**Exercise 01:**

Create a stored procedure named **GetOrderDetails** that takes an order ID as a parameter and returns the details of the products in that order, including product name, quantity, and unit price.

**Answer:**

DELIMITER //

CREATE PROCEDURE GetOrderDetails(IN orderId INT)

BEGIN

SELECT

od.ProductID,

p.ProductName,

od.Quantity,

od.UnitPrice

FROM

orderdetails od

JOIN

products p ON od.ProductID = p.ProductID

WHERE

od.OrderID = orderId;

END //

DELIMITER ;

Call the stored procedure:

Call the **GetOrderDetails** stored procedure with a specific order ID (replace **<your\_order\_id>** with an actual order ID from the "orders" table).

CALL GetOrderDetails(10248);

**Exercise 02:**

Create a stored procedure named **GetCategoryProducts** that takes a category ID as a parameter and returns the list of products within that category, including product name, unit price, and quantity available in stock.

**Answer:**

DELIMITER //

CREATE PROCEDURE GetCategoryProducts(IN categoryId INT)

BEGIN

SELECT

p.ProductName,

p.UnitPrice,

p.UnitsInStock

FROM

products p

JOIN

categories c ON p.CategoryID = c.CategoryID

WHERE

p.CategoryID = categoryId;

END //

DELIMITER ;

Call the stored procedure:

Call the **GetCategoryProducts** stored procedure with a specific category ID (replace **<your\_category\_id>** with an actual category ID from the "categories" table).

CALL GetCategoryProducts(2);

**Exercise 03:**

Create a stored procedure named **GetEmployeeOrders** that takes an employee ID as a parameter and returns the list of orders placed by that employee, including order ID, order date, and total order amount.

**Answer:**

DELIMITER //

CREATE PROCEDURE GetEmployeeOrders(IN employeeId INT)

BEGIN

SELECT

o.OrderID,

o.OrderDate,

SUM(od.UnitPrice \* od.Quantity) AS TotalAmount

FROM

orders o

JOIN

orderdetails od ON o.OrderID = od.OrderID

WHERE

o.EmployeeID = employeeId

GROUP BY

o.OrderID, o.OrderDate;

END //

DELIMITER ;

Call the stored procedure:

Call the **GetEmployeeOrders** stored procedure with a specific employee ID (replace **<your\_employee\_id>** with an actual employee ID from the "employees" table).

CALL GetEmployeeOrders(3);

**Exercise 04:**

Create a stored procedure named **GetCustomerOrders** that takes a customer ID as a parameter and returns the list of orders placed by that customer, including order ID, order date, and total order amount.

**Answer:**

DELIMITER //

CREATE PROCEDURE GetCustomerOrders(IN customerId CHAR(5))

BEGIN

SELECT

o.OrderID,

o.OrderDate,

SUM(od.UnitPrice \* od.Quantity) AS TotalAmount

FROM

orders o

JOIN

orderdetails od ON o.OrderID = od.OrderID

WHERE

o.CustomerID = customerId

GROUP BY

o.OrderID, o.OrderDate;

END //

DELIMITER ;

Call the stored procedure:

CALL GetCustomerOrders('ALFKI');

**Exercise 05:**

Create a stored procedure named **GetTopSellingProducts** that takes a start date and an end date as parameters and returns the top N selling products within that date range, where N is another parameter.

**Answer:**

DELIMITER //

CREATE PROCEDURE **GetTopSellingProducts**(

IN startDate DATE,

IN endDate DATE,

IN topN INT

)

BEGIN

SELECT

p.ProductName,

SUM(od.Quantity) AS TotalSold

FROM

orders o

JOIN

orderdetails od ON o.OrderID = od.OrderID

JOIN

products p ON od.ProductID = p.ProductID

WHERE

o.OrderDate BETWEEN startDate AND endDate

GROUP BY

p.ProductName

ORDER BY

TotalSold DESC

LIMIT

topN;

END //

DELIMITER ;

Call the stored procedure:

CALL GetTopSellingProducts('1997-01-01', '1997-12-31', 3);

**Exercise 06:**

Create a stored procedure named **GetEmployeeSalesSummary** that takes an employee ID and a date range (start date and end date) as parameters. The procedure should return a summary of the sales made by that employee during the specified date range. Include information such as the total number of orders, total revenue, and average order value.

**Answer:**

DELIMITER //

CREATE PROCEDURE **GetEmployeeSalesSummary**(

IN employeeId INT,

IN startDate DATE,

IN endDate DATE

)

BEGIN

SELECT

COUNT(DISTINCT o.OrderID) AS TotalOrders,

SUM(od.Quantity \* od.UnitPrice) AS TotalRevenue,

AVG(od.Quantity \* od.UnitPrice) AS AvgOrderValue

FROM

orders o

JOIN

orderdetails od ON o.OrderID = od.OrderID

WHERE

o.EmployeeID = employeeId

AND o.OrderDate BETWEEN startDate AND endDate;

END //

DELIMITER ;

Call the stored procedure:

CALL GetEmployeeSalesSummary(4, '1997-01-01', '1997-12-31');

**Exercise 07:**

Create a stored procedure named **GetCategorySalesAnalysis** that takes a category ID and a year as parameters. The procedure should return a detailed analysis of sales for that category during the specified year. Include information such as the total revenue, total quantity sold, average unit price, and the month with the highest sales.

**Answer:**

DELIMITER //

CREATE PROCEDURE **GetCategorySalesAnalysis**(

IN categoryId INT,

IN analysisYear INT

)

BEGIN

SELECT

SUM(od.Quantity \* od.UnitPrice) AS TotalRevenue,

SUM(od.Quantity) AS TotalQuantitySold,

AVG(od.UnitPrice) AS AvgUnitPrice,

MAX(MonthlySales.MonthlyTotal) AS MonthWithHighestSales

FROM

orderdetails od

JOIN

orders o ON od.OrderID = o.OrderID

JOIN

products p ON od.ProductID = p.ProductID

JOIN (

SELECT

o2.OrderDate,

SUM(od2.Quantity \* od2.UnitPrice) AS MonthlyTotal

FROM

orders o2

JOIN

orderdetails od2 ON o2.OrderID = od2.OrderID

WHERE

YEAR(o2.OrderDate) = analysisYear

GROUP BY

MONTH(o2.OrderDate)

) AS MonthlySales ON MONTH(o.OrderDate) = MONTH(MonthlySales.OrderDate)

WHERE

p.CategoryID = categoryId

AND YEAR(o.OrderDate) = analysisYear;

END //

DELIMITER ;

Call the stored procedure:

CALL GetCategorySalesAnalysis(2, 1997);

**Exercise 08:**

Create a stored procedure named **GetCustomerOrderFrequency** that takes a customer ID as a parameter. The procedure should return the frequency of orders placed by that customer, categorized by the day of the week. Include information such as the day of the week, the total number of orders, and the average order value for each day.

**Answer:**

DELIMITER //

CREATE PROCEDURE GetCustomerOrderFrequency(

IN customerId CHAR(5)

)

BEGIN

SELECT

DAYNAME(o.OrderDate) AS DayOfWeek,

COUNT(o.OrderID) AS TotalOrders,

AVG(od.Quantity \* od.UnitPrice) AS AvgOrderValue

FROM

orders o

JOIN

orderdetails od ON o.OrderID = od.OrderID

WHERE

o.CustomerID = customerId

GROUP BY

DAYNAME(o.OrderDate);

END //

DELIMITER ;

Call the stored procedure:

CALL GetCustomerOrderFrequency('ALFKI');

**Exercise 09:**

Create a stored procedure named **GetEmployeeSalesSummaryWithOutParam** that takes an employee ID and a date range (start date and end date) as parameters. The procedure should calculate and return the total revenue and average order value for that employee during the specified date range. Use an **OUT** parameter to return the average order value.

