Here are detailed notes on SQL, covering constraints, joins, foreign keys, and key concepts.

1. SQL Constraints

SQL constraints ensure data integrity by limiting the types of data that can be inserted into a table. Here's a breakdown of how to **add and delete constraints** on existing tables.

Primary Key

A `PRIMARY KEY` uniquely identifies each record in a table. It cannot contain `NULL` values.

- **Adding Primary Key:**

  ```
  ALTER TABLE table_name
  ADD PRIMARY KEY (column_name);
  ```

  Example:

  ```
  ALTER TABLE emp
  ADD PRIMARY KEY (id);
  ```

- **Deleting Primary Key:**

  ```
  ALTER TABLE table_name
  DROP PRIMARY KEY;
  ```

NOT NULL

A `NOT NULL` constraint ensures that a column cannot have `NULL` values.

- **Adding NOT NULL Constraint:**

  ```
  ALTER TABLE table_name
  MODIFY column_name datatype NOT NULL;
  ```

- **Deleting NOT NULL Constraint:**

  ```
  ALTER TABLE table_name
  MODIFY column_name datatype NULL;
  ```

Unique Key

A `UNIQUE` constraint ensures that all values in a column are different.

- **Adding Unique Constraint:**

  ```
  ALTER TABLE table_name
  ADD CONSTRAINT constraint_name UNIQUE (column_name);
  ```

- **Deleting Unique Constraint:**

  ```
  ALTER TABLE table_name
  DROP INDEX constraint_name;
  ```

Check

The `CHECK` constraint ensures that all values in a column meet a specific condition.

- **Adding Check Constraint:**

```
ALTER TABLE table_name
ADD CONSTRAINT constraint_name CHECK (condition);
```

- **Deleting Check Constraint:**

```
ALTER TABLE table_name
DROP CONSTRAINT constraint_name;
```

Default

The `DEFAULT` constraint provides a default value for a column when no value is specified.

- **Adding Default Constraint:**

```
ALTER TABLE table_name
ALTER column_name SET DEFAULT value;
```

- **Deleting Default Constraint:**

```
ALTER TABLE table_name
ALTER column_name DROP DEFAULT;
```

Auto Increment

The `AUTO_INCREMENT` constraint allows the automatic generation of a unique value for a column (commonly for primary keys).

- **Setting Auto Increment Starting Value:**

```
ALTER TABLE table_name
AUTO_INCREMENT = value;
```

Foreign Key

A `FOREIGN KEY` is a key used to link two tables together. It references a column in another table.

- **Adding Foreign Key:**

```
ALTER TABLE table_name
ADD CONSTRAINT fk_name FOREIGN KEY (column_name) REFERENCES
parent_table (parent_column);
```

- **Deleting Foreign Key:**

```
ALTER TABLE table_name
DROP FOREIGN KEY fk_name;
```

Composite Key

A `Composite Key` consists of two or more columns that uniquely identify a record.

- **Adding Composite Key:**

```
ALTER TABLE table_name
ADD PRIMARY KEY (column1, column2);
```

- **Deleting Composite Key:**

```
ALTER TABLE table_name
DROP PRIMARY KEY;
```

2. MySQL Joins

A `JOIN` clause is used to combine rows from two or more tables based on a related column between them.

Types of Joins:

- **INNER JOIN**: Returns records that have matching values in both tables.

```
SELECT * FROM table1
INNER JOIN table2
ON table1.column = table2.column;
```

- **LEFT JOIN (LEFT OUTER JOIN)**: Returns all records from the left table and matched records from the right table. Records without a match will have `NULL`.

```
SELECT * FROM table1
LEFT JOIN table2
ON table1.column = table2.column;
```

- **RIGHT JOIN (RIGHT OUTER JOIN)**: Returns all records from the right table and matched records from the left table. Records without a match will have `NULL`.

```
SELECT * FROM table1
RIGHT JOIN table2
ON table1.column = table2.column;
```

- **FULL OUTER JOIN**: Returns all records when there is a match in either table, with unmatched rows being `NULL`.

```
SELECT * FROM table1
FULL OUTER JOIN table2
ON table1.column = table2.column;
```

- **CROSS JOIN**: Returns the Cartesian product of both tables (i.e., all possible combinations of rows).

```
SELECT * FROM table1
CROSS JOIN table2;
```

- **SELF JOIN**: A self join is a regular join, but the table is joined with itself.

```
SELECT a.column_name, b.column_name
FROM table_name a, table_name b
WHERE condition;
```

Example Using Joins:

Let's say we have two tables `customer` and `orders`:

- **customer**:

```
CREATE TABLE customer (
    cid INT PRIMARY KEY AUTO_INCREMENT,
    cname VARCHAR(20) NOT NULL,
    c_address VARCHAR(20) DEFAULT 'Noida'
);
```

- **orders**:

```
CREATE TABLE orders (
    pid INT UNIQUE,
    pname VARCHAR(20) NOT NULL,
    qty INT,
    price FLOAT,
    cid INT,
    FOREIGN KEY (cid) REFERENCES customer(cid)
);
```

- **Inner Join Example**:

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

---

3. Foreign Key Concepts

A **Foreign Key** is used to enforce a link between two tables. It acts as a reference to the primary key in another table.

Key Points:

- **Multiple Foreign Keys**: Yes, a table can have multiple foreign keys.

  Example:

```
ALTER TABLE orders
ADD CONSTRAINT fk_customer FOREIGN KEY (cid) REFERENCES
customer(cid),
ADD CONSTRAINT fk_product FOREIGN KEY (pid) REFERENCES
products(pid);
```

- **Delete Child Table**: You can't delete the parent table (with the foreign key reference) unless the foreign key relationship is dropped first.

- **Foreign Key Conditions**:

  - `ON DELETE CASCADE`: Deletes child records if the parent record is deleted.
  - `ON UPDATE CASCADE`: Updates child records if the parent record is updated.
  - `ON DELETE SET NULL`: Sets the foreign key field to `NULL` if the parent record is deleted.

Foreign Key Example:

```
CREATE TABLE orders (
    pid INT UNIQUE,
    pname VARCHAR(20) NOT NULL,
    qty INT,
    price FLOAT,
    cid INT,
    FOREIGN KEY (cid) REFERENCES customer(cid) ON DELETE CASCADE
);
```

4. InnoDB and Foreign Key Naming

- **InnoDB**: It is a storage engine for MySQL that supports transactions, foreign key constraints, and row-level locking.
- **IBFK_1**: This is a system-generated name for foreign keys in InnoDB. You can specify your own name for clarity.

