Detailed Explanation of Joins in SQL

SQL joins are used to retrieve data from two or more tables based on a related column between them. Below is a detailed breakdown of each type of join, with examples based on the provided `customer` and `orders` tables.

Tables Structure

Customer Table: | cid | cname | c_address | |------|---------|-----------| | 1 | aman | Noida | | 2 | deepak | Noida | | 3 | pankaj | Noida | | 4 | raj | Noida | | 5 | aakash | Noida |

Orders Table: | pid | pname | qty | price | cid | |------|----------|-------|-------|------| | 111 | monitor | 20 | 4500 | 1 | | 222 | keyboard | 50 | 250 | 3 | | 333 | mouse | 200 | 120 | 1 | | 444 | speaker | 400 | 600 | 5 | | 555 | usb | 10 | 200 | NULL | | 666 | webcam | 2 | 450 | 6 |

---

1. INNER JOIN

The `INNER JOIN` retrieves records that have matching values in both tables. If there is no match, the records are excluded from the result.

Query:

```sql
SELECT c.cid, c.cname, o.pname, o.qty, o.price
FROM customer c
INNER JOIN orders o ON c.cid = o.cid;
```

Result: | cid | cname | pname | qty | price | |------|---------|----------|-------|-------| | 1 | aman | monitor | 20 | 4500 | | 1 | aman | mouse | 200 | 120 | | 3 | pankaj | keyboard | 50 | 250 | | 5 | aakash | speaker | 400 | 600 |

In this query, only records where `cid` exists in both the `customer` and `orders` tables are shown.

Alternative Ways:

- Using a comma-separated join (older syntax):

  ```sql
  SELECT c.cid, c.cname, o.pname, o.qty, o.price
  FROM customer c, orders o
  WHERE c.cid = o.cid;
  ```

---

2. LEFT JOIN (LEFT OUTER JOIN)

The `LEFT JOIN` retrieves all records from the left table (`customer`), and the matched records from the right table (`orders`). If there is no match, NULL values are returned for columns from the right table.

Query:

```sql
SELECT c.cid, c.cname, o.pname, o.qty, o.price
FROM customer c
LEFT JOIN orders o ON c.cid = o.cid;
```

Result: | cid | cname | pname | qty | price | |------|---------|----------|-------|-------| | 1 | aman | monitor | 20 | 4500 | | 1 | aman | mouse | 200 | 120 | | 2 | deepak | NULL | NULL | NULL | | 3 | pankaj | keyboard | 50 | 250 | | 4 | raj | NULL | NULL | NULL | | 5 | aakash | speaker | 400 | 600 |

In this case, all records from `customer` are included, even if there's no matching `cid` in `orders`.

---

3. RIGHT JOIN (RIGHT OUTER JOIN)

The `RIGHT JOIN` is the opposite of `LEFT JOIN`. It retrieves all records from the right table (`orders`) and the matched records from the left table (`customer`). If there's no match, NULL values are returned for columns from the left table.

Query:

```sql
SELECT c.cid, c.cname, o.pname, o.qty, o.price
FROM customer c
RIGHT JOIN orders o ON c.cid = o.cid;
```

Result: | cid | cname | pname | qty | price | |------|---------|----------|-------|-------| | 1 | aman | monitor | 20 | 4500 | | 1 | aman | mouse | 200 | 120 | | 3 | pankaj | keyboard | 50 | 250 | | 5 | aakash | speaker | 400 | 600 | | NULL | NULL | usb | 10 | 200 | | NULL | NULL | webcam | 2 | 450 |

Here, all records from `orders` are shown, and any missing `cid` in `customer` results in NULL values for those columns.

---

4. FULL JOIN (FULL OUTER JOIN)

MySQL does not directly support `FULL JOIN`. However, it can be emulated using a `UNION` between a `LEFT JOIN` and a `RIGHT JOIN`.

Query:

```sql
SELECT c.cid, c.cname, o.pname, o.qty, o.price
FROM customer c
LEFT JOIN orders o ON c.cid = o.cid
UNION
SELECT c.cid, c.cname, o.pname, o.qty, o.price
FROM customer c
RIGHT JOIN orders o ON c.cid = o.cid;
```

Result: | cid | cname | pname | qty | price | |------|---------|----------|-------|-------| | 1 | aman | monitor | 20 | 4500 | | 1 | aman | mouse | 200 | 120 | | 2 | deepak | NULL | NULL | NULL | | 3 | pankaj | keyboard | 50 | 250 | | 4 | raj | NULL | NULL | NULL | | 5 | aakash | speaker | 400 | 600 | | NULL | NULL | usb | 10 | 200 | | NULL | NULL | webcam | 2 | 450 |

This query returns all records from both tables, with NULL in places where no match is found.

