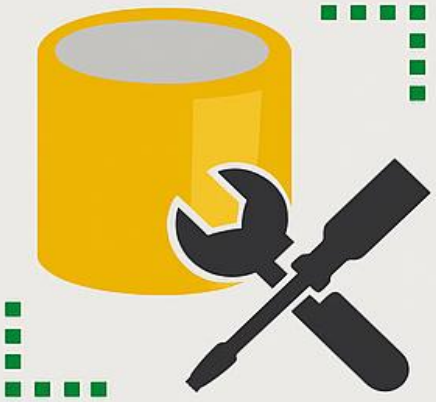

MS SQL NOTES

AMIT KUMAR PRASAD

What is RDBMS?

RDBMS stands for Relational Database management System.

It is a software that:

1. Stores data in tables (Rows and Columns).
2. Maintains relationships between different tables.
3. Allows easy access, management, and manipulation of data using **SQL (Structured Query Language).**

Features of RDBMS:

1. Data is stored in **tabular form**
2. Uses **Primary Keys (Har rows ko unique banata hai)** and **Foreign Keys (Do tables ke relation banata hai)** to maintain relationships
3. Supports **ACID properties (Data safe rahe, reliable rahe)** for data reliability:

- **Atomicity** (All or nothing)
- **Consistency** (Valid state of data)
- **Isolation** (Multiple transactions don't affect each other)
- **Durability** (Data remains even after crash)

4. Supports **Data Integrity** (ensures accuracy and consistency)
5. Allows **multiple users** and **security** features

Examples of RDBMS:

Microsoft SQL Server, **MySQL**, **Oracle**, **PostgreSQL**, **SQLite.**

What is a Database?

A Database is a collection of related data that is organized to be easily accessed, managed, and updated.

- Data is stored in tables.
- Tables contain records (rows) and fields (columns).
- Databases are used in various applications, like banking, schools, hospitals, websites, etc.

Difference between DBMS and RDBMS

Feature	DBMS	RDBMS
Data Structure	File system/ Tables	Only Tables (with relations)
Relationships	❌ Not supported	✔ Supported via Keys
Data Redundancy	High	Low (Keys reduce repetition)
Data Security	Basic	High (Multiple levels)
Integrity Constraints	Not enforced	Enforced (Primary/ Foreign Keys)
Normalization	Not supported	Supported
Examples	MS Access, XML, File based DB	MS SQL, MySQL, Oracle, etc.

How tables are related in RDBMS?

Tables in RDBMS are related using:

- **Primary Key (PK):** Unique identifier for each record in a table.
- **Foreign Key (FK):** A field that links to the primary key in another table.

Advantages of RDBMS

1. **Data Integrity**
   Maintains data accuracy and consistency using constraints like primary Keys and Foreign Keys.
2. **Data Security**
   Provides multiple layers of user access control, permissions, and roles.
3. **Data Redundancy is reduced**
   Same data is not repeated across tables due to **relationships** between tables.
4. **Data Sharing**
   Multiple users can access and modify data simultaneously.
5. **ACID Properties**
   Guarantees reliable transactions: Atomicity, Consistency, Isolation, Durability.
6. **Data Independence**
   Application and data are separated, so changing the database doesn't affect the application directly.
7. **Easy Querying**

Uses SQL (Structured Query Language) to access and manipulate data in an easy and flexible way.

Disadvantages Of RDBMS

1. **Complexity**
   Designing a relational database is **more complicated** than flat files or spreadsheets.
2. **Cost**
   Some RDBMS software (like Oracle, MS SQL Server Enterprise) is **expensive** for commercial use.
3. **Hardware Requirement**
   Requires **powerful hardware** and good storage for large-scale systems.
4. **Performance Issues**
5. With very large databases and many joins, **performance may slow down** if not optimized properly.

Microsoft SQL (SSMS)
Complete Guide

CREATE AND USE DATABASE

✓ **SYNTAX**

```
-- Create a database
CREATE DATABASE Google;
-- Connect to the database
USE Google;
```

❖ **CONCEPTS**

1. **CREATE DATABASE:** Makes a new database.
2. **USE:** Switches to that database for further operations.

❖ **TIPS**

1. End statements with **' ; '** for clarity.
2. Use clear, relevant names.
3. After connecting, you can start creating tables and adding constraints.

✓ **SYNTAX**

```
-- Connect to the database
USE Google;
-- Create the Employee table
CREATE TABLE Employee (
  Id INT,
  Name VARCHAR(255),
  Age INT,
  Gender VARCHAR(255),
  Salary INT,
  Department VARCHAR(255),
  City VARCHAR(255)
);
-- View all records in the table
SELECT * FROM Employee;
```

❖ **CONCEPTS**

1. **CREATE TABLE:** Defines a new table with specified columns and data types.
2. **SELECT** *: Retrieves all rows and columns from the table.

SQL DATATYPES

Column	Data Type	Description
Id	INT	Integer number, typically used for IDs
Name	VARCHAR(255)	Variable-length string, max 255 characters
Age	INT	Integer value for age
Gender	VARCHAR(255)	Text field for gender
Salary	INT	Integer value for salary
Department	VARCHAR(255)	Text field for department name
City	VARCHAR(255)	Text field for city name

SQL INSERT COMMAND

✓ SYNTAX

```
-- Use the target database
USE Google;
-- View current employee records
SELECT * FROM employee;
-- Insert individual employee records
INSERT INTO employee VALUES
(1, 'Amit', 26, 'Male', 40000, 'IT', 'Delhi');
-- Insert multiple employee records in one go
INSERT INTO employee VALUES
(2, 'Badal', 35, 'Male', 65000, 'Accounts', 'Jaipur'),
(3, 'Priya', 40, 'Female', 70000, 'Marketing', 'Surat'),
(4, 'Arun', 39, 'Male', 89000, 'IT', 'Chennai'),
-- Insert records with partial column data (missing age, salary, dept, city)
INSERT INTO employee (id, name, gender) VALUES
(5, 'Vivek', 'Male'),
(6, 'Shalini', 'Female'),
-- Final check of all employee records
SELECT * FROM employee;
```

❖ CONCEPTS

1. **INSERT INTO employee VALUES (...)** adds full records with all columns.
2. The multi-row insert saves time and improves readability.
3. **INSERT INTO employee (id, name, gender)** adds partial data — useful when other columns have defaults or allow NULL.

SQL UPDATE COMMAND

✓ SYNTAX

```
--Update age for a specific employee
UPDATE employee
SET age = 27
WHERE name = 'Amit';
--Update salary for employee with ID 2
UPDATE employee
SET salary = 55000
WHERE id = 2;
--Update multiple fields for employee 'Vivek'
UPDATE employee
SET age = 27, salary = 40000, department = 'HR', city = 'Delhi'
WHERE name = 'Vivek';
--Assign 'Accounts' to employees with missing department
UPDATE employee
SET department = 'Accounts'
WHERE department IS NULL;
```

❖ CONCEPT

1. **UPDATE ... SET ... WHERE ...** modifies specific rows based on conditions.
2. You can update **single** or **multiple columns** in one query.
3. **WHERE** clause ensures only the intended rows are affected.
4. **IS NULL** is used to target rows with missing values.

SQL SELECT QUERY

✓ SYNTAX

```
--Use the target database
USE Google;
--View all employee records
SELECT * FROM employee;
--Select specific columns
SELECT id, name, salary FROM employee;
--Filter by department
SELECT * FROM employee
WHERE department = 'IT';
--Filter by city and gender
SELECT name, city FROM employee
WHERE city = 'Delhi' AND gender = 'Female';
--Employees with salary above 60000
SELECT name, salary FROM employee
WHERE salary > 60000;
```

```
--Employees older than 35
SELECT name, age FROM employee
WHERE age > 35;
--Count total employees
SELECT COUNT(*) AS total_employees FROM employee;
--Average salary by department
SELECT department, AVG(salary) AS avg_salary
FROM employee
GROUP BY department;
--Number of employees per city
SELECT city, COUNT(*) AS employee_count
FROM employee
GROUP BY city;
--Sort by salary descending
SELECT name, salary FROM employee
ORDER BY salary DESC;
```

❖ CONCEPT

1. **SELECT** * shows all columns; use specific columns for cleaner output.
2. **WHERE** filters rows based on conditions.
3. **GROUP BY** helps in aggregation like **AVG, COUNT**.
4. **ORDER BY** sorts results — **ascending by default**, use **DESC** for descending.

SQL TOP

❖ SYNTAX

```
-- Use the target database
USE Google;
--Get top 5 highest-paid employees
SELECT TOP 5 name, salary
FROM employee
ORDER BY salary DESC;
--Get top 3 youngest employees
SELECT TOP 3 name, age
FROM employee
ORDER BY age ASC;
--Get top 1 employee from HR department
SELECT TOP 1 *
FROM employee
WHERE department = 'HR'
ORDER BY salary DESC;
--Get top 10 employees sorted by name
SELECT TOP 10 id, name
FROM employee
ORDER BY name ASC;
```

❖ CONCEPT

1. **TOP N** returns only **the first N rows** from the result set.
2. Combine with **ORDER BY** to control which rows are considered "**top**".

3. Useful for dashboards, previews, or performance tuning.
4. NOTE : **TOP 10** and **TOP (10)** both are same.

SQL DISTINCT QUERY

✓ **SYNTAX**

```
-- Use the target database
USE Google;
--View current employee records
SELECT * FROM employee;
--Insert a new employee record
INSERT INTO employee VALUES (15, 'Naveen', 26, 'Male', 26000, 'Accounts', 'Kolkata');
--View updated employee records
SELECT * FROM employee;
--Remove duplicate rows (if any) using DISTINCT
SELECT DISTINCT * FROM employee;
```

❖ **CONCEPT**

1. **SELECT DISTINCT** removes duplicate rows from the result set.
2. It compares all columns in the **SELECT** clause.
3. If every row is unique (like with different ids), **DISTINCT** will behave like a regular SELECT.

SQL DELETE QUERY

✓ **SYNTAX**

```
--Use the database
USE Google;
--View emp table
SELECT * FROM emp;
--Delete by name and department
DELETE FROM emp WHERE name = 'Badal';
DELETE FROM emp WHERE department = 'Marketing';
--Delete top 2 'Naveen'
DELETE TOP (2) FROM emp WHERE name = 'Naveen';
--Add and delete 'John'
INSERT INTO emp VALUES (16, 'John', 23, 'Male', 45000, 'IT', 'Seattle');
DELETE TOP (6) FROM emp WHERE name = 'John';
--Delete top 1 'Shalini'
DELETE TOP (1) FROM emp WHERE name = 'Shalini';
--Final check
SELECT * FROM emp;
--Employee table operations
SELECT * FROM employee;
```

```
DELETE TOP (2) FROM employee WHERE name = 'Naveen';
SELECT * FROM employee;
```

❖ **CONCEPT**

1. **DELETE FROM ... WHERE ...** removes rows based on conditions.
2. **DELETE TOP(N)** is specific to SQL Server, useful for limiting deletions.
3. Always use **SELECT** before and after **DELETE** to verify changes.

