R-2019

DATABASE MANAGEMENT SYSTEMS LAB

COLLEGE OF ENGINEERING
(AUTONOMOUS)

Gayatri Vidya Parishad College of Engineering

(Autonomous)

Approved by AICTE, Affiliated to JNTUK, Kakinada
Accredited by NBA & NAAC with "A" Grade with a CGPA of 3.47/4.00
Madhurawada, Visakhapatnam-530048

Laboratory Manual

For

Database Management Systems Lab

(Course code: 19CT1105)

Academic Year 2020-2021

B.Tech. I.T III Semester

Department of Information Technology

VISION OF THE DEPARTMENT

- To become a leading Center of Learning and Research in the field of Information Technology.

.

MISSION OF THE DEPARTMENT

- To produce high quality technocrats with original thinking and self-reliance in the application of Information Technology.
- To foster collaborative research and development activities in thrust areas of Information Technology.
- To serve the society with high standards of professional ethics, values and accountability.

PROGRAM OUTCOMES

PO1:	Apply the knowledge of mathematics, science, engineering fundamentals and principles of Information Technology to solve problems in different domains.
PO2:	Analyze a problem, identify and formulate the computing requirements appropriate to its solution.
PO3:	Understand to design, develop and evaluate software components and applications that meet specifications within the realistic constraints including cultural, societal and environmental considerations.
PO4:	Design and conduct experiments, as well as analyze and interpret data
PO5:	Use appropriate techniques and tools to solve domain specific interdisciplinary problems.
PO6:	Understand the impact of Information technology on environment and the evolution and importance of green computing.
PO7:	Analyze the local and global impact of computing on individual as well as on society and incorporate the results in to engineering practice.
PO8:	Demonstrate professional ethical practices and social responsibilities in global and societal contexts.
PO9:	Function effectively as an individual, and as a member or leader in diverse and multidisciplinary teams.
PO10:	Communicate effectively with the engineering community and with society at large.
PO11:	Understand engineering and management principles and apply these to one's own work, as a member and leader in a team, to manage projects.
PO12:	Recognize the need for updating the knowledge in the chosen field and imbibing learning to learn skills.

COURSE OUTCOMES (COs)

CO1	Apply data definitions and data manipulation commands.
CO2	Apply nested queries and subqueries.
CO3	Demonstrate database applications with joins and views.
CO4	Use functions, procedures and procedural extensions of databases.
CO5	Use Cursors, Triggers and Exception Handling mechanism.

CO-PO Mapping

COs	PO1	PO2	PO3	PO4	PO5	PO6	PO7	PO8	PO9	PO10	PO11	PO12
CO-1	M	M	M	M	M							
CO-2	M	M	M	M	M							
CO-3	M	M	M	M	M							
CO-4	M	S	M	M	M							
CO-5	M	S	M	M	M							

DATABASE MANAGEMENT SYSTEMS LAB

COURSE CODE: 19CT1105 L T P C

0 0 3 2

Aim and Objective: The main objective of this lab is to gain the practical hands on experience by exposing the students to various SQL Commands. The students will have an understanding of the concepts involved in maintaining the database.

List of Experiments:

1. Data Definition Commands, Data Manipulation Commands for inserting and deleting of data from Tables.
2. Data Manipulation Commands for updating and retrieving of data from Tables and Transaction Control statements
3. Basic functions like Numeric, String, Date and conversion functions.
4. Database Querying – Simple queries.
5. Queries using aggregate functions, GROUP BY and HAVING clauses.
6. Database Querying – Nested queries, Sub-queries.
7. Queries using Joins
8. Queries using Views

PROGRAMS USING PL/SQL:

9. Procedures and Functions.
10. Implicit and Explicit Cursors
11. Triggers
12. Exception Handling

CASE STUDIES: ***Students shall form in groups at the beginning of the semester and perform the following steps by the end of the semester and submit a project.***

13. Design a Database for any real life application using ER model and normalize it.
14. Connect the Database through any programming language
15. Build real life database applications.

TEXT BOOK:

1. Elmasri Navathe, *Fundamentals of Database Systems*, 7th Edition, Pearson Education, 2017.

REFERENCES:

1. C.J.Date, *Introduction to Database Systems*, 7th Edition, Pearson Education, 2002.
2. Peter Rob & Carlos Coronel, *Database Systems design, Implementation, and Management*, 7th Edition, Pearson Education, 2000.

EXPERIMENT 1

AIM: Data Definition Commands, Data Manipulation Commands for inserting and deleting of data from Tables.

Introduction :

DDL is abbreviation of Data Definition Language. It is used to create and modify the structure of database objects in database. DML is Data Manipulation Language which is used to manipulate data itself. For example: insert, update, delete are instructions in SQL. DCL is abbreviation of Data Control Language. It is used to create roles, permissions, and referential integrity as well it is used to control access to database by securing it.

What is Data Definition Language(DDL)

Data Definition Language helps you to define the database structure or schema. Let's learn about DDL commands with syntax.

Five types of DDL commands in SQL are:

CREATE

CREATE statements is used to define the database structure schema:

Syntax:

CREATE TABLE TABLE_NAME (COLUMN_NAME DATATYPES[,....]);

For example:

Create database university;

Create table students;

Create view for_students;

DROP

Drops commands remove tables and databases from RDBMS.

Syntax

DROP TABLE ;

For example:

Drop object_type object_name;

Drop database university;

Drop table student;

ALTER

Alters command allows you to alter the structure of the database.

Syntax:

To add a new column in the table

ALTER TABLE table_name ADD column_name COLUMN-definition;

To modify an existing column in the table:

ALTER TABLE MODIFY(COLUMN DEFINITION....);

For example:

Alter table guru99 add subject varchar;

TRUNCATE:

This command used to delete all the rows from the table and free the space containing the table.

Syntax:

TRUNCATE TABLE table_name;

Example:

TRUNCATE table students;

What is Data Manipulation Language(DML)

Data Manipulation Language (DML) allows you to modify the database instance by inserting, modifying, and deleting its data. It is responsible for performing all types of data modification in a database.

There are three basic constructs which allow database program and user to enter data and information are:

Here are some important DML commands in SQL:

- INSERT
- UPDATE
- DELETE

INSERT:

This is a statement is a SQL query. This command is used to insert data into the row of a table.

Syntax:

INSERT INTO TABLE_NAME (col1, col2, col3,.... col N)

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

Or

INSERT INTO TABLE_NAME

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

For example:
INSERT INTO students (RollNo, FIrstName, LastName) VALUES ('60', 'Tom', Erichsen');

UPDATE:
This command is used to update or modify the value of a column in the table.
Syntax:
UPDATE table_name SET [column_name1= value1,...column_nameN = valueN] [WHERE CONDITION]

For example:
UPDATE students

SET FirstName = 'Jhon', LastName= 'Wick'

WHERE StudID = 3;

DELETE:
This command is used to remove one or more rows from a table.
Syntax:
DELETE FROM table_name [WHERE condition];

For example:
DELETE FROM students

WHERE FirstName = 'Jhon';

What is Data Control Language(DCL)
DCL (Data Control Language) includes commands like GRANT and REVOKE, which are useful to give "rights & permissions." Other permission controls parameters of the database system.

