

Ex. No. 11	B A S I C P L / S Q L	Date :
-------------------	--------------------------------	---------------

Q1) Write a PL/SQL Block to find the maximum of 3 Numbers

```

Declare
    a number;
    b number;
    c number;
Begin
    dbms_output.put_line('Enter a:');
    a:=&a;
    dbms_output.put_line('Enter b:');
    b:=&b;
    dbms_output.put_line('Enter c:');
    c:=&c;
    if (a>b) and (a>c) then
        dbms_output.putline('A is Maximum');
    elsif (b>a) and (b>c) then
        dbms_output.putline('B is Maximum');
    else
        dbms_output.putline('C is Maximum');
    end if;
End;
/

```

Q2) Write a PL/SQL Block to find the sum of odd numbers upto 100 using loop statement

Declare

Begin

End;
/

Q3) Write a PL/SQL block to get the salary of the employee who has empno=7369 and update his salary as specified below

- if his/her salary < 2500, then increase salary by 25%
- otherwise if salary lies between 2500 and 5000, then increase salary by 20%
- otherwise increase salary by adding commission amount to the salary.

Declare

Salary number(5);

Begin

Select sal into salary from emp where empno=7369;

-- complete remaining statements

End;

/

Q4) Write a PL/SQL Block to modify the department name of the department 71 if it is not 'HRD'.

Declare

deptname dept.dname%type;

Begin

-- complete the block

End;

/

C U R S O R

A cursor is a temporary work area created in the system memory when a SQL statement is executed. A cursor contains information on a select statement and the rows of data accessed by it. This temporary work area is used to store the data retrieved from the database, and manipulate this data. A cursor can hold more than one row, but can process only one row at a time.

There are two types of cursors in PL/SQL. They are Implicit cursors and Explicit cursors.

Implicit cursors

These are created by default when DML statements like, INSERT, UPDATE, and DELETE statements are executed.

Oracle provides few attributes called as implicit cursor attributes to check the status of DML operations. The cursor attributes available are %FOUND, %NOTFOUND, %ROWCOUNT, and %ISOPEN.

For example, When you execute INSERT, UPDATE, or DELETE statements the cursor attributes tell us whether any rows are affected and how many have been affected.

When a SELECT... INTO statement is executed in a PL/SQL Block, implicit cursor attributes can be used to find out whether any row has been returned by the SELECT statement. PL/SQL returns an error when no data is selected.

Implicit Cursor Attributes

%FOUND

The return value is TRUE, if the DML statements like INSERT, DELETE and UPDATE affect at least one row or if SELECT ...INTO statement return at least one row.

Ex. SQL%FOUND

%NOTFOUND

The return value is FALSE, if DML statements affect at least one row or if SELECT. ...INTO statement return at least one row.

Ex. SQL%NOTFOUND

%ROWCOUNT

Return the number of rows affected by the DML operations

Ex. SQL%ROWCOUNT

- Q5)** Write a PL/SQL Block, to update salaries of all the employees who work in deptno 20 by 15%. If none of the employee's salary are updated display a message '*None of the salaries were updated*'. Otherwise display the total number of employee who got salary updated.

Declare

num number(5);

Begin

update emp set sal = sal + sal*0.15 where deptno=20;

if SQL%NOTFOUND then

dbms_output.put_line('none of the salaries were updated');

```

elseif SQL%FOUND then
    num := SQL%ROWCOUNT;
    dbms_output.put_line('salaries for ' || num || 'employees are updated');
end if;
End;

```

Explicit cursors

They must be created when you are executing a SELECT statement that returns more than one row. Even though the cursor stores multiple records, only one record can be processed at a time, which is called as current row. When you fetch a row the current row position moves to next row.

There are four steps in using an Explicit Cursor.

- DECLARE the cursor in the declaration section.
- OPEN the cursor in the Execution Section.
- FETCH the data from cursor into PL/SQL variables or records in the Execution Section.
- CLOSE the cursor in the Execution Section before you end the PL/SQL Block.

Declaring Cursor :

```
CURSOR cursor_name IS select_statement;
```

Opening Cursor :

```
OPEN cursor_name;
```

Fetching Cursor :

```
FETCH cursor_name INTO variable-list/record-type;
```

Closing Cursor :

```
CLOSE cursor_name;
```

Explicit Cursor Attributes

%FOUND

TRUE, if fetch statement returns at least one row.

Ex. Cursor_name%FOUND

%NOTFOUND

TRUE, if fetch statement doesn't return a row.

Ex. Cursor_name%NOTFOUND

%ROWCOUNT

The number of rows fetched by the fetch statement.
Ex. Cursor_name%ROWCOUNT

%ISOPEN

TRUE, if the cursor is already open in the program.
Ex. Cursor_name%ISOPEN

- Q6)** Create a table *emp_grade* with columns *empno* & *grade*. Write PL/SQL block to insert values into the table *emp_grade* by processing *emp* table with the following constraints.

If sal <= 1400 then grade is 'C'

Else if sal between 1401 and 2000 then the grade is 'B' Else the grade is 'A'.

SQL> create table emp_grade(empno number, grade char(1));

```
Declare      Emp_rec emp%rowtype;
              Cursor c is select * into emp_rec from emp;
Begin
    Open c;
    If c%ISOPEN then
        Loop
            Fetch c into emp_rec;
            If c%notfound then Exit; Endif;
            If emp_rec.sal <= 1400 then
                Insert into emp_grade values(emp_rec.empno,'C');
            Elsif emp_rec.sal between 1401 and 2000 then
                Insert into emp_garde values(em_rec.empno,'B');
            Else
                Insert into emp_garde values(em_rec.empno,'A');
            Endif
        End loop;
    Else
        Open c;
    Endif;
End;
```

- Q7)** Write a PL/SQL block to do the following :
- Total wages of the company (Sum of the salaries and commission values of all the employees in *emp* table)
 - Total number of highly paid employees. (Employees with salary > 2000)
 - Total number of employees who get commission that is higher than their salary.

Q8) Write a PL/SQL block to find the name and salary of first five highly paid employees.

Cursor for loop

Cursor for loop automatically opens a cursor, fetches each row and closes the cursor when all rows have been processed.

Ex.

Declare

Cursor s1 is select

Begin

For var *in* s1

Loop

-- statements ---

End loop;

.. .. .

Q9) Solve the program in question number Q4 using cursor for...loop

Q10) Write a PL/SQL block to find the names of employees & job and total number of employees who have more than 28 years of service in the company.(Use for loop)

II. TRIGGER

A trigger is a PL/SQL block structure which is fired when DML statements like Insert, Delete and Update is executed on a database table. A trigger is triggered automatically when an associated DML statement is executed.

Syntax of Trigger

```
CREATE [OR REPLACE ] TRIGGER trigger_name  
{BEFORE | AFTER | INSTEAD OF }  
{INSERT [OR] | UPDATE [OR] | DELETE}  
[OF col_name]  
ON table_name  
[REFERENCING OLD AS o NEW AS n]  
[FOR EACH ROW]  
WHEN (condition)  
BEGIN  
    -- SQL Statements  
END;
```

CREATE [OR REPLACE] TRIGGER trigger_name

This clause creates a trigger with the given name or overwrites an existing trigger with the same name.

BEFORE | AFTER | INSTEAD OF

This clause indicates at what time the trigger should get fired. i.e for example: before or after updating a table. INSTEAD OF is used to create a trigger on a view. Before and after cannot be used to create a trigger on a view.

INSERT [OR] | UPDATE [OR] | DELETE

This clause determines the triggering event. More than one triggering events can be used together separated by OR keyword. The trigger gets fired at all the specified triggering event.

OF col_name

This clause is used with update triggers. This clause is used when you want to trigger an event only when a specific column is updated.

ON table_name

This clause identifies the name of the table/view to which the trigger is associated.

REFERENCING OLD AS o NEW AS n

This clause is used to reference the old and new values of the data being changed. By default, you reference the values as **:old.column_name** or **:new.column_name**. The reference names can also be changed from old (or new) to any other user-defined name. You cannot reference old values when inserting a record, or new values when deleting a record, because they do not exist.

