**Q1]**

**a) Create a table called test\_table with 2 columns RecordNumber (type : Number(3)) and CurrentDate (type : Date)). Write PL/SQL block which will insert 50 records into test\_table. Insert the current date value into the table.**

**Ans.**

create table test\_table(

RecordNumber INT,

CurrentDate DATE

);

-- Change the delimiter

delimiter //

-- Create the stored procedure

create procedure InsertTestData()

begin

declare counter int default 1;

while counter <= 50 do

insert into test\_table (RecordNumber, CurrentDate) values (counter, curdate());

set counter = counter + 1;

end while;

end //

-- Change the delimiter back to semicolon

delimiter ;

call InsertTestData();

select \* from test\_table;

**b) Create a products table products(ProductID number(4),category char(3),detail varchar2(30),price number(10,2),stock number(5)). Insert the sample data. Write PSM procedure with two arguments X & Y which will increase price by X% for all products in category Y. X and Y will be given by user.**

**Ans.**

-- Create the products table

create table products(

productID int(4),

category char(3),

detail varchar(30),

price decimal(10, 2),

stock int(5)

);

ALTER TABLE products

MODIFY COLUMN category VARCHAR(255);

-- Insert sample data

insert into products(productID, category, detail, price, stock)

values (1, 'Cat1', 'Product1', 100.00, 50);

insert into products(productID, category, detail, price, stock)

values (2, 'Cat2', 'Product2', 200.05, 55);

insert into products(productID, category, detail, price, stock)

values (3, 'Cat3', 'Product3', 145.36, 840);

-- Create the procedure to update prices

DELIMITER //

CREATE PROCEDURE UpdatePrices (X DECIMAL(5,2), Y VARCHAR(255))

BEGIN

UPDATE products

SET price = price \* (1 + X / 100)

WHERE category = Y;

COMMIT;

END //

DELIMITER ;

-- Disable safe update mode

-- Unless we do like this we cant make any updations in the table

SET SQL\_SAFE\_UPDATES = 0;

-- Call the procedure

CALL UpdatePrices(10.00, 'Cat1');

-- Re-enable safe update mode (optional, but recommended for safety)

SET SQL\_SAFE\_UPDATES = 1;

select \* from products;

**a) Create Object Table containing field “name” of size 50 characters and member function “countNoOfWords” which returns the no. of words in “name” field. Demonstrate the working by entering different data.**

**Ans.**

-- Create the object type

CREATE OR REPLACE TYPE MyObject AS OBJECT (

name VARCHAR2(50),

MEMBER FUNCTION countNoOfWords RETURN NUMBER

);

-- Create the member function

CREATE OR REPLACE TYPE BODY MyObject AS

MEMBER FUNCTION countNoOfWords RETURN NUMBER IS

word\_count NUMBER := 0;

BEGIN

word\_count := REGEXP\_COUNT(name, '\b\w+\b');

RETURN word\_count;

END;

END;

-- Create the object table

CREATE TABLE ObjectTable OF MyObject;

-- Insert sample data

INSERT INTO ObjectTable VALUES (MyObject('Hello World'));

INSERT INTO ObjectTable VALUES (MyObject('This is a sentence'));

INSERT INTO ObjectTable VALUES (MyObject('One two three'));

-- Demonstrate the working by selecting the name and the count of words

SELECT name, name.countNoOfWords() AS word\_count FROM ObjectTable;

**b) Create an address type with the following attributes : address, city, state & pincode. Include the following methods i. to extract the addresses based on given keyword. j. to return the no. of words in each given field (method should accept the name of attribute/field)**

**Ans.**

-- Create the address type

CREATE OR REPLACE TYPE AddressType AS OBJECT (

address VARCHAR2(100),

city VARCHAR2(50),

state VARCHAR2(50),

pincode VARCHAR2(10),

-- Method to extract addresses based on a given keyword

MEMBER FUNCTION extractAddresses(keyword IN VARCHAR2) RETURN VARCHAR2,

-- Method to return the number of words in a given field

MEMBER FUNCTION countWords(fieldName IN VARCHAR2) RETURN NUMBER

);

/

-- Create the method body for extractAddresses

CREATE OR REPLACE TYPE BODY AddressType AS

MEMBER FUNCTION extractAddresses(keyword IN VARCHAR(50)) RETURNS VARCHAR(4000) DETERMINISTIC

BEGIN

DECLARE result VARCHAR(4000) DEFAULT '';

-- Declare variables to hold address attributes

DECLARE addressVar, cityVar, stateVar, pincodeVar VARCHAR(255);

-- Cursor to iterate over rows in the address\_type table

DECLARE cur CURSOR FOR

SELECT address, city, state, pincode FROM address\_type;

-- Open cursor

OPEN cur;

-- Fetch first row

FETCH cur INTO addressVar, cityVar, stateVar, pincodeVar;

-- Loop through all rows

WHILE NOT (addressVar <=> NULL) DO

-- Check if any field matches the keyword

IF addressVar LIKE CONCAT('%', keyword, '%') THEN

SET result := CONCAT(result, 'Address: ', addressVar, '\n');

END IF;

IF cityVar LIKE CONCAT('%', keyword, '%') THEN

SET result := CONCAT(result, 'City: ', cityVar, '\n');

END IF;

