Lab Manual
for

CSE 314 (Database Management System Lab)

Credit: 1.5, Contact hour: 2.15 Hours Per week

Department of Computer Science & Engineering
Varendra University
Rajshahi, Bangladesh

Varendra University

Department of Computer Science and Engineering

CSE 314

Database Management System Lab

Student ID	
Student Name	
Section	
Name of the Program	
Name of the Department	

INDEX

INSTRUCTIONS FOR LABORATORY

Do's

- ✓ Do wear ID card.
- ✓ Do log off the computers when you finish.
- ✓ Do ask for assistance if you need help.
- ✓ Do keep your voice low when speaking to others in the LAB.
- ✓ Do ask for assistance in downloading any software.
- ✓ Do make suggestions as to how we can improve the LAB.
- ✓ In case of any hardware related problem, ask LAB in charge for solution.
- ✓ If you are the last one leaving the LAB, make sure that the staff in charge of the LAB is informed to close the LAB.
- ✓ Be on time to LAB sessions.
- ✓ Do keep the LAB as clean as possible.

Don'ts

- o Do not use mobile phone inside the lab.
- o Don't do anything that can make the LAB dirty (like eating, throwing waste papers etc).
- o Do not carry any external devices without permission.
- o Don't move the chairs of the LAB.
- o Don't interchange any part of one computer with another.
- o Don't leave the computers of the LAB turned on while leaving the LAB.
- o Do not install or download any software or modify or delete any system files on any lab computers.
- o Do not damage, remove, or disconnect any labels, parts, cables, or equipment.
- o Don't attempt to bypass the computer security system.
- o Do not read or modify other user's file.
- o If you leave the lab, do not leave your personal belongings unattended. We are not responsible for any theft.

Varendra University

COURSE SYLLABUS

1	Faculty	Faculty of Science & Engineering
2	Department	Department of CSE
3	Program	B.Sc. in Computer Science and Engineering
4	Name of Course	Database Management System Lab
5	Course Code	CSE 314
6	Year	Spring 2025
7	Pre-requisites	CSE 124
8	Status	Core Course
9	Credit Hours	2.15
10	Section	A
11	Class Hours	Not Defined
12	Class Location	Not Defined
13	Name (s) of Academic staff / Instructor(s)	Md. Toufikul Islam, Sumaia Rahman
14	Contact	toufikul.cse@gmail.com, ononnaontora@gmail.com ,
15	Office	532, Jahangir Sharoni, Talaimari, Rajshahi-6204
16	Counseling Hours	
17	Text Book	1. A. Silberschatz: **Database System Concepts,** Mcgraw-Hill. 2. R. Ramakrishnan,,Johannes Gehrke : **Database Management System,** McGraw-Hill Higher Education 3. James Martin : **Principles of Database Management**, Prentice-hall 4. Ullman : **Database Management systems**, Prentice-Hall Publication. 5. Abey : **Oracle 8i a Beginners Guide**, McGraw Hill.
18	Reference	1. www.stackoverflow.com 2. www.tutorialspoint.com 3. www.javatpoint.com
19	Equipment & Aids	1. Lab Sheet 2. Text Book (PDF)/Slide 3. Oracle software (10g or higher versions) 4. Toad/Sql Developer Tool

20	Course Description	Developing and managing efficient and effective database applications requires understanding the fundamentals of database management systems, techniques for the design of databases, and principles of database administration. This course covers database design and the use of databases in applications, with a short introduction to the internals of relational database engines. It includes extensive coverage of the relational model, relational algebra, and SQL. The course also features database design and relational design principles based on dependencies and normal forms. Many additional key database topics from the design and application-building perspective are also covered, including indexes, views, transactions, and integrity constraints. Practical design of databases and developing database applications using modern software tools will be emphasized.
21	Course Objectives	The major objective of this lab is to 1. Provide a sound introduction to the discipline of database management as a subject in its own right, rather than as a compendium of techniques and product-specific tools. 2. Familiarise the participant with the nuances of database environments towards an information-oriented data-processing oriented framework 3. Give a good formal foundation on the relational model of data 4. Give an introduction to systematic database design approaches covering conceptual design, logical design and an overview of physical design 5. Motivate the participants to relate all these to one or more commercial product environments as they relate to the developer tasks 6. Present the concepts and techniques relating to query processing by SQL engines 7. Present the concepts and techniques relating to ODBC and its implementations. 8. Introduce the concepts of transactions and transaction processing 9. Present the issues and techniques relating to concurrency and recovery in multi-user database environments

22	Learning Outcomes	After the successful completion of this course, students will be able to 1. Understand the fundamentals of relational, distributed database systems including: data models, database architectures, and database manipulations 2. Understand the theories and techniques in developing database applications and be able to demonstrate the ability to build databases using enterprise DBMS products such as Oracle or SQL Server. 3. Be familiar with managing database systems 4. Understand new developments and trends in databases. **Learning Outcomes:** 1. After undergoing this laboratory module, the participant should be able to: 2. Understand, appreciate and effectively explain the underlying concepts of database technologies 3. Design and implement a database schema for a given problem-domain 4. Populate and query a database using SQL DML/DDL commands. 5. Programming PL/SQL including stored procedures, stored functions
23	Teaching Methods	Lecture, Problem Solving, Brainstorming

24	Assessment Methods	**Assessment Types**	**Marks**
		Attendance	10%
		Laboratory Report	10%
		Laboratory Viva-voce	10%
		Continuous Assessment	20%
		Laboratory Project/ Experimental/ Simulation Test	20%
		Laboratory Final Quiz/ Examination	30%
		Total	**100%**

<table>
<tr><td>25</td><td>Grading Policy</td><td>
<table>
<tr><th>Letter Grade</th><th>Marks %</th><th>Grade Point</th><th>Letter Grade</th><th>Marks %</th><th>Grade Point</th></tr>
<tr><td>A+ (Plus)</td><td>80-100</td><td>4.00</td><td>C+ (Plus)</td><td>50-54</td><td>2.50</td></tr>
<tr><td>A (Plain)</td><td>75-79</td><td>3.75</td><td>C (Plain)</td><td>45-49</td><td>2.25</td></tr>
<tr><td>A- (Minus)</td><td>70-74</td><td>3.50</td><td>D (Plain)</td><td>40-44</td><td>2.00</td></tr>
<tr><td>B+ (Plus)</td><td>65-69</td><td>3.25</td><td>F (Fail)</td><td><40</td><td>0.00</td></tr>
<tr><td>B (Plain)</td><td>60-64</td><td>3.00</td><td></td><td></td><td></td></tr>
<tr><td>B- (Minus)</td><td>55-59</td><td>2.75</td><td></td><td></td><td></td></tr>
</table>
</td></tr>
<tr><td>26</td><td>Additional Course Policies</td><td>1. Lab Reports
Report on previous Experiment must be submitted before the beginning of new experiment. A bonus may be obtained if a student submits a neat, clean and complete lab report.

2. Examination
There will be a lab exam at the end of the semester that will be closed book.

3. Unfair means policy
In case of copying/plagiarism in any of the assessments, the students involved will receive zero marks. Zero Tolerance will be shown in this regard. In case of severe offences, actions will be taken as per university rule.

4. Counseling
Students are expected to follow the counseling hours posted. In case of emergency/unavoidable situations, students can e-mail me to make an appointment. Students are regularly advised to check the piazza course page for updates/materials.</td></tr>
<tr><td></td><td></td><td>5. Policy for Absence in Class/Exam
If a student is absent in the class for anything other than medical reasons, he/she will not receive attendance. If a student misses a class for genuine medical reasons, he/she must apply with the supporting documents (prescription/medical report). He/she will then have to follow the instructions given by the instructor for make- up.

In case of absence in the mid/final exam for medical grounds, the student must also get his/her application forwarded by the head of the department before a make-up exam can be taken.

It is recommended that the students inform the instructor beforehand through mail if they feel that they will miss a class/evaluation due to medical reasons.</td></tr>
</table>

EXPERIMENT NO : 01

Title: Implementation of DDL commands of SQL with suitable examples

- ❖ Create table
- ❖ Alter table
- ❖ Drop Table
- ❖ Sequence

SQL language is sub-divided into several language elements, including:

- **Clauses,** which are in some cases optional, constituent components of statements and queries.
- **Expressions,** which can produce either scalar values or tables consisting of columns and rows of data.
- **Predicates** which specify conditions that can be evaluated to SQL three-valued logic (3VL) Boolean truth values and which are used to limit the effects of statements and queries, or to change program flow.
- **Queries** which retrieve data based on specific criteria.
- **Statements** which may have a persistent effect on schemas and data, or which may control transactions, program flow, connections, sessions, or diagnostics.
- SQL statements also include the semicolon (";") statement terminator. Though not required on every platform, it is defined as a standard part of the SQL grammar.
- **Insignificant white space** is generally ignored in SQL statements and queries, making it easier to format SQL code for readability.

Domain types in SQL-

The SQL standard supports a variety of built in domain types, including-

Char (n)- A fixed length character length string with user specified length .

Varchar (n)- A variable character length string with user specified maximum length n.

Int- An integer.

Small integer- A small integer.

Numeric (p, d)-A Fixed point number with user defined precision.

Real, double precision- Floating point and double precision floating point numbers with machine dependent precision.

Float (n)- A floating point number, with precision of at least n digits.

Date- A calendar date containing a (four digit) year, month and day of the month.

Time- The time of day, in hours, minutes and seconds Eg. Time '09:30:00'.

Number- Number is used to store numbers (fixed or floating point).

There are five types of SQL statements. They are:

- ✓ **Data Definition Language (DDL)** Statements- Create, Alter, Truncate, Drop, Rename.
- ✓ **Data Manipulation Language (DML)** Statements- DELETE, EXPLAIN PLAN, INSERT, LOCK TABLE, MERGE, SELECT, UPDATE
- ✓ **Transaction Control Statements(TCL)-**Commit, Rollback, Savepoint, Set transaction
- ✓ **Data Control Language (DCL)-**grant, revoke.
- ✓ **Session Control Statements**
- ✓ **System Control Statement**
- ✓ **Embedded SQL Statements**

1. DATA DEFINITION LANGUAGE (DDL): The Data Definition Language (DDL) is used to create and destroy databases and database objects. These commands will primarily be used by database administrators during the setup and removal phases of a database project. Let's take a look at the structure and usage of four basic DDL commands:

1. CREATE 2. ALTER 3. DROP 4. RENAME

1. CREATE:

a) CREATE TABLE: This is used to create a new relation (table)

Syntax: CREATE TABLE <relation_name/table_name >

(field_1 data_type(size),field_2 data_type(size), .. .);

Example: SQL> CREATE TABLE Student (sno NUMBER (3), sname CHAR (10), class CHAR (5));

2. ALTER:

a) **ALTER TABLE ...ADD...:** This is used to add some extra fields into existing relation.

Syntax: ALTER TABLE relation_name ADD (new field_1 data_type(size), new field_2 data_type(size),..);

Example: SQL>ALTER TABLE std ADD (Address CHAR(10));

b) **ALTER TABLE...ALTER COLUMN...:** This is used to change the width as well as data type of fields of existing relations.

Syntax: ALTER TABLE relation_name ALTER COLUMN (field_1 newdata_type(Size), field_2 newdata_type(Size),....field_newdata_type(Size));

Example:SQL>ALTER TABLE student ALTER COLUMN (sname VARCHAR(10),class VARCHAR(5));

c) **ALTER TABLE..DROP...:** This is used to remove any field of existing relations.

Syntax: ALTER TABLE relation_name DROP COLUMN (field_name);

Example:SQL>ALTER TABLE student DROP column (sname);

d) **ALTER TABLE..RENAME...:** This is used to change the name of fields in existing relations.

