DBMS NOTES

1]### What is DBMS?

A **Database Management System (DBMS)** is software that helps store, manage, and organize data efficiently. It allows users to create, read, update, and delete data in a structured way.

Why DBMS?

Before DBMS, data was stored in files, which made it difficult to manage. DBMS solves many problems by providing:

✓ **Easy Data Management** – Organizes data properly and avoids duplication.

✓ **Data Security** – Controls access to sensitive data.

✓ **Fast Data Retrieval** – Quickly fetches information using queries.

✓ **Data Consistency** – Ensures accurate and up-to-date data.

✓ **Multi-user Access** – Many users can work on the same data without issues.

Popular DBMS examples: **MySQL, Oracle, PostgreSQL, MongoDB.**

2]### **Difference Between SQL and NoSQL (Simple Explanation)**

Difference Between SQL and NoSQL (Simple Explanation) ...NoSQL also called as N-RDBMS.

1️⃣ **Data Storage:**

- SQL databases store data in tables with rows and columns, just like an Excel sheet.

- NoSQL databases store data in different formats like documents (JSON), key-value pairs, graphs, or wide columns.

2️⃣ **Structure:**

- SQL databases have a **fixed schema**, meaning you must define the structure before storing data.

- NoSQL databases are **flexible**, allowing you to store data without a strict structure.

3️⃣ **Use Cases:**

- SQL is best for applications where data needs to be highly structured, like banking, payroll, and inventory systems.

- NoSQL is best for applications where data is constantly changing, like social media platforms, IoT, and big data analytics.

4️⃣ **Scalability:**

- SQL databases scale **vertically**, meaning you need a bigger, more powerful server to handle more data.

- NoSQL databases scale **horizontally**, meaning you can add more servers to distribute the load.

5️⃣ **Query Language:**

- SQL databases use **SQL** (Structured Query Language) to manage and

retrieve data. Example:

```sql
SELECT * FROM students WHERE marks > 90;
```

- NoSQL databases use different methods depending on the type, like JSON-based queries in MongoDB. Example:

```json
{ "marks": { "$gt": 90 } }
```

6️⃣ **Transaction Handling:**

- SQL databases follow **ACID (Atomicity, Consistency, Isolation, Durability)** rules, ensuring data is reliable and consistent.

- NoSQL databases focus on **speed and flexibility**, and not all support full ACID properties.

7️⃣ **Examples:**

- SQL Databases: MySQL, PostgreSQL, Oracle, SQL Server.

- NoSQL Databases: MongoDB, Cassandra, Firebase, DynamoDB.

When to Use What?

✅ **Choose SQL** when working with structured data that needs strict

rules (e.g., banking, finance, HR systems).

✅ **Choose NoSQL** when handling large, evolving data that requires flexibility and speed (e.g., social media, streaming, IoT).

👉 **Think of SQL as a well-organized bookshelf with labeled sections** and **NoSQL as a big storage box where you can put things without strict order!** 🚀

3] Keys in DBMS

☐Primary Key (PK)

✓ A Primary Key is a column (or a combination of columns) that uniquely identifies each row in a table.

✓ It cannot have duplicate or NULL values.

✓ A table can have only one Primary Key.

💡 Example: In a Students table, student_id is the Primary Key because every student has a unique ID.

```
CREATE TABLE Students (
   student_id INT PRIMARY KEY,
   name VARCHAR(50),
   age INT
);
```

☐Foreign Key (FK)

✓ A Foreign Key is a column that establishes a relationship between two tables.

✓ It refers to the Primary Key of another table.

✓ It can have duplicate values and NULL values (if not mandatory).

💡 Example: In a Courses table, student_id is a Foreign Key that refers to the Students table.

```
CREATE TABLE Courses (
   course_id INT PRIMARY KEY,
   course_name VARCHAR(50),
   student_id INT,
   FOREIGN KEY (student_id) REFERENCES Students(student_id)
);
```

📌 Why Foreign Key? To maintain data integrity, ensuring students exist before enrolling in a course.

☐Composite Key

✓ A Composite Key is a combination of two or more columns that together uniquely identify a row.

✓ Used when a single column is not enough to uniquely identify a row.

