Database Systems Project 2 Report

"We have done this assignment completely on our own. We have not copied it, nor have we given our solution to anyone else. We understand that if we are involved in plagiarism or cheating we will have to sign an official form that we have cheated and that this form will be stored in our official university records. We also understand that we will receive a grade of 0 for the involved assignment and our grades will be reduced by one level (e.g., from A to A- or from B+ to B) for our first offense, and that we will receive a grade of "F" for the course for any additional offense of any kind."

1) **Sequences**:

Sequence is an Oracle object which generates unique key values in a sequence of integer values.

The sequence for pur# needs to generate numbers that start with 10001

```
create sequence seqpur#
increment by 1
start with 10001;
```

Deleting all tuples from table Purchases and re-populate this table using the same tuples except that pur# will be generated by values using the sequence for pur# like updating insert script as below.

```
insert into purchases values (seqpur#.nextval, 'e01', 'p002', 'c001',
to_date('12-AUG-2022', 'DD-MON-YYYY'), 1, 211.65, 211.65, 37.35);
insert into purchases values (seqpur#.nextval, 'e01', 'p003', 'c001',
to_date('20-DEC-2022', 'DD-MON-YYYY'), 1, 118.40, 118.40, 29.60);
insert into purchases values (seqpur#.nextval, 'e02', 'p004', 'c002',
to_date('08-OCT-2022', 'DD-MON-YYYY'), 5, 0.99, 4.45, 0.5);
insert into purchases values (seqpur#.nextval, 'e01', 'p005', 'c003',
to_date('23-NOV-2022', 'DD-MON-YYYY'), 2, 39.98, 79.96, 20);
insert into purchases values (seqpur#.nextval, 'e04', 'p007', 'c004',
to_date('15-JAN-2023', 'DD-MON-YYYY'), 1, 179.10, 179.10, 19.90);
```

The sequence for log# needs to generate numbers that start with 1001:

```
create sequence seqlog#
increment by 1
start with 1001;
```

2) **Show Tables:**

To get the results to our JDBC interface program, we created a function for each table in the RBMSPackage and used ref cursor to get desired output.

For example :
Function for Tables Employees:

Specification:

```
function retrieveEmployees
   return my_cursor;
```

This is the function specification for retrieving tuples from employees table. We are using ref cursor to retrieve each tuple one by one.
First, my_cursor points to the row and then prints the data from that row.

Implementation:

```
function retrieveEmployees
     return my_cursor is
     my_ref my_cursor;
     begin
          open my_ref for select * from employees;
          return my_ref;
     end;
```

It returns a "my_cursor" cursor object.
A local variable "my_ref" of type "my_cursor" is defined within the method to store the reference to the cursor object.
The function then executes a SQL query on the cursor "my_ref" that retrieves all rows from the "employees" table.
Finally, as the function result, the cursor reference "my_ref" is returned.
To summarize, this function retrieves all the records from the "employees" table and returns them as a cursor that can be used to fetch the data row by row.

Similarly, we have implemented functions for the remaining tables by changing the function name and sql query.

In Java,

- "begin ? := RBMSPackage.retrieveEmployees(); end;" is used to execute the PL/SQL function "retrieveEmployees" from a package named "RBMSPackage" using JDBC. Same way all functions to retrieve tables are called.
- cs = conn.prepareCall(query) : this creates a new CallableStatement object cs for the query.
- cs.registerOutParameter(1, OracleTypes.CURSOR); - This registers an OUT parameter at index one of the CallableStatement object cs. The data type of the

parameter is OracleTypes.CURSOR, indicating that the function returns a cursor object. The index 1 refers to the first parameter of the query, which is expected to be the returned cursor object.

- Then through ResultSet, iterate through each row and print.

3) Monthly sale activities for any given employee:

```
function monthlySaleActvities(
        employeeId in employees.eid%type)
        return my_cursor is
        my_ref my_cursor;
    begin
        dbms_output.enable();
         open my_ref for    SELECT Employees.eid, name, to_char(pur_time, 'MON'),
to_char(pur_time, 'YYYY'),
                        COUNT(*), SUM(quantity), SUM(payment)
                        FROM Employees
                        JOIN Purchases ON Employees.eid = Purchases.eid
                        GROUP BY Employees.eid,name, to_char(pur_time, 'MON'),
to_char(pur_time, 'YYYY')
                        HAVING Employees.eid = employeeId;
        return my_ref;
    end;
```

This function named "monthlySaleActivities" takes an input parameter "employeeId" of type "employees.eid%type" i.e type of eid of employees table and returns a cursor of type "my_cursor".

