joins used	Data is **spread across 3 or more related tables**

* UNION :

Is used to combine the results of two or more SELECT queries into a single result set.

✅ **Basic Syntax:**

```sql
SELECT column1, column2, ...
FROM table1
UNION
SELECT column1, column2, ...
FROM table2;
```

=> Key rules :-

- The number & order of columns in all select statements must be the same.
- The data types of corresponding columns must be compatible.

- By default, UNION removes duplicates. (like DISTINCT)
- To include duplicates, use UNION ALL.

=> When to use.

- want to combine rows from multiple tables or queries.
- Need to merge data from different time periods.
- Want to create a single list from different sources.

String Functions

	Function	Use Case	Example	Output
0	LENGTH()	Length in bytes	LENGTH('MySQL')	5
1	CHAR_LENGTH()	Length in characters	CHAR_LENGTH('MySQL')	5
2	UPPER()	Convert to uppercase	UPPER('hello')	HELLO
3	LOWER()	Convert to lowercase	LOWER('HELLO')	hello
4	CONCAT()	Combine strings	CONCAT('My', 'SQL')	MySQL
5	CONCAT_WS()	Combine with separator	CONCAT_WS('-', '2025', '06', '09')	2025-06-09
6	SUBSTRING()	Extract part of string	SUBSTRING('Database', 1, 4)	Data
7	LEFT()	Get left N chars	LEFT('Database', 4)	Data
8	RIGHT()	Get right N chars	RIGHT('Database', 4)	base
9	TRIM()	Trim both sides	TRIM(' Hello ')	Hello
10	LTRIM()	Trim left	LTRIM(' Hello')	Hello
11	RTRIM()	Trim right	RTRIM('Hello ')	Hello
12	REPLACE()	Replace substring	REPLACE('data science', 'data', 'AI')	AI science
13	INSTR()	Position of substring	INSTR('data', 'a')	2
14	LOCATE()	Locate with start pos	LOCATE('a', 'banana', 3)	5
15	POSITION()	Find position	POSITION('at' IN 'data')	3
16	REVERSE()	Reverse string	REVERSE('MySQL')	LQSyM
17	LPAD()	Left pad string	LPAD('abc', 5, '*')	**abc
18	RPAD()	Right pad string	RPAD('abc', 5, '*')	abc**
19	FORMAT()	Format number	FORMAT(1234567.89, 2)	1,234,567.89
20	ASCII()	ASCII of char	ASCII('A')	65
21	CHAR()	Char from ASCII	CHAR(65)	A
22	ELT()	Get N-th item	ELT(2, 'red', 'green', 'blue')	green
23	FIELD()	Find item position	FIELD('green', 'red', 'green', 'blue')	2
24	FIND_IN_SET()	Find in comma list	FIND_IN_SET('b', 'a,b,c')	2
25	MAKE_SET()	Create set from bits	MAKE_SET(5, 'a', 'b', 'c', 'd')	a,c
26	REPEAT()	Repeat string	REPEAT('abc', 3)	abcabcabc
27	SPACE()	Add spaces	CONCAT('A', SPACE(3), 'B')	A B
28	STRCMP()	Compare strings	STRCMP('abc', 'ABC')	1

* CASE Statement.

The case statement in SQL is like an if-else ladder. It lets you perform conditional logic in queries, especially in Select, update, order by, and where clause.

🧠 Basic Syntax

```sql
SELECT
  column1,
  CASE
    WHEN condition1 THEN result1
    WHEN condition2 THEN result2
    ...
    ELSE default_result
  END AS alias_name
FROM table_name;
```

✅ Tips

Tip	Description
`CASE` returns the first `WHEN` that matches	So ordering your conditions matters
Always include `ELSE`	To catch any unmatched cases
Can be used in `SELECT`, `UPDATE`, `ORDER BY`, `GROUP BY`, and `HAVING`	Very flexible
Can return any data type	As long as all `THEN` and `ELSE` return compatible types

* Subqueries IN SQL :-

A subquery is a SQL query nested inside another query. It's used to return data that the outer query can use.

[its a query inside a query].

```sql
SELECT column
FROM table
WHERE column IN (SELECT column FROM another_table WHERE condition);
```

Why use ???