💡 Example: In a Orders table, order_id alone may not be unique if a customer places multiple orders. So, we use (customer_id, order_id) as a Composite Key.

```
CREATE TABLE Orders (
   customer_id INT,
   order_id INT,
   order_date DATE,
   PRIMARY KEY (customer_id, order_id)
);
```

☐Candidate Key

✓ A Candidate Key is a column (or set of columns) that can be a potential Primary Key.

✓ A table can have multiple Candidate Keys, but only one can be the Primary Key.

💡 Example: In an Employees table, both emp_id and email can uniquely identify employees.

```
CREATE TABLE Employees (
   emp_id INT UNIQUE,
   email VARCHAR(100) UNIQUE,
```

```
    phone VARCHAR(15),
    PRIMARY KEY (emp_id)
);
```

📌 Here, emp_id is chosen as the Primary Key, but email is also a Candidate Key.

Summary (Easy Analogy)

Primary Key → Like a Roll Number in school (Unique for each student).

Foreign Key → Like a reference to another school's Roll Number in a scholarship record.

Composite Key → Like using (Name + DOB) to identify a person if no Roll Number is available.

Candidate Key → Like having both an Aadhaar Number and Passport Number (both unique, but only one is chosen as the main identifier).

👉 Understanding keys ensures data consistency, avoids duplication, and maintains relationships between tables! 🚀

4]### **ACID Properties in DBMS (Simple Explanation)**

ACID properties ensure that database transactions are **reliable, consistent, and safe** from failures. A transaction is a set of operations that either **all succeed together or fail together** (like an ATM withdrawal or online purchase).

👉 **ACID stands for:**

☐**Atomicity**

☐**Consistency**

☐**Isolation**

☐**Durability**

---

*☐Atomicity (All or Nothing)**

✓ **Atomicity ensures that a transaction is either fully completed or fully rolled back.**

✓ If any part of the transaction fails, the entire transaction is undone.

✓ This prevents partial updates, avoiding data corruption.

💡 **Example:**

- You transfer ₹500 from **Account A to Account B**.

- If the money is deducted from A **but not added to B**, the transaction **must be rolled back**.

- Either **both happen together** or **nothing happens at all**.

🔹 **Without Atomicity:** ₹500 is deducted but not added.

🔹 **With Atomicity:** If anything fails, no money is deducted.

---

*Consistency (Data Should Always Be Valid)**

✓ **Consistency ensures that the database remains in a valid state before and after a transaction.**

✓ Transactions must follow the database rules (constraints, relations, etc.).

✓ Prevents corrupt or invalid data from being stored.

💡 **Example:**

- A **bank account balance cannot be negative**.

- If a withdrawal of ₹10,000 is attempted from an account with ₹5,000, the database **must reject** the transaction.

🔹 **Without Consistency:** The system allows a negative balance (-₹5,000).

🔹 **With Consistency:** The withdrawal is rejected, keeping valid data.

---

*Isolation (No Interference Between Transactions)**

✓ **Isolation ensures that multiple transactions happening at the same time do not affect each other.**

✓ Transactions should act as if they are executed one by one, even if they run in parallel.

💡 **Example:**

- Two users **buy the last ticket** for a movie at the same time.

- Without Isolation, both transactions might see the last available seat and both may be confirmed.

- With Isolation, **only one will succeed, and the other will see "Sold Out."**

🔹 **Without Isolation:** Two people get the same seat.

🔹 **With Isolation:** Only one person gets the seat, avoiding conflicts.

---

*Durability (Data Should Not Be Lost)**

✓ **Durability ensures that once a transaction is committed, it is permanently saved in the database—even if a system crash happens.**

✓ Data is stored in non-volatile memory (hard disk, SSD) to prevent loss.

💡 **Example:**

- After booking a train ticket online, **even if the server crashes**, your booking remains safe.

- Once the payment is successful, the seat is confirmed forever.

🔻 **Without Durability:** System crashes → Ticket disappears.

🔻 **With Durability:** System crashes → Ticket is still booked.

Summary (Easy Analogy 🎯)

Imagine you're ordering food online:

✅ **Atomicity:** If payment fails, your order is canceled (not half-placed).

✅ **Consistency:** The restaurant won't let you order "negative" pizzas.

✅ **Isolation:** If 2 people order the last pizza at the same time, only

one will succeed.

✅ **Durability:** Even if the app crashes, your order is still confirmed.

ACID properties ensure database transactions are safe, accurate, and reliable! 🚀

5]### **Types of Relationships in DBMS (Simple Explanation with Examples)**

In a **Database Management System (DBMS)**, relationships define how tables (entities) are connected. There are **three main types** of relationships:

*1️⃣One-to-One (1:1) Relationship**

What is it?