**Answer:**

DELIMITER //

CREATE PROCEDURE **GetEmployeeSalesSummaryWithOutParam**(

IN employeeId INT,

IN startDate DATE,

IN endDate DATE,

OUT avgOrderValue DECIMAL(10, 2) -- OUT parameter for average order value

)

BEGIN

SELECT

SUM(od.Quantity \* od.UnitPrice) AS TotalRevenue,

AVG(od.Quantity \* od.UnitPrice) INTO avgOrderValue

FROM

orders o

JOIN

orderdetails od ON o.OrderID = od.OrderID

WHERE

o.EmployeeID = employeeId

AND o.OrderDate BETWEEN startDate AND endDate;

END //

DELIMITER ;

Call the stored procedure:

CALL GetEmployeeSalesSummaryWithOutParam(4, '1997-01-01', '1997-12-31', @avgOrderValue);

-- Retrieve the value of the OUT parameter

SELECT @avgOrderValue AS AvgOrderValue;

**Exercise 10:**

Create a stored procedure named **GetCustomerOrderStatistics** that takes a customer ID as a parameter and returns statistics about the customer's orders. The procedure should calculate and return the total number of orders, the total revenue, and the average order value. Use **OUT** parameters for each of these statistics.

**Answer:**

DELIMITER //

CREATE PROCEDURE GetCustomerOrderStatistics(

IN customerId CHAR(5),

OUT totalOrders INT,

OUT totalRevenue DECIMAL(10, 2),

OUT avgOrderValue DECIMAL(10, 2)

)

BEGIN

SELECT

COUNT(DISTINCT o.OrderID) INTO totalOrders,

SUM(od.Quantity \* od.UnitPrice) INTO totalRevenue,

AVG(od.Quantity \* od.UnitPrice) INTO avgOrderValue

FROM

orders o

JOIN

orderdetails od ON o.OrderID = od.OrderID

WHERE

o.CustomerID = customerId;

END //

DELIMITER ;

Call the stored procedure:

CALL GetCustomerOrderStatistics('ALFKI', @totalOrders, @totalRevenue, @avgOrderValue);

-- Retrieve the values of the OUT parameters

SELECT @totalOrders AS TotalOrders, @totalRevenue AS TotalRevenue, @avgOrderValue AS AvgOrderValue;

**Exercise 11:**

Create a stored procedure named **UpdateEmployeeSalary** that takes an employee ID as an IN parameter and a percentage increase as an INOUT parameter. The procedure should update the salary of the specified employee by applying the percentage increase.

**Answer:**

DELIMITER //

CREATE PROCEDURE **UpdateEmployeeSalary**(

IN employeeId INT,

INOUT percentageIncrease DECIMAL(5, 2)

)

BEGIN

DECLARE currentSalary DECIMAL(10, 2);

-- Retrieve the current salary of the employee

SELECT salary INTO currentSalary FROM employees WHERE id = employeeId;

-- Update the salary with the percentage increase

UPDATE employees SET salary = currentSalary \* (1 + percentageIncrease / 100) WHERE id = employeeId;

-- Update the percentage increase to reflect the actual increase applied

SET percentageIncrease = (currentSalary \* percentageIncrease) / 100;

END //

DELIMITER ;

Call the stored procedure:

CALL UpdateEmployeeSalary(1, 10.0, @percentageIncrease);

-- Retrieve the updated percentage increase

SELECT @percentageIncrease AS UpdatedPercentageIncrease;

**Exercise 12**

Create a stored procedure named **ProcessOrder** that takes an order ID as an IN parameter and an INOUT parameter representing the order status. The procedure should update the status of the specified order and provide feedback about the processing.

**Answer:**

DELIMITER //

CREATE PROCEDURE ProcessOrder(

IN orderId INT,

INOUT orderStatus VARCHAR(20)

)

BEGIN

-- Check the current status of the order

DECLARE currentStatus VARCHAR(20);

SELECT status INTO currentStatus FROM orders WHERE OrderID = orderId;

-- Update the order status

UPDATE orders SET status = 'Processed' WHERE OrderID = orderId;

-- Update the INOUT parameter to reflect the new status

SET orderStatus = 'Processed';

-- Provide feedback about the processing

SELECT

CASE

WHEN currentStatus IS NULL THEN 'Order not found.'

WHEN currentStatus = 'Processed' THEN 'Order is already processed.'

ELSE 'Order processed successfully.'

END AS Feedback;

END //

DELIMITER ;

Call the stored procedure:

CALL ProcessOrder(10248, 'Pending', @orderStatus);

-- Retrieve the feedback and the updated order status

SELECT @orderStatus AS UpdatedOrderStatus, Feedback;

**Exercise 13:**

Create a stored procedure named **UpdateProductStock** that takes a product ID as an IN parameter, an INOUT parameter representing the quantity to add to the stock, and an OUT parameter representing the updated stock level. The procedure should update the stock level of the specified product and provide feedback about the update.