  To remove a foreign key:

  ```
  ALTER TABLE orders
  DROP FOREIGN KEY IBFK_1;
  ```

5. Difference Between Primary Key and Unique Key

Feature	Primary Key	Unique Key
Uniqueness	Ensures unique values	Ensures unique values
NULL Values	Cannot contain NULL	Can contain a single NULL value
Indexing	Creates a clustered index	Creates a non-clustered index
Number Allowed	Only one primary key per table	Multiple unique keys allowed

6. Questions Answered

- **Can we make multiple foreign keys in a single table?**
  - Yes, a table can have multiple foreign keys referencing different parent tables.
- **Can we delete the child table?**
  - You cannot delete the parent record without handling the foreign key constraints.
- **Purpose of Foreign Key**:
  - Enforces referential integrity between tables by ensuring that a value in one table corresponds to a valid value in another.

Final Example Queries:

- **Insert into `customer` table**:

  ```
  INSERT INTO customer (cname) VALUES ('aman'), ('deepak'),
  ('pankaj'), ('raj'), ('aakash');
  ```

- **Insert into `orders` table**:

```
INSERT INTO orders (pid, pname, qty, price, cid)
VALUES (111, 'monitor', 20, 4500, 1), (222, 'keyboard', 50,
250, 3);
```

- **Select from Joined Tables**:

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

Comprehensive SQL Guide: Constraints, Joins, Foreign Keys, and Advanced Concepts

This guide provides detailed notes and examples to help you become an SQL expert, focusing on **MySQL**. It covers various SQL constraints, operations to add and delete them, comprehensive explanations of **Joins**, **Foreign Keys**, and other advanced concepts. All examples are based on the following tables:

```
CREATE TABLE customer (
    cid INT PRIMARY KEY AUTO_INCREMENT,
    cname VARCHAR(20) NOT NULL,
    c_address VARCHAR(20) DEFAULT 'Noida'
);

CREATE TABLE orders (
    pid INT UNIQUE,
    pname VARCHAR(20) NOT NULL,
    qty INT,
    price FLOAT,
    cid INT,
    FOREIGN KEY (cid) REFERENCES customer(cid)
);
```

Table of Contents

---

1. SQL Constraints

SQL **constraints** enforce rules on the data in tables to maintain data integrity and accuracy. Below are the primary constraints in MySQL, how to add and delete them, along with examples.

1.1 Primary Key

Definition: A `PRIMARY KEY` uniquely identifies each record in a table. It must contain unique values and cannot contain `NULL` values.

Purpose: Ensures each row can be uniquely identified, facilitating efficient data retrieval and relationships between tables.

Syntax:

```
-- During Table Creation
CREATE TABLE table_name (
    column1 datatype PRIMARY KEY,
    column2 datatype,
    ...
);

-- Adding Primary Key to Existing Table
ALTER TABLE table_name
ADD PRIMARY KEY (column_name);
```

Example:

```
-- Table already has PRIMARY KEY
CREATE TABLE customer (
    cid INT PRIMARY KEY AUTO_INCREMENT,
    cname VARCHAR(20) NOT NULL,
    c_address VARCHAR(20) DEFAULT 'Noida'
);

-- Adding PRIMARY KEY to `orders` table if not already present
ALTER TABLE orders
ADD PRIMARY KEY (pid);
```

Deleting Primary Key:

```
ALTER TABLE table_name
DROP PRIMARY KEY;
```

Example:

```
ALTER TABLE emp
DROP PRIMARY KEY;
```

1.2 NOT NULL

Definition: The `NOT NULL` constraint ensures that a column cannot have `NULL` values.

Purpose: Enforces that essential data is always provided, preventing incomplete records.

Syntax:

```
-- During Table Creation
CREATE TABLE table_name (
    column1 datatype NOT NULL,
    column2 datatype,
    ...
);

-- Adding NOT NULL Constraint to Existing Table
ALTER TABLE table_name
MODIFY column_name datatype NOT NULL;
```

Example:

```
CREATE TABLE orders (
    pid INT UNIQUE,
    pname VARCHAR(20) NOT NULL,
    qty INT,
    price FLOAT,
    cid INT,
    FOREIGN KEY (cid) REFERENCES customer(cid)
);

-- Adding NOT NULL to `city` column in `customer` table
ALTER TABLE customer
MODIFY c_address VARCHAR(20) NOT NULL;
```

Deleting NOT NULL Constraint:

```
ALTER TABLE table_name
MODIFY column_name datatype NULL;
```

Example:

```
ALTER TABLE emp
MODIFY name VARCHAR(20) NULL;
```

1.3 Unique Key

Definition: The `UNIQUE` constraint ensures that all values in a column are different. Unlike `PRIMARY KEY`, it can accept `NULL` values (only one per column).

Purpose: Prevents duplicate entries in columns where uniqueness is required but not as critical as the primary key.

Syntax:

```
-- During Table Creation
CREATE TABLE table_name (
    column1 datatype UNIQUE,
    column2 datatype,
    ...
);

-- Adding UNIQUE Constraint to Existing Table
ALTER TABLE table_name
ADD CONSTRAINT constraint_name UNIQUE (column_name);
```

Example:

```
CREATE TABLE orders (
    pid INT UNIQUE,
    pname VARCHAR(20) NOT NULL,
    qty INT,
    price FLOAT,
    cid INT,
    FOREIGN KEY (cid) REFERENCES customer(cid)
);

-- Adding UNIQUE constraint to `email` column in `customer` table
ALTER TABLE customer
ADD CONSTRAINT unique_email UNIQUE (email);
```

Deleting Unique Constraint:

```
ALTER TABLE table_name
DROP INDEX constraint_name;
```

Example:

```
ALTER TABLE emp
DROP INDEX email;
```

1.4 Check

Definition: The `CHECK` constraint ensures that all values in a column satisfy a specific condition.

Purpose: Enforces domain integrity by restricting the range of values that can be stored in a column.

Syntax:

```
-- During Table Creation
CREATE TABLE table_name (
    column1 datatype,
    column2 datatype,
    CHECK (condition)
);

-- Adding CHECK Constraint to Existing Table
ALTER TABLE table_name
ADD CONSTRAINT constraint_name CHECK (condition);
```

Example:

```
CREATE TABLE emp (
    id INT,
    name VARCHAR(20),
    salary FLOAT,
    city VARCHAR(20) DEFAULT 'Noida'
);

-- Adding CHECK constraint to ensure salary > 6000
ALTER TABLE emp
ADD CONSTRAINT chk_salary CHECK (salary > 6000);
```

Deleting CHECK Constraint:

```
ALTER TABLE table_name
DROP CONSTRAINT constraint_name;
```

Example:

```
ALTER TABLE emp
DROP CONSTRAINT chk_salary;
```

Postmortem:

- **Purpose**: Ensures data validity by enforcing business rules directly at the database level.
- **Behavior**: Any `INSERT` or `UPDATE` operation violating the `CHECK` condition will fail.
- **MySQL Note**: Enforced from MySQL **8.0** onwards. Earlier versions parse but ignore `CHECK` constraints.