Syntax: ALTER TABLE relation_name RENAME COLUMN (OLD field_name) to (NEW field_name);

Example: SQL>ALTER TABLE student RENAME COLUMN sname to stu_name;

e) **Sp_rename:**

Syntax: sp_rename 'table_name.oldColumnName', 'newColumnName', 'COLUMN'

Example: sp_rename 'EMPLOYEE.DEPT_NAME', 'DEPARTMENT', 'COLUMN'

DATA MANIPULATION LANGUAGE (DML): The Data Manipulation Language (DML) is used to retrieve, insert and modify database information. These commands will be used by all database users during the routine operation of the database. Let's take a brief look at the basic DML commands:

1. INSERT 2. UPDATE 3. DELETE

1. INSERT INTO: This is used to add records into a relation. These are three type of INSERT INTO queries which are as

a) Inserting a single record

Syntax: INSERT INTO < relation/table name> (field_1,field_2,….,field_n)VALUES (data_1,data_2,…..,data_n);

Example: SQL>INSERT INTO student(sno,sname,class,address) VALUES (1,'Ravi','M.Tech','Palakol');

b) Inserting all records from another relation

Syntax: INSERT INTO relation_name_1 SELECT Field_1,field_2,field_n

FROM relation_name_2 WHERE field_x=data;

Example: SQL>INSERT INTO std SELECT sno,sname FROM student WHERE name = 'Ramu';

c) Inserting multiple records

Syntax: INSERT INTO relation_name (field_1,field_2, …,field_n) VALUES (data_1_1,data_2_1,….,data_n_1), (data_1_2,data_2_2,….,data_n_2), (data_1_3,data_2_3,….,data_n_3);

Example: SQL>INSERT INTO student (sno, sname, semester) VALUES (1,john,1st), (2,michael,1st), (3,david,1st);

2. UPDATE-SET-WHERE: This is used to update the content of a record in a relation.

Syntax: SQL>UPDATE relation name SET Field_name1=data,field_name2=data, WHERE field_name=data;

Example: SQL>UPDATE student SET sname = 'peter' WHERE sno=1;

UPDATE WITH JOIN

Syntax: UPDATE t1 SET t1.c1 = t2.c2,t1.c2 = expression, ... FROM t1 [INNER | LEFT] JOIN t2 ON join_predicate WHERE where_predicate;

Example: UPDATE commissions SET commissions.commission = c.base_amount * t.percentage FROM commissions c INNER JOIN targets t ON c.target_id = t.target_id;

3. DELETE-FROM: This is used to delete all the records of a relation but it will retain the structure of that relation.

a) DELETE-FROM: This is used to delete all the records of relation.

Syntax: SQL>DELETE FROM relation_name;

Example: SQL>DELETE FROM student;

b) DELETE -FROM-WHERE: This is used to delete a selected record from a relation.

Syntax: SQL>DELETE FROM relation_name WHERE condition;

Example: SQL>DELETE FROM student WHERE sno = 2;

c) DELETE WITH JOIN

Syntax: DELETE t1 FROM t1 JOIN t2 ON join_predicate WHERE where_predicate;

Example: DELETE Table1 FROM Table1 t1 INNER JOIN Table2 t2 ON t1.Col1 = t2.Col1 WHERE t2.Col3 IN ('Two-Three','Two-Four')

4. TRUNCATE: This command will remove the data permanently. But structure will not be removed.

Syntax: SQL> TRUNCATE TABLE relation_name

Example: SQL> TRUNCATE TABLE student

Difference between Truncate & Delete:-

- ✓ By using truncate command data will be removed permanently & will not get back where as by using delete command data will be removed temporally & get back by using roll back command.
- ✓ By using delete command data will be removed based on the condition where as by using truncate command there is no condition.
- ✓ Truncate is a DDL command & delete is a DML command.

Syntax: TRUNCATE TABLE <Table name>

Example: TRUNCATE TABLE student;

5. INSERT INTO SELECT: The INSERT INTO SELECT statement copies data from one table and inserts it into another table.

- INSERT INTO SELECT requires that data types in source and target tables match
- The existing records in the target table are unaffected

Syntax: SQL>INSERT INTO table2 SELECT * FROM table1 WHERE condition;

Or

INSERT INTO table2 (column1, column2, column3, ...)
SELECT column1, column2, column3, ... FROM table1 WHERE condition;

Example: SQL> INSERT INTO Customers (CustomerName, City, Country)
SELECT SupplierName, City, Country FROM Suppliers;

6. SELECT INTO: The SELECT INTO statement copies data from one table into a new table.

Syntax: SQL> SELECT * INTO newtable [IN externaldb] FROM oldtable
WHERE condition;
Or

SELECT column1, column2, column3, ... INTO newtable [IN externaldb] FROM oldtable WHERE condition;

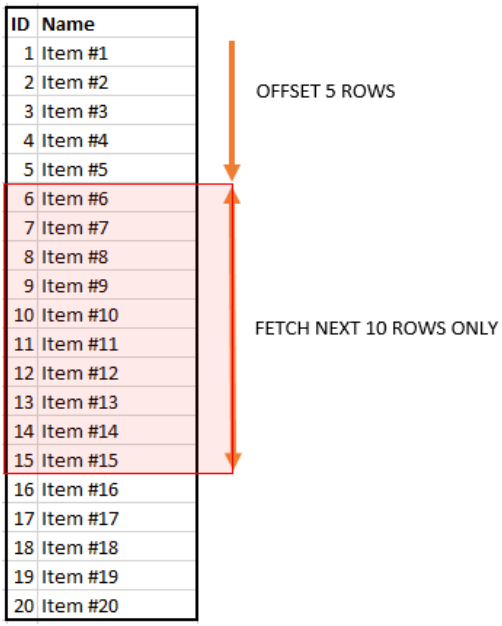
Example: SQL> SELECT * INTO CustomersBackup2017 FROM Customers;

7. SELECT with OFFSET FETCH

Syntax: ORDER BY column_list [ASC |DESC] OFFSET offset_row_count {ROW | ROWS} FETCH {FIRST | NEXT} fetch_row_count {ROW | ROWS} ONLY
In this syntax:

- The OFFSET clause specifies the number of rows to skip before starting to return rows from the query. The offset_row_count can be a constant, variable, or parameter that is greater or equal to zero.
- The FETCH clause specifies the number of rows to return after the OFFSET clause has been processed. The offset_row_count can a constant, variable or scalar that is greater or equal to one.
- The OFFSET clause is mandatory while the FETCH clause is optional. Also, the FIRST and NEXT are synonyms respectively so you can use them interchangeably. Similarly, you can use the FIRST and NEXT interchangeably.

Example: SELECT product_name, list_price FROM production.products ORDER BY list_price DESC, product_name OFFSET 10 ROWS FETCH FIRST 10 ROWS ONLY;



EXPERIMENT NO : 02

Title: Study & Implementation of different types of constraints

CONSTRAINTS:

Constraints are used to specify rules for the data in a table. If there is any violation between the constraint and the data action, the action is aborted by the constraint. It can be specified when the table is created (using CREATE TABLE statement) or after the table is created (using ALTER TABLE statement).

NOT NULL:

When a column is defined as NOTNULL, then that column becomes a mandatory column. It implies that a value must be entered into the column if the record is to be accepted for storage in the table.

Syntax:

CREATE TABLE Table_Name (column_name data_type (size) **NOT NULL)**;

Example:

CREATE TABLE student (sno **NUMBER (3) NOT NULL,** name **CHAR** (**10**));

UNIQUE/UNIQUE():

The purpose of a unique key is to ensure that information in the column(s) is unique i.e. a value entered in column(s) defined in the unique constraint must not be repeated across the column(s). A table may have many unique keys.

Syntax:

CREATE TABLE Table_Name(column_name data_type(size) **UNIQUE, ….**);

Example:

CREATE TABLE student (sno **NUMBER (3) UNIQUE,** name **CHAR** (**10**));

CHECK():

Specifies a condition that each row in the table must satisfy. To satisfy the constraint, each row in the table must make the condition either TRUE or unknown (due to a null).

Syntax:

CREATE TABLE Table_Name(column_name data_type(size)
CHECK(column_name condition), .);

Example:

CREATE TABLE student (sno **NUMBER (3),** name **CHAR**(**10**), class **CHAR(5), CHECK**(class **IN**('CSE','CAD','VLSI'));

PRIMARY KEY/ PRIMARY KEY():

A field which is used to identify a record uniquely. A column or combination of columns can be created as primary key, which can be used as a reference from other tables. A table contains primary key is known as Master Table.

- ✓ It must uniquely identify each record in a table.
- ✓ It must contain unique values.
- ✓ It cannot be a null field.
- ✓ It cannot be multi-port field.
- ✓ It should contain a minimum no. of fields necessary to be called unique.

Syntax:

CREATE TABLE Table_Name(column_name data_type(size) **PRIMARY KEY,** ...);

Example:

CREATE TABLE faculty (fcode **NUMBER (3) PRIMARY KEY,** fname **CHAR** (**10**));

FOREIGN KEY/ FOREIGN KEY():

It is a table level constraint. We cannot add this at column level. To reference any primary key column from other table this constraint can be used. The table in which the foreign key is defined is called a detail table. The table that defines the primary key and is referenced by the foreign key is called the master table.

Syntax:

CREATE TABLE Table_Name(column_name data_type(size)
FOREIGN KEY (column_name) **REFERENCES** table_name); /
FOREIGN KEY REFERENCES table_name(column_name);

Example:

CREATE TABLE subject (scode **NUMBER (3) PRIMARY KEY,** subname **CHAR**(**10**), fcode **NUMBER(3), FOREIGN KEY**(fcode) **REFERENCE** faculty);

Defining integrity constraints in the alter table command:

Syntax: **ALTER TABLE** Table_Name **ADD PRIMARY KEY** (column_name);
Example: **ALTER TABLE** student **ADD PRIMARY KEY** (sno);
(Or)
Syntax: **ALTER TABLE** table_name **ADD CONSTRAINT** constraint_name **PRIMARY KEY** (colname)
Example: **ALTER TABLE** student **ADD CONSTRAINT** SN **PRIMARY KEY(**SNO**)**

Dropping integrity constraints in the alter table command:

Syntax: **ALTER TABLE** Table_Name **DROP** constraint_name;

Example: **ALTER TABLE** student **DROP PRIMARY KEY**;

(or)

Syntax: **ALTER TABLE** student **DROP CONSTRAINT** constraint_name**;**

Example: **ALTER TABLE** student **DROP CONSTRAINT** SN**;**

DEFAULT:

The DEFAULT constraint is used to insert a default value into a column. The default value will be added to all new records, if no other value is specified.

Syntax:

CREATE TABLE Table_Name(col_name1,col_name2,col_name3 DEFAULT '<value>');

Example:

CREATE TABLE student (sno **NUMBER(3) UNIQUE,** name **CHAR**(**10**),address **VARCHAR(20) DEFAULT** 'Aurangabad');

LAB PRACTICE ASSIGNMENT:

WORKER_ID	FIRST_NAME	LAST_NAME	SALARY	DEPT_NAME	JOINING_DATE
1	Rana	Hamid	100000	HR	2014-02-20 09:00:00
2	Sanjoy	Saha	80000	Admin	2014-06-11 09:00:00
3	Mahmudul	Hasan	300000	HR	2014-02-20 09:00:00
4	Asad	Zaman	500000	Admin	2014-02-20 09:00:00
5	Sajib	Mia	500000	Admin	2014-06-11 09:00:00
6	Alamgir	Kabir	200000	Account	2014-06-11 09:00:00
7	Foridul	Islam	75000	Account	2014-01-20 09:00:00
8	Keshob	Ray	90000	Admin	2014-04-11 09:00:00

LAB ASSIGNMENT 1:

1. Create a table **Worker** with following schema:
   (**WORKER_ID(PK)**, **FIRST_NAME**, **LAST_NAME**, **SALARY**, **DEPT_NAME**)
2. Add a new column; **JOINING_DATE** to the existing relation.
3. Change the datatype of **SALARY.**
4. Change the name of column/field **DEPT_NAME to DEPARTMENT**.
5. Modify the column width of the **DEPARTMENT** field of EMPLOYEE table

LAB ASSIGNMENT 2:

1. Allow NULL for all columns except **WORKER_ID.**
2. Add constraints to check, while entering the **SALARY** value (i.e) **SALARY** > 100.
3. Define the field FIRST_NAME as **UNIQUE**.
4. Create a primary key constraint for the column (**WORKER_ID**). .