(syntax)

- To filter results using values from another query.
- To avoid complex joins.
- To compare an item to aggregated data (like min, max).
- To update or delete based on conditions from another table.

⚠️ Subquery vs JOIN

Feature	Subquery	JOIN
Performance	Slower for large data sets	Faster with indexing
Readability	Simpler for nested logic	Better for multiple table relations
Use Case	Filter or compute scalar/table data	Combine data across tables

* Important about aliasing :-

In SqL, column aliases are available only in the SELECT clause output, not always in the ORDER BY (especially in subqueries or when using UNION, LIMIT etc)

✅ Key Rule for Notes:

> Column aliases are reliable **only within the SELECT output scope**. When using them in `ORDER BY`, `GROUP BY`, or outer queries, **use the full qualified reference** (like `table_or_alias.column_name`) especially in subqueries or complex constructs like `UNION` or `LIMIT`.

* Window Function :-

Functions that perform calculations across a set of rows that are related to the current row, without collapsing the result into a single row (unlike GROUP BY).

They are used with the OVER() clause and are very useful when you want to:
- Rank rows.
- Calculate running totals.
- Get previous / next row values.
- Find moving averages.

→ They don't change the number of rows in the output.

🔍 Basic Syntax:

```sql
SELECT column1,
       window_function() OVER (
           PARTITION BY column2
           ORDER BY column3
       ) AS alias
FROM table_name;
```

- Window_function() - like RANK(), SUM() etc.
- OVER() - tells MySQL to treat the function as a window function.
- Partition By - divides rows into groups (optional).
- ORDER BY - orders rows within each partition (optional but common).

⇒ Partition By Vs Order By.

- Partition By : Groups rows without collapsing them like GROUP BY.
- Order By : (within Over) orders rows inside each partition.

📌 Important Notes for Coding

Here's a cheat sheet for your notes:

```text
🔹 WINDOW FUNCTION = Function + OVER()
🔹 OVER() = Defines the window frame
🔹 PARTITION BY = Like GROUP BY, resets function per group
🔹 ORDER BY = Sorts rows within partition
🔹 ROW_NUMBER(), RANK(), DENSE_RANK() – Ranking
🔹 LAG(), LEAD() – Access previous/next row
🔹 SUM(), AVG(), COUNT() – Can be used as window functions
🔹 Doesn't reduce rows like GROUP BY does
🔹 Works great for running totals, comparisons, rankings, trends
```

* CTE - Common Table Expression :-

A CTE is a temporary result set (like a virtual table) that you can reference within a SELECT, INSERT, UPDATE, or DELETE query.

[Lives only for the duration of that SQL query].

📚 Syntax of CTE:

```sql
WITH cte_name AS (
    SELECT ... -- your subquery
)
SELECT * FROM cte_name;
```

← WITH starts the CTE.

AS (....) the query that creates the CTE.

Use CTE for:

- Better readability.
- Avoiding repeating subqueries.
- Complex joins and aggregations.
- Recursive queries.

🔹 You can chain multiple CTEs:

```
WITH first_cte AS (...),
     second_cte AS (...)
SELECT ... FROM second_cte;
```

← Syntax for multiple CTEs.

Recursive CTE. →

🔹 Recursive CTE:

```
WITH RECURSIVE cte_name AS (
    base_case
    UNION ALL
    recursive_part
)
SELECT * FROM cte_name;
```

* Stored Procedures.

- A chunk of SQL (plus flow-control logic) that is compiled once and stored inside the database.
- You call it with CALL procedure_name(...) instead of sending the whole script each time.
- Lives in the data dictionary, survives restarts, and can be granted / revoked like any other object.

```
DELIMITER //
CREATE PROCEDURE demo()
BEGIN
  -- semicolons allowed here
  SELECT NOW();
END //
DELIMITER ;
```

← Syntax.

[Don't forget the delimiter, so embedded ";" don't prematurely end the script].

=> Parameter modes

Mode	Direction	Notes
`IN`	caller ➜ proc	Default; read-only inside proc.
`OUT`	proc ➜ caller	Assign with `SET param = …`.
`INOUT`	both ways	Read initial value, then overwrite.

=> Procedure Vs Function.

- Procedure: CALL, can return 0-N result sets & modify data.
- Function: SELECT func(), must return one scalar value, side-effects disabled by default.