CLONING TABLE

✓ **SYNTAX**

```
--Use the database
USE Google;
--View original table
SELECT * FROM employee;
--Clone entire table structure and data into 'EMP'
SELECT * INTO EMP FROM employee;
--View cloned table
SELECT * FROM emp;
--Clone only female employees into 'EMPF'
SELECT * INTO EMPF FROM employee WHERE gender = 'Female';
--View female-only clone
SELECT * FROM empf;
--Clone selected columns into 'EMPT'
SELECT id, name, age, gender INTO EMPT FROM employee;
--View column-specific clone
SELECT * FROM EMPT;
```

❖ **CONCEPT**

1. **SELECT * INTO new_table FROM existing_table** clones both **structure and data**.
2. You can apply **WHERE conditions** to clone **filtered data**.
3. You can specify **columns** to clone only **selected fields**.
4. This method **creates a new table** — it won't work if the target table already exists.
5. Useful for **backups**, **testing**, or **segmenting data** (e.g., by gender or department).

DELETE, TRUNCATE AND DROP

✓ **SYNTAX**

```
--Use the database
USE Google;
--View table before deletion
SELECT * FROM empf;
--DELETE specific row
DELETE FROM empf WHERE name = 'Anjali';
--Check after DELETE
```

```
SELECT * FROM empf;
--TRUNCATE entire table (removes all rows, keeps structure)
TRUNCATE TABLE empf;
--Check after TRUNCATE
SELECT * FROM empf;
--DROP table (removes structure and data)
DROP TABLE empf;
--Check after DROP (will throw error if table doesn't exist)
SELECT * FROM empf;
--Work with another table
SELECT * FROM empt;
--DELETE rows by condition
DELETE FROM empt WHERE gender = 'female';
--Check after DELETE
SELECT * FROM empt;
--TRUNCATE all rows
TRUNCATE TABLE empt;
--Check after TRUNCATE
SELECT * FROM empt;
--DROP table
DROP TABLE empt;
```

❖ **CONCEPT**

- **DELETE FROM table WHERE...**
  - Removes specific rows based on condition.
  - Can be rolled back if inside a transaction.
  - Triggers are fired.
- **TRUNCATE TABLE table_name**
  - Removes all rows quickly.
  - Cannot delete specific rows.
  - Resets identity columns.
  - Cannot be rolled back unless inside a transaction.
  - Triggers are **not** fired.
- **DROP TABLE table_name**
  - Deletes the entire table structure and data.
  - Cannot be rolled back.
  - Table becomes unavailable after execution.

DELETE, TRUNCATE AND DROP

✓ **SYNTAX**

```
--Add one column
ALTER TABLE employee ADD country VARCHAR(255);
--Check after adding 'country'
```

```
SELECT * FROM employee;
--Add multiple columns
ALTER TABLE employee ADD email VARCHAR(255), domain VARCHAR(255), Pincode INT;
--Check after adding multiple columns
SELECT * FROM employee;
--Drop a column
ALTER TABLE employee DROP COLUMN Pincode;
--Check after dropping 'Pincode'
SELECT * FROM employee;
--Update newly added column
UPDATE employee SET country = 'India';
--Final check
SELECT * FROM employee;
```

❖ **CONCEPT**

1. **ALTER TABLE ... ADD column_name datatype** adds new columns to an existing table.
2. You can add multiple columns in one statement using commas.
3. **ALTER TABLE ... DROP COLUMN column_name** removes a column permanently.
4. Always verify changes using **SELECT *** after each operation.
5. **UPDATE** is used to assign values to newly added columns.
6. **Be cautious: dropping a column deletes its data permanently**.

DROP DATABASE

✓ **SYNTAX**

```
--Create a new database
CREATE DATABASE School;
--Delete the database
DROP DATABASE School;
```

❖ **CONCEPT**

1. **CREATE DATABASE** initializes a new database container.
2. **DROP DATABASE** permanently deletes the database and all its tables and data.
3. You cannot undo a **DROP DATABASE** command — use it with extreme caution.
4. Make sure you're not connected to the database you're trying to drop.

❖ **BEST and SAFE**

✓ **SYNTAX**

```
--Switch to a different database
USE master;
-- or any other safe DEFAULT DATABSE
--Then drop
DROP DATABASE School;
```

RENAMING COLUMN

✓ **SYNTAX**

```
--Use the target database
USE Google;
--View current table
SELECT * FROM employee;
--Rename 'name' to 'Firstname'
EXEC sp_rename 'employee.name', 'Firstname', 'COLUMN';
--Rename 'salary' to 'Earnings'
--Putting EXEC is optional.
EXEC sp_rename 'employee.salary', 'Earnings', 'COLUMN';
--Rename 'Earnings' back to 'Salary'
EXEC sp_rename 'employee.Earnings', 'Salary', 'COLUMN';
--Rename 'Firstname' back to 'Name'
EXEC sp_rename 'employee.Firstname', 'Name', 'COLUMN';
--Final check
SELECT * FROM employee;
--View table structure
EXEC sp_help 'employee';
--or simply
sp_help 'employee';
```

❖ **CONCEPT**

1. **sp_rename 'table.old_column', 'new_column', 'COLUMN'** is the correct format.
2. The **'COLUMN'** keyword is **optional but recommended for clarity**.
3. You can use **EXEC or omit it**—both works.
4. **sp_help 'table_name'** shows column names, types, and constraints.
5. Renaming columns does not affect the data, only the label.

CHANGING DATATYPE OF A COLUMN

✓ **SYNTAX**

```
--Use the database
USE Google;
--View current data
SELECT * FROM employee;
--Change 'id' column from INT to VARCHAR
ALTER TABLE employee ALTER COLUMN id VARCHAR(255);
--Check updated structure
SELECT * FROM employee;
sp_help 'employee';
--Change 'id' column back to INT
ALTER TABLE employee ALTER COLUMN id INT;
--Final structure check
sp_help 'employee';
```

❖ **CONCEPT**

1. **ALTER TABLE ... ALTER COLUMN column_name new_datatype** changes the datatype of an existing column.
2. Make sure the existing data is compatible with the new datatype.
   - Example: You can't convert text to **INT** if it contains non-numeric values.
3. Use **sp_help 'table_name'** to verify column types after changes.
4. Changing datatypes may affect:
   - Constraints
   - Indexes
   - Stored procedures or views that depend on the column

TEMPORARY TABLE IN SQL

✓ **SYNTAX**

```
-- Use the target database
USE Google;
-- View existing data
SELECT * FROM employee;
-- Create a temporary table
CREATE TABLE #Lion (
   ID INT,
   location VARCHAR(255),
   Count INT
);
-- View structure of temp table
SELECT * FROM #Lion;
-- Insert sample data
INSERT INTO #Lion VALUES
(1, 'Chennai', 125),
(2, 'Bangalore', 235),
(3, 'Delhi', 425);
-- View inserted data
SELECT * FROM #Lion;
```

❖ **CONCEPT**

1. Prefix **#** indicates a local temporary table.
2. Temp tables are stored in **TempDB** under the Temporary Tables folder.
3. They are **session-specific**:
   - Automatically **deleted** when the **session ends**.
4. Useful for:
   - Storing intermediate results.
   - Simplifying complex queries.
   - Testing logic without affecting permanent tables.

❖ TEMP TABLE TYPES

Type	Prefix	Scope	Auto-Drop
Local Temp Table	#	Current session	Yes
Global Temp Table	##	All sessions	Yes (after last session ends)

SQL OPERATORS

✓ SYNTAX

```
--Use the database
USE Google;
--View full table
SELECT * FROM employee;
--AND: Both conditions must be true
SELECT * FROM employee WHERE gender = 'male' AND department = 'IT';
SELECT * FROM employee WHERE gender = 'female' AND department = 'HR';
--OR: At least one condition must be true
SELECT * FROM employee WHERE gender = 'male' OR department = 'Accounts';
SELECT * FROM employee WHERE salary = 4000 OR city = 'Delhi';
--NOT: Condition must not be true
--Method 1
SELECT * FROM employee WHERE gender != 'male';
--Method 2
SELECT * FROM employee WHERE gender <> 'male';
--Method 3
SELECT * FROM employee WHERE NOT gender = 'male';
--Exclude department
SELECT * FROM employee WHERE department != 'IT';
--Combined conditions with AND/OR
SELECT * FROM employee WHERE
(gender = 'male' AND department = 'HR')
OR
(gender = 'female' AND department = 'marketing');
--INTERSECT: Return common rows from both queries
SELECT * FROM employee WHERE salary = 40000
INTERSECT
SELECT * FROM employee WHERE gender = 'male';
--Multiple INTERSECTs
SELECT * FROM employee WHERE salary = 40000
INTERSECT
SELECT * FROM employee WHERE gender = 'male'
INTERSECT
SELECT * FROM employee WHERE id = 12;
--UNION: Combine results, remove duplicates
SELECT * FROM employee WHERE salary = 40000
UNION
SELECT * FROM employee WHERE gender = 'male';
--UNION ALL: Combine results, keep duplicates
SELECT * FROM employee WHERE salary = 40000
UNION ALL
```

```
SELECT * FROM employee WHERE gender = 'male';
```

❖ **CONCEPT**

1. **AND:** All conditions must be true.
2. **OR:** At least one condition must be true.
3. **NOT / != / <>:** Negates the condition.
4. **INTERSECT:** Returns **only rows common** to both queries.
5. **UNION:** Combines results and **removes duplicates.**
6. **UNION ALL:** Combines results and **retains duplicates.**
7. Logical operators help **filter and combine data** based on multiple criteria.

✓ **SYNTAX**

```
--Use the database
USE Google;
--View full table
SELECT * FROM employee;
--EXCEPT: Return rows from first query that are NOT in second
SELECT * FROM employee WHERE gender = 'male'
EXCEPT
SELECT * FROM employee WHERE department = 'IT';
--Greater than
SELECT * FROM employee WHERE age > 35;
--Greater than or equal to
SELECT * FROM employee WHERE age >= 35;
--Less than
SELECT * FROM employee WHERE age < 35;
--Less than or equal to
SELECT * FROM employee WHERE age <= 35;
```

❖ **CONCEPT**

EXCEPT

- Returns rows from the first query that do not exist in the second query.
- Removes duplicates by default.
- Both queries must return the same number of columns with compatible data types.