LAB ASSIGNMENT 3:

1. Insert at least 10 rows in the table.
2. Display all the information of WORKER table.
3. Display all the information of 1st 5 employees of WORKER table with
   FIRST_NAME+LASTNAME as FULL_NAME.
4. Display the complete record of employees working in Admin Department
5. Find the name of employees whose salary is greater than 10000
6. Write down the SQL Query to find out which age has bigger that is more than Sanjoy.
7. Update the Salary of Worker by 95000 whose ID is 8 .
8. Delete the record of employee whose FIRST_NAME is Asad.

EXPERIMENT NO : 03

Title: Implementation of different types of functions with suitable examples.

- Number Function
- Aggregate Function
- Character Function
- Conversion Function
- Date Function

NUMBER FUNCTION:

Abs(n) :Select abs(-15) from dual;

Exp(n): Select exp(4) from dual;

Power(m,n): Select power(4,2) from dual;

Mod(m,n): Select mod(10,3) from dual;

Round(m,n): Select round(100.256,2) from dual;

Trunc(m,n): ;Select trunc(100.256,2) from dual;

Sqrt(m,n);Select sqrt(16) from dual;

AGGREGATIVE FUNCTIONS: In addition to simply retrieving data, we often want to perform some computation or summarization. SQL allows the use of arithmetic expressions. We now consider a powerful class of constructs for computing aggregate values such as MIN and SUM.

1. **COUNT:** COUNT following by a column name returns the count of tuple in that column. If DISTINCT keyword is used then it will return only the count of unique tuple in the column. Otherwise, it will return count of all the tuples (including duplicates) count (*) indicates all the tuples of the column.
   Syntax: COUNT (Column name)
   Example: SELECT COUNT (Sal) FROM emp;

2. **SUM:** SUM followed by a column name returns the sum of all the values in that column.
   Syntax: SUM (Column name)
   Example: SELECT SUM (Sal) From emp;

3. **AVG:** AVG followed by a column name returns the average value of that column values.
   N.B.:NULL values are ignored.

Syntax: AVG (n1, n2...)

Example: Select AVG (sal) FROM emp;

4. **MAX:** MAX followed by a column name returns the maximum value of that column.

   Syntax: MAX (Column name)

   Example: SELECT MAX (Sal) FROM emp;

5. **MIN:** MIN followed by column name returns the minimum value of that column.

   Syntax: MIN (Column name)

   Example: SELECT MIN (Sal) FROM emp;

GROUPING DATA FROM TABLES:

There are circumstances where we would like to apply the aggregate function not only to a single set of tuples, but also to a group of sets of tuples, we specify this wish in SQL using the group by clause. The attribute or attributes given in the group by clause are used to form group. Tuples with the same value on all attributes in the group by clause are placed in one group.

Syntax: SELECT <set of fields> FROM <relation_name>
GROUP BY <field_name>;

Example: SQL> SELECT EMPNO, SUM (SALARY) FROM EMP GROUP BY
EMPNO;

GROUP BY-HAVING : The HAVING clause was added to SQL because the WHERE keyword could not be used with aggregate functions. The HAVING clause must follow the GROUP BY clause in a query and must also precede the ORDER BY clause if used.

Syntax: SELECT column_name, aggregate_function(column_name) FROM table_name
WHERE column_name operator value GROUP BY
column_name
HAVING aggregate_function(column_name) operator value;

Example : SELECT Employees.LastName, COUNT(Orders.OrderlD) AS NumberOfOrders
FROM (Orders INNER JOIN Employees
ON Orders.EmployeeID=Employees.EmployeeID) GROUP BY LastName
HAVING COUNT (Orders.OrderID) > 10;

ORDER BY: This query is used to display a selected set of fields from a relation in an ordered manner base on some field.

Syntax: SELECT <set of fields> FROM <relation_name>
ORDER BY <field_name>;

Example: SQL> SELECT empno, ename, job FROM emp ORDER BY job;

Sequence:

SELECT column_name(s)
FROM table_name
WHERE condition
GROUP BY column_name(s)
HAVING condition
ORDER BY column_name(s);

CHARACTER FUNCTION:

initcap(char) : select initcap("hello") from dual;

lower (char): select lower ('HELLO') from dual;

upper (char) :select upper ('hello') from dual;

ltrim (char,[set]): select ltrim ('cseit', 'cse') from dual;

rtrim (char,[set]): select rtrim ('cseit', 'it') from dual;

replace (char,search): select replace('jack and jue','j','bl') from dual;

STRING FUNCTIONS:

ASCII: Return the ASCII value of the first character

SQL>SELECT ASCII(CustomerName) AS NumCodeOfFirstChar FROM Customers;

Output:

CustomerName	NumCodeOfFirstChar
Alfreds Futterkiste	65

CHAR: Convers an integer ASCII code to character. The integer expression should be between 0 to 255.

SQL>SELECT CHAR(97)

Output: a

CHARINDEX: Return the starting position of the specified expression in a character string. Remember its starts from 1 not 0.

SQL>SELECT CHARINDEX('@','nurcse09@gmail.com')

Output: 9

CONCAT: Add two strings together:

SQL>SELECT CONCAT('W3Schools', '.com');

Output: W3Schools.com

CONCAT with '+': The + operator allows you to add two or more strings together.

SQL>SELECT 'W3Schools' + '.com';

Output: W3Schools.com

LEFT: Extract 3 characters from a string (starting from left):

SQL>SELECT LEFT('SQL Tutorial', 3) AS ExtractString

Output: SQL

RIGHT: Extract 3 characters from a string (starting from right):

SQL>SELECT RIGHT('SQL Tutorial', 3) AS ExtractString

Output: ial

LTRIM: Remove blanks on the left hand side of the given expression.

SQL>SELECT LTRIM(' Nur')

Output: Nur

RTRIM: Remove blanks on the right hand side of the given expression.

SQL>SELECT RTRIM('Nur ')

Output: Nur

LOWER : Converts all the character in the given character expression to lowercase letters.

SQL>SELECT LOWER('NUR')

Output: nur

UPPER: Converts all the character in the given character expression to uppercase letters.

SQL>SELECT UPPER(‘nur')

Output: NUR

REVERSE: Reverse all the characters in the given character expression to uppercase letters.

SQL>SELECT REVERSE('nur')

Output: run

LEN: Returns the count of total characters in the given string expression excluding the blanks at the end of the expression.

SQL>SELECT LEN(‘ nur')

Output: 4

SUBSTRING: Return substring (part of string) from given expression.

SQL>SELECT SUBSTRING('nurcse09@gmail.com',1,8)

Output: nurcse09

REPLICATE: Replace the given string to specified number of times.

SQL>SELECT REPLICATE(‘Nur ',2)

Output: Nur Nur

REPLACE: Replaces all occurrences of a specified string value with another string value.

SQL>SELECT REPLACE(‘nurcse09@gmail.com ‘,’ com’, ’net’)

Output: nurcse09@gmail.net

SPACE(Number_Of_Spaces) : Returns number of spaces, specified by Number_Of_Spaces argument.

Example: The SPACE(5) function, inserts 5 spaces between FirstName and LastName

SQL>Select FirstName + **SPACE**(5) + LastName as FullName From tblEmployee

DATE FUNCTIONS:

ISDATE() - Checks if the given value, is a valid date, time, or datetime. Returns 1 for success, 0 for failure.
SQL>Select ISDATE('2012-08-31 21:02:04.167') -- returns 1

DAY() : Returns the 'Day number of the Month' of the given date
SQL>Select DAY('01/31/2012') -- Returns 31

MONTH() : Returns the 'Month number of the year' of the given date
SQL>Select Month('01/31/2012') -- Returns 1

YEAR() : Returns the 'Year number' of the given date
SQL>Select Year('01/31/2012') -- Returns 2012

GETDATE() : Return the current database system date and time:
SQL>SELECT GETDATE() -- returns 2020-07-08 18:02:55.173

CURRENT_TIMESTAMP : Return the current date and time:
SQL> SELECT CURRENT_TIMESTAMP -- returns 2020-07-08 18:04:37.877

SYSDATETIME() : Return the date and time of the SQL Server:
SQL> SELECT SYSDATETIME() -- returns 2020-07-08 18:06:13.1440795

GETUTCDATE(): Return the current UTC date and time:
SQL> SELECT GETUTCDATE() -- returns 2020-07-08 17:07:23.740

DATEADD (**datepart, NumberToAdd, date**) : Returns the DateTime, after adding specified NumberToAdd, to the datepart specified of the given date.

SQL> Select DateAdd(DAY, 20, '2012-08-30 19:45:31.793') -- Returns 2012-09-19 19:45:31.793
SQL>Select DateAdd(DAY, -20, '2012-08-30') -- Returns 2012-08-10 00:00:00.000
SQL>SELECT DATEADD(month, 2, '2017/08/25') Returns 2017-10-25 00:00:00.000

DATEDIFF(datepart, startdate, enddate) : Returns the count of the specified datepart boundaries crossed between the specified startdate and enddate.

SQL>Select DATEDIFF(MONTH, '11/30/2005', '01/31/2006') -- returns 2

SQL>Select DATEDIFF(DAY, '11/30/2005', '01/31/2006') -- returns 62

SQL>SELECT DATEDIFF(year, '2017/08/25', '2011/08/25') – returns -6

DATEPART(interval, date): The DATEPART() function returns a specified part of a date.

SQL> SELECT DATEPART(year, '2017/08/25') returns – 2017

SQL> SELECT DATEPART(month, '2017/08/25') returns -- 8

SQL> SELECT DATEPART(week, '2017/08/25') returns -- 34

CONVERSION FUNCTIONS:

CAST(expression AS datatype(length)): The CAST() function converts a value (of any type) into a specified datatype.

SQL>SELECT CAST(25.65 AS int) -- returns 25

SQL>SELECT CAST('2017-08-25' AS datetime) -- returns 2017-08-25 00:00:00.000

CONVERT(data_type(length-optional), expression, style): The CONVERT() function converts a value (of any type) into a specified datatype.

SQL>SELECT CONVERT(int, 25.65) -- returns 25

SQL>SELECT CONVERT(varchar, '2017-08-25', 101) -- returns 2017-08-25

Without century	With century	Input/Output	Standard
0	100	mon dd yyyy hh:miAM/PM	Default
1	101	mm/dd/yyyy	US
2	102	yyyy.mm.dd	ANSI
3	103	dd/mm/yyyy	British/French
4	104	dd.mm.yyyy	German
5	105	dd-mm-yyyy	Italian
6	106	dd mon yyyy	-
7	107	Mon dd, yyyy	-
8	108	hh:mm:ss	-
9	109	mon dd yyyy hh:mi:ss:mmmAM (or PM)	Default + millisec
10	110	mm-dd-yyyy	USA

11	111	yyyy/mm/dd	Japan
12	112	yyyymmdd	ISO
13	113	dd mon yyyy hh:mi:ss:mmm	Europe (24 hour clock)>
14	114	hh:mi:ss:mmm	24 hour clock
20	120	yyyy-mm-dd hh:mi:ss	ODBC canonical (24 hour clock)

Title : Implementation of different types of operators in SQL.

- Arithmetic Operator
- Logical Operator
- Comparision Operator
- Special Operator
- Set Operator

ARIHMETIC OPERATORS:

(+) : Addition - Adds values on either side of the operator .

(-):Subtraction - Subtracts right hand operand from left hand operand .

(*):Multiplication - Multiplies values on either side of the operator .

(/):Division - Divides left hand operand by right hand operand .

(^):Power- raise to power of .

(%):Modulus - Divides left hand operand by right hand operand and returns remainder.

LOGICAL OPERATORS:

AND : The AND operator allows the existence of multiple conditions in an SQL statement's WHERE clause. The AND operator displays a record if all the conditions separated by AND are TRUE.

SQL>SELECT * FROM Customers WHERE Country='Germany' AND City='Berlin';

OR: The OR operator is used to combine multiple conditions in an SQL statement's WHERE clause. The OR operator displays a record if any of the conditions separated by OR is TRUE.

SQL>SELECT * FROM Customers WHERE City='Berlin' OR City='München';

NOT: The NOT operator reverses the meaning of the logical operator with which it is used.

