#### **1. Basic Terminology**

* **Schema**: Logical structure of the database (tables, views, indexes, etc.)
* **Instance**: Actual content of the database at a point in time
* **DDL (Data Definition Language)**: SQL commands to define schema

#### **2. Common DDL Commands**

| **Command** | **Description** |
| --- | --- |
| CREATE | Create database objects (tables, views, indexes, etc.) |
| ALTER | Modify existing database objects. |
| DROP | Delete database objects |
| TRUNCATE | Remove all records from a table |

#### 3. **Creating a Table**

Syntax:

CREATE TABLE table\_name (column1 datatype [constraints], column2 datatype [constraints],

... PRIMARY KEY (column), FOREIGN KEY (column) REFERENCES other\_table(column));

#### 4. **Common Data Types**

| **Type** | **Description** |
| --- | --- |
| INT | Integer |
| VARCHAR(n) | Variable-length string |
| DATE | Date |
| FLOAT | Floating-point number |
| BOOLEAN | True/False |

#### 5. **Common Constraints**

| **Constraint** | **Use** |
| --- | --- |
| PRIMARY KEY | Uniquely identifies each row |
| FOREIGN KEY | Enforces referential integrity |
| NOT NULL | Prevents null values |
| UNIQUE | Ensures all values in a column are unique |
| CHECK | Ensures values meet a condition |
| DEFAULT | Provides a default value |

#### 6. **Altering a Schema**

Ver1: ALTER TABLE table\_name ADD column\_name datatype;

Ver2: ALTER TABLE table\_name MODIFY column\_name new\_datatype;

Ver3:ALTER TABLE table\_name DROP COLUMN column\_name;

#### **7. Dropping a Schema**

DROP TABLE table\_name;

#### **8**. **Creating Relationships**

CREATE TABLE Orders (OrderID INT PRIMARY KEY, CustomerID INT, FOREIGN KEY (CustomerID) REFERENCES Customers(CustomerID));

On running this query table named order was created in the database.

To show the structure of the table, describe the command

**Desc orders;**

Name Null? Type

---------- -------- ----------

ORDERID NOT NULL NUMBER(38)

CUSTOMERID NUMBER(38)

Here are concise notes on the **INSERT**, **SELECT**, **DELETE**, and **UPDATE** commands in SQL, which are used to manage data in relational databases:

### **9. INSERT Command**

**Purpose**: Adds new records (rows) into a table.

**Syntax**: INSERT INTO table\_name (column1, column2, ...)

VALUES (value1, value2, ...);

Or

INSERT INTO table\_name

VALUES (value1, value2, value3, ...);

**Example**:

INSERT INTO students (id, name, age)

VALUES (1, 'Alice', 20);

**Pin Points to remember**:

* Column list is optional if values are provided for all columns.
* You can insert multiple rows in one command using:
* INSERT INTO table\_name (col1, col2)
* VALUES (val1, val2), (val3, val4), ...;

### **2. SELECT Command**

**Purpose**: Retrieves data from one or more tables.

**Syntax**:

SELECT column1, column2, ...

FROM table\_name

WHERE condition;

**Example**:

SELECT name, age

FROM students

WHERE age > 18;

**Pin Points to remember**:

* SELECT \* returns all columns.
* Can use ORDER BY, GROUP BY, JOIN, LIMIT, etc., for advanced queries.

### **3. DELETE Command**

**Purpose**: Removes existing records from a table.

**Syntax**:

DELETE FROM table\_name

WHERE condition;

**Example**:

DELETE FROM students

WHERE id = 1;

**Pin Points to remember**:

* **Always include a WHERE clause** to avoid deleting all rows.
* To remove all rows without deleting the table:
* DELETE FROM table\_name;

### **4. UPDATE Command**

**Purpose**: Modifies existing data in a table.

**Syntax**:

UPDATE table\_name

SET column1 = value1, column2 = value2, ...