Comparison Operators

Operator	Meaning	Example
>	**Greater than**	**age > 35**

>=	Greater than or equal to	age >= 35
<	Less than	age < 35
<=	Less than or equal to	age <= 35

✓ SYNTAX

```
--Use the database
USE Google;
--View full table
SELECT * FROM employee;
--LIKE (Wildcard Operator)
--Starts with 'a'
SELECT * FROM employee WHERE name LIKE 'a%';
--Ends with 'a'
SELECT * FROM employee WHERE name LIKE '%a';
--Second letter is 'a'
SELECT * FROM employee WHERE name LIKE '_a%';
--Third letter is 'a'
SELECT * FROM employee WHERE name LIKE '__a%';
--Contains 'a' anywhere
SELECT * FROM employee WHERE name LIKE '%a%';

--Ends with 'ni'
SELECT * FROM employee WHERE name LIKE '%ni';
--IN (Multiple value match)
SELECT * FROM employee WHERE id = 2;
SELECT * FROM employee WHERE name = 'Amit';
--Match multiple IDs
SELECT * FROM employee WHERE id IN (1, 3, 4, 5, 7, 8, 10);
--Match multiple names
SELECT * FROM employee WHERE name IN ('Amit', 'Priya', 'Badal');
--Match multiple departments
SELECT * FROM employee WHERE department IN ('HR', 'Accounts');
--Match multiple cities
SELECT * FROM employee WHERE city IN ('Delhi', 'Pune', 'Bangalore', 'Jaipur');
--ORDER BY (Sorting)
--Ascending by name
SELECT * FROM employee ORDER BY name;
--Descending by name
SELECT * FROM employee ORDER BY name DESC;
--Ascending by age
SELECT * FROM employee ORDER BY age;
--Descending by age
SELECT * FROM employee ORDER BY age DESC;
--Sort by department
SELECT * FROM employee ORDER BY department;
--Sort filtered results
SELECT * FROM employee WHERE name LIKE 'a%' ORDER BY name;
--BETWEEN (Range filtering)
--Age between 35 and 39
SELECT * FROM employee WHERE age BETWEEN 35 AND 39;
--ID between 5 and 11
SELECT * FROM employee WHERE id BETWEEN 5 AND 11;
```

```
--Names alphabetically between 'Arun' and 'Neeta'
SELECT * FROM employee WHERE name BETWEEN 'Arun' AND 'Neeta';
--Names between 'Anjali' and 'Neeta' sorted
SELECT * FROM employee WHERE name BETWEEN 'Anjali' AND 'Neeta' ORDER BY name;
```

❖ **CONCEPT**

LIKE (also known as Wild Card Operator)

Pattern	Meaning
'a%'	Starts with 'a'
'%a'	Ends with 'a'
'_a%'	Second letter is 'a'
'__a%'	Third letter is 'a'
'%a%'	Contains 'a' anywhere
'%ni'	Ends with 'ni'

❖ **IN**

1. Matches **any value** from a list.
2. Cleaner than multiple OR conditions.

❖ **ORDER BY**

1. Sorts results by one or more columns.
2. **NULL values** appear **first in ASC**, **last in DESC**.

❖ **BETWEEN**

1. Filters values **within a range**, inclusive.
2. Works with **numbers**, **dates**, and **text** (alphabetical range).

SQL ALIAS

✓ **SYNTAX**

```
--Use the database
USE Google;
--View full table
SELECT * FROM employee;
--Select specific columns without alias
SELECT id, name, age, gender, salary, department FROM employee;
--Alias columns for readability
SELECT
  id,
  name AS Firstname,
  age,
  gender,
```

```
    salary AS Earning,
    department AS Domain
FROM employee;
--Alias with underscore (no spaces allowed)
SELECT
    id,
    name AS First_name,
    age,
    gender,
    salary AS Earning,
    department AS Domain
FROM employee;
```

✓ **CONCEPT**

1. Alias renames a column or table temporarily in the result set.
2. Use **AS** keyword **(optional but recommended for clarity).**
3. **Aliases cannot contain spaces** unless enclosed in square brackets or double quotes:
    a. ✅ **AS First_name**
    b. ❌ **AS First name** (invalid without brackets)
    c. ✅ **AS [First name] or AS "First name"** (valid in some SQL dialects)
4. Aliases improve readability, especially in reports or joins.

SQL GROUP BY, AGGREGATE FUNCTION

What is GROUP BY?

The GROUP BY clause in SQL is used to group rows that have the same values in specific columns it's typically used with aggregate functions to perform calculations on each group.

What is AGGREGATE FUNCTION?

AGGREGATE FUNCTIONS perform calculations on multiple rows and return a single summary value common aggregate function include:

COUNT( ), SUM( ), AVG( ), MIN( ), MAX( )

✓ **SYNTAX – basic setup**

```
USE Google;
SELECT * FROM employee;
```

✓ **AGGREGATE FUNCTIONS – also known as MATHEMATICAL FUNCTIONS**

```
-- Total salary across all employees
SELECT SUM(salary) FROM employee;
-- With alias
SELECT SUM(salary) AS Total_salary FROM employee;
```

- **CONCEPT**
  - **Aggregate functions** perform calculations on a set of values:
    - **SUM( )** -- Calculate the total sum of the data.
    - **AVG( )** -- Calculate the average of the data.
    - **MAX( )** -- It helps you to find the maximum value among all the selected data.
    - **MIN( )** -- It helps you to find the minimum value among all the selected data.
    - **COUNT( )** -- It helps in counting of data.

✓ **SYNTAX – grouping data**

```
-- Total salary by department
SELECT department, SUM(salary) AS Total_sal
FROM employee
GROUP BY department;
-- Total salary by gender
SELECT gender, SUM(salary) AS Total_salary
FROM employee
GROUP BY gender;
-- Total salary by department and gender
SELECT department, gender, SUM(salary) AS Total_salary
FROM employee
GROUP BY department, gender;
-- With ordering
SELECT department, gender, SUM(salary) AS Total_salary
FROM employee
GROUP BY department, gender
ORDER BY department;
-- Group and order by department
SELECT department, SUM(salary) AS Total_salary
FROM employee
GROUP BY department
ORDER BY department;
```

✓ **SYNTAX – common mistake and fix**

```
-- ❌ Incorrect: gender is not in GROUP BY
SELECT department, gender, SUM(salary)
FROM employee
GROUP BY department;
-- ☑ Correct: include all non-aggregated columns in GROUP BY
SELECT department, gender, SUM(salary)
FROM employee
GROUP BY department, gender;
-- View full table again
SELECT * FROM employee;
```

✓ **CONCEPT**

- ✅ **Rule**: Every column in **SELECT** that is **not** inside **an aggregate function** must be listed in **GROUP BY**.

HAVING CLAUSE

The HAVING clause in SQL is used to filter grouped data based on aggregate functions like SUM( ), AVG( ), COUNT( ), MAX( ), and MIN( ).

It's similar to the WHERE clause, but while WHERE filters individual rows, HAVING filters groups of rows after they've been aggregated

✓ **SYNTAX**

```
-- Filter grouped results using HAVING
SELECT department, SUM(salary) AS Total_salary
FROM employee
GROUP BY department
HAVING department IN ('IT', 'HR', 'Accounts');
-- With ORDER BY
SELECT department, SUM(salary) AS Total_salary
FROM employee
GROUP BY department
HAVING department IN ('IT', 'HR', 'Accounts')
ORDER BY department DESC;
-- Filter by gender after grouping
SELECT department, gender, SUM(salary) AS Total_salary
FROM employee
GROUP BY department, gender
HAVING gender = 'male';
-- Multiple conditions
SELECT department, gender, SUM(salary) AS Total_salary
FROM employee
GROUP BY department, gender
HAVING gender = 'male' OR department = 'HR';
```

✓ **CONCEPT**

1. **HAVING** filters grouped results, unlike **WHERE** which filters rows.
2. Use with **GROUP BY** and aggregate functions.
3. Supports conditions like **IN, =, >, etc**.

CONSTRAINTS

Constraints are rules applied to table columns to enforce data integrity and control the type of data that can be stored.

✓ **SYNTAX – basic setup**

```
USE google;
SELECT * FROM employee;
```

❖ **NOT NULL Constraint**

1. Ensures a column cannot have **NULL** Values.
2. Applied using **ALTER TABLE**

✓ **SYNTAX**

```
-- Make 'id' column NOT NULL
ALTER TABLE employee ALTER COLUMN id INT NOT NULL;
-- Update 'id' to a non-null value
UPDATE employee SET id = 16 WHERE name = 'Aditya';
-- Try setting 'id' to NULL (will fail if NOT NULL is active)
UPDATE employee SET id = NULL WHERE id = 1;
```

❖ **NOT NULL Constraint**

- Allows a column to store **NULL** values.

✓ **SYNTAX**

```
-- Change 'id' column to allow NULLs
ALTER TABLE employee ALTER COLUMN id INT NULL;
-- Insert a row without specifying 'id'
INSERT INTO employee (name, age, gender, salary) VALUES ('Mark', 45, 'Male', 65000);
```

❖ **Column Info**

✓ **SYNTAX**

```
-- View table structure and constraints
sp_help 'employee';
```

❖ **Changing constraints on name Column**

✓ **SYNTAX**

```
-- Make 'name' NOT NULL
ALTER TABLE employee ALTER COLUMN name VARCHAR(255) NOT NULL;
-- Make 'name' NULL again
ALTER TABLE employee ALTER COLUMN name VARCHAR(255) NULL;
```

✓ **CONCEPT**

1. Use **ALTER TABLE ... ALTER COLUMN** to change constraints.
2. **NOT NULL** ensures data integrity by preventing missing values.
3. Use **sp_help 'table_name'** to inspect column constraints.

CHECK CONSTRAINT

Ensures that values in a column meet a specific condition.

✓ **SYNTAX**

```
-- Add CHECK constraint on age
ALTER TABLE employee ADD CONSTRAINT Chk_age CHECK (age BETWEEN 22 AND 60);
-- Fix conflicting data before applying constraint
UPDATE employee SET age = 47 WHERE name = 'Ravi';
-- View table
SELECT * FROM employee;
-- Insert valid data
INSERT INTO employee VALUES (19, 'Shivansh', 20, 'Male', 45000, 'HR', 'Pune', 'India');
-- Will fail if constraint exists
-- View structure
sp_help 'employee';
-- Drop CHECK constraint
ALTER TABLE employee DROP CONSTRAINT Chk_age;
-- Insert after dropping constraint
INSERT INTO employee VALUES (19, 'Shivansh', 20, 'Male', 45000, 'HR', 'Pune', 'India');
```

❖ **CONCEPT**

1. **CHECK enforces range or condition** (e.g., age between 22 and 60).
2. Must update invalid data before applying the constraint.
3. Use **sp_help 'table_name'** to view constraints.
4. Use **ALTER TABLE DROP CONSTRAINT** to remove it.

DEFAULT CONSTRAINT

Automatically assigns a default values to a column if no value is provided during INSERT.

✓ **SYNTAX**

```
-- Add DEFAULT constraint to 'country' column
ALTER TABLE employee ADD CONSTRAINT Df_C DEFAULT 'India' FOR country;

-- Insert without specifying 'country'
INSERT INTO employee (id, name, age, gender) VALUES (21, 'Rahul', 37, 'Male');
-- 'country' will be set to 'India' automatically
-- Insert with all values (overrides default)
INSERT INTO employee VALUES (22, 'Joseph', 24, 'Male', 35000, 'Marketing', 'San Jose', 'USA');
```

❖ **CONCEPT**

1. **DEFAULT is not restrictive** — it fills in missing values, **unlike CHECK which validates** them.
2. You can still override the default by explicitly providing a value.
3. Useful for columns like country, status, created_date, etc.

UNIQUE CONSTRAINT

Ensures that all values in a column are distinct – no duplicates allowed.