Examples of DCL commands:
Commands that come under DCL:

- Grant
- Revoke

Grant:

This command is use to give user access privileges to a database.

Syntax:

GRANT SELECT, UPDATE ON MY_TABLE TO SOME_USER, ANOTHER_USER;

For example:

GRANT SELECT ON Users TO'Tom'@'localhost;

Revoke:

It is useful to back permissions from the user.

Syntax:

REVOKE privilege_nameON object_nameFROM {user_name |PUBLIC |role_name}

For example:

REVOKE SELECT, UPDATE ON student FROM BCA, MCA;

What is TCL?

Transaction control language or TCL commands deal with the transaction within the database.

Commit

This command is used to save all the transactions to the database.

Syntax:

Commit;

For example:

DELETE FROM Students

WHERE RollNo =25;

COMMIT;

Rollback

Rollback command allows you to undo transactions that have not already been saved to the database.

Syntax:
ROLLBACK;

Example:
DELETE FROM Students

WHERE RollNo =25;

SAVEPOINT
This command helps you to sets a savepoint within a transaction.
Syntax:
SAVEPOINT SAVEPOINT_NAME;

Example:
SAVEPOINT RollNo;

What is DQL?
Data Query Language (DQL) is used to fetch the data from the database. It uses only one command:

SELECT:
This command helps you to select the attribute based on the condition described by the WHERE clause.
Syntax:
SELECT expressions

FROM TABLES

WHERE conditions;

For example:
SELECT FirstName

FROM Student

WHERE RollNo > 15;

EXPERIMENT 2

AIM: Data Manipulation Commands for updating and retrieving of data from Tables and Transaction Control statements

Introduction

What is Data Manipulation Language(DML)

Data Manipulation Language (DML) allows you to modify the database instance by inserting, modifying, and deleting its data. It is responsible for performing all types of data modification in a database.

There are three basic constructs which allow database program and user to enter data and information are:

Here are some important DML commands in SQL:

- INSERT
- UPDATE
- DELETE

Data manipulation commands are used to manipulate data in the database.

Some of the Data Manipulation Commands are-

Select

Select statement retrieves the data from database according to the constraints specifies alongside.

SELECT <COLUMN NAME>

FROM <TABLE NAME>

WHERE <CONDITION>

GROUP BY <COLUMN LIST>

HAVING <CRITERIA FOR FUNCTION RESULTS>

ORDER BY <COLUMN LIST>

General syntax −

Example: select * from employee where e_id>100;

Insert

Insert statement is used to insert data into database tables.

General Syntax −

INSERT INTO <TABLE NAME> (<COLUMNS TO INSERT>) VALUES (<VALUES TO INSERT>)

Example: insert into Employee (name, dept_id) values ('ABC', 3);

Update

The update command updates existing data within a table.

General syntax

UPDATE <TABLE NAME>

SET <COLUMN NAME> = <UPDATED COLUMN VALUE>,

<COLUMN NAME> = <UPDATED COLUMN VALUE>,

<COLUMN NAME> = <UPDATED COLUMN VALUE>,…

WHERE <CONDITION>

Example: update Employee set Name='AMIT' where E_id=5;

Delete

Deletes records from the database table according to the given constraints.

General Syntax

DELETE FROM <TABLE NAME>

WHERE <CONDITION>

Example −
delete from Employee where e_id=5;

To delete all records from the table −
Delete * from <TABLE NAME>;

Merge

Use the MERGE statement to select rows from one table for update or insertion into another table. The decision whether to update or insert into the target table is based on a condition in the ON clause. It is also known as UPSERT i.e. combination of UPDATE and INSERT.

General Syntax (SQL)

MERGE <TARGET TABLE> [AS TARGET]

USING <SOURCE TABLE> [AS SOURCE]

ON <SEARCH CONDITION>

[WHEN MATCHED

THEN <MERGE MATCHED >]

[WHEN NOT MATCHED [BY TARGET]

THEN < MERGE NOT MATCHED >]

[WHEN NOT MATCHED BY SOURCE

THEN <MERGE MATCHED >];

General Syntax (Oracle)

MERGE INTO <TARGET TABLE>

USING <SOURCE TABLE>

ON <SEARCH CONDITION>

[WHEN MATCHED

THEN <MERGE MATCHED >]

[WHEN NOT MATCHED

THEN < MERGE NOT MATCHED >];

EXPERIMENT 3

AIM: . Basic Functions Like Numeric, String, Date and Conversion Functions.

Introduction

Oracle inbuilt function library contains type conversion functions. There may be scenarios where the query expects input in a specific data type, but it receives it in a different data type. In such cases, Oracle implicitly tries to convert the unexpected value to a compatible data type which can be substituted in place and application continuity is not compromised. Type conversion can be either implicitly done by Oracle or explicitly done by the programmer.

Implicit data type conversion works based on a matrix which showcases the Oracle's support for internal type casting. Besides these rules, Oracle offers type conversion functions which can be used in the queries for explicit conversion and formatting. As a matter of fact, it is recommended to perform explicit conversion instead of relying on software intelligence. Though implicit conversion works well, but to eliminate the skew chances where bad inputs could be difficult to typecast internally.

Explicit Data Type Conversion:

SQL Conversion functions are single row functions which are capable of typecasting column value, literal or an expression . TO_CHAR, TO_NUMBER and TO_DATE are the three functions which perform cross modification of data types.

TO_CHAR function is used to typecast a numeric or date input to character type with a format model.

Syntax:

TO_CHAR(number1, [format], [nls_parameter])

For number to character conversion, nls parameters can be used to specify decimal characters, group separator, local currency model, or international currency model.

It is an optional specification - if not available, session level nls settings will be used. For date to character conversion, the nls parameter can be used to specify the day and month names, as applicable.

Dates can be formatted in multiple formats after converting to character types using TO_CHAR function. The TO_CHAR function is used to have Oracle 11g display dates in a particular format. Format models are case sensitive and must be enclosed within single quotes.

Consider the below SELECT query. The query format the HIRE_DATE and SALARY columns of EMPLOYEES table using TO_CHAR function.

```
SELECT first_name,
       TO_CHAR (hire_date, 'MONTH DD, YYYY') HIRE_DATE,
           TO_CHAR (salary, '$99999.99') Salary
FROM employees
WHERE rownum < 5;

FIRST_NAME           HIRE_DATE          SALARY
-------------------- ------------------ ----------
Steven               JUNE      17, 2003  $24000.00
Neena                SEPTEMBER 21, 2005  $17000.00
Lex                  JANUARY   13, 2001  $17000.00
Alexander            JANUARY   03, 2006   $9000.00
```

The first TO_CHAR is used to convert the hire date to the date format MONTH DD, YYYY i.e. month spelled out and padded with spaces, followed by the two-digit day of the month, and then the four-digit year. If you prefer displaying the month name in mixed case (that is, "December"), simply use this case in the format argument: ('Month DD, YYYY').

TO_NUMBER function:

The TO_NUMBER function converts a character value to a numeric datatype. If the string being converted contains nonnumeric characters, the function returns an error.

Syntax

TO_NUMBER (string1, [format], [nls_parameter])

Format Model	Description
CC	Century
SCC	Century BC prefixed with -
YYYY	Year with 4 numbers
SYYY	Year BC prefixed with -
IYYY	ISO Year with 4 numbers
YY	Year with 2 numbers
RR	Year with 2 numbers with Y2k compatibility
YEAR	Year in characters
SYEAR	Year in characters, BC prefixed with -
BC	BC/AD Indicator
Q	Quarter in numbers (1,2,3,4)
MM	Month of year 01, 02...12
MONTH	Month in characters (i.e. January)
MON	JAN, FEB
WW	Week number (i.e. 1)
W	Week number of the month (i.e. 5)

The SELECT queries below accept numbers as character inputs and prints them following the format specifier.

```
SELECT  TO_NUMBER('121.23', '9G999D99')
FROM DUAL

TO_NUMBER('121.23','9G999D99')
------------------------------
                        121.23

SELECT  TO_NUMBER('1210.73', '9999.99')
FROM DUAL;

TO_NUMBER('1210.73','9999.99')
------------------------------
                       1210.73
```

TO_DATE function:

The function takes character values as input and returns formatted date equivalent of the same. The TO_DATE function allows users to enter a date in any format, and then it converts the entry into the default format used by Oracle 11g.

Syntax:

TO_DATE(string1, [format_mask], [nls_language])

A format_mask argument consists of a series of elements representing exactly what the data should look like and must be entered in single quotation

marks.

Format Model	Description
YEAR	Year, spelled out
YYYY	4-digit year
YYY,YY,Y	Last 3, 2, or 1 digit(s) of year.
IYY,IY,I	Last 3, 2, or 1 digit(s) of ISO year.
IYYY	4-digit year based on the ISO standard
RRRR	Accepts a 2-digit year and returns a 4-digit year.
Q	Quarter of year (1, 2, 3, 4; JAN-MAR = 1).
MM	Month (01-12; JAN = 01).
MON	Abbreviated name of month.
MONTH	Name of month, padded with blanks to length of 9 characters.
RM	Roman numeral month (I-XII; JAN = I).
WW	Week of year (1-53) where week 1 starts on the first day of the year and continues to the seventh day of the year.
W	Week of month (1-5) where week 1 starts on the first day of the month and ends on the seventh.
IW	Week of year (1-52 or 1-53) based on the ISO standard.
D	Day of week (1-7).

CONCAT(str1,str2,...)

Returns the string that results from concatenating the arguments. May have one or more arguments. If all arguments are non-binary strings, the result is a non-binary string. If the arguments include any binary strings, the result is a binary string. A numeric argument is converted to its equivalent binary string form

```
SQL> SELECT CONCAT('My', 'S', 'QL');
+-------------------------------------------------------+
| CONCAT('My', 'S', 'QL')                               |
+-------------------------------------------------------+
| MySQL                                                 |
+-------------------------------------------------------+
1 row in set (0.00 sec)
```

LPAD(str,len,padstr)

Returns the string str, left-padded with the string padstr to a length of len characters. If str is longer than len, the return value is shortened to len characters.

```
SQL> SELECT LPAD('hi',4,'??');
+-------------------------------------------------------+
| LPAD('hi',4,'??')                                     |
+-------------------------------------------------------+
| ??hi                                                  |
+-------------------------------------------------------+
1 row in set (0.00 sec)
```

RPAD(str,len,padstr)

Returns the string str, right-padded with the string padstr to a length of len characters. If str is longer than len, the return value is shortened to len characters.

```
SQL> SELECT RPAD('hi',5,'?');
+-------------------------------------------------------+
| RPAD('hi',5,'?')                                      |
+-------------------------------------------------------+
| hi???                                                 |
+-------------------------------------------------------+
1 row in set (0.00 sec)
```

LTRIM(str)

Returns the string str with leading space characters removed

```
SQL> SELECT LTRIM('  barbar');
+-------------------------------------------------------+
| LTRIM('  barbar')                                     |
+-------------------------------------------------------+
| barbar                                                |
+-------------------------------------------------------+
1 row in set (0.00 sec)
```

RTRIM(str)

Returns the string str with trailing space characters removed.

```
SQL> SELECT RTRIM('barbar   ');
+-------------------------------------------------------+
| RTRIM('barbar   ')                                    |
+-------------------------------------------------------+
| barbar                                                |
+-------------------------------------------------------+
1 row in set (0.00 sec)
```

LENGTH(str)

Returns the length of the string str, measured in bytes. A multi-byte character counts as multiple bytes. This means that for a string containing five two-byte characters, LENGTH() returns 10, whereas CHAR_LENGTH() returns 5.

```
SQL> SELECT LENGTH('text');
+-------------------------------------------------------+
| LENGTH('text')                                        |
+-------------------------------------------------------+
| 4                                                     |
+-------------------------------------------------------+
1 row in set (0.00 sec)
```

LOWER(str):

Returns the string str with all characters changed to lowercase according to the current character set mapping.

```
SQL> SELECT LOWER('QUADRATICALLY');
+-----------------------------------------------------------+
| LOWER('QUADRATICALLY')                                    |
+-----------------------------------------------------------+
| quadratically                                             |
+-----------------------------------------------------------+
1 row in set (0.00 sec)
```

UPPER(str):

Returns the string str with all characters changed to uppercase according to the current character set mapping.

```
SQL> SELECT UPPER('Allah-hus-samad');
+-----------------------------------------------------------+
| UPPER('Allah-hus-samad')                                  |
+-----------------------------------------------------------+
| ALLAH-HUS-SAMAD                                           |
+-----------------------------------------------------------+
1 row in set (0.00 sec)
```

SUBSTRING(str,pos)

SUBSTRING(str FROM pos)

SUBSTRING(str,pos,len)

SUBSTRING(str FROM pos FOR len)

The forms without a len argument return a substring from string str starting at position pos. The forms with a len argument return a substring len characters long from string str, starting at position pos. The forms that use FROM are standard SQL syntax. It is also possible to use a negative value for pos. In this case, the beginning of the substring is pos characters from the end of the string, rather than the beginning. A negative value may be used for pos in any of the forms of this function.

```
SQL> SELECT SUBSTRING('Quadratically',5);
+-----------------------------------------------------------+
| SSUBSTRING('Quadratically',5)                             |
+-----------------------------------------------------------+
| ratically                                                 |
+-----------------------------------------------------------+
1 row in set (0.00 sec)

SQL> SELECT SUBSTRING('foobarbar' FROM 4);
+-----------------------------------------------------------+
| SUBSTRING('foobarbar' FROM 4)                             |
+-----------------------------------------------------------+
| barbar                                                    |
+-----------------------------------------------------------+
1 row in set (0.00 sec)

SQL> SELECT SUBSTRING('Quadratically',5,6);
+-----------------------------------------------------------+
| SUBSTRING('Quadratically',5,6)                            |
+-----------------------------------------------------------+
| ratica                                                    |
+-----------------------------------------------------------+
1 row in set (0.00 sec)
```

INSTR(str,substr)

Returns the position of the first occurrence of substring substr in string str. This is the same as the two-argument form of LOCATE(), except that the order of the arguments is reversed.

```
SQL> SELECT INSTR('foobarbar', 'bar');
+-----------------------------------------------------------+
| INSTR('foobarbar', 'bar')                                 |
+-----------------------------------------------------------+
| 4                                                         |
+-----------------------------------------------------------+
1 row in set (0.00 sec)
```

DATE FUNCTIONS:

SYSDATE(): Returns the current date and time as a value in 'YYYY-MM-DD HH:MM:SS' or YYYYMMDDHHMMSS format, depending on whether the function is used in a string or numeric context.

Example: Select SYSDATE();

NEXT_DAY (): The Oracle/PLSQL NEXT_DAY function returns the first weekday that is greater than a date.

Syntax:

NEXT_DAY(date, weekday)

ADD_MONTHS():

The Oracle/PLSQL ADD_MONTHS function returns a date with a specified number of months added.

Syntax: ADD_MONTHS(date1, number_months)

LAST_DAY()

The Oracle/PLSQL LAST_DAY function returns the last day of the month based on a date value.

Syntax: LAST_DAY(date)

TRUNC (date):

The TRUNC (date) function returns date with the time portion of the day truncated to the unit specified by the format model fmt. The value returned is always of datatype DATE, even if you specify a different datetime datatype for date. If you omit fmt, then date is truncated to the nearest day.

Examples:

SELECT TRUNC(TO_DATE('27-OCT-92','DD-MON-YY'), 'YEAR')

"New Year" FROM DUAL;

New Year

01-JAN-92

EXPERIMENT 4

AIM: Database Querying – Simple Queries

ANY:

The ANY comparison condition is used to compare a value to a list or sub-query. It must be preceded by =, !=, >, <, <=, >= and followed by a list or sub-query.

When the ANY condition is followed by a list, the optimizer expands the initial condition to all elements of the list and strings them together with OR operators

Example:

SELECT empno, sal FROM emp WHERE sal > ANY (2000, 3000, 4000);

Transformed to equivalent statement without ANY

SELECT empno, sal FROM emp WHERE sal > 2000 OR sal > 3000 OR sal > 4000;

ALL:

The ALL comparison condition is used to compare a value to a list or sub query. It must be preceded by =, !=, >, <, <=, >= and followed by a list or sub query.

When the ALL condition is followed by a list, the optimizer expands the initial condition to all elements of the list and strings them together with AND operators

Example:

SELECT empno, sal FROM emp WHERE sal > ALL (2000, 3000, 4000);

Transformed to equivalent statement without ALL.

SELECT empno, sal FROM emp WHERE sal > 2000 AND sal > 3000 AND sal > 4000;

IN operator:

The IN operator allows you to specify multiple values in a WHERE clause.

Syntax:

SELECT column_name(s) FROM table_name WHERE column_name IN (value1,value2,...);

Example:

SELECT * FROM Customers WHERE City IN ('Paris','London');

EXISTS operator:

The SQL EXISTS condition is used in combination with a sub-query and is considered to be met, if the sub-query returns at least one row. It can be used in a SELECT, INSERT, UPDATE, or DELETE statement.

Syntax:

WHERE EXISTS (subquery);

Example - Using EXISTS Condition with the SELECT Statement we have a customers table with the following data:

customer_id	last_name	first_name	favorite_website
4000	Jackson	Joe	techonthenet.com
5000	Smith	Jane	digminecraft.com
6000	Ferguson	Samantha	bigactivities.com
7000	Reynolds	Allen	checkyourmath.com
8000	Anderson	Paige	NULL
9000	Johnson	Derek	techonthenet.com

Orders Table with Following data

order_id	customer_id	order_date
1	7000	2016/04/18
2	5000	2016/04/18
3	8000	2016/04/19
4	4000	2016/04/20

Now let's find all of the records from the customers table where there is at least one record in the orders table with the same customer_id.

SELECT * FROM customers WHERE EXISTS (SELECT * FROM orders WHERE customers.customer_id = orders.customer_id);

The Output for the above query

customer_id	last_name	first_name	favorite_website
4000	Jackson	Joe	techonthenet.com
5000	Smith	Jane	digminecraft.com
7000	Reynolds	Allen	checkyourmath.com
8000	Anderson	Paige	NULL

NOT EXISTS:

NOT EXISTS works like EXISTS, except the WHERE clause in which it is used is satisfied if no rows are returned by the sub-query.

Example:

SELECT * FROM customers WHERE NOT EXISTS (SELECT * FROM orders WHERE customers.customer_id = orders.customer_id);

UNION:

The UNION operator is used to combine the result-set of two or more SELECT statements. Notice that each SELECT statement within the UNION must have the same number of columns. The columns must also have similar data types. Also, the columns in each SELECT statement must be in the same order.

Syntax:

SELECT column_name(s) FROM table1 UNION
SELECT column_name(s) FROM table2;

The UNION operator selects only distinct values by default. To allow duplicate values, use the ALL keyword with UNION.

UNION ALL:

SELECT column_name(s) FROM table1 UNION
ALL
SELECT column_name(s) FROM table2;

The column names in the result-set of a UNION are usually equal to the column names in the first SELECT statement in the UNION.

Example:

SELECT City FROM Customers UNION
SELECT City FROM Suppliers ORDER BY City; (Before

Execution make sure that table data is available)

UNION ALL Example:

SELECT City FROM Customers UNION
ALL
SELECT City FROM Suppliers ORDER BY City;

INTERSECT

The SQL INTERSECT clause/operator is used to combine two SELECT statements, but returns rows only from the first SELECT statement that are identical to a row in the second SELECT statement. This means INTERSECT returns only common rows returned by the two SELECT statements.
Just as with the UNION operator, the same rules apply when using the INTERSECT operator.

Syntax:
SELECT column1 [, column2] FROM table1 [, table2] [WHERE condition]

INTERSECT

SELECT column1 [, column2] FROM table1 [, table2] [WHERE condition]

EXPERIMENT 5

AIM : Queries using aggregate functions, GROUP BY and HAVING clauses.

The queries that contain the GROUP BY clause are called grouped queries and only return a single row for every grouped item.

SQL GROUP BY Syntax

Now that we know what the SQL GROUP BY clause is, let's look at the syntax for a basic group by query.

```
SELECT statements... GROUP BY column_name1[,column_name2,...]
[HAVING condition];
```

- "SELECT statements..." is the standard SQL SELECT command query.
- "**GROUP BY** *column_name1*" is the clause that performs the grouping based on column_name1.
- "[,column_name2,...]" is optional; represents other column names when the grouping is done on more than one column.
- "[HAVING condition]" is optional; it is used to restrict the rows affected by the GROUP BY clause. It is similar to the WHERE clause.

Grouping using a Single Column

In order to help understand the effect of SQL Group By clause, let's execute a simple query that returns all the gender entries from the members table.

```
SELECT `gender` FROM `members` ;
```

Gender
Female
Female
Male
Female
Male
Male
Male
Male
Male

Suppose we want to get the unique values for genders. We can use a following query -

```
SELECT `gender` FROM `members` GROUP BY `gender`;
```

Executing the above script in MySQL workbench against the Myflixdb gives us the following results.

Gender
Female
Male

Note only two results have been returned. This is because we only have two gender types Male and Female. The GROUP BY clause in SQL grouped all the "Male" members together and returned only a single row for it. It did the same with the "Female" members.

Grouping using multiple columns

Suppose that we want to get a list of movie category_id and corresponding years in which they were released.

The output of this simple query

```
SELECT `category_id`,`year_released` FROM `movies` ;
```

category_id	year_released
1	2011
2	2008
NULL	2008
NULL	2010
8	2007
6	2007
6	2007
8	2005
NULL	2012
7	1920
8	NULL
8	1920

The above result has many duplicates.

Let's execute the same query using group by in SQL -

```
SELECT `category_id`,`year_released` FROM `movies`
GROUP BY `category_id`,`year_released`;
```

Executing the above script in MySQL workbench against the myflixdb gives us following results shown below.

category_id	year_released
NULL	2008
NULL	2010
NULL	2012
1	2011
2	2008
6	2007
7	1920
8	1920
8	2005
8	2007

The GROUP BY clause operates on both the category id and year released to identify **unique** rows in our above example.

If the category id is the same but the year released is different, then a row is treated as a unique one .If the category id and the year released is the same for more than one row, then it's considered a duplicate and only one row is shown.

Grouping and aggregate functions

Suppose we want total number of males and females in our database. We can use the following script shown below to do that.

```
SELECT `gender`,COUNT(`membership_number`) FROM `members` GROUP BY
`gender`;
```

Executing the above script in MySQL workbench against the myflixdb gives us the following results.

gender	COUNT('membership_number')
Female	3
Male	5

The results shown below are grouped by every unique gender value posted and the number of grouped rows is counted using the COUNT aggregate function.

Restricting query results using the HAVING clause

It's not always that we will want to perform groupings on all the data in a given table. There will be times when we will want to restrict our results to a certain given criteria. In such cases , we can use the HAVING clause

Suppose we want to know all the release years for movie category id 8. We would use the following script to achieve our results.

```
SELECT * FROM `movies` GROUP BY `category_id`,`year_released` HAVING `
category_id` = 8;
```

Executing the above script in MySQL workbench against the Myflixdb gives us the following results shown below.

movie_id	title	director	year_released	category_id
9	Honey mooners	John Schultz	2005	8
5	Daddy's Little Girls	NULL	2007	8

EXPERIMENT 6

AIM: Database Querying-Nested queries, sub-queries.

SUB QUERIES:

A Sub query or Inner query or Nested query is a query within another SQL query and embedded within the WHERE clause. A sub query is used to return data that will be used in the main query as a condition to further restrict the data to be retrieved. Sub queries can be used with the SELECT, INSERT, UPDATE, and DELETE statements along with the operators like =, <, >, >=, <=, IN, BETWEEN etc. There are a few rules that sub queries must follow:

- Sub-queries must be enclosed within parentheses.
- A sub-query can have only one column in the SELECT clause, unless multiple columns are in the main query for the sub-query to compare its selected columns.
- An ORDER BY cannot be used in a sub-query, although the main query can use an ORDER BY. The GROUP BY can be used to perform the same function as the ORDER BY in a sub-query.
- Sub queries that return more than one row can only be used with multiple value operators, such as the IN operator.
- The SELECT list cannot include any references to values that evaluate to a BLOB, ARRAY, CLOB, or NCLOB.
- A sub query cannot be immediately enclosed in a set function.
- The BETWEEN operator cannot be used with a sub-query; however, the BETWEEN operator can be used within the sub-query.

Sub queries with the SELECT Statement:

```
SELECT column_name [, column_name ] FROM   table1 [, table2 ]
WHERE  column_name OPERATOR   (SELECT column_name [, column_name ]
  FROM table1 [, table2 ]      [WHERE])
```

Example:

Consider the CUSTOMERS table having the following records:

ID	NAME	AGE	ADDRESS	SALARY
1	Ramesh	35	Ahmedabad	2000.00
2	Khilan	25	Delhi	1500.00
3	kaushik	23	Kota	2000.00
4	Chaitali	25	Mumbai	6500.00
5	Hardik	27	Bhopal	8500.00
6	Komal	22	MP	4500.00
7	Muffy	24	Indore	10000.00

SQL> SELECT * FROM CUSTOMERS WHERE ID IN (SELECT ID FROM CUSTOMERS WHERE SALARY > 4500) ;

This would produce the following result:

ID	NAME	AGE	ADDRESS	SALARY
4	Chaitali	25	Mumbai	6500.00
5	Hardik	27	Bhopal	8500.00
7	Muffy	24	Indore	10000.00

Subqueries with the UPDATE Statement:

The subquery can be used in conjunction with the UPDATE statement. Either single or multiple columns in a table can be updated when using a subquery with the UPDATE statement.The basic syntax is as follows:

UPDATE table SET column_name = new_value [WHERE OPERATOR [VALUE] (SELECT COLUMN_NAME FROM TABLE_NAME)

[WHERE)]

Example:

SQL> UPDATE CUSTOMERS SET SALARY = SALARY * 0.25 WHERE AGE IN (SELECT AGE FROM CUSTOMERS_BKP WHERE AGE >= 27);

This would impact two rows and finally CUSTOMERS table would have the following records:

ID	NAME	AGE	ADDRESS	SALARY
1	Ramesh	35	Ahmedabad	125.00
2	Khilan	25	Delhi	1500.00
3	kaushik	23	Kota	2000.00
4	Chaitali	25	Mumbai	6500.00
5	Hardik	27	Bhopal	2125.00
6	Komal	22	MP	4500.00
7	Muffy	24	Indore	10000.00

EXPERIMENT 7

Aim: Queries using Joins.

The SQL **Joins** clause is used to combine records from two or more tables in a database. A JOIN is a means for combining fields from two tables by using values common to each.

Consider the following two table

Table 1 − CUSTOMERS Table

```
| ID | NAME     | AGE | ADDRESS   | SALARY   |
|  1 | Ramesh   |  32 | Ahmedabad |  2000.00 |
|  2 | Khilan   |  25 | Delhi     |  1500.00 |
|  3 | kaushik  |  23 | Kota      |  2000.00 |
|  4 | Chaitali |  25 | Mumbai    |  6500.00 |
|  5 | Hardik   |  27 | Bhopal    |  8500.00 |
|  6 | Komal    |  22 | MP        |  4500.00 |
|  7 | Muffy    |  24 | Indore    | 10000.00 |
```

Table 2 − ORDERS Table

```
|OID  | DATE                | CUSTOMER_ID | AMOUNT |
| 102 | 2009-10-08 00:00:00 |           3 |   3000 |
| 100 | 2009-10-08 00:00:00 |           3 |   1500 |
| 101 | 2009-11-20 00:00:00 |           2 |   1560 |
| 103 | 2008-05-20 00:00:00 |           4 |   2060 |
```

Join these two tables in our SELECT statement as shown below.

```
SQL> SELECT ID, NAME, AGE, AMOUNT
  FROM CUSTOMERS, ORDERS
  WHERE  CUSTOMERS.ID = ORDERS.CUSTOMER_ID;
```

This would produce the following result.

```
+----+----------+-----+--------+
| ID | NAME     | AGE | AMOUNT |
+----+----------+-----+--------+
|  3 | kaushik  |  23 |   3000 |
|  3 | kaushik  |  23 |   1500 |
|  2 | Khilan   |  25 |   1560 |
|  4 | Chaitali |  25 |   2060 |
+----+----------+-----+--------+
```

Here, it is noticeable that the join is performed in the WHERE clause. Several operators can be used to join tables, such as =, <, >, <>, <=, >=, !=, BETWEEN, LIKE, and NOT; they can all be used to join tables. However, the most common operator is the equal to symbol.

There are different types of joins available in SQL −

- INNER JOIN − returns rows when there is a match in both tables.
- LEFT JOIN − returns all rows from the left table, even if there are no matches in the
- right table.
- RIGHT JOIN − returns all rows from the right table, even if there are no matches in the left table.
- FULL JOIN − returns rows when there is a match in one of the tables.
- SELF JOIN − is used to join a table to itself as if the table were two tables, temporarily renaming at least one table in the SQL statement.
- CARTESIAN JOIN − returns the Cartesian product of the sets of records from the two or more joined tables.

Let us now discuss each of these joins in detail.

INNER JOIN

The most important and frequently used of the joins is the **INNER JOIN**. They are also referred to as an **EQUIJOIN**.

The INNER JOIN creates a new result table by combining column values of two tables (table1 and table2) based upon the join-predicate. The query compares each row of table1 with each row of table2 to find all pairs of rows which satisfy the join-predicate. When the join-predicate is satisfied, column values for each matched pair of rows of A and B are combined into a result row.

Syntax

The basic syntax of the **INNER JOIN** is as follows.

```
SELECT table1.column1, table2.column2...
FROM table1
INNER JOIN table2
ON table1.common_field = table2.common_field;
```

Example

Consider the following two tables.

Table 1 − CUSTOMERS Table is as follows.

```
+----+----------+-----+-----------+----------+
| ID | NAME     | AGE | ADDRESS   | SALARY   |
+----+----------+-----+-----------+----------+
|  1 | Ramesh   |  32 | Ahmedabad |  2000.00 |
|  2 | Khilan   |  25 | Delhi     |  1500.00 |
|  3 | kaushik  |  23 | Kota      |  2000.00 |
|  4 | Chaitali |  25 | Mumbai    |  6500.00 |
|  5 | Hardik   |  27 | Bhopal    |  8500.00 |
|  6 | Komal    |  22 | MP        |  4500.00 |
|  7 | Muffy    |  24 | Indore    | 10000.00 |
+----+----------+-----+-----------+----------+
```

Table 2 − ORDERS Table is as follows.

```
+-----+---------------------+-------------+--------+
| OID | DATE                | CUSTOMER_ID | AMOUNT |
+-----+---------------------+-------------+--------+
| 102 | 2009-10-08 00:00:00 |           3 |   3000 |
| 100 | 2009-10-08 00:00:00 |           3 |   1500 |
| 101 | 2009-11-20 00:00:00 |           2 |   1560 |
| 103 | 2008-05-20 00:00:00 |           4 |   2060 |
+-----+---------------------+-------------+--------+
```

Now, let us join these two tables using the INNER JOIN as follows −

```
SQL> SELECT  ID, NAME, AMOUNT, DATE
   FROM CUSTOMERS
   INNER JOIN ORDERS
   ON CUSTOMERS.ID = ORDERS.CUSTOMER_ID;
```

This would produce the following result.

```
+----+----------+--------+---------------------+
```

```
| ID | NAME     | AMOUNT | DATE                |
+----+----------+--------+---------------------+
|  3 | kaushik  |   3000 | 2009-10-08 00:00:00 |
|  3 | kaushik  |   1500 | 2009-10-08 00:00:00 |
|  2 | Khilan   |   1560 | 2009-11-20 00:00:00 |
|  4 | Chaitali |   2060 | 2008-05-20 00:00:00 |
+----+----------+--------+---------------------+
```

SELF JOIN

The SQL **SELF JOIN** is used to join a table to itself as if the table were two tables; temporarily renaming at least one table in the SQL statement.

Syntax

The basic syntax of SELF JOIN is as follows −

```
SELECT a.column_name, b.column_name...
FROM table1 a, table1 b
WHERE a.common_field = b.common_field;
```

Here, the WHERE clause could be any given expression based on your requirement.

Example

Consider the following table.

