**SQL Mock**

**Q. I have table-a with primary key and table-b with foreign key which referenced to primary key of table-a. If I delete table-a, will it affect table-b?**

Yes, **deleting a row from table-a** (which has the primary key) **can affect table-b** (which has a foreign key referencing table-a) — **depending on how the foreign key constraint is defined**.

Here’s how it works:

**1. Without any special ON DELETE rule (default)**

* If you try to delete a row from table-a that is still being referenced in table-b, the database will **raise an error** (foreign key violation).
* This is to **prevent orphan records** in table-b.

**2. If ON DELETE CASCADE is specified**

* Deleting a row in table-a will automatically **delete all referencing rows in table-b**.
* Useful when you want child records (in table-b) to be removed with the parent (in table-a).

**3. If ON DELETE SET NULL is specified**

* Deleting a row in table-a will **set the foreign key in table-b to NULL**, assuming the column allows nulls.

**4. If ON DELETE SET DEFAULT is specified**

* The foreign key field in table-b is set to its default value.

**Example:**

CREATE TABLE table\_a (

id INT PRIMARY KEY

);

CREATE TABLE table\_b (

id INT,

a\_id INT,

FOREIGN KEY (a\_id) REFERENCES table\_a(id)

);

In this setup:

* Deleting a row in table\_a that is referenced in table\_b.a\_id will result in an error **unless** you define one of the ON DELETE behaviors.

**Q. How to use these on delete in query ? Give me an examples.**

**🔹 1. Using ON DELETE CASCADE**

This will delete child rows in table\_b when the parent row in table\_a is deleted.

CREATE TABLE table\_a (

id INT PRIMARY KEY

);

CREATE TABLE table\_b (

id INT,

a\_id INT,

FOREIGN KEY (a\_id) REFERENCES table\_a(id) ON DELETE CASCADE

);

**🔹 2. Using ON DELETE SET NULL**

This will set the a\_id field to NULL in table\_b if the corresponding row in table\_a is deleted.

CREATE TABLE table\_a (

id INT PRIMARY KEY

);

CREATE TABLE table\_b (

id INT,

a\_id INT,

FOREIGN KEY (a\_id) REFERENCES table\_a(id) ON DELETE SET NULL

);

⚠️ Note: The a\_id column must allow NULL in this case.

CREATE TABLE table\_b (

id INT,

a\_id INT NULL,

FOREIGN KEY (a\_id) REFERENCES table\_a(id) ON DELETE SET NULL

);

**🔹 3. Using ON DELETE SET DEFAULT**

Sets the foreign key to a default value (e.g., 0 or some placeholder id).

CREATE TABLE table\_a (

id INT PRIMARY KEY

);

CREATE TABLE table\_b (

id INT,

a\_id INT DEFAULT 0,

FOREIGN KEY (a\_id) REFERENCES table\_a(id) ON DELETE SET DEFAULT

);

⚠️ Make sure the default value exists in table\_a or doesn't violate the foreign key constraint.

**🔹 4. Adding ON DELETE rule to an existing table**

You can also add it later using ALTER TABLE:

ALTER TABLE table\_b

ADD CONSTRAINT fk\_a\_id

FOREIGN KEY (a\_id) REFERENCES table\_a(id) ON DELETE CASCADE;

**Q. If delete some records from table\_b or drop table\_b, how will it affect table\_a?**

Dropping table\_b or deleting child rows (table\_b) has **no effect** on the parent (table\_a) — the relationship is one-way:  
table\_b → table\_a (via foreign key).

Q. How to create temporary table in MySQL?

In MySQL, a **temporary table** is a table that exists only during the session in which it was created. Once the session ends (or you manually drop the table), the temporary table is automatically deleted.