✓ **SYNTAX**

```
-- Add UNIQUE constraint on 'id'
ALTER TABLE employee ADD CONSTRAINT Uni_key UNIQUE(id);
-- Fix duplicate values before applying
UPDATE employee SET id = 23 WHERE name = 'Rolf';
-- Drop UNIQUE constraint
ALTER TABLE employee DROP CONSTRAINT Uni_key;
```

❖ **CONCEPT**

1. Prevents **duplicate entries** in the specified column.
2. You must **resolve duplicates before applying** the constraint.
3. Unlike **PRIMARY KEY, UNIQUE allows NULL values** (but only one if applied to a single column).
4. Useful for columns like **email, username, employee_code**, etc.
5. **SYNTAX**

```
-- This will fail if 'id' already exists
INSERT INTO employee VALUES (22, 'Rolf', 25, 'Male', 42000, 'HR', 'Aukland', 'New Zealand');
```

PRIMARY KEY

Uniquely identifies each row in a table. Combines:

- **NOT NULL (no missing value).**
- **UNIQUE (no duplicates).**

✓ **SYNTAX**

```
-- Ensure column is NOT NULL
ALTER TABLE employee ALTER COLUMN id INT NOT NULL;
-- Add PRIMARY KEY on 'id'
ALTER TABLE employee ADD CONSTRAINT Pri_key_id PRIMARY KEY(id);

-- Drop PRIMARY KEY
ALTER TABLE employee DROP CONSTRAINT Pri_key_id;
-- Apply PRIMARY KEY on 'name'
ALTER TABLE employee ALTER COLUMN name VARCHAR(255) NOT NULL;
ALTER TABLE employee ADD CONSTRAINT Pri_name PRIMARY KEY(name);
-- Drop PRIMARY KEY on 'name'
ALTER TABLE employee DROP CONSTRAINT Pri_name;
-- Reapply PRIMARY KEY on 'id'
ALTER TABLE employee ADD CONSTRAINT Pri_key_id PRIMARY KEY(id);
```

❖ **CONCEPT**

1. Only one **PRIMARY KEY** allowed per table.
2. Column must be **NOT NULL** and unique.

3. Use **sp_help 'table_name'** to inspect constraints.
4. Commonly used on id, email, or username fields

CONSTRAINT DURING TABLE CREATION

1. SYNTAX

```
-- Create 'school' table with NOT NULL constraints
CREATE TABLE school (
   Roll_number INT NOT NULL,
   Name VARCHAR(255) NOT NULL,
   Marks INT,
   Sport VARCHAR(255)
);
-- View structure
SELECT * FROM school;
SP_HELP 'school';
-- Create 'sports' table with CHECK constraint
CREATE TABLE sports (
   SID INT NOT NULL,
   Name VARCHAR(255),
   Players INT NOT NULL,
   CONSTRAINT chkp_play CHECK (Players > 5)
);
-- View structure
SELECT * FROM sports;
SP_HELP 'sports';
```

❖ CONCEPT

1. **NOT NULL**: Prevents missing values.
2. **CHECK**: Validates data against a condition (e.g., Players > 5).
3. Constraints can be added inline or using **CONSTRAINT** keyword.
4. Invalid data (e.g., Players = 4) will terminate the insert.

❖ EXAMPLE ERROR 🚫

2. SYNTAX

```
-- Valid insert
INSERT INTO sports VALUES (1, 'Cricket', 11);
-- ❌ Invalid insert (fails due to CHECK constraint)
INSERT INTO sports VALUES (2, 'Badminton', 4);
```

FOREIGN KEY CONSTRAINT

3. SYNTAX

```
-- Add foreign key to 'project' table referencing 'employee'
ALTER TABLE project
ADD CONSTRAINT For_key FOREIGN KEY (PID) REFERENCES employee(id);
```

❖ CONCEPT

Concept	Explanation
Foreign Key	Links one table to another; ensures referenced value exists in parent table.
Duplicates Allowed	Foreign keys can have duplicate values.
Referential Integrity	Prevents deletion of parent rows if child rows exist.
Deletion Order	Must delete child rows first before deleting parent rows.
Dropping Constraints	Must drop foreign key before dropping primary key it depends on.

❖ Common Errors 🚫 & Fixes

1. **Insert fails** if foreign key value doesn't exist in parent table.
2. **Update fails** if new value violates foreign key constraint.
3. **Delete fails** if referenced in child table.

4. SYNTAX

```
-- ❌ Will fail if PID = 16 doesn't exist in employee
UPDATE project SET PID = 16 WHERE Project_name = 'JERRY';
-- ❌ Cannot delete employee with id = 7 if referenced in project
DELETE FROM employee WHERE id = 7;
-- ✅ First delete from project
DELETE FROM project WHERE PID = 7;
DELETE FROM employee WHERE id = 7;
```

❖ DROPPING CONSTRAINT

```
-- ❌ Will fail if foreign key still exists
ALTER TABLE employee DROP CONSTRAINT PRI_KEY_ID;
-- We have to ✅ Drop foreign key first
ALTER TABLE project DROP CONSTRAINT For_key;
ALTER TABLE employee DROP CONSTRAINT PRI_KEY_ID;
```


Import CSV

Method 1: Full Import using BULK INSERT

5. SYNTAX

```
USE Google;
-- Create table
CREATE TABLE Rabbit (
  id INT,
  lastname VARCHAR(255),
```

```
    firstname VARCHAR(255),
    middlename VARCHAR(255),
    suffix VARCHAR(255)
);
-- Import CSV (skip header row)
BULK INSERT dbo.Rabbit
FROM 'C:\sql file\Tiger.csv'
WITH (
    FORMAT = 'CSV',
    FIRSTROW = 2
);
-- View imported data
SELECT * FROM Rabbit;
```

Method 2: Limited Import (First to Last Row)

6. SYNTAX

```
-- Create another table
CREATE TABLE Parrot (
    id INT,
    lastname VARCHAR(255),
    firstname VARCHAR(255),
    middlename VARCHAR(255),
    suffix VARCHAR(255)
);
-- Import selected rows
BULK INSERT dbo.Parrot
FROM 'C:\sql file\Tiger.csv'
WITH (
    FORMAT = 'CSV',
    FIRSTROW = 2,
    LASTROW = 21
);
-- View imported data
SELECT * FROM Parrot;
```

Method 3: GUI Import (Flat File Wizard)

1. **Right-click** on your database → **Import Flat File.**
2. **Browse** and select your **.csv** file.
3. **Name your table.**
4. **Choose keys** (Primary/Foreign) **if needed.**
5. Click **Finish** to complete import.

KEY TAKEWAYS

- **dbo** = **Database Owner** schema.
- **FIRSTROW** = 2 skips header row.
- **LASTROW** limits how many rows are imported.
- **GUI** method is **beginner-friendly** and flexible.

EXPORTING DATA FROM SQL SERVER

Method 1: Using SQL Server Management Studio (SSMS)

1. **SYNTAX –** run this query first

```
USE Google;
SELECT * FROM employee;
```

2. **Right-click on result grid → Save Results As...**
3. **Choose:**
   - **CSV file (.csv)**
   - **Text file (.txt)**
   - **Excel-compatible CSV**
4. **Name your file** and click **Save.**

Method 2: Using SQL Server Import and Export Wizard

1. In SSMS, go to:
   - **Tasks → Export Data...** (Right-click on database).
2. Choose:
   - **Source:** SQL Server
   - **Destination:** Flat File, Excel, or another DB.
3. Select:
   - **Table or Query** (e.g., **SELECT * FROM employee**).
4. Configure file path and format.
5. Click **Finish** to export.

TIP

- For automation, use **bcp (Bulk Copy Program)** in command line:
- **SYNTAX**

```
bcp "SELECT * FROM Google.dbo.employee" queryout "C:\export\employee.csv" -c -t, -T -S localhost
```

SYSTEM DATABASES

SQL Server System Databases

Database	Purpose	Key Notes

master	Central configuration	1. Stores system-level information: login accounts, system configuration, linked servers, and metadata for all other databases. 2. Critical for startup: SQL Server cannot start without a working master database.
model	Template for new databases	1. Acts as a blueprint: any new database inherits objects and settings from model. 2. You can customize model to include default tables, views, or settings.
tempdb	Temporary workspace	1. Stores temporary tables, table variables, cursors, and intermediate results. 2. Recreated every time SQL Server restarts. 3. Performance-critical: often optimized with fast storage.
msdb	SQL Server Agent & Jobs	1. Stores job schedules, alerts, operators, and backup history. 2. Used by SQL Server Agent for automation tasks like backups and maintenance plans.

- You can view these system databases in SSMS under the "System Databases" folder.
- Avoid modifying master, model, or msdb unless you're absolutely sure—it can affect the entire SQL Server instance.

AGGREGATE FUNCTIONS

Function	Purpose	Example
SUM()	Total of values	SELECT SUM(salary) FROM employee;
MAX()	Highest value	SELECT MAX(salary) FROM employee;
MIN()	Lowest value	SELECT MIN(age) FROM employee;
AVG()	Average value	SELECT AVG(salary) FROM employee;
COUNT()	Number of rows	SELECT COUNT(*) FROM employee;

- **Note: COUNT(column)** ignores **NULL** values.

GROUPED AGGREGATE FUNCTIONS

Purpose	Example

Avg age by dept	SELECT department, AVG(age) FROM employee GROUP BY department;
Avg salary by gender	SELECT gender, AVG(salary) FROM employee GROUP BY gender;

STRING FUNCTIONS – Text Manipulation

Function	Purpose	Example
LOWER( )	To lowercase	SELECT LOWER(name) FROM employee;
UPPER( )	To uppercase	SELECT UPPER(name) FROM employee;
REVERSE( )	Flip text	SELECT REVERSE(name) FROM employee;
LEN( )	Text length	SELECT LEN(name) FROM employee;
CONCAT( )	Merge text	SELECT CONCAT(name, ' is ', age, ' yrs') FROM employee;
REPLACE( )	Swap text	SELECT REPLACE(city, 'Delhi', 'New Delhi') FROM employee;
SUBSTRING( )	Extract part	SELECT SUBSTRING(name, 1, 3) FROM employee;

- **CONCEPT**
  1. **Aggregate functions** work on columns, not rows.
  2. **GROUP BY** lets you apply aggregates per category.
  3. **String functions** help format, clean, and present data.
  4. **COUNT(*) vs COUNT(column): *** counts all rows, **column** skips **NULLs**.
  5. **CONCAT( )** can build readable sentences or clone into new tables.

USER-DEFINED FUNCTIONS

WHAT ARE USER-DEFINED FUNCTIONS?

- Custom functions created by users to **encapsulate logic**.
- Can return **scalar values** (single result) or **table results**.
- Useful for **reusability, calculations and cleaner queries**.