FOR EACH ROW

This clause is used to determine whether a trigger must fire when each row gets affected (i.e. a Row Level Trigger) or just once when the entire SQL statement is executed (i.e. statement level Trigger).

WHEN (condition)

This clause is valid only for row level triggers. The trigger is fired only for rows that satisfy the condition specified.

Types of Triggers

There are two types of triggers based on which level it is triggered.

- *Row level trigger* : An event is triggered for each row updated, inserted or deleted.
- *Statement level trigger* : An event is triggered for each SQL statement executed.

Before and After Triggers : Since triggers occur because of events, they may be set to occur immediately *before* or *after* those events. Within the trigger, we are able to refer *old* and *new* values involved in transactions. *Old* refers to the data as it existed prior to the transaction. *New* refer to the data that the transaction creates.

Q11) Before Update : Row level Trigger

Employees may get promoted and continue servicing with new designation. To maintain the job history of the employees, create a table *job_history* with columns *empno*, *ename*, *job*, *pro_date*, and create a trigger to update the table *job_history* whenever there is an updation in *job* column of any row in *emp* table.

```
Create or replace trigger job_history_trigger
Before update of job
on emp
For each row
Begin
    Insert into job_history values(:old.empno,:old.ename,:old.job,sysdate);
End;
```

Q12) Before Insert : Row level Trigger

Create a trigger to convert *employee name* into upper case, before we insert any row into the table *emp* with *employee name* in either case.
(Hint. :new.ename refers the value that is to be inserted)

Q13) After delete : Row level Trigger

Consider tables *dept* and *deptold* with same structure. Create a trigger to move the row into second table whenever a row is removed from first table.

Q14) Create a trigger which will not allow you to enter duplicate or null values in column *empno* of *emp* table.

```
create or replace trigger dubb
before insert on emp
for each row
declare
    cursor c1 is select * from emp;
    x emp%rowtype;
begin
    open c1;
    loop
        fetch c1 into x;
        if :new.empno = x.empno then
            dbms_output.put_line('you entered duplicated no');
        elseif :new.empno is null then
            dbms_output.put_line('you empno is null');
        end if;
        exit when c1%notfound;
    end loop;
    close c1;
end;
```

Q15) Before Insert/Update/Delete : Statement level Trigger

Create a database trigger that allows changes to employee table only during the business hours(i.e. from 8 a.m to 5 p.m.) from Monday to Friday. There is no restriction on viewing data from the table

```
Create or replace trigger time_check
Before insert or update or delete on emp
Begin
    if to_number(to_char(sysdate,'hh24')) < 8 or
       to_number(to_char(sysdate,'hh24')) >= 17 or
       to_char(sysdate,'DY') = 'SAT' or to_char(sysdate,'day') = 'SUN' then
        raise_application_error(-20004,'you can access only between 8am
        to 5pm on Monday to Friday');
    end if;
end;
```

Information about Triggers

We can use the data dictionary 'USER_TRIGGERS' to obtain information about any trigger. The below statement shows the structure of 'USER_TRIGGERS'.

```
SQL> desc user_triggers;
```

Q16) Find the trigger type, trigger event and table name of the trigger 'time_check'.

```
SQL> select trigger_type, trigger_event, table_name
       from user_triggers where trigger_name = 'TIME_CHECK';
```

Enabling and Disabling Triggers

Syntax : **ALTER TRIGGER trigger_name ENABLE | DISABLE** (or)
 ALTER TABLE table_name ENABLE | DISABLE ALL TRIGGERS;

Q17) Disable the trigger 'job_history_trigger'

```
SQL>
```

Q18) Disable all the triggers of emp table.

```
SQL>
```

Q19) Drop the trigger 'dubb'

```
SQL> drop trigger dubb;
```

Verified by

Staff In-charge Sign :	Date :
-------------------------------	---------------

PL/SQL INTRODUCTION

PL/SQL

- PL/SQL bridges the gap between database technology and procedural programming languages.
- PL/SQL uses the facilities of the sophisticated RDBMS and extends the standard SQL database language
- Not only PL/SQL allow you to insert, delete, update and retrieve data, it lets you use procedural techniques such as looping and branching to process the data.
- Thus PL/SQL provides the data manipulating power of SQL with the data processing power of procedural languages

Advantage of PL/SQL

PL/SQL is a completely portable, high performance transaction processing language. It provides the following advantages :

- Procedural Capabilities
 - It supports many of constructs like constants, variable, selection and iterative statements
- Improved Performance
 - Block of SQL statements can be executed at a time
- Enhanced Productivity
 - PL/SQL brings added functionality to non procedural tools such as SQL Forms.
- Portability
 - PL/SQL can run anywhere the RDBMS can run
- Integration with RDBMS
 - Most PL/SQL variables have data types native to the RDBMS data dictionary. Combined with the direct access to SQL, these native data type declarations allow easy integration of PL/SQL with RDBMS.

Character Set

It is either ASCII or EBCDIC format

Identifiers

It begins with a letter and can be followed by letters, numbers, \$ or #. Maximum size is 30 characters in length.

Variable Declaration

The data types (number, varchar2, real, date, ...) discussed in SQL are all applicable in PL/SQL.

Ex. Salary *Number(7,2);*
 Sex *Boolean;*
 Count *smallint :=0;*
 Tax *number default 750;*
 Name *varchar2(20) not null;*

Constant declaration

Ex. Phi *Constant Number(7,2) := 3.1417;*

Comment

Line can be commented with double hyphen at the beginning of the line.

Ex. - - This is a comment line

Assignments

Variable assignment sets the current value of a variable. You can assign values to a variable as follows

(i) Assignment operator (:=)

Ex. d := b*b – 4*a*c;

(ii) Select ... into statement

Ex. *Select sal into salary from emp where empno=7655;*

Operators

Operators used in SQL are all applicable to PL/SQL also.

Block Structure

PL/SQL code is grouped into structures called blocks. If you create a stored procedure or package, you give the block of PL/SQL code a name. If the block of PL/SQL code is not given a name, then it is called an anonymous block.

The PL/SQL block divided into three section: declaration section, the executable section and the exception section

The structure of a typical PL/SQL block is shown in the listing:

```
declare
    < declaration section >
begin
    < executable commands>
exception
    <exception handling>
end;
```

Declaration Section :

Defines and initializes the variables and cursor used in the block

Executable commands :

Uses flow-control commands (such as IF command and loops) to execute the commands and assign values to the declared variables

Exception handling :

Provides handling of error conditions

Declaration Using attributes

(i) %type attribute

The %TYPE attribute provides the data type of a variable, constant, or database column. Variables and constants declared using %TYPE are treated like those declared using a data type name.

For example in the declaration below, PL/SQL treats debit like a REAL(7,2) variable.

```
credit REAL(7,2);
debit credit%TYPE;
```

The %TYPE attribute is particularly useful when declaring variables that refer to database columns. You can reference a table and column, or you can reference an owner, table, and column.

```
my_dname dept.dname%TYPE;
```

Using %TYPE to declare my_dname has two advantages.

- First, you need not know the exact datatype of dname.
- Second, if the database definition of dname changes, the datatype of my_dname changes accordingly at run time.

(ii) %rowtype attribute

The %ROWTYPE attribute provides a record type that represents a row in a table (or view). The record can store an entire row of data selected from the table or fetched by a cursor.

```
DECLARE
    emp_rec emp%ROWTYPE;
...
BEGIN
    SELECT * INTO emp_rec FROM emp WHERE ...
    ...
END;
```

Columns in a row and corresponding fields in a record have the same names and data types.

The column values returned by the SELECT statement are stored in fields. To reference a field, you use the dot notation.

```
IF emp_rec.deptno = 20 THEN ...
```

In addition, you can assign the value of an expression to a specific field.

```
emp_rec.ename := 'JOHNSON';
```

A %ROWTYPE declaration cannot include an initialization clause. However, there are two ways to assign values to all fields in a record at once.