- One record in **Table A** is related to **only one** record in **Table B**.

- Used when **one entity has a unique relation** with another entity.

Example: Passport & Person

- **Person Table** (Person_ID, Name, Age)

- **Passport Table** (Passport_ID, Issue_Date, Person_ID)

☞ **Each person has only one passport, and each passport belongs to only one person.**

Real-World Example

✓ A **country & its president** (one country has one president at a time).

✓ A **vehicle & its registration number** (one vehicle has only one registration number).

One-to-Many (1:M) Relationship

What is it?

- One record in **Table A** can be linked to **multiple** records in **Table B**.

- Most common relationship in databases.

Example: Teacher & Students

- **Teacher Table** (Teacher_ID, Name, Subject)

- **Student Table** (Student_ID, Name, Teacher_ID)

☞ **One teacher teaches many students, but each student has only one teacher.**

Real-World Example

✓ **Bank & Customers** (one bank can have many customers).

✓ **Company & Employees** (one company has many employees).

Many-to-Many (M:N) Relationship

What is it?

- One record in **Table A** is linked to **many** records in **Table B**, and vice versa.

- Requires a **junction table** (third table) to connect them properly.

Example: Students & Courses

- **Student Table** (Student_ID, Name)

- **Course Table** (Course_ID, Course_Name)

- **Student_Course Table** (Student_ID, Course_ID) → **Junction Table**

☞ **One student can enroll in multiple courses, and one course can have many students.**

Real-World Example

✓ **Doctors & Patients** (one doctor has many patients, and one patient can visit multiple doctors).

✓ **Actors & Movies** (one actor can work in multiple movies, and one movie can have multiple actors).

Key Takeaways 🚀

- **One-to-One (1:1)** → Each record has **only one** matching record. _(e.g., Passport & Person)_

- **One-to-Many (1:M)** → One record is linked to **multiple** records. _(e.g., Teacher & Students)_

- **Many-to-Many (M:N)** → Many records in one table are linked to

many in another. _(e.g., Students & Courses)_

Why Are These Relationships Important?

✅ Helps in **organizing data efficiently**.

✅ Reduces **data redundancy** (avoiding duplicate data).

✅ Ensures **data consistency** (correct relationships between tables).

Hope this makes it super easy to understand! ☺

6]## **Normalization & Denormalization in DBMS (Simple Explanation)**

What is Normalization?

Normalization is the process of **organizing a database** to reduce **redundancy (duplicate data)** and improve **data consistency**. It ensures that data is **stored efficiently** and avoids problems like **insertion, update, and deletion anomalies**.

Types (Forms) of Normalization

Normalization is done in **steps**, called **normal forms (NF)**. Each step **fixes a specific issue** in the database.

*First Normal Form (1NF) – Remove Duplicate Data & Ensure Atomicity**

✅ **Each column should have unique values** (no duplicate data in the same column).

✅ **Each column should have atomic (indivisible) values** (no multiple values in one cell).

Example (Before 1NF - Problem: Multiple Values in One Cell)

| Student_ID | Name | Courses |

|---|---|---|

| 101 | Alice | Math, Science |

| 102 | Bob | English |

❌ **Problem:** One cell contains multiple values (Math, Science).

After Applying 1NF (Fix: Separate Rows for Each Course)

| Student_ID | Name | Course |

|------------|--------|---------|

| 101 | Alice | Math |

| 101 | Alice | Science |

| 102 | Bob | English |

✅ Now, each column has atomic values, and there are **no duplicate values**.

2️⃣ Second Normal Form (2NF) – Remove Partial Dependencies

✅ **Table should be in 1NF**.

✅ **Every non-key column should depend on the entire primary key, not just part of it.**

Example (Before 2NF - Problem: Partial Dependency)

| Order_ID | Product_ID | Product_Name | Order_Date |

|----------|-----------|--------------|------------|

| 1 | P101 | Laptop | 2024-03-08 |

| 2 | P102 | Mobile | 2024-03-08 |

❌ **Problem:**

- **Product_Name** depends only on **Product_ID**, not on **Order_ID**.

