**🔹 RDBMS (Relational Database Management System)**

An **RDBMS (Relational Database Management System)** is a type of database management system that **stores data in tables** (rows and columns) and enforces relationships between tables using **primary keys and foreign keys**.

**🔹 Key Features of RDBMS**

1. **Structured Storage** – Stores data in tables (relations).
2. **ACID Properties** – Ensures **Atomicity, Consistency, Isolation, and Durability**.
3. **Normalization** – Organizes data to reduce redundancy.
4. **Relationships** – Uses **Primary Key & Foreign Key** for table linking.
5. **SQL Support** – Uses **Structured Query Language (SQL)** for queries.
6. **Data Integrity** – Enforces constraints like **Unique, NOT NULL, Foreign Key**.

**🔹 Popular RDBMS Examples**

* **MySQL** 🐬
* **Microsoft SQL Server** 🏢
* **PostgreSQL** 🐘
* **Oracle Database** 🏛️

**🔹 What is Normalization?**

**Normalization** is the process of **organizing data** in an RDBMS to: ✅ **Minimize redundancy** (remove duplicate data).  
✅ **Improve data integrity** (avoid anomalies).  
✅ **Optimize storage and retrieval performance**.

**🔹 Normal Forms in RDBMS**

Normalization is divided into **different normal forms (NF)**. Each form eliminates specific types of anomalies.

| **Normal Form** | **Purpose** | **Rule** |
| --- | --- | --- |
| **1NF (First Normal Form)** | Remove duplicate columns | Ensure atomicity (single values per column). |
| **2NF (Second Normal Form)** | Remove partial dependencies | Every non-key column must depend on the **whole** primary key. |
| **3NF (Third Normal Form)** | Remove transitive dependencies | No column should depend on a non-primary key column. |
| **BCNF (Boyce-Codd NF)** | Stronger 3NF | Every determinant must be a candidate key. |
| **4NF (Fourth Normal Form)** | Remove multi-valued dependencies | No multi-valued dependencies allowed. |
| **5NF (Fifth Normal Form)** | Remove join dependencies | Tables should be decomposed correctly to remove redundancy. |

**🔹 Example of Normalization**

**🎯 Unnormalized Table (UNF)**

OrderID | Customer | Product | Quantity | Supplier

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

1 | Alice | Laptop | 2 | Dell

2 | Bob | Mobile | 1 | Samsung

3 | Alice | Keyboard | 1 | Logitech

1 | Alice | Mouse | 2 | HP

* Repetitive customer names.
* Repeating OrderID for multiple products.

**✅ 1NF (First Normal Form) - Remove Repeating Groups**

Each column must have **atomic (single) values**.

OrderID | Customer | Product | Quantity | Supplier

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

1 | Alice | Laptop | 2 | Dell

1 | Alice | Mouse | 2 | HP

2 | Bob | Mobile | 1 | Samsung

3 | Alice | Keyboard | 1 | Logitech

**✅ 2NF (Second Normal Form) - Remove Partial Dependencies**

Every **non-key column** must depend on the **entire** primary key.

**🔹 Issues:** Customer depends only on OrderID, not on Product.

✅ **Solution**: Split into Orders and OrderDetails tables.

**Orders Table**

OrderID | Customer

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

1 | Alice

2 | Bob

3 | Alice

**OrderDetails Table**

OrderID | Product | Quantity | Supplier

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

1 | Laptop | 2 | Dell

1 | Mouse | 2 | HP

2 | Mobile | 1 | Samsung

3 | Keyboard | 1 | Logitech

**✅ 3NF (Third Normal Form) - Remove Transitive Dependencies**

Every **non-key column** must depend **only** on the primary key.

**🔹 Issues:** Supplier depends on Product, not OrderID.

✅ **Solution**: Create a separate Suppliers table.

**Products Table**

Product | Supplier

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

Laptop | Dell

Mouse | HP

Mobile | Samsung

Keyboard | Logitech

**Final Structure (Normalized Database)**

1. **Orders** (OrderID, Customer)
2. **OrderDetails** (OrderID, Product, Quantity)
3. **Products** (Product, Supplier)

**🔹 Advantages of Normalization**

✅ **Reduces redundancy** – Saves storage space.  
✅ **Improves consistency** – No duplicate data.  
✅ **Prevents anomalies** – No update, insert, or delete anomalies.  
✅ **Enhances query performance** – Well-structured data retrieval.