WHERE condition;

**Example**:

UPDATE students

SET age = 21

WHERE name = 'Alice

**5. Joins**

**5.1 Join Operations**

* JOINs can be used to **combine tables**
* Join is a **derivative of the Cartesian** product.
* Equivalent to performing a Selection, using a join predicate as selection formula, over the Cartesian product of the two operand relations.
* One of the **most difficult operations** to implement efficiently in an RDBMS, and **one reason why RDBMSs have an intrinsic performance problem**

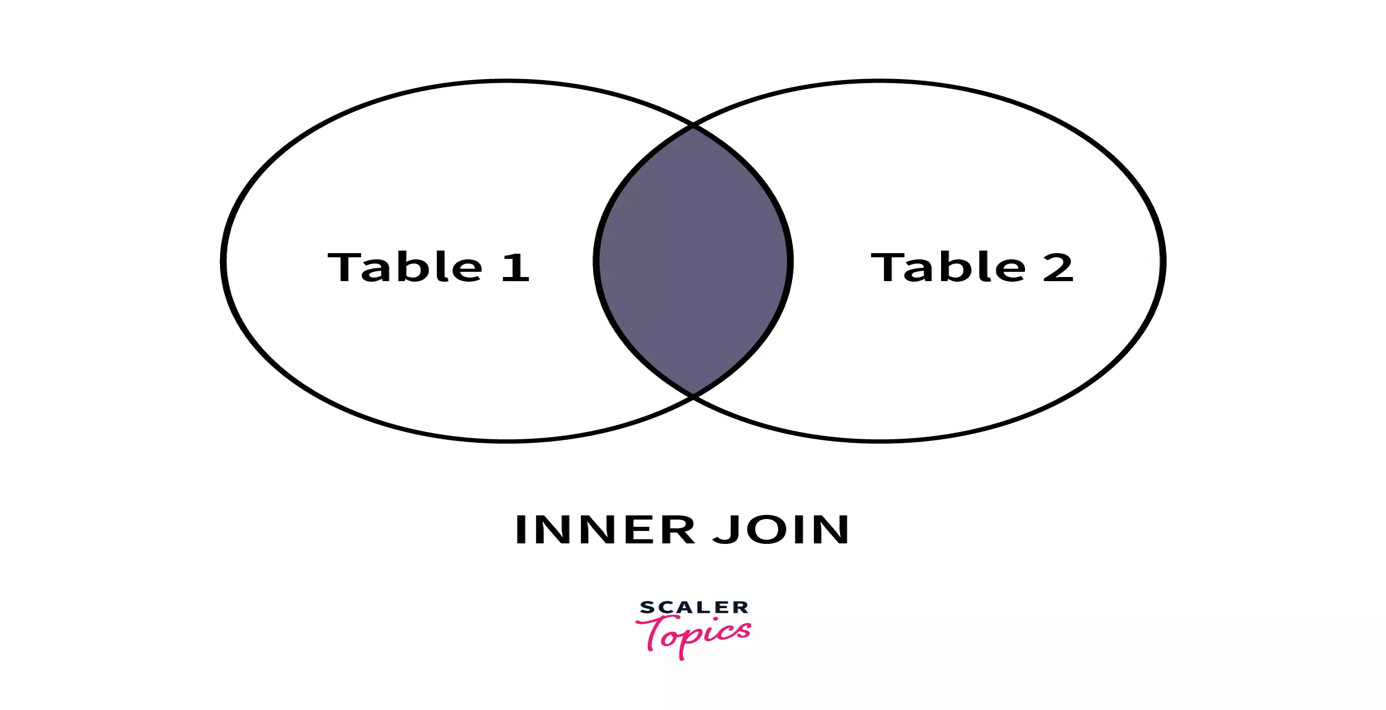
**5.2 Various forms of join operation**

* JOIN (Inner Join): Return rows when there is at least one match in both tables
* LEFT JOIN(Left Outer Join): Return all rows from the left table, even if there are no matches in the right table
* RIGHT JOIN(Right Outer Join): Return all rows from the right table, even if there are no matches in the left table
* FULL JOIN: Return rows when there is a match in

One of the tables

**5.3 SQL INNER JOIN**

* Returns only the rows that have matching values in both tables.
* Matching rows only.



**Syntax**

SELECT column\_name(s) FROM table\_name1 INNER JOIN table\_name2 ON table\_name1.column\_name=table\_name2.column\_name

**INNER JOIN is the same as JOIN. The word "INNER" is optional**

### **Special Case of INNER JOIN: NATURAL JOIN**

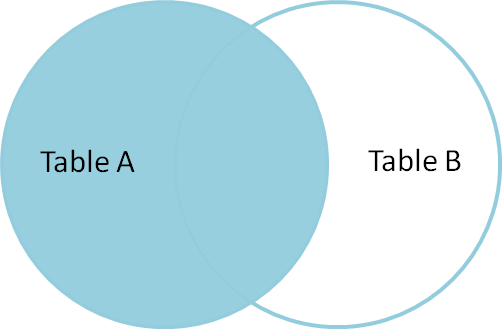
SQL Natural Join is a type of Inner join based on the condition that columns having the same name and datatype are present in both the tables to be joined.