SQL>SELECT * FROM Customers WHERE NOT Country='Germany';

Combining AND, OR and NOT

SQL>SELECT * FROM CustomersWHERE Country='Germany' AND (City='Berlin' OR Cit y='München');

Eg: NOT EXISTS, NOT BETWEEN, NOT IN, etc. This is a negate operator.

COMPARISION OPERATORS:

(=): Checks if the values of two operands are equal or not, if yes then condition becomes true.

(!=): Checks if the values of two operands are equal or not, if values are not equal then condition becomes true.

(< >): Checks if the values of two operands are equal or not, if values are not equal then condition becomes true.

(>): Checks if the value of left operand is greater than the value of right operand, if yes then condition becomes true

(<):Checks if the value of left operand is less than the value of right operand, if yes then condition becomes true.

(>=): Checks if the value of left operand is greater than or equal to the value of right operand, if yes then condition becomes true.

(<=): Checks if the value of left operand is less than or equal to the value of right operand, if yes then condition becomes true.

SPECIAL OPERATOR:

BETWEEN: The BETWEEN operator is used to search for values that are within a set of values, given the minimum value and the maximum value.

The BETWEEN operator is inclusive: begin and end values are included.

a) **BETWEEN Syntax**

SELECT column_name(s)
FROM table_name
WHERE column_name BETWEEN value1 AND value2;

Example: Find the names of all instructors with salary between $90,000 and $100,000 (that is, ³ $90,000 and £ $100,000)

SQL> select name from instructor where salary between 90000 and 100000

b) NOT BETWEEN Example

To display the products outside the range of the previous example, use NOT BETWEEN:

Example:

```
SELECT * FROM Products
WHERE Price NOT BETWEEN 10 AND 20;
```

c) BETWEEN with IN Example

The following SQL statement selects all products with a price BETWEEN 10 and 20. In addition; do not show products with a CategoryID of 1,2, or 3:

Example:

```
SELECT * FROM Products
WHERE (Price BETWEEN 10 AND 20)   AND  CategoryID NOT  IN (1,2,3);
```

COALESCE(val1, val2, ..., val_n) : The COALESCE() function returns the first non-null value in a list.

SQL> SELECT COALESCE(NULL, 1, 2, 'W3Schools.com') -- returns 1

ISNULL(expression, value): The ISNULL() function returns a specified value if the expression is NULL. If the expression is NOT NULL, this function returns the expression.

SQL> SELECT ISNULL('Hello', 'W3Schools.com') -- returns Hello

SQL> SELECT ISNULL(NULL, 500) -- returns 500

CASE….END : Can replace NULL values with specified name.

SQL>Select Employee_Name, CASE when Employee_Name is null then 'Owner' ELSE Employee_Name END as Manager_Name From Employee_Detail

ALL: The ALL operator returns TRUE if all of the subquery values meet the condition. The following SQL statement returns TRUE and lists the product names if ALL the records in the OrderDetails table has quantity = 10 (so, this example will return FALSE, because not ALL records in the OrderDetails table has quantity = 10):

SQL>SELECT ProductName FROM Products WHERE ProductID
= ALL (SELECT ProductID FROM OrderDetails WHERE Quantity = 10);

ANY: The ANY operator returns true if any of the subquery values meet the condition.

The following SQL statement returns TRUE and lists all the product names if it finds ANY records in the OrderDetails table that quantity = 10:

SQL>SELECT ProductName FROM Products WHERE ProductID

= ANY (SELECT ProductID FROM OrderDetails WHERE Quantity = 10);

(It can return more than 1 records)

LIKE: The LIKE operator is used in a WHERE clause to search for a specified pattern in a column.It allows to use percent sign(%) and underscore (_) to match a given string pattern.

LIKE Operator	Description
WHERE CustomerName LIKE 'a%'	Finds any values that start with "a"
WHERE CustomerName LIKE '%a'	Finds any values that end with "a"
WHERE CustomerName LIKE '%or%'	Finds any values that have "or" in any position
WHERE CustomerName LIKE '_r%'	Finds any values that have "r" in the second position
WHERE CustomerName LIKE 'a_%'	Finds any values that start with "a" and are at least 2 characters in length
WHERE CustomerName LIKE 'a__%'	Finds any values that start with "a" and are at least 3 characters in length
WHERE ContactName LIKE 'a%o'	Finds any values that start with "a" and ends with "o"

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

The following SQL statement selects all customers that are located in "Germany", "France" or "UK":

SQL> SELECT * FROM Customers WHERE Country IN ('Germany', 'France', 'UK');

SQL> SELECT * FROM Customers

The following SQL statement selects all customers that are from the same countries as the suppliers:

WHERE Country IN (SELECT Country FROM Suppliers);

EXIST: The EXISTS operator is used to search for the presence of a row in a specified table that meets certain criteria.

C) EXISTS vs. IN example
The following statement uses the IN operator to find the orders of the customers from San Jose:

```
SELECT
   *
FROM
   sales.orders
WHERE
   customer_id IN (
      SELECT
         customer_id
      FROM
         sales.customers
      WHERE
         city = 'San Jose'
   )
ORDER BY
   customer_id,
   order_date;
```

The following statement uses the EXISTS operator that returns the same result:

```
SELECT
   *
FROM
   sales.orders o
WHERE
   EXISTS (
      SELECT
         customer_id
      FROM
         sales.customers c
      WHERE
         o.customer_id = c.customer_id
      AND city = 'San Jose'
   )
ORDER BY
   o.customer_id,
   order_date;
```

SET OPERATORS:

The Set operator combines the result of 2 queries into a single result. The following are the operators:

- **Union**
- **Union all**
- **Intersect**
- **Except**

UNION: Returns all distinct rows selected by both the queries

Output: = Records only in Query one + records only in Query two +
A single set of records which is common in the both Queries.

Syntax:

SELECT columnname, columname FROM tablename 1

UNION

SELECT columnname, columnname From tablename2;

Union all: Returns all rows selected by either query including the duplicates.

INTERSECT: Returns rows selected that are common to both queries.

Syntax:

SELECT columnname, columnname FROM tablename 1

INTERSECT

SELECT columnname, columnname FROM tablename 2;

Output =A single set of records which are common in both Queries

EXCEPT: Returns all distinct rows selected by the first query and are not by the second

output= records only in Query one

Syntax:

SELECT columnname, columnname FROM tablename ;

EXCEPT

SELECT columnname, columnname FROM tablename ;

LAB ASSIGNMENT:

Sample Table – Worker

WORKER_ID	FIRST_NAME	LAST_NAME	SALARY	JOINING_DATE	DEPARTMENT
1	Rana	Hamid	100000	2014-02-20 09:00:00	HR
2	Sanjoy	Saha	80000	2014-06-11 09:00:00	Admin
3	Mahmudul	Hasan	300000	2014-02-20 09:00:00	HR
4	Asad	Zaman	500000	2014-02-20 09:00:00	Admin
5	Sajib	Mia	500000	2014-06-11 09:00:00	Admin
6	Alamgir	Kabir	200000	2014-06-11 09:00:00	Account
7	Foridul	Islam	75000	2014-01-20 09:00:00	Account
8	Keshob	Ray	90000	2014-04-11 09:00:00	Admin

1. Write an SQL query to print first three characters of FIRST_NAME from Worker table.
2. Write an SQL query to print details of the Workers who have joined from Feb 2014 to March 2014.
3. Write an SQL query to print details of the Workers who have served for at least 6 months.
4. Write an SQL query to update all worker salary whose title is manager.
5. Write an SQL query to update all worker bonus 10% whose joining_date before '2014-04-11 09:00:00' otherwise bonus update 5% and also check department name is 'Admin'.
6. Write an SQL query to delete all workers who have not taken any bonus.
7. Write an SQL query to print details for Workers with the first name "Rana" and "Sajib" from Worker table.
8. Write an SQL query to print details of workers excluding first names, "Rana" and "Sajib" from Worker table.
9. Write an SQL query to print details of the Workers whose FIRST_NAME contains 'a'.
10. Write an SQL query to print details of the Workers whose FIRST_NAME starts with 'k'.
11. Write an SQL query to print details of the Workers whose FIRST_NAME ends with 'r' and contains seven alphabets.
12. Write an SQL query to find the position of the alphabet ('n') in the FIRST_NAME column 'Sanjoy' from Worker table.
13. Find the average salary of employees for each department.
14. List all the employees who have maximum or minimum salary in each department
15. Write an SQL query to find the position of the alphabet ('r') in the **FIRST_NAME** column 'Rana' from Worker table.
16. Write an SQL query to print the **FIRST_NAME** from Worker table after removing white spaces from the right side.
17. Write an SQL query that fetches the unique values of **FIRST_NAME** from Worker table and prints its length.
18. Write an SQL query to print the **FIRST_NAME** from Worker table after replacing 'a' with 'A'.

EXPERIMENT NO : 04

Title: Implementation of Join and subqueries

Join is a query that is used to combine rows from two or more tables, views, or materialized views. It retrieves data from multiple tables and creates a new table.

Join Conditions

There may be at least one join condition either in the FROM clause or in the WHERE clause for joining two tables. It compares two columns from different tables and combines pair of rows, each containing one row from each table, for which join condition is true.

Types of Joins

- Inner Joins (Simple Join)
- Outer Joins
    - Left Outer Join (Left Join)
    - Right Outer Join (Right Join)
    - Full Outer Join (Full Join)
- Equi joins
- Non-equi join
- Self Joins
- Cross Joins (Cartesian Products)
- Antijoins

1. INNER JOIN : Inner Join is the simplest and most common type of join. It is also known as simple join. It returns all rows from multiple tables where the join condition is met.

Syntax:

```
SELECT columns   FROM table1
INNER JOIN table2  ON table1.column = table2.column;
```

2. **NATURAL JOIN**: A natural join is such a join that compares the common columns of both tables with each other. The SQL NATURAL JOIN is a type of EQUI JOIN and is structured in such a way that, columns with the same name of associated tables will appear once only.

Syntax:

```
SELECT  columns/*  FROM table1
NATURAL JOIN table2  ;
```

3. **LEFT OUTER JOIN:** Left Outer Join returns all rows from the left (first) table specified in the ON condition and only those rows from the right (second) table where the join condition is met.

Syntax:
SELECT columns FROM table1
LEFT [OUTER] JOIN table2 ON table1.column = table2.column;

Example:
SELECT suppliers.supplier_id, suppliers.supplier_name, order1.order_number
FROM suppliers LEFT OUTER JOIN order1 ON suppliers.supplier_id = order1.supplier_id;

4. **RIGHT OUTER JOIN:** The Right Outer Join returns all rows from the right-hand table specified in the ON condition and only those rows from the other table where the join condition is met.

Syntax
SELECT columns FROM table1
RIGHT [OUTER] JOIN table2 ON table1.column = table2.column;

Example:
SELECT order1.order_number, order1.city, suppliers.supplier_name FROM suppliers
RIGHT OUTER JOIN order1 ON suppliers.supplier_id = order1.supplier_id;

5. **FULL OUTER JOIN :**The Full Outer Join returns all rows from the left hand table and right hand table. It places NULL where the join condition is not met.

Syntax
SELECT columns FROM table1
FULL [OUTER] JOIN table2 ON table1.column = table2.column;

Example:
SELECT suppliers.supplier_id, suppliers.supplier_name, order1.order_number
FROM suppliers FULL OUTER JOIN order1
ON suppliers.supplier_id = order1.supplier_id;

6. **EQUI JOIN:** Equi join returns the matching column values of the associated tables. It uses a comparison operator in the WHERE clause to refer equality.

Syntax
SELECT column_list FROM table1, table2....
WHERE table1.column_name = table2.column_name;

Equijoin also can be performed by using JOIN keyword followed by ON keyword and then specifying names of the columns along with their associated tables to check equality.

Syntax
SELECT * FROM table1 JOIN table2 [ON (join_condition)]

Example:
SELECT agents.agent_city,customer.last_name, customer.first_name FROM agents,customer
WHERE agents.agent_id=customer.customer_id;

7. **NON-EQUI JOIN:** The nonequijoins is such a join which match column values from different tables based on an inequality (instead of the equal sign like **>, <, >=, <=**) expression. The value of the join column in each row in the source table is compared to the corresponding values in the target table. A match is found if the expression based on an inequality operator used in the join, evaluates to true.

