DATABASE

Data is a collection of raw, unorganized, unprocessed facts and details.

- Text
- Media
- Figures
- Symbols
- Observations
- Descriptions of things etc.

➔ Data does not carry any specific purpose and has no significance by itself
➔ Raw data alone is insufficient for decision making
➔ Data is information that can be translated into a form that is efficient for movement and processing.

Example :

- Test Score of Students
- Temperature Readings
- Match Scroe
- Name
- Age

What is Database?

Database is a collection of data stored in some organized fashion, so that it can be easily accessed and managed.

➔ Database is a container (usually a file or set of files) to store organized data.
➔ We often use the term Database to refer to the database software we are using.
    - MySQL
    - ORACLE
    - MongoDB
    - SQL Server
➔ Database software is actually called the Database Management System (or DBMS)
➔ The database is the container created and manipulated via the DBMS.

Example :

- ➔ School Registry
  - All the details of the students are entered in a single file. You get the details regarding the students in the file
- ➔ Facebook
  - It needs to store, manipulate , and present data related to users, their friends, users activities, messages, advertisements, and lot more.

- ➔ Amazon/ Flipkart/ Mytra
  - They store data related to customer, product, orders, payment, carts and order tracking details etc...

Types of Database

- Relational Databases
  - Oracle, MySQL, Microsoft SQL Server, PostgreSQL and IBM Db2
- NoSQL databases
  - Apache Cassandra, MongoDB, CouchDB, and CouchBase
- Cloud databases
  - Microsoft Azure SQL Database, Amazon Relational Database Service
- Document Databases
  - MongoDB, Amazon DocumentDB, Apache CouchDB
- Wide column databases
  - BigTable, Apache Cassandra and Scylla
- Object-oriented databases
  - Wakanda, ObjectStore
- Key-Value databases
  - Amazon DynamoDB, Redis
- Hierarchical databases
  - IBM Information Management System (IMS), Windows Registry
- Graph databases
  - Datastax Enterprise Graph, Neo4J
- Columnar databases
  - Google BigQuery, Cassandra, HBase, MariaDB, Azure SQL Data Warehouse

Important Terms of Database :

- Database
- Tables
- Column
- Row
- Primary Key
- Datatype

➔ Database
  - A container (usually a file or set of files) to store organized data.

➔ Relational Database
  - In relational database data is stored in one or more tables (or "relations") of columns and rows (or "tuples).

➔ TABLE
  - A structured list of data of specific type.
  - Tables are also called "Relations".

➔ Column
  - A single field in a table. All tables are made of one or more columns.
  - Think table as a Excel sheet grid so the vertical columns in the grid are the table columns.
  - Constraints -> Primary Key, Foreign Key , Not null...

➔ Row
  - Data in a table is stored in rows, each record stored in its own row.
  - Think table as a Excel sheet grid so the horizontal rows are the table rows
  - Row are also knows as TUPLE.

➔ Primary Key
  - A column (or set of columns) whose values uniquely identify every row in a table.

➔ DATATYPE
  - A type of allowed data. Every table column has an associated datatype that restricts (or allows) specific data in that column.

What is SQL ?

- SQL(Pronounced as the letters S-Q-L or sequel) is an abbreviation for Structured Query Language
- It used to communicate with the Relational Database.
- SQL is made up of very few words.

- SQL provide you with a simple and efficient way to read and write the data from a database.
- All the RDBMS like MySQL, Oracle, MS Access , SQL Server and Informix use SQL as their standard database language.
- Extensions provide additional functionality to the language

Advantages of SQL :

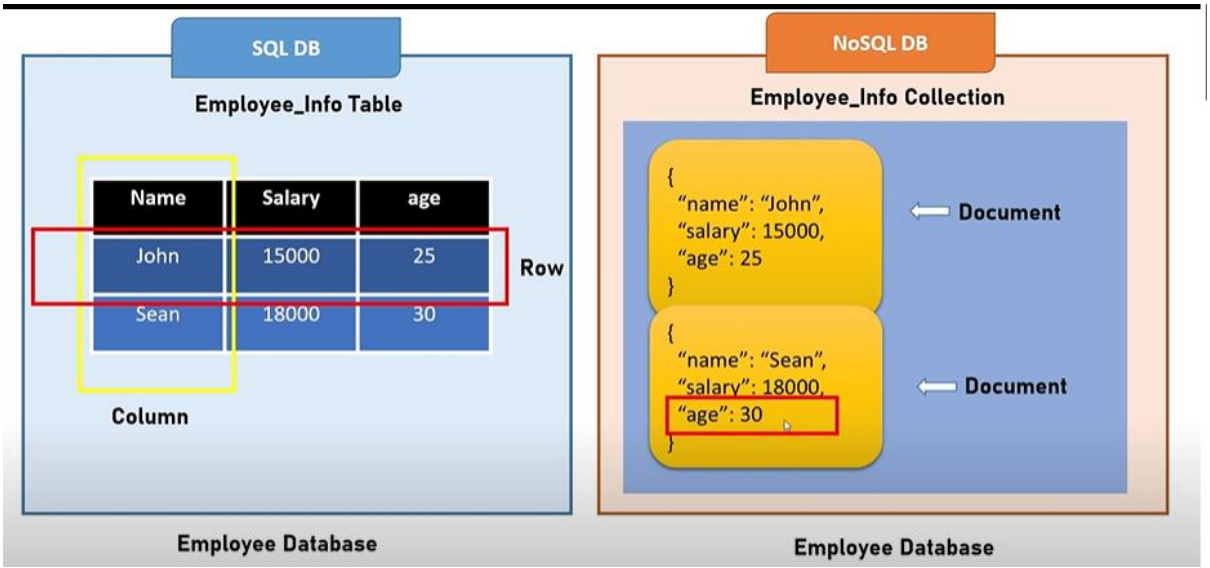
- SQL is not a language used by specific database. Almost every major DBMS supports SQL.
- SQL is easy to learn and don't need any coding skills
- SQL is also portable
- SQL is standardized language
- SQL is a very powerful language which can perform very complex and sophisticated database operations.

NOTE :

- SQL is not case sensitive. Generally, keywords of SQL are written in uppercase.
- Statements of SQL are dependent on text lines. We can use a single SQL statement on one or multiple text line.
- SQL depends on tuple relational calculus and relational algebra.

Differences Between SQL vs NoSQL :

	SQL Database	NoSQL Database
Defination	SQL databases are primarily called RDBMS or Relational Databases	NoSQL Databases are primarily called as Non-Relational or distributed databases
Query Language	Structured Query Language (sql)	No declarative query language
Type	SQL databases are table based databases	NoSQL datbases can be documented based, key-value pairs, graph databases
Examples	MySQL, Oracle, Postgres, and MS-SQL.	MongoDB, Redis, Neo4j, Cassandra, Hbase.
Schema	SQL databases have a predefined schema.	NoSQL databases use dynamic schema for unstructured data.



	SQL Database	NoSQL Database
ACID vs CAP	ACID (Atomicity, Consistency, Isolation, and Durability) is a standard for RDBMS	CAP (Consistent , Available & Partition tolerance)
Best suited for	An ideal choice for the complex query intensive environment.	It is not good fit complex queries.
Importance	It should be used when data validity is super important.	Use when it's more important to have fast data than correct data.

SQL Commands :

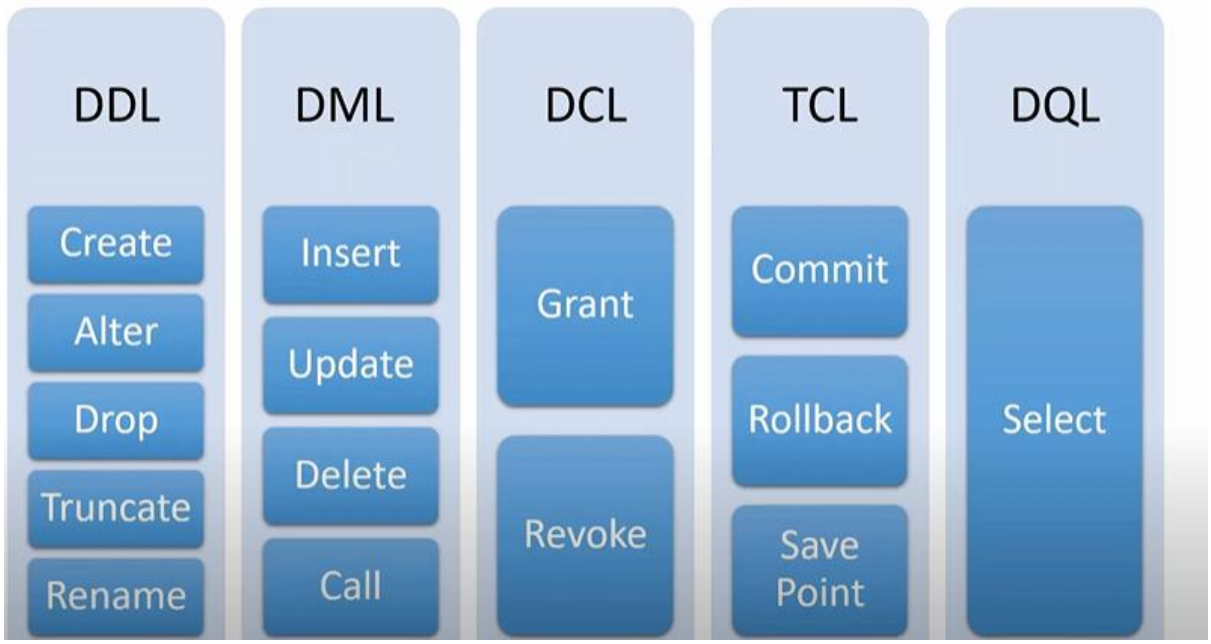
➔ SQL commands are used to communicate with a database.
➔ SQL commands are the instructions.
➔ It can perform different tasks, functions, and queries with data.

SQL commands can perform various tasks like –

- Create table
- Add data to tables
- Update data of a table
- Drop Table

- Modify Table
- Set permission for users.

These SQL commands are divided into 5 categories those are

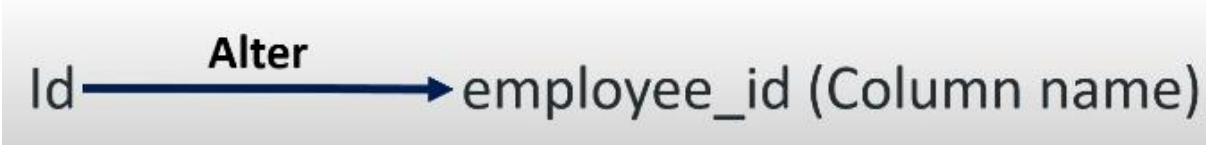


DDL (Data Definition Language)

➔ DDL commands can be used to define the database schema.
➔ It is a set of SQL commands used to create, modify and delete database structures but not data.
➔ These commands are normally not used by a general user, who should be accessing the database via an application
➔ These commands are auto-commited.

⇨ CREATE
  - This command is used to create the database of its objects (like table, index, function, views, store procedure, and triggers).

⇨ ALTER
  - This is used to alter the structure of the database.



- ⇨ DROP
  - This command is used to delete objects from the database.
- ⇨ TRUNCATE
  - This is used to remove all records from a table.
  - This operation drop and recreate the table.
- ⇨ RENAME
  - This is used to rename an object existing in the database.

DML (Data Manipulation Language)

- ➔ DML commands deals with the manipulation of data present in the database.
- ➔ DML commands are not auto-commited.

- ⇨ INSERT
  - It is used to insert data into a table
- ⇨ UPDATE
  - It is used to update existing data within a table.
- ⇨ DELETE
  - It is used to delete records from a database table.
- ⇨ CALL
  - It is used to invoke a stored procedure.

DCL (Data Control Language)

- ➔ DCL commands are mainly deal with the rights, permissions, are other controls of the database system.
- ➔ DCL commands are used to grant and take back authority form any database user.

- ⇨ GRANT
  - This command gives users access privileges to the database.
- ⇨ REVOKE
  - This command withdraws the user's access privileges given by using the GRANT Command.

TCL (Transaction Control Language)

- ➔ TCL Commands deal with the transaction within the database.
- ➔ Database, and all served as a single logical unit of work – taking place wholly or not at all.
- ➔ The Transaction Control Language is used to maintain the integrity and consistency of the data stored in the database.
- ➔ These commands can only use with DML commands like INSERT, DELETE and UPDATE only.

- ⇨ COMMIT
  - o It is used to Commits a Transaction
- ⇨ ROLLBACK
  - o It helps Rollbacks a transaction in case of any error occurs.
- ⇨ SAVEPOINT
  - o Sets a savepoint within a transaction.

DQL (Data Query Language)

- ➔ DQL is used to fetch or retrieve the data from the database.
- ➔ DQL statements are used for performing queries on the data within schema objects.
- ➔ This command allows getting the data out of the database to perform operations with it.

- ⇨ SELECT
  - o It is used to retrieve data from the database.

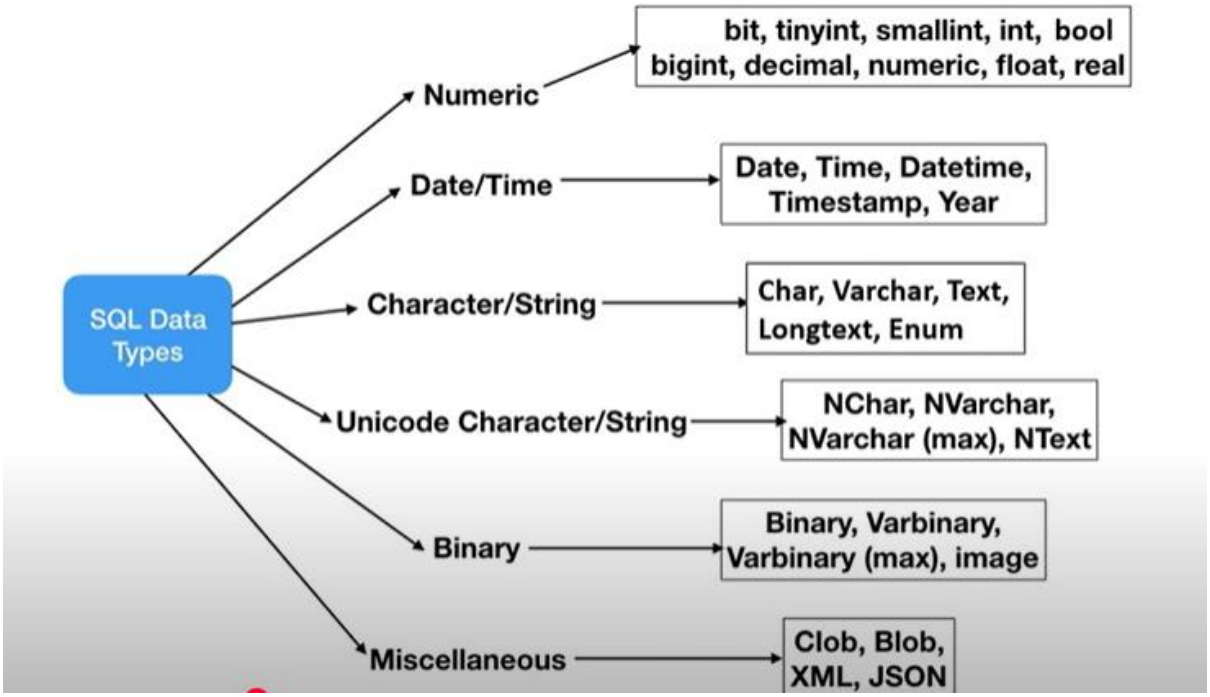
DATA TYPES IN SQL :

- ➔ Datatypes are essentially rules that define what data value the column can hold(integer, character, data and time, binary, currency and so on) and how that data is actually stored.
- ➔ Each column in a database table is required to have a name and a data type.

Datatypes are used for several reasons :

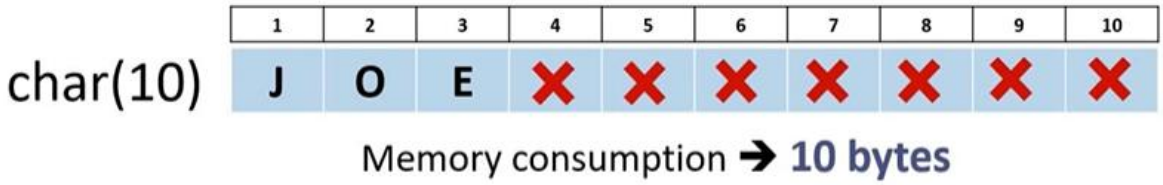
- ➔ Datatypes enable you to restrict the type of data that can be stored in a column

➔ Datatypes allow for more efficient storage, internally.
➔ Datatypes allow for alternative sorting orders.

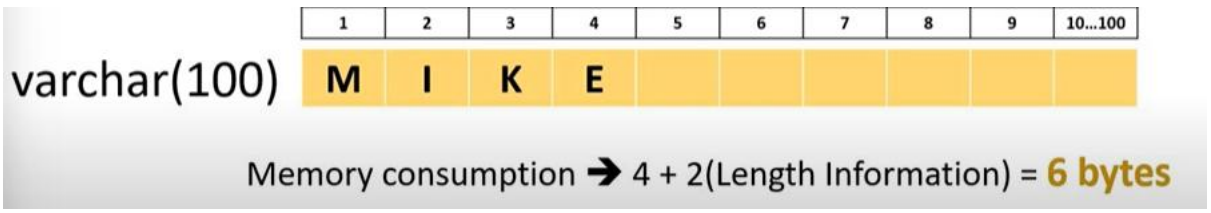


Character / String :

⇨ CHAR(size) –
- It is used to specify a fixed length string that can contain numbers, letters, and special characters. Its size can be 0 to 255 characters. Default is 1.



⇨ VARCHAR(size) –
- It is used to specify a variable length string that can contain numbers, letters, and special characters. Its size can be from 0 to 65535 characters.



- ⇨ String values must always be surrounded by single quotes.
    - Example – ‘A’, ‘Book’, ‘7542’, ‘abc@gmail.com’ etc…

Create :

Create (DDL) command is used to create a database or table in the database with the structure specified by the user.

This structure includes the number of columns to be present in the table and name of columns, data type of the column , size of data etc.

```
CREATE DATABASE SchoolDb;

USE SchoolDb;

CREATE TABLE Students (
    StudentID INT PRIMARY KEY,
    StudentName VARCHAR(50),
    Age INT,
    EnrollmentDate DATE
);

CREATE TABLE Enrollments (
    EnrollmentID INT PRIMARY KEY,
    StudentID INT,
    CourseName VARCHAR(100),
    CourseCode VARCHAR(20),
    InstructorName VARCHAR(50),
    EnrollmentDate DATE,
);
```

NOTE :

- In one Database we cannot create two tables with the same name.
- Don’t use DBMS specific keyword as Table or Database name.
- The name and definition of the table columns separated by commas,
- Use lowercase for table and column names.
- To separate words in Table name use CamelCase or an underscore(“_”)
    - For Example start_date

- All the column name within a table must be unique
- A table must have a Primary Key.

📘 What does EXEC sp_help 'Students'; do?

This command retrieves metadata about the Students table. Here's what it shows:

1. Column Information

- Column names
- Data types
- Lengths
- Nullability (whether NULLs are allowed)

2. Identity Columns

- If any column is an identity (auto-increment), it shows the seed and increment values.

3. Indexes

- Lists all indexes on the table, including primary keys and unique constraints.

4. Constraints

- Foreign keys, check constraints, and default values.

5. Table Properties

- If applicable, it shows whether the table is a system table, user table, or view.

🛠️ Syntax Breakdown

```
sql

EXEC sp_help 'Students';
```

- EXEC: Executes a stored procedure.
- sp_help: The system procedure that fetches object details.
- 'Students': The name of the table you're inspecting.

By Copying selected rows form another table, we can create a new table also

Example :

⇨ CREATE TABLE emp2 AS SELECT * FROM employee WHERE id = 102;

INSERT Command :

- Insert is a DML Command
- Insert Command is used to add one or more rows of data to the database table with specified column values.

Syntax :
INSERT INTO *table_name(column1, column2, column3..)*
VALUES*(value1, value2, value3...);*

➔ If you are adding values for all the columns of the table, you do not need to specify the column names.

Syntax :
INSERT INTO *table_name*
VALUES*(value1, value2, value3...);*

➔ The Order of the values is in the same order as the columns in the table.

```
INSERT INTO Students VALUES (1, 'Kasi', 25, '2025-07-15');
INSERT INTO Students VALUES (2, 'Surya', 20, '2025-07-15');
INSERT INTO Students VALUES (3, 'Ravi Kumar', 18, '2025-07-15');
INSERT INTO Students VALUES (4, 'Kiran', 22, '2025-07-15');
INSERT INTO Students VALUES (5, 'Rohin', 25, '2025-07-15');
```

```
INSERT INTO Enrollments VALUES
(101, 1, 'Introduction to Computer Science', 'CS101', 'Dr. Meera Reddy', '2025-08-01'),
(102, 2, 'Data Structures and Algorithms', 'CS201', 'Prof. Arjun Verma', '2025-08-03'),
(103, 3, 'Database Management Systems', 'CS301', 'Dr. Kavitha Rao', '2025-08-05'),
(104, 1, 'Web Development Fundamentals', 'WD101', 'Ms. Sneha Kulkarni', '2025-08-07'),
(105, 4, 'Machine Learning Basics', 'ML101', 'Dr. Rajesh Nair', '2025-08-10');
```

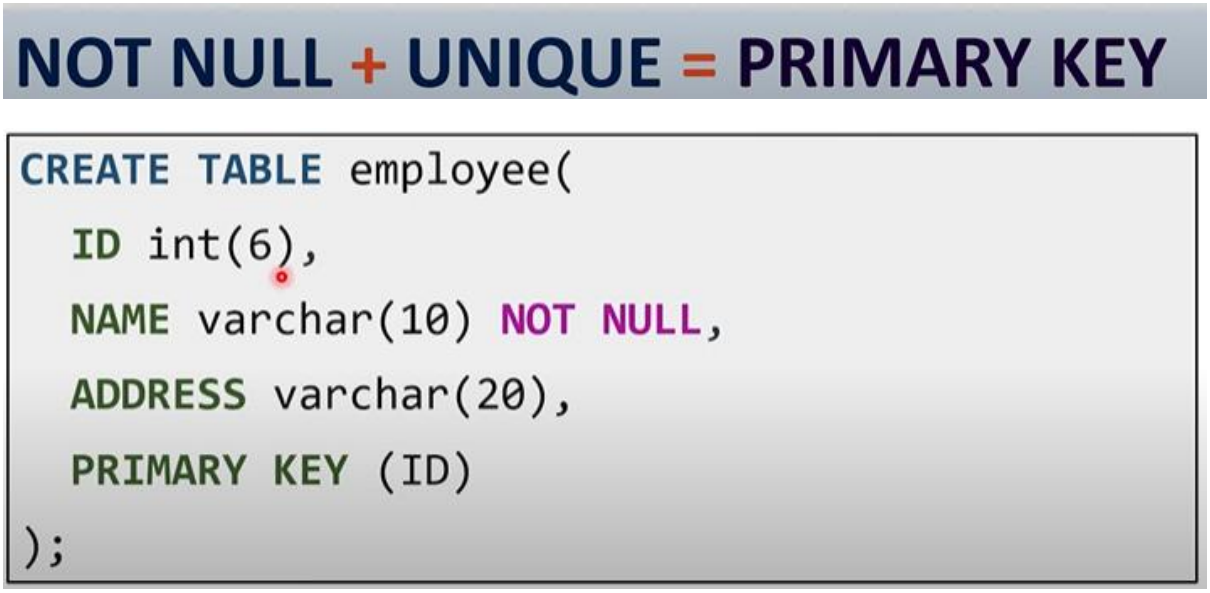
Constraints In the SQL :

What are SQL Constraints?