CUSTOMERS Table is as follows.

```
| ID | NAME     | AGE | ADDRESS   | SALARY   |
+----+----------+-----+-----------+----------+
|  1 | Ramesh   |  32 | Ahmedabad |  2000.00 |
|  2 | Khilan   |  25 | Delhi     |  1500.00 |
|  3 | kaushik  |  23 | Kota      |  2000.00 |
|  4 | Chaitali |  25 | Mumbai    |  6500.00 |
|  5 | Hardik   |  27 | Bhopal    |  8500.00 |
|  6 | Komal    |  22 | MP        |  4500.00 |
|  7 | Muffy    |  24 | Indore    | 10000.00 |
+----+----------+-----+-----------+----------+
```

SELF JOIN

```
SQL> SELECT  a.ID, b.NAME, a.SALARY
  FROM CUSTOMERS a, CUSTOMERS b
  WHERE a.SALARY < b.SALARY;
```

This would produce the following result −

```
+----+----------+---------+
| ID | NAME     | SALARY  |
+----+----------+---------+
|  2 | Ramesh   | 1500.00 |
|  2 | kaushik  | 1500.00 |
|  1 | Chaitali | 2000.00 |
|  2 | Chaitali | 1500.00 |
|  3 | Chaitali | 2000.00 |
|  6 | Chaitali | 4500.00 |
|  1 | Hardik   | 2000.00 |
|  2 | Hardik   | 1500.00 |
|  3 | Hardik   | 2000.00 |
|  4 | Hardik   | 6500.00 |
|  6 | Hardik   | 4500.00 |
|  1 | Komal    | 2000.00 |
|  2 | Komal    | 1500.00 |
|  3 | Komal    | 2000.00 |
|  1 | Muffy    | 2000.00 |
|  2 | Muffy    | 1500.00 |
|  3 | Muffy    | 2000.00 |
|  4 | Muffy    | 6500.00 |
|  5 | Muffy    | 8500.00 |
|  6 | Muffy    | 4500.00 |
+----+----------+---------+
```

EXPERIMENT 8

AIM : Queries using Views.

VIEWS:

A view is nothing more than a SQL statement that is stored in the database with an associated name. A view is actually a composition of a table in the form of a predefined SQL query. A view can contain all rows of a table or select rows from a table. A view can be created from one or many tables which depends on the written SQL query to create a view. Views, which are kind of virtual tables, allow users to do the following:

- Structure data in a way that users or classes of users find natural or intuitive.
- Restrict access to the data such that a user can see and (sometimes) modify exactly what they need and no more.
- Summarize data from various tables which can be used to generate reports.

Creating Views:

Database views are created using the CREATE VIEW statement. Views can be created from a single table, multiple tables, or another view.

To create a view, a user must have the appropriate system privilege according to the specific implementation.

The basic **CREATE VIEW syntax** is as follows:

CREATE VIEW view_name AS SELECT column1, column2.....

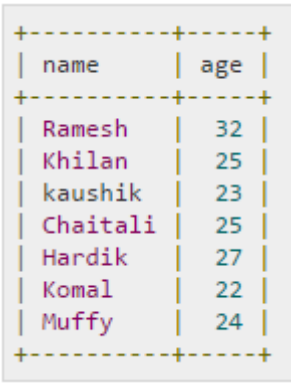
FROM table_name WHERE [condition];

Example:

Consider the CUSTOMERS table having the following records:

ID	NAME	AGE	ADDRESS	SALARY
1	Ramesh	32	Ahmedabad	2000.00
2	Khilan	25	Delhi	1500.00
3	kaushik	23	Kota	2000.00
4	Chaitali	25	Mumbai	6500.00
5	Hardik	27	Bhopal	8500.00
6	Komal	22	MP	4500.00
7	Muffy	24	Indore	10000.00

SQL > CREATE VIEW CUSTOMERS_VIEW AS
SELECT name, age FROM CUSTOMERS;
SQL> SELECT * FROM CUSTOMERS_VIEW;



name	age
Ramesh	32
Khilan	25
kaushik	23
Chaitali	25
Hardik	27
Komal	22
Muffy	24

Dropping Views:

If you have a view, you need a way to drop the view if it is no longer needed.

Syntax:

DROP VIEW view_name;

Following is an example to drop CUSTOMERS_VIEW from CUSTOMERS table:

DROP VIEW CUSTOMERS_VIEW;

EXPERIMENT 9

AIM : Procedures and Functions

Oracle PL/SQL Stored Procedure & Functions with Examples

Procedures and Functions are the subprograms which can be created and saved in the database as database objects. They can be called or referred inside the other blocks also.

Terminologies in PL/SQL Subprograms

Parameter:

The parameter is variable or placeholder of any valid PL/SQL datatype through which the PL/SQL subprogram exchange the values with the main code. This parameter allows to give input to the subprograms and to extract from these subprograms.

- These parameters should be defined along with the subprograms at the time of creation.
- These parameters are included n the calling statement of these subprograms to interact the values with the subprograms.
- The datatype of the parameter in the subprogram and the calling statement should be same.
- The size of the datatype should not mention at the time of parameter declaration, as the size is dynamic for this type.

Based on their purpose parameters are classified as

1. IN Parameter
2. OUT Parameter
3. IN OUT Parameter

IN Parameter:

- This parameter is used for giving input to the subprograms.
- It is a read-only variable inside the subprograms. Their values cannot be changed inside the subprogram.
- In the calling statement, these parameters can be a variable or a literal value or an expression, for example, it could be the arithmetic expression like '5*8' or 'a/b' where 'a' and 'b' are variables.
- By default, the parameters are of IN type.

OUT Parameter:

- This parameter is used for getting output from the subprograms.
- It is a read-write variable inside the subprograms. Their values can be changed inside the subprograms.
- In the calling statement, these parameters should always be a variable to hold the value from the current subprograms.

IN OUT Parameter:

- This parameter is used for both giving input and for getting output from the subprograms.
- It is a read-write variable inside the subprograms. Their values can be changed inside the subprograms.
- In the calling statement, these parameters should always be a variable to hold the value from the subprograms.

Syntax:

CREATE OR REPLACE PROCEDURE

<procedure_name>

(

<parameterl IN/OUT <datatype>

..

.

)

[IS | AS]

<declaration_part>

BEGIN

<execution part>

EXCEPTION

<exception handling part>

END;

- CREATE PROCEDURE instructs the compiler to create new procedure in Oracle. Keyword 'OR REPLACE' instructs the compile to replace the existing Procedure (if any) with the current one.
- Procedure name should be unique.
- Keyword 'IS' will be used, when the stored procedure in Oracle is nested into some other blocks. If the procedure is standalone then 'AS' will be used. Other than this coding standard, both have the same meaning.

Creating Procedure and calling it using EXEC

In this example, we are going to create an Oracle procedure that takes the name as input and prints the welcome message as output. We are going to use EXEC command to call procedure.

```
CREATE OR REPLACE PROCEDURE welcome_msg (p_name IN VARCHAR2)

IS

BEGIN

dbms_output.put_line (‘Welcome '|| p_name);

END;

/

EXEC welcome_msg (‘Guru99’);
```

What is Function

Functions is a standalone PL/SQL subprogram. Like PL/SQL procedure, functions have a unique name by which it can be referred. These are stored as PL/SQL database objects. Below are some of the characteristics of functions.

- Functions are a standalone block that is mainly used for calculation purpose.
- Function use RETURN keyword to return the value, and the data type of this is
- defined at the time of creation.
- A Function should either return a value or raise the exception, i.e. return is mandatory in functions.
- Function with no DML statements can be directly called in SELECT query whereas the function with DML operation can only be called from other PL/SQL blocks.
- It can have nested blocks, or it can be defined and nested inside the other

blocks or packages.