---

5. NATURAL JOIN

`NATURAL JOIN` automatically joins tables based on all columns with the same name in both tables. It doesn't require an `ON` condition but can lead to unexpected results if column names overlap.

Example:

```
SELECT * FROM student NATURAL JOIN marks;
```

If `student` and `marks` share a column (e.g., `rollno`), this query will join on that column.

Drawback:

- Ambiguity in joins if tables have multiple columns with the same name but unrelated data.

Difference between INNER JOIN and NATURAL JOIN: | Feature | INNER JOIN | NATURAL JOIN | |----------|----------|----------| | Join condition | Requires an explicit condition (ON clause) | Automatically uses matching columns | | Flexibility | Offers more control over join conditions | Limited to columns with identical names | | Column names | Need not be identical | Must be identical for join |

---

6. CROSS JOIN (CARTESIAN PRODUCT)

A `CROSS JOIN` returns the Cartesian product of two tables, meaning every row of the first table is combined with every row of the second table.

Query:

```
SELECT * FROM customer CROSS JOIN orders;
```

This will return all combinations of `customer` and `orders`.

---

7. Joining Three Tables

When joining three tables, you simply extend the join condition.

For example, if you want to join `Employee`, `Department`, and `Register`:

```
SELECT e.emp_name, d.dept_name
FROM Employee e
JOIN Register r ON e.emp_id = r.emp_id
JOIN Department d ON r.dept_id = d.dept_id;
```

This joins all three tables based on their related keys.

MySQL Notes for Becoming an Expert: Comprehensive Breakdown

In this guide, we'll go through essential MySQL concepts, with detailed explanations, examples, and operations on joins, constraints, and advanced SQL queries.

Table Setup

We'll use the following three tables to demonstrate various SQL concepts:

Employee Table

```sql
CREATE TABLE Employee (
    emp_id INT NOT NULL AUTO_INCREMENT,
    emp_name VARCHAR(20),
    salary FLOAT,
    PRIMARY KEY(emp_id)
);
```

Department Table

```sql
CREATE TABLE Department (
    dept_id INT PRIMARY KEY,
    dept_name VARCHAR(20)
);
```

Register Table (Used for many-to-many relationships)

```sql
CREATE TABLE Register (
    emp_id INT,
    dept_id INT,
    FOREIGN KEY (emp_id) REFERENCES Employee(emp_id),
    FOREIGN KEY (dept_id) REFERENCES Department(dept_id)
);
```

Insert Sample Data

```sql
-- Insert data into Employee table
INSERT INTO Employee(emp_name, salary) VALUES ('aman', 6700);
INSERT INTO Employee(emp_name, salary) VALUES ('deepak', 9000);
INSERT INTO Employee(emp_name, salary) VALUES ('pankaj', 5000);
INSERT INTO Employee(emp_name, salary) VALUES ('akash', 7800);
INSERT INTO Employee(emp_name, salary) VALUES ('raj', 8400);

-- Insert data into Department table
INSERT INTO Department VALUES (100, 'sales');
INSERT INTO Department VALUES (101, 'HR');
INSERT INTO Department VALUES (102, 'Admin');
INSERT INTO Department VALUES (103, 'Finance');
INSERT INTO Department VALUES (104, 'Accounts');
INSERT INTO Department VALUES (105, 'IT');
INSERT INTO Department VALUES (106, 'marketing');

-- Insert data into Register table
INSERT INTO Register VALUES (1, 101);  -- aman works in HR
```

```sql
INSERT INTO Register VALUES (3, 100);  -- pankaj works in sales
INSERT INTO Register VALUES (4, 106);  -- akash works in marketing
INSERT INTO Register VALUES (2, 104);  -- deepak works in Accounts
```

---

1. Joins in MySQL

Joins are used to combine records from two or more tables based on a related column between them.

Inner Join

Returns records that have matching values in both tables.

Syntax:

```sql
SELECT e.emp_name, d.dept_name
FROM Employee e
INNER JOIN Register r ON e.emp_id = r.emp_id
INNER JOIN Department d ON r.dept_id = d.dept_id;
```

Output:

```
+----------+-----------+
| emp_name | dept_name |
+----------+-----------+
| aman     | HR        |
| pankaj   | sales     |
| akash    | marketing |
| deepak   | Accounts  |
+----------+-----------+
```

Left Join

Returns all records from the left table (Employee), and the matched records from the right table (Department). If no match, NULL values are returned for the unmatched right table records.