First, PL/SQL allows aggregate assignment between entire records if their declarations refer to the same table or cursor.

```
DECLARE
    dept_rec1 dept%ROWTYPE;
    dept_rec2 dept%ROWTYPE;
    ....
BEGIN
    ..
    ....
    dept_rec1 := dept_rec2;
    ....
END;
```

Second, you can assign a list of column values to a record by using the SELECT and FETCH statement, as the example below shows. The column names must appear in the order in which they were defined by the CREATE TABLE or CREATE VIEW statement.

```
DECLARE
    dept_rec dept%ROWTYPE;
    ....
BEGIN
    SELECT deptno, dname, loc INTO dept_rec FROM dept
    WHERE deptno = 30;
    ....
END;
```

However, you cannot assign a list of column values to a record by using an assignment statement. Although you can retrieve entire records, you cannot insert them.

For example, the following statement is illegal:

```
INSERT INTO dept VALUES (dept_rec); -- illegal
```

Creating and Executing PL/SQL Programs

Edit your PL/SQL program in your favourite editor as text file.

Execute the following command once for a session to get displayed the output.

```
SQL> set serveroutput on;
```

Now execute the program using the following command.

```
SQL> start filename;          (or)   SQL> @filename;
```

Note : Give absolute path of the filename if you saved the file in some directory.

Ex. SQL> start z:\plsql\ex11; (or) SQL> @ z:\plsql\ex11;

Control Structures

(i) IF Statements

There are three forms of IF statements: IF-THEN, IF-THEN-ELSE, and IF-THEN-ELSIF. The third form of IF statement uses the keyword ELSIF (NOT ELSEIF) to introduce additional conditions, as follows:

```
IF condition1 THEN
    sequence_of_statements1;
ELSIF condition2 THEN
    sequence_of_statements2;
ELSE
    sequence_of_statements3;
END IF;
```

(ii) LOOP and EXIT Statements

There are three forms of LOOP statements. They are LOOP, WHILE-LOOP, and FOR-LOOP.

LOOP

The simplest form of LOOP statement is the basic (or infinite) loop, which encloses a sequence of statements between the keywords LOOP and END LOOP, as follows:

```
LOOP
    sequence_of_statements3;
    ...
END LOOP;
```

With each iteration of the loop, the sequence of statements is executed, then control resumes at the top of the loop. If further processing is undesirable or impossible, you can use the EXIT statement to complete the loop. You can place one or more EXIT statements anywhere inside a loop, but nowhere outside a loop. There are two forms of EXIT statements: EXIT and EXIT-WHEN.

The EXIT statement forces a loop to complete unconditionally. When an EXIT statement is encountered, the loop completes immediately and control passes to the next statement.

```
LOOP
    ...
    IF ... THEN
        ...
        EXIT; -- exit loop immediately
    END IF;
END LOOP;
-- control resumes here
```

The EXIT-WHEN statement allows a loop to complete conditionally. When the EXIT statement is encountered, the condition in the WHEN clause is evaluated. If the condition evaluates to TRUE, the loop completes and control passes to the next statement after the loop.

```
LOOP
    ....
    EXIT WHEN i>n; -- exit loop if condition is true
    ....
END LOOP;
....
```

Until the condition evaluates to TRUE, the loop cannot complete. So, statements within the loop must change the value of the condition.

Like PL/SQL blocks, loops can be labeled. The label, an undeclared identifier enclosed by double angle brackets, must appear at the beginning of the LOOP statement, as follows:

```
<<label_name>>
LOOP
    sequence_of_statements;
    ...
END LOOP [label_name];
```

Optionally, the label name can also appear at the end of the LOOP statement.

With either form of EXIT statement, you can complete not only the current loop, but any enclosing loop. Simply label the enclosing loop that you want to complete, then use the label in an EXIT statement.

```

<<outer>>
LOOP
    ...
    LOOP
        ...
        EXIT outer WHEN ... -- exit both loops
    END LOOP;
...
END LOOP outer;

```

(iii) **WHILE-LOOP**

The WHILE-LOOP statement associates a condition with a sequence of statements enclosed by the keywords LOOP and END LOOP, as follows:

```

WHILE condition LOOP
    sequence_of_statements;
...
END LOOP;

```

Before each iteration of the loop, the condition is evaluated. If the condition evaluates to TRUE, the sequence of statements is executed, then control resumes at the top of the loop. If the condition evaluates to FALSE or NULL, the loop is bypassed and control passes to the next statement. Since the condition is tested at the top of the loop, the sequence might execute zero times.

(iv) **FOR-LOOP**

Whereas the number of iteration through a WHILE loop is unknown until the loop completes, the number of iterations through a FOR loop is known before the loop is entered. FOR loops iterate over a specified range of integers. The range is part of an iteration scheme, which is enclosed by the keywords FOR and LOOP.

```

FOR counter IN [REVERSE] lower_bound..upper_bound LOOP
    sequence_of_statements;
...
END LOOP;

```

The lower bound need not be 1. However, the loop counter increment (or decrement) must be 1. PL/SQL lets you determine the loop range dynamically at run time, as the following example shows:

```

SELECT COUNT(empno) INTO emp_count FROM emp;
FOR i IN 1..emp_count LOOP
    ...
END LOOP;

```

The loop counter is defined only within the loop. You cannot reference it outside the loop. You need not explicitly declare the loop counter because it is implicitly declared as a local variable of type INTEGER.

The EXIT statement allows a FOR loop to complete prematurely. You can complete not only the current loop, but any enclosing loop.

(v) GOTO and NULL statements

Unlike the IF and LOOP statements, the GOTO and NULL statements are not crucial to PL/SQL programming. The structure of PL/SQL is such that the GOTO statement is seldom needed. Occasionally, it can simplify logic enough to warrant its use. The NULL statement can make the meaning and action of conditional statements clear and so improve readability.

```
BEGIN
    ...
    GOTO insert_row;
    ...
    <<insert_row>>
    INSERT INTO emp VALUES ...
END;
```

A GOTO statement cannot branch into an IF statement, LOOP statement, or sub-block. A GOTO statement cannot branch from one IF statement clause to another. A GOTO statement cannot branch out of a subprogram. Finally, a GOTO statement cannot branch from an exception handler into the current block.

The NULL statement explicitly specifies inaction; it does nothing other than pass control to the next statement. It can, however, improve readability. Also, the NULL statement is a handy way to create stubs when designing applications from the top down.

* * *

Ex. No. 11	B A S I C P L / S Q L	Date :
-------------------	--------------------------------	---------------

Q1) Write a PL/SQL Block to find the maximum of 3 Numbers

```

Declare
    a number;
    b number;
    c number;
Begin
    dbms_output.put_line('Enter a:');
    a:=&a;
    dbms_output.put_line('Enter b:');
    b:=&b;
    dbms_output.put_line('Enter c:');
    c:=&c;
    if (a>b) and (a>c) then
        dbms_output.putline('A is Maximum');
    elsif (b>a) and (b>c) then
        dbms_output.putline('B is Maximum');
    else
        dbms_output.putline('C is Maximum');
    end if;
End;
/

```

Q2) Write a PL/SQL Block to find the sum of odd numbers upto 100 using loop statement

Declare

Begin

End;
/

Ex. No. 7	SUB QUERIES	Date :
-----------	-------------	--------

- Nesting of queries, one within the other is termed as sub query.

Syntax

```

SELECT    select_list
FROM      table
WHERE     expr operator ( SELECT    select_list
                                FROM      table);

```

- The subquery (inner query) executes once before the main query.
- The result of the subquery is used by the main query (outer query).

Guidelines for Subqueries

- Enclose subqueries in parentheses.
- Place subqueries on the right side of the comparison operator.
- Do not add an ORDER BY clause to a subquery.
- Use single-row operators with single-row subqueries.
- Use multiple-row operators with multiple-row subqueries.