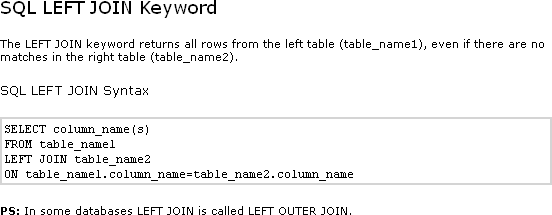
SELECT \* FROM

table-1 NATURAL JOIN table-2;

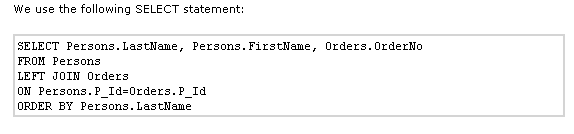
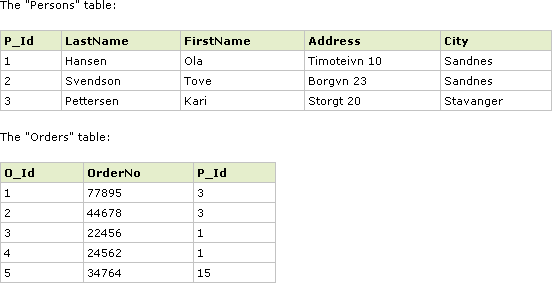
**5.4 Left Join**

* Returns all records from the left table and matched records from the right table. If no match, NULLs on the right.
* All from A matched B.





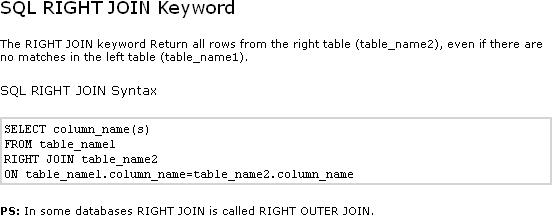
**Example - Left Join**



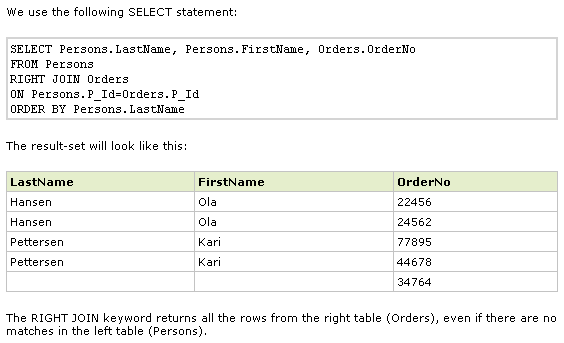
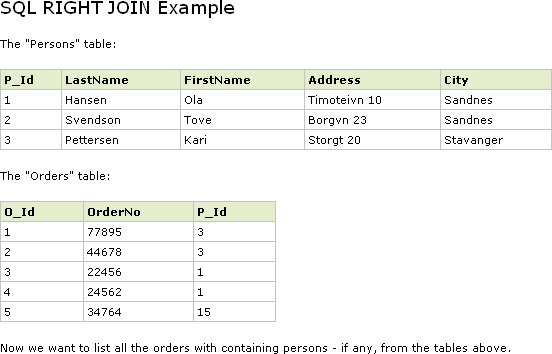
**5.5 Right Join**

* Returns all records from the right table and matched records from the left table. If no match, NULLs on the left.
* All from B matched A.