1.5 Default

Definition: The `DEFAULT` constraint provides a default value for a column when no value is specified during `INSERT`.

Purpose: Automatically assigns a standard value to a column, ensuring data consistency.

Syntax:

```
-- During Table Creation
CREATE TABLE table_name (
    column1 datatype DEFAULT default_value,
    column2 datatype,
    ...
);

-- Adding DEFAULT Constraint to Existing Table
ALTER TABLE table_name
ALTER column_name SET DEFAULT default_value;
```

Example:

```
CREATE TABLE customer (
    cid INT PRIMARY KEY AUTO_INCREMENT,
    cname VARCHAR(20) NOT NULL,
    c_address VARCHAR(20) DEFAULT 'Noida'
);

-- Adding DEFAULT constraint to `status` column in `orders` table
ALTER TABLE orders
ADD COLUMN status VARCHAR(10) DEFAULT 'Pending';
```

Deleting Default Constraint:

```
ALTER TABLE table_name
ALTER column_name DROP DEFAULT;
```

Example:

```
ALTER TABLE orders
ALTER COLUMN status DROP DEFAULT;
```

1.6 Auto Increment

Definition: The `AUTO_INCREMENT` constraint automatically generates a unique number for a column, typically used for primary keys.

Purpose: Simplifies the insertion of unique identifiers without manual input.

Syntax:

```
-- During Table Creation
CREATE TABLE table_name (
    column1 INT AUTO_INCREMENT,
    column2 datatype,
    PRIMARY KEY (column1)
);

-- Adding AUTO_INCREMENT to Existing Column
ALTER TABLE table_name
MODIFY column_name INT AUTO_INCREMENT;
```

Example:

```
CREATE TABLE customer (
    cid INT PRIMARY KEY AUTO_INCREMENT,
    cname VARCHAR(20) NOT NULL,
    c_address VARCHAR(20) DEFAULT 'Noida'
);
```

```
-- Setting AUTO_INCREMENT to start from 100
ALTER TABLE customer AUTO_INCREMENT = 100;
```

Postmortem:

- **Purpose**: Ensures each new record has a unique identifier, enhancing data management and relationships.
- **Behavior**: Automatically increments the value each time a new row is inserted.
- **Use Case**: Ideal for primary keys where uniqueness is crucial.

1.7 Foreign Key

Definition: A `FOREIGN KEY` is a key used to link two tables together by referencing the `PRIMARY KEY` of another table.

Purpose: Enforces referential integrity by ensuring that the value in one table corresponds to a valid value in another table.

Syntax:

```
-- During Table Creation
CREATE TABLE child_table (
    column1 datatype,
    column2 datatype,
    FOREIGN KEY (column_name) REFERENCES
parent_table(parent_column)
);

-- Adding Foreign Key Constraint to Existing Table
ALTER TABLE child_table
ADD CONSTRAINT fk_name FOREIGN KEY (column_name) REFERENCES
parent_table(parent_column);
```

Example:

```
CREATE TABLE orders (
    pid INT UNIQUE,
    pname VARCHAR(20) NOT NULL,
    qty INT,
    price FLOAT,
    cid INT,
    FOREIGN KEY (cid) REFERENCES customer(cid)
);

-- Adding another FOREIGN KEY to `supplier` table (assuming it
exists)
ALTER TABLE orders
ADD CONSTRAINT fk_supplier FOREIGN KEY (supplier_id) REFERENCES
supplier(sid);
```

Deleting Foreign Key Constraint:

```
ALTER TABLE table_name
DROP FOREIGN KEY constraint_name;
```

Example:

```
ALTER TABLE orders
DROP FOREIGN KEY fk_customer;
```

Postmortem:

- **Purpose**: Maintains data consistency across related tables.
- **Behavior**: Prevents insertion of orphan records; ensures that foreign key values exist in the referenced primary key.
- **On Delete/Update Actions**: Can define actions like `CASCADE`, `SET NULL`, etc.

1.8 Composite Key

Definition: A `COMPOSITE KEY` is a combination of two or more columns that together uniquely identify a record in a table.

Purpose: Ensures uniqueness across multiple columns where a single column is insufficient.

Syntax:

```
-- During Table Creation
CREATE TABLE table_name (
    column1 datatype,
    column2 datatype,
    column3 datatype,
    PRIMARY KEY (column1, column2)
);

-- Adding Composite Primary Key to Existing Table
ALTER TABLE table_name
ADD PRIMARY KEY (column1, column2);
```

Example:

```
CREATE TABLE order_details (
    order_id INT,
    product_id INT,
    quantity INT,
    price FLOAT,
    PRIMARY KEY (order_id, product_id)
);

-- Deleting Composite Primary Key
ALTER TABLE order_details
DROP PRIMARY KEY;
```

Postmortem:

- **Purpose**: Used when the combination of columns is necessary to ensure uniqueness.
- **Behavior**: Prevents duplicate records based on the combination of specified columns.
- **Use Case**: Common in junction tables for many-to-many relationships.

1.9 Deleting Constraints

Below are methods to delete various constraints in an existing table.

Deleting Primary Key

```
ALTER TABLE table_name
DROP PRIMARY KEY;
```

Example:

```
ALTER TABLE emp
DROP PRIMARY KEY;
```

Deleting Unique Key

```
ALTER TABLE table_name
DROP INDEX constraint_name;
```

Example:

```
ALTER TABLE emp
DROP INDEX email;
```

Deleting NOT NULL Constraint

```
ALTER TABLE table_name
MODIFY column_name datatype NULL;
```

Example:

```
ALTER TABLE emp
MODIFY name VARCHAR(20) NULL;
```

Deleting Foreign Key

```
ALTER TABLE table_name
DROP FOREIGN KEY constraint_name;
```

Example:

```
ALTER TABLE orders
DROP FOREIGN KEY fk_customer;
```

Deleting Composite Key

```
ALTER TABLE table_name
DROP PRIMARY KEY;
```

Example:

```
ALTER TABLE order_details
DROP PRIMARY KEY;
```

1.10 Differences Between Primary Key and Unique Key

Feature	Primary Key	Unique Key
Uniqueness	Ensures unique values	Ensures unique values
NULL Values	Cannot contain `NULL` values	Can contain a single `NULL` value
Number Allowed	Only one `PRIMARY KEY` per table	Multiple `UNIQUE` keys per table

Feature	Primary Key	Unique Key
Indexing	Creates a clustered index by default	Creates a non-clustered index
Purpose	Uniquely identifies each record	Prevents duplicate entries on columns
Example	`PRIMARY KEY (id)`	`UNIQUE (email)`

Key Points:

- A table can have only one `PRIMARY KEY` but multiple `UNIQUE` keys.
- `PRIMARY KEY` columns are implicitly `NOT NULL`, while `UNIQUE` keys can accept `NULL` values.
- `PRIMARY KEY` is often used for row identification, whereas `UNIQUE` is used to enforce uniqueness on other columns.