Syntax:

```sql
SELECT e.emp_name, d.dept_name
FROM Employee e
LEFT JOIN Register r ON e.emp_id = r.emp_id
LEFT JOIN Department d ON r.dept_id = d.dept_id;
```

Output:

```
+----------+-----------+
| emp_name | dept_name |
+----------+-----------+
| aman     | HR        |
| deepak   | Accounts  |
| pankaj   | sales     |
| akash    | marketing |
```

```
| raj       | NULL       | -- No department for raj
+-----------+------------+
```

Right Join

Returns all records from the right table (Department), and the matched records from the left table (Employee). If no match, NULL values are returned for the unmatched left table records.

Syntax:

```sql
SELECT e.emp_name, d.dept_name
FROM Employee e
RIGHT JOIN Register r ON e.emp_id = r.emp_id
RIGHT JOIN Department d ON r.dept_id = d.dept_id;
```

Output:

```
+-----------+------------+
| emp_name  | dept_name  |
+-----------+------------+
| aman      | HR         |
| deepak    | Accounts   |
| pankaj    | sales      |
| akash     | marketing  |
+-----------+------------+
```

---

3. Advanced Query Examples

Multiple Joins on Three Tables

Here's how to join three tables:

1. `Employee`
2. `Register`
3. `Department`

```sql
SELECT e.emp_name, d.dept_name
FROM Employee e
INNER JOIN Register r ON e.emp_id = r.emp_id
INNER JOIN Department d ON r.dept_id = d.dept_id;
```

---

5. Differences Between Key Concepts

Primary Key vs Unique Key

Primary Key	Unique Key
Ensures uniqueness and does not allow NULL values	Ensures uniqueness but allows NULL values
One primary key per table	Multiple unique keys allowed

Primary Key	Unique Key
Auto-increment can be used	No auto-increment

Delete vs Truncate vs Drop

Delete	Truncate	Drop
Deletes rows based on condition	Removes all rows from the table	Deletes the entire table
Can use WHERE clause	Cannot use WHERE clause	Completely removes table structure

---

Detailed Notes on SQL Self-Join with Examples

Self-Join Overview

A **self-join** is a join in which a table is joined with itself. It is useful when you want to compare rows within the same table. In the case of hierarchical or relational data, like an employee-manager relationship or parent-child categories, self-joins help relate a record to another record in the same table.

Self-Join Syntax

```sql
SELECT t1.column, t2.column
FROM table t1, table t2
WHERE t1.column = t2.column;
```

Or using **JOIN** syntax:

```sql
SELECT t1.column, t2.column
FROM table t1
JOIN table t2 ON t1.column = t2.column;
```

Example Table Structure: Category Hierarchy

```sql
CREATE TABLE categories (
    cat_id INT,
    cat_name VARCHAR(20),
    parent_id INT
);
```

This table stores categories and subcategories. The `parent_id` column relates subcategories to their parent categories. If `parent_id` is 0, the category is a root category (e.g., Mens, Women).

Inserting Data: Root and Subcategories

```sql
INSERT INTO categories VALUES (1, 'Mens', 0);
INSERT INTO categories VALUES (2, 'Women', 0);
INSERT INTO categories VALUES (3, 'Boys', 0);
INSERT INTO categories VALUES (4, 'Girls', 0);
INSERT INTO categories VALUES (5, 'shoes', 1);  -- Subcategory of
'Mens'
INSERT INTO categories VALUES (6, 'shirt', 1);  -- Subcategory of
```

```
'Mens'
INSERT INTO categories VALUES (7, 'cap', 3);    -- Subcategory of
'Boys'
INSERT INTO categories VALUES (8, 'sandals', 2); -- Subcategory of
'Women'
INSERT INTO categories VALUES (9, 'frock', 4);   -- Subcategory of
'Girls'
INSERT INTO categories VALUES (10, 'suit', 2);   -- Subcategory of
'Women'
INSERT INTO categories VALUES (11, 'jewelry', 2); -- Subcategory
of 'Women'
INSERT INTO categories VALUES (12, 'jeans', 3);  -- Subcategory of
'Boys'
```

Selecting All Data

```
SELECT * FROM categories;
```

This shows all the categories with their IDs and parent IDs.

cat_id	cat_name	parent_id
1	Mens	0
2	Women	0
3	Boys	0
4	Girls	0
5	shoes	1
6	shirt	1
7	cap	3
8	sandals	2
9	frock	4
10	suit	2
11	jewelry	2
12	jeans	3

Self-Join: Parent-Child Relationship

We can now use a self-join to match each subcategory with its parent category.