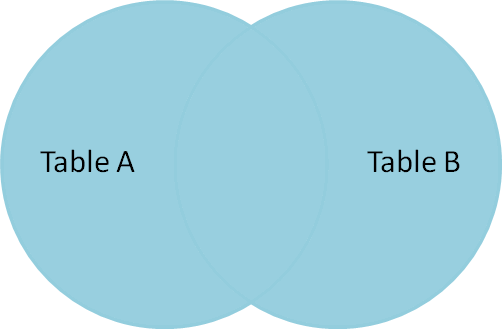
****

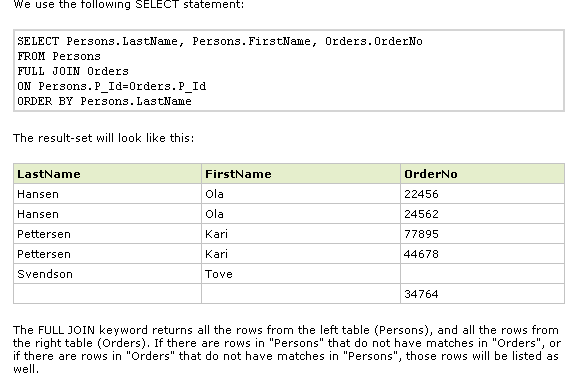
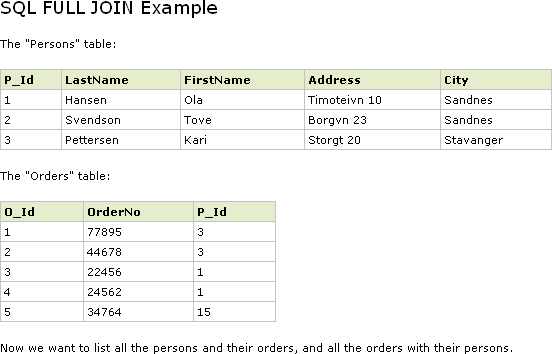
**Example – Right Join**



**5.5 FULL JOIN (FULL OUTER JOIN)**:

* Returns all records when there is a match in either the left or right table. Fills NULLs where there's no match.
* All from A and B.





### 📌 **1.**

**6. Subqueries**

Subqueries are nested queries used to perform operations that require multiple steps.

### **Types of Subqueries**

**6.1.1 Single-row Subquery**

* + Returns only one row.

Example:

SELECT name FROM employees WHERE salary = (SELECT MAX(salary) FROM employees);

* + 1. **Multiple-row Subquery**
  + Returns multiple rows.
  + Use IN, ANY, ALL.

Example:

SELECT name FROM employees WHERE department\_id IN (SELECT id FROM departments WHERE location = 'NY');

* + 1. **Correlated Subquery**

References columns from the outer query.

Executed once per row of the outer query.

Example:

SELECT e.name FROM employees WHERE salary > (SELECT AVG(salary) FROM employees WHERE department\_id = e.department\_id);

### 6.**2 Clauses Supporting Subqueries**

#### **SELECT**

* Used to filter or compute values in the output.

Example:

SELECT name, (SELECT department\_name FROM departments d WHERE d.id = e.department\_id) FROM employees ;

#### **FROM**

* Subquery acts as a table (a derived table).

Example:

SELECT dept\_name, total\_salary FROM (SELECT department\_id, SUM(salary) AS total\_salary FROM employees GROUP BY department\_id) AS dept\_salary

JOIN departments ON dept\_salary.department\_id = departments.id;

#### **WHERE**

* Filters rows using a subquery.

Example:

* SELECT name FROM employees
* WHERE department\_id = (SELECT id FROM departments WHERE name = 'HR');

#### **HAVING**

* Filters groups after aggregation.

Example:

SELECT department\_id, COUNT(\*) FROM employees GROUP BY department\_id HAVING COUNT(\*) > (SELECT AVG(emp\_count FROM (SELECT department\_id, COUNT(\*) AS emp\_count FROM employees GROUP BY department\_id) AS dept\_counts);

#### **EXISTS**

* Tests for the existence of rows.

Example:

SELECT name FROM departments WHERE EXISTS (SELECT 1 FROM employees e WHERE e.department\_id = d.id);

#### **IN, ANY, ALL**

* Compare against multiple results.

Examples:

* -- IN

SELECT name FROM employees WHERE department\_id IN (SELECT id FROM departments WHERE location = 'NY');

* -- ANY

SELECT name FROM employees WHERE salary > ANY (SELECT salary FROM employees WHERE department\_id = 10);

* -- ALL

SELECT name FROM employees WHERE salary > ALL (SELECT salary FROM employees WHERE department\_id = 10);

### **6.3. Subquery Rules**

* Must be enclosed in parentheses.
* Must return a single column if used with =, <, etc.
* Correlated subqueries cannot be used in FROM in some databases (like MySQL).
* Alias subqueries in the FROM clause.