CREATE TEMPORARY TABLE table\_name (

column1 datatype,

column2 datatype,

);

Note:

* The temporary table will **not copy indexes or constraints.**
* **AS not used like we use while creating views.**

To create a **temporary table from an existing table:**

CREATE TEMPORARY TABLE temp\_table\_name

SELECT \* FROM existing\_table;

Q. How to copy schema into new table from existing table?

CREATE TABLE new\_table\_name LIKE existing\_table\_name;

Q. How to copy whole schema as well as data of existing table into new table?

CREATE TABLE Employees\_Backup AS SELECT \* FROM Employees;

Q. Create temporary table with top 5 records with max salaried persons from employee table.

Create temporary table new\_table as select \* from employee order by salary desc limit 5;

Q. Use all operators in a query.

select \* from emp where sal > 20000 or experience > 3 and name like 'S%' and dept in ('IT','HR') and salary between 40000 and 50000;

Q. Select name column such that for every NULL value, it should pick corresponding value from name1 column.

name name1

Ajinkya Ajinkya1

Monika Monika1

NULL Sonam

Karan Karan1

Praful Praful1

NULL Jayant

You can use the COALESCE() function in MySQL to select the first non-NULL value from a list of columns.

SELECT COALESCE(name, name1) AS final\_name

FROM your\_table\_name;

Q. 3rd highest salaried records.

with emp\_cte as(

select \*,dense\_rank() over(order by salary desc) as rnk from employees

)

select \* from emp\_cte where rnk=3;

select \* from employees where salary = (select distinct(salary) from employees order by salary desc limit 1 offset 2);

Q. 3rd highest salaried records from each departments.

with emp\_cte as(

select \*,dense\_rank() over(partition by department order by salary desc) as rnk from employees

)

select \* from emp\_cte where rnk=3;

select \* from employees where salary = (select distinct(max(salary)) from employees order by group by department salary desc limit 1 offset 2);

Q. How to create view from table? What is real time use case of views?

create view table\_temp as select \* from employees;

create view table\_temp as select \* from employees e inner join dept d on e.deptno=d.deptno;

* You can restrict access to specific columns or rows by giving users access to a view instead of the actual table.
* Useful for hiding sensitive data like salary, passwords, or internal notes.
* If you change the underlying table structure, you can adjust the view instead of updating every application or report.
* Use views to create predefined reports or summaries (e.g., weekly sales, monthly users).
* Helps generate consistent data outputs for dashboards or BI tools.

Q. Windowing functions and their usecases?

row\_number()

rank()

dense\_rank()

lead()

lag()

**Q. Indexing type their real time usecases? How to create indexing on existing table(Syntax)?**

**🔍 1. Clustered Index**

**✅ What It Is:**

* MySQL (specifically InnoDB) **automatically creates a clustered index** on the **primary key**.
* **The actual table data is stored in the index itself** (i.e., rows are physically ordered by the primary key).

**🎯 Real-Time Use Case:**

* When you frequently query by the **primary key** or do range scans (e.g., BETWEEN, <, >).
* Example: Searching for users by user\_id.

**🛠️ Syntax (Usually automatic via PRIMARY KEY):**

ALTER TABLE employees

ADD PRIMARY KEY (id);

**🔗 2. Non-Clustered Index**

**✅ What It Is:**

* A separate structure from the actual table data.
* Contains a pointer (row ID or PK) to locate the data.

**🎯 Real-Time Use Case:**

* Ideal for searching/filtering by **non-primary columns**, e.g., searching by name, email, status.
* Speeds up queries with WHERE, ORDER BY, JOIN, etc.

**🛠️ Syntax to Create:**

CREATE INDEX idx\_employee\_name

ON employees (name);

You can also index multiple columns:

CREATE INDEX idx\_employee\_dept\_salary

ON employees (department, salary);

You can create index while creating the table only:

CREATE TABLE employees (

id INT PRIMARY KEY,

name VARCHAR(100),

department VARCHAR(50),

INDEX idx\_name (name)

);

**🆔 3. Unique Index**

**✅ What It Is:**

* Ensures that all values in the indexed column(s) are **unique**.
* Prevents duplicate entries.

**🎯 Real-Time Use Case:**

* Enforce uniqueness on fields like email, username, or phone\_number.

**🛠️ Syntax to Create:**

CREATE UNIQUE INDEX idx\_unique\_email

ON employees (email);

**🧠 Summary Table**

| **Index Type** | **Description** | **Use Case Example** |
| --- | --- | --- |
| Clustered Index | Data physically stored by index order (primary key) | WHERE id = 1001 |
| Non-Clustered | Separate from table, points to rows | WHERE name = 'John' |
| Unique Index | Ensures column values are unique | email, username, SSN |

**🛠️ How to View Indexes on a Table:**

SHOW INDEXES FROM employees;

Q. Stored procedures and it's types ?

**Stored Procedures in MySQL**

A **Stored Procedure** is a set of SQL statements that can be executed as a single unit. These procedures are stored in the database and can be reused multiple times. They help to **encapsulate complex logic**, **reduce redundancy**, and **improve performance** by reducing the need to send multiple queries from the client application.