2. MySQL Joins

Joins are used to combine rows from two or more tables based on related columns. Understanding joins is crucial for querying relational databases effectively.

2.1 Types of Joins

1. **INNER JOIN**
2. **LEFT JOIN (LEFT OUTER JOIN)**
3. **RIGHT JOIN (RIGHT OUTER JOIN)**
4. **FULL OUTER JOIN**
5. **CROSS JOIN**
6. **SELF JOIN**

2.1.1 INNER JOIN

Definition: Returns records that have matching values in both tables.

Syntax:

```sql
SELECT columns
FROM table1
INNER JOIN table2
ON table1.common_column = table2.common_column;
```

Example:

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

Postmortem:

- **Use Case**: Fetching related data that exists in both tables.
- **Behavior**: Only returns rows where there is a match in both tables.
- **Result**: Combines customer names with their respective orders.

2.1.2 LEFT JOIN (LEFT OUTER JOIN)

Definition: Returns all records from the left table and the matched records from the right table. If there is no match, the result is `NULL` on the right side.

Syntax:

```
SELECT columns
FROM table1
LEFT JOIN table2
ON table1.common_column = table2.common_column;
```

Example:

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

Postmortem:

- **Use Case**: Finding all customers and their orders, including those who haven't placed any orders.
- **Behavior**: Returns all customers, with `NULL` for orders where no match exists.
- **Result**: Lists all customers, showing order details where available.

2.1.3 RIGHT JOIN (RIGHT OUTER JOIN)

Definition: Returns all records from the right table and the matched records from the left table. If there is no match, the result is `NULL` on the left side.

Syntax:

```
SELECT columns
FROM table1
RIGHT JOIN table2
ON table1.common_column = table2.common_column;
```

Example:

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

Postmortem:

- **Use Case**: Finding all orders and their corresponding customers, including orders with no associated customer.
- **Behavior**: Returns all orders, with `NULL` for customers where no match exists.
- **Result**: Lists all orders, showing customer names where available.

2.1.4 FULL OUTER JOIN

Definition: Returns all records when there is a match in either left or right table. **Note**: MySQL does not support `FULL OUTER JOIN` directly, but it can be emulated using

`UNION`.

Syntax:

```
SELECT columns
FROM table1
LEFT JOIN table2
ON table1.common_column = table2.common_column
UNION
SELECT columns
FROM table1
RIGHT JOIN table2
ON table1.common_column = table2.common_column;
```

Example:

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

Postmortem:

- **Use Case**: Retrieving all customers and all orders, showing all possible relationships and non-matches.
- **Behavior**: Combines results of `LEFT JOIN` and `RIGHT JOIN` to include all records.
- **Result**: Comprehensive list including all customers and orders, with `NULL` where relationships do not exist.

2.1.5 CROSS JOIN

Definition: Returns the Cartesian product of both tables, meaning all possible combinations of rows.

Syntax:

```
SELECT columns
FROM table1
CROSS JOIN table2;
```

Example:

```
SELECT customer.cname, orders.pname
FROM customer
CROSS JOIN orders;
```

Postmortem:

- **Use Case**: Situations where all combinations are needed, such as generating a matrix of options.

- **Behavior**: Multiplies the number of rows in the first table by the number of rows in the second table.
- **Result**: Each customer is paired with every order, regardless of any relationship.

2.1.6 SELF JOIN

Definition: A join where a table is joined with itself. Useful for hierarchical data or comparing rows within the same table.

Syntax:

```
SELECT a.columns, b.columns
FROM table_name a
JOIN table_name b
ON a.common_column = b.common_column;
```

Example:

```
-- Assuming 'manager_id' in 'employee' table refers to another
employee
SELECT e1.name AS Employee, e2.name AS Manager
FROM employee e1
INNER JOIN employee e2
ON e1.manager_id = e2.id;
```

Postmortem:

- **Use Case**: Hierarchical relationships, such as employees and their managers within the same table.
- **Behavior**: Allows comparison or relationship mapping within the same table.
- **Result**: Lists employees alongside their managers.

2.2 Join Examples

Using the provided `customer` and `orders` tables, let's explore various join operations.

Example 1: INNER JOIN

Query:

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

Explanation:

- Retrieves only those customers who have placed orders.
- Combines `customer` and `orders` based on matching `cid`.

Result:

```
+---------+-----------+--------+
| cname   | pname     | price |
+---------+-----------+--------+
| aman   | monitor  | 4500   |
| pankaj | keyboard | 250    |
| aman   | mouse    | 120    |
| aakash | speaker  | 600    |
+--------+----------+--------+
```

Example 2: LEFT JOIN

Query:

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

Explanation:

- Retrieves all customers, including those without orders.
- For customers without orders, `pname` and `price` will be `NULL`.

Result:

```
+--------+----------+--------+
| cname  | pname    | price |
+--------+----------+--------+
| aman   | monitor  | 4500   |
| deepak | NULL     | NULL   |
| pankaj | keyboard | 250    |
| raj    | NULL     | NULL   |
| aakash | speaker  | 600    |
+--------+----------+--------+
```

Example 3: RIGHT JOIN

Query:

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

Explanation:

- Retrieves all orders, including those without corresponding customers.
- For orders without a matching customer, `cname` will be `NULL`.

Result:

```
+--------+----------+--------+
| cname  | pname    | price |
+--------+----------+--------+
| aman   | monitor  | 4500   |
| pankaj | keyboard | 250    |
| aman   | mouse    | 120    |
| aakash | speaker  | 600    |
| NULL   | usb      | 200    |
+--------+----------+--------+
```

Example 4: FULL OUTER JOIN (Emulated in MySQL)

Query:

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

Explanation:

- Retrieves all customers and all orders, including non-matching rows.
- Combines results of `LEFT JOIN` and `RIGHT JOIN` using `UNION` to eliminate duplicates.

Result:

```
+---------+----------+-------+
| cname   | pname    | price |
+---------+----------+-------+
| aman    | monitor  | 4500  |
| deepak  | NULL     | NULL  |
| pankaj  | keyboard | 250   |
| raj     | NULL     | NULL  |
| aakash  | speaker  | 600   |
| NULL    | usb      | 200   |
+---------+----------+-------+
```

Example 5: CROSS JOIN

Query:

```
SELECT customer.cname, orders.pname
FROM customer
CROSS JOIN orders;
```

Explanation:

- Returns all possible combinations of customers and orders.
- No relationship is considered; it's purely combinatorial.