Syntax:
SELECT table1.column, table2.column FROM table1
[JOIN table2 ON (table1.column_name < table2.column_name)]|
[JOIN table2 ON (table1.column_name > table2.column_name)]|
[JOIN table2 ON (table1.column_name <= table2.column_name)]|
[JOIN table2 ON (table1.column_name >= table2.column_name)]|
[JOIN table2 ON (table1.column BETWEEN table2.col1 AND table2.col2)]|

SELECT * FROM table_name1, table_name2
WHERE table_name1.column [> | < | >= | <=] table_name2.column;

8. **SELF JOIN**: Self Join is a specific type of Join. In Self Join, a table is joined with itself (Unary relationship). A self join simply specifies that each rows of a table is combined with itself and every other row of the table.

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

Example:
SELECT a.name, b.age, a.SALARY
FROM CUSTOMERS a, CUSTOMERS b
WHERE a.SALARY < b.SALARY;

9. **CROSS JOIN**: The CROSS JOIN specifies that all rows from first table join with all of the rows of second table. If there are "x" rows in table1 and "y" rows in table2 then the cross join result set have x*y rows. It normally happens when no matching join columns are specified.

In simple words you can say that if two tables in a join query have no join condition, then the Oracle returns their Cartesian product.

Syntax

SELECT * FROM table1 CROSS JOIN table2;

Or

SELECT * FROM table1, table2

Example:

```
SELECT emp_no,emp_name,job_name,dep_name,location FROM emp_mast
CROSS JOIN dep_mast;
```

10. ANTI JOIN: Anti-join is used to make the queries run faster. It is a very powerful SQL construct Oracle offers for faster queries.

Anti-join between two tables returns rows from the first table where no matches are found in the second table. It is opposite of a semi-join. An anti-join returns one copy of each row in the first table for which no match is found.

Anti-joins are written using the **NOT EXISTS** or **NOT IN** constructs.

Example:

```
SELECT departments.department_id, departments.department_name FROM department
s WHERE NOT EXISTS
(
        SELECT 1
        FROM customer
        WHERE customer.department_id = departments.department_id
)
ORDER BY departments.department_id;
```

LAB ASSIGNMENT:

Sample Table – Worker

WORKER_ID	FIRST_NAME	LAST_NAME	SALARY	JOINING_DATE	DEPARTMENT
1	Rana	Hamid	100000	2014-02-20 09:00:00	HR
2	Sanjoy	Saha	80000	2014-06-11 09:00:00	Admin
3	Mahmudul	Hasan	300000	2014-02-20 09:00:00	HR
4	Asad	Zaman	500000	2014-02-20 09:00:00	Admin
5	Sajib	Mia	500000	2014-06-11 09:00:00	Admin
6	Alamgir	Kabir	200000	2014-06-11 09:00:00	Account
7	Foridul	Islam	75000	2014-01-20 09:00:00	Account
8	Keshob	Ray	90000	2014-04-11 09:00:00	Admin

Sample Table – Bonus

WORKER_REF_ID	BONUS_DATE	BONUS_AMOUNT
1	2019-02-20	5000
2	2019-06-11	3000
3	2019-02-20	4000
4	2019-02-20	4500
5	2019-06-11	3500
6	2019-06-12	NULL

Sample Table – Title

WORKER_REF_ID	WORKER_TITLE	AFFECTED_FROM
1	Manager	2019-02-20
2	Executive	2019-06-11
8	Executive	2019-06-11
5	Manager	2019-06-11
4	Asst. Manager	2019-06-11
7	Executive	2019-06-11
6	Lead	2019-06-11
3	Lead	2019-06-11

1. List all the employees except ‘Manager’ & ‘Asst. Manager’.
2. List the workers in the ascending order of Designations of those joined after April 2014.
3. Write an SQL query to fetch the number of employees working in the department ‘Admin’.
4. Write an SQL query to fetch worker names with salaries >= 50000 and <= 100000.
5. Write an SQL query to fetch the no. of workers for each department in the descending order.
6. Write an SQL query to print details of the Workers who are also Managers.
7. Write an SQL query to show only odd rows from a table.
8. Write an SQL query to show only even rows from a table.
9. Write an SQL query to clone a new table from another table.
10. Write an SQL query to show the current date and time.
11. Write an SQL query to show the top n (say 10) records of a table with Name and Designation.
12. Write an SQL query to determine the nth (say n=5) highest salary from a table.
13. Write an SQL query to fetch the list of employees with the same salary.
14. Write an SQL query to show the second highest salary from a table.
15. Write an SQL query to fetch the first 50% records from a table.
16. Write an SQL query to fetch the departments that have less than five people in it.
17. Write an SQL query to show all departments along with the number of people in there.
18. Write an SQL query to show the last record from table.
19. Write an SQL query to fetch the first row of a table.
20. Write an SQL query to fetch the last five records from table.
21. Write an SQL query to print the name of employees having the highest salary in each department.
22. Write an SQL query to fetch three max salaries from table.

SUBQUERIES: The query within another is known as a sub query. A statement containing sub query is called parent statement. The rows returned by sub query are used by the parent statement or in other words A subquery is a SELECT statement that is embedded in a clause of another SELECT statement You can place the subquery in a number of SQL clauses:

- WHERE clause
- HAVING clause
- FROM clause
- OPERATORS(IN.ANY,ALL,<,>,>=,<= etc..)

Types:

1. **Sub queries that return several values**

Sub queries can also return more than one value. Such results should be made use along with the operators in and any.

2. **Multiple queries**

Here more than one sub query is used. These multiple sub queries are combined by means of 'and' & 'or' keywords.

3. **Correlated sub query**

A sub query is evaluated once for the entire parent statement whereas a correlated Sub query is evaluated once per row processed by the parent statement.

LAB ASSIGNMENT 01:

TID	FirstName	LastName	Dept	Age	Salary
1	Mizanur	Rahman	CSE	28	35000
2	Delwar	Hossain	CSE	26	33000
3	Shafiul	Islam	EEE	24	30000
4	Faisal	Imran	CSE	30	50000
5	Ahsan	Habib	English	28	28000

deptID	deptName	location
1	CSE	Talaimari
2	EEE	Talaimari
3	English	Kazla
4	BBA	Talaimari

1. Update the Salary of Teacher by 15% whose DeptName is 'CSE, otherwise update by 10% Salary.
2. Write a query to insert/copy the values of all attributes from one table to another using (ID in) subquery.
3. Write a query to find firstname and lastname as fullname , age whose salary is maximum.
4. Write a query to find firstname, age,dept whose age is between 23 to 27.

5. Write a query to find TID,firstname whose salary is less than average salary.
6. Write a query to update Dept by 'English' where Dept is 'EEE' using subquery.
7. Write a query to update salary by multiplying the salary by 100 where salary is greater than 5000 using subquery..
8. Write a query to find the name that starts with 'k/s' using a subquery.
9. Find the Firstname,salary for all the teachers of CSE who have a higher salary than Delwar Hossain using subquery.
10. Find out the id,names of all teachers who belong to the same department as the teacher 'Mizanur' .
11. Find TID, salary, deptID whose salary is greater than average salary
12. Find min salary from Teacher for each department where min salary is less than average salary
13. Find firstname,lastname,Dept where location name is kajla using subquery.
14. Write a query to find the TID,firsname,salary where the length of the firstname is at least 6.

EXPERIMENT NO: 05

Title : Study & Implementation of

- Views
- Materialized View

VIEW: 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 is simply any SELECT query that has been given a name and saved in the database. For this reason, a view is sometimes called a named query or a stored query.

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 View:

Syntax:

CREATE VIEW <view_name> AS SELECT <set of fields> FROM relation_name WHERE (Condition)

Example:

SQL> CREATE VIEW EMPLOYEE AS SELECT empno,ename,job FROM EMP WHERE job = ‘clerk’;

SQL> View created.

Now, you can query employee in similar way as you query an actual table. Following is the example:

SQL > SELECT * FROM EMPLOYEE

The WITH CHECK OPTION:

The WITH CHECK OPTION is a CREATE VIEW statement option. The purpose of the WITH CHECK OPTION is to ensure that all UPDATE and INSERTs satisfy the condition(s) in the view definition.

If they do not satisfy the condition(s), the UPDATE or INSERT returns an error.

Example:

CREATE VIEW CUSTOMERS_VIEW AS SELECT name, age FROM CUSTOMERS WHERE age IS NOT NULL WITH CHECK OPTION;

UPDATING A VIEW : A view can updated by using the following syntax :

Syntax :
CREATE OR REPLACE VIEW view_name AS SELECT column_name(s)

FROM table_name WHERE condition

DELETING ROWS INTO A VIEW: Rows of data can be deleted from a view. The same rules that apply to the UPDATE and INSERT commands apply to the DELETE command.

Following is an example to delete a record having AGE= 22.

Syntax :
SQL > DELETE FROM EMPLOYEE WHERE AGE= 22.

DROPPING A VIEW: A view can deleted with the DROP VIEW command.

Syntax: DROP VIEW <view_name> ;

Materialized View (MV)

A materialized view in Oracle is a database object that contains the results of a query. It stores data physically and get updated periodically. While querying Materialized View, it gives data directly from Materialized View and not from table.

What are the uses of materialized view?

1. Better Performance with complex joins

If our join queries are using many tables, group by and aggregate functions on millions of rows, then it takes much time to execute.

In such scenarios, Materialized views help us to get data faster. Materialized views are physically exist in database. Whenever the base table is updated the Materialized view gets updated. Once MV is updated, query on that single MV gives very fast results.

2. Data Warehouses

In data warehouses, materialized views can be used to pre-compute and store aggregated data such as sum of sales. Materialized views in these environments are typically referred to as summaries since they store summarized data.

They can also be used to pre-compute joins with or without aggregations. So a materialized view is used to eliminate overhead associated with expensive joins or aggregations for a large or important class of queries.

Basic Syntax

```
CREATE VIEW mv_my_view

WITH SCHEMABINDING

AS

SELECT col1, sum(col2) as total

FROM <table_name>

GROUP BY col1;

GO //Batch separator

CREATE UNIQUE CLUSTERED INDEX xv

ON mv_my_view (col1);
```

Example:

```
CREATE VIEW ContractJobs
WITH SCHEMABINDING
AS
SELECT
   J.JobId,
   J.ContractNumber,
   SJ.JobName,
   SJ.StandardPrice,
   C.ContractValue
FROM
   dbo.Jobs J
INNER JOIN
   dbo.StandardJobs SJ
ON
   J.StandardJobId = SJ.StandardJobId
INNER JOIN
   dbo.Contracts C
ON
   J.ContractNumber = C.ContractNumber
WHERE
   C.RenewalDate IS NOT NULL
```

Notice the option WITH SCHEMABINDING after the view name. The rest is the same as a regular view.