Single-Row Subqueries

- Return only one row
- Use single-row comparison operators (ie; relational operators)

Multiple-Row Subqueries

- Return more than one row
- Use multiple-row comparison operators

<i>Operator</i>	<i>Meaning</i>
IN	Equal to any member in the list
ANY	Compare value to each value returned by the subquery
ALL	Compare value to every value returned by the subquery

Note:

‘=any’ is equivalent to ‘in’

‘!=all’ is equivalent to ‘not in’

- Q1)** List the name of the employees whose salary is greater than that of employee with empno 7566.

```
SQL> select ename from employee
      where sal > (select sal from employee
                  where empno=7566);
```

- Q2)** List the name of the employees whose job is equal to the job of employee with empno 7369 and salary is greater than that of employee with empno 7876.

```
SQL>
```

- Q3)** List the *ename, job, sal* of the employee who get minimum salary in the company.

```
SQL> select ename, job, sal from employee
      where sal = (select min(sal) from employee);
```

- Q4)** List *deptno* & *min(salary)* departmentwise, only if *min(sal)* is greater than the *min(sal)* of *deptno* 20.

```
SQL> select deptno, min(sal) from employee
      group by deptno
      having min(sal) > (select min(sal) from employee
                        where deptno = 20);
```

- Q5)** List *empno, ename, job* of the employees whose *job* is not a 'CLERK' and whose *salary* is less than at least one of the salaries of the employees whose *job* is 'CLERK'.

```
SQL> select empno, ename, job from employee
      where sal < any (select sa from employee
                     where job = 'CLERK')
      and job <> 'CLERK';
```

- Q6)** List *empno*, *ename*, *job* of the employees whose salary is greater than the average salary of each department.

SQL>

- Q7)** Display the *name*, *dept. no*, *salary*, and *commission* of any employee whose *salary* and *commission* matches both the *commission* and *salary* of any employee in department 30.

```
SQL> select  ename, deptno, sal, comm  
          from  employee  
          where (sal, nvl(comm,-1)) in ( select sal, nvl(comm,-1)  
                                     from  employee  
                                     where deptno = 30);
```

- Q8)** List *ename* *sal*, *deptno*, *average salary* of the dept where he/she works, if salary of the employee is greater than his/her department average salary.

```
SQL> select  a.ename, a.sal, a.deptno, b.salavg  
          from  employee a, ( select  deptno, avg(sal) salavg  
                             from  employee  
                             group by deptno) b  
          where a.deptno = b.deptno  
          and   a.sal > b.salavg;
```

- Q9)** Execute and Write the output of the following query in words.

```
SQL> with summary as  
      (select  dname,sum(sal) as dept_total  from employee a , department b  
        where  a.deptno = b.deptno  
        group by dname);  
      select dname,dept_total  from summary  
      where dept_total > (select sum(dept_total)*1/3  from summary)  
      order by dept_total desc;
```

Q10) List *ename, job, sal* of the employees whose salary is equal to any one of the salary of the employee 'SCOTT' and 'WARD'.

SQL>

Q11) List *ename, job, sal* of the employees whose salary and job is equal to the employee 'FORD'.

SQL>

Q12) List *ename, job, deptno, sal* of the employees whose job is same as 'JONES' and *salary* is greater than the employee 'FORD'.

SQL>

Q13) List *ename, job* of the employees who work in *deptno* 10 and his/her *job* is any one of the job in the department 'SALES'.

SQL>

Q14) Execute the following query and write the result in word

**SQL> select *job,ename,empno,deptno* from *emp s*
 where exists (select * from *emp*
 where *s.empno=mgr*)
 order by *empno*;**

Verified by

Staff In-charge Sign :	Date :
-------------------------------	---------------

Ex. No. 6	JOINING TABLES	Date :
------------------	-----------------------	---------------

Set operators

- Set operators combine the results of two queries into a single.

Operator	Function
Union	Returns all distinct rows selected by either query
Union all	Returns all rows selected by either query including duplicates
Intersect	Returns only rows that are common to both the queries
Minus	Returns all distinct rows selected only by the first query and not by the second.

Q1) Create the following tables :

depositor(cus_name,acno) & borrower(cus_name,loanno)

SQL>

SQL>

Q2) List the names of distinct customers who have either loan or account

SQL>

Q3) List the names of customers (with duplicates) who have either loan or account

**SQL> (select *cus_name* from *borrower*)
union all (select *cus_name* from *depositor*)**

Q4) List the names of customers who have both loan and account

SQL>

Q5) List the names of customers who have loan but not account

SQL>

Joins

- Used to combine the data spread across tables

Syntax

```
SELECT    table1.column, table2.column
FROM      table1, table2
WHERE     table1.column1 = table2.column2;
```

- A JOIN Basically involves more than one Table to interact with.
- Where clause specifies the JOIN Condition.
- Ambiguous Column names are identified by the Table name.
- If join condition is omitted, then a **Cartesian product** is formed. That is all rows in the first table are joined to all rows in the second table

Types of Joins

- Inner Join (Simple Join) : It retrieves rows from 2 tables having a common column.
 - Equi Join : A join condition with relationship = .
 - Non Equi Join : A join condition with relationship other than = .
- Self Join : Joining of a table to itself
- Outer Join : Returns all the rows returned by simple join as well as those rows from one table that do not match any row from the other table. The symbol (+) represents outer joins.

Q6) List *empno*, *ename*, *deptno* from *emp* and *dept* tables.

SQL>

Q7) Create a table *Salgrade* with the following data .

	Grade	Losal	Hisal
1		700	1400
2		1401	2000
3		2001	5000
4		5001	9999

Now, list *ename*, *sal* and *salgrade* of all employees.

SQL>

Q8) List *ename*, *deptno* and *deptname* from *emp* and *dept* tables, including the rows of *emp* table that does not match with any of the rows in *dept* table.
SQL>

Q9) List *ename*, *deptno* and *deptname* from *emp* and *dept* tables, including the rows of *dept* table that does not match with any of the rows in *emp* table.
SQL>

Q10) List the names of the employee with name of his/her manager from *emp* table.
SQL>

Verified by

Staff In-charge Sign :	Date :
-------------------------------	---------------

AIM

To study the various SQL view operations on the database.

1. CREATE VIEW command is used to define a view.
2. INSERT command is used to insert a new row into the view.
3. DELETE command is used to delete a row from the view.
4. UPDATE command is used to change a value in a tuple without changing all values in the tuple.
5. DROP command is used to drop the

view table **Commands Execution**

Creation Of Table

```
-----  
Sql> Create Table  
Employee (  
Employee_Name varchar  
2(10),  
Employee_No number(  
8), Dept_Name  
Varchar2(10),  
Dept_No Number  
(5), Date_Of_Join Date); Table  
Created.
```

Table Description

```
-----  
Sql> Desc  
Employee;  
Name Null?  
Type
```

```
-----  
Employee_Name  
Varchar2(10)  
Employee_No  
Number(8)  
Dept_Name  
Varchar2(10)  
Dept_No Number(5)  
Date_Of_Join Date
```

Syntax For Creation Of View

Sql> Create <View> <View Name> As Select
<Column_Name_1>, <Column_Name_2> From <Table
Name>; Creation Of View

Sql> Create View Empview As Select
Employee_Name,Employee_No,Dept_Name,Dept_No,Date_Of_J
oin From Employee;
View Created.
Description Of View

Sql> Desc
Empview;
Name Null?
Type

Employee_Name
Varchar2(10)
Employee_No
Number(8)
Dept_Name
Varchar2(10)
Dept_No Number(5)

Display View:

Sql> Select * From Empview;
Employee_N Employee_No Dept_Name Dept_No

Ravi 124 Ece 89
Vijay 345 Cse 21
Raj 98 It 22
Giri 100

Cse 67

Insertion

Into View

Insert Statement:
Syntax:

```

Sql> Insert Into <View_Name> (Column
Name1,.....) Values(Value1,...);
Sql> Insert Into Empview Values ('Sri', 120,'Cse', 67,'16-
Nov-1981'); 1 Row Created.
Sql> Select * From Empview;
Employee_N Employee_No Dept_Name Dept_No

```

```

-----
Ravi 124 Ece 89
Vijay 345 Cse 21
Raj 98 It 22
Giri 100 Cse 67
Sri 120 Cse 67
Sql> Select * From Employee;

```

```

-----
Employee_N Employee_No Dept_Name Dept_No Date_Of_J
Ravi 124 Ece 89 15-Jun-05
Vijay 345 Cse 21 21-Jun-06
Raj 98 It 22 30-Sep-06
Giri 100 Cse 67 14-Nov-81
Sri 120 Cse 67
16-Nov-81

```

Deletion Of

View:

Delete Statement:

Syntax:

```

Sql> Delete <View_Nmae>Where <Column Nmae>
='Value'; Sql> Delete From Empview Where
Employee_Name='Sri';
1 Row Deleted.