- We use SQL Constraints to specify the rules for the data in a table.
- Constraints are used to limit which type of data must be stored in the database.
- SQL Constraints increase the accuracy and reliability of the data stored in the database.
- Constraints make sure that there is no violation in terms of transaction of the data.

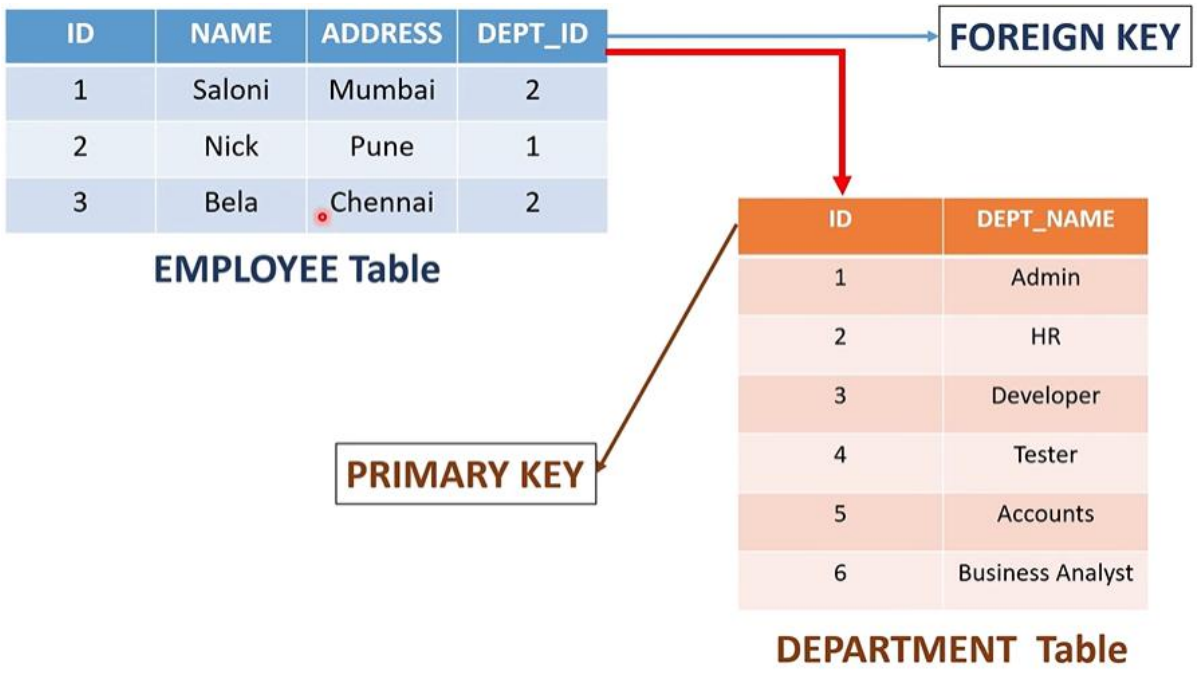
Primary Key?

- A primary key is a field which can uniquely identify each row in a table.
- Primary keys must contain UNIQUE values, and cannot contain NULL values.
- A table can have only ONE primary key and this primary key can consist of single or multiple columns.
- We can use this constraint on the CREATE and ALTER table command.



Foreign Key?

- A foreign key is a field (or collection of fields) points to the PRIMARY KEY of another table.
- Table with the foreign key is called the **child table** and the table with the primary key is called the referenced or **parent table**.
- The foreign key constraint is used to prevent actions that would destroy links between tables.
- We can use this constraint on the CREATE and ALTER table command.




```
CREATE TABLE department(
  ID int(6) PRIMARY KEY,
  DEPT_NAME varchar(10) NOT NULL
);
```

```
CREATE TABLE employee(
  ID int(6) NOT NULL,
  NAME varchar(10) NOT NULL,
  ADDRESS varchar(20),
  DEPT_ID int(6),
  FOREIGN KEY (DEPT_ID) REFERENCES department(ID)
);
```

Not NULL?

- The Not null Constraint makes sure that a column cannot have a NULL value.
- We can use the NOT NULL constraint either while creating the table in the database or while modifying it.
- We can have more than one NOT NULL columns in a table.

```
CREATE TABLE employee(
  ID int(6) NOT NULL,
  NAME varchar(10) NOT NULL,
  ADDRESS varchar(20)
);
```

Unique ?

- The UNIQUE constraint makes sure that all values in a column must be unique.
- This constraint helps to uniquely identify each row in the table.
- We can have more than one UNIQUE columns in a table.
- We can use this constraint on the CREATE and ALTER table command.

```
CREATE TABLE employee(
  ID int(6) UNIQUE,
  NAME varchar(10) NOT NULL,
  ADDRESS varchar(20)
);
```

Check?

- The CHECK constraint makes sure that all values in a column satisfy a specific condition.

- We can use this constraint on the CREATE and ALTER table command.

```
CREATE TABLE employee(
  ID int(6),
  NAME varchar(10) CHECK(NAME != 'Saloni'),
  ADDRESS varchar(20)
);
```

Default?

- The DEFAULT constraint is used to set default values for a column when no value specified.
- The default value will be added to all new records, if no other value is specified.
- We can use this constraint on the CREATE and ALTER table command.

```
CREATE TABLE employee(
  ID int(6),
  NAME varchar(30) DEFAULT 'NEW USER',
  ADDRESS varchar(20)
);
```

SELECT and SELECT DISTINCT

SELECT Statement

- SELECT is a DQL or a limited form of DML Command.

- The SELECT statement is SQL is used to fetch data from one or more tables.
- We can retrieve records of all fields(columns) or specified field(columns).
- With SELECT , we can specify conditions using WHERE Clause.

FROM Clause

- FROM Clause is used to list the tables and any joins required for the query in SQL.

```
SELECT * FROM Students;

SELECT * FROM Enrollments;
```

➔ Specify particular field name instead of asterisk(*) to get specified fields.

```
SELECT StudentName, AGE FROM Students;
SELECT CourseCode, InstructorName FROM Enrollments;
```

SELECT DISTINCT Statement

➔ **DISTINCT** clause is used to remove duplicate records form the table and fetch only the unique records.
➔ The DISTINCT clause is only used with the SELECT statement.

SQL Aliases

➔ Aliases are used to give a table, or a column in a table, a temporary name.
➔ Aliases are often used to make column names more readable.
➔ These are used when we combine columns or when we use functions in the query
➔ An alias only exists for the duration of that query.
➔ An alias is created with the AS keyword (optional)

Example :

```
SELECT  id+salary AS addition
FROM employee;
```

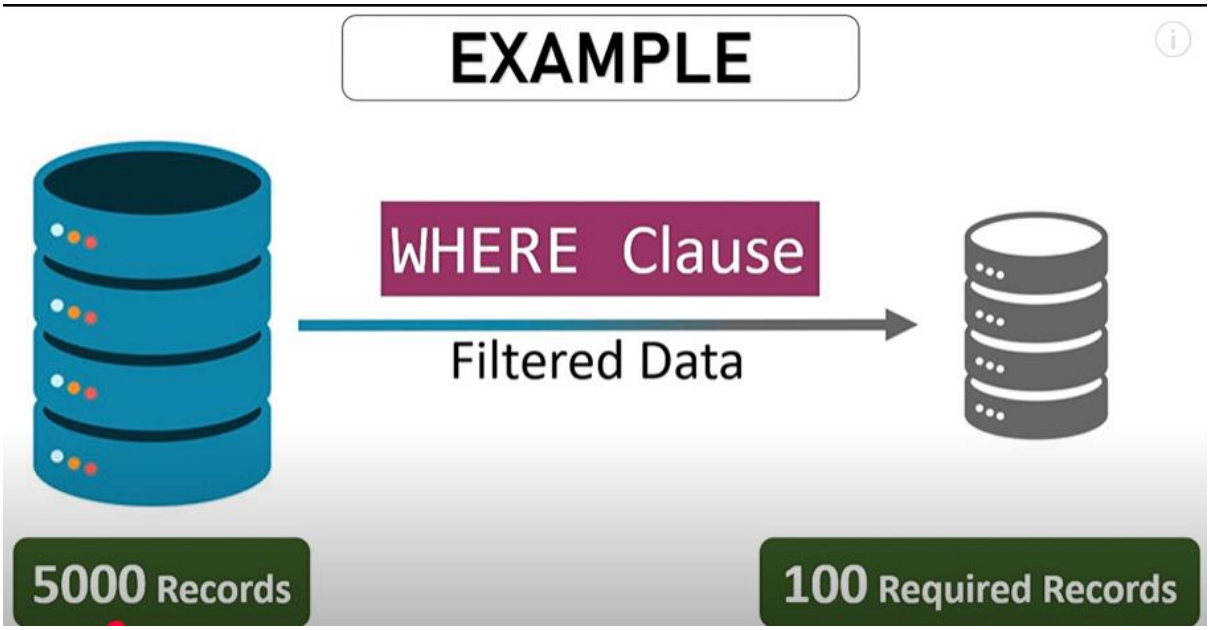
➔ Use Expression or functions instead of asterisk(*) or specific column name to get processed or calculated column results.

Example :

```
SELECT  CONCAT(name,'-',id) AS employee_code
FROM employee;
```

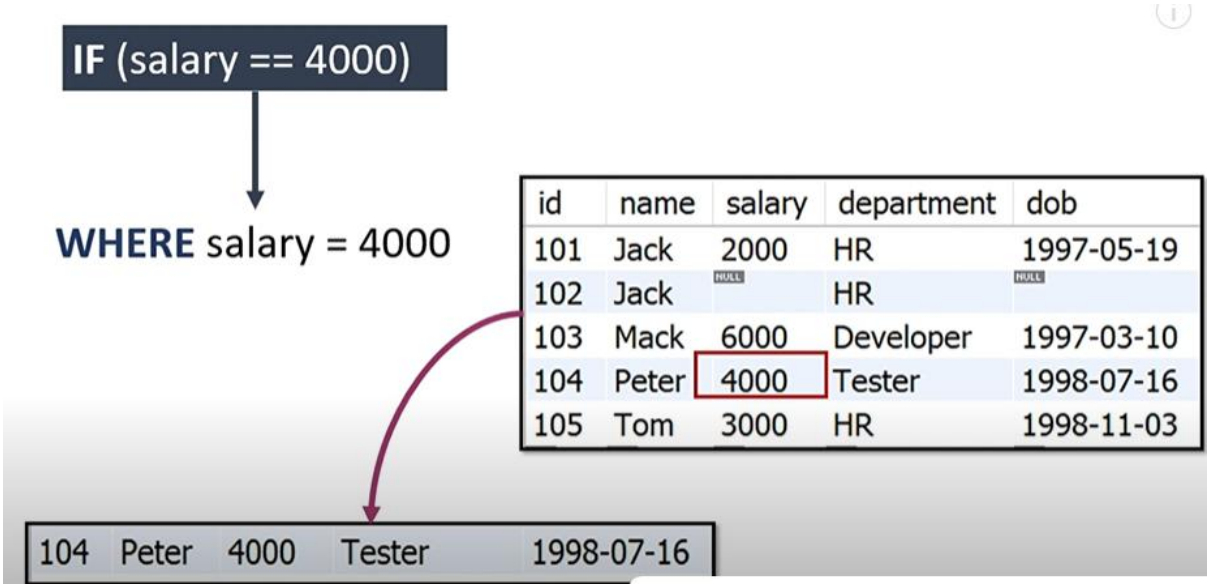
employee_code
Jack-101
Jack-102
Mack-103
Peter-104
Tom-105

WHERE Clause



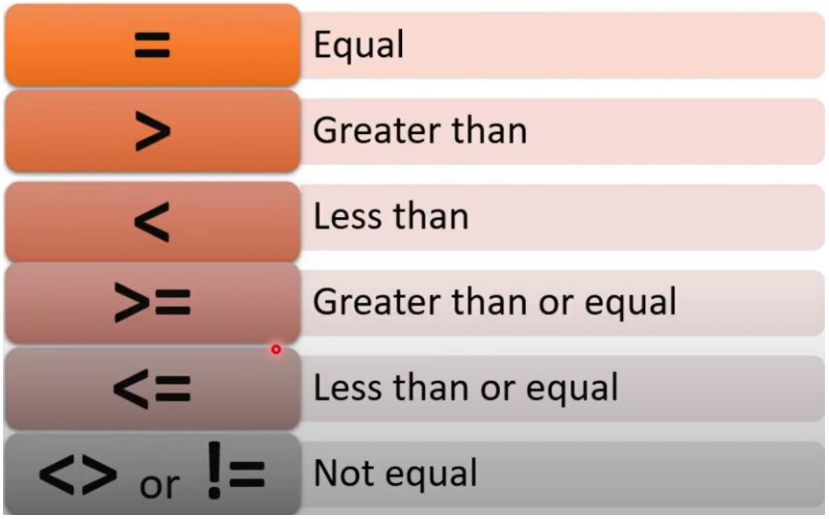
- The WHERE clause is used to filter records.

- It is used to extract only those records that fulfil a specified condition
- The WHERE clause works like an if condition in any programming language.
- WHERE clause is used with INSERT , UPDATE, DELETE, etc.!



SQL Operators

- Arithmetic Operators
- Bitwise Operators
- Comparison Operators
- Compound Operators
- Logical Operators



```
SELECT column1,column2,...
FROM table_name
WHERE [condition];
```

```
SELECT * FROM Students WHERE AGE>20;
```

Results | Messages

	StudentID	StudentName	AGE	EnrollmentDate
1	1	Kasi	25	2025-06-15
2	4	Kiran	22	2025-07-15
3	5	Rohin	25	2025-09-19

```
SELECT * FROM Students WHERE StudentName = 'Kasi';
```

Results | Messages

	StudentID	StudentName	AGE	EnrollmentDate
1	1	Kasi	25	2025-06-15

AND , OR & NOT Operators

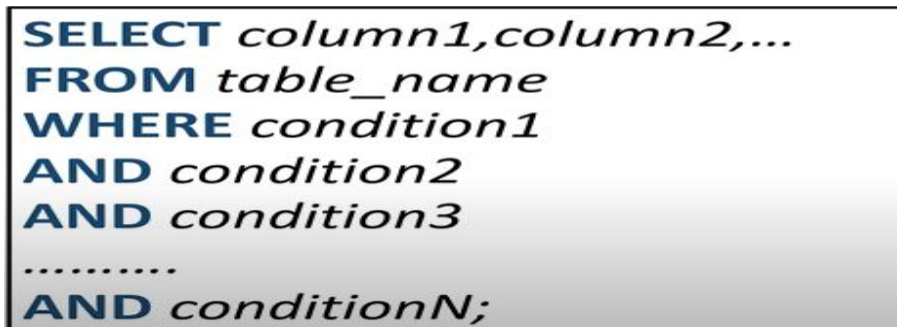
- The WHERE clause can be combined with AND, OR, and NOT operators
- AND and OR operators are used to filter records based on multiple condition.

AND Operator

➔ To filter by more than one column, you can use the AND operator to append conditions to your WHERE clause.

➔ The AND operator displays a record if all the conditions separated by AND are TRUE.

A	B	A AND B
TRUE	TRUE	TRUE
TRUE	FALSE	FALSE
FALSE	TRUE	FALSE
FALSE	FALSE	FALSE




```
SELECT StudentID, StudentName FROM Students WHERE StudentName = 'Kasi' AND AGE = 25;
```

	StudentID	StudentName
1	1	Kasi

OR Operator

➔ The OR operator is exactly the opposite of AND operator.

➔ The OR operator displays a record if any of the conditions separated by OR is TRUE.

A	B	A OR B
TRUE	TRUE	TRUE
TRUE	FALSE	TRUE
FALSE	TRUE	TRUE
FALSE	FALSE	FALSE

Syntax :

```
SELECT column1,column2,...
FROM table_name
WHERE condition1
OR condition2
OR condition3
.....
OR conditionN;
```

NOT Operator

➔ The NOT operator displays a record if the condition(s) is NOT TRUE.

- TRUE -> FALSE
- FALSE -> TRUE

A	NOT A
TRUE	FALSE
FALSE	TRUE

Syntax :

```
SELECT column1,column2,...
FROM table_name
WHERE NOT condition;
```

Example :

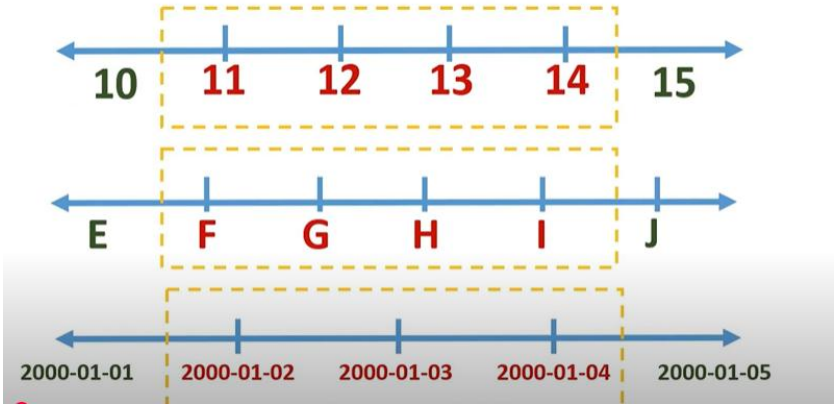
```
SELECT *
FROM employee
WHERE NOT salary <= 3000;
```

BETWEEN and IN Operator

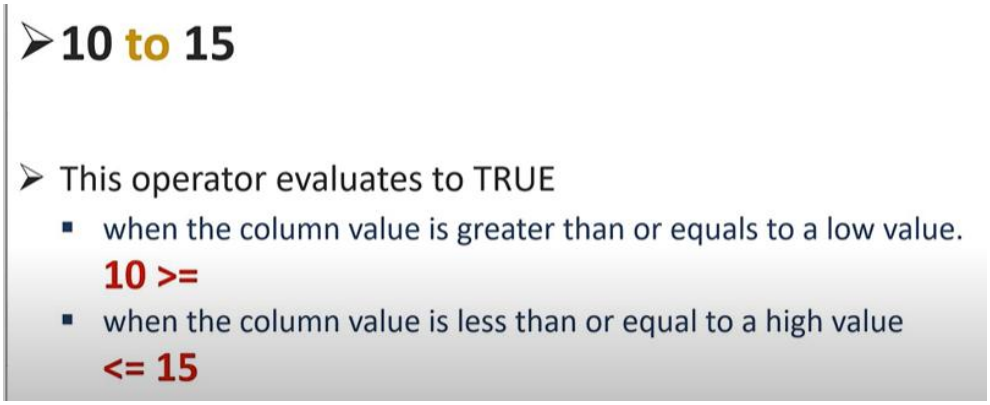
BETWEEN Operator

➔ BETWEEN operator used with WHERE Clause

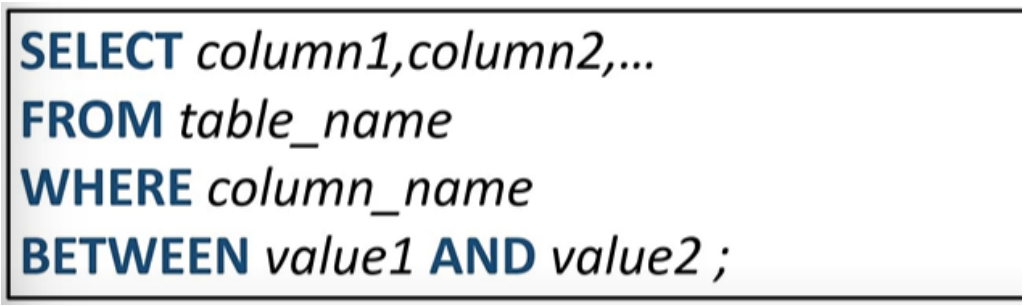
➔ The BETWEEN Operator selects values within a given range. The values can be numbers, text, or dates.



➔ The BETWEEN Operator is inclusive : begin and end values are included.



➔ It can be used with SELECT, INSERT, UPDATE, or DELETE statement.