* Can contain multiple SQL statements (e.g., SELECT, INSERT, UPDATE, DELETE).
* Can use input (IN), output (OUT), and in-out (INOUT) parameters.
* May modify database objects, perform transactions, and return multiple values.
* Invoked using CALL.

**Types of Stored Procedures**

1. **Simple Stored Procedure**
2. **Parameterized Stored Procedure**
3. **IN, OUT, and INOUT Parameters**
4. **Stored Procedure with Control Flow (Conditionals & Loops)**

**1. Simple Stored Procedure**

A **simple stored procedure** performs a basic task, like retrieving or modifying data.

**🔧 Syntax:**

DELIMITER $$

CREATE PROCEDURE procedure\_name()

BEGIN

-- SQL statements here

SELECT \* FROM employees;

END $$

DELIMITER ;

This procedure doesn't take parameters and just performs a simple operation like fetching records.

**Example:**

DELIMITER $$

CREATE PROCEDURE GetEmployees()

BEGIN

SELECT \* FROM employees;

END $$

DELIMITER ;

**2. Parameterized Stored Procedure**

You can create **parameterized stored procedures** that accept input parameters and use those to modify behavior or query data.

**🔧 Syntax:**

DELIMITER $$

CREATE PROCEDURE procedure\_name(IN param1 datatype)

BEGIN

-- SQL statements here

SELECT \* FROM employees WHERE department = param1;

END $$

DELIMITER ;

**Example:**

DELIMITER $$

CREATE PROCEDURE GetEmployeesByDepartment(IN dept\_name VARCHAR(50))

BEGIN

SELECT \* FROM employees WHERE department = dept\_name;

END $$

DELIMITER ;

You can then call it like this:

CALL GetEmployeesByDepartment('HR');

**3. IN, OUT, and INOUT Parameters**

* **IN parameters** are used to pass data into the procedure.
* **OUT parameters** allow the procedure to send data back to the caller.
* **INOUT parameters** allow the procedure to pass data in and out.

**🔧 Syntax:**

DELIMITER $$

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

BEGIN

-- SQL statements here

SELECT COUNT(\*) INTO param2 FROM employees WHERE department = param1;

END $$

DELIMITER ;

**Example (INOUT Parameter):**

DELIMITER $$

CREATE PROCEDURE UpdateSalary(INOUT emp\_id INT, IN new\_salary DECIMAL(10, 2))

BEGIN

UPDATE employees SET salary = new\_salary WHERE id = emp\_id;

SELECT salary INTO emp\_id FROM employees WHERE id = emp\_id;

END $$

DELIMITER ;

Here, the emp\_id is both an input and output parameter.

**4. Stored Procedure with Control Flow (Conditionals & Loops)**

Stored procedures can include **control-flow logic** like IF, CASE, LOOP, and WHILE to execute complex tasks.

**Example with IF condition:**

DELIMITER $$

CREATE PROCEDURE GetEmployeeBySalary(IN min\_salary DECIMAL(10, 2))

BEGIN

IF min\_salary > 50000 THEN

SELECT \* FROM employees WHERE salary > min\_salary;

ELSE

SELECT \* FROM employees WHERE salary <= min\_salary;

END IF;

END $$

DELIMITER ;

**Example with LOOP:**

DELIMITER $$

CREATE PROCEDURE IncrementSalaries(IN percentage INT)

BEGIN

DECLARE done INT DEFAULT 0;

DECLARE emp\_id INT;

DECLARE cur CURSOR FOR SELECT id FROM employees;

OPEN cur;

read\_loop: LOOP

FETCH cur INTO emp\_id;

IF done THEN

LEAVE read\_loop;

END IF;

UPDATE employees

SET salary = salary + (salary \* percentage / 100)

WHERE id = emp\_id;

END LOOP;

CLOSE cur;

END $$

DELIMITER ;

**How to Call a Stored Procedure**

To **invoke** a stored procedure:

CALL procedure\_name(parameters);

For example:

CALL GetEmployeesByDepartment('Engineering');

**Types of Stored Procedures in Practice**

| **Type** | **Description** | **Example Usage** |
| --- | --- | --- |
| **Simple Stored Procedure** | Performs a simple task (e.g., fetching data) | SELECT, UPDATE, DELETE queries |
| **Parameterized Stored Procedure** | Takes input parameters to control the query behavior | Filter data by department, search by name, etc. |
| **IN Parameter** | Used to pass values into a procedure | Searching for employees by department (IN dept) |
| **OUT Parameter** | Used to return a value from the procedure | Return the total count of employees in a dept |
| **INOUT Parameter** | Both input and output – allows passing and receiving values | Update salary and return updated value |
| **Control Flow Stored Procedure** | Includes IF, LOOP, and CASE for complex logic | Bulk updates, iterations, and conditional queries |

Q. What are functions in SQL?

* Returns a single value (e.g., INT, VARCHAR).
* Cannot modify the database directly (i.e., no UPDATE, INSERT, DELETE).
* Cannot perform transactions.
* Used within SQL queries or expressions (e.g., in a SELECT statement).
* Can be called within other expressions like WHERE, ORDER BY, or HAVING.

e.g.

CREATE FUNCTION GetEmployeeSalary(emp\_id INT)

RETURNS DECIMAL(10, 2)

BEGIN

DECLARE salary DECIMAL(10, 2);

SELECT salary INTO salary FROM employees WHERE id = emp\_id;

RETURN salary;

END;

SELECT id, name, GetEmployeeSalary(id) AS salary FROM employees;

Q. What is the difference between stored procedures and functions?

| **Feature** | **Stored Procedure** | **Function** |
| --- | --- | --- |
| **Purpose** | Performs a specific task like modifying data or performing multiple operations. | Returns a single value, often used for calculations or transformations. |
| **Return Type** | Can return **multiple** values (using OUT parameters or result sets). | Must return a **single value** (e.g., INT, VARCHAR, DATE). |
| **Usage** | Can be invoked using CALL. Can perform SELECT, INSERT, UPDATE, DELETE. | Used as part of an expression or query. Cannot modify database tables directly. |
| **Performance** | Generally, stored procedures may be slower due to overhead in executing procedural logic. | Typically faster when used in queries due to **limited functionality** and simpler logic. |

Q. Exception handling in MySQL

In MySQL, **error handling** is typically done using the DECLARE ... HANDLER syntax within **stored procedures** and **functions**.

DECLARE handler\_type HANDLER FOR condition

statement;

* handler\_type: Specifies the type of handler (CONTINUE, EXIT, or UNDO).
* condition: The type of error or condition that the handler will react to (SQLEXCEPTION, SQLWARNING, NOT FOUND).
* statement: The action or series of actions to take when the exception is raised.

Q. All joins and union and union all on below table.

table\_a table\_b

A B

1 1

1 NULL

1 NULL

0 0

0 3

NULL 4

2 0

Inner join:

|  |  |
| --- | --- |
| Table\_a | Table\_b |
| A | B |
| 1 | 1 |
| 1 | 1 |
| 1 | 1 |
| 0 | 0 |
| 0 | 0 |
| 0 | 0 |
| 0 | 0 |

Left join:

|  |  |
| --- | --- |
| Table\_a | Table\_b |
| A | B |
| 1 | 1 |
| 1 | 1 |
| 1 | 1 |
| 0 | 0 |
| 0 | 0 |
| 0 | 0 |
| 0 | 0 |
| NULL | NULL |
| 2 | NULL |

Right join:

|  |  |
| --- | --- |
| Table\_a | Table\_b |
| A | B |
| 1 | 1 |
| 1 | 1 |
| 1 | 1 |
| NULL | 0 |
| NULL | 0 |
| 0 | 0 |
| 0 | 0 |
| NULL | 3 |
| NULL | 4 |
| 0 | 0 |
| 0 | 0 |

Union:

|  |
| --- |
| Table\_a |
| A |
| 1 |
| 0 |
| NULL |
| 2 |
| 3 |
| 4 |

Union all:

|  |
| --- |
| Table\_a |
| A |
| 1 |
| 1 |
| 1 |
| 0 |
| 0 |
| NULL |
| 2 |
| 1 |
| NULL |
| NULL |
| 0 |
| 3 |
| 4 |
| 0 |