The function starts by enabling the DBMS output by dbms_output.enable(). It then opens a cursor "my_ref" by selecting the "eid", "name", "to_char(pur_time, 'MON')" (formats the pur_time month to take only 3 characters, "to_char(pur_time, 'YYYY')"(formats the pur time year to take only 4 characters) , "count(*)", "sum(quantity)", and "sum(payment)" fields from the "Employees" and "Purchases" tables, where the "eid" field of the "Employees" table is equal to the input parameter "employeeId". The result is grouped by "eid", "name", "to_char(pur_time, 'MON')", and "to_char(pur_time, 'YYYY')".

The function then returns the "my_ref" cursor, which can be used to fetch the results of the select statement.

In Java,

- "begin ? := RBMSPackage.monthlySaleActvities(?); end;" is used to execute the PL/SQL function "monthlySaleActivities" from a package named "RBMSPackage" using JDBC.

- Take the employeeId from the user and set the cursor and taken employeeId to the callableStatement and print by iterating over ResultSet

4) **Procedure for adding tuples to the Employees table:**

```
PROCEDURE add_employee(e_id IN employees.eid%type, e_name IN
employees.name%type,
            e_telephone IN employees.telephone#%type, e_email IN
employees.email%type) AS

    BEGIN
     INSERT INTO Employees (eid, name, telephone#, email) VALUES (e_id, e_name,
e_telephone, e_email);
     COMMIT;
    END add_employee;
```

This procedure named "add_employee" takes four input parameters of respective data types "employees.eid%type","employees.name%type","employees.telephone#%type",and "employees.email%type" from employees tables.

The procedure starts by using an INSERT statement to add a new row to the "Employees" table. The values to be inserted are taken from the input parameters passed to the procedure. The insert statement includes the column names and their corresponding values, so the values are inserted into the specified columns.

Then procedure includes a COMMIT statement, which saves the changes made by the insert statement to the database.

In summary, this procedure is used to add a new employee to the "Employees" table by inserting a new row with the input parameters provided by the user.

Then need to create a trigger that can add a tuple to the logs table automatically whenever a new employee is added to the Employees table

```
create or replace trigger employees_trigger
  after insert on employees
  for each row
  begin
     insert into Logs (log#, user_name, operation, op_time, table_name, tuple_pkey)
     values (seqlog#.nextval, USER, 'insert', SYSDATE, 'employees', :new.eid);
  end;
/
```

This trigger named "employees_trigger" is fired "after insert" on the "employees" table "for each row" inserted.

The trigger starts with the keyword "begin" and includes a sql insert statement that inserts a new row into the Logs table. The values to be inserted are the next value generated by the "seqlog#" sequence, which is a sequence object that generates a unique number for each row inserted, the current user who initiated the operation, the string 'insert' to indicate that an insert operation occurred, the current date and time using SYSDATE, the name of the "employees" table, and the primary key value of the newly inserted row.

The primary key value of the newly inserted row is referenced using the ":new.eid" syntax. The ":new" prefix refers to the newly inserted row, and ".eid" refers to the value of the "eid" column of the new row and the trigger ends .

In summary, this trigger creates a log entry in the "Logs" table whenever a new row is inserted into the "employees" table. The log entry includes information about the user who initiated the operation, the operation performed (insert), the date and time of the operation, the name of the table on which the operation was performed, and the primary key value of the newly inserted row.