**Answer:**

DELIMITER //

CREATE PROCEDURE UpdateProductStock(

IN productId INT,

INOUT quantityToAdd INT,

OUT updatedStock INT

)

BEGIN

-- Check the current stock level of the product

DECLARE currentStock INT;

SELECT UnitsInStock INTO currentStock FROM products WHERE ProductID = productId;

-- Update the stock level

UPDATE products SET UnitsInStock = currentStock + quantityToAdd WHERE ProductID = productId;

-- Update the OUT parameter to reflect the new stock level

SET updatedStock = currentStock + quantityToAdd;

-- Provide feedback about the update

SELECT

CASE

WHEN currentStock IS NULL THEN 'Product not found.'

ELSE 'Stock updated successfully.'

END AS Feedback;

END //

DELIMITER ;

Call the stored procedure:

CALL UpdateProductStock(5, 20, @updatedStock);

-- Retrieve the feedback and the updated stock level

SELECT @updatedStock AS UpdatedStock, Feedback;

**Exercise 14:**

Create a stored procedure named **CalculateDiscountedPrice** that takes a product ID as an IN parameter, an INOUT parameter representing the original price, and an OUT parameter representing the discounted price. The procedure should calculate and provide feedback about the discounted price.

**Answer:**

DELIMITER //

CREATE PROCEDURE CalculateDiscountedPrice(

IN productId INT,

INOUT originalPrice DECIMAL(10, 2),

OUT discountedPrice DECIMAL(10, 2)

)

BEGIN

-- Check if the product exists

DECLARE productExists INT;

SELECT COUNT(\*) INTO productExists FROM products WHERE ProductID = productId;

-- Calculate the discounted price (assuming a 10% discount)

SET discountedPrice = originalPrice \* 0.9;

-- Provide feedback about the discounted price

SELECT

CASE

WHEN productExists = 0 THEN 'Product not found.'

ELSE 'Discounted price calculated successfully.'

END AS Feedback;

END //

DELIMITER ;

Call the stored procedure:

CALL CalculateDiscountedPrice(7, 50.00, @discountedPrice);

-- Retrieve the feedback and the discounted price

SELECT @discountedPrice AS DiscountedPrice, Feedback;

**Exercise 15:**

Create a stored procedure named **ProcessOrderDetails** that takes an order ID as an IN parameter, an INOUT parameter representing the order status, and an OUT parameter representing the total order value. The procedure should perform the following tasks:

* Update the order status to 'Processing'.
* Calculate the total order value by summing the product of quantity and unit price for each item in the order.
* Update the INOUT parameter to reflect the new order status.
* Provide feedback about the order processing.

**Answer:**

DELIMITER //

CREATE PROCEDURE ProcessOrderDetails(

IN orderId INT,

INOUT orderStatus VARCHAR(20),

OUT totalOrderValue DECIMAL(10, 2)

)

BEGIN

-- Check the current status of the order

DECLARE currentStatus VARCHAR(20);

SELECT status INTO currentStatus FROM orders WHERE OrderID = orderId;

-- Update the order status to 'Processing'

UPDATE orders SET status = 'Processing' WHERE OrderID = orderId;

-- Calculate the total order value

SELECT

SUM(od.Quantity \* od.UnitPrice) INTO totalOrderValue

FROM

orderdetails od

WHERE

od.OrderID = orderId;

-- Update the INOUT parameter to reflect the new order status

SET orderStatus = 'Processing';

-- Provide feedback about the order processing

SELECT

CASE

WHEN currentStatus IS NULL THEN 'Order not found.'

WHEN currentStatus = 'Processing' THEN 'Order is already in processing state.'

ELSE 'Order processing initiated successfully.'

END AS Feedback;

END //

DELIMITER ;

Call the stored procedure:

CALL ProcessOrderDetails(10250, 'Pending', @totalOrderValue);

-- Retrieve the feedback, updated order status, and the total order value

SELECT @totalOrderValue AS TotalOrderValue, Feedback;