**🎯 Conclusion**

✔ **RDBMS** organizes data into **tables** with relationships.  
✔ **Normalization** improves **data integrity** and **eliminates redundancy**.  
✔ Follow **1NF → 2NF → 3NF** for efficient database design.

**🔹 Subqueries in RDBMS (SQL)**

A **subquery** is a SQL query inside another query. It is used to retrieve data that will be used by the main query.

📌 **Also Known As:** Nested Query, Inner Query.  
📌 **Use Cases:** Filtering, aggregation, and data lookup.

**🔹 Types of Subqueries**

1. **Scalar Subquery** – Returns a **single value**.
2. **Single-Row Subquery** – Returns **one row**.
3. **Multi-Row Subquery** – Returns **multiple rows**.
4. **Correlated Subquery** – Dependent on the outer query.
5. **Nested Subquery** – A subquery inside another subquery.

**🔹 Example Database: Employee Table**

| **EmpID** | **Name** | **Department** | **Salary** |
| --- | --- | --- | --- |
| 1 | Alice | IT | 50000 |
| 2 | Bob | HR | 60000 |
| 3 | Carol | IT | 70000 |
| 4 | Dave | Finance | 55000 |
| 5 | Eve | IT | 65000 |

**🔹 1. Scalar Subquery (Returns a Single Value)**

Find employees **who earn more than the average salary**.

SELECT Name, Salary

FROM Employee

WHERE Salary > (SELECT AVG(Salary) FROM Employee);

✅ **Explanation:**

* The subquery (SELECT AVG(Salary) FROM Employee) calculates the **average salary**.
* The outer query selects employees earning **above this value**.

**🔹 2. Single-Row Subquery (Returns One Row)**

Find the **employee with the highest salary**.

SELECT Name, Salary

FROM Employee

WHERE Salary =(SELECT MAX(Salary) FROM Employee);

✅ **Explanation:**

* The subquery returns the **highest salary**.
* The outer query finds the **employee with that salary**.

**🔹 3. Multi-Row Subquery (Returns Multiple Rows)**

Find employees who **work in the same department as ‘Alice’**.

SELECT Name, Department

FROM Employee

WHERE Department = (SELECT Department FROM Employee WHERE Name = 'Alice');

✅ **Explanation:**

* The subquery gets **Alice's department**.
* The outer query selects all **employees in that department**.

**🔹 4. Correlated Subquery (Row-by-Row Comparison)**

Find employees who earn **above the average salary of their department**.

SELECT Name, Department, Salary

FROM Employee e1

WHERE Salary > (

SELECT AVG(Salary)

FROM Employee e2

WHERE e1.Department = e2.Department

);

✅ **Explanation:**

* The subquery calculates the **average salary for each department**.
* The outer query filters employees earning **above their department's average**.

**🔹 5. Nested Subquery**

Find employees **in IT who earn more than the average salary of the entire company**.

SELECT Name, Salary

FROM Employee

WHERE Department = 'IT'

AND Salary > (

SELECT AVG(Salary)

FROM Employee

);

✅ **Explanation:**

* The **inner subquery** calculates the **company-wide average salary**.
* The **outer query** filters IT employees earning **above this salary**.

**🔹 Using Subqueries with IN, ANY, ALL**

**Using IN (Multiple Values)**

Find employees who **work in the same department as Bob or Dave**.

SELECT Name

FROM Employee

WHERE Department IN (

SELECT Department

FROM Employee

WHERE Name IN ('Bob', 'Dave')

);

✅ **Explanation:**

* The **subquery gets Bob and Dave’s departments**.
* The **outer query selects employees** in those departments.

**Using ANY (Comparison with Multiple Values)**

Find employees **earning more than at least one HR employee**.