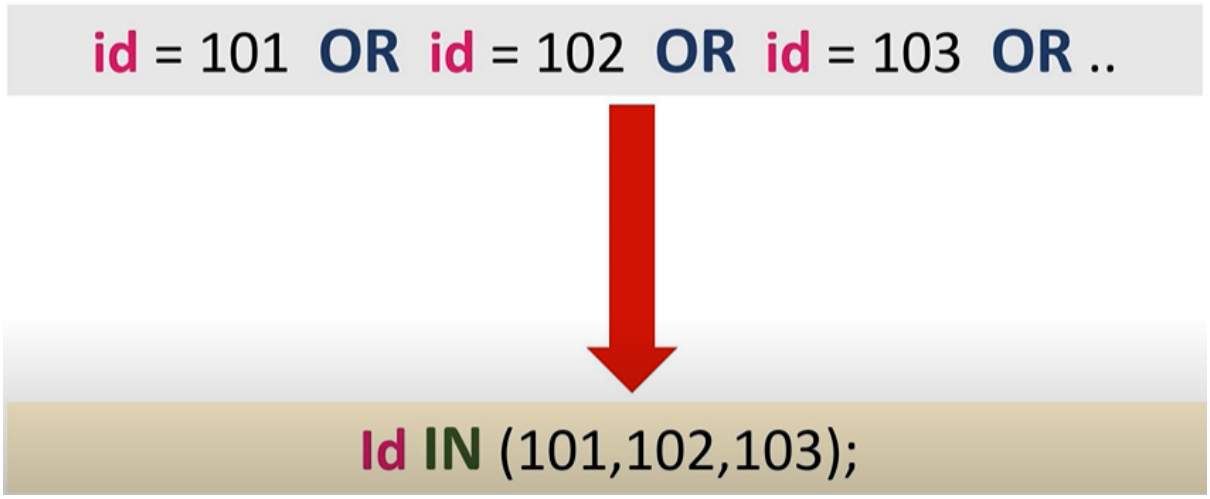


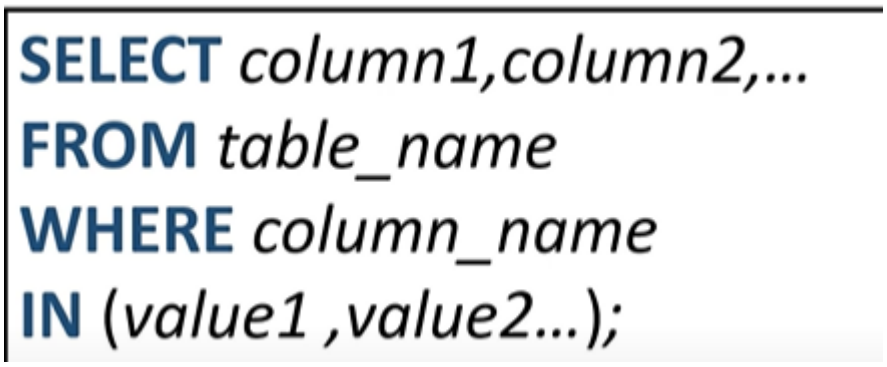

```
SELECT StudentID, StudentName, AGE FROM Students WHERE AGE BETWEEN 20 AND 25;
```

	StudentID	StudentName	AGE
1	1	Kasi	25
2	2	Surya	20
3	4	Kiran	22
4	5	Rohin	25

IN Operator

- ➔ The IN operator allows you to specify multiple values in a WHERE clause.
- ➔ It determines if a value matches any value in a list of values.
- ➔ IN operator returns 1 (true) if the value equals any value in the list (value1, value2, value3,.....). Otherwise, it returns 0.
- ➔ The IN operator is functionally equivalent to the combination of multiple OR operators.






```
SELECT StudentID, StudentName, AGE FROM Students WHERE StudentID IN(2,5,7);
```

Results Messages

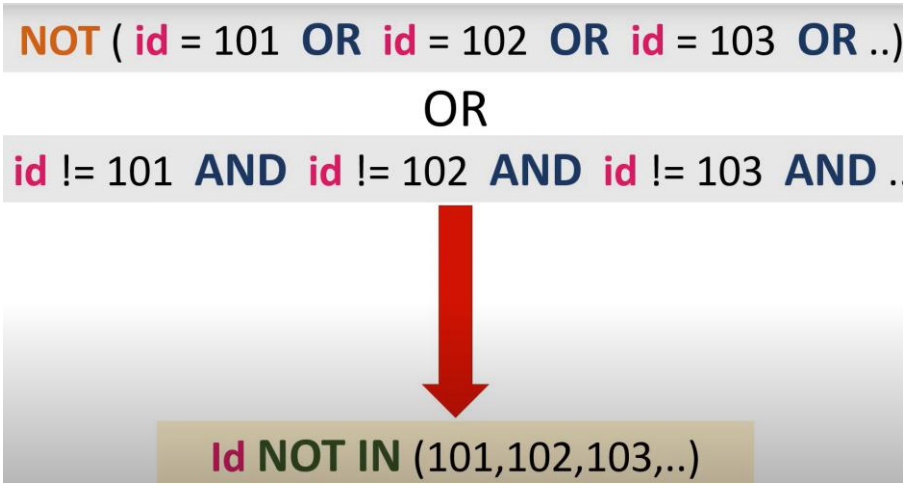
	StudentID	StudentName	AGE
1	2	Surya	20
2	5	Rohin	25

NOT IN Operator

➔ The NOT operator negates the IN Operator.

➔ The NOT IN operator returns 1 (true) if the value doesn't equal any value in the list. Otherwise, it returns 0.

➔ NOT IN() makes sure that the expression proceeded does not have any of the values present in the list.




```
SELECT column1,column2,...
FROM table_name
WHERE column_name
NOT IN (value1 ,value2...);
```

```
SELECT StudentID, StudentName, AGE FROM Students WHERE StudentID NOT IN(2,5,7);
```

Results | Messages

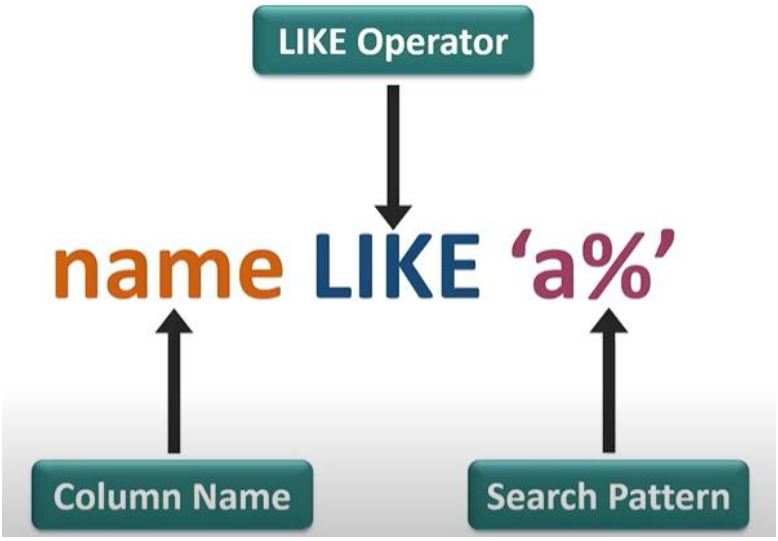
	StudentID	StudentName	AGE
1	1	Kasi	25
2	3	Ravi Kumar	18
3	4	Kiran	22

LIKE Operator & Wildcards

Filter Data Using
Comparison & Logical Operator

```
SELECT * FROM employee WHERE name = 'John' ;
SELECT * FROM employee WHERE salary > 10000 OR salary < 30000 ;
SELECT * FROM employee WHERE dob BETWEEN '1999-05-10' AND '1999-11-10' ;
SELECT * FROM employee WHERE department != 'HR' ;
```

➔ The LIKE Operator is used with the WHERE clause to get a result set that matches the given search pattern in a column.



➔ LIKE operator can be used in the WHERE clause of a SELECT, INSERT, UPDATE, or DELETE statement.

- Improves the performance of an application
- Saves time in filtering records from the table.

Syntax :

```
SELECT column1,column2,...
FROM table_name
WHERE column_name
LIKE 'search pattern';
```

Search Pattern

➔ A search condition made up of literal text, wildcard characters, or any combination of both.

Wildcard Characters

➔ There are two wildcards often used in conjunction with the LIKE Operator.

Wildcard Character	Description
%	represents zero, one, or multiple characters
_	represents one (single) character

➔ The percent sign and the underscore can also be used in combinations.

LIKE Operator	Description
LIKE 'j%'	Finds any values that starts with "j"
LIKE '%a'	Finds any values that ends with "a"
LIKE '%er%'	Finds any values that have "er" in any position
LIKE '_r%'	Finds any values that have "r" in the second position
LIKE 'b%g'	Finds any values that starts with "b" and ends with "g"

```
SELECT StudentID, StudentName, AGE FROM Students WHERE StudentName LIKE 'K%';
```

Results Messages

	StudentID	StudentName	AGE
1	1	Kasi	25
2	4	Kiran	22

```
SELECT StudentID, StudentName, AGE FROM Students WHERE StudentName LIKE '%i';
```

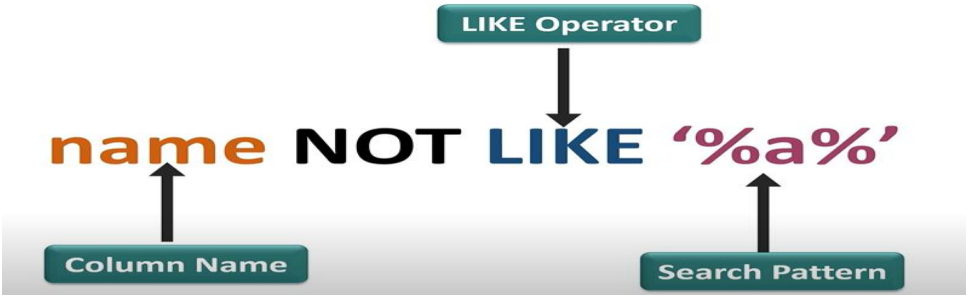
Results Messages

	StudentID	StudentName	AGE
1	1	Kasi	25

NOT LIKE Operator

➔ We can also invert the working of LIKE operator and ignore the result set matching with the given string pattern by using the NOT operator.

- Find all the names which doesn't contain 'a' in their names.




```
SELECT StudentID, StudentName, AGE FROM Students WHERE StudentName NOT LIKE '%i';
```

	StudentID	StudentName	AGE
1	2	Surya	20
2	3	Ravi Kumar	18
3	4	Kiran	22
4	5	Rohin	25

ESCAPE Character

➔ The ESCAPE character is used to escape pattern matching characters (wildcard characters) such as the (%) percentage and underscore (_) if they form part of the data.

➔ SQL treats the % character as a literal instead of a wildcard.

➔ Default value of escape character is '\' (Blackslash)

➔ You can also specify your own escape character that MySQL/SQL will use at the time of matching search pattern.

➔ ESCAPE '!'.

```
SELECT * FROM Students
WHERE StudentName LIKE '%95\%%' ESCAPE '\';
```

- This finds names that contain 95%.

- \ is the escape character, so \% means a literal %.

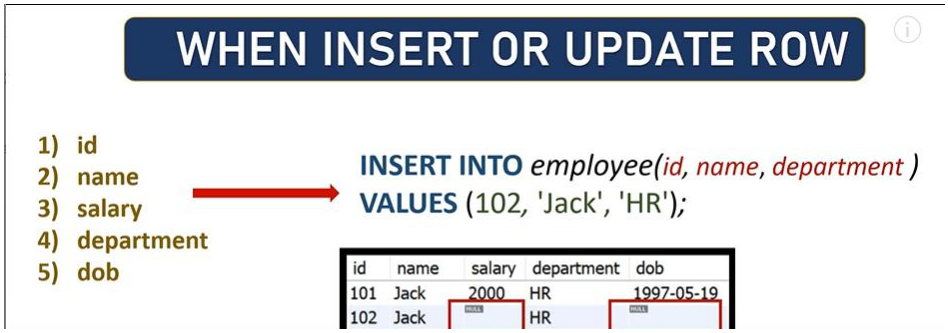
```
SELECT * FROM Students
WHERE StudentName LIKE 'A\_B' ESCAPE '\';
```

- This matches names like A_B, not A1B or A2B.

IS NULL & IS NOT NULL

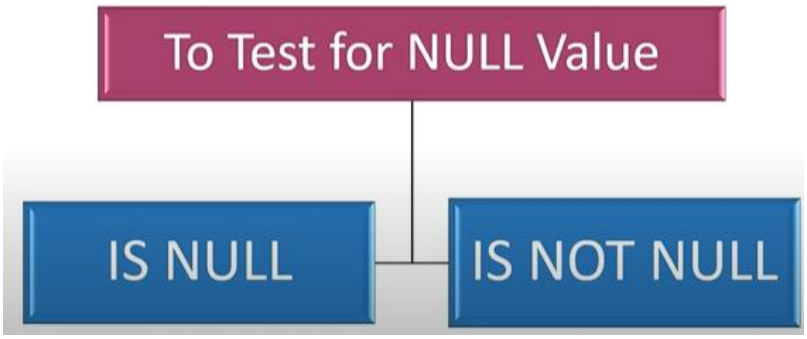
What is NULL?

➔ In SQL, NULL is both a value as well as a keyword.
➔ A field with a NULL value ; is a field with no value
➔ Its value is unknown.



➔ A NULL value is different from a
  - Zero (0)
  - Spaces ( )
  - Empty string (“”)
➔ NULL is not a data type.
➔ Arithmetic operations involving NULL always return NULL.

➔ All aggregate functions affect only rows that do not have NULL values.
➔ It is not possible to test for NULL values with comparison operators, such as =, <, or <>.



IS NULL Operator

➔ The IS NULL operator is used to test for NULL values.
➔ IS NULL operator is used with WHERE clause to test for a NULL value in a SELECT, INSERT, UPDATE or DELETE statement.

SYNTAX :

```
SELECT column1,column2,...
FROM table_name
WHERE column_name
IS NULL ;
```

```
SELECT * FROM Students WHERE EnrollmentDate IS NULL;
```

Results | Messages

	StudentID	StudentName	AGE	EnrollmentDate
1	6	Roudy	21	NULL

IS NOT NULL Operator

➔ The IS NOT NULL Operator is use to test for NOT NULL values.
➔ Return TRUE if the field doesn't contain NULL value.

```
SELECT column1,column2,...
FROM table_name
WHERE column_name
IS NOT NULL ;
```

```
SELECT * FROM Students WHERE EnrollmentDate IS NOT NULL;
```

Results | Messages

	StudentID	StudentName	AGE	EnrollmentDate
1	1	Kasi	25	2025-06-15
2	2	Surya	20	2025-06-15
3	3	Ravi Kumar	18	2025-06-15
4	4	Kiran	22	2025-07-15
5	5	Rohin	25	2025-09-19

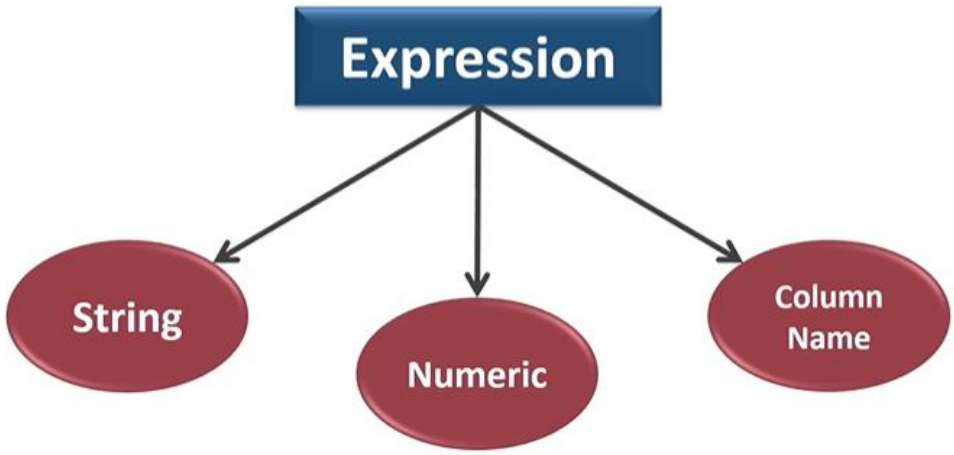
IFNULL()

➔ The IFNULL function accepts two expressions
- If the first expression is not NULL, it returns the first arguments.
- If the first expression is NULL, it returns the second argument.

➔ IFNULL() is mostly used with SELECT statement.

➔ Avoid using IFNULL() in where clause as it reduce performance of the query.

IFNULL *(Expression1, Expression 2)*



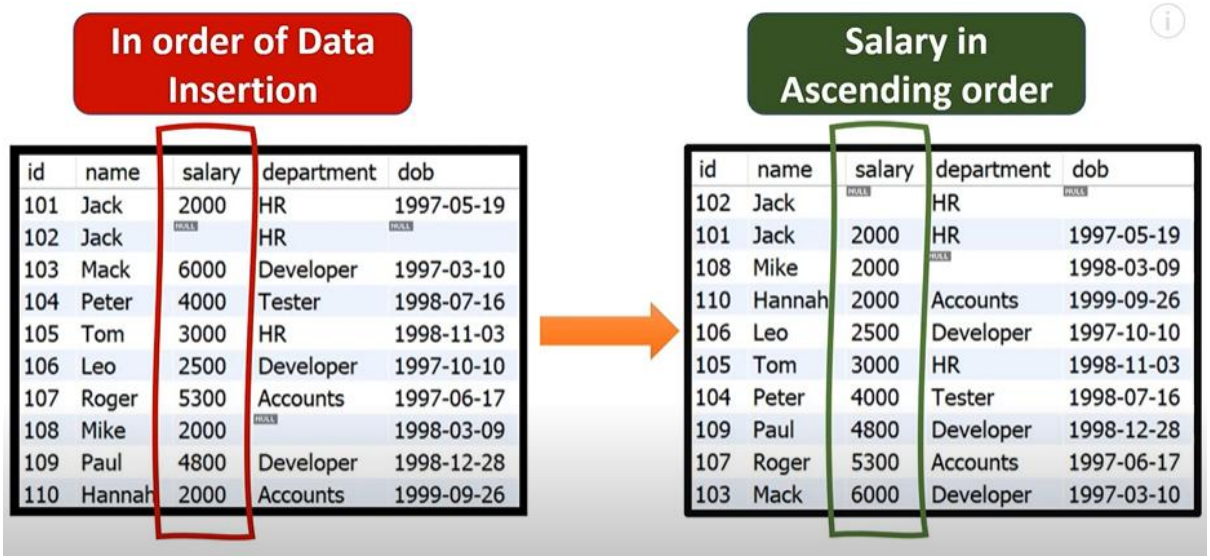
NOTE : IN **MSSQL** WE USE **ISNULL()** INSTEAD OF **IFNULL().**

```
SELECT StudentName, ISNULL(AGE, 0) AS AGE, EnrollmentDate
FROM Students WHERE StudentID != 6;
```

Results | Messages

	StudentName	AGE	EnrollmentDate
1	Kasi	25	2025-06-15
2	Surya	20	2025-06-15
3	Ravi Kumar	18	2025-06-15
4	Kiran	22	2025-07-15
5	Rohin	25	2025-09-19
6	MAMA	0	2025-06-15

ORDER BY Clause



id	name	salary	department	dob
101	Jack	2000	HR	1997-05-19
102	Jack	NULL	HR	NULL
103	Mack	6000	Developer	1997-03-10
104	Peter	4000	Tester	1998-07-16
105	Tom	3000	HR	1998-11-03
106	Leo	2500	Developer	1997-10-10
107	Roger	5300	Accounts	1997-06-17
108	Mike	2000	NULL	1998-03-09
109	Paul	4800	Developer	1998-12-28
110	Hannah	2000	Accounts	1999-09-26

id	name	salary	department	dob
102	Jack	NULL	HR	NULL
101	Jack	2000	HR	1997-05-19
108	Mike	2000	NULL	1998-03-09
110	Hannah	2000	Accounts	1999-09-26
106	Leo	2500	Developer	1997-10-10
105	Tom	3000	HR	1998-11-03
104	Peter	4000	Tester	1998-07-16
109	Paul	4800	Developer	1998-12-28
107	Roger	5300	Accounts	1997-06-17
103	Mack	6000	Developer	1997-03-10

➔ 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.

SYNTAX :

```
SELECT column1,column2,...
FROM table_name
ORDER BY column_name [ ASC|DESC ];
```

```
SELECT * FROM Students ORDER BY AGE;
```

	StudentID	StudentName	AGE	EnrollmentDate
1	7	MAMA	NULL	2025-06-15
2	3	Ravi Kumar	18	2025-06-15
3	2	Surya	20	2025-06-15
4	6	Roudy	21	NULL
5	4	Kiran	22	2025-07-15
6	5	Rohin	25	2025-09-19

```
SELECT * FROM Students ORDER BY AGE DESC;
```

Results Messages

	StudentID	StudentName	AGE	EnrollmentDate
1	1	Kasi	25	2025-06-15
2	5	Rohin	25	2025-09-19
3	4	Kiran	22	2025-07-15
4	6	Roudy	21	NULL
5	2	Surya	20	2025-06-15
6	3	Ravi Kumar	18	2025-06-15
7	7	MAMA	NULL	2025-06-15

For Multiple Columns

SYNTAX :

SELECT *column1,column2,...*
FROM *table_name*
ORDER BY *column1* **[ASC|DESC]**, *column2* **[ASC|DESC]** ,.....;

```
SELECT * FROM Students ORDER BY StudentName ASC, AGE DESC;
```

Results Messages

	StudentID	StudentName	AGE	EnrollmentDate
1	1	Kasi	25	2025-06-15
2	4	Kiran	22	2025-07-15
3	7	MAMA	NULL	2025-06-15
4	3	Ravi Kumar	18	2025-06-15
5	5	Rohin	25	2025-09-19
6	6	Roudy	21	NULL
7	2	Surya	20	2025-06-15

📘 SQL Server Row Limiting Techniques: TOP vs OFFSET-FETCH

✅ **Purpose:**

Used to return a fixed number of rows from a query result.

✅ Syntax:

Sql

```sql
SELECT TOP (n) column1, column2, ...
FROM table_name
ORDER BY column_name;
```

✅ Example:

```sql
SELECT TOP 10 StudentName, Age
FROM Students
ORDER BY Age DESC;
```

- Returns the top 10 oldest students.