Result:

```
+---------+----------+
| cname   | pname    |
+---------+----------+
| aman    | monitor  |
| aman    | keyboard |
| aman    | mouse    |
| aman    | speaker  |
| aman    | usb      |
| deepak  | monitor  |
| deepak | keyboard |
| deepak | mouse    |
| deepak | speaker  |
| deepak | usb      |
| pankaj | monitor  |
| pankaj | keyboard |
| pankaj | mouse    |
| pankaj | speaker  |
| pankaj | usb      |
| raj    | monitor  |
| raj    | keyboard |
| raj    | mouse    |
| raj    | speaker  |
| raj    | usb      |
| aakash | monitor  |
| aakash | keyboard |
| aakash | mouse    |
| aakash | speaker  |
| aakash | usb      |
+--------+----------+
```

Example 6: SELF JOIN

Scenario: Suppose we have an `employee` table where each employee can have a manager, who is also an employee.

Query:

```
SELECT e1.name AS Employee, e2.name AS Manager
FROM employee e1
LEFT JOIN employee e2
ON e1.manager_id = e2.id;
```

Explanation:

- Joins the `employee` table with itself.
- Retrieves employees along with their managers.

Result:

```
+------------+----------+
| Employee   | Manager  |
+------------+----------+
| John       | NULL     |
| Alice      | John     |
| Bob        | John     |
| Charlie    | Alice    |
+------------+----------+
```

2.2 Join Examples with Provided Tables

Using the `customer` and `orders` tables, let's perform various join operations.

Example 1: INNER JOIN

Query:

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

Result:

```
+---------+----------+--------+
| cname   | pname    | price |
+---------+----------+--------+
| aman    | monitor  | 4500  |
| pankaj | keyboard | 250   |
| aman    | mouse    | 120   |
| aakash | speaker  | 600   |
+---------+----------+--------+
```

Example 2: LEFT JOIN

Query:

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

Result:

```
+---------+----------+--------+
| cname   | pname    | price |
+---------+----------+--------+
| aman    | monitor  | 4500  |
| deepak | NULL     | NULL  |
| pankaj | keyboard | 250   |
| raj     | NULL     | NULL  |
| aakash | speaker  | 600   |
+---------+----------+--------+
```

Example 3: RIGHT JOIN

Query:

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

Result:

```
+---------+----------+--------+
| cname   | pname    | price |
+---------+----------+--------+
| aman    | monitor  | 4500  |
| pankaj | keyboard | 250   |
| aman    | mouse    | 120   |
| aakash | speaker  | 600   |
| NULL    | usb      | 200   |
+---------+----------+--------+
```

Example 4: FULL OUTER JOIN (Emulated)

Query:

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

Result:

```
+---------+-----------+--------+
| cname   | pname     | price |
+---------+-----------+--------+
| aman    | monitor   | 4500   |
| deepak  | NULL      | NULL   |
| pankaj  | keyboard  | 250    |
| raj     | NULL      | NULL   |
| aakash  | speaker   | 600    |
| NULL    | usb       | 200    |
+---------+-----------+--------+
```

Example 5: CROSS JOIN

Query:

```
SELECT customer.cname, orders.pname
FROM customer
CROSS JOIN orders;
```

Result:

```
+---------+-----------+
| cname   | pname     |
+---------+-----------+
| aman    | monitor   |
| aman    | keyboard  |
| aman    | mouse     |
| aman    | speaker   |
| aman    | usb       |
| deepak  | monitor   |
| deepak  | keyboard  |
| deepak  | mouse     |
| deepak  | speaker   |
| deepak  | usb       |
| pankaj  | monitor   |
| pankaj  | keyboard  |
| pankaj  | mouse     |
| pankaj  | speaker   |
| pankaj  | usb       |
| raj     | monitor   |
| raj     | keyboard  |
| raj     | mouse     |
| raj    | speaker  |
| raj    | usb      |
| aakash | monitor  |
| aakash | keyboard |
| aakash | mouse    |
| aakash | speaker  |
| aakash | usb      |
+--------+----------+
```

Example 6: SELF JOIN

Scenario: Suppose we have an additional table `employee` where each employee can have a manager.

Additional Table:

```
CREATE TABLE employee (
    eid INT PRIMARY KEY AUTO_INCREMENT,
    ename VARCHAR(20) NOT NULL,
    manager_id INT,
    FOREIGN KEY (manager_id) REFERENCES employee(eid)
);
```

Self Join Query:

```
SELECT e1.ename AS Employee, e2.ename AS Manager
FROM employee e1
LEFT JOIN employee e2
ON e1.manager_id = e2.eid;
```

Result:

```
+----------+----------+
| Employee | Manager  |
+----------+----------+
| John     | NULL     |
| Alice    | John     |
| Bob      | John     |
| Charlie  | Alice    |
+----------+----------+
```

---

3. Foreign Key Concepts

3.1 Multiple Foreign Keys

Question: *Can we make multiple foreign keys in a single table?*

Answer: Yes, a table can have multiple foreign keys referencing different parent tables.

Example: Assume we have another table `supplier`:

```
CREATE TABLE supplier (
    sid INT PRIMARY KEY AUTO_INCREMENT,
    sname VARCHAR(20) NOT NULL
);
```

Now, modify the `orders` table to include a `sid` foreign key:

```
ALTER TABLE orders
ADD COLUMN sid INT,
ADD CONSTRAINT fk_supplier FOREIGN KEY (sid) REFERENCES
supplier(sid);
```

Explanation:

- The `orders` table now references both `customer` (`cid`) and `supplier` (`sid`).

3.2 Deleting Child Tables

Question: *Can we delete the child table?*

Answer: Yes, but you must handle foreign key constraints before deletion. Specifically, you need to drop the foreign keys that reference the parent table.

Steps to Delete Child Table:

1. **Drop Foreign Keys**: Remove any foreign key constraints.
2. **Drop Table**: Delete the table.

Example:

```
-- Step 1: Drop Foreign Key
ALTER TABLE orders
DROP FOREIGN KEY fk_customer;

-- Step 2: Drop Table
DROP TABLE orders;
```

Note: If you attempt to drop the child table without removing foreign keys, MySQL will throw an error due to referential integrity constraints.

3.3 Foreign Key Conditions

When defining foreign keys, you can specify actions to take when the referenced data changes. These conditions help maintain referential integrity.

Common Foreign Key Conditions:

1. **ON DELETE CASCADE**: Automatically deletes child records when the parent record is deleted.
2. **ON UPDATE CASCADE**: Automatically updates child records when the parent record's key is updated.
3. **ON DELETE SET NULL**: Sets the foreign key in child records to `NULL` when the parent record is deleted.
4. **ON UPDATE SET NULL**: Sets the foreign key in child records to `NULL` when the parent record's key is updated.
5. **ON DELETE RESTRICT**: Prevents deletion of parent records if there are related child records.