- It contains declaration part (optional), execution part, exception handling part (optional).
- The values can be passed into the function or fetched from the procedure through the parameters.
- These parameters should be included in the calling statement.
- A PLSQL function can also return the value through OUT parameters other than using RETURN.
- Since it will always return the value, in calling statement it always accompanies with assignment operator to populate the variables.

Syntax

CREATE OR REPLACE FUNCTION

<procedure_name>

(

<parameterl IN/OUT <datatype>

)

RETURN <datatype>

[IS | AS]

<declaration_part>

BEGIN

<execution part>

EXCEPTION

<exception handling part>

END;

Creating Function and calling it using Anonymous Block

In this program, we are going to create a function that takes the name as input and returns the welcome message as output. We are going to use anonymous block and select statement to call the function.

```
CREATE OR REPLACE FUNCTION welcome_msgJune ( p_name IN VARCHAR2)
RETURN VAR.CHAR2
IS
BEGIN
RETURN (‘Welcome ‘|| p_name);
END;
/
DECLARE
lv_msg VARCHAR2(250);
BEGIN
lv_msg := welcome_msg_func (‘Guru99’);
dbms_output.put_line(lv_msg);
END;
SELECT welcome_msg_func(‘Guru99:) FROM DUAL;
```

EXPERIMENT 10

AIM : Implicit and Explicit Cursors

Cursor– We have seen how oracle executes an SQL statement. Oracle DBA uses a work area for its internal processing. This work area is private to SQL's operation and is called a **cursor.**

The data that is stored in the cursor is called the **Active Data set.** The size of the cursor in memory is the size required to hold the number of rows in the Active Data Set.

Explicit Cursor- You can explicitly declare a cursor to process the rows individually. A cursor declared by the user is called **Explicit Cursor.** For Queries that return more than one row, You must declare a cursor explicitly.

The data that is stored in the cursor is called the **Active Data set.** The size of the cursor in memory is the size required to hold the number of rows in the Active

Why use an Explicit Cursor- Cursor can be used when the user wants to process data one row at a time.

Explicit Cursor Management-

The steps involved in declaring a cursor and manipulating data in the active data set are:-

- Declare a cursor that specifies the SQL select statement that you want to process.
- Open the Cursor.
- Fetch the data from the cursor one row at a time.
- Close the cursor.

Explicit Cursor Attributes-

Oracle provides certain attributes/ cursor variables to control the execution of the cursor. Whenever any cursor(explicit or implicit) is opened and used Oracle creates a set of four system variables via which Oracle keeps track of the 'Current' status of the cursor.

- Declare a cursor that specifies the SQL select statement that you want to process.
- Open the Cursor.
- Fetch the data from the cursor one row at a time.
- Close the cursor.

How to Declare the Cursor:-

The General Syntax to create any particular cursor is as follows:-

Cursor <Cursorname> is Sql Statement;

How to Open the Cursor:-

The General Syntax to Open any particular cursor is as follows:-

Open Cursorname

Fetching a record From the Cursor:

The fetch statement retrieves the rows from the active set to the variables one at a time. Each time a fetch is executed. The focus of the DBA cursor advances to the next row in the Active set.

One can make use of any loop structure(Loop-End Loop along with While,For) to fetch the records from the cursor into variable one row at a time.

The General Syntax to Fetch the records from the cursor is as follows:-

Fetch cursorname into variable1,variable2, ——

Closing a Cursor:-

The General Syntax to Close the cursor is as follows:-

Close <cursorname>;

EXPERIMENT 11

AIM : Triggers

Database Triggers:-

Database triggers are procedures that are stored in the database and are implicitly executed(fired) when the contents of a table are changed.

Use of Database Triggers:-

Database triggers support Oracle to provide a highly customized database management system. Some of the uses to which the database triggers can be put to customize management information in Oracle are as follows:-

- A Trigger can permit DML statements against a table any if they are issued, during regular business hours or on predetermined weekdays.
- A trigger can also be used to keep an audit trail of a table along with the operation performed and the time on which the operation was performed.
- It can be used to prevent invalid transactions.
- Enforce complex security authorizations

How to apply DataBase Triggers:-

A trigger has three basic parts:-

1. A triggering event or statement.
2. A trigger restriction
3. A trigger action.

Types of Triggers:-

Using the various options , four types of triggers can be created:-

Before Statement Trigger:- Before executing the triggering statement, the trigger action is executed.

Before Row Trigger:- Before modifying the each row affected by the triggering statement and before appropriate integrity constraints, the trigger is executed if the trigger restriction either evaluated to TRUE or was not included.'

After Statement Trigger:- After executing the triggering statement and applying any deferred integrity constraints, the trigger action is executed.

After row Trigger:- After modifying each row affected by the triggering statement and possibly applying appropriate integrity constraints, the trigger action is executed for the current row if the trigger restriction either evaluates to TRUE or was not included.

Syntax For Creating Trigger:-

The syntax for Creating the Trigger is as follows:-

Create or replace Trigger<Triggername> {Before,After} {Delete, Insert, Update } On <Tablename> For Each row when
Condition Declare
<Variable declarations>;
<Constant
Declarations>;
Begin
<PL/SQL> Subprogram
Body; Exception
Exception Pl/SQL
block;
End;

How to Delete a Trigger:-

The syntax for Deleting the Trigger is as follows:-

Drop Trigger <Triggername>;

EXPERIMENT 12

AIM: Exception Handling

Exceptions (PL/SQL runtime errors) can arise from design faults, coding mistakes, hardware failures, and many other sources. You cannot anticipate all possible exceptions, but you can write exception handlers that let your program to continue to operate in their presence.

Any PL/SQL block can have an exception-handling part, which can have one or more exception handlers. For example, an exception-handling part could have this syntax:

```
EXCEPTION
  WHEN ex_name_1 THEN statements_1          -- Exception handler
  WHEN ex_name_2 OR ex_name_3 THEN statements_2  -- Exception
handler
  WHEN OTHERS THEN statements_3             -- Exception handler
END;
```

Declaring, Raising, and Handling User-Defined Exception

```
CREATE PROCEDURE account_status (
  due_date DATE,
  today    DATE
) AUTHID DEFINER
IS
  past_due  EXCEPTION;  -- declare exception
BEGIN
  IF due_date < today THEN
    RAISE past_due;  -- explicitly raise exception
  END IF;
EXCEPTION
  WHEN past_due THEN  -- handle exception
    DBMS_OUTPUT.PUT_LINE ('Account past due.');
END;
```

```
/
 BEGIN
  account_status (TO_DATE('01-JUL-2010', 'DD-MON-YYYY'),
                TO_DATE('09-JUL-2010', 'DD-MON-YYYY'));
END;
/Result : Account past due.
```