✅ **Notes:**

- TOP is simple and efficient for quick row limits.
- Works without ORDER BY, but ordering ensures predictable results.
- Cannot be used for pagination.

🔹 2. OFFSET-FETCH Clause

✅ **Purpose:**

Used for pagination — skipping a number of rows and fetching the next set.

✅ **Syntax:**

```
SELECT column1, column2, ...
FROM table_name
ORDER BY column_name
OFFSET n ROWS FETCH NEXT m ROWS ONLY;
```

✅ **Example:**

```
SELECT StudentName, Age
FROM Students
ORDER BY StudentID
OFFSET 10 ROWS FETCH NEXT 10 ROWS ONLY;
```

- **Skips the first 10 rows and returns the next 10.**

✅ **Notes:**

- **Requires ORDER BY clause.**
- **Introduced in SQL Server 2012.**
- **Ideal for paging through large datasets.**

🔄 **Comparison Table**

Feature	**`TOP` Clause**	**`OFFSET-FETCH` Clause**
Use Case	Limit rows	Pagination
Syntax Simplicity	Simple	Slightly more complex
Requires `ORDER BY`	Optional	Mandatory
Pagination Support	❌ No	✅ Yes

Aggregate Functions

➔ An Aggregate Function in SQL performs a calculation on multiple values and returns a single value as a result.

➔ They are also Known as Multiple Row Function OR Group Function.

➔ An aggregate function ignores NULL values when it performs the calculation, except for the count function.

➔ The aggregate functions are commonly used with the SELECT statement to generate summary reports from a database.

➔ Aggregate functions often used with GROUP BY Clause.

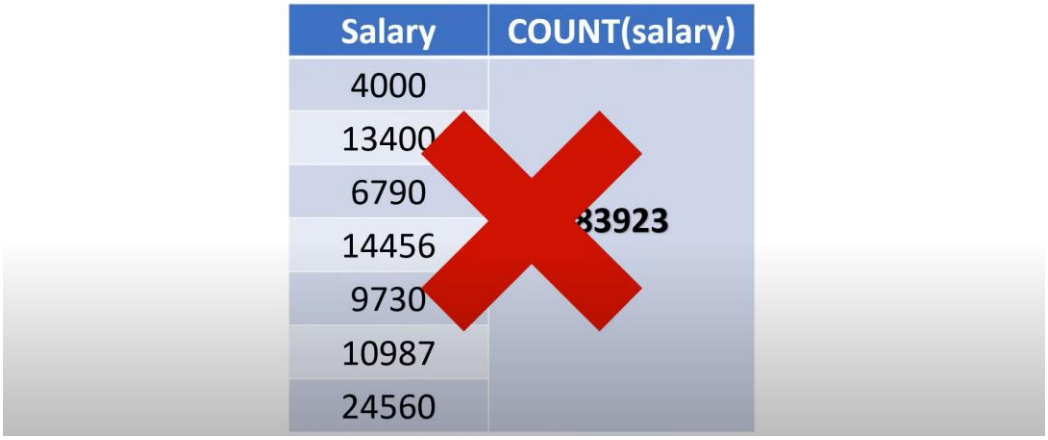
- **COUNT( )**
  - The COUNT() aggregate function returns the total number of rows from a database table that matches the defined criteria in the SQL query.
  - It can work on both numeric and non-numeric data types.

SYNTAX :

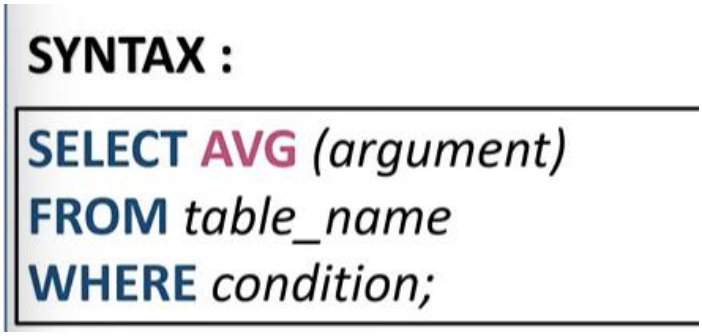
```
SELECT COUNT (argument)
FROM table_name
WHERE condition;
```

Argument	COUNT ()
*	COUNT (*) consider NULL Values also
column_name	COUNT(id)
DISTINCT column_name	COUNT(DISTINCT id)

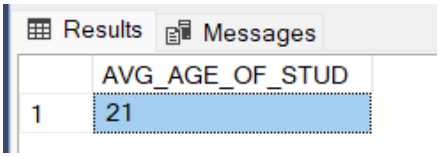
```
SELECT salary, COUNT (salary) FROM employee;
```



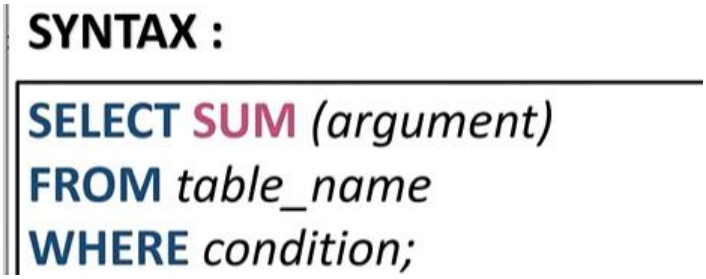
- **AVG( )**
  - The AVG() is used to calculate the average value of the specific column.
  - It works only on numeric and non-NULL values.




```
SELECT AVG(AGE) AS AVG_AGE_OF_STUD FROM Students;
```



- **SUM( )**
  - The SUM() function takes the name of the column as an argument and returns the sum of all the non-NULL values in that column.
  - It sum only non-null value.
  - It works on numeric fields only.
  - When we applied to columns containing non-numeric (ex – strings, text), the function will return 0.



Argument	COUNT()
Column_name	SUM(id)
DISTINCT column_name	SUM(DISTINT id)
expression	SUM(id+salary)

```
SELECT SUM(AGE) AS SUM_OF_AGES FROM Students WHERE AGE>20;
```

Results | Messages

	SUM_OF_AGES
1	93

- **MIN( )**
  - MIN function is used to find the minimum value of a certain column.
  - It can work on both numeric and non-numeric data types.
  - It doesn't consider NULL values.

SYNTAX :

```
SELECT MIN (column_name)
FROM table_name
WHERE condition;
```

```
SELECT MIN(AGE) AS MIN_AGE_OF_STUD FROM Students;
```

Results | Messages

	MIN_AGE_OF_STUD
1	18

- **MAX( )**
  - MAX function is used to find the maximum value of a certain column
  - It can work on both numeric and non-numeric data types.
  - It doesn't consider NULL values.

SYNTAX :

```
SELECT MAX (column_name)
FROM table_name
WHERE condition;
```

```
SELECT MAX(AGE) AS MAX_AGE_OF_STUD FROM Students;
```

Results | Messages

	MAX_AGE_OF_STUD
1	25

GROUP BY Clause

- GROUP BY statement is used for organizing or arranging data into groups that have identical value for one or more columns.

Employee Table

Department	Salary
HR	10000
Developer	3500
Developer	4000
Tester	3000
HR	5500
Developer	15000

HR	Developer	Tester
10000	3500	3000
5500	4000	
	15000	

- GROUP BY statement is often used with aggregate functions [COUNT(), SUM(), AVG(), MIN(), MAX()] to group the result-set by one or more columns.
- NULL values are considered equivalent for grouping purposes.

IMPORTANT Points

➔ GROUP BY clause is mostly used with the SELECT statement.
➔ In the query, GROUP BY clause is placed after the WHERE clause.
➔ In the query, GROUP BY clause is placed before HAVING & ORDER BY clause if used any.

SYNTAX :

```
SELECT column_name(s), function_name(argument)
FROM table_name
WHERE condition
GROUP BY column_name(s)
HAVING condition
ORDER BY column _name(s);
```

EXAMPLE :

```
SELECT department
FROM employee
GROUP BY department;
```

EXAMPLE :

```
SELECT department, SUM(salary)
FROM employee
GROUP BY department;
```

Employee Table

Department	Salary
HR	10000
Developer	3500
Developer	4000
Tester	3000
HR	5500
Developer	15000

```
SELECT department ,SUM(salary)
FROM employee
GROUP BY department;
```

Department	SUM(salary)
HR	15500
Developer	22500
Tester	3000

Q) Find the number of employee working in each department form the “employee” table ?

```
1 • SELECT department,COUNT(*) FROM employee GROUP BY department;
```

department	COUNT(*)
HR	3
Developer	3
Tester	1
Accounts	2
NULL	1

id	name	salary	department	dob
101	Jack	2000	HR	1997-05-19
102	Jack	NULL	HR	NULL
103	Mack	6000	Developer	1997-03-10
104	Peter	4000	Tester	1998-07-16
105	Tom	3000	HR	1998-11-03
106	Leo	2500	Developer	1997-10-10
107	Roger	5300	Accounts	1997-06-17
108	Mike	2000	NULL	1998-03-09
109	Paul	4800	Developer	1998-12-28
110	Hannah	2000	Accounts	1999-09-26

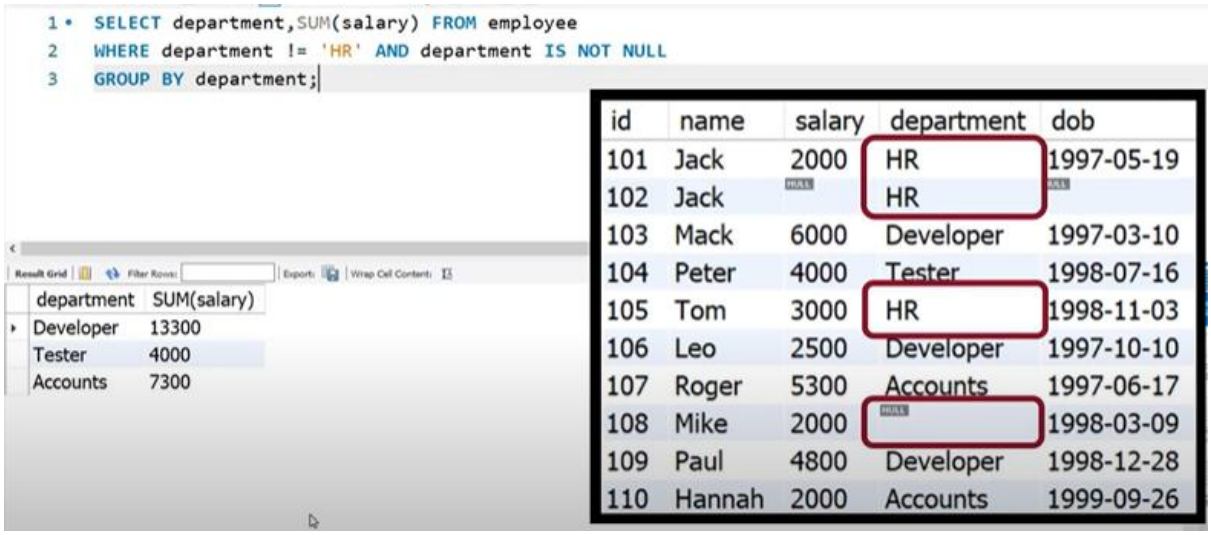
Q) Find the sum of salary of each department from “employee” table and sort the result in ascending order by sum of salary ?

```
1 • SELECT department,SUM(salary) FROM employee
2   GROUP BY department
3   ORDER BY SUM(salary);
```

department	SUM(salary)
NULL	2000
Tester	4000
HR	5000
Accounts	7300
Developer	13300

id	name	salary	department	dob
101	Jack	2000	HR	1997-05-19
102	Jack	NULL	HR	NULL
103	Mack	6000	Developer	1997-03-10
104	Peter	4000	Tester	1998-07-16
105	Tom	3000	HR	1998-11-03
106	Leo	2500	Developer	1997-10-10
107	Roger	5300	Accounts	1997-06-17
108	Mike	2000	NULL	1998-03-09
109	Paul	4800	Developer	1998-12-28
110	Hannah	2000	Accounts	1999-09-26

Q) Find the sum of salary of each department from “employee” table except “HR” department and department should not be NULL ?

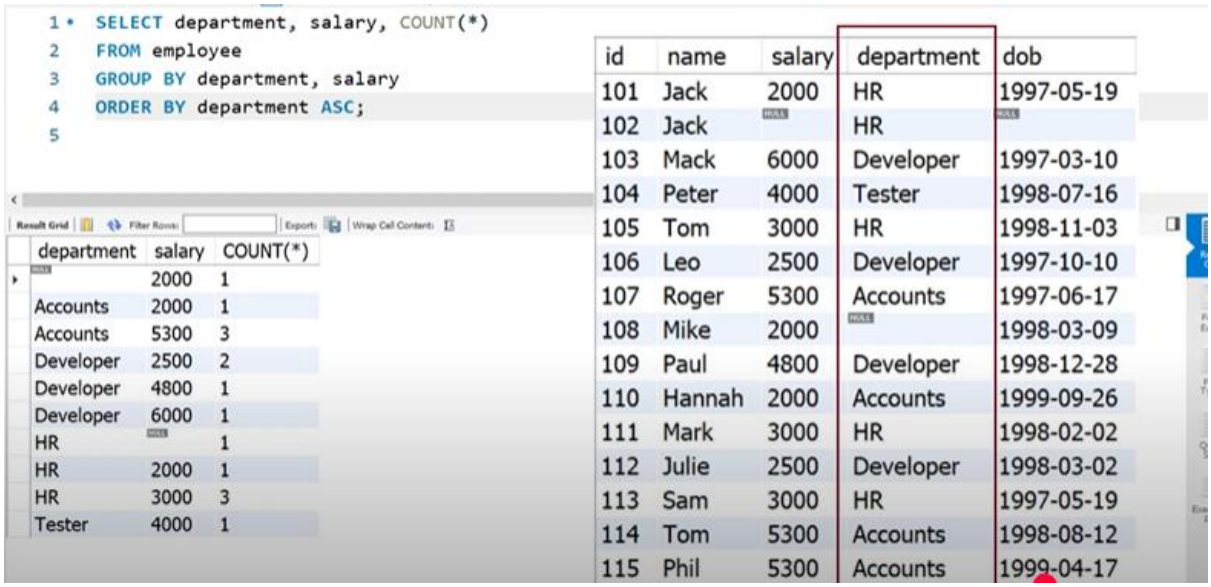


```
1 • SELECT department,SUM(salary) FROM employee
2   WHERE department != 'HR' AND department IS NOT NULL
3   GROUP BY department;
```

department	SUM(salary)
Developer	13300
Tester	4000
Accounts	7300

id	name	salary	department	dob
101	Jack	2000	HR	1997-05-19
102	Jack	NULL	HR	NULL
103	Mack	6000	Developer	1997-03-10
104	Peter	4000	Tester	1998-07-16
105	Tom	3000	HR	1998-11-03
106	Leo	2500	Developer	1997-10-10
107	Roger	5300	Accounts	1997-06-17
108	Mike	2000	NULL	1998-03-09
109	Paul	4800	Developer	1998-12-28
110	Hannah	2000	Accounts	1999-09-26

Q) Find the number of employee who has same department and salary form “employee” table and sort the result by department in ascending order ?



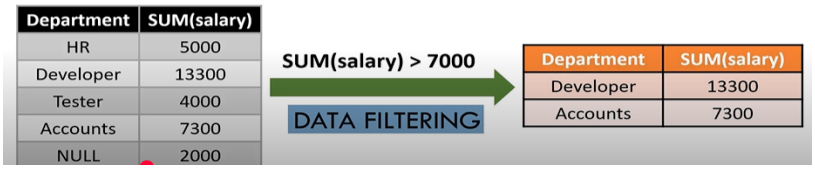
```
1 • SELECT department, salary, COUNT(*)
2   FROM employee
3   GROUP BY department, salary
4   ORDER BY department ASC;
5
```

department	salary	COUNT(*)
NULL	2000	1
Accounts	2000	1
Accounts	5300	3
Developer	2500	2
Developer	4800	1
Developer	6000	1
HR	NULL	1
HR	2000	1
HR	3000	3
Tester	4000	1

id	name	salary	department	dob
101	Jack	2000	HR	1997-05-19
102	Jack	NULL	HR	NULL
103	Mack	6000	Developer	1997-03-10
104	Peter	4000	Tester	1998-07-16
105	Tom	3000	HR	1998-11-03
106	Leo	2500	Developer	1997-10-10
107	Roger	5300	Accounts	1997-06-17
108	Mike	2000	NULL	1998-03-09
109	Paul	4800	Developer	1998-12-28
110	Hannah	2000	Accounts	1999-09-26
111	Mark	3000	HR	1998-02-02
112	Julie	2500	Developer	1998-03-02
113	Sam	3000	HR	1997-05-19
114	Tom	5300	Accounts	1998-08-12
115	Phil	5300	Accounts	1999-04-17

HAVING Clause

```
SELECT department, SUM(salary)
FROM employee
GROUP BY department
```



- ➔ SQL HAVING clause is similar to the WHERE clause, they are both used to filter rows in a table based on conditions.
- ➔ HAVING clause was added to SQL because the WHERE keyword cannot be used with aggregate functions.
- ➔ The HAVING clause filters the grouped rows.
- ➔ The HAVING clause must be followed by the GROUP BY clause.
- ➔ It can also contain multiple conditions to filter the grouped results.
- ➔ These conditions are separated by various operators like AND, OR etc.

SYNTAX :

```
SELECT column_name(s), function_name(argument)
FROM table_name
WHERE condition
GROUP BY column_name(s)
HAVING condition
ORDER BY column _name(s);
```

Q) Find the number of students enrolled in each course as total_students from the Students table. Only include courses with more than one student. Also, sort the result by total_students ?

```
SELECT Course, COUNT(*) AS total_students
FROM Students
GROUP BY Course
HAVING COUNT(*) > 1
ORDER BY total_students;
```

SYNTAX :

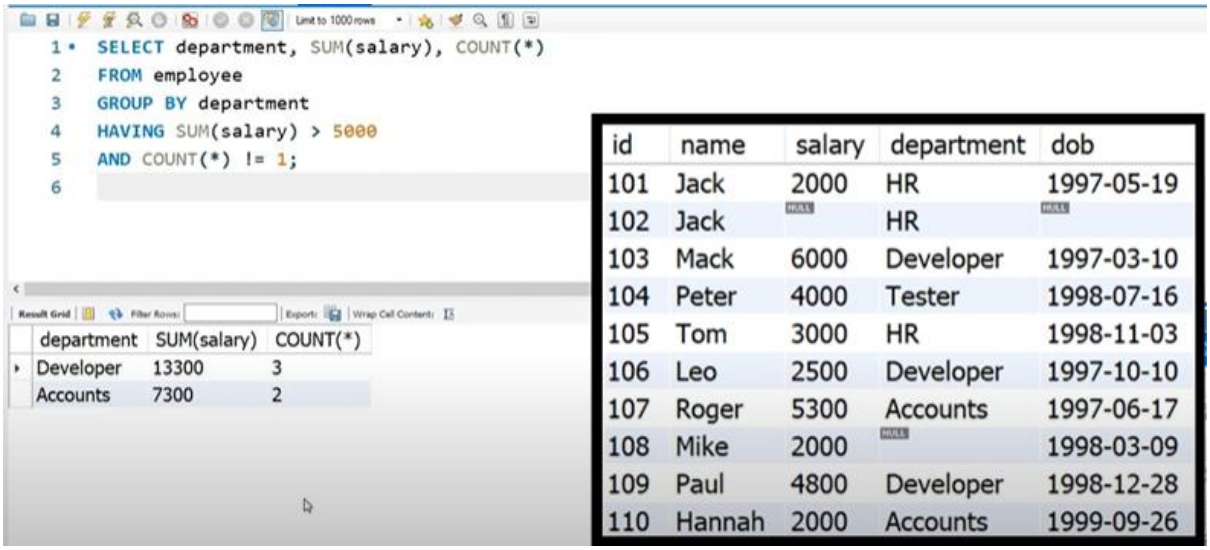
```
SELECT column_name(s), function_name(argument
FROM table_name
WHERE condition
GROUP BY column_name(s)
HAVING condition1
AND|OR condition2
AND|OR condition3.....
ORDER BY column _name(s);
```

Q) Write SQL query to calculate the total salary and the number of employees in each department where the total salary is more than 5000 and number of employee is other than 1.

Question Breakdown

1. Calculate the total salary.
   `SUM(salary)`
2. Calculate number of employees.
   `COUNT(*)`
3. In each department.
   `GROUP BY department`
4. Total salary is more than 5000
   `HAVING SUM(salary) > 5000`
5. Number of employee is other than 1.
   `HAVING COUNT(*) != 1`

```
SELECT department, SUM(salary),
COUNT(*)
FROM employee
GROUP BY department
HAVING SUM(salary) > 5000
AND COUNT(*) != 1;
```



SYNTAX :

```
SELECT column_name(s), function_name(argument)
FROM table_name
WHERE condition
GROUP BY column_name(s)
HAVING column_name BETWEEN value1 AND value2
AND|OR condition(s)
ORDER BY column _name(s);
```

Q) Write a SQL query to retrieve the total salary of the employees in each department of a company, where the total salary of each department falls between 2000 and 6000 and the department field is not null ?