SCALAR FUNCTIONS EXAMPLES

❖ **Multiply 3 Numbers**

❖ **SYNTAX**

```
--Defining a function
CREATE FUNCTION Multiply3 (@a INT, @b INT, @c INT)
RETURNS INT
AS BEGIN
   RETURN @a * @b * @c
END


-- Calling a function
```

```
SELECT dbo.Multiply3(165, 455, 154);
SELECT id, age, salary, dbo.Multiply3(id, age, salary) AS Result FROM employee;
```

❖ **Simple interest function**

❖ **SYNTAX**

```
--Defining a function
CREATE FUNCTION SI (@a FLOAT, @b INT, @c INT)
RETURNS FLOAT
AS BEGIN
   RETURN (@a * @b * @c) / 100
END
-- Function calling.
SELECT dbo.SI(13, 21, 1254);
-- Implication on our data
SELECT
   name,
   salary AS Principle,
   age AS ROI,
   id AS Time,
   dbo.SI(id, age, salary) AS Interest,
   (salary + dbo.SI(id, age, salary)) AS Amount
FROM employee;
```

❖ **CONCEPT**

1. Use **CREATE FUNCTION** to define logic.
2. Use dbo.FunctionName( ) to call it.
3. Must specify **input parameters** and **return type**.
4. Use **ALTER FUNCTION** to modify and **DROP FUNCTION** to delete.

SQL JOINS

Joins combine rows from two tables based on a related column (usually a key).

Common Join types:

INNER, LEFT, RIGHT, FULL, CROSS, SELF

✓ **SYNTAX – sample tables**

```
SELECT * FROM employee;
SELECT * FROM project;
```

INNER JOIN

Returns rows with matching values in both tables.

✓ **SYNTAX**

```
SELECT * FROM employee e
INNER JOIN
project p ON e.id = p.pid;
```

LEFT JOIN

Returns all rows from left table + matched rows from right.

✓ **SYNTAX**

```
SELECT * FROM employee e
LEFT JOIN
project p ON e.id = p.pid;
```

RIGHT JOIN

Returns all rows from right table + matched rows from left.

❖ **SYNTAX**

```
SELECT * FROM employee e
RIGHT JOIN
project p ON e.id = p.pid;
```

FULL JOIN

Returns all rows from both tables, matched or not.

❖ **SYNTAX**

```
SELECT * FROM employee e
FULL JOIN project p ON e.id = p.pid;
```

CROSS JOIN

Returns Cartesian product —

every row from employee paired with every row from project.

Use with caution: **Can produce large result sets**

❖ **SYNTAX**

```
SELECT * FROM employee e
CROSS JOIN
project p;
```

Example: If employee has 5 rows and project has 5 rows → result = 25 rows.

SELF JOIN

Joins a table with itself – **useful for hierarchical or comparative data**.

Self Join uses aliases to differentiate instances.

❖ **SYNTAX**

```
SELECT
e1.id,
e1.name AS Employee,
e2.name AS Manager
FROM employee e1
```

```
JOIN
employee e2 ON e1.manager_id = e2.id;
```

❖ **Example Use Case**

Find employee and their managers from the same table.

❖ **CONCEPT TABLE**

Join Type	Description	Use Case Example
INNER JOIN	Matched rows from both tables	Employees with assigned projects
LEFT JOIN	All from left + matched from right	All employees, even if no project
RIGHT JOIN	All from right + matched from left	All projects, even if no employee
FULL JOIN	All rows from both tables	Complete view of employees & projects
CROSS JOIN	All combinations of rows	Testing combinations or permutations
SELF JOIN	Table joined with itself	Employee-manager relationships

COMPANY TABLE – SETUP & DATA INSERTION

❖ **SYNTAX -** table creation

```
USE Google;
--Create table
CREATE TABLE company (
   cid INT,
   company_name VARCHAR(255),
   headquaters VARCHAR(255)
);
```

❖ **Data Insertion**

```
INSERT INTO company VALUES
(1, 'TCS', 'Mumbai'),
(2, 'IBM', 'Sanjose'),
(3, 'Wipro', 'Noida'),
(4, 'Infosys', 'Banglore'),
(5, 'HCL', 'Noida'),
(6, 'Accenture', 'Sydney'),
(7, 'Cognizant', 'Tokyo'),
(8, 'Meta', 'Seattle'),
(9, 'Microsoft', 'Seattle');
```

❖ **View table - SYNTAX**

```
SELECT * FROM company;
```

❖ **CONCEPT**

- **cid:** Unique company **ID** (can be set as **PRIMARY KEY** if needed).

- **company_name:** Stores company names.
- **headquaters:** Stores city of headquarters.

Joining Multiple Tables — Summary

❖ CONCEPT

You can join **three or more tables using multiple INNER JOIN clauses**, linking each table through a common key.

❖ Tables Involved - SYNTAX

```
SELECT * FROM employee;
SELECT * FROM project;
SELECT * FROM company;
```

❖ Basic Multi-Table Join – SYNTAX

```
SELECT * FROM employee e
INNER JOIN
project p ON e.id = p.pid
INNER JOIN
company c ON p.pid = c.cid;
```

❖ Filtered by Headquarters - SYNTAX

```
SELECT e.id, e.name, e.gender, e.age, e.salary, e.department, p.project_name,
p.technology, p.pincode, c.company_name, c.headquater
 FROM
employee e
INNER JOIN
project p ON e.id = p.pid
INNER JOIN
company c ON p.pid = c.cid
WHERE c.headquaters = 'Seattle';
```

❖ Filtered by Gender + Ordered – SYNTAX

```
SELECT e.id, e.name, e.gender, e.age, e.salary, e.department, p.project_name,
p.technology, p.pincode, c.company_name, c.headquaters
FROM
employee e
INNER JOIN
project p ON e.id = p.pid
INNER JOIN
company c ON p.pid = c.cid
WHERE e.gender = 'Male'
ORDER BY e.name;
```

❖ Aliased & Formatted Output – SYNTAX

```
SELECT e.id AS EmpId, e.name AS FirstName, e.gender, e.age,
e.salary AS Earnings, e.department, p.project_name, p.technology AS Domain,
```

```
p.pincode, c.company_name, c.headquaters
FROM employee e
INNER JOIN
project p ON e.id = p.pid
INNER JOIN
company c ON p.pid = c.cid;
```

❖ Aggregated Join (Group By) – SYNTAX

```
SELECT p.technology, SUM(e.salary) AS Total_salary
FROM employee e
INNER JOIN
project p ON e.id = p.pid
INNER JOIN
company c ON p.pid = c.cid
GROUP BY p.technology;
```

❖ Key Concept

1. Always match keys correctly between tables.
2. Use aliases **(e, p, c)** for cleaner queries.
3. You can apply **WHERE, ORDER BY, and GROUP BY** just like in two-table joins.

SQL SUBQUERIES

A subquery is a query nested inside another query.

Used for filtering, comparisons, and ranking

❖ Basic subquery – SYNTAX

```
-- Highest salary
SELECT MAX(salary) FROM employee;
-- 2nd highest salary
SELECT MAX(salary)
FROM employee
WHERE salary < (SELECT MAX(salary) FROM employee);
```

❖ Nth Highest Salary (Nested Subqueries) – SYNTAX

```
-- 3rd highest
SELECT MAX(salary)
 FROM employee
WHERE salary < (
 SELECT MAX(salary)
 FROM employee
 WHERE salary < (
  SELECT MAX(salary) FROM employee
 )
);
-- 4th highest (same pattern, one more level)
```

❖ **Nth HIGHEST SALARY (TOP + MIN Method)**

```
-- 5th highest salary
SELECT MIN(salary)
FROM employee
WHERE salary IN (
  SELECT DISTINCT TOP(5) salary
  FROM employee
  ORDER BY salary DESC);
```

❖ **Formula**

```
SELECT MIN(salary)
FROM employee
WHERE salary IN (
  SELECT DISTINCT TOP(N) salary
  FROM employee
  ORDER BY salary DESC
);
```

❖ **CONCEPT**

- Subqueries can be used in **WHERE, FROM**, or **SELECT**.
- **TOP(N) + MIN()** is a smart way to get the Nth highest.

RANK( ) AND DENSE_RANK( )

RANK( ) and DENSE_RANK( ) are window functions used to assign ranks to rows based on a sorted column.

Function	Behaviour
RANK()	Skips ranks if there are ties (gaps in ranking)
DENSE_RANK()	No gaps; assigns consecutive ranks even if values are tied

❖ **SYNTAX and EXAMPLES**

```
-- Rank by salary (descending)
SELECT *, RANK() OVER (ORDER BY salary DESC) AS Rank_salary
FROM employee;
-- Rank alphabetically by name
SELECT *, RANK() OVER (ORDER BY name) AS Name_rank
FROM employee;
-- Dense rank by salary
  SELECT *, DENSE_RANK() OVER (ORDER BY salary DESC) AS dense_rank_sal
FROM employee;
-- Compare both
SELECT *,
    RANK() OVER (ORDER BY salary DESC) AS Rank_sal,
    DENSE_RANK() OVER (ORDER BY salary DESC) AS Dense_rank_sal
FROM employee;
```

❖ **CONCEPT**

1. Use **OVER(ORDER BY column)** to define ranking logic.

2. Works only on **sortable columns** (e.g., **salary, name, age**).
3. Ideal for **top-N queries, leaderboards, and ranking reports**.

SQL INDEXES – In Short

Indexes improve query performance by allowing SQL Server to quickly locate rows.

They work like a book's index — faster lookup, especially for large tables.

TYPES OF INDEXES

Type	Description
Clustered Index	Sorts and stores data rows in table order (only one per table)
Non-Clustered Index	Creates a separate structure for faster lookup (can have many)

❖ CREATING INDEX

```
-- Non-clustered index on 'department'
CREATE INDEX inx_dep ON employee(department);
-- Non-clustered index on 'name'
CREATE NONCLUSTERED INDEX inx_name_nonc ON employee(name);
```

❖ QUERY EXAMPLES

```
-- Uses index on 'department'
SELECT * FROM employee WHERE department = 'Marketing';
-- Uses index on 'name'
SELECT * FROM employee WHERE name = 'Amit';
```

❖ DROPPING INDEX

```
DROP INDEX employee.inx_dep;
DROP INDEX employee.inx_name_nonc;
```

❖ You can view table info using – SP_HELP 'employee'

❖ CONCEPT

1. Indexes **speed up SELECT** queries, but **may slow down INSERT/UPDATE/DELETE.**
2. **Use indexes** on columns **that** are frequently **searched or filtered**.
3. **Avoid** indexing columns with **low uniqueness** (e.g., gender).

CLUSTERED INDEX

A clustered index determines the physical order of data in a table. Only one clustered index is allowed per table because rows can only be sorted one way.

Key Characteristics

Feature	Description
Sorts table data	Data rows are stored in index order
Only one per table	Because it defines the actual row order
Automatically created	When a Primary Key is added (unless specified otherwise)
Improves range queries	Great for queries using BETWEEN, ORDER BY, etc.