5) **Procedure for adding tuples to the Purchases table:**

```
PROCEDURE add_purchase(e_id in purchases.eid%type,
                       p_id in purchases.pid%type,
                       c_id in purchases.cid%type,
                       pur_qty in purchases.quantity%type,
                       pur_unit_price in purchases.unit_price%type) is
    -- local declarations
    v_qoh products.qoh%type;
    v_orig_price products.orig_price%type;
    v_saving purchases.saving%type;
    v_payment purchases.payment%type;
    begin
    dbms_output.enable();
    -- selecting the values of qoh and orig_price from products table to local
declarations.
    select qoh, orig_price into v_qoh, v_orig_price from Products where pid = p_id;
    -- if qoh is less then purchase quantity showing the error.
    if v_qoh < pur_qty then
        dbms_output.put_line('Insufficient quantity in stock.');
        return;
    end if;
    -- calculating the payment and saving
    v_payment :=  pur_qty * pur_unit_price;
    v_saving := pur_qty * (v_orig_price - pur_unit_price);
    dbms_output.put_line('entered the add_purchase procedure...');
    -- inserting the tuple into the purchase table
```

```
    insert into purchases (pur#, eid, pid, cid, pur_time, quantity, unit_price, payment,
saving)
    values (seqpur#.nextval, e_id, p_id, c_id, SYSDATE, pur_qty, pur_unit_price,
v_payment , v_saving);

    dbms_output.put_line('Purchase successfully completed.....');
    -- To show exceptions occured while inserting the tuple
    exception
    when others then
      dbms_output.put_line('Error while adding purchase.....' || SQLERRM);
    end;
```

This procedure named "add_purchase" that takes five input parameters of respective data types "purchases.eid%type", "purchases.pid%type", "purchases.cid%type", "purchases.quantity%type", and "purchases.unit_price%type" from purchases table.

The procedure starts by enabling output using "dbms_output.enable()".

Next, the procedure selects the "qoh" (quantity on hand) and "orig_price" (original price) values of the product with the specified "p_id" from the "Products" table into local variables "v_qoh" and "v_orig_price", respectively.

If the "qoh" value is less than the purchase quantity ("pur_qty"), an error message is displayed using "dbms_output.put_line()" and the procedure returns.

If the "qoh" value is sufficient, the procedure calculates the purchase payment by multiplying the purchase quantity by the purchase unit price and stores the result in the "v_payment" variable. The procedure also calculates the saving on the purchase by multiplying the purchase quantity by the difference between the original price and the purchase unit price and stores the result in the "v_saving" variable.

Then, the procedure inserts a new row into the "purchases" table using the input parameters passed to the procedure and the calculated values. The "sysdate" function is used to record the purchase time.

The procedure ends with an exception block that catches any errors that occur during the procedure's execution and displays the error message using "dbms_output.put_line()".

In summary, this procedure is used to add a new purchase to the "purchases" table by inserting a new row with the input parameters provided by the user. It also updates the "qoh" value of the corresponding product in the "Products" table and calculates the purchase payment and saving.

Trigger to update qoh in products table after making a purchase:

```
create or replace trigger trig_update_qoh
after insert on purchases
for each row
begin
   update products
   set qoh = qoh - :new.quantity
   where pid = :new.pid;
   -- To show execution of trigger
   dbms_output.put_line('trigger trig_update_qoh completed');
end;
/
```

This trigger is named trig_update_qoh. It is designed to execute automatically after a new row is inserted into the purchases table. The trigger performs an update operation on the products table for the row that has the same pid value as the newly inserted row in the purchases table.

The set clause in the UPDATE statement subtracts the value of the quantity column of the newly inserted row from the current value of the qoh (quantity on hand) column of the products table for the matching pid value.

The :new keyword refers to the newly inserted row in the purchases table, and the :new.quantity and :new.pid variables are used to obtain the corresponding values for the quantity and pid columns of the newly inserted row.

The dbms_output.put_line() statement is used to print a message indicating that the trigger has been executed. This message will be displayed in the console or output window of the database management system.

In summary, this trigger is designed to automatically update the qoh column of the products table after a new purchase is made by subtracting the quantity of the purchased product from the current quantity on hand.