Question Breakdown

1. The total salary of the employee.

   `SUM(salary)`

2. In each department.

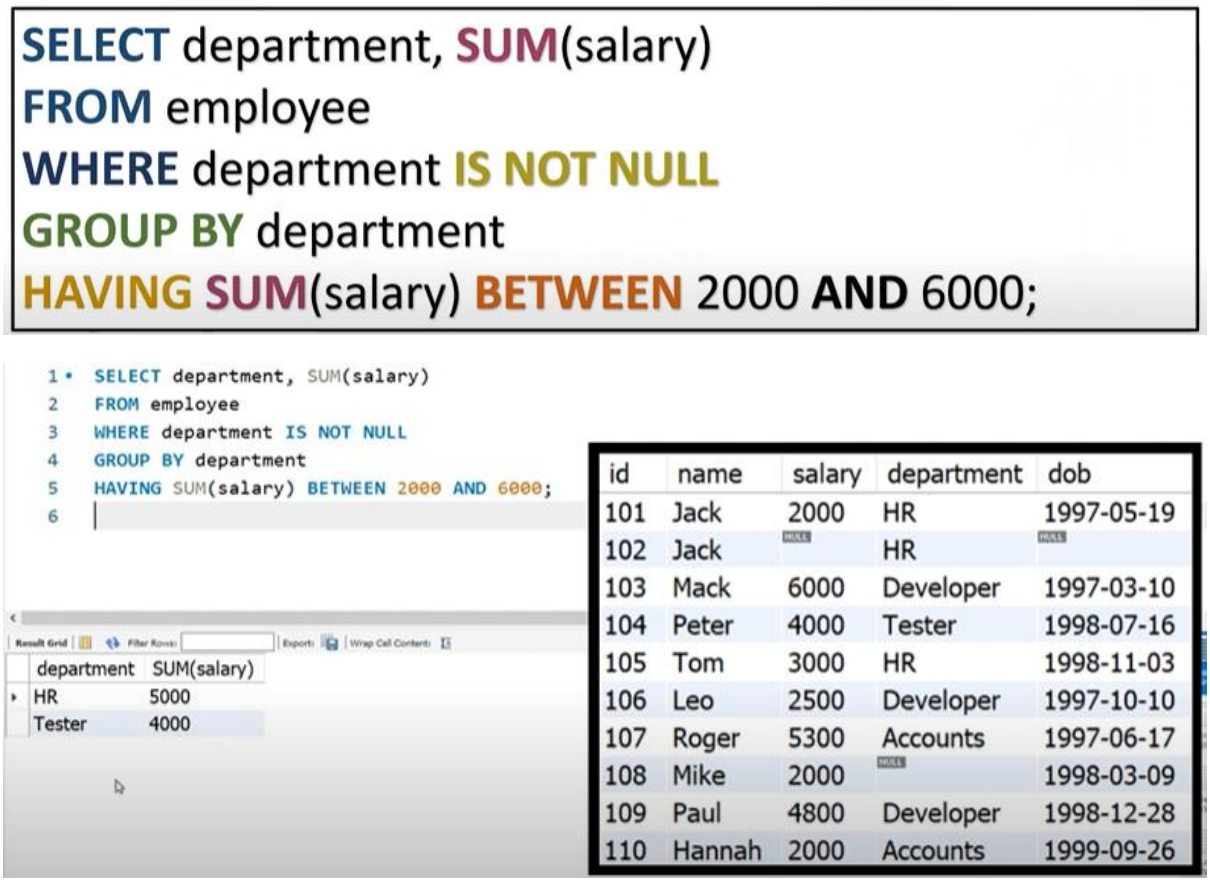
   `GROUP BY department`

3. Total salary falls between 2000 &6000

   `HAVING SUM(salary) BETWEEN 2000 AND 6000`

4. Department field is not null.

   `WHERE department IS NOT NULL`



🆚 WHERE vs HAVING Clause in SQL

Feature	`WHERE` Clause	`HAVING` Clause
🔍 Purpose	Filters **rows** before grouping or aggregation	Filters **groups** after aggregation
🧠 Applies To	Individual rows	Grouped data (after `GROUP BY`)
📊 Aggregate Functions	❌ Cannot use aggregate functions	✅ Can use aggregate functions like `SUM`, `AVG`
📌 Position in Query	Comes **before** `GROUP BY`	Comes **after** `GROUP BY`
🛠️ Usage	Used with `SELECT`, `UPDATE`, `DELETE`	Only used with `SELECT`
🧪 Example Functions	`UPPER()`, `LOWER()`	`COUNT()`, `SUM()`, `AVG()`

✅ Examples

🔹 Using WHERE

```sql
SELECT StudentName, Age
FROM Students
WHERE Age >= 18;
```

- Filters students who are 18 or older **before** any grouping.

🔹 Using HAVING

```sql
SELECT Course, COUNT(*) AS total_students
FROM Students
GROUP BY Course
HAVING COUNT(*) > 1;
```

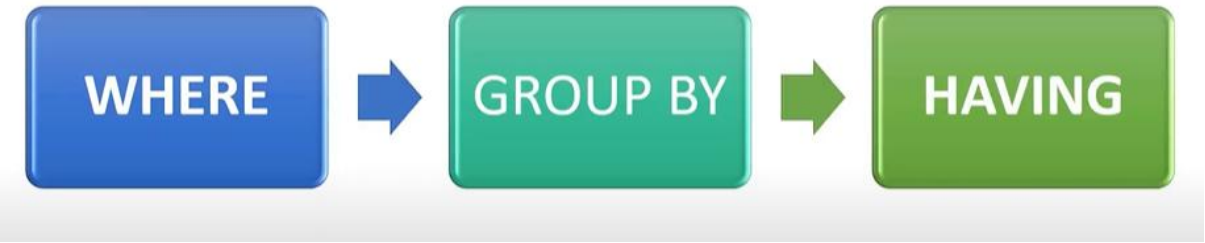
Groups students by course, then filters groups with more than 1 student.

🧠 Summary

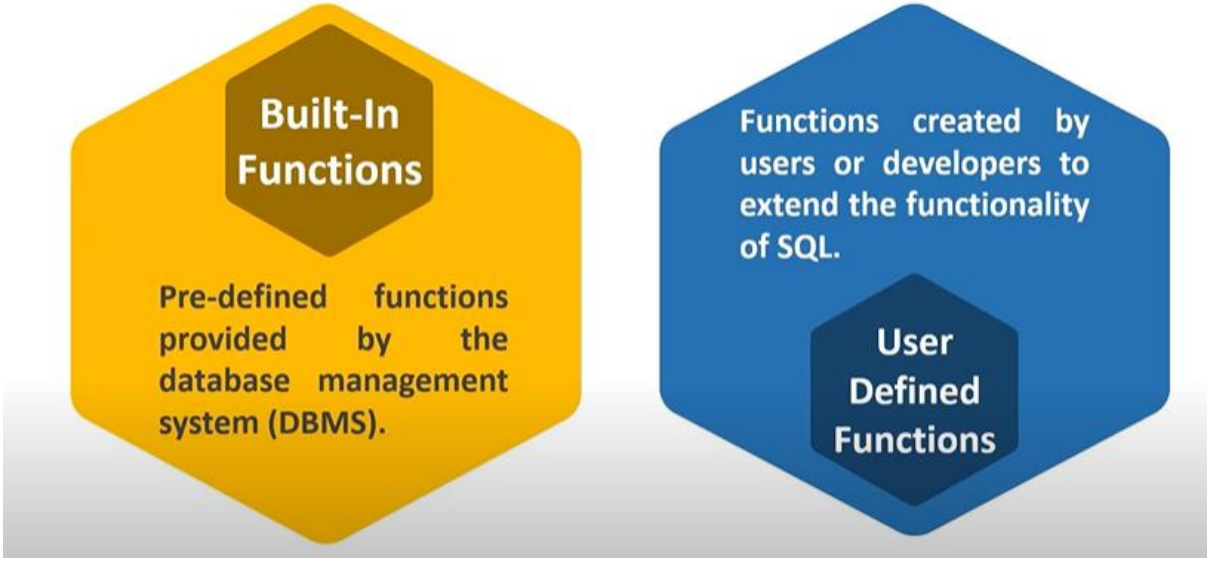
- Use WHERE to filter **individual rows**.
- Use HAVING to filter **aggregated groups**.

WHERE Clause	HAVING Clause
WHERE Clause is used to filter the records from the table based on the specified condition.	HAVING Clause is used to filter record from the groups based on the specified condition.
WHERE Clause implements in row operations	HAVING Clause implements in column operation
WHERE Clause can be used without GROUP BY Clause	HAVING Clause cannot be used without GROUP BY Clause
WHERE Clause can be used with SELECT, UPDATE, DELETE statement.	HAVING Clause can only be used with SELECT statement.

WHERE Clause	HAVING Clause
WHERE Clause cannot contain aggregate functions	HAVING Clause can contain aggregate functions
The WHERE clause is used to filter rows *before* the grouping is performed.	The HAVING clause is used to filter rows *after* the grouping is performed.
GROUP BY Clause is used after WHERE Clause	GROUP BY Clause is used before HAVING Clause
WHERE Clause is used with single row function like UPPER, LOWER etc.	HAVING Clause is used with multiple row function like SUM, COUNT etc.



SQL FUNCTIONS



Types Of Built-In Functions

- **Aggregate Functions**
- **String Functions**
- **Numeric Functions**
- **Date and Time Functions**
- **Conversion Functions**

Aggregate Functions :

➔ Aggregate Functions in SQL are used to perform calculations on sets of rows and return a single value as a result.

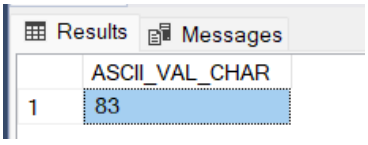
➔ These functions summarize or aggregate data within a column or across multiple columns.

➔ Types
- COUNT()
- SUM()
- AVG()
- MIN()
- MAX()

String Functions :

- **ASCII**
  - **This function is used to find the ASCII value of a character.**


```
SELECT ASCII('S') AS ASCII_VAL_CHAR;
```



- # CONACT
  - **This function is used to add two words or strings together.**

SYNTAX :

```
SELECT column_name(s), CONCAT(string1,string2,...)
FROM table_name;
```

```
SELECT Course, CONCAT(StudentName, ' has id -', StudentID) AS student_id
FROM Students;
```

Results | Messages

	Course	student_id
1	Math	Kasi has id -1
2	Science	Surya has id -2
3	Math	Ravi Kumar has id -3
4	English	Kiran has id -4
5	Science	Rohin has id -5
6	Math	Roudy has id -6
7	English	MAMA has id -7

- # CONCAT_WS
  - **This function is used to concatenate multiple strings together with a specified separator.**

SYNTAX :

```
SELECT column_name(s),
CONCAT_WS(separator,string1,string2,...)
FROM table_name;
```

```
SELECT Course, CONCAT_WS('_',StudentID,StudentName,Grade) AS concat_string
FROM Students;
```

Results | Messages

	Course	concat_string
1	Math	1_Kasi_A
2	Science	2_Surya_B
3	Math	3_Ravi Kumar_A
4	English	4_Kiran_C
5	Science	5_Rohin_B
6	Math	6_Roudy_B
7	English	7_MAMA_A

- # LENGTH
  - **This function is used to find the length of a word or string.**

```
SELECT DATALENGTH('Manohar');
```

Results | Messages

	(No column name)
1	7

SYNTAX :

SELECT *column_name(s),* **LENGTH**(*column_name|string*)
FROM *table_name;*

```
SELECT StudentID, StudentName,LEN(StudentName) AS Student_Name_Length
FROM Students;
```

Results | Messages

	StudentID	StudentName	Student_Name_Length
1	1	Kasi	4
2	2	Surya	5
3	3	Ravi Kumar	10
4	4	Kiran	5
5	5	Rohin	5
6	6	Roudy	5
7	7	MAMA	4

- # UPPER or UCASE
  - **This function is used to make the string in uppercase.**

SYNTAX :

SELECT *column_name(s),* **UPPER**(*column_name|string*)
FROM *table_name;*

```
SELECT StudentID, UPPER(StudentName) FROM Students;
```

Results | Messages

	StudentID	(No column name)
1	1	KASI
2	2	SURYA
3	3	RAVI KUMAR
4	4	KIRAN
5	5	ROHIN
6	6	ROUDY
7	7	MAMA

- # LOWER or LCASE
  - **This function is used to make the string in lowercase.**

SYNTAX :

```
SELECT column_name(s), LOWER(column_name|string)
FROM table_name;
```

```
SELECT StudentID, LOWER(StudentName) AS Lower_Case_StudentName FROM Students;
```

Results | Messages

	StudentID	Lower_Case_StudentName
1	1	kasi
2	2	surya
3	3	ravi kumar
4	4	kiran
5	5	rohin
6	6	roudy
7	7	mama

- # SUBSTRING | SUBSTR
  - **It allows you to extract a portion (substring) of a string based on a specified starting position and Length.**

SYNTAX :

```
SELECT column_name(s),
SUBSTRING(string_expression, start_position, length)
FROM table_name;
```

  - **String_expression:**
    - **The string from which you want to extract a substring.**
  - **Start_position:**
    - **The starting position of the substring within the string. It is usually 1-based (the first character is at position 1).**
  - **Length (Optional):**
    - **The length of the substring to be extracted in the absence of length, it will return the remaining characters from the starting position to the end of the string.**

```sql
SELECT
    StudentID,
    StudentName,
    SUBSTRING(CONVERT(VARCHAR(10), EnrollmentDate, 120), 1, 4) AS result
FROM Students;
```

Here's what each part does:

- **StudentID, StudentName → Just selecting those columns as-is from the Students table.**
- **CONVERT(VARCHAR(10), EnrollmentDate, 120) →**
  - **EnrollmentDate is likely stored as a DATE or DATETIME.**
  - **CONVERT(..., 120) changes it into a string in the format YYYY-MM-DD.**
  - **VARCHAR(10) means you only keep the first 10 characters (e.g., "2025-09-22").**
- **SUBSTRING(..., 1, 4) → Takes the first 4 characters from that string, which is the year ("2025").**
- **AS result → Names that extracted year column as result.**

Results | Messages

	StudentID	StudentName	result
1	1	Kasi	2025
2	2	Surya	2025
3	3	Ravi Kumar	2025
4	4	Kiran	2025
5	5	Rohin	2025
6	6	Roudy	NULL
7	7	MAMA	2025

2️⃣ Why You're Seeing NULL for One Row

In your output, one student (Roudy) has NULL in the result column. That happens because:

- **If EnrollmentDate is NULL for that row, → CONVERT() returns NULL → SUBSTRING() on NULL also returns NULL.**

So the NULL in result means that student has no enrollment date recorded in the table.

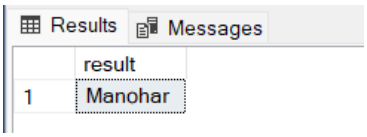
- **TRIM | LTRIM | RTRIM**
  - **Remove specified characters (or spaces) from the beginning and/or end of a string. It is commonly used to eliminate leading or trailing spaces from string values**

SYNTAX :

```
SELECT column_name(s),
TRIM([trim_character FROM ] string_expression)
FROM table_name;
```

  - **trim_character is optional**

```
SELECT TRIM('     Manohar     ') AS result;
```



2. Remove specific characters

```
SELECT TRIM('#$' FROM '$$#Hello#$') AS Cleaned;
```

Result:

```
Hello
```

Here, all # and $ at the start or end are removed.

3. Remove only from one side (SQL Server 2022+)

```
SELECT TRIM(LEADING '0' FROM '000123400') AS Cleaned;
```

```
123400
```

```
SELECT TRIM(LEADING '0' FROM '000456') AS result;
```

```
SELECT TRIM(TRAILING '/' FROM 'http://') AS result;
```

Result:

⚡ Key Notes

- **Before SQL Server 2017, you had to combine LTRIM() and RTRIM():**

  sql

  RTRIM(LTRIM(string))

- **TRIM() returns NULL if the input is NULL.**
- **Works with CHAR, NCHAR, VARCHAR, and NVARCHAR (but not MAX types for the characters argument).**

- # REPLACE
  - **Replaces all occurrences of a substring within a string.**

SYNTAX :

```
SELECT column_name(s),
REPLACE(string_expression, search_string, replacement_string)
FROM table_name;
```

```
SELECT StudentID, StudentName, REPLACE(Course,'Math','Mathematics')
AS result FROM Students
```

Results | Messages

	StudentID	StudentName	result
1	1	Kasi	Mathematics
2	2	Surya	Science
3	3	Ravi Kumar	Mathematics
4	4	Kiran	English
5	5	Rohin	Science
6	6	Roudy	Mathematics
7	7	MAMA	English

- # REVERSE
  - **This function is used to reverse a string.**

SYNTAX :

```
SELECT column_name(s), REVERSE(column_name|string)
FROM table_name;
```

```
SELECT StudentID, REVERSE(StudentName) AS result
FROM Students;
```

Results | Messages

	StudentID	result
1	1	isaK
2	2	ayruS
3	3	ramuK ivaR
4	4	nariK
5	5	nihoR
6	6	yduoR
7	7	AMAM

- # LEFT | RIGHT
  - **LEFT – Returns a specified number of characters from the left side of a string.**
  - **RIGHT – Returns a specified number of characters from the right side of a string.**

SYNTAX :

```
SELECT column_name(s), LEFT | RIGHT
(string_expression,length)
FROM table_name;
```

1️⃣ Using LEFT() — Extract characters from the start of a string

Example: Get the first 3 letters of each student's name

```
SELECT
    StudentID,
    StudentName,
    LEFT(StudentName, 3) AS First3Chars
FROM Students;
```

	StudentID	StudentName	First3Chars
1	1	Kasi	Kas
2	2	Surya	Sur
3	3	Ravi Kumar	Rav
4	4	Kiran	Kir
5	5	Rohin	Roh
6	6	Roudy	Rou
7	7	MAMA	MAM

2️⃣ Using RIGHT() — Extract characters from the end of a string

```sql
SELECT
    StudentID,
    StudentName,
    RIGHT(StudentName, 2) AS Last2Chars
FROM Students;
```

Results Messages

	StudentID	StudentName	Last2Chars
1	1	Kasi	si
2	2	Surya	ya
3	3	Ravi Kumar	ar
4	4	Kiran	an
5	5	Rohin	in
6	6	Roudy	dy
7	7	MAMA	MA

```sql
SELECT
    StudentID,
    StudentName,
    LEFT(StudentName, 1) AS FirstLetter,
    RIGHT(StudentName, 1) AS LastLetter,
    LEFT(CONVERT(VARCHAR(10), EnrollmentDate, 120), 4) AS YearPart,
    RIGHT(CONVERT(VARCHAR(10), EnrollmentDate, 120), 2) AS DayPart
FROM Students;
```

Results Messages

	StudentID	StudentName	FirstLetter	LastLetter	YearPart	DayPart
1	1	Kasi	K	i	2025	15
2	2	Surya	S	a	2025	15
3	3	Ravi Kumar	R	r	2025	15
4	4	Kiran	K	n	2025	15
5	5	Rohin	R	n	2025	19
6	6	Roudy	R	y	NULL	NULL
7	7	MAMA	M	A	2025	15

Numeric Functions :

- ABS
- ROUND
- CEIL
- FLOOR
- MOD
- POWER
- SQRT
- GREATEST
- LEAST
- RAND

- **ABS**
  - **This function returns the absolute(positive) value of a number.**

SYNTAX :

```
SELECT ABS(number);
```

```
SELECT ABS(-76.5);
```

	result
1	76.5

- **ROUND**
  - This function rounds a number to a specified number of decimal places.

SYNTAX :

```
SELECT ROUND(number | column , decimal_place);
```

```
SELECT CAST(ROUND(752.878787887, 2) AS DECIMAL(10,2)) AS result_roun
```

Results | Messages

	result_round
1	752.88

- CEIL/ CEILING
  - Returns the smallest integer greater than or equal to a number.
  - It rounds the number, away form zero, always up.

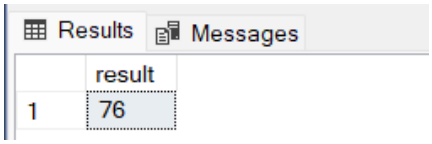
```
SELECT CEILING(76.6) AS result;
```

Results | Messages

	result
1	77

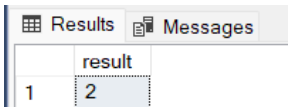
- FLOOR
  - Returns the largest integer less than or equal to a number.
  - It rounds the number, towards zero, always down.

```
SELECT FLOOR(76.6) AS result;
```



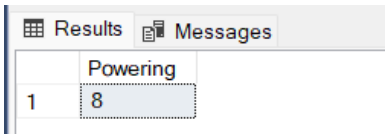
- MOD
  - Returns the remainder (modulus) of a division operation.
  - In **SQL Server**, there is no built-in MOD() function like in MySQL or Oracle — instead, you use the **modulus operator** %.

```
SELECT 11 % 3 AS result;
```



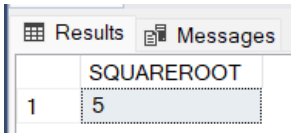
- POWER
  - This function raises a number to a specified power i.e. m^n.

```
SELECT POWER(2,3) AS Powering;
```



- SQRT
  - This function returns the square root of a number (√n)

```
SELECT SQRT(25) AS SQUAREROOT;
```



- RAND
  - It returns a random number between 0(inclusive) and 1(exclusive).

```
SELECT RAND()
```

Results | Messages

	(No column name)
1	0.529616001527694

- Date & Time Functions
  - CURRENT_DATE
    - Returns current date.
    - In SQL Server, there's no CURRENT_DATE keyword — instead, you can use:
    - Current date (no time):

```
SELECT CAST(GETDATE() AS DATE) AS current_date;
```

Example output:

```
current_date
------------
2025-09-22
```

      - GETDATE() returns the current date & time.
      - CAST(... AS DATE) strips off the time part.

  - CURRENT_TIME
    - Returns current time.
    - Current time (no date):

```
SELECT CAST(GETDATE() AS TIME) AS currentTime;
```

Results | Messages

	currentTime
1	11:11:13.3966667

  - CURRENT_TIMESTAMP
    - Return current timestamp.
    - Current timestamp (date + time):

```
SELECT CURRENT_TIMESTAMP AS currentTimestamp;  -- same as GETDATE()
```

Results | Messages

	currentTimestamp
1	2025-09-22 11:12:28.270

Date and Time Functions

- **YEAR**
  - Extracts the year from date or a datetime expression.
- **MONTH**
  - Extracts the month from date or a datetime expression.
- **DAY**
  - Extracts the day from date or a datetime expression.

```sql
SELECT
    YEAR(GETDATE())  AS current_year,
    MONTH(GETDATE()) AS current_month,
    DAY(GETDATE())   AS current_day;
```

Results | Messages

	current_year	current_month	current_day
1	2025	9	22

Date and Time Functions

- **HOUR**
  - Extracts the hour from time or a datetime expression.
- **MINUTE**
  - Extracts the minute from time or a datetime expression.
- **SECOND**
  - Extracts the second from time or a datetime expression.

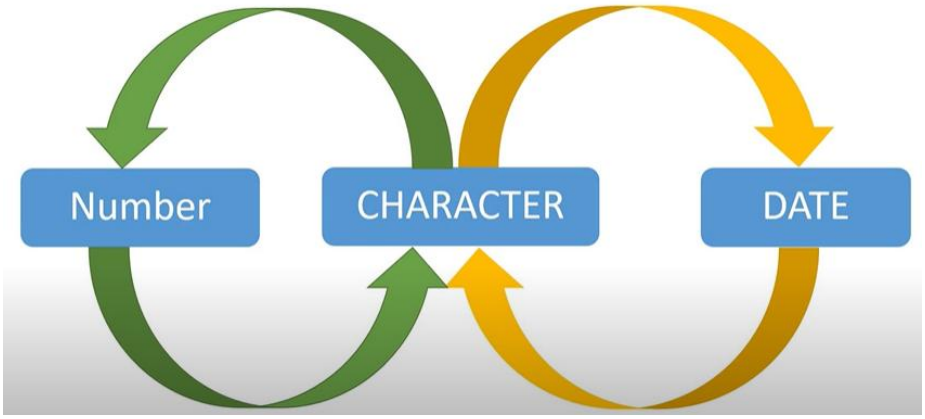
```sql
SELECT
    StudentID,
    DATEPART(HOUR,   CAST(EnrollmentDate AS DATETIME)) AS enrollment_hour,
    DATEPART(MINUTE, CAST(EnrollmentDate AS DATETIME)) AS enrollment_minute,
    DATEPART(SECOND, CAST(EnrollmentDate AS DATETIME)) AS enrollment_second
FROM Students;
```

Results Messages

	StudentID	enrollment_hour	enrollment_minute	enrollment_second
1	1	0	0	0
2	2	0	0	0
3	3	0	0	0
4	4	0	0	0
5	5	0	0	0
6	6	NULL	NULL	NULL
7	7	0	0	0

CONVERSION FUNCTIONS :

➔ **Conversion functions that allow you to explicitly convert data from one type to another.**



Conversion functions

- **CAST**
  - **The CAST() function converts a value (of any type) into the specified datatype.**

```
SELECT CAST('2025-09-22' AS DATE) AS converted_date;
SELECT CAST(123.456 AS INT) AS converted_int;
```

Results Messages

	converted_date
1	2025-09-22

	converted_int
1	123

- **CONVERT**
  - **This function converts a value into the specified datatype or character set.**

```
-- Convert string to date with U.S. format (mm/dd/yyyy)
SELECT CONVERT(DATE, '09/22/2025', 101) AS us_date;

-- Convert datetime to string in ISO format
SELECT CONVERT(VARCHAR(10), GETDATE(), 120) AS iso_date;
```

Results Messages

	us_date
1	2025-09-22

	iso_date
1	2025-09-22

SET Operators

➔ **Set operators allow you to combine the results of two or more SELECT (queries) statements into a single result set based on set theory principles.**

➔ **These operators allow you to perform operations such as combining, intersecting, or subtracting sets of data**

- **UNION**
  - **Purpose:**
    - **Combines the results of two or more SELECT queries into a single result set, removing duplicates.**

Rules:

- Each `SELECT` must have the same number of columns.
- Corresponding columns must have compatible data types.

```
SELECT StudentName FROM Students
UNION
SELECT InstructorName FROM Enrollments;
```

Results Messages

	StudentName
1	Dr. Anjali Menon
2	Dr. Kavitha Rao
3	Dr. Meera Reddy
4	Dr. Neha Sharma
5	Kasi
6	Kiran
7	MAMA
8	Ms. Sneha Kulkarni
9	Prof. Arjun Verma
10	Ravi Kumar

- **EXCEPT or MINUS**
  - **Purpose: Returns rows from the first query that are not present in the second query.**

```
SELECT StudentName FROM Students
EXCEPT
SELECT InstructorName FROM Enrollments;
```

Results | Message

	StudentName
1	Kasi
2	Kiran
3	MAMA
4	Ravi Kumar
5	Rohin
6	Roudy
7	Surya

- **UNION ALL**
  - **Purpose: Same as UNION, but keeps duplicates.**

```
SELECT StudentName FROM Students
UNION ALL
SELECT InstructorName FROM Enrollments;
```

Results | Messages

	StudentName
1	Kasi
2	Surya
3	Ravi Kumar
4	Kiran
5	Rohin
6	Roudy
7	MAMA
8	Dr. Meera Reddy
9	Prof. Arjun Verma
10	Dr. Kavitha Rao

- **INTERSECT**
  - **Purpose: Returns only the rows that exist in both result sets.**

```
SELECT StudentName FROM Students
INTERSECT
SELECT InstructorName FROM Enrollments;
```

🔹 **Quick Comparison Table**

Operator	Keeps Duplicates?	Returns...
`UNION`	❌ No	All unique rows from both queries
`UNION ALL`	✅ Yes	All rows from both queries
`EXCEPT`	❌ No	Rows in first query but not in second
`INTERSECT`	❌ No	Rows common to both queries

SQL JOINS

➔ **SQL's capability to join tables on-the-fly within data retrieval queries is a powerful feature.**
➔ **Joins are one of the most important operations in SQL SELECT queries.**

Reasons :

➔ **Repeating the information for each record is a waste of time and storage space.**
➔ **If any information changes, you would need to update every occurrence of that information.**
➔ **When data is repeated, there is a high likelihood that the data will not be entered identically each time. Inconsistent data is extremely difficult to use in reporting.**

FOREIGN KEY :

➔ **A foreign key is a column or a set of columns in a table that establishes a link or relationship to the primary key or another table.**
➔ **It ensures referential integrity, which means that the values in the foreign key column(s) must correspond to the values in the primary key column(s) of the referenced table.**
➔ **This constraint helps in maintaining the integrity and consistency of the data across related tables.**

What Are JOINS ?

➔ **A SQL Join statement is used to combine data or rows from two or more tables based on a common field between them.**

Different Types of Join

- **INNER JOIN**
  - **An inner join is a type of SQL join that combines rows from two or more tables based on a specified condition.**
  - **It returns only the rows where the join condition is true, meaning there is a match between the values in the columns being compared.**

SYNTAX :

```
SELECT column_name(s)
FROM table1
INNER JOIN table2
ON table1.column = table2.column
WHERE condition
ORDER BY column _name(s);
```

```
-- 🔍 1. INNER JOIN – Only matching records from both tables
SELECT
    s.StudentID,
    s.StudentName,
    e.CourseName,
    e.InstructorName
FROM
    Students s
INNER JOIN
    Enrollments e ON s.StudentID = e.StudentID;
```

Results | Messages

	StudentID	StudentName	CourseName	InstructorName
1	1	Kasi	Introduction to Computer Science	Dr. Meera Reddy
2	2	Surya	Data Structures and Algorithms	Prof. Arjun Verma
3	3	Ravi Kumar	Database Management Systems	Dr. Kavitha Rao
4	1	Kasi	Web Development Fundamentals	Ms. Sneha Kulkarni
5	4	Kiran	AI for Beginners	Dr. Neha Sharma

Inner Join with Multiple tables

SYNTAX :

```
SELECT column_name(s)
FROM table1
INNER JOIN table2
ON table1.column = table2.column
INNER JOIN table3
ON table1.column = table3.column
INNER JOIN tableN.....
WHERE condition
ORDER BY column _name(s);
```

```
SELECT
    s.StudentID,
    s.StudentName,
    e.EnrollmentID,
    e.CourseCode,
    c.CourseName,
    c.Credits,
    c.Department,
    e.InstructorName,
    e.EnrollmentDate
FROM Students s
INNER JOIN Enrollments e
    ON s.StudentID = e.StudentID
INNER JOIN Courses c
    ON e.CourseCode = c.CourseCode;
```

Results | Messages

	StudentID	StudentName	EnrollmentID	CourseCode	CourseName	Credits	Department	InstructorName	EnrollmentDate
1	1	Kasi	101	CS101	Introduction to Computer Science	4	Computer Science	Dr. Meera Reddy	2025-08-01
2	2	Surya	102	CS201	Data Structures and Algorithms	4	Computer Science	Prof. Arjun Verma	2025-08-03
3	3	Ravi Kumar	103	CS301	Database Management Systems	3	Computer Science	Dr. Kavitha Rao	2025-08-05
4	1	Kasi	104	WD101	Web Development Fundamentals	3	Information Technology	Ms. Sneha Kulkarni	2025-08-07
5	4	Kiran	107	AI101	AI for Beginners	3	Artificial Intelligence	Dr. Neha Sharma	2025-08-20

Q) "Write an SQL query to retrieve the Student ID, Student Name, Fees, along with their Course Name and Instructor Name, sorted by Student ID, using the tables Students, Enrollments, and Courses." ?

```
SELECT
    s.StudentID,
    s.StudentName,
    s.Fees,
    c.CourseName,
    e.InstructorName
FROM Students s
INNER JOIN Enrollments e
    ON s.StudentID = e.StudentID
INNER JOIN Courses c
    ON e.CourseCode = c.CourseCode
ORDER BY s.StudentID;
```

Results | Messages

	StudentID	StudentName	Fees	CourseName	InstructorName
1	1	Kasi	50000.00	Introduction to Computer Science	Dr. Meera Reddy
2	1	Kasi	50000.00	Web Development Fundamentals	Ms. Sneha Kulkarni
3	2	Surya	45000.00	Data Structures and Algorithms	Prof. Arjun Verma
4	3	Ravi Kumar	60000.00	Database Management Systems	Dr. Kavitha Rao
5	4	Kiran	40000.00	AI for Beginners	Dr. Neha Sharma

Q) "Write an SQL query to retrieve the Student ID, Enrollment ID, and calculate the total sum of course fees for each enrollment placed by the students, using the tables Students, Enrollments, and Courses."

```
SELECT
    s.StudentID,
    e.EnrollmentID,
    SUM(s.Fees) AS total_fees
FROM Students s
INNER JOIN Enrollments e
    ON s.StudentID = e.StudentID
INNER JOIN Courses c
    ON e.CourseCode = c.CourseCode
GROUP BY s.StudentID, e.EnrollmentID
ORDER BY s.StudentID, e.EnrollmentID;
```

Results | Messages

	StudentID	EnrollmentID	total_fees
1	1	101	50000.00
2	1	104	50000.00
3	2	102	45000.00
4	3	103	60000.00
5	4	107	40000.00

Q) "Write an SQL query to select the Course Code, Student Name, Course Department, Enrollment Date, and Course Fee for the student named ‘Kasi’ from the tables Students, Enrollments, and Courses."

```
SELECT
    c.CourseCode,
    s.StudentName,
    c.Department,
    e.EnrollmentDate,
    s.Fees AS CourseFee
FROM Students s
INNER JOIN Enrollments e
    ON s.StudentID = e.StudentID
INNER JOIN Courses c
    ON e.CourseCode = c.CourseCode
WHERE s.StudentName = 'Kasi';
```

Results | Messages

	CourseCode	StudentName	Department	EnrollmentDate	CourseFee
1	CS101	Kasi	Computer Science	2025-08-01	50000.00
2	WD101	Kasi	Information Technology	2025-08-07	50000.00

- **NATURAL JOIN**
- **LEFT JOIN (Left Outer Join)**
  - **A left join returns all rows from the left (or first) table and the matching rows from the right(or second) table.**
  - **If there is no match, NULL values are returned for the columns form the right table.**

SYNTAX :

```
SELECT column_name(s)
FROM table1 LEFT JOIN table2
ON table1.column = table2.column
WHERE condition
ORDER BY column _name(s);
```

```
--🡸 2. LEFT JOIN – All students, even if they're not enrolled
SELECT
    s.StudentID,
    s.StudentName,
    e.CourseName,
    e.InstructorName
FROM
    Students s
LEFT JOIN
    Enrollments e ON s.StudentID = e.StudentID;
```

Results | Messages

	StudentID	StudentName	CourseName	InstructorName
1	1	Kasi	Introduction to Computer Science	Dr. Meera Reddy
2	1	Kasi	Web Development Fundamentals	Ms. Sneha Kulkarni
3	2	Surya	Data Structures and Algorithms	Prof. Arjun Verma
4	3	Ravi Kumar	Database Management Systems	Dr. Kavitha Rao
5	4	Kiran	AI for Beginners	Dr. Neha Sharma
6	5	Rohin	NULL	NULL
7	6	Roudy	NULL	NULL
8	7	MAMA	NULL	NULL

Q) "Retrieve all the Student IDs, Student Names, Fees, and their Course Names — including students who are not enrolled in any course — from the Students and Enrollments tables, where the Fees are greater than or equal to 30000." ?

```sql
SELECT
    s.StudentID,
    s.StudentName,
    s.Fees,
    c.CourseName
FROM Students s
LEFT JOIN Enrollments e
    ON s.StudentID = e.StudentID
LEFT JOIN Courses c
    ON e.CourseCode = c.CourseCode
WHERE s.Fees >= 30000
ORDER BY s.StudentID;
```

Results | Messages

	StudentID	StudentName	Fees	CourseName
1	1	Kasi	50000.00	Introduction to Computer Science
2	1	Kasi	50000.00	Web Development Fundamentals
3	2	Surya	45000.00	Data Structures and Algorithms
4	3	Ravi Kumar	60000.00	Database Management Systems
5	4	Kiran	40000.00	AI for Beginners
6	5	Rohin	55000.00	NULL
7	6	Roudy	48000.00	NULL
8	7	MAMA	52000.00	NULL

- **RIGHT JOIN (Right Outer Join)**
  - **A right join returns all rows form the right (or second) table and the matching rows form the left (or first) table.**
  - **If there is no match, NULL values are returned for the columns form the left table.**

SYNTAX :

```sql
SELECT column_name(s)
FROM table1 RIGHT JOIN table2
ON table1.column = table2.column
WHERE condition
ORDER BY column _name(s);
```

```sql
--➔ 3. RIGHT JOIN – All enrollments, even if the student record is missing
SELECT
    s.StudentID,
    s.StudentName,
    e.CourseName,
    e.InstructorName
FROM
    Students s
RIGHT JOIN
    Enrollments e ON s.StudentID = e.StudentID;
```

Results | Messages

	StudentID	StudentName	CourseName	InstructorName
1	1	Kasi	Introduction to Computer Science	Dr. Meera Reddy
2	2	Surya	Data Structures and Algorithms	Prof. Arjun Verma
3	3	Ravi Kumar	Database Management Systems	Dr. Kavitha Rao
4	1	Kasi	Web Development Fundamentals	Ms. Sneha Kulkarni
5	4	Kiran	AI for Beginners	Dr. Neha Sharma
6	NULL	NULL	Cybersecurity Essentials	Dr. Anjali Menon

- **FULL JOIN (Full Outer Join)**
  - **Full join is a type of join that combines the results of both left join and right join.**
  - **It returns all rows from both tables, matching rows where the join condition is met and including unmatched rows as well.**

SYNTAX :

```
SELECT column_name(s)
FROM table1 FULL JOIN table2
ON table1.column = table2.column
WHERE condition
ORDER BY column _name(s);
```

```
--4. FULL OUTER JOIN – All students and all enrollments
SELECT
    s.StudentID,
    s.StudentName,
    e.CourseName,
    e.InstructorName
FROM
    Students s
FULL OUTER JOIN
    Enrollments e ON s.StudentID = e.StudentID;
```

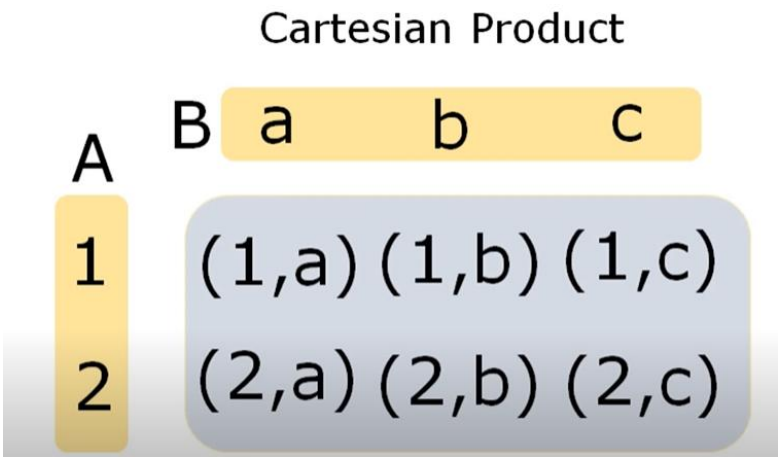
Results | Messages

	StudentID	StudentName	CourseName	InstructorName
1	1	Kasi	Introduction to Computer Science	Dr. Meera Reddy
2	1	Kasi	Web Development Fundamentals	Ms. Sneha Kulkarni
3	2	Surya	Data Structures and Algorithms	Prof. Arjun Verma
4	3	Ravi Kumar	Database Management Systems	Dr. Kavitha Rao
5	4	Kiran	AI for Beginners	Dr. Neha Sharma
6	5	Rohin	NULL	NULL
7	6	Roudy	NULL	NULL
8	7	MAMA	NULL	NULL
9	NULL	NULL	Cybersecurity Essentials	Dr. Anjali Menon

- **SELF JOIN**
- **CROSS JOIN**
    - **In SQL, a cross join, also known as a Cartesian join, is a join operation that produces the Cartesian product of two or more tables.**
    - **It combines each row from the first table with every row from the second table, resulting in a new table with a combination of all possible pairs of rows.**

Cartesian Product

- **The Cartesian Product in mathematics is the set of all possible combinations of elements from two or more sets, resulting in ordered pairs or tuples.**



1️⃣ Example: CROSS JOIN Students × Courses

```
SELECT
    s.StudentID,
    s.StudentName,
    c.CourseCode,
    c.CourseName
FROM Students s
CROSS JOIN Courses c
ORDER BY s.StudentID, c.CourseCode;
```

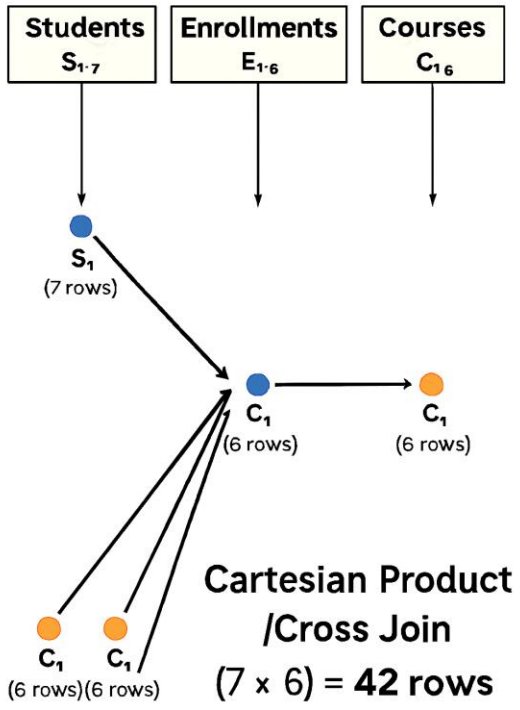
Results Messages

	StudentID	StudentName	CourseCode	CourseName
1	1	Kasi	AI101	AI for Beginners
2	1	Kasi	CS101	Introduction to Computer Science
3	1	Kasi	CS201	Data Structures and Algorithms
4	1	Kasi	CS301	Database Management Systems
5	1	Kasi	CS401	Cybersecurity Essentials
6	1	Kasi	WD101	Web Development Fundamentals
7	2	Surya	AI101	AI for Beginners
8	2	Surya	CS101	Introduction to Computer Science
9	2	Surya	CS201	Data Structures and Algorithms
10	2	Surya	CS301	Database Management Systems
11	2	Surya	CS401	Cybersecurity Essentials
12	2	Surya	WD101	Web Development Fundamentals
13	3	Ravi Kumar	AI101	AI for Beginners
14	3	Ravi Kumar	CS101	Introduction to Computer Science
15	3	Ravi Kumar	CS201	Data Structures and Algorithms
16	3	Ravi Kumar	CS301	Database Management Systems
17	3	Ravi Kumar	CS401	Cybersecurity Essentials
18	3	Ravi Kumar	WD101	Web Development Fundamentals
19	4	Kiran	AI101	AI for Beginners
20	4	Kiran	CS101	Introduction to Computer Science

2 Example: CROSS JOIN with all three tables

```
SELECT
    s.StudentID,
    s.StudentName,
    e.EnrollmentID,
    e.CourseCode AS EnrolledCourseCode,
    c.CourseCode AS CourseListCode,
    c.CourseName
FROM Students s
CROSS JOIN Enrollments e
CROSS JOIN Courses c
ORDER BY s.StudentID, e.EnrollmentID, c.CourseCode;
```

	StudentID	StudentName	EnrollmentID	EnrolledCourseCode	CourseListCode	CourseName
1	1	Kasi	101	CS101	AI101	AI for Beginners
2	1	Kasi	101	CS101	CS101	Introduction to Computer Science
3	1	Kasi	101	CS101	CS201	Data Structures and Algorithms
4	1	Kasi	101	CS101	CS301	Database Management Systems
5	1	Kasi	101	CS101	CS401	Cybersecurity Essentials
6	1	Kasi	101	CS101	WD101	Web Development Fundamentals
7	1	Kasi	102	CS201	AI101	AI for Beginners
8	1	Kasi	102	CS201	CS101	Introduction to Computer Science
9	1	Kasi	102	CS201	CS201	Data Structures and Algorithms
10	1	Kasi	102	CS201	CS301	Database Management Systems
11	1	Kasi	102	CS201	CS401	Cybersecurity Essentials
12	1	Kasi	102	CS201	WD101	Web Development Fundamentals
13	1	Kasi	103	CS301	AI101	AI for Beginners
14	1	Kasi	103	CS301	CS101	Introduction to Computer Science
15	1	Kasi	103	CS301	CS201	Data Structures and Algorithms
16	1	Kasi	103	CS301	CS301	Database Management Systems
17	1	Kasi	103	CS301	CS401	Cybersecurity Essentials
18	1	Kasi	103	CS301	WD101	Web Development Fundamentals
19	1	Kasi	104	WD101	AI101	AI for Beginners
20	1	Kasi	104	WD101	CS101	Introduction to Computer Science



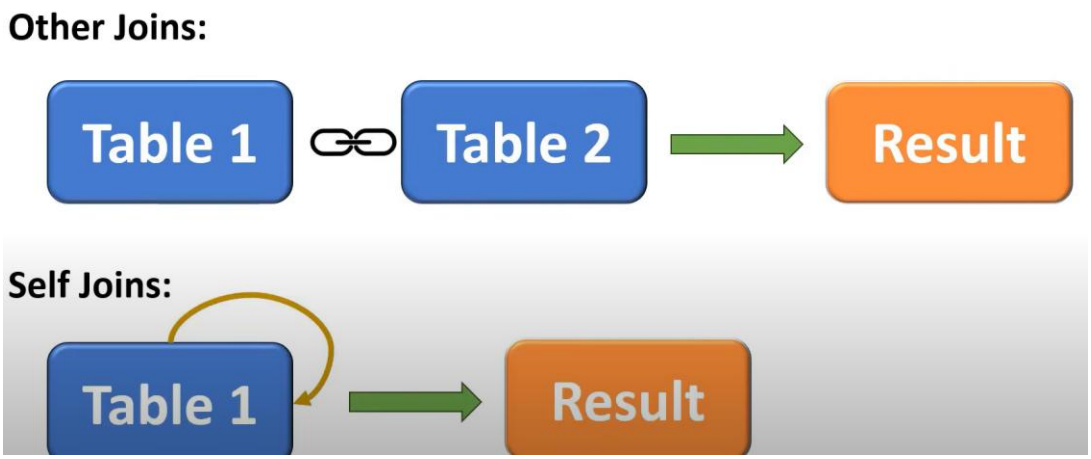
Here's the visual you asked for — it shows exactly how a CROSS JOIN (Cartesian product) expands with your tables:

- **Students table → 7 rows**
- **Courses table → 6 rows**
- **Result of Students CROSS JOIN Courses → 7 × 6 = 42 rows**

If we added Enrollments into the mix, the multiplication continues: 7 Students × 6 Enrollments × 6 Courses = 252 rows

This is why CROSS JOINs can grow very quickly — they generate all possible combinations of rows between the tables, regardless of whether they logically match.