## **7. Views**

**Definition:** A virtual table based on the result set of an SQL statement.

**7.1 Create a View**

CREATE VIEW view\_name AS SELECT column1, column2 FROM table\_name WHERE condition;

### **7.2 Update a View**

In SQL, a **view** is a virtual table created by a query. If you want to **update** or **modify a view**, there are a few different meanings to "updation" depending on context:

If you want to change how a view is defined (e.g., modify the SELECT statement), you use:

CREATE OR REPLACE VIEW view\_name AS SELECT column1, column2

FROM table\_name WHERE condition;

This is the standard way in **SQL**, **Oracle**, and some other databases.

For **SQL Server**, use:

ALTER VIEW view\_name AS SELECT column1, column2

FROM table\_name WHERE condition;

### **7.3 Update Data Through a View**

You can **update records** through a view **only if**:

* The view is **updatable** (i.e., based on a single table, no aggregates or GROUP BY, etc.).
* You have the proper permissions.

UPDATE view\_name SET column1 = value WHERE condition;

### **7.3.1 If the View Is Not Updatable**

Some views (like those involving joins, aggregations, or DISTINCT) **cannot be directly updated**. You would need to update the base tables instead.

### **Example**

**Original View:**

CREATE VIEW employee\_view AS SELECT id, name, department FROM employees

WHERE status = 'active';

**Update View Definition:**

CREATE OR REPLACE VIEW employee\_view AS SELECT id, name, department, salary

FROM employees WHERE status = 'active';

**Update Data via View:**

UPDATE employee\_view SET department = 'HR' WHERE id = 101;

This updates the department in the employees table, assuming the view is updatable.

## **8. Indexes**

**Definition:** A Performance tuning method to speed up data retrieval.

### **Create Index**

CREATE INDEX index\_name ON table\_name (column1, column2);

### **8.2 Drop Index**

DROP INDEX index\_name;

## **9. Stored Procedures**

**Definition:** A group of SQL statements saved to be reused.

### **Create a Stored Procedure**

CREATE PROCEDURE procedure\_name (param1 datatype, param2 datatype)

BEGIN

-- SQL statements

END;

### **9.2** **Execute Stored Procedure**

CALL procedure\_name(param1, param2);

### **9.3** **Drop Stored Procedure**

DROP PROCEDURE procedure\_name;

## **10. Functions**

**Definition:** Similar to procedures, but must return a value.

### **10.1 Create Function**

CREATE FUNCTION function\_name (param1 datatype)

RETURNS datatype

BEGIN

DECLARE result datatype;

-- SQL logic

RETURN result;

END;

### **10.2 Call Function**

SELECT function\_name(param1);

### **10.3 Drop Function**

DROP FUNCTION function\_name;

**Company-based SQL questions**:

### **Schema Creation, INSERT/DELETE/UPDATE/SELECT**

1. **Create a table** for employees with fields: emp\_id, name, department, salary, and joining\_date.
2. **Insert** 3 records into the employees table.
3. **Update** the salary of an employee with emp\_id = 101.
4. **Delete** employees from the Sales department.
5. **Select** all employees who joined after '2023-01-01'.

### J**oins: INNER, LEFT, RIGHT, FULL**

Given two tables: Employees(emp\_id, name, dept\_id) and Departments(dept\_id, dept\_name):

1. Write a query to **INNER JOIN** these tables and show employee names with department names.
2. Use a **LEFT JOIN** to list all employees and their departments, including employees with no department.
3. Use a **RIGHT JOIN** to find all departments and the employees in them, even if no employees exist.
4. Use a **FULL OUTER JOIN** to get all employees and departments, matching if possible.

### **Subqueries & WITH Clauses**

1. Write a query to find employees who earn more than the **average salary** (using a subquery).
2. Using a **correlated subquery**, find employees who earn more than the average salary of their department.
3. Use a **WITH** clause (Common Table Expression) to get the top 3 highest-paid employees.
4. Find the second-highest salary using a subquery.
5. Write a query using **EXISTS** to check if any employee exists in the 'HR' department.

### **Views, Indexes, Stored Procedures, Functions**

1. Create a **view** called HighEarners showing employees with a salary > 100000.
2. Create an **index** on the salary column of the employees table. Explain why you would do that.
3. Write a **stored procedure** to give a bonus (10%) to employees in a given department.
4. Create a **function** that takes a department ID and returns the count of employees in that department.
5. What are the **advantages/disadvantages** of views and indexes?