Trigger if the purchase causes the qoh of the product to be below qoh_threshold.

```
create or replace trigger trig_check_qoh
after insert on purchases
for each row
declare
   var_qoh_threshold products.qoh_threshold%type;  -- variable to store the required
threshold of QOH for a product
   var_qoh products.qoh%type; -- variable to store the current qoh of the product with
the given ID
begin
```

```
    dbms_output.enable();
    select qoh, qoh_threshold into var_qoh, var_qoh_threshold
    from products
    where pid = :new.pid;
    -- Check if the qoh is below the threshold
    if ((var_qoh - :new.quantity) < var_qoh_threshold) then
        dbms_output.put_line('The current qoh of the product is below the required
threshold and new supply is required.');
        -- Update the qoh to the threshold value plus 20
        update products
        set qoh = var_qoh_threshold + 20
        where pid = :new.pid;
        -- Retrieve the updated qoh for the product
        select qoh into var_qoh
        from products
        where pid = :new.pid;
         dbms_output.put_line('The new value of qoh for the product is ' || var_qoh);
    end if;
    -- To show execution of trigger
    dbms_output.put_line('trigger trig_check_qoh completed');

end;
/
```

This trigger named "trig_check_qoh" is defined after insert trigger on the "purchases" table. The trigger executes for each row that is inserted into the "purchases" table.

The purpose of the trigger is to check if the qoh of a product falls below the threshold level after a new purchase is made, and if so, update the qoh to a new value.

The trigger starts by declaring two variables: "var_qoh_threshold" to store the required threshold of qoh for a product, and "var_qoh" to store the current qoh of the product with the given id.

Then, it enables dbms_output so that it can display messages to the user during the execution of the trigger.

Next, the trigger selects the current qoh and qoh threshold for the product that was just purchased using the "select into" statement.

After that, it checks whether the new quantity will cause the qoh to fall below the threshold using an if statement. If the qoh falls below the threshold, the trigger displays a message to the user using dbms_output, updates the qoh to the threshold value plus 20, retrieves the updated QOH for the product, and displays another message to the user with the new value.

Finally, the trigger displays a message indicating that the trigger has completed its execution.

Trigger to update visits_made and last_visit_made in customers table:

```
create or replace trigger trig_update_customer
after insert on purchases
for each row
begin
   update customers
   set visits_made = visits_made + 1,
      last_visit_date = case
                        when last_visit_date < :new.pur_time then :new.pur_time
                        else last_visit_date
                       end
   where cid = :new.cid;
   -- To show execution of trigger
   dbms_output.put_line('trigger trig_update_customer completed');
end;
/
```

This trigger named "trig_update_customer" is created every time a new record is inserted into the "purchases" table.

The trigger updates the "visits_made" and "last_visit_date" fields of the "customers" table for the customer associated with the purchase. Specifically, "visits_made" is incremented by 1 and "last_visit_date" is updated to the purchase time if it is later than the current value in the "last_visit_date" field.

The trigger does not declare any local variables, so it directly updates the "customers" table based on the values in the "purchases" table.

Finally, the trigger outputs a message indicating that it has completed its execution.

6) **Triggers in log table:**

a) **Trigger that adds tuple in the logs table when last_visit_date is updated in customers table**

```
create or replace trigger trig_log_last_visit_date_customers_update
after update on customers
for each row
begin
   dbms_output.enable();
```

```
    if :new.last_visit_date <> :old.last_visit_date then
        insert into logs(log#, user_name, operation, op_time, table_name, tuple_pkey)
        values(seqlog#.nextval, 'user', 'update', sysdate, 'customers', :new.cid);
        -- To show execution of trigger
        dbms_output.put_line('trigger trig_log_last_visit_date_customers_update
completed');
    end if;
end;
/
```

This trigger named trig_log_last_visit_date_customers_update is an after update trigger that fires once for each row that is updated in the customers table.

When the trigger fires, it enables the output buffer using dbms_output.enable(). Then it checks if the last_visit_date column has been updated by comparing :new.last_visit_date with :old.last_visit_date. If the last_visit_date has been updated, the trigger inserts a new row into the logs table using the insert into ... values statement. The logs table records information about changes made to the database. The inserted row includes a unique log# value generated by the sequence seqlog#.nextval, the user_name, operation ('update' in this case), the current time using sysdate, the name of the table_name ('customers'), and the primary key of the modified tuple, which is :new.cid.