❖ CLUSTERED INDEX CREATION – SYNTAX

```
-- Ensure 'id' is NOT NULL before setting as Primary Key
ALTER TABLE employee ALTER COLUMN id INT NOT NULL;
-- Add Primary Key (creates clustered index by default)
ALTER TABLE employee ADD CONSTRAINT Pri_key_id PRIMARY KEY(id);
-- Drop Primary Key (and clustered index)
ALTER TABLE employee DROP CONSTRAINT Pri_key_id;
-- Create clustered index manually on 'name'
CREATE CLUSTERED INDEX inx_name ON employee(name);
-- Drop clustered index
DROP INDEX employee.inx_name;
-- Create clustered index on 'department'
CREATE CLUSTERED INDEX inx_d ON employee(department);
-- Drop clustered index
DROP INDEX employee.inx_d;
```

❖ INSERT EXAMPLE – SYNTAX

```
INSERT INTO employee VALUES (30, 'Ravi', 40, 'male', 120000, 'HR', 'Pune', 'India');
```

❖ VIEW TABLE – SYNTAX

```
SELECT * FROM employee;
SP_HELP 'employee';
```

❖ **CONCEPT**

1. A **Primary Key** automatically creates a **clustered index** unless one already exists.
2. Clustered indexes **physically reorder** the table but **don't change its structure**.
3. Avoid creating clustered indexes on columns with **frequent updates or low selectivity.**

NON - CLUSTERED INDEX

A non-clustered index creates a separate structure from the actual data. It stores logical pointers to the data rows rather than reordering the data physically.

KEY FEATURES

Feature	Description
Structure	Separate index structure (not tied to row order)
Pointers	Each entry links to the actual data row via a RowID
Multiple Indexes	You can create many non-clustered indexes per table
Use Case	Perfect for search/filter-heavy columns
Performance	Speeds up queries, especially with SELECT statements and WHERE clauses

✓ SYNTAX

```
-- Create a non-clustered index
CREATE NONCLUSTERED INDEX idx_emp_dept ON employee(department);
-- Drop a non-clustered index
DROP INDEX employee.idx_emp_dept;
```

✓ Usage In Queries

```
-- Query that benefits from non-clustered index
SELECT * FROM employee WHERE department = 'Sales';
-- This query hits the index first to locate matching RowIDs, then fetches full data
```


CLUSTERED VS NON-CLUSTERED — QUICK COMPARISON

Criteria	Clustered Index	Non-Clustered Index
Physical Order	Reorders table	Doesn't reorder table
Structure	Integrated with table	Separate from table
Count	Only one per table	Many allowed
Storage	Table rows sorted	Points to rows via RowID
Best For	Primary keys, fast lookups	Filters, multiple search fields

INDEX – Extra clarity 😊

❖ **What Is an Index in SQL?**

An index is like a search shortcut in a book:

- Without it, you flip page by page.
- With it, you jump straight to the topic.

There are two main types:

1. **Clustered Index** – books selves
2. **Non-Clustered Index** – searches

Clustered Index (Think it as "Main Shelf Order")

❖ **CONCEPT:**

- **Physically reorders the table data** based on the indexed column.
- Table rows are **stored in sorted order**.
- Only **ONE clustered index** allowed per table.

❖ **Example:**

If you create a clustered index on id, your employee records are physically stored sorted by id.

❖ **SYNTAX:**

```
CREATE CLUSTERED INDEX inx_id ON employee(id);
```

Non-Clustered Index (Think "Side Index Card")

❖ **CONCEPT:**

- Doesn't affect how data is stored.
- Creates a **separate lookup structure** with pointers to rows.
- You can have **multiple non-clustered indexes**.

❖ **Example:**

If you query WHERE name = 'Amit', a non-clustered index on name will quickly find matching rows without touching row order.

❖ **SYNTAX:**

```
CREATE NONCLUSTERED INDEX inx_name ON employee(name);
```

Clustered vs Non-Clustered — Summary

Feature	Clustered Index	Non-Clustered Index
Physical reorder	☑ Yes	❌ No
Storage	In table itself	Separate structure with row pointers
Quantity per table	🧍 Only one	🌐 Many allowed
Ideal for	Range queries, sorting	Search-heavy columns (like names)

ANALOGY RECAP

- Clustered index = **book sorted by chapter**
- Non-clustered index = **bookmark with topic and page number**

STORED PROCEDURES

- A **precompiled block** of SQL code stored in the database.
- Enhances **performance, reusability, and security**.
- Created using **CREATE PROCEDURE**, executed using **EXEC**.

Key Characteristics

Feature	Description
Reusable logic	Write once, run many times
Accepts parameters	Can take input values to customize output
Improves performance	Precompiled and stored in the database
Supports ALTER & DROP	Can be modified or deleted easily
Ideal for business logic	Encapsulates complex operations like salary calculations or reporting

❖ STORED PROCEDURE CREATION – SYNTAX

```
-- Basic structure
CREATE PROCEDURE procedure_name
AS
BEGIN
  -- SQL statements
END;
-- Run the procedure
EXEC procedure_name;
-- View procedure code
SP_HELPTEXT 'procedure_name';
-- Modify procedure
ALTER PROCEDURE procedure_name
AS
BEGIN
  -- Updated SQL statements
END;
-- Delete procedure
DROP PROCEDURE procedure_name;
```

❖ PRACTICAL EXAMPLE

Hike Procedure – 10% / 20% Salary Increase - **SYNTAX**

```
-- Create 10% hike
CREATE PROCEDURE hike
AS
SELECT *, (salary * 1.1) AS New_salary FROM employee;
-- Alter to 20% hike
ALTER PROCEDURE hike
AS
SELECT *, (salary * 1.2) AS New_salary FROM employee;
-- Run
EXEC hike;
```

CTC Calculator – Monthly & Annual Package - **SYNTAX**

```
-- Create with 25% hike
CREATE PROCEDURE CTC
AS
SELECT *,
(salary * 1.25) AS New_salary,
((salary * 1.25) * 12) AS yearly_package
FROM employee;
-- Alter to 30% hike
ALTER PROCEDURE CTC AS
SELECT *,
(salary * 1.3) AS New_salary,
((salary * 1.3) * 12) AS yearly_package
FROM employee;
-- Run
EXEC CTC;
```

Salary Slip Generator – Employee + Project - **SYNTAX**

```
CREATE PROCEDURE Salary_slip
AS
SELECT e.id, e.name,
(e.salary * 0.2) AS HRA,
(e.salary * 0.3) AS Special_allowence,
(e.age * 20) AS Professional_tax,
(e.age * 25) AS Food_allowence,
(p.pincode * 0.8) AS Income_tax,
(p.pincode * 0.5) AS Transport_allowence,
(e.salary * 0.2) AS Shift_allowence
FROM employee e
INNER JOIN
project p ON e.id = p.pid;
-- Run
EXEC Salary_slip;
```

Department-Based Hike – With Parameters - SYNTAX

```
-- Create procedure with input
CREATE PROCEDURE dept (@a VARCHAR(255))
AS
SELECT *, (salary * 1.25) AS New_salary
  FROM employee
  WHERE department = @a;
-- Run for specific departments
EXEC dept @a = 'IT';
EXEC dept @a = 'HR';
EXEC dept @a = 'Accounts';
```


❖ CONCEPT

1. Stored procedures encapsulate logic and reduce repetition.
2. They can be altered without affecting table structure.

3. Use parameters to make procedures dynamic and reusable.
4. Avoid putting heavy logic in SELECT queries — use procedures instead.

VIEWS

A view is a virtual table based on the result of a SQL query.
It does not store data physically, but allows you to filter, simplify, and secure access to data.

KEY CHARACTERISTICS

Feature	Description
Virtual table	Based on a SELECT query
No physical storage	Data is fetched from base tables
Can be queried	Just like a regular table
Can be updatable	If based on a single table without joins or aggregations
Used for filtering	Commonly used to create filtered subsets of data

❖ VIEW CREATION – SYNTAX

```
-- Create a view for male employees
CREATE VIEW vi_male
AS
SELECT * FROM employee WHERE gender = 'male';
-- Query the view
SELECT * FROM vi_male;
-- Insert into view (if allowed)
INSERT INTO vi_male VALUES (31, 'Raju', 28, 'Male', 54000, 'HR', 'Chennai', 'India');
-- Delete from view (if allowed)
DELETE FROM vi_male WHERE name = 'Joseph';
```

❖ PRACTICAL EXAMPLES

```
CREATE VIEW vi_male
AS
SELECT * FROM employee WHERE gender = 'male';
SELECT * FROM vi_male;
```

CREATE AND QUERY VIEW

```
CREATE VIEW vi_male
AS
SELECT * FROM employee WHERE gender = 'male';
SELECT * FROM vi_male;
```

INSERT INTO BASE TABLE

```
INSERT INTO employee VALUES
(32, 'Ganesh', 29, 'male', 56000, 'Accounts', 'Bangalore', 'India'),
```

(33, 'Neha', 30, 'Female', 60000, 'Marketing', 'Surat', 'India');

Automatically reflects in the view

if the data matches the view condition.

INSERT INTO VIEW

```
INSERT INTO vi_male VALUES
(31, 'Raju', 28, 'Male', 54000, 'HR', 'Chennai', 'India');
```

IT WORKS BECAUSE VI_MALE IS BASED ON SINGLE TABLE AND INCLUDE ALL COLUMNS

DELETE FROM VIEW

```
DELETE FROM vi_male WHERE name = 'Joseph'
```

Deletes from the base table if the row exists and meets the view condition.

❖ **CONCEPT**

1. Views simplify complex queries and improve readability.
2. Views can be used to restrict access to sensitive columns.
3. You can insert/delete from views if they are simple and updatable.
4. Views are dynamic — they reflect changes made to the base table.

FLOOR & CEILING

❖ **FLOOR**

Returns the **largest integer less than or equal to** the given number.

Think of it as **rounding down**.

✓ **SYNTAX – EXAMPLE**

```
SELECT FLOOR(23.2);    -- 23
SELECT FLOOR(23.9);    -- 23
SELECT FLOOR(1.89);    -- 1

SELECT FLOOR(0.01);    -- 0

SELECT FLOOR(-10.1);   -- -11
SELECT FLOOR(-5.9);    -- -6
```

✓ **Application on Table**

```
SELECT *, FLOOR(age) AS floored_age FROM employee;
```

❖ **CEILING**

Returns the **smallest integer greater than or equal to** the given number.

Think of it as **rounding up**.

✓ **SYNTAX – EXAMPLE**

```
SELECT CEILING(10.7);    -- 11
SELECT CEILING(0.7);     -- 1
SELECT CEILING(-1.2);    -- -1
SELECT CEILING(-10.7);   -- -10
```

✓ **Application on table**

```
SELECT *, CEILING(age) AS ceiled_age FROM employee;
```

LAG & LEAD

Both are window functions used to access data from previous or next rows.

❖ **LAG**

Returns the value from the previous row based on the specified order.

✓ **SYNTAX**

```
SELECT name, LAG(name) OVER (ORDER BY name) AS lag_name
FROM employee;
```

❖ **LEAD**

Returns the value from the next row based on the specified order

✓ **SYNTAX**

```
SELECT name, LEAD(name) OVER (ORDER BY name) AS lead_name
FROM employee;
```

❖ **Combined LAG & LEAD - SYNTAX**

```
SELECT name,
    LAG(name) OVER (ORDER BY name) AS lag_name,
    LEAD(name) OVER (ORDER BY name) AS lead_name
FROM employee;
```

❖ ❌ **Nested LAG & LEAD** – not recommended ❌

```
-- This is syntactically incorrect and not supported
SELECT name, LAG(LAG(name)
        OVER (ORDER BY name)) OVER (ORDER BY name)
FROM employee;
```

❖ **NOTE:**

Avoid nesting LAG() or LEAD() — use multiple columns instead.