SELF JOIN



- **Self join is a type of join operation where a table is joined with itself.**
- **It is used when you want to combine rows from the same table based on a related column.**
- **To perform a self join, you need to use table aliases to create two or more references to the same table with the SQL query.**

SYNTAX :

```
SELECT column_name(s)
FROM table_name t1 JOIN table_name t2
ON t1.column = t2.column
WHERE condition
ORDER BY column _name(s);
```

```sql
SELECT DISTINCT s1.StudentName AS Student1
       s2.StudentName AS Student2,
       s1.Role
FROM Students s1
JOIN Students s2
    ON s1.Role = s2.Role
   AND s1.StudentID <> s2.StudentID
ORDER BY s1.Role, s1.StudentName;
```

Results | Messages

	Student1	Student2	Role
1	Kasi	Rohin	CR
2	Rohin	Kasi	CR
3	Kiran	MAMA	Leader
4	Kiran	Surya	Leader
5	MAMA	Kiran	Leader
6	MAMA	Surya	Leader
7	Surya	Kiran	Leader
8	Surya	MAMA	Leader
9	Ravi Kumar	Roudy	Merit
10	Roudy	Ravi Kumar	Merit

How this works

- **Self-join: We join Students to itself (s1 and s2 are aliases).**
- **Match on Role: s1.Role = s2.Role finds students with the same role.**
- **Exclude self-matches: s1.StudentID <> s2.StudentID avoids pairing a student with themselves.**
- **DISTINCT: Prevents duplicate rows in the output.**

Q) "Write an SQL query to retrieve StudentID, StudentName, and Fees of all students whether they have a mentor or not. Also display the mentor's name for those who have a mentor, using the Students table."

```sql
SELECT
    s.StudentID,
    s.StudentName,
    s.Fees,
    m.StudentName AS MentorName
FROM Students s
LEFT JOIN Students m
    ON s.MentorID = m.StudentID;
```

Results | Messages

	StudentID	StudentName	Fees	MentorName
1	1	Kasi	50000.00	NULL
2	2	Surya	45000.00	Kasi
3	3	Ravi Kumar	60000.00	Kasi
4	4	Kiran	40000.00	Surya
5	5	Rohin	55000.00	Ravi Kumar
6	6	Roudy	48000.00	Kiran
7	7	MAMA	52000.00	NULL

Q) "Write an SQL query to retrieve the student name and the count of their direct mentees (students they mentor) for all students who have at least one mentee."

```
SELECT
    m.StudentName AS MentorName,
    COUNT(s.StudentID) AS MenteeCount
FROM Students m
JOIN Students s
    ON m.StudentID = s.MentorID
GROUP BY m.StudentName
HAVING COUNT(s.StudentID) > 0;
```

Results | Messages

	MentorName	MenteeCount
1	Kasi	2
2	Kiran	1
3	Ravi Kumar	1
4	Surya	1

- **INNER JOIN**
  - **An inner join is a type of SQL join that combines rows from two or more tables based on a specified condition.**
  - **It returns only the rows where the join condition is true, meaning there is a match between the values in the columns being compared.**

SYNTAX :

```
SELECT column_name(s)
FROM table1 INNER JOIN table2
ON table1.column = table2.column
WHERE condition
ORDER BY column _name(s);
```

SQL SUBQUERY

- ➔ **A SQL subquery is a query inside another query.**
- ➔ **The outer query in which the subquery is inserted is the main query.**
- ➔ **We can include a subquery into the SELECT, FROM, JOIN, WHERE or HAVING clause.**
- ➔ **We can also use subqueries when updating the database (i.e. in INSERT, UPDATE, and DELETE statements).**
- ➔ **ORDER BY command cannot be used in a subquery. GROUP BY command can be used to perform same function as ORDER BY command.**
- ➔ **Terms**
    - **Inner Query**
        - **Subquery used for filtering in the WHERE or HAVING clause.**

```
SELECT name
FROM employee
WHERE salary <
(SELECT AVG(salary) FROM employee);
```

    - **Nested Query**
        - **Subquery inside another subquery.**

```
SELECT * FROM employee WHERE department IN
( SELECT department FROM
        ( SELECT department, AVG(Salary) AS AverageSalary
          FROM employee
          GROUP BY department )
          AS t
WHERE t.AverageSalary > 50000 );
```

Example :

```
SELECT name
FROM employee
WHERE salary < (SELECT AVG(salary) FROM employee);
```

```
1) SELECT AVG(salary) FROM employee
   [subquery]

2) SELECT name FROM employee WHERE salary
   [Main or Outer Query]
```

NOTE : #Always write subquery inside Parenthesis()

TYPES

- **Scalar Subqueries**
  - **A scalar subquery is a subquery that returns a single value.**

  - **Mostly Aggregate Functions (COUNT, MIN, MAX, SUM, AVG) used in scalar subquery.**
  - **Scalar subquery can also be a query with expression which returns single value as output.**
  - **It is typically used in a context where a single value is expected, such as in the SELECT clause or a comparison operation (WHERE or HAVING Clause).**
  - **If the comparison operator is any of the ones in the following table, the subquery must be a single-row/scalar subquery.**

Single-row Operators

Symbol	Meaning
=	equal
>	greater than
>=	greater than or equal
<	less than
<=	less than or equal
<>	not equal
!=	not equal

Q) "Write an SQL query to retrieve the names of students whose fees are less than the average fees of all students, using a subquery." ?

```
SELECT StudentName, Fees
FROM Students
WHERE Fees < (
    SELECT AVG(Fees)
    FROM Students
);
```

Results | Messages

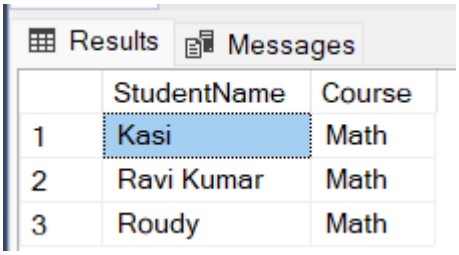
	StudentName	Fees
1	Surya	45000.00
2	Kiran	40000.00
3	Roudy	48000.00

- **Multirow Subqueries**
  - **It is a type of subquery in SQL that returns multiple rows of data.**
  - **Multi-row subqueries can return multiple rows that can be used as a set of values for comparison, filtering, or other operations in the outer query.**
  - **The operators in the following table can use multiple-row subqueries**

Symbol	Meaning
IN	equal to any member in a list
NOT IN	not equal to any member in a list
ANY	returns rows that match any value on a list
ALL	returns rows that match all the values in a list

Q) ***"Write an SQL query to retrieve the names of students who are enrolled in courses that have more than 2 students, using a multi-row subquery."***

```
SELECT StudentName, Course
FROM Students
WHERE Course IN (
    SELECT Course
    FROM Students
    GROUP BY Course
    HAVING COUNT(StudentID) > 2
);
```



	StudentName	Course
1	Kasi	Math
2	Ravi Kumar	Math
3	Roudy	Math

- **Correlated Subqueries**
  - **A correlated subquery refers to one or more column form the outer query within its own query block.**
  - **It is evaluated for each row the outer query and can be used to filter or retrieve data based on the values of the outer query.**
  - **It can be slower compared to other types of subqueries.**

Q) "Retrieve the details of those students whose fees are higher than the average fees of students in their respective courses, using a correlated subquery."

```
SELECT StudentID, StudentName, Course, Fees
FROM Students s
WHERE Fees > (
    SELECT AVG(Fees)
    FROM Students
    WHERE Course = s.Course
);
```

Results | Messages

	StudentID	StudentName	Course	Fees
1	7	MAMA	English	52000.00
2	3	Ravi Kumar	Math	60000.00
3	5	Rohin	Science	55000.00

ANY, ALL & EXISTS Operator

- **In SQL, the operators ANY, ALL, and EXISTS are used in combination with subqueries to perform comparisons and logical operations.**
-
- **ANY Operator**
  - **The ANY Operator is used in combination with comparison operators to compare a value with a set of values returned by a subquery. (Multiple-row subquery)**
  - **It returns true if the comparison holds true for at least one value in the set.**
  - **The ANY operator is typically used with the WHERE or HAVING clause to filter rows based on specific conditions.**

SYNTAX :

```
SELECT column_name(s)
FROM table_name
WHERE column_name operator ANY (Subquery);
```

The operator must be a standard comparison operator (=, <>, !=, >, >=, <, or <=)

Q) ***Retrieve all students whose fees are equal to any fees of students enrolled in the 'Math' course, using the*** **ANY** ***operator.?***

```
SELECT StudentID, StudentName, Course, Fees
FROM Students
WHERE Fees = ANY (
    SELECT Fees
    FROM Students
    WHERE Course = 'Math'
);
```

Results | Messages

	StudentID	StudentName	Course	Fees
1	1	Kasi	Math	50000.00
2	3	Ravi Kumar	Math	60000.00
3	6	Roudy	Math	48000.00

- **ALL Operator**
  - **The ALL Operator is used in combination with comparison operators to compare value with a set of values returned by a subquery (Multiple-row subquery)**
  - **It returns true if the comparison holds true for all values in the set.**
  - **The ALL operator is often used with the WHERE or HAVING clause.**

SYNTAX :

```
SELECT column_name(s)
FROM table_name
WHERE column_name operator ALL (Subquery);
```

The operator must be a standard comparison operator (=, <>, !=, >, >=, <, or <=)

Q) ***"Retrieve all students whose fees are greater than the fees of all students enrolled in the 'Science' course, using the*** **ALL** ***operator."***

```
SELECT StudentID, StudentName, Course, Fees
FROM Students
WHERE Fees > ALL (
    SELECT Fees
    FROM Students
    WHERE Course = 'Science'
);
```

Results | Messages

	StudentID	StudentName	Course	Fees
1	3	Ravi Kumar	Math	60000.00

- **EXISTS Operator**
  - **The EXISTS Operator is used to test for the existence of any record in a subquery.**
  - **It returns TRUE if the subquery returns one or more records.**
  - **It is useful when you want to conditionally retrieve data based on the existence of related records in same or another table.**
  - **EXISTS operator generally used with Correlated Subqueries.**
  - **The EXISTS operator is typically used in combination with the WHERE clause in SQL.**

SYNTAX :

```
SELECT column_name(s)
FROM table_name
WHERE EXISTS (Subquery);
```

Q) ***"Retrieve the names of students who have at least one enrollment record, using the* EXISTS *operator."***

```
SELECT DISTINCT s.StudentName
FROM Students s
WHERE EXISTS (
    SELECT 1
    FROM Enrollments e
    WHERE e.StudentID = s.StudentID
);
```

Results | Messages

	StudentName
1	Kasi
2	Kiran
3	Ravi Kumar
4	Surya

EXISTS vs IN Operator :

EXISTS Operator

1. **The EXISTS Operator is often preferred when checking for the existence of rows.**
2. **It is efficient since it only needs to find the presence of at least one row.**
3. **It is useful for checking existence based on correlated subqueries.**
4. **It is typically more performant than the IN operator when dealing with larger subquery result sets.**

IN Operator

1. **The IN operator is intuitive and easy to understand.**
2. **It is suitable when you have a small set of values to compare against.**
3. **However, using the IN operator with a large subquery result set can impact performance.**

UPDATE Command

- **The UPDATE SQL command is a data manipulation language (DML) command.**
- **This command is used to modify existing records in a database table.**
- **It allows you to change the values of one or more columns in one or multiple rows that match a specified condition.**

SYNTAX :

```
UPDATE table_name
SET column1 = value1,column2 = value2,....
WHERE condition;
```

ALTER Command

- **In SQL, the ALTER command is used to modify the structure or properties of existing database objects, such as tables, views, or indexes.**
- **It is a DDL (Data Definition language) command**

- **It allows you to make changes to the schema without the need to drop and recreate the entire object, preserving the data and other associated elements.**
- **We can use the ALTER TABLE command to**
    1. **Rename the table.**

       SYNTAX :

       ```
       ALTER TABLE old_table
       RENAME TO new_table;
       ```

    2. **Add a new column to a table**

       SYNTAX :

       ```
       ALTER TABLE table_name
       ADD column_name data_type [optional_constraints];
       ```

    3. **Rename a column to the table**

       SYNTAX :

       ```
       ALTER TABLE table_name
       RENAME COLUMN old_column TO new_column
       ```

    4. **Modify an existing column's data type or constraints.**

       SYNTAX :

       ```
       ALTER TABLE table_name
       MODIFY column_name new_data_type
       [optional_constraints];
       ```

    5. **Drop a column from the table.**

       SYNTAX :

       ```
       ALTER TABLE table_name
       DROP COLUMN column_name;
       ```

DELETE Command

➔ **Delete is a Data Manipulation Language (DML) Command.**

- ➔ **It is used to delete single or multiple rows from the database table.**
- ➔ **Using "DELETE" command we can either delete all the rows in one go or can delete row one by one based on some conditions.**
- ➔ **DELETE operations are transaction-safe.**

SYNTAX :

```
DELETE FROM table_name
WHERE condition;
```

TRUNCATE Command

- ➔ **It is also a Data Definition Language (DDL) Command.**
- ➔ **It is used to delete all the rows of a relation (table) in one go.**
- ➔ **Using "TRUNCATE" command we can't delete single row as here WHERE clause is not used.**
- ➔ **Truncate operations are not transaction-safe.**

SYNTAX :

```
TRUNACTE TABLE table_name ;
            OR
TRUNACTE  table_name ;
```

Q) Why Truncate is faster than Delete?

Truncate operations drop and re-create the table, which is much faster than deleting rows one by one, particularly for large tables.

DROP Command

- ➔ **DROP is a Data Definition Language (DDL) Command.**
- ➔ **It is used to drop the whole table.**
- ➔ **Using "DROP" command we can delete the whole structure of the table.**
- ➔ **DROP command finish the existence of the whole table.**
- ➔ **DROP operations are not transaction-safe.**

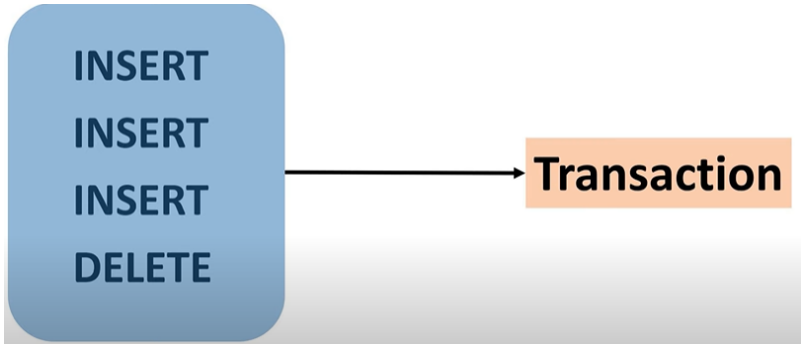
SYNTAX :

```
DROP TABLE table_name ;
```

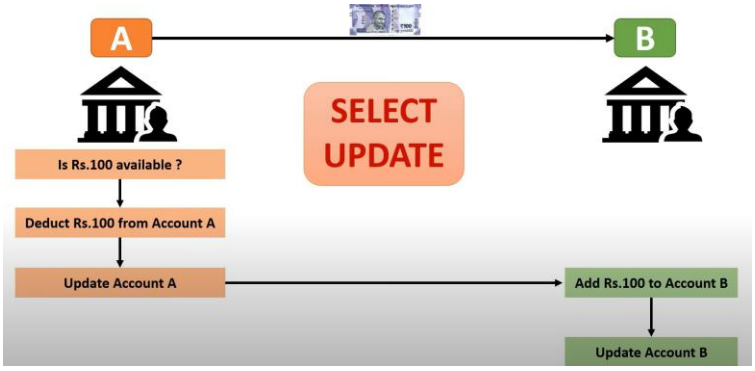
Delete	Truncate	Drop
It is a DML (Data Manipulation Language) command.	It is a DDL (Data Definition Language) command.	It is a DDL (Data Definition Language) command.
It can use the WHERE clause to filter any specific row or data from the table.	It does not use the WHERE clause to filter records from the table.	It does not use the WHERE clause to filter records from the table.
It allows us to restore the deleted data by using the COMMIT or ROLLBACK statement.	We cannot restore the deleted data after using executing this command.	We cannot restore the deleted data after using executing this command.
Schema remain the same	Schema remain the same	Schema get deleted
Its speed is slow as compare to truncate	Truncate is faster	Drop remove the complete table existence
This command eliminates records one by one.	This command deletes the entire data page containing the records.	Drop remove the complete table existence in once

TRANSACTIONAL Commands

- ➔ **Transactional commands are used to manage transactions.**
- ➔ **These are the sequences of one or more SQL statements (select, insert, update or delete) that are executed as a single unit of work**



What is Transaction?