Creating a Unique Clustered Index

```
CREATE UNIQUE CLUSTERED INDEX IX_ContractJobs_JobId ON ContractJobs (JobId)
```

LAB ASSIGNMENT:

Sample table: salesman

salesman_id	name	city	commission
5001	James Hoog	New York	0.15
5002	Nail Knite	Paris	0.13
5005	Pit Alex	London	0.11
5006	Mc Lyon	Paris	0.14
5003	Lauson Hen	Berlin	0.12
5007	Paul Adam	Rome	0.13

Sample table: customer

customer_id	cust_name	city	grade	salesman_id
3002	Nick Rimando	New York	100	5001
3005	Graham Zusi	California	200	5002
3001	Brad Guzan	London	300	5005
3004	Fabian Johns	Paris	300	5006
3007	Brad Davis	New York	200	5001
3009	Geoff Camero	Berlin	100	5003
3008	Julian Green	London	300	5002
3003	Jozy Altidor	Moscow	200	5007

Sample table: orders

ord_no	purch_amt	ord_date	customer_id	salesman_id
70001	150.5	2012-10-05	3005	5002
70009	270.65	2012-09-10	3001	5005
70002	65.26	2012-10-05	3002	5001
70004	110.5	2012-08-17	3009	5003
70007	948.5	2012-09-10	3005	5002
70005	2400.6	2012-07-27	3007	5001
70008	5760	2012-09-10	3002	5001
70010	1983.43	2012-10-10	3004	5006
70003	2480.4	2012-10-10	3009	5003
70012	250.45	2012-06-27	3008	5002
70011	75.29	2012-08-17	3003	5007
70013	3045.6	2012-04-25	3002	5001

1. Write a query to create a view for those salesmen belongs to the city New York.
2. Write a query to create a view for all salesmen with columns salesman_id, name and city.
3. Write a query to find the salesmen of the city New York who achieved the commission more than 13%.
4. Write a query to create a view to getting a count of how many customers we have at each level of a grade.
5. Write a query to create a view to keeping track the number of customers ordering, number of salesmen attached, average amount of orders and the total amount of orders in a day.
6. Write a query to create a view that shows for each order the salesman and customer by name.
7. Write a query to create a view that finds the salesman who has the customer with the highest order of a day.
8. Write a query to create a view that shows all of the customers who have the highest grade.
9. Write a query to create a view that shows the number of the salesman in each city.
10. Write a query to create a view that shows the average and total orders for each salesman after his or her name. (Assume all names are unique)
11. Write a query to create a view that shows each salesman with more than one customers.
12. Write a query to create a view that shows all matches of customers with salesman such that at least one customer in the city of customer served by a salesman in the city of the salesman.
13. Write a query to create a view that shows the number of orders in each day.
14. Write a query to create a view that finds the salesmen who issued orders on October 10th, 2012.
15. Write a query to create a view that finds the salesmen who issued orders on either August 17th, 2012 or October 10th, 2012.

EXPERIMENT NO : 07

Title: Implementation of Indexing & Cascading

INDEXING: An **index** is an ordered set of pointers to the data in a table. It is based on the data values in one or more columns of the table. SQL Base stores indexes separately from tables. When multiple columns are included in the index then it is called composite index.

An index provides two benefits:

- It improves performance because it makes data access faster.
- It ensures uniqueness. A table with a unique index cannot have two rows with the same values in the column or columns that form the index key.

Syntax:

CREATE INDEX <index_name> on <table_name> (attrib1,attrib 2.. ..attrib n);

Example:

CREATE INDEX id1 on emp(empno,dept_no);

DROPPING AN INDEX:

Syntax:

DROP INDEX <index_name> ;

FOREIGN KEYS WITH CASCADE DELETE/UPDATE

DELETE CASCADE: When we create a foreign key using this option, it deletes the referencing rows in the child table when the referenced row is deleted in the parent table which has a primary key.

UPDATE CASCADE: When we create a foreign key using UPDATE CASCADE the referencing rows are updated in the child table when the referenced row is updated in the parent table which has a primary key.

A foreign key with a cascade delete/update can be defined in either a CREATE TABLE statement or an ALTER TABLE statement.

Using a CREATE TABLE statement

Syntax

The syntax for creating a foreign key with cascade delete using a CREATE TABLE statement is:

```
CREATE TABLE table_name
(
  column1 datatype null/not null,
  column2 datatype null/not null,
  ...

  CONSTRAINT fk_column
   FOREIGN KEY (column1, column2, ... column_n)
   REFERENCES parent_table (column1, column2, ... column_n)
   ON DELETE CASCADE
);
```

Example

Let's look at an example of how to create a foreign key with cascade delete using the CREATE TABLE statement in Oracle/PLSQL.

For example:

```
CREATE TABLE supplier
( supplier_id numeric(10) not null,
  supplier_name varchar2(50) not null,
  contact_name varchar2(50),
  CONSTRAINT supplier_pk PRIMARY KEY (supplier_id)
);


CREATE TABLE products
( product_id numeric(10) not null,
  supplier_id numeric(10) not null,
  CONSTRAINT fk_supplier
  FOREIGN KEY (supplier_id)
  REFERENCES supplier(supplier_id)
  ON DELETE CASCADE
);
```

In this example, we've created a primary key on the supplier table called supplier_pk. It consists of only one field - the supplier_id field. Then we've created a foreign key called fk_supplier on the products table that references the supplier table based on the supplier_id field.

Because of the cascade delete, when a record in the supplier table is deleted, all records in the products table will also be deleted that have the same supplier_id value.

We could also create a foreign key (with a cascade delete) with more than one field as in the example below:

```
CREATE TABLE supplier
( supplier_id numeric(10) not null,
  supplier_name varchar2(50) not null,
  contact_name varchar2(50),
  CONSTRAINT supplier_pk PRIMARY KEY (supplier_id, supplier_name)
);


CREATE TABLE products
( product_id numeric(10) not null,
  supplier_id numeric(10) not null,
  supplier_name varchar2(50) not null,
  CONSTRAINT fk_supplier_comp
  FOREIGN KEY (supplier_id, supplier_name)
  REFERENCES supplier(supplier_id, supplier_name)
  ON DELETE CASCADE
);
```

In this example, our foreign key called fk_foreign_comp references the supplier table based on two fields - the supplier_id and supplier_name fields.

The cascade delete on the foreign key called fk_foreign_comp causes all corresponding records in the products table to be cascade deleted when a record in the supplier table is deleted, based on supplier_id and supplier_name.

ON UPDATE CASCADE:

```
CREATE TABLE Countries
(CountryID INT PRIMARY KEY,
CountryName VARCHAR(50),
CountryCode VARCHAR(3))


CREATE TABLE States
(StateID INT PRIMARY KEY,
StateName VARCHAR(50),
StateCode VARCHAR(3),
CountryID INT)

GO

INSERT INTO Countries VALUES (1,'United States','USA')
INSERT INTO Countries VALUES (2,'United Kingdom','UK')
INSERT INTO States VALUES (1,'Texas','TX',1)
INSERT INTO States VALUES (2,'Arizona','AZ',1)

GO
```

```
ALTER TABLE [dbo].[States]  WITH CHECK ADD  CONSTRAINT [FK_States_Countries
] FOREIGN KEY([CountryID])
REFERENCES [dbo].[Countries] ([CountryID])
ON UPDATE CASCADE
GO

ALTER TABLE [dbo].[States] CHECK CONSTRAINT [FK_States_Countries]
GO
```

*Now update CountryID in the Countries for a row which also updates the referencing rows in the child table States.

Using an ALTER TABLE statement

Syntax

The syntax for creating a foreign key with cascade delete in an ALTER TABLE statement in Oracle/PLSQL is:

```
ALTER TABLE table_name
ADD CONSTRAINT constraint_name
FOREIGN KEY (column1, column2, ... column_n)
REFERENCES parent_table (column1, column2, ... column_n)
ON DELETE CASCADE;
```

Example

Let's look at an example of how to create a foreign key with cascade delete using the ALTER TABLE statement in Oracle/PLSQL.

For example:

```
ALTER TABLE products
ADD CONSTRAINT fk_supplier
FOREIGN KEY (supplier_id)
REFERENCES supplier(supplier_id)
ON DELETE CASCADE;
```

In this example, we've created a foreign key (with a cascade delete) called fk_supplier that references the supplier table based on the supplier_id field.

We could also create a foreign key (with a cascade delete) with more than one field as in the example below:

```
ALTER TABLE products
ADD CONSTRAINT fk_supplier
FOREIGN KEY (supplier_id, supplier_name)
REFERENCES supplier(supplier_id, supplier_name)
ON DELETE CASCADE;
```

A more complex example in which a **product_order** table has foreign keys for two other tables. One foreign key references a two-column index in the **product** table. The other references a single-column index in the **customer** table:

```
CREATE TABLE product (
   category INT NOT NULL, id INT NOT NULL,
   price DECIMAL,
   PRIMARY KEY(category, id)
)  ENGINE=INNODB;

CREATE TABLE customer (
   id INT NOT NULL,
   PRIMARY KEY (id)
)  ENGINE=INNODB;

CREATE TABLE product_order (
   no INT NOT NULL AUTO_INCREMENT,
   product_category INT NOT NULL,
   product_id INT NOT NULL,
   customer_id INT NOT NULL,

   PRIMARY KEY(no),
   INDEX (product_category, product_id),
   INDEX (customer_id),

   FOREIGN KEY (product_category, product_id)
   REFERENCES product(category, id)
   ON UPDATE CASCADE
   ON DELETE RESTRICT,

   FOREIGN KEY (customer_id)
    REFERENCES customer(id)
)
```

EXPERIMENT NO : 08

Title: Implementation of Stored Procedure

STORED PROCEDURE: Stored procedure is a named collection of SQL statements and procedural logic. A stored procedure is a prepared SQL code that you can save, so the code can be reused over and over again.

Or
A stored procedure is group of T-SQL (Transact SQL) statements. If you have a situation, where you write the same query over and over again, you can save that specific query as a stored procedure and call it just by it's name.

CREATING A PROCEDURE

A procedure is created with the **CREATE OR REPLACE PROCEDURE** statement. The simplified syntax for the CREATE OR REPLACE PROCEDURE statement is as follows –

Syntax:

```
CREATE PROC procName
AS
BEGIN
        (  body part  )
END
```

N.B. We can use proc or procedure as a keyword

EXECUTING A PROCEDURE:

EXEC procName

N.B. or we can simply write the procedure name

Example

Suppose we want to write a stored procedure that will return name and gender from tblemployee table

```
CREATE PROC spGetEmployee
AS
BEGIN
        SELECT name,gender FROM tblemployee
END
```

Now to run the stored procedure we will write

```
EXEC spGetEmployee
```

STORED PROCEDURE WITH PARAMETERS

Now we want to create a store procedure that will have two parameter Gender and DeptId

```
CREATE PROC getEmployeeByGenderAndDepartmentId
@Gender varchar(10),
@DepartmentId int
AS
BEGIN
SELECT name,gender,departmentId FROM tblemployee WHERE gender=@Gender and
departmentId=@DepartmentId
END
```

Now to execute this procedure we need to write as follow

```
EXEC getEmployeeByGenderAndDepartmentId 'Male',1
```

- ✓ Another way to do the task

```
EXEC getEmployeeByGenderAndDepartmentId @DepartmentId=1,@gender='male'
```

- ✓ **But the following statement is wrong**

```
EXEC getEmployeeByGenderAndDepartmentId 1, 'Male‘
```

ALTER & DROP PROCEDURE

Suppose we want to see the name of the employee in ascending order

Now to do the change we need to alter the procedure

```
ALTER PROC spGetEmployee
AS
BEGIN
SELECT name,gender FROM tblemployee ORDER BY name
END
```

To drop any procedure the syntax is

```
DROP PROC procName
```

Example :

```
DROP PROC spGetEmployee
```

ENCRYPT STORED PROCEDURE

For security issue encryption is required .The format of encryption is

```
ALTER PROC getEmployeeByGenderAndDepartmentId
@Gender varchar(10),
@DepartmentId int
WITH ENCRYPTION
AS
BEGIN
SELECT name,gender,departmentId FROM tblemployee WHERE gender=@Gender and
departmentId=@DepartmentId
END
```

Characteristics of Encrypted Stored Procedure:

- ✓ When a procedure is encrypted we can only use it.
- ✓ It is not possible to view the text of the procedure

  because it is encrypted or locked.
- ✓ But encrypted procedure can be deleted in a formal way.

STORED PROCEDURE WITH OUTPUT PARAMETER

```
CREATE PROC spGetEmployeeByGender
@Gender varchar(20),
@EmployeeCount int OUTPUT
AS
BEGIN
SELECT @EmployeeCount=count(*) FROM tblemployee WHERE gender=@Gender
END
```

N.B. Here EmployeeCount is an output parameter.

To execute this stored procedure the process is

```
DECLARE @TotalCount int
EXEC spGetEmployeeByGender 'Male', @TotalCount OUTPUT
PRINT @TotalCount
```

Now if we do not use output keyword in the 2nd line of query our result will be empty or null.

Declare @TotalCount int

Exec spGetEmployeeByGender 'Male', @TotalCount

If (@TotalCount is Null)

print 'Output is Null'

Else

print 'Output is not Null‘

Output:

Output is Null

PROCEDURE WITH RETURN VALUE

```
CREATE PROC spGetNameById2
@id int
AS
BEGIN
        RETURN (Select name from tblemployee where id=@id)  //returns only integer type
END
```

Print:

```
DECLARE @EmployeeName varchar(20)
EXECUTE @EmployeeName=spGetNameById2 1
PRINT @EmployeeName
```

SYSTEM STORED PROCEDURE

sp_help

- ✓ View the information about stored procedure
- ✓ Like parameter name, datatype.

EXEC SP_HELP spGetEmployeeByGender

Results | Messages

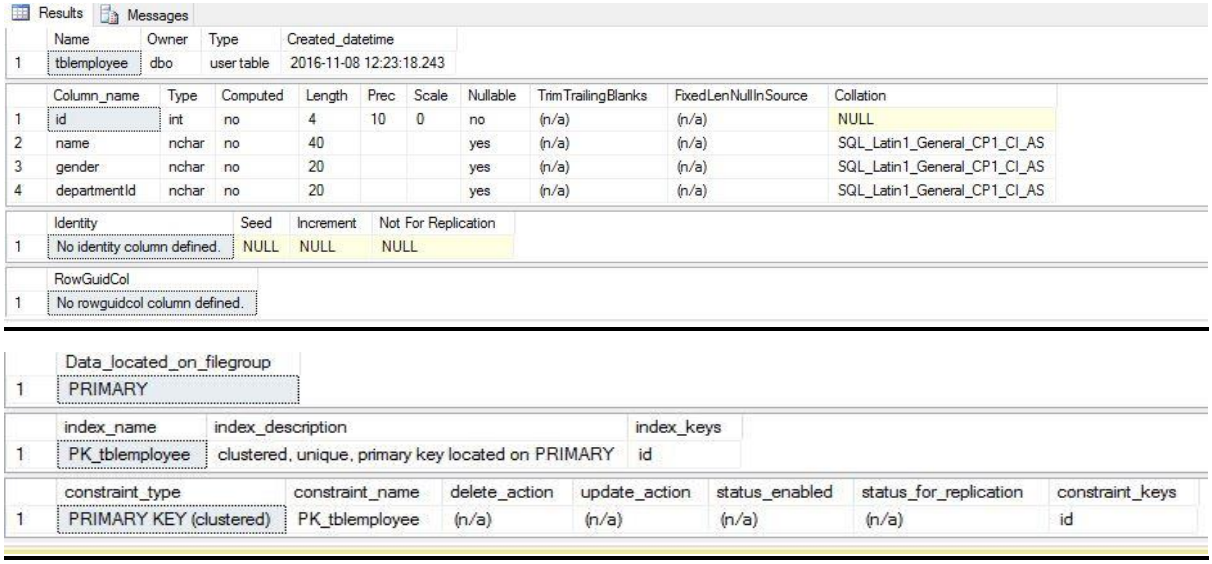
	Name	Owner	Type	Created_datetime
1	spGetEmployeeByGender	dbo	stored procedure	2016-11-08 15:35:12.610

	Parameter_name	Type	Length	Prec	Scale	Param_order	Collation
1	@Gender	varchar	20	20	NULL	1	SQL_Latin1_General_CP1_CI_AS
2	@EmployeeCount	int	4	10	0	2	NULL

sp_help

It can also be applied to database table

Exec sp_help tblEmployee



	Name	Owner	Type	Created_datetime
1	tblemployee	dbo	user table	2016-11-08 12:23:18.243

	Column_name	Type	Computed	Length	Prec	Scale	Nullable	TrimTrailingBlanks	FixedLenNullInSource	Collation
1	id	int	no	4	10	0	no	(n/a)	(n/a)	NULL
2	name	nchar	no	40			yes	(n/a)	(n/a)	SQL_Latin1_General_CP1_CI_AS
3	gender	nchar	no	20			yes	(n/a)	(n/a)	SQL_Latin1_General_CP1_CI_AS
4	departmentId	nchar	no	20			yes	(n/a)	(n/a)	SQL_Latin1_General_CP1_CI_AS

	Identity	Seed	Increment	Not For Replication
1	No identity column defined.	NULL	NULL	NULL

	RowGuidCol
1	No rowguidcol column defined.

	Data_located_on_filegroup
1	PRIMARY

	index_name	index_description	index_keys
1	PK_tblemployee	clustered, unique, primary key located on PRIMARY	id

	constraint_type	constraint_name	delete_action	update_action	status_enabled	status_for_replication	constraint_keys
1	PRIMARY KEY (clustered)	PK_tblemployee	(n/a)	(n/a)	(n/a)	(n/a)	id

sp_helptext

- ✓ View the text of the stored procedure

EXEC SP_HELPTEXT spGetEmployeeByGender

sp_depends

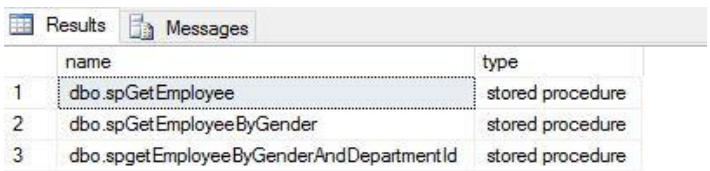
- ✓ View the dependency of the stored procedure
- ✓ This SP is very useful especially to check if there is any dependency in any other table

EXEC SP_DEPENDS spGetEmployeeByGender

	name	type	updated	selected	column
1	dbo.tblemployee	user table	no	yes	gender

sp_depends can also be applied to database table

EXEC SP_DEPENDS tblEmployee



	name	type
1	dbo.spGetEmployee	stored procedure
2	dbo.spGetEmployeeByGender	stored procedure
3	dbo.spgetEmployeeByGenderAndDepartmentId	stored procedure

LAB ASSIGNMENT:

Consider first table as **Account_Detail** table having Account_no as a primary key, second table as **Branch** table having Br_Id as Primary key and third table as **Zone** table where Zone_Id is the primary key of that table.

Account_no	Acc_holder_name	Amount	Branch_Id	Zone_Id
1992212	Mr. Nazmuzzaman	200000	B-101	Z-803
1992213	Mr. Jibon	170000	B-102	Z-803
1882212	Bushra	180000	B-103	Z-802
1882213	Sajib	170000	B-104	Z-801

Br_Id	Branch_Name
B-101	Bonani
B-102	Romna
B-103	Shaheb bazar
B-104	Ullapara

Zone_Id	Name
Z-801	Sirajgonj
Z-802	Rajshahi
Z-803	Dhaka
Z-804	Chittagong

1. Create a simple stored procedure "SPdetails" to find Acc_holder_name, Amount, Branch_Name and Zone_Name.
2. Create a simple stored procedure "SPaverage" to find Branch _name and Amount of Branch where amount will be greater than particular amount (say 17000). Here branch_name and amount will be passed by parameter
3. Create a simple stored procedure "SPbalance" to find Amount of a particular zone. Here zone name will be passed by parameter and amount will be shown by using return value ().
4. Create a simple stored procedure "SPamount" to Find all account holders name with their branch name and zone name whose name has substring 'Mr.' and Amount Less than Maximum Amount
5. Create a simple stored procedure "SPdetailsInfo" to find number of customer of each Zone. Here number of customers need to be printed as output parameter and zone_name will be passed as parameter
6. Create procedure like **"spEmployeeSalaryDetails1"** which has four parameter. three parameter match the Start**Amount**, End**Amount** value, Branch_Name Value and another parameter return this value, in this procedure find the number of **Branch_Name** where Start**Amount**, End**Amount** value, Branch_Name value pass by parameter.
7. Create a simple stored procedure "SPdetailsInfo" to find Zone_name, number of customer of a specific Zone.
8. Create a simple stored procedure "SPdetailsInfo1" to find Zone_name, number of Branch of a specific Zone(Zone name pass by parameter).

EXPERIMENT NO: 09

Title: Implementation of Functions

FUNCTION: A function is same as a procedure except that it returns a value.

Scalar Functions: The user-defined scalar function also returns a single value as a result of actions performed by the function. We return any datatype value from a function.

Example:

```
CREATE FUNCTION fnGetEmpFullName (
 @FirstName varchar(50),
 @LastName varchar(50)
)
RETURNS varchar(101)
AS
BEGIN
  RETURN (Select @FirstName + ' '+ @LastName);
END
```

Call the above function

```
SELECT dbo.fnGetEmpFullName(FirstName,LastName) as Name, Salary FROM Employee
```

Example 2:

```
CREATE FUNCTION CalculateSalary(@Salary float)
RETURNS FLOAT
AS
BEGIN
  DECLARE @maxSalary float
  SET @maxSalary=(@Salary)*100
  RETURN @maxSalary
END
```

Call the above function:

```
SELECT dbo.CalculateSalary (30000)
```

Inline Table-Valued Functions: The user-defined inline table-valued function returns a table variable as a result of actions performed by the function. The value of the table variable should be derived from a single SELECT statement.

```
--Create function to get employees
CREATE FUNCTION fnGetEmployee()
RETURNS TABLE
AS
 RETURN (Select * from Employee)
```

Call the above function:

```
SELECT * FROM fnGetEmployee()
```

Example 2:

```
CREATE FUNCTION dept1(@dept varchar(50))
RETURNS TABLE
AS
  RETURN (select Tid, FirstName, Dept from Teacher where Dept=@dept)
```

Call the above function:

```
SELECT * FROM dept1('CSE' )
```

Multi-Statement Table-Valued Functions: A user-defined multi-statement table-valued function returns a table variable as a result of actions performed by the function. In this, a table variable must be explicitly declared and defined whose value can be derived from multiple **SQL statements.**

```
--Create function for EmpID,FirstName and Salary of Employee
CREATE FUNCTION fnGetMulEmployee()
RETURNS @Emp Table
(
EmpID int,
FirstName varchar(50),
Salary int
)
AS
BEGIN
  INSERT INTO @Emp Select e.EmpID,e.FirstName,e.Salary FROM Employee e;
  --Now update salary of first employee
   UPDATE @Emp SET Salary=25000 WHERE EmpID=1;
  --It will update only in @Emp table not in Original Employee table
RETURN
END
```

Call the above function:

```
Select * from fnGetMulEmployee()
```

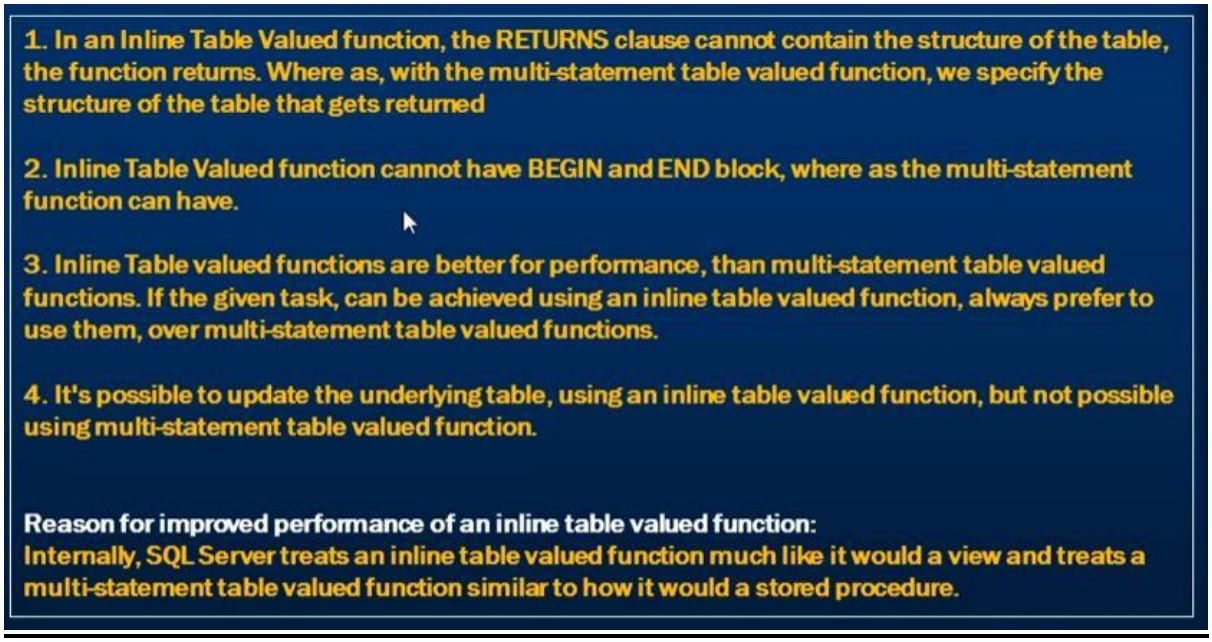
Example 2:

```
CREATE FUNCTION multivaluedTable()
RETURNS @Table Table (Id int, Name varchar(50),dept varchar(50))
AS
BEGIN
  INSERT INTO @Table
  SELECT tid,firstname,Dept FROM teacher
  RETURN
END
```

Call the above function:

```
SELECT * FROM multivaluedTable()
```

Difference between Inline Function & Multi Value Function



Ranking Functions in SQL Server

The Ranking functions in SQL Server returns ranking value for each row in a partition. The SQL Server provides various Rank Functions, which allows us to assign different ranks.