```

```

Sql> Select * From Empview;
Employee_N Employee_No Dept_Name Dept_No

```

```

-----
Ravi 124 Ece 89
Vijay 345 Cse 21
Raj 98 It 22
Giri 100
Cse 67

```

Update

Stateme

nt:

Syntax:

```

Aql>Update <View_Name> Set< Column Name> = <Column Name>
+<View> Where <Columnname>=Value;
Sql> Update Empkaviview Set

```

Employee_Name='Kavi' Where
Employee_Name='Ravi';
1 Row Updated.
Sql> Select * From Empkaviview;
Employee_N Employee_No Dept_Name Dept_No

Kavi 124 Ece 89

Vijay 345 Cse 21

Raj 98 It 22

Giri

100

Cse 67

Drop

A

View:

Syntax:

Sql> Drop View

<View_Name>

Example

Sql>Drop View Empview;

View Dropped

Create A View With Selected Fields:

Syntax:

Sql>Create [Or Replace] View <View Name>As Select

<Column Name1>.....From <Table Anme>;

Example-2:

Sql> Create Or Replace View Empl_View1 As Select Empno,

Ename, Salary From Empl;

Sql> Select * From

Empl_View1; Example-

3:

Sql> Create Or Replace View Empl_View2 As Select * From

Empl Where Deptno=10;

Sql> Select * From

Empl_View2; Note:

- Replace Is The Keyword To Avoid The Error “Ora_0095:Name Is Already Used By An Existing Object”.

Changing The Column(S) Name M The View During

As Select Statement:

Type-1:

```
Sql> Create Or Replace View Emp_Totsal(Eid,Name,Sal) As
Select Empno,Ename,Salary From Empl;
View Created.
Empno Ename Salary
```

```
-----
7369 Smith 1000
7499 Mark 1050
7565 Will 1500
7678 John 1800
7578 Tom 1500
7548 Turner 1500
```

```
6 Rows
Selecte
d.
```

```
View
Created.
```

```
Empno Ename Salary Mgrno Deptno
```

```
-----
7578 Tom 1500 7298 10
7548 Turner 1500 7298 10
```

```
View Created.
```

```
Sql> Select * From
Emp_Totsal; Type-2:
```

```
Sql> Create Or Replace View Emp_Totsal As Select Empno
"Eid",Ename "Name",Salary "Sal" From Empl;
```

```
Sql> Select * From
Emp_Totsal; Example
```

```
For Join View:
```

```
Type-3:
```

```
Sql> Create Or Replace View Dept_Emp As Select A.Empno
"Eid",A.Ename "Empname",A.Deptno "Dno",B.Dnam
E "D_Name",B.Loc "D_Loc" From Empl A,Depmt
B Where A.Deptno=B.Deptno;
```

```
Sql> Select * From
Dept_Emp; Eid Name
Sal
```

```
-----
7369 Smith 1000
7499 Mark 1050
7565 Will 1500
7678 John 1800
7578 Tom 1500
7548 Turner 1500
```

```
6 Rows
```


Selecte
d.
View
Created.
Eid Name Sal

7369 Smith 1000
7499 Mark 1050
7565 Will 1500
7678 John 1800
7578 Tom 1500
7548 Turner 1500

6 Rows
Selecte
d.

View
Created.

Eid Empname Dno D_Name D_Loc

7578 Tom 10 Account New York
7548 Turner 10 Account New York
7369 Smith 20 Sales Chicago
7678 John 20 Sales Chicago
7499 Mark 30 Research Zurich
7565 WILL 30 RESEARCH ZURICH

RESULT:

Thus the SQL views have been executed successfully.

Ex.No. 3	INTEGRITY CONSTRAINTS	Date :
----------	-----------------------	--------

CONSTRAINTS

- Constraints enforce rules at the table level. Constraints prevent the deletion of a table if there are dependencies.
- The following constraint types are valid in Oracle:
 - NOT NULL
 - UNIQUE Key
 - PRIMARY KEY
 - FOREIGN KEY
 - CHECK
- Name a constraint or the Oracle Server will generate a name by using the SYS_Cn format.
- Create a constraint:
 - At the same time as the table is created
 - After the table has been created
- Define a constraint at the column or table level.
- View a constraint in the data dictionary.

DEFINING CONSTRAINTS

- Column constraint level
 column [CONSTRAINT constraint_name] constraint_type
- Table constraint level
 [CONSTRAINT constraint_name] constraint_type(column)

```
CREATE TABLE table (column data type, column_constraint,
                     .... .... ,
                     ... .. ,
                     table_constraint);
```

- Q1)** Create table EMP1 with columns similar to EMP table and create NOT NULL (column) constraint for DEPTNO column and PRIMARY KEY (table) constraint for EMPNO column.

```
SQL> CREATE TABLE emp1( empno number(4),
    ename varchar2(10), job char(20), mgr number(10),
    hiredate date, sal number(5), comm number(5),
    deptno number(7,2) NOT NULL,
    CONSTRAINT emp1_pk PRIMARY KEY (empno));
```

NOT NULL Constraint

- Ensures that null values are not permitted for the column

CHECK Constraint

- Defines a condition that each row must satisfy

UNIQUE Constraint

- Prevent the duplication of values within the rows of a specified column

PRIMARY KEY Constraint

- Avoids duplication of rows and does not allow NULL values

FOREIGN KEY Constraint

- To establish a 'parent-child' or a 'master-detail' relationship between two tables having a common column, we make use of Foreign key (referential integrity) constraints.
- To do this we should define the column in the parent table as primary key and the same column in the child table as a foreign key referring to the corresponding parent entry.

FOREIGN KEY

- Defines the column in the child table at the table constraint level

REFERENCES

- Identifies the table and column in the parent table

ON DELETE CASCADE

- Allows deletion in the parent table and deletion of the dependent rows in the child table

ADDING A CONSTRAINT

- Add or drop, but not modify, a constraint
- Add a NOT NULL constraint by using the MODIFY clause

ALTER TABLE *table* ADD CONSTRAINT *const-name* cons-type (*column*);

- Q2)** Add NOT NULL constraint to the columns ENAME and JOB of EMP table.
**SQL> ALTER TABLE emp MODIFY(ename varchar2(20) NOT NULL,
job char(20) NOT NULL);**

- Q3)** Add Primary key constraint to the column EMPNO of EMP table
**SQL> ALTER TABLE emp ADD CONSTRAINT emp_pk
PRIMARY KEY(empno);**
- Q4)** Add Primary key constraint to the column DEPTNO of DEPT table
SQL>
- Q5)** Add Unique key constraint to the column DNAME of DEPT table
SQL>
- Q6)** Add Check constraint to the table EMP to restrict the values of EMPNO lies between 7000 and 8000.
**SQL> ALTER TABLE emp ADD CONSTRAINT emp_ck
CHECK(empno BETWEEN 7000 AND 8000)**
- Q7)** Add Foreign key constraint to the column DEPTNO of EMP table references DEPTNO of DEPT table.
**SQL> ALTER TABLE emp ADD CONSTRAINT emp_fk
FOREIGN KEY(deptno) REFERENCES DEPT(deptno);**
- Q8)** Add a Foreign key constraint to the EMP1 table indicating that a manager must already exist as a valid employee in the EMP1 table.
SQL>

DROPPING CONSTRAINTS

- Removing constraints from the table

ALTER TABLE *table* DROP CONSTRAINT *const-name*;

- Q9)** Remove the Manager constraint (added in Q8) from EMP table
SQL>
- Q10)** Remove the primary key constraint on the DEPT table and drop the associated foreign key constraint on the EMP.DEPTNO column.
SQL> ALTER TABLE dept DROP PRIMARY KEY CASCADE;

DISABLE and ENABLE Constraint

- Execute the DISABLE clause of the ALTER TABLE statement to deactivate an integrity constraint.
- Apply the CASCADE option to disable dependent integrity constraints.
- Activate an integrity constraint currently disabled in the table definition by using the ENABLE clause.
A UNIQUE or PRIMARY KEY index is automatically created if you enable a UNIQUE key or PRIMARY KEY constraint.