6. **ON UPDATE RESTRICT**: Prevents updating of parent records' keys if there are related child records.

Syntax with Conditions:

```
ALTER TABLE child_table
ADD CONSTRAINT fk_name FOREIGN KEY (column_name) REFERENCES
parent_table(parent_column)
ON DELETE CASCADE
ON UPDATE CASCADE;
```

Example:

```
CREATE TABLE orders (
    pid INT UNIQUE,
    pname VARCHAR(20) NOT NULL,
    qty INT,
    price FLOAT,
    cid INT,
    FOREIGN KEY (cid) REFERENCES customer(cid)
    ON DELETE SET NULL
    ON UPDATE CASCADE
);
```

Explanation:

- **ON DELETE SET NULL**: If a customer is deleted, the `cid` in `orders` is set to `NULL`.
- **ON UPDATE CASCADE**: If a customer's `cid` is updated, all related `orders.cid` values are updated accordingly.

---

4. Advanced Concepts

4.1 InnoDB

Definition: **InnoDB** is a storage engine for MySQL that supports transactions, row-level locking, and foreign key constraints. It is the default storage engine in MySQL versions 5.5 and later.

Key Features:

- **Transactions**: Ensures ACID (Atomicity, Consistency, Isolation, Durability) compliance.
- **Foreign Keys**: Supports referential integrity constraints.
- **Row-Level Locking**: Improves concurrency and performance by locking individual rows rather than entire tables.
- **Crash Recovery**: Automatically recovers from crashes using its redo logs.

Example:

```
-- Ensure InnoDB is being used
SHOW TABLE STATUS WHERE Name = 'orders';
```

Postmortem:

- **Why Use InnoDB**: Provides robust data integrity features and better performance in multi-user environments.
- **Behavior**: Enforces foreign key constraints and supports transactions, making it ideal for applications requiring reliable data management.

4.2 IBFK_1

Definition: `IBFK_1` is a default naming convention used by InnoDB for foreign key constraints when no explicit name is provided. It stands for "InnoDB Foreign Key 1".

Purpose: To uniquely identify foreign key constraints within the database.

Example: When creating a foreign key without specifying a name:

```
CREATE TABLE orders (
    pid INT UNIQUE,
    pname VARCHAR(20) NOT NULL,
    qty INT,
    price FLOAT,
    cid INT,
    FOREIGN KEY (cid) REFERENCES customer(cid)
);
```

The foreign key might be named `ibfk_1` automatically.

Managing Foreign Keys:

- **Dropping Foreign Key with Default Name**:

  ```
  ALTER TABLE orders
  DROP FOREIGN KEY ibfk_1;
  ```

- **Dropping Foreign Key with Custom Name**:

  ```
  ALTER TABLE orders
  DROP FOREIGN KEY fk_customer;
  ```

Postmortem:

- **Importance**: Knowing the constraint name is essential for managing foreign keys, especially when they are auto-named by InnoDB.
- **Best Practice**: Always explicitly name foreign key constraints for easier management.

---

5. Practical Examples and Queries

5.1 Inserting Data

Insert into `customer` Table:

```
INSERT INTO customer (cname) VALUES ('aman');
INSERT INTO customer (cname) VALUES ('deepak');
INSERT INTO customer (cname) VALUES ('pankaj');
INSERT INTO customer (cname) VALUES ('raj');
INSERT INTO customer (cname) VALUES ('aakash');
```

Result:

```
+-----+--------+-----------+
| cid | cname  | c_address |
+-----+--------+-----------+
|   1 | aman   | Noida     |
|   2 | deepak | Noida     |
|   3 | pankaj | Noida     |
|   4 | raj    | Noida     |
|   5 | aakash | Noida     |
+-----+--------+-----------+
```

Insert into `orders` Table:

```
INSERT INTO orders VALUES (111, 'monitor', 20, 4500, 1);
INSERT INTO orders VALUES (222, 'keyboard', 50, 250, 3);
INSERT INTO orders VALUES (333, 'mouse', 200, 120, 1);
INSERT INTO orders VALUES (444, 'speaker', 400, 600, 5);
INSERT INTO orders VALUES (555, 'usb', 10, 200, 6); -- cid=6 does
not exist
```

Explanation:

- The last insert (`cid=6`) violates the foreign key constraint since `cid=6` doesn't exist in the `customer` table.

Attempted Insert with Missing Foreign Key:

```
INSERT INTO orders(pid, pname, qty, price) VALUES (555, 'usb', 10,
200);
```

Result:

- `cid` is implicitly set to `NULL` if not specified.
- Since `cid` has a foreign key constraint referencing `customer(cid)` , and `cid` is `NULL` , this operation will **succeed** if `cid` allows `NULL` .

Final `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 | -- If inserted
after dropping FK
+------+----------+------+-------+------+
```

5.2 Purpose of Foreign Key

Question: *What is the purpose of a foreign key?*

Answer: A **foreign key** is used to establish and enforce a link between the data in two tables. It ensures referential integrity by making sure that the value in the foreign key column corresponds to a valid value in the referenced primary key column of another table.

Benefits:

- **Data Integrity**: Prevents orphan records by ensuring that relationships between tables remain consistent.
- **Relationship Mapping**: Clearly defines how tables are related, making data management more straightforward.
- **Cascading Actions**: Allows automatic updates or deletions in child tables when changes occur in parent tables.

Example:

```
-- Foreign Key in `orders` referencing `customer`
FOREIGN KEY (cid) REFERENCES customer(cid)
```

Usage:

- Ensures that every order is linked to an existing customer.
- Prevents inserting an order with a `cid` that doesn't exist in `customer`.

5.3 Deleting Foreign Keys

Scenario: Suppose you have a foreign key named `fk_customer` in the `orders` table.

Delete Foreign Key:

```
ALTER TABLE orders
DROP FOREIGN KEY fk_customer;
```

Example with Default Name (`ibfk_1`):

```
ALTER TABLE orders
DROP FOREIGN KEY ibfk_1;
```

Explanation:

- **Why Delete**: To remove the referential constraint, allowing operations that were previously restricted (e.g., inserting orders with non-existent `cid`).
- **Impact**: Once the foreign key is deleted, the `orders` table can have `cid` values that do not correspond to any `cid` in the `customer` table, potentially leading to data inconsistency.

5.4 Final Data View

After deleting the foreign key and inserting a record with `cid=6`, the `orders` table allows `cid` to be `NULL` or any value, even if it doesn't exist in `customer`.

Final `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 |
+------+-----------+------+-------+------+
```

Explanation:

- Records with `cid=6` and `NULL` are now allowed since the foreign key constraint has been removed.
- This may lead to inconsistencies if `cid=6` does not exist in the `customer` table.

---

6. Additional Questions and Answers

Q1: Can we make multiple foreign keys in a single table?