❖ **CONCEPT**

1. **FLOOR()** and **CEILING()** are used for rounding numbers.
2. **LAG()** and **LEAD()** help compare current row with previous/next row.
3. These functions are useful in trend analysis, ranking, and reporting.
4. Always use **ORDER BY** inside **OVER()** to define row sequence.

Why You Should Avoid Nesting LAG( ) or LEAD( )

Window functions like **LAG()** and **LEAD()** are designed to work
row by row within a defined window.
Nesting them (putting one inside another) leads to
syntax errors or **unexpected behaviour**
because SQL Server doesn't support **multi-layered window function calls**.

❌ Incorrect Example (Nested LAG) ❌

```
SELECT name,
    LAG(LAG(name) OVER (ORDER BY name)) OVER (ORDER BY name)
FROM employee;
```

Why This Fails?

- The inner **LAG()** returns a value.
- The outer **LAG()** tries to treat that value as a column — which it isn't.
- **SQL Server** doesn't allow a window function to be used inside another window function.

✓ Correct Approach – Use Multiple Columns ✓

```
SELECT name,
    LAG(name, 1) OVER (ORDER BY name) AS prev_name,
    LAG(name, 2) OVER (ORDER BY name) AS two_steps_back
FROM employee;
```

Why This Works?

- Each **LAG()** is independent.
- You can **access multiple previous rows without nesting**.

❖ **CONCEPT**

1. Window functions like **LAG()** and **LEAD()** must be used independently.
2. Nesting them breaks **SQL** syntax rules.
3. Use multiple columns to access **multiple previous/next values**.
4. This approach is **cleaner**, **readable**, and **supported by SQL Server**.

TRANSACTION IN SQL

A **transaction** is a group of SQL operations (**INSERT**, **UPDATE**, **DELETE**) that are executed as a **single unit of work**.

The goal is **atomicity**:

either all operations succeed, **or none do** — ensuring **data integrity**.

Key Characteristics

Feature	Description
Atomicity	All operations succeed or all fail
Consistency	Keeps database in a valid state
Isolation	Transactions don't interfere with each other
Durability	Once committed, changes are permanent
Rollback	Reverts all changes if an error occurs

✓ **SYNTAX**

```
BEGIN TRY
  BEGIN TRANSACTION
    -- SQL operations
  COMMIT TRANSACTION
  PRINT 'Transaction committed'
END TRY
BEGIN CATCH
  ROLLBACK TRANSACTION
  PRINT 'Transaction rollback'
END CATCH
```

❖ Transaction Rollback – SYNTAX

✓ Fails due to division by zero → triggers rollback.

```
--Beginning transaction inside try block
BEGIN TRY
  BEGIN TRANSACTION
    UPDATE employee SET age = 30 WHERE name = 'Amit';
    UPDATE employee SET department = 'Marketing' WHERE name = 'Anjali';
    UPDATE employee SET city = 'Dehradun' WHERE name = 'Badal';
    -- Error line i.e. 90000 / 0
    UPDATE employee SET salary = 90000 / 0 WHERE name = 'Priya';
  COMMIT TRANSACTION
  PRINT 'Transaction committed'
END TRY
--After the error occur the transaction will be handled by catch block
BEGIN CATCH
  ROLLBACK TRANSACTION
  PRINT 'Transaction rollback'
END CATCH
```

- **All changes are undone due to the error.**

❖ Transaction Committed – SYNTAX

✓ All operations succeed → changes are saved.

```
--As there are no error then the code will be executed and saved successfully.
BEGIN TRY
  BEGIN TRANSACTION
    UPDATE employee SET age = 35 WHERE name = 'Amit';
    UPDATE employee SET salary = 64000 WHERE name = 'Anjali';
    UPDATE employee SET city = 'Kolkata' WHERE name = 'Badal';
    UPDATE employee SET department = 'Accounts' WHERE name = 'Priya';
    UPDATE employee SET salary = 71000 WHERE name = 'Arun';
  COMMIT TRANSACTION
  PRINT 'Transaction committed'
END TRY
BEGIN CATCH
  ROLLBACK TRANSACTION
  PRINT 'Transaction rollback'
END CATCH
```

- **All changes are saved successfully.**

❖ CONCEPT

1. Use transactions to group critical updates together.
2. If any part fails, the entire transaction is rolled back.
3. Helps prevent partial updates and data corruption.
4. Always wrap transactions in **TRY...CATCH** for safety.

OFFSET - FETCH

OFFSET and **FETCH** are used for **pagination** — retrieving a **subset of rows** from a query result.

Perfect for displaying data in **pages or chunks**, like on websites or dashboards.

KEY CHARACTERISTICS

Feature	Description
OFFSET	Skips a specified number of rows
FETCH	Limits the number of rows returned after the offset
Requires ORDER BY	Must be used with ORDER BY clause
Ideal for	Paginated tables, infinite scroll, large datasets

❖ PAGINATION SYNTAX

```
SELECT * FROM employee
ORDER BY id
OFFSET 0 ROWS FETCH NEXT 5 ROWS ONLY;
```

EXAMPLE

❖ First Page – Rows 1 to 5

```
SELECT * FROM employee
ORDER BY id
OFFSET 0 ROWS FETCH NEXT 5 ROWS ONLY;
```

❖ Second Page – Rows 6 to 10

```
SELECT * FROM employee
ORDER BY id
OFFSET 5 ROWS FETCH NEXT 5 ROWS ONLY;
```

❖ **Third Page – Rows 11 to 15**

```
SELECT * FROM employee
ORDER BY id
OFFSET 10 ROWS FETCH NEXT 5 ROWS ONLY;
```

❖ **CONCEPT**

1. **OFFSET** skips rows — useful for moving to the next page.
2. **FETCH NEXT** limits how many rows are returned.
3. Must be used with **ORDER BY** to ensure consistent row order.
4. Commonly used in **web apps**, **reports**, and **data browsing tools**.

EXCEPTION HANDLING

✓ Exception handling allows you to **gracefully manage errors** during query execution using:

```
BEGIN TRY
  -- Code that might throw an error
END TRY
BEGIN CATCH
  -- Code to handle the error
END CATCH
```

Useful Error Functions in CATCH Block

Function	Description
ERROR_MESSAGE()	Returns the full error message
ERROR_NUMBER()	Returns the error number
ERROR_LINE()	Line number where the error occurred
ERROR_SEVERITY()	Severity level of the error
ERROR_STATE()	State number of the error

EXAMPLES

❖ **Division by zero**

```
BEGIN TRANSACTION salary_employee
BEGIN TRY
   SELECT salary / 0 FROM employee;
END TRY
BEGIN CATCH
   SELECT ERROR_MESSAGE() AS error;
```

```
END CATCH;
```

✓ **CATCH block will catch the divide-by-zero error and return a readable message.**

❖ **Valid Query – No Error**

```
BEGIN TRANSACTION salary_123
BEGIN TRY
   SELECT salary FROM employee;
END TRY
BEGIN CATCH
   SELECT ERROR_MESSAGE() AS error;
END CATCH;
```

✓ **No error here, so the CATCH block won't execute**

❖ **CONCEPT – Best Practice**

1. Always use **TRY...CATCH** around risky operations (e.g., division, casting, transactions).
2. Use **ROLLBACK** inside **CATCH** if a transaction fails.
3. Log errors using a custom error table or **RAISERROR**.

SQL TRIGGERS

Triggers are **special stored procedures** that automatically execute in response to **DML events** like **INSERT, UPDATE**, or **DELETE**.

TYPES OF TRIGGERS

Trigger Type	Description
AFTER Trigger	Executes **after** the DML operation completes
INSTEAD OF Trigger	**Intercepts** the DML operation and replaces it with custom logic

EXAMPLES

❖ **AFTER UPDATE Trigger**

```
CREATE TRIGGER trig_upd ON employee
AFTER UPDATE
AS
BEGIN
   PRINT 'Your data has been updated'
END;
```

```
--update example
UPDATE employee SET age = 29 WHERE name = 'Amit';
```

❖ ALTER Trigger Message

```
ALTER TRIGGER trig_upd ON employee
AFTER UPDATE
AS
BEGIN
   PRINT 'You have successfully updated the data in employee table'
END;
```

❖ AFTER INSERT & DELETE Trigger

```
CREATE TRIGGER tri_in_del ON employee
AFTER INSERT, DELETE
AS
BEGIN
   PRINT 'You have inserted or deleted row(s) from employee table'
END;
```

❖ INSTEAD OF UPDATE Trigger

```
CREATE TRIGGER t_u_i ON employee
INSTEAD OF UPDATE
AS
BEGIN
   PRINT 'You cannot update any data in the employee table'
END;
```

❖ Enable/Disable Trigger

```
ALTER TABLE employee DISABLE TRIGGER ti_del_inst;
ALTER TABLE employee ENABLE TRIGGER ti_del_inst;
```

❖ Dropping Triggers

```
DROP TRIGGER trig_upd, tri_in_del, t_u_i, ti_del_inst;
```

❖ CONCEPT

1. Triggers are **automated responses** to data changes.
2. AFTER triggers run **after** the DML operation completes.
3. INSTEAD OF triggers **block the original operation** and run custom logic.
4. Useful for **auditing**, **enforcing business rules**, and **preventing unwanted changes**.
5. Triggers can be **enabled/disabled** without dropping them.

SQL CAST FUNCTION

The **CAST( )** function is used to
convert a value from one data type to another.
It's especially useful when you need to format data or perform operations across different types.

❖ SYNTAX

```
CAST(expression AS target_data_type)
```

❖ Example - Convert Decimal to Integer

```
SELECT CAST(56.78 AS INT);
-- Output: 56 (decimal part is truncated)
```

❖ Convert Column to Varchar in SELECT

```
SELECT *, CAST(salary AS VARCHAR(255)) FROM employee;
-- Adds a new column with salary as text
```


CONCEPT RECAP

Feature	Description
Purpose	Convert data types explicitly
Common Use Cases	Formatting output, joining mismatched types, preparing for string functions
Truncation	Decimal to INT removes fractional part
Flexibility	Works with INT, VARCHAR, DATE, FLOAT, etc.

ROW_NUMBER( )

The ROW_NUMBER() function assigns a **unique sequential number** to each row in a result set.
It's a **window function**, meaning it works over a defined set of rows using the **OVER()** clause.

✓ SYNTAX

```
ROW_NUMBER() OVER ([PARTITION BY column] ORDER BY column)
```

- **ORDER BY:** Required. Defines the sequence.
- **PARTITION BY:** Optional. Resets numbering for each group.