Q11) Disable the primary key constraint of EMP table.

SQL> ALTER TABLE emp DISABLE CONSTRAINT emp_pk CASCADE;

Q12) Enable the primary key constraint of EMP table.

SQL>

Q13) Query the USER_CONSTRAINTS table to view all constraint definitions and names

**SQL> SELECT constraint_name, constraint_type, search_condition
FROM user_constraints
WHERE table_name = 'EMP';**

Q14) View the columns associated with the constraint names in the USER_CONS_COLUMNS view

**SQL> SELECT constraint_name, column_name
FROM user_cons_columns
WHERE table_name = 'EMP';**

Verified by

Staff In-charge Sign :	Date :
-------------------------------	---------------

Inbuilt Functions in SQL

Ex. No : 4

Character Functions

It calculates the ASCII equivalent of the first character of the given

input string. ASCII(<Character>)

ascii('A')	would return 65
------------	--------------------

ascii('a')	would return 97
------------	--------------------

ascii('a8')	would re- turn 97
-------------	----------------------

CHR(<Character>)

Returns the character equivalent of the given integer. Example

```
SELECT CHR(65), CHR(97) FROM dual;
```

O/P A a

CONCAT(<string1>,<string2>)

This function returns String2 appended to

String1. Example:

```
SELECT CONCAT('Fname', 'Lname') Emp_name FROM emp;
```

INITCAP(<String>)

This function returns String with the first character of each word in upper case and rest of all in lower case.

Example:

```
SELECT INITCAP('oracle tutorial')
```

```
FROM Dual;
```

O/p Oracle Tutorial

INSTR

instr(string1, string2 [, start_position [, nth_Appearance]]):

where,

1. string1 is the string to search.
2. string2 is the substring to search for in string1.
3. start_position is the position in string1 from where the search will start.

This argument is optional. If not mentioned, it defaults to 1.

The first position in the string is 1. If the start_position is negative, the function counts back-ward direction.

4. nth_appearance is the nth appearance of string2. This is optional. If not defined, it defaults to Example

```
SELECT INSTR('Character','r',1,1) POS1, INSTR('Character','r',1,2) POS2,
INSTR('Character','a',-1,2) POS3,INSTR('character','c',) POS4
```

```
FROM
Dual;
```

pos1	pos2	pos3	pos4
4	9	3	6

LENGTH(<Str>)

Returns length of a string

```
select length('Sql Tutorial') as len
from dual;
O/p len
12
```

LOWER(<Str>)

This function returns a character string with all characters in lower case.

UPPER(<Str>)

This function returns a character string with all characters in upper case.

LPAD(<Str1>,<i>[,<Str2>])

This function returns the character string Str1 expanded in length to i characters, using Str2 to fill in space as needed on the left side of Str1.

Example

```
SELECT LPAD('Oracle',10,'.') lapd_doted from Dual, would return Oracle
SELECT LPAD('RAM', 7) lapd_exa from Dual would return ' RAM'
```

RPAD(<Str1>,<i>[,<Str2>])

RPAD is same as LPAD but Str2 is padded at the right side

LTRIM(<Str1>[,<Str2>])

The LTRIM function removes characters from the left side of the character String, with all the leftmost characters that appear in another text expression removed.

This function returns Str1 without any leading character that appears in Str2. If Str2 characters are leading character in Str1, then Str1 is returned unchanged. Str2 defaults to a single space.

Example

Select

LTRIM('datawarehousing','ing')


trim1 , LTRIM('datawarehousing ')

trim2

, LTRIM(' datawarehousing') trim3

, LTRIM('datawarehousing','data')

trim4 from dual

trim1	trim2	trim3	trim4
			
datawarehousing	datawarehousing	datawarehousing	warehousing

RTRIM(<Str1>[,<Str2>])

Same as LTRIM but the characters are trimmed from the right side

TRIM([(<Str1>|<Str2> FROM)<Str3>])

If present Str1 can be one of the following literal: LEADING, TRAILING, BOTH.

This function returns Str3 with all C1(leading trailing or both) occurrences of characters in Str2 removed.

If any of Str1, Str2 or Str3 is Null, this function returns a Null.

Str1 defaults to BOTH, and Str2 defaults to a space character.

Example

```
SELECT TRIM(' Oracle ') trim1, TRIM('Oracle ') trim2 FROM Dual;
```

Ans trim1 trim2

Oracle Oracle

It'll remove the space from both
string.

REPLACE(<Str1>,<Str2>[,<Str3>]

r 3>]

This function returns Str1 with all occurrence of Str2 replaced with Str3

Example

```
SELECT REPLACE (,'Oracle', 'Ora', 'Arti') replace_exa  
FROM Dual;
```

O/p replace_exa

Article

Essential Numeric

Functions

ABS()

Select Absolute value

```
SELECT ABS(-25) "Abs" FROM DUAL;
```

Abs

15 _____

ACOS ()

Select cos value

```
SELECT ACOS(.28)"Arc_Cosine" FROM DUAL;
```

ASIN ()

Select sin value

```
SELECT ASIN(.6)"Arc_Cosine" FROM DUAL;
```

ATAN()

Select tan value

```
SELECT ATAN(.6)"Arc_Cosine" FROM DUAL;
```

CEIL()

Returns the smallest integer greater than or equal to the order total of a specified SELECT CEIL(239.8) FROM Dual would return 240

FLOOR()

Returns the largest integer equal to or less than value.

```
SELECT FLOOR(15.65) "Floor" FROM DUAL;  
Floor
```

15

MOD()

Return modulus value

```
SELECT MOD(11,3) "Mod" FROM DUAL;
```

Modulus

2

POWER()

```
SELECT POWER(3,2) "Power" FROM DUAL;  
power
```

9

ROUND (number)

```
SELECT ROUND(43.698,1) "Round" FROM DUAL;  
Round
```

43.7

TRUNC (number)

The TRUNC (number) function returns n1 truncated to n2 decimal places. If n2 is omitted, then n1 is truncated to 0 places. n2 can be negative to truncate (make zero) n2 digits left of the decimal point.

```
SELECT TRUNC(12.75,1) "Trunc" FROM DUAL;
```

Trunc

12.75

```
SELECT TRUNC(12.75,-1) "Trunc" FROM DUAL;
```

Trunc

10

Date And Time Function

ADD_MONTHS(date,number_of_month)

```
SELECT SYSDATE, ADD_MONTHS(SYSDATE,2), ADD_MONTHS(SYSDATE,-2) FROM
```

DUAL;

Result:

SYSDATE

ADD MONTH

10-Feb-13 10-Apr-13 10-Dec-13

EXTRACT(<type> FROM <date>)

'Type' can be YEAR, MONTH, DAY, HOUR, MIN, SECOND, TIME_ZONE_HOUR, TIME_ZONE_MINUTE, TIME_ZONE_REGION

```
SELECT SYSDATE, EXTRACT(YEAR FROM SYSDATE)YEAR, EXTRACT(DAY FROM SYSDATE)DAY , EXTRACT(TIMEZONE_HOUR FROM SYSTIMESTAMP) TZH FROM DUAL;
```

LAST_DAY(<date>)

Extract last day of month Example:

```
SELECT SYSDATE, LAST_DAY(SYSDATE) END_OF_MONTH FROM DUAL;
```

Result: SYS-
DATE END_OF_MO

DATE	END_OF_MO
4-Aug-18	31-Aug-18

NEXT_DAY(<date>,<day>)

SELECT NEXT_DAY('31-Aug-18','SUN') "FIRST MONDAY OF
SEPTEMBER" FROM DUAL;

O/P FIRST MONDAY OF SEPTEMBER

03-Sep-18

ROUND (date[,<fmt>])

SELECT SYSDATE, ROUND(SYSDATE,'MM'),
ROUND(SYSDATE,'YYYY') FROM DUAL;

Result:

SYSDATE	ROUND(SYSDATE,'MM')	ROUND(SYSDATE,'YYYY')
10-FEB-18	01-MAR-18	01-JAN-18

TRUNC(date[,<fmt>])

SELECT SYSDATE, TRUNC(SYSDATE,'MM'), TRUNC(SYSDATE,'YYYY')
FROM DUAL;

Result:	SYSDATE	TRUNC(SYSDATE,'MM')	TRUNC(SYSDATE,'YYYY')
	10-FEB-18	01-FEB-18	01-JAN-18

MONTHS_BETWEEN function returns the number of months between

date1 and date2. SYNTAX

The syntax for the Oracle/PLSQL MONTHS_BETWEEN function is:

MONTHS_BETWEEN(date1, date2)

Parameters or Arguments

date1 and date2 are the dates used to calculate the number of months.