Depending on the function you select, they return different rank value. The following table will show you the list of available Ranking Functions

RANK FUNCTIONS	DESCRIPTION
RANK	It will assign the rank number to each record present in a partition.
DENSE_RANK	It will assign the rank number to each record within a partition without skipping the rank numbers
NTILE	This rank function will assign the rank number to each record present in a partition.
ROW_NUMBER	It will assign the sequential rank number to each unique record present in a partition.

Ranking Consider Table Example:

ID	dept_name	GPA	Grade
15132201	CSE	3.25	B+
15132202	ECE	3.5	A-
15132203	CSE	4.00	A+

1. Find the rank of each student

Select ID,rank()over (order by GPA desc) as s_rank from dept_grades

Output:

	ID	s_rank
1	15132203	1
2	15132202	2
3	15132201	3

2. Find the rank and grade of each student

Select ID,Grade,(1 +(select count(*) from dept_gradesB
Where B.GPA>A.GPA)) as s_rank from dept_grades A orderby s_rank;

ID	Grade	s_rank
15132203	A+	1
15132202	A-	2
15132201	B+	3

3. "Find the rank of students within each department."

Select id,dept_name, rank() over (partition by dept_name order by GPA desc) as dept_rank from dept_grades order by dept_name,dept_rank;

id	dept_name	dept_rank
15132203	CSE	1
15132201	CSE	2
15132202	ECE	1

Select id,dept_name,rank() over (order by GPA desc) as dept_rank from dept_grades
Order bydept_name,dept_rank;
Output:

id	dept_name	dept_rank
15132203	CSE	1
15132201	CSE	3
15132202	ECE	2

The SQL Server NTILE() is a window function that distributes rows of an ordered partition into a specified number of approximately equal groups, or buckets. It assigns each group a bucket number starting from one. For each row in a group, the NTILE() function assigns a bucket number representing the group to which the row belongs.

v	buckets	
1	1	
2	1	4
3	1	
4	1	
5	2	
6	2	3
7	2	
8	3	
9	3	3
10	3	

```
SELECT [FirstName]
      ,[LastName]
      ,[Education]
      ,[Occupation]
      ,[YearlyIncome]
      ,RANK() OVER (ORDER BY [YearlyIncome] DESC) AS RANK
      ,DENSE_RANK() OVER (ORDER BY [YearlyIncome] DESC) AS [DENSE_RANK]
      ,ROW_NUMBER() OVER (ORDER BY [YearlyIncome] DESC) AS [ROW NUMBER]
      ,NTILE(3) OVER (ORDER BY [YearlyIncome] DESC) AS [NTILE NUMBER]
  FROM [Customers]
```

100 % @tutorialgateway.org

Results Messages

	FirstName	LastName	Education	Occupation	YearlyIncome	RANK	DENSE_RANK	ROW NUMBER	NTILE NUMBER
1	John	Yang	Bachelors	Professional	90000.00	1	1	1	1
2	Rob	Johnson	Bachelors	Management	80000.00	2	2	2	1
3	Christy	Zhu	Bachelors	Professional	80000.00	2	2	3	1
4	John	Miller	Masters Degree	Management	80000.00	2	2	4	1
5	John	Ruiz	Bachelors	Professional	70000.00	5	3	5	2
6	Christy	Carlson	Graduate Degree	Management	70000.00	5	3	6	2
7	Rob	Huang	High School	Skilled Manual	60000.00	7	4	7	2
8	Ruben	Torres	Partial College	Skilled Manual	50000.00	8	5	8	3
9	Christy	Mehta	Partial High School	Clerical	50000.00	8	5	9	3
10	Rob	Verhoff	Partial High School	Clerical	45000.00	10	6	10	3

LAB ASSIGNMENT:

Tbl_Management

Mgt_id	Mgt_Name	Joining_date	Salary	Position
M2015	Keshob	2001-01-18	250000	Managing Director
M2016	Rana	2003-01-30	180000	Secretary
M2017	Jasim	2004-04-12	150000	Join secretary
M2018	Rajon	2004-06-18	140000	Join secretary

Tbl_Emp

Emp_id	Emp_Name	Joining_Date	Salary	Division
E1001	Suman	2003-04-25	92000	Software
E1002	Rasel	2004-03-13	86000	Network
E1003	Hossain	2004-06-21	82000	Software
E1004	polash	2005-05-05	9800	Network

Tbl_Project

P_id	P_Name	Mgt_id	E_id	P_Cost	Delivery_date
P3001	Office Automation	M2016	E1001	2050000	2016-05-08
P3002	Repair Hub	M2016	E1004	1200000	2017-06-14
P3003	Server Installation	M2018	E1001	1500500	2018-02-13
P3004	Network setup	M2017	E1002	2505000	2018-03-12

1. Write a sql query to show Project name, cost and Rank according to cost, assign employee name and rearrange the project according to cost ascending order.
2. Write a sql create UDF query to show Project name, cost and assign employee name and rearrange the project according to cost ascending order. Where Project name and employee name pass by parameter.

3. Write a sql query to find the rank of management Team according to their joining Date.
4. Write a sql create scalar function that has one parameter. In this function calculate the Salary of employee whose salary is maximum and that salary increase 10%. Where salary column pass by parameter
5. Write a sql UDF to show the Name of maximum Cost Project.
6. Write a sql Inline Table Valued function to show the Project name and Cost where cost in between 1200000 and 2050000. Costs are passed by parameter.
7. Create Inline Function like **"fnEmployee",** in this function find the Mgt_id, Mgt_Name, Emp_Name, Joining_Date, Salary, P_Name, P_Cost, Delivery_date. Where P_id, Mgt_id, Emp_id pass by parameter

EXPERIMENT NO: 11

Title: Implementation of Trigger

A trigger is a special type of stored procedure that automatically runs when an event occurs in the database server. DML triggers run when a user tries to modify data through a data manipulation language (DML) event. DML events are INSERT, UPDATE, or DELETE statements on a table or view. These triggers fire when any valid event fires, whether table rows are affected or not.

CREATING TRIGGERS

The syntax for creating a trigger is –

Syntax:

```
CREATE TRIGGER [schema_name.]trigger_name
ON table_name
{ FOR | AFTER | INSTEAD OF }
{ [ INSERT ] [ , ] [ UPDATE ] [ , ] [ DELETE ] }
NOT FOR REPLICATION]
AS
BEGIN
{sql_statements}

END
```

In this syntax:

- The schema_name is the name of the schema to which the new trigger belongs. The schema name is optional.
- The trigger_name is the user-defined name for the new trigger.
- The table_name is the table to which the trigger applies.
- The event is listed in the FOR/AFTER clause. The event could be INSERT, UPDATE, or DELETE. A single trigger can fire in response to one or more actions against the table.
- The NOT FOR REPLICATION option instructs SQL Server not to fire the trigger when data modification is made as part of a replication process.
- The sql_statements is one or more Transact-SQL used to carry out actions once an event occurs

Extra Notes:

FOR | AFTER
FOR or AFTER specifies that the DML trigger fires only when all operations specified in the triggering SQL statement have launched successfully. All referential cascade actions and constraint checks must also succeed before this trigger fires.

You can't define AFTER triggers on views.

INSTEAD OF
Specifies that the DML trigger launches *instead of* the triggering SQL statement, thus, overriding the actions of the triggering statements. You can't specify INSTEAD OF for DDL or logon triggers.

At most, you can define one INSTEAD OF trigger per INSERT, UPDATE, or DELETE statement on a table or view. You can also define views on views where each view has its own INSTEAD OF trigger.

You can't define INSTEAD OF triggers on updatable views that use WITH CHECK OPTION. Doing so results in an error when an INSTEAD OF trigger is added to an updatable view WITH CHECK OPTION specified. You remove that option by using ALTER VIEW before defining the INSTEAD OF trigger.

{ [DELETE] [,] [INSERT] [,] [UPDATE] }
Specifies the data modification statements that activate the DML trigger when it's tried against this table or view. Specify at least one option. Use any combination of these options in any order in the trigger definition.

For INSTEAD OF triggers, you can't use the DELETE option on tables that have a referential relationship, specifying a cascade action ON DELETE. Similarly, the UPDATE option isn't allowed on tables that have a referential relationship, specifying a cascade action ON UPDATE.

Use the inserted and deleted Tables

DML trigger statements use two special tables: the deleted table and the inserted tables. SQL Server automatically creates and manages these tables. You can use these temporary, memory-resident tables to test the effects of certain data modifications and to set conditions for DML trigger actions. You cannot directly modify the data in the tables or perform data definition language (DDL) operations on the tables, such as CREATE INDEX.

In DML triggers, the inserted and deleted tables are primarily used to perform the following:

- Extend referential integrity between tables.
- Insert or update data in base tables underlying a view.
- Test for errors and take action based on the error.
- Find the difference between the state of a table before and after a data modification and take actions based on that difference.

The deleted table stores copies of the affected rows during DELETE and UPDATE statements. During the execution of a DELETE or UPDATE statement, rows are deleted from the trigger table and transferred to the deleted table. The deleted table and the trigger table ordinarily have no rows in common.

The inserted table stores copies of the affected rows during INSERT and UPDATE statements. During an insert or update transaction, new rows are added to both the inserted table and the trigger table. The rows in the inserted table are copies of the new rows in the trigger table.

An update transaction is similar to a delete operation followed by an insert operation; the old rows are copied to the deleted table first, and then the new rows are copied to the trigger table and to the inserted table.

Example
TRIGGER AFTER INSERT

```
CREATE TRIGGER trInsertInAudit
ON tblemployee
FOR INSERT
AS
BEGIN
  DECLARE @id int
  SELECT @id=id from INSERTED
  INSERT INTO tblEmployeeAuditData
  VALUES('New Employee with ID '+ CAST(@id as varchar(10)) + ' is inserted at '+
cast(GETDATE() as varchar(20)) )
END
```

N.B. tblemlpoyee table has id, name, gender, deptId column and tblEmployeeAuditData has id, audaitData column

TRIGGER AFTER DELETE

```
CREATE TRIGGER trDeleteInAudit
ON tblemployee
FOR DELETE
AS
BEGIN
  DECLARE @id int
  SELECT @id=id FROM DELETED
  INSERT INTO tblEmployeeAuditData
  VALUES('New Employee with ID '+ CAST(@id as varchar(10)) + ' is deleted at '+
cast(GETDATE() as varchar(20)) )

END
```

LAB ASSIGNMENT:

Table name is **tblInfo**

ProductId	Name	UnitPrice	Store_Location
1	English	200	Rajshahi
2	Sociology	140	Dhaka
3	Bangla	110	Dhaka
4	Math	100	Rajshahi

1. Write a trigger to the given table (tblInfo) as trInsert and after insertion any new row it will store a string ("Id 5 product Computer_book has been added at 26-july-2017") in the Data column of **tblAudit** table which will be predefined.

2. Write a trigger to the given table (tblInfo) as trDelete and after deleting any new row it will store a string ("Id 5 product Computer_book has been deleted at 26-july-2017") in the Data column of **tblAudit** table which will be predefined.