Answer: Yes, a table can have multiple foreign keys, each referencing different parent tables.

Example: Assume we have another table `supplier`:

```
CREATE TABLE supplier (
    sid INT PRIMARY KEY AUTO_INCREMENT,
    sname VARCHAR(20) NOT NULL
);
```

Now, modify the `orders` table to include a foreign key to `supplier`:

```
ALTER TABLE orders
ADD COLUMN sid INT,
ADD CONSTRAINT fk_supplier FOREIGN KEY (sid) REFERENCES
supplier(sid);
```

Explanation:

- The `orders` table now references both `customer` (`cid`) and `supplier` (`sid`).
- This allows each order to be associated with a customer and a supplier.

Q2: Can we delete the child table?

Answer: Yes, you can delete a child table. However, you must first remove any foreign key constraints that reference the parent table. Failing to do so will result in an error due to referential integrity rules.

Steps to Delete a Child Table:

1. **Drop Foreign Keys**: Remove any foreign key constraints referencing the parent table.
2. **Drop the Table**: Delete the table.

Example:

```
-- Step 1: Drop Foreign Key
ALTER TABLE orders
DROP FOREIGN KEY fk_customer;

-- Step 2: Drop the Child Table
DROP TABLE orders;
```

Important Notes:

- **Cascade Options**: Depending on how foreign keys are set (e.g., `ON DELETE CASCADE`), deleting records in the parent table can automatically delete related records in the child table.
- **Dependency Management**: Always ensure that dependent tables are handled appropriately to maintain data integrity.

Q3: Foreign Key Conditions

When defining foreign keys, you can specify actions to take when the referenced data changes. These conditions help maintain referential integrity.

Common Foreign Key Conditions:

1. **ON DELETE CASCADE**

   - **Description**: Automatically deletes child records when the parent record is deleted.
   - **Syntax**:

     ```
     ALTER TABLE child_table
     ADD CONSTRAINT fk_name FOREIGN KEY (column_name) REFERENCES
     parent_table(parent_column)
     ON DELETE CASCADE;
     ```

   - **Example**:

     ```
     ALTER TABLE orders
     ADD CONSTRAINT fk_customer FOREIGN KEY (cid) REFERENCES
     customer(cid)
     ON DELETE CASCADE;
     ```

2. **ON UPDATE CASCADE**

   - **Description**: Automatically updates child records when the parent record's key is updated.
   - **Syntax**:

     ```
     ALTER TABLE child_table
     ADD CONSTRAINT fk_name FOREIGN KEY (column_name) REFERENCES
     parent_table(parent_column)
     ON UPDATE CASCADE;
     ```

   - **Example**:

```
ALTER TABLE orders
ADD CONSTRAINT fk_customer FOREIGN KEY (cid) REFERENCES
customer(cid)
ON UPDATE CASCADE;
```

3. **ON DELETE SET NULL**

   - **Description**: Sets the foreign key in child records to `NULL` when the parent record is deleted.
   - **Syntax**:

     ```
     ALTER TABLE child_table
     ADD CONSTRAINT fk_name FOREIGN KEY (column_name) REFERENCES
     parent_table(parent_column)
     ON DELETE SET NULL;
     ```

   - **Example**:

     ```
     ALTER TABLE orders
     ADD CONSTRAINT fk_customer FOREIGN KEY (cid) REFERENCES
     customer(cid)
     ON DELETE SET NULL;
     ```

4. **ON UPDATE SET NULL**

   - **Description**: Sets the foreign key in child records to `NULL` when the parent record's key is updated.
   - **Syntax**:

     ```
     ALTER TABLE child_table
     ADD CONSTRAINT fk_name FOREIGN KEY (column_name) REFERENCES
     parent_table(parent_column)
     ON UPDATE SET NULL;
     ```

   - **Example**:

     ```
     ALTER TABLE orders
     ADD CONSTRAINT fk_customer FOREIGN KEY (cid) REFERENCES
     customer(cid)
     ON UPDATE SET NULL;
     ```

5. **ON DELETE RESTRICT**

   - **Description**: Prevents deletion of parent records if there are related child records.
   - **Syntax**:

     ```
     ALTER TABLE child_table
     ADD CONSTRAINT fk_name FOREIGN KEY (column_name) REFERENCES
     parent_table(parent_column)
     ON DELETE RESTRICT;
     ```

   - **Example**:

     ```
     ALTER TABLE orders
     ADD CONSTRAINT fk_customer FOREIGN KEY (cid) REFERENCES
     customer(cid)
     ON DELETE RESTRICT;
     ```

6. **ON UPDATE RESTRICT**

   - **Description**: Prevents updating of parent records' keys if there are related child records.

- **Syntax**:

  ```
  ALTER TABLE child_table
  ADD CONSTRAINT fk_name FOREIGN KEY (column_name) REFERENCES
  parent_table(parent_column)
  ON UPDATE RESTRICT;
  ```

- **Example**:

  ```
  ALTER TABLE orders
  ADD CONSTRAINT fk_customer FOREIGN KEY (cid) REFERENCES
  customer(cid)
  ON UPDATE RESTRICT;
  ```

Postmortem:

- **ON DELETE CASCADE**: Ideal for maintaining data consistency by removing dependent records automatically.
- **ON DELETE SET NULL**: Useful when dependent records can exist without the parent, but with `NULL` references.
- **ON DELETE RESTRICT**: Prevents accidental deletion of important parent records by ensuring dependencies are handled first.

Q4: What is InnoDB?

Answer: **InnoDB** is a storage engine for MySQL that provides robust transactional support, referential integrity through foreign keys, row-level locking, and crash recovery capabilities. It is the default storage engine in MySQL versions 5.5 and later.

Key Features:

- **Transactions**: Supports `BEGIN`, `COMMIT`, and `ROLLBACK`.
- **Foreign Keys**: Enforces referential integrity.
- **Row-Level Locking**: Enhances concurrency and performance.
- **Crash Recovery**: Automatically recovers from crashes using its logs.

Example: When creating a table, you can specify the storage engine:

```
CREATE TABLE customer (
    cid INT PRIMARY KEY AUTO_INCREMENT,
    cname VARCHAR(20) NOT NULL,
    c_address VARCHAR(20) DEFAULT 'Noida'
) ENGINE=InnoDB;
```

Postmortem:

- **Why Use InnoDB**: Essential for applications requiring data integrity, reliability, and high performance in multi-user environments.
- **Behavior**: Enforces constraints strictly, supports transactions, and optimizes performance with row-level locking.

Q5: What is IBFK_1?

Answer: `IBFK_1` is a default name automatically assigned by MySQL's InnoDB storage engine to a foreign key constraint if no explicit name is provided during creation. It stands for "InnoDB Foreign Key 1".

Purpose: To uniquely identify foreign key constraints within the database.