➔ **All modification is successful when the transaction is committed.**

➔ **All modifications are undone when the transaction is rollback.**

- **Consistency**
- **Integrity**

ACID Properties

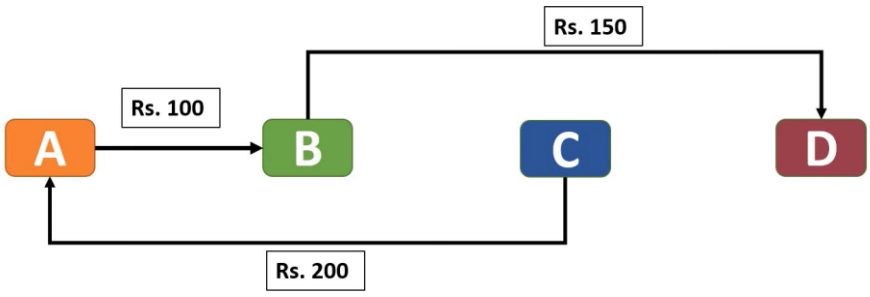
- **A - Atomicity**
  1. **Atomicity ensures that the entire transaction is treated as a single unit of work.**
  2. **Atomicity simply means that all queries in a transaction must succeed for the transaction to succeed.**
  3. **If one query fails, the entire transaction fails.**

- **C – Consistency**
  1. **Consistency ensures that the database remains in a valid state before and after the transaction.**

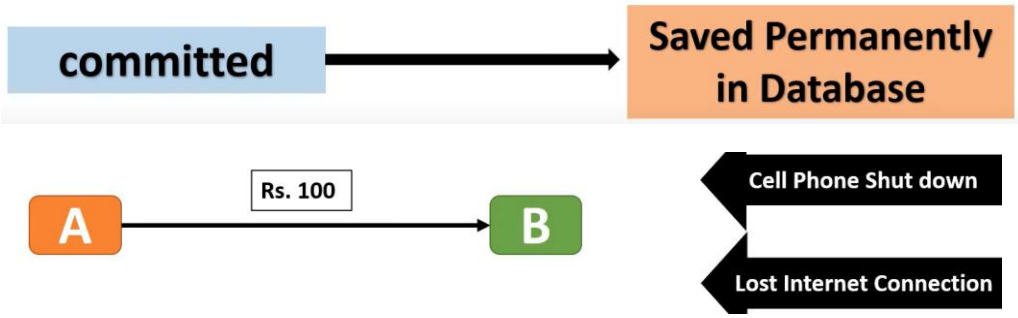
Before Transaction	
A	B
5000	3000
Total = 8000	

After Transaction	
A	B
4900	3100
Total = 8000	

- **I – Isolation**
  1. **Isolation ensures that the execution of a transaction is independent of other transactions running concurrently.**

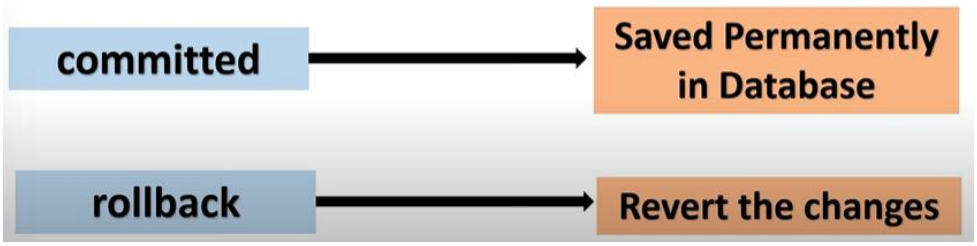


- **D – Durability**
  - **Durability ensures that once a transaction is committed, its effects are permanent and service system failures.**



START TRANSACTION

- ➔ **This command marks the beginning of a transaction.**
- ➔ **All the subsequent SQL statements/queries within the same session will be considered part of the same transaction until it is either committed or rolled back.**



SYNTAX :

```
START TRANSACTION ;
BEGIN;
```

```
BEGIN TRANSACTION ;
```

COMMIT

- ➔ **This command is used to permanently save the changes made during the current transaction.**
- ➔ **Once a transaction is committed, its changes are made permanent and cannot be rolled back.**

```sql
START TRANSACTION;
Select * from emp;
update emp set name='Saloni' where id=105;
commit;
```

ROLLBACK

- ➔ **This command is used to undo the changes made during the current transaction.**
- ➔ **It rolls back the transaction to its starting point, discarding any modifications made within the transaction.**

SYNTAX :

```
ROLLBACK ;
```

```sql
START TRANSACTION;
SELECT * from emp;
update emp set salary = 5000 where id=101;
ROLLBACK;
```

```
-- Start the transaction
START TRANSACTION;

-- Perform some SQL operations
> INSERT INTO employee (Name, department) VALUES ('Alice', 'Developer');
> UPDATE employee SET department = 'Senior Developer' WHERE Name = 'Alice';

-- If everything is correct, either commit the transaction
COMMIT;

OR

-- If there is an error, rollback the transaction
ROLLBACK;
```

AUTO COMMIT MODE

- ➔ **By Default all DDL commands are auto committed.**
- ➔ **By default, many databases operate in auto commit mode, where each individual SQL statement is automatically committed.**
- ➔ **In auto commit mode each SQL statement is transaction.**

➔ **To execute multiple statements as single transaction, we must explicitly start the transaction.**

➔ **To Disable Auto Commit Mode**

```
SET autocommit = 0;
        OR
SET autocommit OFF;
```

SAVEPOINT

➔ **This command allows you to set a point within a transaction to which you can later roll back.**

➔ **It is useful when you want to undo part of a transaction without rolling back the entire transaction.**

SYNTAX :

```
SAVEPOINT savepoint_name ;
```

ROLLBACK TO

➔ **After setting a savepoint, you can roll back to it using.**

SYNTAX :

```
ROLLBACK TO savepoint_name ;
```

```sql
START TRANSACTION;
update emp set salary = 5000 where id=101;
update emp set salary = 7000 where id=103;
select * from emp;
SAVEPOINT salary_save;
update emp set name = 'Saloni' where id=101;
ROLLBACK TO salary_save;
```

WINDOW Functions

➔ **A window function in SQL performs calculations across a set of rows related to the current row, without grouping or reducing the result set.**

➔ **It allow you to calculate things like running totals, ranks, or averages, while still keeping all the individual rows in your query results.**

1. **ROW_NUMBER()**

   - **Purpose:** Assigns a unique sequential number to each row in the result set, starting at 1 for the first row in each partition.
   - **Example:** Number students by their `Fees` in descending order.

```sql
SELECT
    StudentID,
    StudentName,
    Course,
    Fees,
    ROW_NUMBER() OVER (ORDER BY Fees DESC) AS RowNum
FROM Students;
```

Results | Messages

	StudentID	StudentName	Course	Fees	RowNum
1	3	Ravi Kumar	Math	60000.00	1
2	5	Rohin	Science	55000.00	2
3	7	MAMA	English	52000.00	3
4	1	Kasi	Math	50000.00	4
5	6	Roudy	Math	48000.00	5
6	2	Surya	Science	45000.00	6
7	4	Kiran	English	40000.00	7

How it works with your data:

- Highest fee gets RowNum = 1, next highest = 2, and so on.
- No ties — every row gets a unique number.

2. RANK()

- **Purpose:** Assigns a rank to each row within a partition, with **gaps** in ranking when there are ties.
- **Example:** Rank students by `Fees` in descending order.

```
SELECT
    StudentID,
    StudentName,
    Course,
    Fees,
    RANK() OVER (ORDER BY Fees DESC) AS FeeRank
FROM Students;
```

Results | Messages

	StudentID	StudentName	Course	Fees	FeeRank
1	3	Ravi Kumar	Math	60000.00	1
2	5	Rohin	Science	55000.00	2
3	7	MAMA	English	52000.00	3
4	1	Kasi	Math	50000.00	4
5	6	Roudy	Math	48000.00	5
6	2	Surya	Science	45000.00	6
7	4	Kiran	English	40000.00	7

How it works with your data:

- **If two students have the same fee, they get the same rank.**
- **The next rank skips numbers (e.g., 1, 2, 2, 4).**

3. DENSE_RANK()

- **Purpose:** Similar to `RANK()`, but **no gaps** in ranking when there are ties.
- **Example:** Dense rank students by `Fees` in descending order.

```
SELECT
    StudentID,
    StudentName,
    Course,
    Fees,
    DENSE_RANK() OVER (ORDER BY Fees DESC) AS DenseFeeRank
FROM Students;
```

Results | Messages

	StudentID	StudentName	Course	Fees	DenseFeeRank
1	3	Ravi Kumar	Math	60000.00	1
2	5	Rohin	Science	55000.00	2
3	7	MAMA	English	52000.00	3
4	1	Kasi	Math	50000.00	4
5	6	Roudy	Math	48000.00	5
6	2	Surya	Science	45000.00	6
7	4	Kiran	English	40000.00	7

How it works with your data:

- If two students have the same fee, they get the same rank.
- The next rank is consecutive (e.g., 1, 2, 2, 3).

4. LEAD()

- **Purpose:** Accesses data from the **next row** in the result set without using a self-join.
- **Example:** Show each student's fee and the fee of the next student when ordered by `Fees` descending.

```
SELECT
    StudentID,
    StudentName,
    Fees,
    LEAD(Fees) OVER (ORDER BY Fees DESC) AS NextStudentFee
FROM Students;
```

Results | Messages

	StudentID	StudentName	Fees	NextStudentFee
1	3	Ravi Kumar	60000.00	55000.00
2	5	Rohin	55000.00	52000.00
3	7	MAMA	52000.00	50000.00
4	1	Kasi	50000.00	48000.00
5	6	Roudy	48000.00	45000.00
6	2	Surya	45000.00	40000.00
7	4	Kiran	40000.00	NULL

How it works with your data:

- For Rohin (60000), NextStudentFee might be Kasi's 50000.
- The last row will have NULL because there's no next row.

5. LAG()

- **Purpose:** Accesses data from the **previous row** in the result set without using a self-join.
- **Example:** Show each student's fee and the fee of the previous student when ordered by `Fees` descending.

```
SELECT
    StudentID,
    StudentName,
    Fees,
    LAG(Fees) OVER (ORDER BY Fees DESC) AS PrevStudentFee
FROM Students;
```

Results | Messages

	StudentID	StudentName	Fees	PrevStudentFee
1	3	Ravi Kumar	60000.00	NULL
2	5	Rohin	55000.00	60000.00
3	7	MAMA	52000.00	55000.00
4	1	Kasi	50000.00	52000.00
5	6	Roudy	48000.00	50000.00
6	2	Surya	45000.00	48000.00
7	4	Kiran	40000.00	45000.00

How it works with your data:

- For Kasi (50000), PrevStudentFee might be Rohin's 60000.
- The first row will have NULL because there's no previous row.

Key Notes for SSMS:

- All these functions require an OVER() clause.
- You can use PARTITION BY inside OVER() to reset numbering/ranking for each group (e.g., per course).
- They **do not reduce rows** — unlike GROUP BY, you still see every row.

What PARTITION BY Does

- It **divides** your result set into groups (partitions) based on one or more columns.
- The window function (like ROW_NUMBER(), RANK(), AVG(), etc.) is then applied **within each partition** separately.
- Unlike GROUP BY, it **does not reduce rows** — you still see all rows.

1️⃣ ROW_NUMBER() with PARTITION BY

Example: Number students **within each course** by their fees (highest first).

```
SELECT
    StudentID,
    StudentName,
    Course,
    Fees,
    ROW_NUMBER() OVER (
        PARTITION BY Course
        ORDER BY Fees DESC
    ) AS RowNumInCourse
FROM Students;
```

Results Messages

	StudentID	StudentName	Course	Fees	RowNumInCourse
1	7	MAMA	English	52000.00	1
2	4	Kiran	English	40000.00	2
3	3	Ravi Kumar	Math	60000.00	1
4	1	Kasi	Math	50000.00	2
5	6	Roudy	Math	48000.00	3
6	5	Rohin	Science	55000.00	1
7	2	Surya	Science	45000.00	2

What happens:

- Each course starts numbering at 1.
- In **Math**, the highest fee student gets RowNum = 1, next = 2, etc.
- In **Science**, numbering starts again from 1.

2 RANK() with PARTITION BY

Example: Rank students **within each course** by fees, allowing gaps for ties.

```
SELECT
    StudentID,
    StudentName,
    Course,
    Fees,
    RANK() OVER (
        PARTITION BY Course
        ORDER BY Fees DESC
    ) AS FeeRankInCourse
FROM Students;
```

	StudentID	StudentName	Course	Fees	FeeRankInCourse
1	7	MAMA	English	52000.00	1
2	4	Kiran	English	40000.00	2
3	3	Ravi Kumar	Math	60000.00	1
4	1	Kasi	Math	50000.00	2
5	6	Roudy	Math	48000.00	3
6	5	Rohin	Science	55000.00	1
7	2	Surya	Science	45000.00	2

What happens:

- If two students in the same course have the same fee, they share the same rank.
- The next rank skips numbers (e.g., 1, 2, 2, 4).

3 DENSE_RANK() with PARTITION BY

- **Example:** Rank students **within each course** by fees, no gaps for ties.

```
SELECT
    StudentID,
    StudentName,
    Course,
    Fees,
    DENSE_RANK() OVER (
        PARTITION BY Course
        ORDER BY Fees DESC
    ) AS DenseFeeRankInCourse
FROM Students;
```

Results | Messages

	StudentID	StudentName	Course	Fees	DenseFeeRankInCourse
1	7	MAMA	English	52000.00	1
2	4	Kiran	English	40000.00	2
3	3	Ravi Kumar	Math	60000.00	1
4	1	Kasi	Math	50000.00	2
5	6	Roudy	Math	48000.00	3
6	5	Rohin	Science	55000.00	1
7	2	Surya	Science	45000.00	2

What happens:

- Ties get the same rank.
- The next rank is consecutive (e.g., 1, 2, 2, 3).

4 AVG() with PARTITION BY

- **Example:** Show each student's fee and the **average fee of their course**.

```
SELECT
    StudentID,
    StudentName,
    Course,
    Fees,
    AVG(Fees) OVER (
        PARTITION BY Course
    ) AS AvgFeeInCourse
FROM Students;
```

Results | Messages

	StudentID	StudentName	Course	Fees	AvgFeeInCourse
1	4	Kiran	English	40000.00	46000.000000
2	7	MAMA	English	52000.00	46000.000000
3	3	Ravi Kumar	Math	60000.00	52666.666666
4	6	Roudy	Math	48000.00	52666.666666
5	1	Kasi	Math	50000.00	52666.666666
6	2	Surya	Science	45000.00	50000.000000
7	5	Rohin	Science	55000.00	50000.000000

What happens:

- Every student in the same course sees the same average fee value for that course.
- No rows are removed — you see individual fees alongside the course average.

5 LAG() with PARTITION BY

Example: Show each student's fee and the fee of the **previous student in the same course**.

```
SELECT
    StudentID,
    StudentName,
    Course,
    Fees,
    LAG(Fees) OVER (
        PARTITION BY Course
        ORDER BY Fees DESC
    ) AS PrevFeeInCourse
FROM Students;
```

Results | Messages

	StudentID	StudentName	Course	Fees	PrevFeeInCourse
1	7	MAMA	English	52000.00	NULL
2	4	Kiran	English	40000.00	52000.00
3	3	Ravi Kumar	Math	60000.00	NULL
4	1	Kasi	Math	50000.00	60000.00
5	6	Roudy	Math	48000.00	50000.00
6	5	Rohin	Science	55000.00	NULL
7	2	Surya	Science	45000.00	55000.00

What happens:

- Looks at the previous row **within the same course**.
- First student in each course partition gets NULL for PrevFeeInCourse.

1 What is an Index?

An **index** in SQL Server is like the index in a book — it helps the database engine find rows faster without scanning the entire table.

- **Without an index**: SQL Server may have to read every row (full table scan) to find matches.
- **With an index**: SQL Server can jump directly to the relevant rows, improving performance.

2️⃣ Types of Indexes

a) Clustered Index

- Determines the **physical order** of data in the table.
- Each table can have **only one clustered index**.
- By default, if you create a **PRIMARY KEY**, SQL Server makes it clustered (unless you specify otherwise).

Example with your Students table:

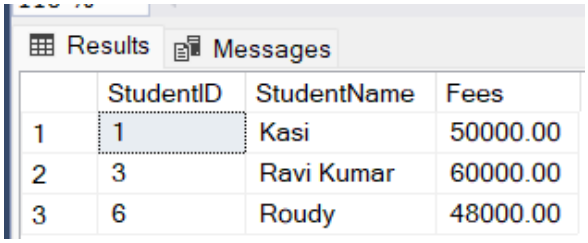
b) Non-Clustered Index

- Stores a separate structure with pointers to the actual data.
- You can have **multiple non-clustered indexes** per table.
- Great for speeding up searches on columns that are not the primary key.

Example: Searching students by Course

```
SELECT * FROM Students
CREATE NONCLUSTERED INDEX IX_Students_Course
ON Students(Course);

SELECT StudentID, StudentName, Fees
FROM Students
WHERE Course = 'Math';
```



Results | Messages

	StudentID	StudentName	Fees
1	1	Kasi	50000.00
2	3	Ravi Kumar	60000.00
3	6	Roudy	48000.00

1️⃣ What is a View?

A **View** in SQL Server is essentially a **saved query** that behaves like a virtual table.

- It **does not store data itself** — it stores the SQL definition.
- When you query a view, SQL Server runs the underlying query and returns the results.
- Views are great for:
    - Simplifying complex queries

- Hiding sensitive columns
- Providing a consistent interface to data
- Reusing logic without rewriting SQL

2️⃣ Key Points

- **Read-only by default** — you can SELECT from a view just like a table.
- You can sometimes INSERT, UPDATE, or DELETE through a view if it meets certain rules (no aggregates, no joins, etc.).
- Views can be **indexed** (Indexed Views) for performance, but that's an advanced topic.

3️⃣ Example 1 — Simple View

Goal: Show only student names and courses from the Students table.

```
CREATE VIEW vw_StudentCourses
AS
SELECT StudentName, Course
FROM Students;
```

Usage:

```
SELECT * FROM vw_StudentCourses;
```

💡 This hides other columns like Fees or Role — useful for security or clarity.

8️⃣ Why Views are useful for you

Given your focus on **interview-ready blueprints**:

- You can create views for **common joins** and **filtered datasets**.
- In interviews, you can explain that views:
  - Improve maintainability
  - Provide abstraction
  - Can be secured with permissions (grant access to view, not base table)
- You can keep your **complex logic** in views and query them simply.

1️⃣ What is a Stored Procedure?

A **Stored Procedure** is a **precompiled block of SQL code** stored in the database. Think of it as a **named script** you can run anytime without rewriting the SQL.

Key benefits:

- **Reusability** — write once, run many times.
- **Performance** — SQL Server compiles and caches the execution plan.
- **Security** — you can grant users permission to run the procedure without giving direct table access.
- **Maintainability** — change the procedure logic in one place, and all callers benefit.

Example 1 — Simple SELECT procedure

Goal: Get all students from the Students table.

```
CREATE PROCEDURE usp_GetAllStudents
AS
BEGIN
    SELECT StudentID, StudentName, Course, Fees, Role
    FROM Students;
END;
```

Results Messages

	StudentID	StudentName	Course	Fees	Role
1	1	Kasi	Math	50000.00	CR
2	2	Surya	Science	45000.00	Leader
3	3	Ravi Kumar	Math	60000.00	Merit
4	4	Kiran	English	40000.00	Leader
5	5	Rohin	Science	55000.00	CR
6	6	Roudy	Math	48000.00	Merit
7	7	MAMA	English	52000.00	Leader

Best Practices for Stored Procedures

- Prefix with usp_ (User Stored Procedure) for clarity.
- Use **parameters** to make them flexible.
- Avoid SELECT * — specify columns for performance and clarity.
- Add comments inside for maintainability.
- Keep them focused — one clear purpose per procedure.

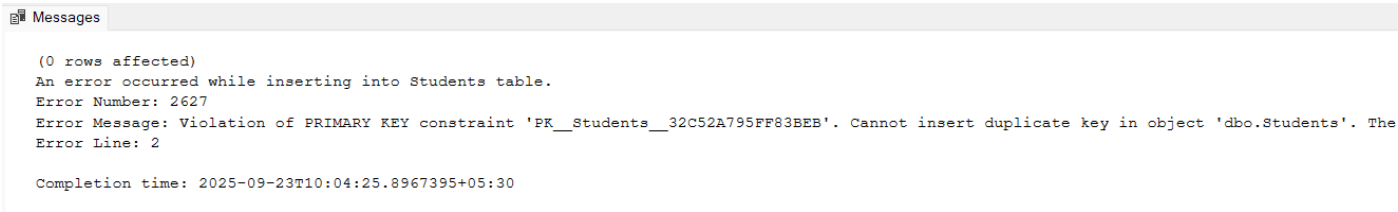
1️⃣ What is TRY…CATCH in SQL Server?

- **Purpose:** To gracefully handle errors in T-SQL code without abruptly stopping execution.
- **TRY block:** Contains the code you want to execute. If an error occurs here, control jumps to the CATCH block.
- **CATCH block:** Contains the code to handle the error — logging, showing a message, rolling back a transaction, etc.
- **Error functions inside CATCH:**
  - ERROR_NUMBER() — Returns the error number.
  - ERROR_MESSAGE() — Returns the error description.
  - ERROR_LINE() — Returns the line number where the error occurred.
  - ERROR_SEVERITY() — Returns the severity level.
  - ERROR_STATE() — Returns the state number.

Example 1 — Handling a constraint violation in Students

Scenario: We try to insert a duplicate StudentID (primary key violation).

```
BEGIN TRY
    INSERT INTO Students (StudentID, StudentName, Course, Fees, Role)
    VALUES (1, 'Duplicate Kasi', 'Math', 50000, 'CR'); -- StudentID 1 already exists
END TRY
BEGIN CATCH
    PRINT 'An error occurred while inserting into Students table.';
    PRINT 'Error Number: ' + CAST(ERROR_NUMBER() AS VARCHAR);
    PRINT 'Error Message: ' + ERROR_MESSAGE();
    PRINT 'Error Line: ' + CAST(ERROR_LINE() AS VARCHAR);
END CATCH;
```



What happens:

- The duplicate key error triggers the CATCH block.
- You'll see the error details printed instead of the query failing abruptly.

Example 2 — Handling division by zero

Scenario: We try to calculate average fees per student but divide by zero.

```sql
BEGIN TRY
    DECLARE @TotalFees DECIMAL(10,2) = (SELECT SUM(Fees) FROM Students);
    DECLARE @StudentCount INT = (SELECT COUNT(*) FROM Students WHERE Course = 'NonExistingCourse');

    DECLARE @AvgFees DECIMAL(10,2) = @TotalFees / @StudentCount; -- Will cause divide by zero
    PRINT 'Average Fees: ' + CAST(@AvgFees AS VARCHAR);
END TRY
BEGIN CATCH
    PRINT 'Error Number: ' + CAST(ERROR_NUMBER() AS VARCHAR);
    PRINT 'Error Message: ' + ERROR_MESSAGE();
END CATCH;
```

```
Messages
    Error Number: 8134
    Error Message: Divide by zero error encountered.

    Completion time: 2025-09-23T10:06:44.8844455+05:30
```

Example 3 — TRY…CATCH with Transactions

Scenario: Insert into Students and Enrollments together — rollback if either fails.

```sql
BEGIN TRY
    BEGIN TRANSACTION;

    INSERT INTO Students (StudentID, StudentName, Course, Fees, Role)
    VALUES (8, 'New Student', 'Science', 42000, 'Leader');

    -- This will fail if CourseCode doesn't exist in Courses table (FK violation)
    INSERT INTO Enrollments (EnrollmentID, StudentID, CourseCode, CourseName, InstructorName)
    VALUES (101, 8, 'INVALID_CODE', 'Science', 'Dr. Meera Reddy');

    COMMIT TRANSACTION;
END TRY
BEGIN CATCH
    ROLLBACK TRANSACTION;
    PRINT 'Transaction rolled back due to error.';
    PRINT 'Error Number: ' + CAST(ERROR_NUMBER() AS VARCHAR);
    PRINT 'Error Message: ' + ERROR_MESSAGE();
END CATCH;
```

```
Messages

  (1 row affected)

  (0 rows affected)
  Transaction rolled back due to error.
  Error Number: 2627
  Error Message: Violation of PRIMARY KEY constraint 'PK__Enrollme__7F6877FB62656848'. Cannot insert duplicate key in object 'dbo.Enrollments'. The duplicate key value is (101).

  Completion time: 2025-09-23T10:08:32.0024552+05:30
```

What happens:

- If the second insert fails, the first insert is rolled back — keeping your data consistent.

Why TRY…CATCH is important for you

Given your focus on **interview-ready blueprints**:

- You can demonstrate **robust error handling** in SQL Server.
- You can show how to **log errors** and **maintain data integrity** with transactions.
- You can adapt these examples to **real-world scenarios** like payment processing, student enrollment validation, etc.