If a fractional month is calculated, the MONTHS_BETWEEN function calculates the fraction based on a 31-day month.

Ex. No: 3

SQL DCL & TCL COMMANDS

Date:

AIM:

To write SQL queries to execute different DCL and TCL commands.

Data base created for this exercise is:

customer_id integer	sale_date date	sale_amount numeric	salesperson character varying (255)	store_state character varying (255)	order_id character varying (255)
1001	2020-05-23	1200	Raj K	KA	1001
1001	2020-05-22	1200	M K	NULL	1002
1002	2020-05-23	1200	Malika Rakesh	MH	1003
1003	2020-05-22	1500	Malika Rakesh	MH	1004
1004	2020-05-22	1210	M K	NULL	1003
1005	2019-12-12	4200	R K Rakesh	MH	1007
1002	2020-05-21	1200	Molly Samberg	DL	1001

Data Control Language (DCL) Commands:

DCL includes commands such as GRANT and REVOKE which mainly deal with the rights, permissions, and other controls of the database system.

List of DCL commands:

- **GRANT:** This command gives users access privileges to the database.

Syntax,

```
GRANT privileges_names ON object TO user;
```

Example:

Create user first identified by passwd;

Grant select on customers to first;

- **REVOKE:** This command withdraws the user's access privileges given by using the GRANT command.

Syntax,

```
REVOKE privileges ON object FROM user;
```

Example:

Revoke select on customers from first;

Transaction Control Language (TCL) Commands:

- **COMMIT**: Commits a Transaction.

Syntax:

```
COMMIT;
```

Example:

```
INSERT INTO customers  
VALUES ('1006','2020-03-04',3200,'DL', '1008');
```

Commit;

Select * from customers;

- **ROLLBACK**: Rollbacks a transaction in case of any error occurs.

Syntax:

```
Rollback;
```

Example:

```
DELETE FROM customers  
WHERE store_state = 'MH'  
AND customer_id = '1002';
```

Select * from customers;

Rollback;

Select * from customers;

- **SAVEPOINT**: Sets a savepoint within a transaction.

Syntax:

```
SAVEPOINT SAVEPOINT_NAME;
```

This command is used only in the creation of SAVEPOINT among all the transactions.

In general ROLLBACK is used to undo a group of transactions.

Syntax for rolling back to Savepoint command:

```
ROLLBACK TO SAVEPOINT_NAME;
```

Example:

```
SAVEPOINT SP1;  
DELETE FROM customers  
WHERE store_state = 'MH'  
AND customer_id = '1002';  
SAVEPOINT SP2;  
ROLLBACK TO SP1;  
Select * from customers;
```

Result:

Thus the DCL and TCL commands are used to modify or manipulate data records present in the customer database tables.

Ex. No: 2**SQL DML COMMANDS****Date:****AIM:**

To write SQL queries to execute different DML commands.

Data base created for this exercise is:

customer_id integer	sale_date date	sale_amount numeric	salesperson character varying (255)	store_state character varying (255)	order_id character varying (255)
1001	2020-05-23	1200	Raj K	KA	1001
1001	2020-05-22	1200	M K	NULL	1002
1002	2020-05-23	1200	Malika Rakesh	MH	1003
1003	2020-05-22	1500	Malika Rakesh	MH	1004
1004	2020-05-22	1210	M K	NULL	1003
1005	2019-12-12	4200	R K Rakesh	MH	1007
1002	2020-05-21	1200	Molly Samberg	DL	1001

DML Commands:

- **SELECT** - Used to query or fetch selected fields or columns from a database table

Syntax,

SELECT column_name1, column_name2, ...

FROM table_name

WHERE condition_expression;

Example:

Select customer_id, sale_date, order_id, store_state from customers;

Select * from customers;

- **INSERT** - Used to insert new data records or rows in the database table

Syntax,

INSERT INTO table_name (column_name_1, column_name_2, column_name_3, ...)

VALUES (value1, value2, value3, ...)

Example:

```
INSERT INTO customers(  
customer_id, sale_date, sale_amount, salesperson, store_state, order_id)  
VALUES (1005,'2019-12-12',4200,'R K Rakesh','MH','1007');
```

(or)

```
INSERT INTO customers  
VALUES ('1006','2020-03-04',3200,'DL', '1008');
```

- **UPDATE** - Used to set the value of a field or column for a particular record to a new value

Syntax,

UPDATE table_name

SET column_name_1 = value1, column_name_2 = value2, ...

WHERE condition;

Example,

```
UPDATE customers  
SET store_state = 'DL'  
WHERE store_state = 'NY';
```

- **DELETE** - Used to remove one or more rows from the database table

Syntax,

DELETE FROM table_name WHERE condition;

Example,

```
DELETE FROM customers  
WHERE store_state = 'MH'  
AND customer_id = '1001';
```

Result:

Thus the DML commands are used to modify or manipulate data records present in the customer database tables.

Ex. No: 1

SQL BASIC COMMANDS

Date:

AIM:

To write SQL queries to execute basic SQL commands.

QUERIES:

1. Create table

Query:

```
CREATE TABLE emp
(
    empno NUMBER,
    empname VARCHAR2(255),
    DOB DATE,
    salary NUMBER,
    designation VARCHAR2(20)
);
```

Output:

Table created.

2. Insert values

Query:

```
INSERT INTO emp VALUES(100,'John','4.21.1994', 50000,'Manager');
```

```
INSERT INTO emp VALUES(101,'Greg','6.20.1994',25000,'Clerk');
```

Output:

2 rows inserted

3. Display values

Query:

```
SELECT * FROM emp;
```

Output:

EMPNO	EMPNAME	DOB	SALARY	DESIGNATION
100	John	04/21/1994	50000	Manager
101	Greg	06/20/1994	25000	Clerk

Query:

```
SELECT empname,salary FROM emp;
```

Output:

EMPNAME	SALARY
John	50000
Greg	25000

4. Modify values

Query:

```
UPDATE emp SET salary = salary + 1000;
```

Output:

2 row(s) updated.

Query:

```
SELECT * FROM emp;
```

Output:

EMPNO	EMPNAME	DOB	SALARY	DESIGNATION
100	John	04/21/1994	51000	Manager
101	Greg	06/20/1994	26000	Clerk

5. Delete values

Query:

```
DELETE FROM emp WHERE empno = 100;
```

Output:

1 row(s) deleted.

Query:

```
SELECT * FROM emp;
```

Output:

EMPNO	EMPNAME	DOB	SALARY	DESIGNATION
101	Greg	06/20/1994	26000	Clerk

RESULT:

Thus the basic SQL queries were successfully executed and verified.

Ex. No: 2 DATA DEFINITION LANGUAGE (DDL)

Date :

AIM:

To write the SQL queries using DDL Commands with and without constraints.