- **Solution:** Split into two tables:

After Applying 2NF (Fix: Separate Product & Order Tables)

Order Table:

| Order_ID | Order_Date |

|----------|------------|

| 1 | 2024-03-08 |

| 2 | 2024-03-08 |

Product Table:

| Product_ID | Product_Name |

|------------|--------------|

| P101 | Laptop |

| P102 | Mobile |

✅ Now, **each table depends entirely on its primary key**.

---

*Third Normal Form (3NF) – Remove Transitive Dependencies**

✅ **Table should be in 2NF**.

✅ **Non-key columns should depend ONLY on the primary key, not on another non-key column.**

Example (Before 3NF - Problem: Transitive Dependency)

| Student_ID | Name | Department | Department_Location |

|---|---|---|---|

| 1 | Alice | IT | Building A |

| 2 | Bob | HR | Building B |

❌ **Problem:**

- **Department_Location** depends on **Department**, not directly on **Student_ID**.

After Applying 3NF (Fix: Create a Separate Department Table)

Student Table:

| Student_ID | Name | Department |

|------------|-------|------------|

| 1 | Alice | IT |

| 2 | Bob | HR |

Department Table:

| Department | Department_Location |

|------------|--------------------|

| IT | Building A |

| HR | Building B |

✅ Now, **each column depends only on the primary key**.

---

*Boyce-Codd Normal Form (BCNF) – Stronger 3NF**

✅ **Table should be in 3NF**.

✅ **If there are multiple candidate keys, one of them should be chosen as the primary key**.

Example (Before BCNF - Problem: Multiple Candidate Keys)

| Professor_ID | Course | Department |

|---|---|---|

| 101 | Math | Science |

| 102 | English| Arts |

❌ **Problem:**

- **Course** depends on **Professor_ID** but also on **Department**.

After Applying BCNF (Fix: Split into Two Tables)

Professor Table:

| Professor_ID | Course |

|---|---|

| 101 | Math |

| 102 | English|

Department Table:

| Course | Department |

|---|---|

| Math | Science |

| English| Arts |

✅ Now, each table follows **strict primary key dependency**.

Denormalization – The Opposite of Normalization

🔹 **Denormalization** is when we **combine tables back** to improve **performance and speed**.

🔹 Sometimes, **too much normalization** can slow down queries because we need to **join many tables**.

Example of Denormalization

Instead of having separate **Student** and **Department** tables, we can **merge them**:

| Student_ID | Name | Department | Department_Location |

|---|---|---|---|

| 1 | Alice | IT | Building A |

| 2 | Bob | HR | Building B |

🔻 **Denormalization is used in Data Warehouses and Analytical Databases** where fast reading is more important than avoiding duplicate data.

Final Summary 🎯

- **Normalization** removes **redundancy, improves consistency, and avoids anomalies**.

- **Denormalization** combines tables for **faster query performance**.

✅ **Use Normalization** when data integrity is important.

✅ **Use Denormalization** when performance (speed) is more important than avoiding redundancy.

Hope this makes it super simple! 😊

7]### **Locks in Database – Simple Explanation**

Imagine you are in a library, and you want to read a book. But if

someone else is already reading that book, you have to wait until they return it. This is similar to how **locks** work in a database.

Locks are used to **control access** to data in a database when multiple users or processes are trying to read or modify it at the same time. They help maintain **data consistency and prevent conflicts**.

Types of Locks

Shared Lock (Read Lock)

- Multiple users can **read** the data at the same time.

- No one can **modify** the data while it's locked.

- Example: In a library, many people can read the same book, but no one can write in it.

Exclusive Lock (Write Lock)

- Only **one user** can read and modify the data.

- Others have to wait until the lock is released.

- Example: If you borrow a book from the library, others cannot use it until you return it.

Locking Levels

Locks can be applied at different levels in a database:

| **Lock Level** | **Description** |

|---|---|

| **Row-Level Lock** | Locks a single row in a table. Other rows can still be accessed. |

| **Table-Level Lock** | Locks the entire table, blocking access to all rows. |

| **Database-Level Lock** | Locks the entire database, stopping all operations. |

Problems with Locks

● **Deadlock** – When two processes wait for each other to release locks, causing a cycle.

● **Blocking** – A process has to wait because another process is

holding a lock.

● **Performance Issues** – Too many locks can slow down the database.

Optimistic vs. Pessimistic Locking

● **Optimistic Locking**: Assumes no one else will change the data, so it checks for conflicts before saving.