EXAMPLES

❖ **Basic Row Number by Name**

```
SELECT *, ROW_NUMBER() OVER(ORDER BY name) AS roll_number FROM employee;
```

❖ **Row Number by Age (Descending)**

```
SELECT *, ROW_NUMBER() OVER(ORDER BY age DESC) AS senior FROM employee;
```

❖ **Row Number by Department**

```
SELECT *, ROW_NUMBER() OVER(ORDER BY department) AS dept_no FROM employee;
```

❖ **Partition by Gender, Order by Name**

```
SELECT *, ROW_NUMBER() OVER(PARTITION BY gender ORDER BY name) AS Roll_no FROM employee;
```

❖ **Partition by Department, Order by Department**

```
SELECT *, ROW_NUMBER() OVER(PARTITION BY department ORDER BY department) AS dep_no FROM
employee;
```

❖ **DENSE_RANK by Department**

```
SELECT *, DENSE_RANK() OVER(ORDER BY department) AS dep_no FROM employee;
```

❖ **Insert New Row**

```
INSERT INTO employee VALUES (34, 'Mark', 27, 'Male', 45000, 'IT', 'Seattle', 'USA');
```


SUMMARY TABLE

Function	Unique?	Gaps in Rank?	Partition Support	Use Case
ROW_NUMBER()	Yes	Yes	Yes	Unique row IDs, pagination
DENSE_RANK()	No	No	Yes	Grouped ranking without gaps

COALESCE( )

The **COALESCE() function** returns the first **non-NULL value** from a list of expressions.

It's commonly used to handle **NULLs** and **provide default values**.

KEY CHARACTERISTICS

Feature	Description
NULL handling	Replaces NULL with the first non-null value
Left-to-right eval	Evaluates expressions in order
Flexible types	Works with strings, numbers, dates, etc.
Common use cases	Display fallback values, clean up NULLs in reports

✓ **SYNTAX**

```
COALESCE(expression1, expression2, ..., expressionN)
```

EXAMPLE

❖ **BASIC USAGE**

```
SELECT COALESCE(NULL, NULL, NULL, 'SQL', NULL, 'Python');
-- Output: 'SQL'
```

❖ **REPLACE NULL IN COLUMN**

```
SELECT *, COALESCE(city, 'Indore') AS Updated_city FROM employee;
```

❖ **UPDATE TABLE WITH DEFAULT**

```
UPDATE employee SET city = COALESCE(city, 'Indore');
```

❖ **CONCEPT**

1. **COALESCE()** is ideal for cleaning **NULL**s in queries.
2. It stops at the first non-null value.
3. All expressions should be of compatible data types.
4. Use it in **SELECT, UPDATE**, or even inside **JOIN** conditions.

STUFF( )

The STUFF() function **deletes a portion of a string** and **inserts another string** at a specified position.

KEY CHARACTERISTICS

Feature	Description
String manipulation	Deletes and inserts characters in a string
Position-based	Works using start position and length
Useful for	Formatting, masking, dynamic replacements

Returns NULL	If position or length is invalid

✓ **SYNTAX**

```
STUFF(string, start_position, length_to_replace, new_string)
```

EXAMPLES

❖ **Replace Part of a String**

```
SELECT STUFF('This is SQL class', 9, 3, 'Data Science');
-- Output: 'This is Data Science class'
```

❖ **Replace Month name**

```
SELECT STUFF('This month is June', 15, 4, 'July');
-- Output: 'This month is July'
```

❖ **Add Prefix to Column**

```
SELECT *, STUFF(department, 1, 0, 'dept-') AS updated_dept FROM employee;
```

❖ **Modify static Text**

```
SELECT *, STUFF('IT', 2, 1, 'T-Engineer') AS engineer_type FROM employee;
```

❖ **CONCEPT**

1. **STUFF()** is great for dynamic string edits.
2. It can **insert, replace, or mask** parts of a string.
3. Use it for **data formatting, custom labels, or report generation**.
4. Unlike **REPLACE(), STUFF()** works with position and length.

CTE (Common Table Expression)

A **CTE** is a temporary, named result set defined using the **WITH** keyword.
It helps simplify complex queries by breaking them into **modular, readable blocks**.

KEY CHARACTERISTICS

Feature	Description
Temporary result set	Exists only during query execution
Named expression	Can be referenced like a table
Improves readability	Makes complex queries easier to manage
Supports DML	Can be used in SELECT, INSERT, UPDATE, DELETE
Recursive support	Can be used for hierarchical or recursive queries

❖ **CTE CREATION – SYNTAX**

```
WITH cte_name AS (
```

```
SELECT column1, column2, ...
FROM table_name
WHERE condition
)
SELECT * FROM cte_name;
```

EXAMPLES

❖ Create CTE with ROW_NUMBER( )

```
WITH Lion AS (
SELECT *, ROW_NUMBER()
OVER (PARTITION BY id ORDER BY id) AS Id_no
FROM employee
)
DELETE FROM Lion WHERE Id_no > 1;
```

- **Deletes duplicate rows based on id.**

❖ Another CTE for Cleanup

```
WITH Class AS (
SELECT *, ROW_NUMBER()
OVER (PARTITION BY id ORDER BY id) AS Rownumber
FROM employee
)
DELETE FROM Class WHERE Rownumber > 1;
```

- **Removes duplicate entries while keeping the first.**

❖ Insert New Data

```
INSERT INTO employee VALUES
(39, 'Siva', 29, 'Male', 150000, 'IT', 'Delhi', 'India'),
(40, 'Payal', 55, 'Female', 45000, 'HR', 'Mumbai', 'India'),
(41, 'Varun', 34, 'Male', 50000, 'Marketing', 'Amritsar', 'India'),
(42, 'Sharline', 40, 'Female', 45000, 'HR', 'Aukland', 'New Zealand');
```

Why use CTEs?

Benefit	Description
Readability	Breaks complex logic into manageable blocks
Reusability	Can be referenced multiple times in the main query
Recursive Support	Ideal for hierarchical data like org charts
DML Integration	Can be used in SELECT, INSERT, UPDATE, DELETE, MERGE

❖ CONCEPT

1. **CTEs** are defined using the **WITH** keyword and act like **temporary views**.
2. They improve **query readability**, especially with **window functions** and **joins**.
3. CTEs can be used in **DELETE**, **UPDATE**, and **INSERT** operations.
4. They are **not stored** — they exist only during query execution.

5. Recursive **CTEs** allow you to **handle hierarchical data** like org charts or tree structures.

IIF( ) Statement – Conditional

The **IIF() function** is a shorthand alternative to the **CASE** statement, used for inline conditional logic in **SELECT** queries.

It's **only for viewing**, **not usable in constraints** or procedural logic.

KEY CHARACTERISTICS

Feature	Description
Inline conditional	Returns one of two values based on a condition
Readability	Cleaner than CASE for simple conditions
Nestable	Can be nested for multiple conditions
Viewing only	Used in SELECT, not in constraints or stored procedures
SQL Server specific	Available in SQL Server 2012+

✓ **SYNTAX**

```
IIF(condition, true_value, false_value)
```

EXAMPLES

❖ **Basic Role Assignment**

```
SELECT *,
IIF(salary > 60000, 'Senior Engineer', 'Junior Engineer')
AS Role
FROM employee;
```

❖ **Gender-Based Prefix**

```
SELECT *,
IIF(gender = 'Male', WO('Mr.', name),
CONCAT('Miss.', name)) AS Emp_name
FROM employee;
```

❖ **Department-Based Domain**

```
SELECT *,
IIF(department = 'IT', 'Engineering Domain',
   IIF(department = 'Marketting', 'Sales Domain',
   IIF(department = 'Accounts', 'Finance Domain', 'Management Domain')))
AS Domain
FROM employee;
```

❖ **Save Result into New Table**

```
SELECT *, IIF(department = 'IT', 'Engineering Domain',
   IIF(department = 'Marketting', 'Sales Domain',
   IIF(department = 'Accounts', 'Finance Domain', 'Management Domain'))) AS Domain
INTO emp_dom
FROM employee;
SELECT * FROM emp_dom;
```

❖ Age-Based Role Assignment

```
SELECT *, IIF(age < 42, 'Junior Engineer', 'Senior Engineer')
AS Role
FROM employee
ORDER BY age;
```

❖ Multi-Level Designation Based on Age

```
SELECT *, IIF(age < 23, 'Intern',
   IIF(age < 27, 'Junior Engineer',
   IIF(age < 29, 'Senior Engineer',
   IIF(age < 33, 'Team Leader',
   IIF(age < 41, 'Manager',
   IIF(age < 50, 'Senior Manager', 'President')))))) AS Designation
FROM employee
ORDER BY age;
```

❖ Project Assignment Based on Name Initial

```
-- Using SUBSTRING
SELECT *,
IIF(SUBSTRING(name, 1, 1) IN ('a', 'b', 'c', 'd', 'e', 'f'), 'Western Union',
IIF(SUBSTRING(name, 1, 1) IN ('g', 'h', 'i', 'j', 'k', 'l'), 'Johnson', 'Vodafone')) AS Project_name
FROM employee;

-- Using LIKE pattern
SELECT name,
   IIF(name LIKE '[A-F]%', 'Western Union',
      IIF(name LIKE '[H-L]%', 'Johnson', 'Vodafone')) AS Project_name
FROM employee;
```

❖ CONCEPT

1. **IIF()** is a **simplified conditional function** for quick logic in **SELECT** queries.
2. It's **nestable**, allowing multiple conditions in a compact format.
3. Best used for **display logic**, not for procedural or constraint-based operations.
4. For complex logic or performance-critical queries, prefer **CASE**.

ROLLUP & ROUND – FINAL TOPIC

❖ ROLLUP — Advanced Grouping for Subtotals & Totals

ROLLUP is a **GROUP BY** extension that adds subtotal and grand total rows to grouped query results.

❖ SYNTAX

```
SELECT column1, column2, AGG_FUNC(column3)
FROM table_name
GROUP BY ROLLUP (column1, column2);
```

❖ BEHAVIOR:

1. Adds subtotal for each group level.
2. Adds grand total row with **NULL** in grouped columns.
3. Use **COALESCE( )** to replace **NULL** with readable labels.

❖ EXAMPLE – SYNTAX

```
SELECT COALESCE(department, 'Total'),
    COALESCE(gender, 'Subtotal'),
    SUM(salary) AS Total_salary
FROM employee
GROUP BY ROLLUP(department, gender);
```

OUTPUT STRUCTURE

Department	Gender	Total_salary
HR	Male	50,000
HR	Female	45,000
HR	NULL	95,000 ← Subtotal for HR
IT	Male	60,000
IT	NULL	60,000 ← Subtotal for IT
NULL	NULL	155,000 ← Grand Total

❖ ROUND — Precision Control for Numeric Values

ROUND() is a scalar function that **rounds a numeric value** to a specified number of decimal places.

✓ SYNTAX

```
ROUND(numeric_expression, decimal_places)
```

❖ EXAMPLE

```
SELECT ROUND(12.4567, 3);  -- Output: 12.457
SELECT ROUND(12.4567, 2);  -- Output: 12.46
SELECT ROUND(12.4567, 1);  -- Output: 12.5
```

❖ **USE CASES**

1. Formatting salaries, prices, percentages.
2. Cleaning up data for reports or dashboards.
3. Ensuring consistent decimal precision.

MS SQL NOTES

AMIT KUMAR PRASAD