DDL STATEMENTS

- CREATE TABLE
- ALTER TABLE
- DROP TABLE

SYNTAX:

1. Create Table

The CREATE TABLE statement is used to create a relational table

```
CREATE TABLE table_name
(
    column_name1 data_type [constraints],
    column_name1 data_type [constraints],
    column_n
ame1 data_type [constraints],
.....
```

);

2. Alter Table

The ALTER TABLE statement is used to add, delete, or modify columns in an existing table

a. To Add a column

```
ALTER TABLE table_name ADD column_name datatype
```

b. To delete a column in a table

```
ALTER TABLE table_name DROP (column_name)
```

c. To change the data type of a column in a table

```
ALTER TABLE table_name MODIFY(column_name datatype )
```

3. Drop Table

Used to delete the table permanently from the storage

```
DROP TABLE table_name
```

QUERIES:

1. CREATE THE TABLE (with no constraint)

Query:

```
CREATE TABLE emp
(
    empno NUMBER,
    empname VARCHAR2(25),
    dob DATE,
    salary NUMBER,
    designation VARCHAR2(20)
);
```

Output:

Table Created

Query:

```
DESC emp;
```

Output:

Table	Column	Data Type	Length	Precision	Scale	Primary Key	Nullable	Default	Comment
<u>EMP</u>	<u>EMPNO</u>	NUMBER	22	-	-	-	-	-	-
	<u>EMPNAME</u>	VARCHAR2	255	-	-	-	-	-	-
	<u>DOB</u>	DATE	7	-	-	-	-	-	-
	<u>SALARY</u>	NUMBER	22	-	-	-	-	-	-
	<u>DESIGNATION</u>	VARCHAR2	20	-	-	-	-	-	-

2. ALTER THE TABLE

a. ADD

// To alter the table emp by adding new attribute department

Query:

```
ALTER TABLE emp ADD department VARCHAR2(50);
```

Output:

Table Altered

Query:

DESC emp;

Output:

Table	Column	Data Type	Length	Precision	Scale	Primary Key	Nullable	Default	Comment
<u>EMP</u>	<u>EMPNO</u>	NUMBER	22	-	-	-		-	-
	<u>EMPNAME</u>	VARCHAR2	255	-	-	-		-	-
	<u>DOB</u>	DATE	7	-	-	-		-	-
	<u>SALARY</u>	NUMBER	22	-	-	-		-	-
	<u>DESIGNATION</u>	VARCHAR2	20	-	-	-		-	-
	<u>DEPARTMENT</u>	VARCHAR2	50	-	-	-		-	-

b. MODIFY

//To alter the table emp by modifying the size of the attribute department

Query:

ALTER TABLE emp MODIFY (department VARCHAR2(100));

Output:

Table Altered

Query:

DESC emp;

Output:

Table	Column	Data Type	Length	Precision	Scale	Primary Key	Nullable	Default	Comment
<u>EMP</u>	<u>EMPNO</u>	NUMBER	22	-	-	-		-	-

<u>EMPNAME</u>	VARCHAR2	255	-	-	-	-	-
<u>DOB</u>	DATE	7	-	-	-	-	-
<u>SALARY</u>	NUMBER	22	-	-	-	-	-
<u>DESIGNATION</u>	VARCHAR2	20	-	-	-	-	-
<u>DEPARTMENT</u>	VARCHAR2	100	-	-	-	-	-

c. DROP

// To alter the table emp by deleting the attribute department

Query:

```
ALTER TABLE emp DROP(department);
```

Output:

Table Altered

Query:

```
DESC emp;
```

Output:

Table	Column	Data Type	Length	Precision	Scale	Primary Key	Nullable	Default	Comment
<u>EMP</u>	<u>EMPNO</u>	NUMBER	22	-	-	-	-	-	-
	<u>EMPNAME</u>	VARCHAR2	255	-	-	-	-	-	-
	<u>DOB</u>	DATE	7	-	-	-	-	-	-
	<u>SALARY</u>	NUMBER	22	-	-	-	-	-	-
	<u>DESIGNATION</u>	VARCHAR2	20	-	-	-	-	-	-

d. RENAME

// To alter the table name by using rename keyword

Query:

```
ALTER TABLE emp RENAME TO emp1 ;
```

Output:

Table Altered

Query:

DESC emp1;

Output:

Table	Column	Data Type	Length	Precision	Scale	Primary Key	Nullable	Default	Comment
<u>EMP1</u>	<u>EMPNO</u>	NUMBER	22	-	-	-		-	-
	<u>EMPNAME</u>	VARCHAR2	255	-	-	-		-	-
	<u>DOB</u>	DATE	7	-	-	-		-	-
	<u>SALARY</u>	NUMBER	22	-	-	-		-	-
	<u>DESIGNATION</u>	VARCHAR2	20	-	-	-		-	-
	<u>DEPARTMENT</u>	VARCHAR2	100	-	-	-		-	-

3. DROP

//To delete the table from the database

Query:

DROP TABLE emp1;

Output:

Table Dropped

Query:

DESC emp1;

Output:

Object to be described could not be found.

CONSTRAINT TYPES:

- NOT NULL
- UNIQUE
- PRIMARY KEY
- FOREIGN KEY
- CHECK

- DEFAULT

QUERIES:

1. CREATE THE TABLE

Query:

```
CREATE TABLE student
(
    studentID NUMBER PRIMARY KEY,
    sname VARCHAR2(30) NOT NULL,
    department CHAR(5),
    sem NUMBER,
    dob DATE,
    email_id VARCHAR2(20) UNIQUE,
    college VARCHAR2(20) DEFAULT 'MEC'
);
```

Output:

Table created.

Query:

```
DESC student;
```

Output:

Table	Column	Data Type	Length	Precision	Scale	Primary Key	Nullable	Default	Comment
<u>STUDENT</u>	<u>STUDENTID</u>	NUMBER	22	-	-	1	-	-	-
	<u>SNAME</u>	VARCHAR2	30	-	-	-	-	-	-
	<u>DEPARTMENT</u>	CHAR	5	-	-	-	✓	-	-
	<u>SEM</u>	NUMBER	22	-	-	-	✓	-	-
	<u>DOB</u>	DATE	7	-	-	-	✓	-	-
	<u>EMAIL_ID</u>	VARCHAR2	20	-	-	-	✓	-	-
	<u>COLLEGE</u>	VARCHAR2	20	-	-	-	✓	'MEC'	-

Query:

```

CREATE TABLE exam
(
    examID NUMBER ,
    studentID NUMBER REFERENCES student(studentID),
    department CHAR(5) NOT NULL,
    mark1 NUMBER CHECK (mark1<=100 and mark1>=0),
    mark2 NUMBER CHECK (mark2<=100 and mark2>=0),
    mark3 NUMBER CHECK (mark3<=100 and mark3>=0),
    mark4 NUMBER CHECK (mark4<=100 and mark4>=0),
    mark5 NUMBER CHECK (mark5<=100 and mark5>=0),
    total NUMBER,
    average NUMBER,
    grade CHAR(1)
);

```

Output:

Table created.

//To alter the table student by adding new constraint to the examID attribute

Query:

```

ALTER TABLE student ADD CONSTRAINT pr
                                PRIMARY KEY (examid);

```

Output:

Table altered.

2. CREATE THE TABLE USING COMPOSITE PRIMARY KEY

Create the following table with the attributes reg_no and stu_name as primary key.

stu_details (reg_no, stu_name, DOB, address, city)

Query:

```

CREATE TABLE stu_details
(

```

```

reg_no number,
stu_name varchar2(30),
DOB date,
address varchar2(30),
city char(30),
primary key(reg_no, stu_name)
);

```

Output:

Table created.

Query:

DESCstu_details

Output:

Table	Column	Data Type	Length	Precision	Scale	Primary Key	Nullable	Default	Comment
<u>STU_DETAILS</u>	<u>REG_NO</u>	NUMBER	22	-	-	1	-	-	-
	<u>STU_NAME</u>	VARCHAR2	30	-	-	2	-	-	-
	<u>DOB</u>	DATE	7	-	-	-	✓	-	-
	<u>ADDRESS</u>	VARCHAR2	30	-	-	-	✓	-	-
	<u>CITY</u>	CHAR	30	-	-	-	✓	-	-

RESULT:

Thus the SQL queries using DDL Commands with and without constraints were successfully executed and verified.