Finally, the trigger outputs a message using dbms_output.put_line() indicating that the trigger has completed its execution.

b) Trigger that adds tuple in the logs table when visits_made is updated in customers table

```
create or replace trigger trig_log_visits_made_customers_update
after update on customers
for each row
begin
    dbms_output.enable();
    if :new.visits_made <> :old.visits_made then
        insert into logs(log#, user_name, operation, op_time, table_name, tuple_pkey)
        values(seqlog#.nextval, 'user', 'update', sysdate, 'customers', :new.cid);
        -- To show execution of trigger
        dbms_output.put_line('trigger trig_log_visits_made_customers_update
completed');
    end if;
end;
/
```

This trigger named trig_log_visits_made_customers_update that is fired after an update on the customers table. For each row that is updated, the trigger checks whether the visits_made column has been modified. If it has been modified, the trigger inserts a new record into the logs table, recording the user who performed the operation, the operation type (update), the operation time (using sysdate), the name of the updated table (customers), and the primary key value of the updated row (:new.cid).

The trigger also enables dbms_output, which allows for messages to be printed to the console during execution of the trigger. If the trigger successfully inserts a new record into the logs table, it prints a message to the console indicating that the trigger has completed its execution.

c) Trigger that adds tuple in the logs table when new tuple inserted into purchases table

```
create or replace trigger trig_log_purchases_insert
after insert on purchases
for each row
begin
   dbms_output.enable();
   insert into logs(log#, user_name, operation, op_time, table_name, tuple_pkey)
   values(seqlog#.nextval, 'user', 'insert', sysdate, 'purchases', :new.pur#);
   -- To show execution of trigger
   dbms_output.put_line('trigger trig_log_purchases_insert completed');
end;
/
```

This trigger named trig_log_purchases_insert fires after an insert is made on the purchases table. It logs information about the insert operation into the logs table. Specifically, it inserts a new row into the logs table, where log# is the next value of the sequence seqlog#, user_name is set to 'user', operation is set to 'insert', op_time is set to the current system date, table_name is set to 'purchases', and tuple_pkey is set to the value of the pur# column for the row that was just inserted. The dbms_output procedure is also used to print a message to the console to indicate that the trigger has completed its execution.

d) Trigger that adds tuple in the logs table when qoh is updated in products table

```
create or replace trigger trig_log_qoh_products_update
after update on products
for each row
begin
```

```
    dbms_output.enable();
    if :new.qoh <> :old.qoh then
        insert into logs(log#, user_name, operation, op_time, table_name, tuple_pkey)
        values(seqlog#.nextval, 'user', 'update', sysdate, 'products', :new.pid);
        -- To show execution of trigger
        dbms_output.put_line('trigger trig_log_qoh_products_update completed');
    end if;
end;
/
```

This trigger trig_log_qoh_products_update and is created to log the changes made to the "qoh" (quantity on hand) field in the "products" table. It is an "after update" trigger, meaning it will be executed automatically after a record has been updated in the "products" table.

The trigger starts by enabling dbms_output so that messages can be displayed in the console. It then checks if the "qoh" value in the old record is different from the "qoh" value in the new record. If they are different, it means that the "qoh" field was updated and an insert statement is executed to add a new record to the "logs" table. The new record will contain information about the update, such as the log#, user_name, operation, op_time, table_name, and tuple_pkey. Finally, a message is displayed using dbms_output to confirm that the trigger has completed.

Discussion Meetings:

Date	Plan	Execution
03/28/23	Study and read the project document that has been posted. Understand all that has to be done.	Executed according to plan.
04/01/23	The project will be discussed and divided into segments, as well as who will take on which portion.	Executed according to plan. Rakesh is responsible for the JDBC connection and Java, Shruti is responsible for PL/SQL procedures and functions, and Rashmika is responsible for PL/SQL triggers.
04/04/23	Get started with the project.	Delayed by a day and was started on 04/05/23.

04/07/23	Completion of 40% of the project.	Executed according to plan.
04/10/23	Discussion of errors and issues faced and work on them together.	We had difficulty determining a solution to the error.
04/11/23	Work on the error.	Executed according to plan.
04/15/23	Completion of 70% of the project.	Executed according to plan.
04/17/23	Finish the project and start with documentation.	Executed according to plan.
04/19/23	Finish the documentation and prepare for the demo.	Delayed and was completed on 04/21/23.

The team had few minor issues but worked well together.

This project was executed successfully by:

Shruti Dhande B00983135
Rashmika Alishetty B00982026
Rakesh Palle B00981960