● **Pessimistic Locking**: Locks the data immediately to prevent others from making changes.

Conclusion

Locks are necessary to **ensure data consistency** but should be used wisely to avoid slow performance. Modern databases use techniques like **multi-version concurrency control (MVCC)** to reduce locking problems.

Sure! Below are all the basic SQL commands with their respective

queries and explanations.

8]SQL Basic query

1. Data Definition Language (DDL) - Defines Structure

CREATE (To create a new table)

```sql
CREATE TABLE Students (
    ID INT PRIMARY KEY,
    Name VARCHAR(50),
    Age INT
);
```

👉 This creates a `Students` table with three columns: `ID`, `Name`, and `Age`.

ALTER (To modify an existing table)

Add a new column:

```sql
ALTER TABLE Students ADD Gender VARCHAR(10);
```

☞ This adds a new column `Gender` to the `Students` table.

Modify a column's data type:

```sql
ALTER TABLE Students MODIFY Age BIGINT;
```

☞ This changes the data type of the `Age` column from `INT` to `BIGINT`.

Rename a column:

```sql
ALTER TABLE Students RENAME COLUMN Gender TO Sex;
```

☞ This renames the column `Gender` to `Sex`.

Delete a column:

```sql
ALTER TABLE Students DROP COLUMN Age;
```

☞ This removes the `Age` column from the `Students` table.

DROP (To delete a table)

```sql
DROP TABLE Students;
```

☞ This permanently deletes the `Students` table from the database.

TRUNCATE (To delete all rows but keep the structure)

```sql
TRUNCATE TABLE Students;
```

☞ This removes all records from the `Students` table but keeps the

table structure.

---

2. Data Manipulation Language (DML) - Modifies Data

INSERT (To add new records)

```sql
INSERT INTO Students (ID, Name, Age) VALUES (1, 'Neha', 22);
```

☞ This inserts a new student with `ID = 1`, `Name = Neha`, and `Age = 22`.

UPDATE (To modify existing records)

```sql
UPDATE Students SET Age = 23 WHERE ID = 1;
```

☞ This updates the `Age` of the student with `ID = 1` to `23`.

DELETE (To remove specific records)

```sql
DELETE FROM Students WHERE ID = 1;
```

☞ This removes the student whose `ID = 1` from the `Students` table.

---

3. Data Query Language (DQL) - Retrieves Data

SELECT (To retrieve data from a table)

Select all records:

```sql
SELECT * FROM Students;
```

☞ This fetches all rows and columns from the `Students` table.

Select specific columns:

```sql
SELECT Name, Age FROM Students;
```

☞ This fetches only the `Name` and `Age` columns from the `Students` table.

Select with a condition:

```sql
SELECT * FROM Students WHERE Age > 20;
```

☞ This fetches all students whose `Age` is greater than `20`.

Select with sorting:

```sql
SELECT * FROM Students ORDER BY Age DESC;
```

☞ This fetches all students sorted by `Age` in descending order.

Select with a limit:

```sql

SELECT * FROM Students LIMIT 5;

```

👉 This fetches only the first `5` records from the `Students` table.

---

4. Data Control Language (DCL) - Manages Permissions

GRANT (To give access to a user)

```sql

GRANT SELECT ON Students TO user1;

```

👉 This allows `user1` to use the `SELECT` command on the `Students` table.

REVOKE (To remove access from a user)

```sql
```

REVOKE SELECT ON Students FROM user1;

```

☞ This removes the `SELECT` permission from `user1`.

---

## **5. Transaction Control Language (TCL) - Manages Transactions**

### **BEGIN TRANSACTION (To start a transaction)**

```sql

BEGIN TRANSACTION;

```

☞ This starts a transaction.

COMMIT (To save changes permanently)

```sql

COMMIT;

```

☞ This permanently saves all the changes made in the transaction.

ROLLBACK (To undo changes made in a transaction)

```sql
ROLLBACK;
```

☞ This reverts all changes made in the transaction.

SAVEPOINT (To create a checkpoint in a transaction)

```sql
SAVEPOINT sp1;
```

☞ This creates a savepoint named `sp1` in the transaction.

```

ROLLBACK TO SAVEPOINT (To undo changes up to a savepoint)

```sql

ROLLBACK TO sp1;

```

☞ This reverts changes only up to the savepoint `sp1`, without rolling back the entire transaction.