IF stateVar LIKE CONCAT('%', keyword, '%') THEN

SET result := CONCAT(result, 'State: ', stateVar, '\n');

END IF;

IF pincodeVar LIKE CONCAT('%', keyword, '%') THEN

SET result := CONCAT(result, 'Pincode: ', pincodeVar, '\n');

END IF;

-- Fetch next row

FETCH cur INTO addressVar, cityVar, stateVar, pincodeVar;

END WHILE;

-- Close cursor

CLOSE cur;

-- Return result

return result;

END //

-- Method to count words in a given field

MEMBER FUNCTION countWords(fieldName IN VARCHAR2) RETURN NUMBER IS

word\_count NUMBER := 0;

field\_value VARCHAR2(100);

BEGIN

-- Get the value of the specified field

CASE fieldName

WHEN 'address' THEN field\_value := self.address;

WHEN 'city' THEN field\_value := self.city;

WHEN 'state' THEN field\_value := self.state;

WHEN 'pincode' THEN field\_value := self.pincode;

ELSE NULL;

END CASE;

-- Count the number of words in the field value

IF field\_value IS NOT NULL THEN

word\_count := REGEXP\_COUNT(field\_value, '\b\w+\b');

END IF;

RETURN word\_count;

END countWords;

END;

/

1. **DELIMITER //**: This sets the delimiter to **//** instead of the default **;**. This is used because the function definition contains multiple **;** inside it, and changing the delimiter prevents conflicts.
2. **CREATE FUNCTION extractAddresses(addressInput VARCHAR(100), cityInput VARCHAR(50), stateInput VARCHAR(50), pincodeInput VARCHAR(10), keyword VARCHAR(50)) RETURNS VARCHAR(4000) DETERMINISTIC**: This line starts the definition of the **extractAddresses** function. It declares a function named **extractAddresses** that takes five input parameters (**addressInput**, **cityInput**, **stateInput**, **pincodeInput**, and **keyword**) of specific data types and returns a **VARCHAR** value of length 4000. The **DETERMINISTIC** keyword indicates that the function returns the same result for the same inputs every time it is called.
3. **BEGIN**: Marks the beginning of the function's executable code block.
4. **DECLARE result VARCHAR(4000) DEFAULT '';**: Declares a local variable named **result** of type **VARCHAR(4000)** and initializes it with an empty string.
5. **IF addressInput LIKE CONCAT('%', keyword, '%') THEN**: Checks if the **addressInput** parameter contains the **keyword**. The **LIKE** operator is used for pattern matching, and **CONCAT** is used to concatenate **%** (wildcards) with the **keyword** to match any occurrence of **keyword** within **addressInput**.
6. **SET result := CONCAT(result, 'Address: ', addressInput, '\n');**: If the **addressInput** contains the **keyword**, this line appends **'Address: '**, the **addressInput**, and a newline character (**\n**) to the **result** variable using the **CONCAT** function and assigns the result back to **result**.
7. **END IF;**: Marks the end of the **IF** condition.
8. Similar **IF** conditions follow for **cityInput**, **stateInput**, and **pincodeInput**, appending the corresponding information to the **result** variable if the respective input contains the **keyword**.
9. **RETURN result;**: Returns the final value of the **result** variable as the output of the function.
10. **END //**: Marks the end of the function's code block using the custom delimiter **//**.
11. **DELIMITER ;**: Resets the delimiter back to **;**.

**c) Create a user defined data type course\_Type with 2 attributes course\_id, description : i. Create an object table based on the type created. j. Insert rows into the table Demonstrate the working with different data sets.**

**Ans.**

-- Create the user-defined data type course\_Type

CREATE OR REPLACE TYPE course\_Type AS OBJECT (

course\_id NUMBER,

description VARCHAR2(100)

);

/

-- Create the object table based on the course\_Type

CREATE TABLE course\_table OF course\_Type;

-- Insert rows into the table

INSERT INTO course\_table VALUES (1, 'Introduction to Programming');

INSERT INTO course\_table VALUES (2, 'Data Structures and Algorithms');

INSERT INTO course\_table VALUES (3, 'Database Management Systems');

-- Demonstrate the working with different data sets

-- Select all rows from the table

SELECT \* FROM course\_table;

-- Insert more rows into the table

INSERT INTO course\_table VALUES (4, 'Web Development');

INSERT INTO course\_table VALUES (5, 'Machine Learning');

-- Select all rows from the table again

SELECT \* FROM course\_table;

Explanation:

1. **Create Type**: We define a user-defined data type **course\_Type** using the **CREATE TYPE** statement. This type has two attributes: **course\_id** of type **NUMBER** and **description** of type **VARCHAR2(100)**.
2. **Create Table**: We create an object table **course\_table** using the **CREATE TABLE ... OF course\_Type** syntax. This creates a table where each row corresponds to an instance of the **course\_Type** object.
3. **Insert Data**: We insert rows into the **course\_table** using the **INSERT INTO** statement, providing values for the **course\_id** and **description** attributes.
4. **Demonstrate Working**: We demonstrate the working of the table by selecting all rows from it before and after inserting more data. This shows that we can insert and retrieve data from the object table just like any other table in the database.

You can customize the data sets by changing the values passed to the **INSERT INTO** statements to suit your needs.