SELECT Name, Salary

FROM Employee

WHERE Salary > ANY (

SELECT Salary

FROM Employee

WHERE Department = 'HR'

);

✅ **Explanation:**

* The **subquery gets salaries of HR employees**.
* The **outer query filters employees** earning more than **any** HR employee.

**Using ALL (Comparison with All Values)**

Find employees earning **more than all HR employees**.

SELECT Name, Salary

FROM Employee

WHERE Salary > ALL (

SELECT Salary

FROM Employee

WHERE Department = 'HR'

);

✅ **Explanation:**

* The **subquery gets the highest HR salary**.
* The **outer query selects employees** earning **above that value**.

**🔹 Key Differences: IN, ANY, ALL**

| **Clause** | **Returns True If** |
| --- | --- |
| IN | The value is **in** the subquery result. |
| ANY | The value is **greater/less than at least one** result. |
| ALL | The value is **greater/less than all** results. |

**🔹 Subqueries vs. Joins**

| **Feature** | **Subquery** | **Join** |
| --- | --- | --- |
| Performance | Slower for large datasets | Faster with indexed tables |
| Readability | Easier to understand | Complex queries can be harder |
| Use Case | When working with **aggregations** | When retrieving **related data** |

🚀 **Use Joins** when working with multiple tables efficiently.  
📌 **Use Subqueries** for complex conditions or when filtering based on aggregated values.

**🎯 Conclusion**

✔ **Subqueries** allow powerful filtering and aggregation.  
✔ **Use IN, ANY, ALL** for handling multiple results.  
✔ **Correlated subqueries** process each row individually.  
✔ **Nested subqueries** can make complex queries more readable.

Here are all the **SQL Server (MSSQL)** queries for **JOINS**, using Employees and Departments tables.

## ****🔹 Create Tables in MSSQL****

CREATE TABLE Employees (

EmpID INT PRIMARY KEY,

Name NVARCHAR(50),

DeptID INT

);

CREATE TABLE Departments (

DeptID INT PRIMARY KEY,

DeptName NVARCHAR(50)

);

## ****🔹 Insert Sample Data****

INSERT INTO Employees (EmpID, Name, DeptID) VALUES

(1, 'Alice', 101),

(2, 'Bob', 102),

(3, 'Carol', 103),

(4, 'Dave', 101),

(5, 'Eve', NULL);

INSERT INTO Departments (DeptID, DeptName) VALUES

(101, 'IT'),

(102, 'HR'),

(103, 'Finance'),

(104, 'Marketing');

## ****🔹 1. INNER JOIN (Common Data Only)****

SELECT E.Name, D.DeptName

FROM Employees E

INNER JOIN Departments D ON E.DeptID = D.DeptID;

## ****🔹 2. LEFT JOIN (All Left + Matching Right)****

SELECT E.Name, D.DeptName

FROM Employees E

LEFT JOIN Departments D ON E.DeptID = D.DeptID;

## ****🔹 3. RIGHT JOIN (All Right + Matching Left)****

SELECT E.Name, D.DeptName

FROM Employees E

RIGHT JOIN Departments D ON E.DeptID = D.DeptID;

## ****🔹 4. FULL OUTER JOIN (All Data from Both Tables)****

SELECT E.Name, D.DeptName

FROM Employees E

FULL OUTER JOIN Departments D ON E.DeptID = D.DeptID;

## ****🔹 5. CROSS JOIN (Cartesian Product)****

SELECT E.Name, D.DeptName

FROM Employees E

CROSS JOIN Departments D;

## ****🔹 6. SELF JOIN (Finding Employees in the Same Department)****

SELECT E1.Name AS Employee1, E2.Name AS Employee2, E1.DeptID

FROM Employees E1

JOIN Employees E2 ON E1.DeptID = E2.DeptID

WHERE E1.EmpID <> E2.EmpID;

## ****🎯 Additional Notes****

✅ All these queries are **MSSQL-compatible**.  
✅ NULL values will appear in LEFT JOIN, RIGHT JOIN, and FULL OUTER JOIN where no matches exist.  
✅ Use **WHERE conditions** to filter specific results.

Let me know if you need **modifications** or **examples with more tables**! 🚀