Summary:

- **DDL**: `CREATE`, `ALTER`, `DROP`, `TRUNCATE`
- **DML**: `INSERT`, `UPDATE`, `DELETE`
- **DQL**: `SELECT`
- **DCL**: `GRANT`, `REVOKE`
- **TCL**: `COMMIT`, `ROLLBACK`, `SAVEPOINT`

9] **SQL Joins & Their Types (With Examples in Simple Words)**

SQL **JOIN** is used to combine data from two or more tables based

on a related column (like a primary key and a foreign key).

Types of Joins in SQL:

1. **INNER JOIN** – Returns matching rows from both tables.

2. **LEFT JOIN (LEFT OUTER JOIN)** – Returns all rows from the left table and matching rows from the right.

3. **RIGHT JOIN (RIGHT OUTER JOIN)** – Returns all rows from the right table and matching rows from the left.

4. **FULL JOIN (FULL OUTER JOIN)** – Returns all rows when there is a match in either table.

5. **SELF JOIN** – Joins a table with itself.

6. **CROSS JOIN** – Returns the Cartesian product of both tables.

---

1. INNER JOIN (Only Matching Records)

🔹 It returns rows where there is a match in **both tables**.

Example:

```sql

SELECT Employees.EmpID, Employees.Name, Departments.DeptName

FROM Employees
```

INNER JOIN Departments

ON Employees.DeptID = Departments.DeptID;

```

✅ **Result:** Only employees who have a matching department will be displayed.

📌 **If an employee is not assigned to a department, they won't be included!**

---

### **2. LEFT JOIN (All from Left Table, Matches from Right Table)**

🔹 It returns **all rows from the left table** and **only matching rows from the right table**. If no match is found, NULL is returned for right table columns.

**Example:**

```sql

SELECT Employees.EmpID, Employees.Name, Departments.DeptName

FROM Employees

LEFT JOIN Departments

ON Employees.DeptID = Departments.DeptID;

```

✅ **Result:**

- All employees are displayed.

- If an employee doesn't belong to any department, `DeptName` will be `NULL`.

📌 **Use Case:** When we want to see all employees, even if they don't belong to a department.

---

3. RIGHT JOIN (All from Right Table, Matches from Left Table)

🔽 It returns **all rows from the right table** and **only matching rows from the left table**.

Example:

```sql

SELECT Employees.EmpID, Employees.Name, Departments.DeptName

FROM Employees

RIGHT JOIN Departments

ON Employees.DeptID = Departments.DeptID;

```

✅ **Result:**

- All departments are displayed.

- If a department has no employees, `EmpID` and `Name` will be `NULL`.

📌 **Use Case:** When we want to see all departments, even if there are no employees assigned.

---

4. FULL JOIN (Full Outer Join - All from Both Tables)

🔻 It returns **all rows from both tables**. If there is a match, it returns data from both; if there is no match, `NULL` is returned for the missing side.

Example:

```sql

SELECT Employees.EmpID, Employees.Name, Departments.DeptName

FROM Employees

FULL JOIN Departments

ON Employees.DeptID = Departments.DeptID;

```

✅ **Result:**

- All employees and all departments are displayed.

- If an employee doesn't belong to a department, `DeptName` is `NULL`.

- If a department has no employees, `EmpID` and `Name` are `NULL`.

📌 **Use Case:** When we want to see everything, even unmatched data.

---

5. SELF JOIN (Joining a Table to Itself)

🔹 It is used when we need to compare rows in the **same table**.

Example: Find employees who have the same manager

```sql

SELECT A.Name AS Employee, B.Name AS Manager

FROM Employees A

JOIN Employees B

ON A.ManagerID = B.EmpID;

```

✅ **Result:**

- Shows employees along with their manager’s name.

📌 **Use Case:** When employees are stored in the same table but linked via a manager.

---

6. CROSS JOIN (Cartesian Product)

🔷 It returns **every combination** of rows from both tables (Multiplication of both tables' row count).

Example:

```sql

SELECT Employees.Name, Departments.DeptName

FROM Employees

CROSS JOIN Departments;

```

✅ **Result:**

- If Employees has 5 rows and Departments has 3 rows, the result will have `5 × 3 = 15` rows.

📌 **Use Case:** Used when **all possible combinations** of two tables are needed.

💡 **Key Takeaways**

- **INNER JOIN** → Only common data.
- **LEFT JOIN** → All from **left** + matches from right.
- **RIGHT JOIN** → All from **right** + matches from left.
- **FULL JOIN** → Everything.
- **SELF JOIN** → Join the same table.
- **CROSS JOIN** → Every combination.
```

10] Indexing

What is Indexing?

Indexing in SQL is like the **table of contents in a book**. It helps the database **find data faster** without scanning the entire table.

🔹 **Without an index** → The database searches row by row (slow).

🔹 **With an index** → The database jumps directly to the required row (fast).

Types of Indexes in SQL

1. **Primary Index** – Automatically created when a primary key is defined.

2. **Unique Index** – Ensures all values in a column are unique.

3. **Clustered Index** – Sorts and stores the table based on the index.

4. **Non-Clustered Index** – Creates a separate structure to point to the actual table.

5. **Composite Index** – Indexes multiple columns together.

6. **Full-Text Index** – Used for searching text data efficiently.

---

1. Primary Index (Automatically Created)

🔹 When you create a **Primary Key**, SQL **automatically creates an index**.

🔹 It **ensures uniqueness** and **speeds up searches** on the primary key column.

Example:

```sql
CREATE TABLE Employees (
    EmpID INT PRIMARY KEY,
    Name VARCHAR(100),
    Salary INT
);
```

✅ **Here, `EmpID` automatically gets indexed**.

---

2. Unique Index (Prevents Duplicates)

🔹 Ensures that all values in a column are **unique**.

🔹 Works like a primary key but allows **one NULL value**.

Example:

```sql
CREATE UNIQUE INDEX idx_unique_email
ON Employees (Email);
```

✅ **Now, duplicate emails won't be allowed!**

---

3. Clustered Index (Sorts and Stores Data)

🔹 **Physically sorts the table based on the index column**.

🔹 A table **can have only ONE clustered index** because data can be sorted in only one way.

🔹 By default, **Primary Key = Clustered Index**.

Example:

```sql
CREATE CLUSTERED INDEX idx_salary
```

ON Employees (Salary);

```

✅ **Now, data is stored in ascending order of Salary.**

📌 **Use Case:** Faster sorting and searching in range queries like `WHERE Salary > 50000`.

---

## **4. Non-Clustered Index (Separate Index Structure)**

🔹 Unlike clustered indexes, it **doesn't change the table's order**.

🔹 It **stores pointers** to the actual data instead.

🔹 A table **can have multiple non-clustered indexes**.

**Example:**

```sql

CREATE NONCLUSTERED INDEX idx_name

ON Employees (Name);

```

✅ **Now, searching by `Name` is faster!**
```

📌 **Use Case:** When searching for specific values, like `WHERE Name = 'John'`.

---

5. Composite Index (Multiple Columns)

🔹 Used when searching with **multiple columns together**.

🔹 Improves performance in **queries using multiple conditions**.

Example:

```sql
CREATE INDEX idx_composite
ON Employees (Department, Salary);
```

✅ **Now, searches like `WHERE Department = 'IT' AND Salary > 50000` are optimized.**

📌 **Use Case:** When frequently filtering by multiple columns.

---

6. Full-Text Index (For Searching Large Text Data)

🔹 Used for searching text efficiently in large databases.

🔹 Supports advanced text searches like **"contains word" or "starts with"**.

Example:

```sql
CREATE FULLTEXT INDEX idx_description
ON Products (Description);
```

✅ **Now, searching for keywords in `Description` is optimized.**

📌 **Use Case:** Searching in blogs, articles, product descriptions.

How Indexes Improve Performance?

1. **Without Index:** The database searches the entire table (slow).

2. **With Index:** The database jumps to the required row (fast).

💡 **Think of an index like a dictionary's alphabetical order** → You don't scan every word, you jump to the right letter directly!

When to Use Indexes?

✅ Use indexing when:

✓ Searching large datasets frequently.

✓ Filtering with `WHERE`, `JOIN`, or `ORDER BY`.

✓ Ensuring uniqueness in a column.

⊘ Avoid indexing when:

❌ The table is **small** (indexing overhead > benefits).

❌ Data is updated **too frequently** (index maintenance slows inserts/updates).

Conclusion

🔹 **Indexes speed up searches** by allowing the database to locate data quickly.

🔹 **Different types of indexes** serve different purposes (primary, unique, clustered, etc.).

🔹 **Too many indexes can slow down inserts/updates**, so use them **wisely**!