Example:

```
CREATE TABLE orders (
    pid INT UNIQUE,
    pname VARCHAR(20) NOT NULL,
    qty INT,
    price FLOAT,
    cid INT,
    FOREIGN KEY (cid) REFERENCES customer(cid)
);
```

In this example, if you do not specify a constraint name, MySQL might name it `ibfk_1` by default.

Managing Foreign Keys:

- **Dropping a Foreign Key with Default Name**:

  ```
  ALTER TABLE orders
  DROP FOREIGN KEY ibfk_1;
  ```

- **Dropping a Foreign Key with Custom Name**:

  ```
  ALTER TABLE orders
  DROP FOREIGN KEY fk_customer;
  ```

Best Practice:

- **Explicit Naming**: Always provide meaningful names to foreign key constraints to simplify management.

  ```
  ALTER TABLE orders
  ADD CONSTRAINT fk_customer FOREIGN KEY (cid) REFERENCES
  customer(cid);
  ```

---

6. Practical Examples and Queries

Below are comprehensive examples using the `customer` and `orders` tables, demonstrating various SQL operations.

6.1 Inserting Data

Insert into `customer` Table:

```
INSERT INTO customer (cname) VALUES ('aman');
INSERT INTO customer (cname) VALUES ('deepak');
INSERT INTO customer (cname) VALUES ('pankaj');
INSERT INTO customer (cname) VALUES ('raj');
INSERT INTO customer (cname) VALUES ('aakash');
```

Result:

```
+-----+--------+-----------+
| cid | cname  | c_address |
+-----+--------+-----------+
|   1 | aman   | Noida     |
|   2 | deepak | Noida     |
|   3 | pankaj | Noida     |
|   4 | raj    | Noida     |
|   5 | aakash | Noida     |
+-----+--------+-----------+
```

Insert into `orders` Table:

```
-- Valid Inserts
INSERT INTO orders VALUES (111, 'monitor', 20, 4500, 1);
INSERT INTO orders VALUES (222, 'keyboard', 50, 250, 3);
INSERT INTO orders VALUES (333, 'mouse', 200, 120, 1);
INSERT INTO orders VALUES (444, 'speaker', 400, 600, 5);

-- Invalid Insert: cid=6 does not exist in customer table
INSERT INTO orders VALUES (555, 'usb', 10, 200, 6); -- This will fail if FK is enforced

-- Insert without cid (assuming `cid` allows NULL)
INSERT INTO orders(pid, pname, qty, price) VALUES (555, 'usb', 10, 200);
```

Explanation:

- The first four inserts succeed as `cid` values (1, 3, 1, 5) exist in the `customer` table.
- The fifth insert with `cid=6` fails due to foreign key constraint unless the foreign key allows `NULL` or `cid=6` exists.
- The sixth insert sets `cid` to `NULL` since it's not specified, which is allowed if the foreign key is nullable.

Final `orders` Table After Successful Inserts:

```
+------+----------+------+-------+------+
| 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 |
+------+----------+------+-------+------+
```

Note: The insert with `cid=6` will only succeed if the foreign key constraint is dropped or modified to allow such inserts.

6.2 Purpose of Foreign Key

Purpose: To enforce referential integrity between two related tables, ensuring that the foreign key value in the child table corresponds to an existing primary key value in the parent table.

Benefits:

- **Data Consistency**: Prevents orphan records by ensuring references are valid.
- **Relationship Mapping**: Clearly defines relationships between tables, aiding in complex queries.
- **Automated Integrity**: Facilitates automated actions like cascading deletes or updates.

Example:

```
-- Ensuring that every order's cid exists in the customer table
FOREIGN KEY (cid) REFERENCES customer(cid)
```

Behavior:

- **Valid Insert**: `cid` exists in `customer`.
- **Invalid Insert**: `cid` does not exist in `customer` (unless foreign key is nullable).

6.3 Deleting Foreign Keys

Scenario: You need to remove the foreign key constraint from the `orders` table.

Steps:

1. **Identify Foreign Key Name**: If not explicitly named, find the foreign key name.

   ```
   SHOW CREATE TABLE orders;
   ```

   Output:

   ```
   CREATE TABLE `orders` (
     `pid` int UNIQUE,
     `pname` varchar(20) NOT NULL,
     `qty` int,
     `price` float,
     `cid` int,
     FOREIGN KEY (`cid`) REFERENCES `customer` (`cid`)
   ) ENGINE=InnoDB;
   ```

   If the foreign key is named `ibfk_1`:

   ```
   ALTER TABLE orders
   DROP FOREIGN KEY ibfk_1;
   ```

2. **Delete Foreign Key**:

   ```
   ALTER TABLE orders
   DROP FOREIGN KEY fk_customer;
   ```

   or

   ```
   ALTER TABLE orders
   DROP FOREIGN KEY ibfk_1;
   ```

Example:

```
-- Dropping Foreign Key named 'fk_customer'
ALTER TABLE orders
DROP FOREIGN KEY fk_customer;

-- Alternatively, if auto-named 'ibfk_1'
ALTER TABLE orders
DROP FOREIGN KEY ibfk_1;
```

Postmortem:

- **Purpose**: Removing referential constraints to allow operations that were previously restricted, such as inserting records with non-existent `cid`.
- **Impact**: Potential data inconsistencies as orphan records can be created.

6.4 Final Data View

After deleting the foreign key and inserting a record with `cid=6`, the `orders` table allows `cid` to be `NULL` or any value.

Final `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 |
+-------+-----------+-------+--------+-------+
```

Explanation:

- Records with `cid=6` and `NULL` are now allowed since the foreign key constraint has been removed.
- This may lead to inconsistencies if `cid=6` does not exist in the `customer` table.

Conclusion

This guide has covered critical aspects of SQL, particularly focusing on **MySQL**. By understanding and practicing constraints, joins, and foreign key management, you can ensure data integrity and execute complex queries effectively. Here's a quick recap:

- **Constraints**: Ensure data validity and integrity through rules like `PRIMARY KEY`, `UNIQUE`, `NOT NULL`, `CHECK`, `DEFAULT`, `AUTO_INCREMENT`, `FOREIGN KEY`, and `COMPOSITE KEY`.
- **Joins**: Combine data from multiple tables using `INNER JOIN`, `LEFT JOIN`, `RIGHT JOIN`, `FULL OUTER JOIN`, `CROSS JOIN`, and `SELF JOIN`.
- **Foreign Keys**: Establish and enforce relationships between tables, ensuring referential integrity.

- **Advanced Concepts**: Utilize storage engines like **InnoDB** for robust data management and understand system-generated constraint names like `IBFK_1`.

By mastering these concepts and applying them through practical examples, you'll be well-equipped to handle complex database scenarios and maintain high standards of data integrity in your SQL projects.

In [ ]: