Package, Procedures and Triggers:

the concepts of packages, procedures, and triggers in the context of a relational database like MySQL or Oracle.

Package:

What is it? A package is like a container or a box that holds related things together in a database. It can contain procedures, functions, variables, and other database objects.

Why use it? Packages help organize and manage database code. Instead of scattering code all over, you put related code inside a package, making it easier to find and maintain.

Procedure:

What is it? A procedure is like a predefined set of actions that you can reuse in your database. Think of it as a **mini-program** that you can call whenever you need it.

Or

A procedure is a type of database object that allows you to group SQL statements into a single named block. Procedures are similar to functions, but they don't return values directly. Instead, they can perform a series of SQL statements and can have input and output parameters.

Why use it? Procedures are used to perform common tasks, like inserting data, updating records, or running complex calculations. Instead of writing the same code multiple times, you write it once in a procedure and call the procedure whenever needed.

Trigger:

What is it? A trigger is like an automatic action that happens when something specific occurs in the database. It's like saying, "When this event happens, do this action automatically."

Why use it? Triggers are used to enforce rules or automate tasks. For example, you can create a trigger that automatically updates a timestamp when a record is modified, ensuring that the modification time is always accurate.

In simpler terms:

Packages help you neatly package and store your code.

Procedures are reusable sets of actions.

Triggers are automatic actions triggered by events in the database.

PL/SQL Block Structure

PL/SQL code is structured into blocks. A typical PL/SQL block has three sections:

Declaration Section: Here, you declare variables, constants, types, and cursors. This section is optional.

```
DECLARE
   variable_name datatype [NOT NULL] [:= initial_value];
```

Execution Section: This section contains the code that is executed. It is mandatory and contains SQL statements and PL/SQL code.

```
BEGIN
   -- executable statements
END;
```

Exception Handling Section: This optional section is used to handle runtime errors or exceptions.

```
EXCEPTION
   WHEN exception_name THEN
      -- exception handling statements
END;
```

Basic Syntax

Variables: Variables are declared in the declaration section and can be assigned values using the := operator.

```
DECLARE
   employee_name VARCHAR2(50);
   employee_id   NUMBER := 1001;
```

Constants: Constants are similar to variables but are initialized once and cannot be changed.

```
DECLARE
   pi CONSTANT NUMBER := 3.14159;
```

Procedures:

- let's say we have bunch of SQL statements now by using procedures by single function we can run it all at once
- **Variable**= Something that holds data (x=5)

  We use variable to store unknown data

- In SQL we use it like: (we store the values in variables then later on whenever the value is need to just pass the variable)

  Passinggrade = 50
  SELECT WHERE grade >= Passinggrade

  *Can use **sqlfiddle.com** to work with SQL queries

Variables:

User-defined variables

Declare= we just mentioning that there is going to be a variable in future use
Initialize= Here we specify the value for that variable

Session variable:

A **session variable** in MySQL is a user-defined variable that exists for the duration of a user's session (connection) to the MySQL server. These variables are **unique to each session**, meaning they are **not shared** across multiple users or sessions. Once the session ends (when you disconnect), the session variables are automatically destroyed.

@X is a session variable

Syntax: SET @X=50;

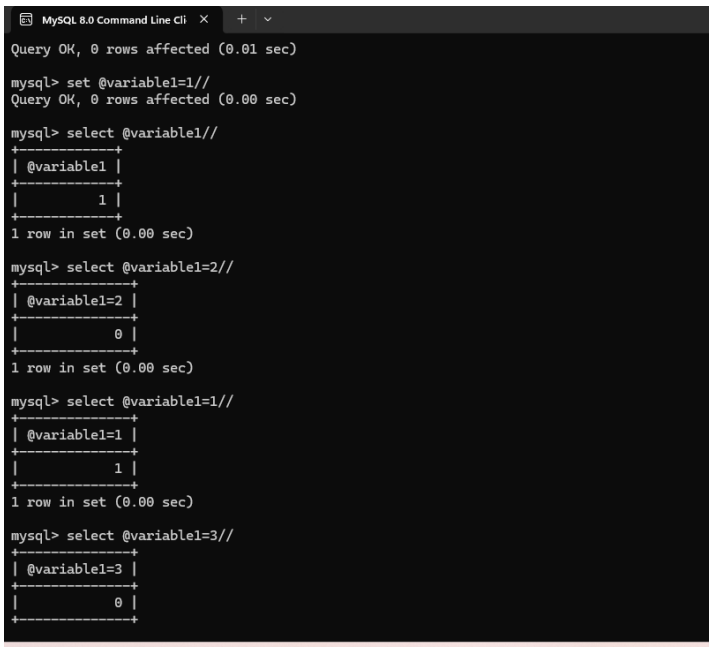
(x is the variable)
(SET is the keyword)
(@ is the identifier so that MySQL understands with @ there is a variable)
(= is the assignment operator)

IN MYSQL (way to create a variable)

SELECT @x := 'charlie'; **{ Here SELECT actually returns values}**

Example1:

- set @variable1 = 1;
- select @variable2 =2;



Example2:

set @variable3 = 3;

set @variable4 = 4;

set @variable5 = 5;

select @variable1, @variable2, @variable3, @variable4, @variable5;

```
mysql> select @variable1, @variable2, @variable3, @variable4, @variable5;
+------------+-----------------------+------------+------------+------------+
| @variable1 | @variable2            | @variable3 | @variable4 | @variable5 |
+------------+-----------------------+------------+------------+------------+
|          1 | NULL                  |          3 |          4 |          5 |
+------------+-----------------------+------------+------------+------------+
1 row in set (0.01 sec)
```

Example3:

set @id= 1;

select @id =1;

{Here it is comparing if in id variable the value is what we mentioned while using SELECT then returns 1 otherwise 0}

{Here but this is just comparing the value}

{if you want it to assign a value to particular variable then we use:}

SELECT @id :=50;

```
mysql> set @id= 1;
Query OK, 0 rows affected (0.00 sec)

mysql> select @id =1;
+--------+
| @id =1 |
+--------+
|      1 |
+--------+
1 row in set (0.00 sec)

mysql> select @id =2;
+--------+
| @id =2 |
+--------+
|      0 |
+--------+
1 row in set (0.00 sec)
```

Example4:

- set @empid=4;
- select * from employees where Emp_ID= @empid;

Here employee id 4 I am storing in a variable @empid then later on when I want to see result for employee id 4 I mentioned **Emp_ID = @empid**

Now if you want to update the value in that variable like @empid = 5

Then use

select @empid := 5; **(updated the value of variable as 5 now)**

select * from employees where Emp_ID= @empid;

```
mysql> Select @empid := 5;
+-------------+
| @empid := 5 |
+-------------+
|           5 |
+-------------+
1 row in set, 1 warning (0.00 sec)

mysql> select * from employees where Emp_ID= @empid;
+--------+-------+---------+------------+--------+
| Emp_ID | Name  | Dept_ID | manager_ID | Salary |
+--------+-------+---------+------------+--------+
|      5 | Priya |     101 |          4 |  60000 |
+--------+-------+---------+------------+--------+
1 row in set (0.00 sec)
```

But if I don't use colon (:)

Select @empid =5;

Here it will compare the variable value with what we have passed earlier if gets matched then returns 1 otherwise 0

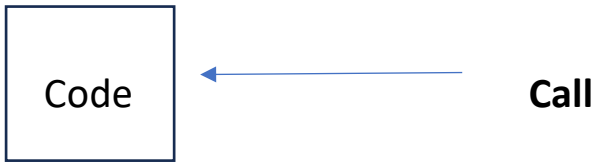
```
mysql> Select @empid =5;
+-----------+
| @empid =5 |
+-----------+
|         1 |
+-----------+
1 row in set (0.00 sec)

mysql> Select @empid =4;
+-----------+
| @empid =4 |
+-----------+
|         0 |
+-----------+
1 row in set (0.00 sec)
```

Stored Routines:

1. Stored procedures
2. Stored function

Stored procedures: Where we stores a section of code later on that we can call



- We use SQL commands to retrieve data
- Then to save time we use SQL scripts where we just run script instead the whole commands
- We every time if you have new data then edit the script and then run it
- But in procedures we just pass data as argument and get results as needed.
- We make procedures and use it again n again
- In procedure we have something called **DELIMITER** (it says statement is completed)
- DELIMITER // states that after writing the procedure we will end it by // only
- For DELIMITER you can use $$ sign also or any other sign

Syntax:

DELIMITER //
……
……
END //
DELIMITER ;
{here in last line DELIMETER ; states now again we can terminate the commands by the default ; (semi-colon)}

***delimiter // (now whenever I write any command in MYSQL that gets end with // instead of ;)**

How to Print something here

BEGIN

DBMS_OUTPUT.PUT_LINE(“hello world”);

END;

PL/SQL Programs compiled by oracle database server and stored inside a database

Example1 for Procedures:

- delimiter //
- select * from employees //

```
mysql> delimiter //
mysql> select * from employees //
+--------+---------+---------+------------+--------+
| Emp_ID | Name    | Dept_ID | manager_ID | Salary |
+--------+---------+---------+------------+--------+
|      1 | Prakash |     101 |          7 |  30000 |
|      2 | Pritosh |     102 |          6 |  40000 |
|      3 | Rimple  |     103 |          5 |  32000 |
|      4 | Ridhima |     104 |          1 |  50000 |
|      5 | Priya   |     101 |          4 |  60000 |
|      6 | Ridha   |     101 |          3 |  70000 |
|      7 | Seema   |     102 |          2 |  80000 |
|      8 | ritika  |    NULL |       NULL |   NULL |
|      9 | manya   |    NULL |       NULL |   NULL |
+--------+---------+---------+------------+--------+
9 rows in set (0.00 sec)
```

- create procedure GetAllEmployees()

  begin

  select * from employees;

  end //

delimiter ;

Executing a Procedure:

To execute the procedure, you can use the **CALL** statement

- call GetAllEmployees();

```
mysql> call GetAllEmployees();
+--------+---------+---------+------------+--------+
| Emp_ID | Name    | Dept_ID | manager_ID | Salary |
+--------+---------+---------+------------+--------+
|      1 | Prakash |     101 |          7 |  30000 |
|      2 | Pritosh |     102 |          6 |  40000 |
|      3 | Rimple  |     103 |          5 |  32000 |
|      4 | Ridhima |     104 |          1 |  50000 |
|      5 | Priya   |     101 |          4 |  60000 |
|      6 | Ridha   |     101 |          3 |  70000 |
|      7 | Seema   |     102 |          2 |  80000 |
|      8 | ritika  |    NULL |       NULL |   NULL |
|      9 | manya   |    NULL |       NULL |   NULL |
+--------+---------+---------+------------+--------+
9 rows in set (0.00 sec)
```

Example2 for Procedures with parameters:

create procedure GetEmpSalary(in **mentionsalary** int)

begin

select Name from employees where Salary = **mentionsalary**;

end //

delimiter ;

call GetEmpSalary(50000);

```
mysql> call GetEmpSalary(50000);
+---------+
| Name    |
+---------+
| Ridhima |
+---------+
1 row in set (0.00 sec)
```

Select * from employees;

```
mysql> select * from employees;
+--------+---------+---------+------------+--------+
| Emp_ID | Name    | Dept_ID | manager_ID | Salary |
+--------+---------+---------+------------+--------+
|      1 | Prakash |     101 |          7 |  30000 |
|      2 | Pritosh |     102 |          6 |  40000 |
|      3 | Rimple  |     103 |          5 |  32000 |
|      4 | Ridhima |     104 |          1 |  50000 |
|      5 | Priya   |     101 |          4 |  60000 |
|      6 | Ridha   |     101 |          3 |  70000 |
|      7 | Seema   |     102 |          2 |  80000 |
|      8 | ritika  |    NULL |       NULL |   NULL |
|      9 | manya   |    NULL |       NULL |   NULL |
+--------+---------+---------+------------+--------+
9 rows in set (0.00 sec)
```

Example 3 procedure with OUT parameter:

create procedure GetEmp(in empid int, out empname varchar(20))

begin

select Name into empname

from employees

where Emp_ID= empid;

end //

delimiter ;

set @employeename= NULL;

call GetEmp(2, @employeename);

select @employeename as Empname;

```
mysql> delimiter //
mysql> create procedure GetEmp(in empid int, out empname varchar(20))
    -> begin
    -> select Name into empname
    -> from employees
    -> where Emp_ID= empid;
    -> end //
Query OK, 0 rows affected (0.01 sec)

mysql> delimiter ;
mysql> set @employeename= NULL;
Query OK, 0 rows affected (0.00 sec)

mysql> call GetEmp(2, @employeename);
Query OK, 1 row affected (0.00 sec)

mysql> select @employeename as Empname;
+---------+
| Empname |
+---------+
| Pritosh |
+---------+
1 row in set (0.00 sec)
```

Example 4 : Using Case statement in procedures

```
delimiter //
create procedure salaryInfo()
   begin
    SELECT Name,
    CASE
            WHEN Salary > 60000 THEN 'High Salary'
            WHEN Salary > 40000 THEN 'Medium Salary'
     ELSE 'Low Salary'
        END AS SalaryCategory
     FROM employees;
    end //

delimiter ;
call SalaryInfo();
```

```
mysql> delimiter ;
mysql> call SalaryInfo();
+---------+----------------+
| Name    | SalaryCategory |
+---------+----------------+
| Prakash | Low Salary     |
| Pritosh | Low Salary     |
| Rimple  | Low Salary     |
| Ridhima | Medium Salary  |
| Priya   | Medium Salary  |
| Ridha   | High Salary    |
| Seema   | High Salary    |
| ritika  | Low Salary     |
| manya   | Low Salary     |
+---------+----------------+
9 rows in set (0.00 sec)
```

Example5: Another example of case expression

```
delimiter //
create procedure EmpInfo(in empid int)
   begin
   declare empname varchar(255);
   declare empdept varchar(255);
   case empid
    when 1 then
                set empname='prakash';
               set empdept= '101';
    when 2 then
               set empname='pritosh';
                set empdept= '102';
   when 3 then
               set empname= 'rimple';
              set empdept= '103';
   else
               set empname= 'Employee not found';
               set empdept= 'N/A';
   end case;
```

```
    select empname, empdept;
    end //
delimiter ;
call EmpInfo(2);
```

```
mysql> delimiter ;
mysql> call EmpInfo(2);
+---------+---------+
| empname | empdept |
+---------+---------+
| pritosh | 102     |
+---------+---------+
1 row in set (0.00 sec)

Query OK, 0 rows affected (0.01 sec)
```

Explanation:

```sql
DELIMITER //
```

- This line sets the delimiter to `//` temporarily. It allows you to define the stored procedure using a different delimiter than the standard semicolon (`;`). This is done to avoid conflicts between the delimiter used within the stored procedure and the delimiter used to terminate SQL statements.

```sql
CREATE PROCEDURE EmpInfo(IN empId INT)
```

- This line begins the definition of a MySQL stored procedure named `EmpInfo`. It takes one input parameter called `empId`, which is expected to be an integer.

```sql
BEGIN
```

```

- This line marks the beginning of the procedure's body. Everything within the `BEGIN` and `END` block constitutes the logic of the stored procedure.

```sql
-- Declare variables to store employee information
DECLARE empname VARCHAR(255);
DECLARE empdept VARCHAR(255);
```

- These lines declare two local variables within the stored procedure. `empName` and `empDept` are used to store the name and department of the employee, respectively.

```sql
-- Use a CASE statement to retrieve employee information based on Emp_ID
CASE empId
  WHEN 1 THEN
    SET empName = 'Prakash';
    SET empDept = '101';
  WHEN 2 THEN
    SET empName = 'Pritosh';
    SET empDept = '102';
  WHEN 3 THEN
    SET empName = 'Rimple';
    SET empDept = '103';
  ELSE
    SET empName = 'Employee not found';
    SET empDept = 'N/A';
END CASE;
```
```

- These lines use a `CASE` statement to conditionally set the values of `empName` and `empDept` based on the value of the `empId` parameter. Here's how it works:

  - If `empId` is 1, it sets `empName` to 'Prakash' and `empDept` to '101'.

  - If `empId` is 2, it sets `empName` to 'Pritosh' and `empDept` to '102'.

  - If `empId` is 3, it sets `empName` to 'Rimple' and `empDept` to '103'.

  - If `empId` doesn't match any of the above cases, it sets `empName` to 'Employee not found' and `empDept` to 'N/A'.

```sql
-- Display the employee information
SELECT empName, empDept;
```

- This line uses a `SELECT` statement to retrieve and display the values of `empName` and `empDept`, which were set in the previous `CASE` statement.

```sql
END //
```

- This line marks the end of the stored procedure's body.

```sql
DELIMITER ;
```

- This line resets the delimiter to the default semicolon (`;`) after defining the stored procedure. This delimiter change was temporary and was used only for defining the stored procedure.

This MySQL stored procedure, `EmpInfo`, takes an employee ID as input, uses a `CASE` statement to determine the employee's name and department based on the provided ID, and then displays this information using a `SELECT` statement. If the provided `empId` doesn't match any known cases, it returns a message indicating that the employee was not found.

Exmple6:

```
DELIMITER //

CREATE PROCEDURE CalculateAvgSalaryofEmployees(
    IN deptId INT,
    OUT avg_salary DECIMAL(10, 2)
)
BEGIN
    SELECT AVG(Salary) INTO avg_salary FROM employees WHERE Dept_ID = deptId;
END;
//

DELIMITER ;
SET @avg_salary = 0;      -- Initialize the output parameter
CALL CalculateAvgSalary(101, @avg_salary);
SELECT @avg_salary;
```

```
mysql> CALL CalculateAvgSalary(101, @avg_salary);
Query OK, 1 row affected (0.00 sec)

mysql> SELECT @avg_salary;
+-------------+
| @avg_salary |
+-------------+
|    55000.00 |
+-------------+
1 row in set (0.00 sec)
```

Advantages of using Procedures in DBMS:

Using procedures in a Database Management System (DBMS) offers several advantages:

1. **Modularity**: Procedures allow you to break down complex database operations into smaller, more manageable units of work. This makes it easier to design, develop, and maintain your database system.

2. **Code Reusability**: Procedures can be reused across different parts of your application or by multiple users. This reduces the need to write the same code logic multiple times, promoting consistency and reducing the risk of errors.

3. **Enhanced Security**: Procedures can be encapsulated with security privileges. This means you can control who has access to execute certain procedures, providing an additional layer of security for your data.

4. **Improved Performance**: Procedures can be precompiled and stored in the database, which can improve execution speed. Additionally, using procedures reduces the amount of data transferred between the database and application, which can lead to better performance.

5. **Transaction Control**: Procedures can be used to encapsulate a series of SQL statements within a transaction. This ensures that a group of operations either all succeed or all fail, maintaining data integrity.

6. **Parameterization**: Procedures can accept parameters, making them versatile. You can pass values to a procedure, making it adaptable to different situations without the need to create multiple versions of the same logic.

7. **Abstraction**: Procedures can provide an abstracted interface to the database, hiding the underlying complexity of the database schema. This can simplify interactions for application developers.

8. **Version Control**: Procedures can be version-controlled like any other code, making it easier to track changes and roll back to previous versions if necessary.

9. **Maintenance and Debugging**: Debugging and maintaining a database is easier when procedures are used. Changes to logic can be made in one place, reducing the risk of introducing bugs or inconsistencies.

10. **Centralized Logic**: Procedures enable you to centralize critical business logic within the database. This ensures that essential data operations are consistent across different parts of your application.

11. **Performance Optimization**: Procedures can be optimized for performance. Database administrators can analyze and tune procedures to execute efficiently, which is particularly important for large-scale databases.

12. **Reduced Network Traffic**: By executing procedures within the database, you minimize the need to send large amounts of data between the application and the database server, reducing network traffic.