Basic Self-Join Example

```
SELECT *
FROM categories c1, categories c2
WHERE c1.cat_id = c2.parent_id;
```

Explanation:

- `c1` represents the parent category.
- `c2` represents the child (subcategory).
- We match rows in `c1` where `cat_id` (the ID of the parent) is equal to `parent_id` in `c2` (the subcategory's parent).

Output: | cat_id | cat_name | parent_id | cat_id | cat_name | parent_id | |--------|---------|-----------|--------|----------|-----------| | 1 | Mens | 0 | 5 | shoes | 1 | | 1 | Mens | 0 | 6 | shirt | 1 | | 3 | Boys | 0 | 7 | cap | 3 | | 2 | Women | 0 | 8 | sandals | 2 | | 4 | Girls | 0 | 9 | frock | 4 | | 2 | Women | 0 | 10 | suit | 2 | | 2 | Women | 0 | 11 | jewelry | 2 | | 3 | Boys | 0 | 12 | jeans | 3 |

More Descriptive Self-Join Example

```
SELECT c1.cat_id, c1.cat_name, c2.cat_id AS subcat_id, c2.cat_name
AS subcat_name
FROM categories c1
INNER JOIN categories c2
ON c1.cat_id = c2.parent_id;
```

Explanation:

- We use aliases `c1` and `c2` to represent the same table.
- `INNER JOIN` joins categories with their subcategories by comparing `c1.cat_id` (parent category) with `c2.parent_id` (child category).
- We give the columns meaningful aliases such as `subcat_id` and `subcat_name`.

Output: | cat_id | cat_name | subcat_id | subcat_name | |--------|----------|-----------|-------------| | 1 | Mens | 5 | shoes | | 1 | Mens | 6 | shirt | | 3 | Boys | 7 | cap | | 2 | Women | 8 | sandals | | 4 | Girls | 9 | frock | | 2 | Women | 10 | suit | | 2 | Women | 11 | jewelry | | 3 | Boys | 12 | jeans |

Key Points on Self-Join:

- **Self-join** is not a separate type of join. It's simply a `JOIN` where a table is joined to itself.
- **Alias usage**: Aliases are crucial because you need to differentiate between the two instances of the same table.
- **Parent-child relationship**: Self-joins are particularly useful in hierarchical data, where each record might refer to another record in the same table (e.g., categories, employees and managers, etc.).

More SQL Concepts

Adding Constraints

- **Primary Key**: Ensures a unique identifier for each record.
- **Foreign Key**: Ensures referential integrity between tables.
- **Unique**: Ensures all values in a column are distinct.
- **Check**: Ensures that values in a column meet a specific condition.
- **Not Null**: Ensures that a column cannot have `NULL` values.
- **Default**: Sets a default value for a column if no value is provided.

Foreign Key Example

In our example, the `orders` table could have a foreign key that references the `customers` table.

```
CREATE TABLE orders (
    order_id INT PRIMARY KEY AUTO_INCREMENT,
    order_name VARCHAR(20) NOT NULL,
    customer_id INT,
    FOREIGN KEY (customer_id) REFERENCES customers(id)
);
```

Deleting Constraints

To delete a foreign key or other constraints, use the `ALTER TABLE` command:

```
ALTER TABLE orders DROP FOREIGN KEY fk_orders_customers;
ALTER TABLE categories DROP PRIMARY KEY;
ALTER TABLE categories MODIFY COLUMN name VARCHAR(30) NOT NULL;
```

Key Foreign Key Constraints

- **ON DELETE CASCADE**: Automatically deletes rows in the child table when the corresponding rows in the parent table are deleted.
- **ON DELETE SET NULL**: Sets the foreign key column to `NULL` if the referenced row is deleted.

Deleting Child Table

You can delete a child table with constraints, but the constraints need to be considered. For example, deleting a parent row can either:

- Cascade the deletion (removing the associated rows in the child table) if `ON DELETE CASCADE` is used.
- Set the child column to `NULL` if `ON DELETE SET NULL` is used.

In [ ]: