DATABASE MANAGEMENT SYSTEMS

[Course Code: V18CSL06]

LABORATORY MANUAL

B.TECH (V-18 Regulation)
(III YEAR – I SEM)
(2020-21)

DEPARTMENT OF
COMPUTER SCIENCE AND ENGINEERING

SRI VASAVI ENGINEERING COLLEGE
(Autonomous Institution – UGC, Govt. of India)

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

Vision

- To evolve as a center of academic and research excellence in the area of Computer Science and Engineering.

Mission

- To utilize innovative learning methods for academic improvement.
- To encourage higher studies and research to meet the futuristic requirements of Computer Science and Engineering.
- To inculcate Ethics and Human values for developing students with good character.

PROGRAMME EDUCATIONAL OBJECTIVES (PEOs)

CSE Graduates of this programme will be able to:

PEO1 – Adapt to evolving technology.

PEO2 – Provide optimal solutions to real time problems.

PEO3 – Demonstrate his/her abilities to support service activities with due consideration of Professional and Ethical Values.

PROGRAM SPECIFIC OUTCOMES (PSOs)

A graduate of the Computer Science and Engineering Programme will be able to:

1. **PSO1: Use Mathematical Abstractions and Algorithmic Design along with Open Source Programming tools to solve complexities involved in Programming. [K3]**.

2. **PSO2: Use Professional Engineering practices and strategies for development and maintenance of software. [K3]**

PROGRAM OUTCOMES (POs)

Engineering Graduates will be able to:

1. **Engineering knowledge**: Apply the knowledge of Mathematics, Science, Engineering Fundamentals and Concepts of Computer Science Engineering to the solution of complex engineering problems.[K3]

2. **Problem analysis**: Identify, formulate, review research literature, and analyze complex engineering problems reaching substantiated conclusions using first principles of Mathematics, Natural Sciences, and Computer sciences.[K4]

3. **Design / development of solutions**: Design solutions for complex engineering problems and design system components or processes that meet the specified needs with appropriate consideration for the public health and safety, and the cultural, societal, and environmental considerations.[K5]
4. **Conduct investigations of complex problems**: Use research-based knowledge and research methods including design of experiments, analysis and interpretation of data, and synthesis of the information to provide valid conclusions.[K5]

5. **Modern tool usage**: Create, select, and apply appropriate techniques, resources, and modern engineering and IT tools including prediction and modeling to complex engineering activities with an understanding of the limitations.[K3]

6. **The engineer and society**: Apply reasoning informed by the contextual knowledge to assess societal, health, safety, legal and cultural issues and the consequent responsibilities relevant to the professional engineering practice.[K3]

7. **Environment and sustainability**: Understand the impact of the professional engineering solutions in societal and environmental contexts, and demonstrate the knowledge of, and need for sustainable development.[K3]

8. **Ethics**: Apply ethical principles and commit to professional ethics and responsibilities and norms of the engineering practice.[K3]

9. **Individual and team work**: Function effectively as an individual, and as a member or leader in diverse teams, and in multidisciplinary settings.[K6]

10. **Communication**: Communicate effectively on complex engineering activities with the engineering community and with society at large, such as, being able to comprehend and write effective reports and design documentation, make effective presentations, and give and receive clear instructions.[K2]

11. **Project management and finance**: Demonstrate knowledge and understanding of the engineering and management principles and apply these to one's own work, as a member and leader in a team, to manage projects and in multi disciplinary environments.[K6]

12. **Life- long learning**: Recognize the need for, and have the preparation and ability to engage in independent and life-long learning in the broadest context of technological change.[K1]

Course Outcomes: After Successful completion of the Course, the student will be able to:

CO1: Build SQL Queries and Constraints (K3)
CO2: Experiment with various Database Indexing Techniques. (K3)
CO3: Construct PL/SQL Cursors and Exceptions (K3)
CO4: Develop application programs using PL/SQL (K3)
CO5: Develop PL/SQL Functions, Procedures, Packages (K3)
CO6: Apply projections and aggregation on collection of MongoDB database (K3)

INDEX

Relational Database Management System

RDBMS is acronym for Relation Database Management System. Dr. E. F. Codd first introduced the Relational Database Model in 1970. The Relational model allows data to be represented in a simple row- column. Each data field is considered as a column and each record is considered as a row. Relational Database is more or less similar to Database Management System. In relational model there is relation between their data elements. Data is stored in tables. Tables have columns, rows and names. Tables can be related to each other if each has a column with a common type of information. The most famous RDBMS packages are Oracle, Sybase and Informix.

E. F. Codd Rules

1. **The Information Rule**

   All information must be store in table as data values.
2. **The Rule of Guaranteed Access**

   Every item in a table must be logically addressable with the help of a table name.
3. **The Systematic Treatment of Null Values**

   The RDBMS must be taken care of null values to represent missing or inapplicable information.
4. **The Database Description Rule**

   A description of database is maintained using the same logical structures with which data was defined by the RDBMS.
5. **Comprehensive Data Sub Language**

   According to the rule the system must support data definition, view definition, data manipulation, integrity constraints, authorization and transaction management operations.
6. **The View Updating Rule**

   All views that are theoretically updatable are also updatable by the system.
7. **The Insert and Update Rule**

   This rule indicates that all the data manipulation commands must be operational on sets of rows having a relation rather than on a single row.
8. **The Physical Independence Rule**

   Application programs must remain unimpaired when any changes are made in storage representation or access methods.
9. **The Logical Data Independence Rule**

   The changes that are made should not affect the user's ability to work with the data.The change can be splitting table into many more tables.
10. **The Integrity Independence Rule**

    The integrity constraints should store in the system catalog or in the database.
11. **The Distribution Rule**

    The system must be access or manipulate the data that is distributed in other systems.
12. **The Non-subversion Rule** If a RDBMS supports a lower level language then it should not bypass any integrity constraints defined in the higher level.

WEEK-1

1. Queries to facilitate acquaintance of Built-In Functions, String Functions, Numeric Functions, Date Functions and Conversion Functions.

Objective:

- ✓ *To understand and implement various types of function in SQL.*

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;

Develop aggregate plan strategies to assist with summarization of several data entries.

Aggregative operators: 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.

__Syntax:__ AVG (n1, n2...)

__Example:__ Select AVG (10, 15, 30) FROM DUAL;

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;

```
SQL> select deptno, max(sal) from emp group by deptno;

    DEPTNO      MAX (SAL)
    -------   ---------
      10      5000
      20      3000
      30      2850

SQL> select deptno, max (sal) from emp group by deptno having max(sal)<3000;

  DEPTNO     MAX(SAL)
  ----      ------
    30      2850
```

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

__Syntax:__ MIN (Column name)

__Example:__ SELECT MIN (Sal) FROM emp;

```
SQL>select deptno,min(sal) from emp group by deptno having min(sal)>1000;

 DEPTNO MIN (SAL)
 ----    -------
   10    1300
```

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;

CONVERSION FUNCTIONS:

To_char: TO_CHAR (number) converts n to a value of VARCHAR2 data type, using the optional number format fmt. The value n can be of type NUMBER, BINARY_FLOAT, or BINARY_DOUBLE.

```
SQL>select to_char(65,'RN')from dual;

LXV
```

To_number : TO_NUMBER converts expr to a value of NUMBER data type.

```
SQL>Select to_number ('1234.64') from Dual;
1234.64
```

To_date:TO_DATE converts char of CHAR, VARCHAR2, NCHAR, or NVARCHAR2 data type to a value of DATE data type.

```
SQL>SELECT TO_DATE('January 15, 1989, 11:00 A.M.')FROM DUAL;

TO_DATE
------------
15-JAN-89
```

STRING FUNCTIONS:

Concat: CONCAT returns char1 concatenated with char2. Both char1 and char2 can be any of the datatypes

```
SQL>SELECT CONCAT('ORACLE','CORPORATION')FROM DUAL;
        ORACLECORPORATION
```

Lpad: LPAD returns expr1, left-padded to length n characters with the sequence of characters in expr2.

```
SQL>SELECT LPAD('ORACLE',15,'*')FROM DUAL;
        *********ORACLE
```

Rpad: RPAD returns expr1, right-padded to length n characters with expr2, replicated as many times as necessary.

```
SQL>SELECT RPAD ('ORACLE',15,'*')FROM DUAL;
        ORACLE*********
```

Ltrim: Returns a character expression after removing leading blanks.

```
SQL>SELECT LTRIM('SSMITHSS','S')FROM DUAL;
        MITHSS
```

Rtrim: Returns a character string after truncating all trailing blanks

```
SQL>SELECT RTRIM('SSMITHSS','S')FROM DUAL;
        SSMITH
```

Lower: Returns a character expression after converting uppercase character data to lowercase.

```
SQL>SELECT LOWER('DBMS')FROM DUAL;
        dbms
```

Upper: Returns a character expression with lowercase character data converted to uppercase

```
SQL>SELECT UPPER('dbms')FROM DUAL;
        DBMS
```

Length: Returns the number of characters, rather than the number of bytes, of the given string expression, excluding trailing blanks.

```
SQL>SELECT LENGTH('DATABASE')FROM DUAL;
        8
```

Substr: Returns part of a character, binary, text, or image expression.

```
SQL>SELECT SUBSTR('ABCDEFGHIJ'3,4)FROM DUAL;
        CDEF
```

Instr: The INSTR functions search string for substring. The function returns an integer indicating the position of the character in string that is the first character of this occurrence.

```
SQL>SELECT INSTR('CORPORATE FLOOR','OR',3,2)FROM DUAL;
        14
```

DATE FUNCTIONS:

Sysdate:

```
SQL>SELECT SYSDATE FROM DUAL;
        29-DEC-08
```

next_day:

```
SQL>SELECT NEXT_DAY(SYSDATE,'WED')FROM DUAL;
        05-JAN-09
```

add_months:

```
SQL>SELECT ADD_MONTHS(SYSDATE,2)FROM DUAL;
        28-FEB-09
```

last_day:

```
SQL>SELECT LAST_DAY(SYSDATE)FROM DUAL;
        31-DEC-08
```

months_between:

```
SQL>SELECT MONTHS_BETWEEN(SYSDATE,HIREDATE)FROM EMP;
 4
```

Least:

```
SQL>SELECT LEAST('10-JAN-07','12-OCT-07')FROM DUAL;
        10-JAN-07
```

Greatest:

```
SQL>SELECT GREATEST('10-JAN-07','12-OCT-07')FROM DUAL;
        10-JAN-07
```

Trunc:

```
SQL>SELECT TRUNC(SYSDATE,'DAY')FROM DUAL;
        28-DEC-08
```

Round:

```
SQL>SELECT ROUND(SYSDATE,'DAY')FROM DUAL;
        28-DEC-08
```

to_char:

```
SQL> select to_char(sysdate, "dd\mm\yy") from dual;
        24-mar-05.
```

to_date:

```
SQL> select to date (sysdate, "dd\mm\yy") from dual;
        24-mar-o5.
```

LAB PRACTICE ASSIGNMENT:

Create a table EMPLOYEE with following schema:

(Emp_no, E_name, E_address, E_ph_no, Dept_no, Dept_name,Job_id, Designation , Salary)

Write SQL statements for the following query.

1. List the E_no, E_name, Salary of all employees working for MANAGER.
2. Display all the details of the employee whose salary is more than the Sal of any IT PROFF..
3. List the employees in the ascending order of Designations of those joined after 1981.
4. List the employees along with their Experience and Daily Salary.
5. List the employees who are either ‘CLERK’ or ‘ANALYST’ .
6. List the employees who joined on 1-MAY-81, 3-DEC-81, 17-DEC-81,19-JAN-80 .
7. List the employees who are working for the Deptno 10 or20.
8. List the Enames those are starting with ‘S’ .
9. Dislay the name as well as the first five characters of name(s) starting with ‘H’
10. List all the emps except ‘PRESIDENT’ & ‘MGR” in asc order of Salaries.

****

2. Queries using operators in SQL

- Arithmetic Operator
- Logical Operator
- Comparison Operator
- Special Operator
- Set Operator

Objective:

✓ To learn different types of operator.

Theory:

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.

OR: The OR operator is used to combine multiple conditions in an SQL statement's WHERE clause.

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

IS NULL: The NULL operator is used to compare a value with a NULL attribute value.

ALL: The ALL operator is used to compare a value to all values in another value set

ANY: The ANY operator is used to compare a value to any applicable value in the list according to the condition.

LIKE: The LIKE operator is used to compare a value to similar values using wildcard operators.It allows to use percent sign(%) and underscore (_) to match a given string pattern.

IN: The IN operator is used to compare a value to a list of literal values that have been specified.

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

SET OPERATORS:

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

- Union
- Union all
- Intersect
- Minus

Union: Returns all distinct rows selected by both the queries

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

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

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

LAB PRACTICE ASSIGNMENT:

1. Display all the dept numbers available with the dept and emp tables avoiding duplicates.
2. Display all the dept numbers available with the dept and emp tables.
3. Display all the dept numbers available in emp and not in dept tables and vice versa.

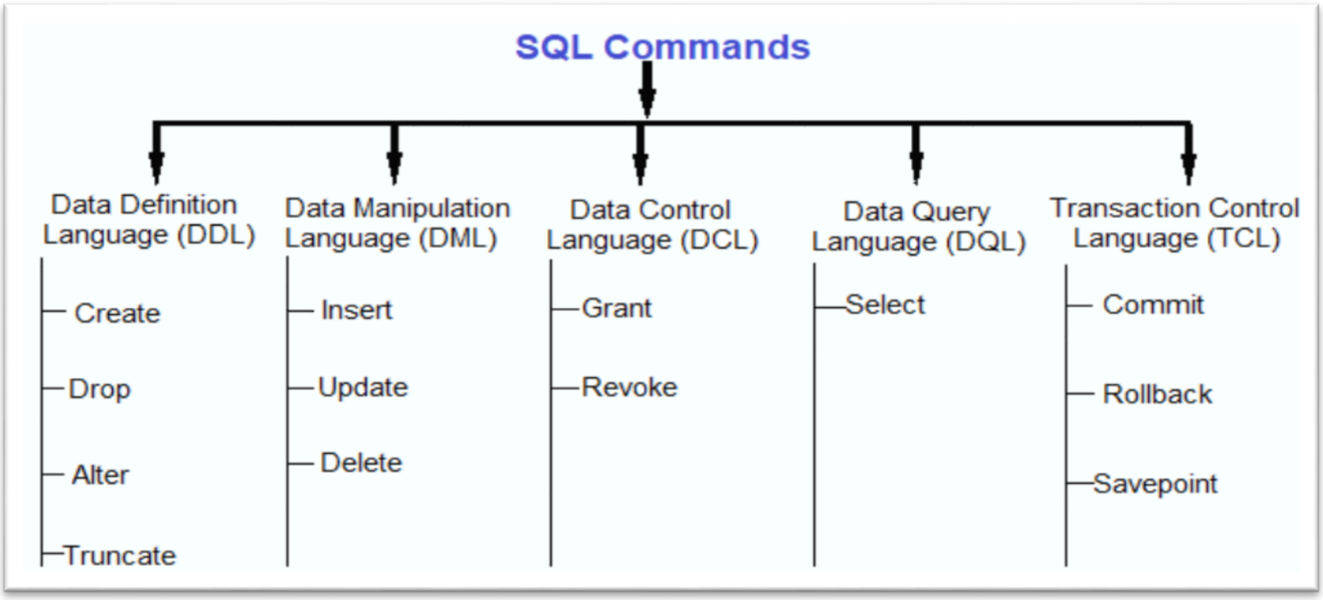
****

3. Queries to Retrieve and Change Data: Select, Insert, Delete, and Update

Objective :

- ✓ To understand the different issues involved in the design and implementation of a database system
- ✓ To understand and use data manipulation language to query, update, and manage a database.

Theory :


SQL Commands
Data Definition Language (DDL)
Create
Drop
Alter
Truncate
Data Manipulation Language (DML)
Insert
Update
Delete
Data Control Language (DCL)
Grant
Revoke
Data Query Language (DQL)
Select
Transaction Control Language (TCL)
Commit
Rollback
Savepoint

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’,’VIZAG’);

b) *Inserting a single record*

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

Example: SQL>INSERT INTO student VALUES (1,'Ravi','M.Tech','VIZAG');

c) *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';

d) *Inserting multiple records*

Syntax: INSERT INTO relation_name field_1,field_2,. ...field_n) VALUES
(&data_1,&data_2,.......&data_n);

Example: SQL>INSERT INTO student (sno, sname, class,address)
VALUES (&sno,'&sname','&class','&address');

```
Enter value for sno: 101 Enter
value for name: Ravi Enter
value for class: M.Tech Enter
value for name: VIZAG
```

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 = 'kumar' WHERE sno=1;

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 std;

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;

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

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;

- ***To Retrieve data from one or more tables.***

1. SELECT FROM: To display all fields for all records.

Syntax : SELECT * FROM relation_name;

Example : SQL> select * from dept;

```
DEPTNO       DNAME            LOC
--------     -----------      ----------
10           ACCOUNTING       NEW YORK
20           RESEARCH         DALLAS
30           SALES            CHICAGO
40           OPERATIONS       BOSTON
```

2. SELECT FROM: To display a set of fields for all records of relation.

Syntax: SELECT a set of fields FROM relation_name;

Example: SQL> select deptno, dname from dept;

```
DEPTNO       DNAME
------       --------
10           ACCOUNTING
 20          RESEARCH
 30          SALES
```

3. SELECT - FROM -WHERE: This query is used to display a selected set of fields for a selected set of records of a relation.

Syntax: SELECT a set of fields FROM relation_name WHERE condition;

Example: SQL> select * FROM dept WHERE deptno<=20;

```
DEPTNO      DNAME           LOC
------      -----------     ------------
10          ACCOUNTING      NEW YORK
20              RESEARCH    DALLAS
```

LAB PRACTICE ASSIGNMENT:

Create a table EMPLOYEE with following schema:

(Emp_no, E_name, E_address, E_ph_no, Dept_no, Dept_name,Job_id , Salary)

Write SQL queries for following question:

1. Insert aleast 5 rows in the table.
2. Display all the information of EMP table.
3. Display the record of each employee who works in department D10.
4. Update the city of Emp_no-12 with current city as Nagpur.
5. Display the details of Employee who works in department MECH.
6. Delete the email_id of employee James.
7. Display the complete record of employees working in SALES Department.

****

4. Queries using Group By, Order By, and Having Clauses

Objective:

To learn the concept of group functions

Theory:

- **GROUP BY:** This query is used to group to all the records in a relation together for each and every value of a specific key(s) and then display them for a selected set of fields the relation.

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

JOIN using GROUP BY: This query is used to display a set of fields from two relations by matching a common field in them and also group the corresponding records for each and every value of a specified key(s) while displaying.

Syntax: SELECT <set of fields (from both relations)> FROM relation_1,relation_2
WHERE relation_1.field_x=relation_2.field_y GROUP BY field_z;

Example:

SQL> SELECT empno,SUM(SALARY) FROM emp,dept
WHERE emp.deptno =20 GROUP BY empno;

- **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;

JOIN using ORDER BY: This query is used to display a set of fields from two relations by matching a common field in them in an ordered manner based on some fields.

Syntax: SELECT <set of fields (from both relations)> FROM relation_1, relation_2
WHERE relation_1.field_x = relation_2.field_y ORDER BY field_z;

Example: SQL> SELECT empno,ename,job,dname FROM emp,dept
WHERE emp.deptno = 20 ORDER BY job;

- **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.

  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);
```

LAB PRACTICE ASSIGNMENT:

Create a relation and implement the following queries.

1. Display total salary spent for each job category.
2. Display lowest paid employee details under each manager.
3. Display number of employees working in each department and their department name.
4. Display the details of employees sorting the salary in increasing order.
5. Show the record of employee earning salary greater than 16000 in each department.
6. Write queries to implement and practice the above clause.

****

5. Queries on Controlling Data: Commit, Rollback, and Save point

Objective:

- ✓ To understand the concept of administrative commands

Theory:

A transaction is a logical unit of work. All changes made to the database can be referred to as a transaction. Transaction changes can be made permanent to the database only if they are committed a transaction begins with an executable SQL statement & ends explicitly with either rollback or commit statement.

1. COMMIT: This command is used to end a transaction only with the help of the commit command transaction changes can be made permanent to the database.

Syntax: SQL> COMMIT;

Example: SQL> COMMIT;

2. SAVE POINT: Save points are like marks to divide a very lengthy transaction to smaller once. They are used to identify a point in a transaction to which we can latter role back. Thus, save point is used in conjunction with role back.

Syntax: SQL> SAVE POINT ID;

Example: SQL> SAVE POINT xyz;

3. ROLLBACK: A role back command is used to undo the current transactions. We can roll back the entire transaction so that all changes made by SQL statements are undo (or) role back a transaction to a save point so that the SQL statements after the save point are roll back.

Syntax: ROLLBACK (current transaction can be roll back) ROLLBACK to save point ID;

Example: SQL> ROLLBACK;

SQL> ROLLBACK TO SAVE POINT xyz;

LAB PRACTICE ASSIGNMENT:

1. Write a query to implement the save point.
2. Write a query to implement the rollback.
3. Write a query to implement the commit.

****

6. Queries to Build Report in SQL *PLUS

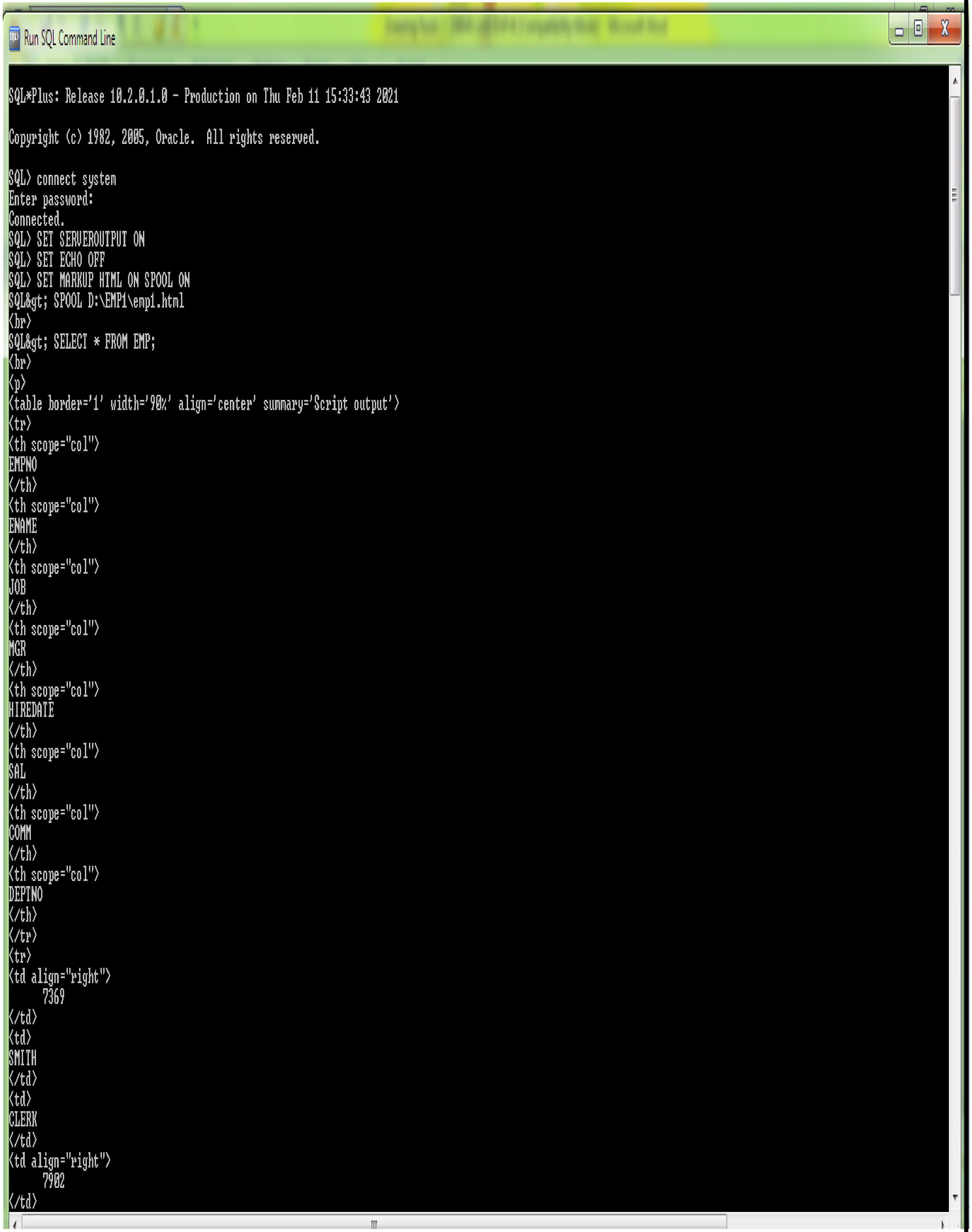
Both Oracle8i and Oracle9i allow the generation of HTML directly from SQL*Plus. This can be used to build complete reports, or output to be embedded in larger reports. The MARKUP HTML ON statement tell SQL*Plus to produce HTML output. The SPOOL ON option of this command indicates that the results should be encapsulated in <html><body>...</body></html> tags. If a complete HTML page is not required the SPOOL OFF option can be used.

The following example displays the contents of the SYSTEM.EMP table as a HTML table.

type the below commands in the :

```
SQL>SET ECHO OFF

SQL>SET MARKUP HTML ON SPOOL ON

SQL>SPOOL D:\EMP1\emp1.html

SQL>SELECT * FROM emp;

SQL>SPOOL OFF
SQL>SET MARKUP HTML OFF
SQL>SET ECHO ON
```

EMPNO	ENAME	JOB	MGR	HIREDATE	SAL	COMM	DEPTNO
7369	SMITH	CLERK	7902	17-DEC-80	800		20
7499	ALLEN	SALESMAN	7698	20-FEB-81	1600	300	30
7521	WARD	SALESMAN	7698	22-FEB-81	1250	500	30
7566	JONES	MANAGER	7839	02-APR-81	2975		20
7654	MARTIN	SALESMAN	7698	28-SEP-81	1250	1400	30
7698	BLAKE	MANAGER	7839	01-MAY-81	2850		30
7782	CLARK	MANAGER	7839	09-JUN-81	2450		10
7788	SCOTT	ANALYST	7566	19-APR-87	3000		20
7839	KING	PRESIDENT		17-NOV-81	5000		10
7844	TURNER	SALESMAN	7698	08-SEP-81	1500	0	30
7876	ADAMS	CLERK	7788	23-MAY-87	1100		20
7900	JAMES	CLERK	7698	03-DEC-81	950		30



Run SQL Command Line

```
SQL*Plus: Release 10.2.0.1.0 - Production on Thu Feb 11 15:33:43 2021

Copyright (c) 1982, 2005, Oracle.  All rights reserved.

SQL> connect system
Enter password:
Connected.
SQL> SET SERVEROUTPUT ON
SQL> SET ECHO OFF
SQL> SET MARKUP HTML ON SPOOL ON
SQL&gt; SPOOL D:\EMP1\emp1.html
<br>
SQL&gt; SELECT * FROM EMP;
<br>
<p>
<table border='1' width='90%' align='center' summary='Script output'>
<tr>
<th scope="col">
EMPNO
</th>
<th scope="col">
ENAME
</th>
<th scope="col">
JOB
</th>
<th scope="col">
MGR
</th>
<th scope="col">
HIREDATE
</th>
<th scope="col">
SAL
</th>
<th scope="col">
COMM
</th>
<th scope="col">
DEPTNO
</th>
</tr>
<tr>
<td align="right">
      7369
</td>
<td>
SMITH
</td>
<td>
CLERK
</td>
<td align="right">
      7902
</td>
```

7. Queries for Creating, Dropping, and Altering Tables, Views, and Constraints

Data Definition Language

The data definition language is used to create an object, alter the structure of an object and also drop already created object. The Data Definition Languages used for table definition can be classified into following:

- Create table command
- Alter table command
- Truncate table command
- Drop table command

1. **CREATION OF TABLES:**

SQL - CREATE TABLE:
Table is a primary object of database, used to store data in form of rows and columns. It is created using following command:

Syntax: CREATE TABLE tablename (column_name data_ type constraints, …)

```
SQL>CREATE TABLE SAILORS ((SID int(10) PRIMARY KEY, SNAME
VARCHAR (10), RATING int (10), AGE int (10));
```

Table Created.

Desc command

The DESCRIBE command is used to view the structure of a table as follows.

```
SQL>DESC SAILORS;
```

Example 1: Create an RESERVES table with fields (SID , BID ,DAY) and display using DESCRIBE command.

```
 CREATE TABLE RESERVES (bid NUMBER(5),sid Number(5),dob DATE);
SQL>DESC RESERVES;
```

Example 2:Create a BOATS table with Fields(BID,BNAME,COLOR)and display using DESCRIBE command

```
CREATE TABLE BOAT(bid NUMBER(4),bname VARCHAR(20),colour VARCHAR(10));
SQL>DESC BOATS;
```

2. <u>ALTER TABLE :</u>

<u>To ADD a column:</u>

SYNTAX: ALTER TABLE <TABLE NAME>ADD (<NEW COLUMN
NAME><DATA TYPE>(<SIZE>), <NEW COLUMN NAME><DATA
TYPE>(<SIZE>)…..................);

<u>To DROP a column</u>:

SYNTAX: ALTER TABLE <TABLE NAME>DROP COLUMN <COLUMN NAME>;.

<u>To MODIFY a column</u>:

SYNTAX: ALTER TABLE <TABLE NAME>MODIFY(<COLUMN NAME>
<NEW DATATYPE>(<NEW SIZE>));

<u>Example1:</u>

SQL>ALTER TABLE SAILOR ADD (SNO NUMBER(10));

3. **<u>RENAME A TABLE</u>**

Rename command is used to give new names for existing tables.

SQL> **RENAME** oldtablename TO newtablename;

4. **<u>TRUNCATE A TABLE</u>**

Truncate command is used to delete all records from a table.

SQL> TRUNCATE TABLE tablename;

5. **<u>DROP A TABLE</u>**

Drop command is used to remove an existing table permanently from database.

SQL> DROP TABLE tablename;

VIEW: In SQL, a view is a virtual table based on the result-set of an SQL statement.

A view contains rows and columns, just like a real table. The fields in a view are fields from one or more real tables in the database.
You can add SQL functions, WHERE, and JOIN statements to a view and present the data as if the data were coming from one single table.
A view is a virtual table, which consists of a set of columns from one or more tables. It is similar to a table but it does not store in the database. View is a query stored as an object.

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.
```

Example:

```
CREATE VIEW [Current Product List] AS
SELECT ProductID, ProductName
FROM Products
WHERE Discontinued=No;
```

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

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

Syntax: DROP VIEW <view_name> ;

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).

1. 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**));

2. 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**));

3. 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(*logical expression*), ….**);

Example:

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

4. 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**));

5. 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);

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**;**

6. 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:

Consider the following schema: Sailors (sid, sname, rating, age) Boats (bid, bname, color)
Reserves (sid, bid, day(date))

Write subquery statement for the following queries.

1. Find all information of sailors who have reserved boat number 101.
2. Find the name of boat reserved by Bob.
3. Find the names of sailors who have reserved a red boat, and list in the order of age.
4. Find the names of sailors who have reserved at least one boat.
5. Find the ids and names of sailors who have reserved two different boats on the same day.
6. Find the ids of sailors who have reserved a red boat or a green boat.
7. Find the name and the age of the youngest sailor.
8. Count the number of different sailor names.
9. Find the average age of sailors for each rating level.
10. Find the average age of sailors for each rating level that has at least two sailors.

1Create a table called EMP with the following structure.

```
Name Type
-------- ------------------------------
EMPNO NUMBER (6)
ENAME VARCHAR2 (20)
JOB VARCHAR2 (10)
DEPTNO NUMBER (3)
SAL NUMBER (7,2)
```

Allow NULL for all columns except ename and job.

2. Add constraints to check, while entering the empno value (i.e) empno > 100.
3. Define the field DEPTNO as unique.
4. Create a primary key constraint for the table(EMPNO).
5. Write queries to implement and practice constraints.

****

8. Queries on Joins and Correlated Sub-Queries

Objective :

- ✓ To implement different types of joins
  - Inner Join
  - Outer Join
  - Natural Join..etc

Theory :

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.The join is actually performed by the ‘where’ clause which combines specified rows of tables.

Syntax:

```
SELECT column 1, column 2, column 3...
FROM table_name1, table_name2
WHERE table_name1.column name =  table_name2.columnname;
```

Types of Joins :

1. Simple Join
2. Self Join
3. Outer Join

Simple Join:

It is the most common type of join. It retrieves the rows from 2 tables having a common column and is further classified into

Equi-join :

A join, which is based on equalities, is called equi-join.

Example:

```
Select * from item, cust where item.id=cust.id;
```

In the above statement, item-id = cust-id performs the join statement. It retrieves rows from both the tables provided they both have the same id as specified by the where clause. Since the where clause uses the comparison operator (=) to perform a join, it is said to be equijoin. It combines the matched rows of tables. It can be used as follows:

- ✓ To insert records in the target table.
- ✓ To create tables and insert records in this table.
- ✓ To update records in the target table.
- ✓ To create views.

Non Equi-join:

It specifies the relationship between columns belonging to different tables by making use of relational operators other than'='.

Example:

Select * from item, cust where item.id<cust.id;

Table Aliases
Table aliases are used to make multiple table queries shorted and more readable. We give an alias name to the table in the 'from' clause and use it instead of the name throughout the query.

Self join:

Joining of a table to itself is known as self-join. It joins one row in a table to another.
It can compare each row of the table to itself and also with other rows of the same table.

Example:

select * from emp x ,emp y where x.salary >= (select avg(salary) from x.emp where x. deptno =y.deptno);

Outer Join:

It extends the result of a simple join. An outer join returns all the rows returned by simple join as well as those rows from one table that do not match any row from the table. The symbol(+) represents outer join.

- Left outer join
- Right outer join
- Full outer join

ORDER TABLE

OrderID	CustomerID	OrderDate
10308	2	1996-09-18
10309	37	1996-09-19
10310	77	1996-09-20

CUSTOMER TABLE

CustomerID	CustomerName	ContactName	Country
1	Alfreds Futterkiste	Maria Anders	Germany
2	Ana Trujillo Emparedados	Ana Trujillo	Mexico
3	Antonio Moreno Taquería	Antonio Moreno	Mexico

The LEFT JOIN keyword returns all rows from the left
table (table_name1), even if there are no matches in the right table (table_name2).

<u>LEFT JOIN</u>

SELECT Customers.CustomerName, Orders.OrderID FROM Customers

LEFT JOIN Orders ON Customers.CustomerID=Orders.CustomerID

ORDER BY Customers.CustomerName;

<u>RIGHT JOIN</u>

The RIGHT JOIN keyword Return all rows from the right
table (table_name2), even if there are no matches in the left table (table_name1).

This is another table named employee.here they have to give field accordingly.

SELECT Orders.OrderID, Employees.FirstName

FROM Orders

RIGHT JOIN Employees

ON Orders.EmployeeID=Employees.EmployeeID

ORDER BY Orders.OrderID;

OUTER JOIN

```
SELECT Customers.CustomerName, Orders.OrderID
FROM Customers
FULL OUTER JOIN Orders
ON Customers.CustomerID=Orders.CustomerID
ORDER BY Customers.CustomerName;
```

LAB PRACTICE ASSIGNMENT:

Consider the following schema: Sailors (sid, sname, rating, age) Boats (bid, bname, color) Reserves (sid, bid, day(date))

1. Find all information of sailors who have reserved boat number 101.
2. Find the name of boat reserved by Bob.
3. Find the names of sailors who have reserved a red boat, and list in the order of age.
4. Find the names of sailors who have reserved at least one boat.
5. Find the ids and names of sailors who have reserved two different boats on the same day.
6. Find the ids of sailors who have reserved a red boat or a green boat.
7. Find the name and the age of the youngest sailor.
8. Count the number of different sailor names.
9. Find the average age of sailors for each rating level.
10. Find the average age of sailors for each rating level that has at least two sailors.

****

Queries (along with sub Queries) using ANY, ALL, IN, EXISTS, NOTEXISTS, UNION, INTERSET, MINUS.

DISTINCT Keyword :- The DISTINCT keyword eliminates the duplicate tuples from the result records set.

Ex:- Find the Names and Ages of all sailors.

SELECT DISTINCT S.SNAME, S.AGE FROM SAILORS S;
Output:

SNAME	AGE
ANDY	25.5
RUSTY	35
BRUTUS	33
ART	25.5
kanth	41
DUSTIN	45
HARTIO	40
LUBBER	55.5
ZORBA	16
HORATIO	35

10 rows returned in 0.16 seconds

The answer is a set of rows, each of which is a pair (sname, age). If two or more sailors have the same name and age, the answer still contains just one pair with that name and age.

UNION, INTERSECT, EXCEPT (MINUS) :- SQL provides three set-manipulation constructs that extend the basic query form. Since the answer to a query is a multiset of rows, it is natural to consider the use of operations such as union, intersection, and difference.
SQL supports these operations under the names **UNION, INTERSECT and MINUS**.

Note : UNION, INTERSECT, and MINUS can be used on any two tables that are **Union-Compatible**, that is, have the same number of columns and the columns, taken in order, have the same data types.

UNION :- It is a set operator used as alternative to **OR** query.
Here is an example of Query using **OR.**
Ex:- Find the names of sailors who have reserved a red or a green boat.
SELECT S.SNAME FROM SAILORS S, RESERVES R, BOATS B WHERE S.SID = R.SID AND R.BID = B.BID AND (B.COLOR = 'RED' OR B.COLOR = 'GREEN');

Output :

SNAME
DUSTIN
DUSTIN
DUSTIN
LUBBER
LUBBER

LUBBER
HORATIO
HARTIO

8 rows returned in 0.03 seconds

Same query can be written using **UNION** as follows.
SELECT S.SNAME FROM SAILORS S, RESERVES R, BOATS B WHERE S.SID = R.SID AND R.BID = B.BID AND B.COLOR = 'RED'
UNION
SELECT S2.SNAME FROM SAILORS S2, BOATS B2, RESERVES R2 WHERE S2.SID = R2.SID AND R2.BID = B2.BID AND B2.COLOR = 'GREEN';
This query says that we want the union of the set of sailors who have reserved red boats and the set of sailors who have reserved green boats.
Output :

SNAME
DUSTIN
HARTIO
HORATIO
LUBBER

4 rows returned in 0.02 seconds

Find all sids of sailors who have a rating of 10 or reserved boat number 1.
SELECT S.SID FROM SAILORS S WHERE S.RATING = 10
UNION
SELECT R.SID FROM RESERVES R WHERE R.BID = 1;
Output :

SID
28
58
71
74

4 rows returned in 0.01 seconds

INTERSECT :- It is a set operator used as alternative to **AND** query. Here is an example of Query using **AND.**
Ex:- Find the names of sailor's who have reserved both a red and a green boat.
SELECT S.SNAME FROM SAILORS S, RESERVES R1, BOATS B1, RESERVES R2, BOATS B2 WHERE S.SID = R1.SID AND R1.BID = B1.BID AND S.SID = R2.SID AND R2.BID = B2.BID AND B1.COLOR='RED' AND B2.COLOR = 'GREEN';

Output :

SNAME
DUSTIN
DUSTIN
LUBBER
LUBBER

4 rows returned in 0.01 seconds

Same query can be written using **INTERSECT** as follows.
SELECT S.SNAME FROM SAILORS S, RESERVES R, BOATS B WHERE S.SID = R.SID AND R.BID = B.BID AND B.COLOR = 'RED'
INTERSECT
SELECT S2.SNAME FROM SAILORS S2, BOATS B2, RESERVES R2 WHERE S2.SID = R2.SID AND R2.BID = B2.BID AND B2.COLOR = 'GREEN';
Output :

SNAME
DUSTIN
LUBBER

2 rows returned in 0.00 seconds

EXCEPT (MINUS) :- It is a set operator used as set-difference. Our next query illustrates the set difference
operation.
Ex:- Find the SID of all sailors who have reserved red boats but not green boats.
SELECT R1.SID FROM BOATS B1, RESERVES R1 WHERE R1.BID = B1.BID AND B1.COLOR = 'RED'
MINUS
SELECT R2.SID FROM BOATS B2, RESERVES R2 WHERE R2.BID = B2.BID AND B2.COLOR = 'GREEN';
Output :

SID
64

1 rows returned in 0.00 seconds

NESTED QUERIES:-
For retrieving data from the tables we have seen the simple & basic queries. These queries extract the data from one or more tables. Here we are going to see some complex & powerful queries that enables us to retrieve the data in desired manner. One of the most powerful features of SQL is nested queries. A nested query is a query that has another query embedded within it; the embedded query is called a subquery.

IN Operator :- The IN operator allows us to test whether a value is in a given set of elements; an SQL query is used to generate the set to be tested.
Ex:- Find the names of sailors who have reserved boat 103 using IN Operator.

SELECT S.SNAME FROM SAILORS S WHERE S.SID
IN (SELECT R.SID FROM RESERVES R WHERE R.BID = 103);

Output :

```
SNAME
DUSTIN
LUBBER
HARTIO
3 rows returned in 0.00
seconds
```

NOT IN Operator :- The NOT IN is used in a opposite manner to IN.
Ex:- Find the names of sailors who have not reserved boat 103 using NOT IN Operator.

SELECT S.SNAME FROM SAILORS S WHERE S.SID
NOT IN (SELECT R.SID FROM RESERVES R WHERE R.BID = 103);

Output :

```
SNAME
BRUTUS
ANDY
RUSTY
HORATIO
ZORBA
ART
DUSTIN
kanth
8 rows returned in 0.00
seconds
```

EXISTS Operator :- This is a Correlated Nested Queries operator. The EXISTS operator is another set comparison operator, such as IN. It allows us to test whether a set is nonempty, an implicit comparison with the empty set.
Ex:- Find the names of sailors who have reserved boat number 103 using EXISTS Operator.

SELECT S.SNAME FROM SAILORS S WHERE
EXISTS (SELECT * FROM RESERVES R WHERE R.BID = 103 AND R.SID = S.SID);
Output :

```
SNAME
DUSTIN
LUBBER
HARTIO
3 rows returned in 0.00
seconds
```

NOT EXISTS Operator :- The NOT EXISTS is used in a opposite manner to EXISTS. Ex:- Find the names of sailors who have not reserved boat number 103 using NOT EXISTS Operator.

SELECT S.SNAME FROM SAILORS S WHERE NOT EXISTS
(SELECT * FROM RESERVES R WHERE R.BID = 103 AND R.SID =
S.SID);

Output :

SNAME
DUSTIN
ZORBA
ART
HORATIO
RUSTY
ANDY
BRUTUS
kanth

8 rows returned in 0.02 seconds

Set-Comparison Operators:- We have already seen the set-comparison operators EXISTS, IN along with their negated versions. SQL also supports **op ANY** and **op ALL**, where **op** is one of the arithmetic comparison operators {<, <=, =, <>, >=, >}. Following are the example which illustrates the use of these Set-Comparison Operators.

op ANY Operator :- It is a comparison operator. It is used to compare a value with any of element in a given set.

Ex:- Find sailors whose rating is better than some sailor called Rajesh using ANY Operator.

SELECT S.SID FROM SAILORS S WHERE S.RATING > ANY (SELECT S2.RATING FROM SAILORS S2 WHERE S2.SNAME = ' RAJESH ');

Note that **IN** and **NOT IN** are equivalent to **= ANY** and **<> ALL**, respectively.

Output :

SID
71
74
58
28

4 rows returned in 0.00 seconds

op ALL Operator :- It is a comparison operator. It is used to compare a value with all the elements in a given set.

Ex:- Find the sailor's with the highest rating using ALL Operator.

SELECT S.SID FROM SAILORS S WHERE S.RATING >= ALL (SELECT S2.RATING FROM SAILORS S2)

Output :

SID
58
71
74
28

4 rows returned in 0.00 seconds

9. Queries on Working with Index, Sequence, Synonym, Controlling Access, and Locking Rows for Update, Creating Password and Security features PL/SQL.

CREATE INDEX:

The CREATE INDEX statement is used to create indexes in tables.

Indexes are used to retrieve data from the database more quickly than otherwise. The users cannot see the indexes, they are just used to speed up searches/queries.

The SQL statement below creates an index named "idx_lastname" on the "LastName" column in the "Persons" table:

```
CREATE INDEX idx_lastname
ON Persons (LastName);

CREATE INDEX idx_pname
ON Persons (LastName, FirstName);

Ex1ample :

select * from emp;
```

EMPNO	ENAME	JOB	MGR	HIREDATE	SAL	COMM	DEPTNO
7369	SMITH	CLERK	7902	17-DEC-80	800	100	10

1 rows returned in 0.00 seconds CSV Export

```
CREATE INDEX idx_ENAME ON emp(ENAME);

Out Put:

Index created.
```

0.28 seconds

DROP INDEX Statement:

The DROP INDEX statement is used to delete an index in a table.

```
DROP INDEX index_name;
DROP INDEX idx_ENAME;
Index dropped.
```

1.73 seconds

SQL | SEQUENCES:

Sequence is a set of integers 1, 2, 3, that are generated and supported by some database systems to produce unique values on demand.

- A sequence is a user defined schema bound object that generates a sequence of numeric values.
- Sequences are frequently used in many databases because many applications require each row in a table to contain a unique value and sequences provides an easy way to generate them.
- The sequence of numeric values is generated in an a**scending or descending order** at defined intervals and can be configured to restart when max_value exceeds.

Syntax:

```
CREATE SEQUENCE sequence_name
START WITH initial_value
INCREMENT BY increment_value
MINVALUE minimum value
MAXVALUE maximum value
CYCLE|NOCYCLE;

CREATE TABLE students
(
ID number (10),
NAME char (20)
);
CREATE SEQUENCE sequence_1
start with 1
increment by 1
minvalue 0
maxvalue 100
cycle;

INSERT into students VALUES (sequence_1.nextval,'Ramesh');
INSERT into students VALUES (sequence_1.nextval,'Suresh');
select * from students;
```

Example2 :

```
create table s1(no int,name char(10));
create sequence se1
start with 1
increment by 1
minvalue 0
maxvalue 100
cycle;
```

insert into s1 values(se1.nextval,'ram');
insert into s1 values(se1.nextval,'hari');
insert into s1 values(se1.nextval,'krishna');
select * from s1;

SQL | SYNONYM :

A **SYNONYM** provides another name for database object, referred to as original object, that may exist on a local or another server. A synonym belongs to schema, name of synonym should be unique. A synonym cannot be original object for an additional synonym and synonym cannot refer to user-defined function.
The query below results in an entry for each synonym in database. This query provides details about synonym metadata such as the name of synonym and name of the base object.

The query below results in an entry for each synonym in database. This query provides details about synonym metadata such as the name of synonym and name of the base object.
select * from sys.synonyms ;
Example 1 :
create synonym syn for students;
select * from syn;
Example 2 :
Synonym
create synonym syn for s1;
select * from syn;

Grant & revoke permissions
Connect SYS as SYSDBA;
Password: manager

Create table emp(no int, name char(10));

Create user sri identified by ramya;

Grant select, update on emp to sri;

Revoke select, update on emp to sri;

10. Write a PL/SQL Code using Basic Variable, Anchored Declarations, and Usage of Assignment Operation.

Variables and Constants in PL/SQL :

- A variable is a meaningful name that provides facility for programmer to store data temporary during execution of code. It helps to manipulate data in PL/SQL.
- A constant is a value used in PL/SQL block that remains unchanged throughout the program and it can be declared and used instead of actual values.

The following program describes, how to define a variable by adding two numbers.

```
Declare
Var1 integer;
Var2 integer;
 Var3 integer;
 Begin Var1:=&var1;
 Var2:=&var2;
 Var3:=var1+var2;
Dbms_output.put_line(var3);
End;
/
```

```
DECLARE
a integer := 10;
b integer := 20;
 c integer;
f real;
BEGIN
c := a + b;
dbms_output.put_line('Value of c: ' || c); f := 70.0/3.0;
dbms_output.put_line('Value of f: ' || f);
END;
```

output :

```
Value of c: 30
Value of f: 23.333333333333333333333333333333333333

Statement processed.
0.06 seconds
```

Anchored Declarations

This section describes the use of the %TYPE declaration attribute to anchor the datatype of one variable to another data structure, such as a PL/SQL variable or a column in a table. When you anchor a datatype, you tell PL/SQL to set the datatype of one variable from the datatype of another element.

The syntax for an anchored datatype is:

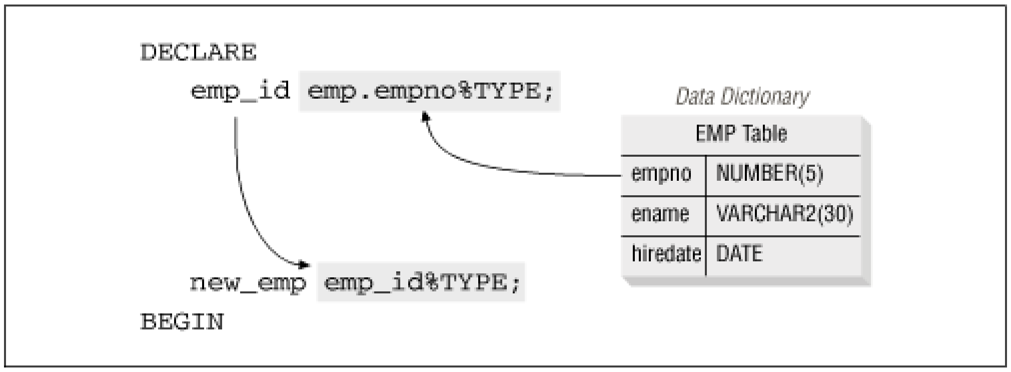
<variable name> <type attribute>%TYPE [optional default value assignment];

where <variable name> is the name of the variable you are declaring and <type attribute> is any of the following:

- Previously declared PL/SQL variable name
- Table column in format "table.column"

Figure shows how the datatype is drawn both from a database table and PL/SQL variable.

Figure : Anchored declarations with %TYPE



```
DECLARE
   emp_id emp.empno%TYPE;


   new_emp emp_id%TYPE;
BEGIN
```

Here are some examples of %TYPE used in declarations:

- Anchor the datatype of monthly_sales to the datatype of total_sales:
- total_sales
  NUMBER (20,2);
  monthly_sales
  total_sales%TYPE;

- Anchor the datatype of the company ID variable to the

  database column: company_id#

  company.company_id%TYPE;

Usage Of Assignment Operators

Anchored declarations provide an excellent illustration of the fact that PL/SQL is not just a procedural-style programming language but was designed specifically as an extension to the Oracle SQL language. A very thorough effort was made by Oracle Corporation to tightly integrate the programming constructs of PL/SQL to the underlying database (accessed through SQL).

The assignment operator is simply the way PL/SQL sets the value of one variable to a given value. There is only one assignment operator, := . I'm not sure where you saw the others listed or used, but they are invalid. Assignment operators are different from a regular equal sign, =, in that they are used to assign a specified value to a PL/SQL variable. For instance, if I wanted to give a variable named V_TEMPERATURE an initial value of 98.6, I'd use the following assignment statement:

```
set serveroutput on
DECLARE
v_temperature number := 98.6 ;
BEGIN
dbms_output.put_line('The temperature is ' || v_temperature) ;
END ;
/
```

```
The temperature is 98.6

Statement processed.

0.00 seconds
```

11. Write a PL/SQL Code Bind and Substitution Variables. Printing in PL/SQL.

Substitution Variables:

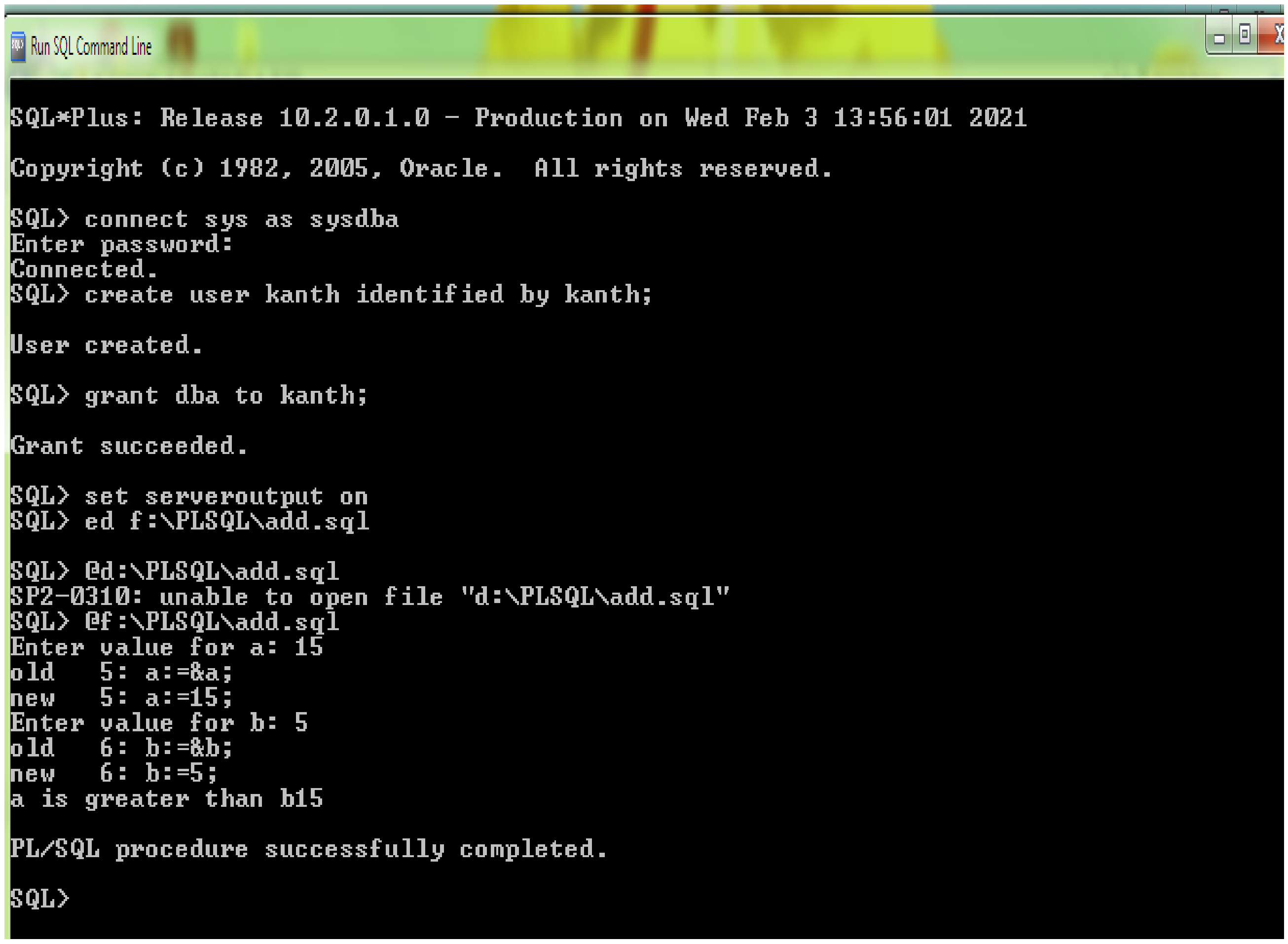
The clue here is in the name... "substitution". It relates to values being substituted into the code before it is submitted to the database.

Steps to execute PL/SQL Code in Command Line.

```
SQL> connect sys as sysdba
Enter the password:password
SQL> create user kanth identified by kanth;
SQL> grant dba to kanth;
Grant Succeeded.
SQL> set serveroutput on
SQL>ed f:\PLSQL\add.sql
```

```
declare
a int;
b int;
Begin
a:=&a;
b:=&b;
if (a>b) then
dbms_output.put_line('a is greater than b'||a);
else
dbms_output.put_line('b is greater than a'||b);
end if;
End;
/
```

```
SQL>@ f:\PLSQL\add.sql
Enter the value for a:15
Enter the value for b: 5
a is greater than b.
```



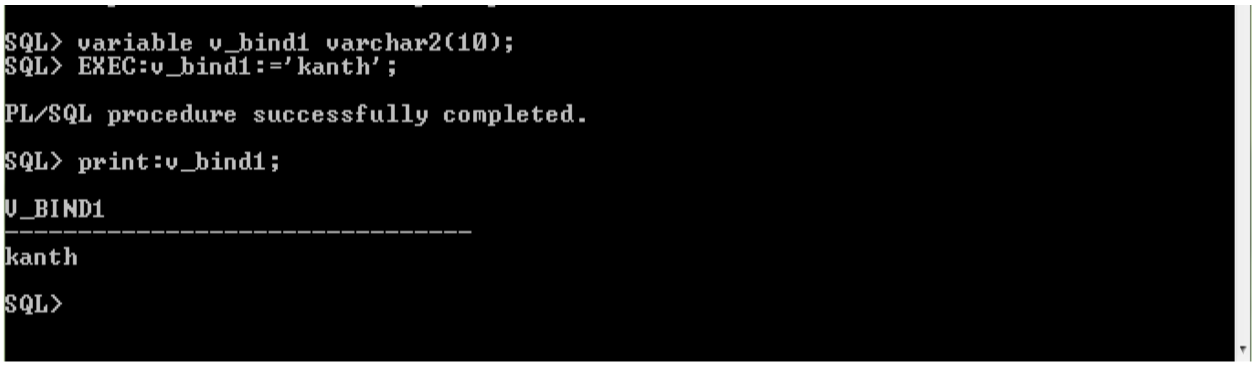
```
SQL*Plus: Release 10.2.0.1.0 - Production on Wed Feb 3 13:56:01 2021

Copyright (c) 1982, 2005, Oracle.  All rights reserved.

SQL> connect sys as sysdba
Enter password:
Connected.
SQL> create user kanth identified by kanth;

User created.

SQL> grant dba to kanth;

Grant succeeded.

SQL> set serveroutput on
SQL> ed f:\PLSQL\add.sql

SQL> @d:\PLSQL\add.sql
SP2-0310: unable to open file "d:\PLSQL\add.sql"
SQL> @f:\PLSQL\add.sql
Enter value for a: 15
old   5: a:=&a;
new   5: a:=15;
Enter value for b: 5
old   6: b:=&b;
new   6: b:=5;
a is greater than b15

PL/SQL procedure successfully completed.

SQL>
```

BIND VARIABLES: Bind variables in oracle database can be defined as variables that we create in SQL*PLUS and then reference in PL/SQL.

way1:
variable v_bind1 varchar2(10);
EXEC :v_bind1:=’kanth’;
way2 :
BEGIN
:V_BIND1:=’KANTH’;
END;
/
How to display:
BEGIN
:V_BIND1:='KANTH';
DBMS_OUTPUT.PUT_LINE(:v_bind1);
END;
/

PRINT :v_bind1

SET AUTOPRINT ON;
VARIABLE V_bind2 varchar2(30);
EXEC :v_bind:='kanth';



```
SQL> variable v_bind1 varchar2(10);
SQL> EXEC:v_bind1:='kanth';

PL/SQL procedure successfully completed.

SQL> print:v_bind1;

V_BIND1
--------------------------------
kanth

SQL>
```

12. Write a PL/SQL block using SQL and Control Structures in PL/SQL?

Program Development using WHILE LOOPS, Numeric FOR LOOPS, Nested Loops using ERROR Handling, BUILT-IN Exceptions, USE Defined Exceptions, RAISE- APPLICATION ERROR.

LOOPING STATEMENTS:- For executing the set of statements repeatedly we can use loops. The oracle supports number of looping statements like GOTO, FOR, WHILE & LOOP. Here is the syntax of these all the types of looping statements.

• **GOTO STATEMENTS**

```
<<LABEL>>
SET OF STATEMENTS
GOTO LABEL;
```

• **FOR LOOP**

```
FOR <VAR> IN [REVERSE] <INI_VALUE>..<END_VALUE>
SET OF STATEMENTS
END LOOP;
```

• **WHILE LOOP**

```
WHILE (CONDITION) LOOP
SET OF STATEMENTS
END LOOP;
```

• **LOOP STATEMENT**

```
LOOP
SET OF STATEMENTS
IF (CONDITION) THEN
EXIT
SET OF STATEMENTS
END LOOP;
```

While using LOOP statement, we have to take care of EXIT condition; otherwise it may go into infinite loop.

Example :- Here are the example for all these types of looping statement where each program prints numbers 1 to 10.

GOTO EXAMPLE

```
DECLARE
I INTEGER := 1;
BEGIN
<<OUTPUT>>
DBMS_OUTPUT.PUT_LINE(I);
I := I + 1;
IF I<=10 THEN
GOTO OUTPUT;
END IF;
END;
/
```

Output:

```
1
2
3
4
5
6
7
8
9
10

Statement processed.
```

0.32 seconds

FOR LOOP EXAMPLE

```
BEGIN
FOR I IN 1..10 LOOP
DBMS_OUTPUT.PUT_LINE (I);
END LOOP;
END;
/
```

Output:

```
1
2
3
4
5
6
7
8
9
10

Statement processed.
```

0.00 seconds

```
DECLARE
I INTEGER := 1;
BEGIN
WHILE(I<=10) LOOP
DBMS_OUTPUT.PUT_LINE(I);
I := I + 1;
END LOOP;
END;
/
```

Output:

```
1
2
3
4
5
6
7
8
9
10

Statement processed.
```

0.00 seconds

```
DECLARE
I INTEGER := 1;
BEGIN
LOOP
DBMS_OUTPUT.PUT_LINE(I);
I := I + 1;
EXIT WHEN I=11;
END LOOP;
END;
/
```

Output:

```
1
2
3
4
5
6
7
8
9
10

Statement processed.

0.00 seconds
```

13. Write a PL/SQL Code using Cursors, Exceptions and Composite Data Types?

Cursors: - Oracle uses temporary work area cursor for storing output of an SQL statement.
Cursors are defined as

CURSOR C1 IS SELECT SID, SNAME, RATING, AND AGE FROM SAILORS;
OR
CURSOR C1 IS SELECT * FROM SAILORS;

Generally while using cursors we have to Open the cursor then extract one row (record) from the cursor using Fetch operation, do the necessary operations on the Record. After completing the work close the cursor. But if we want to do automatic opening & closing to cursor then we can use FOR loop in cursor.

FOR loop in Cursor

The cursor FOR loop gives easy way to handle the Cursor. The FOR loop opens the Cursor, fetches rows and closes the cursor after all rows are processed.

For Eg:

```
FOR Z IN C1 LOOP
- - - - - - -
END LOOP;
```

The cursor FOR loop declares Z as record, which can hold row, returned from cursor.

```
DECLARE
CURSOR C1 IS SELECT * FROM SAILORS;
BEGIN
DBMS_OUTPUT.PUT_LINE('SID'||' '||'SNAME');
FOR Z IN C1 LOOP
DBMS_OUTPUT.PUT_LINE(Z.SID || ' ' || Z.SNAME);
END LOOP;
END;
/
```

```
SID SNAME
22 DUSTIN
29 BRUTUS
31 LUBBER
32 ANDY
58 RUSTY
64 HORATIO
71 ZORBA
74 HARTIO
85 ART
23 DUSTIN
28 kanth

Statement processed.
```

Using While:

```
DECLARE
CURSOR C1 IS SELECT * FROM SAILORS;
Z C1%ROWTYPE;
BEGIN
DBMS_OUTPUT.PUT_LINE('SID'||' '||'SRATING');
OPEN C1;
FETCH C1 INTO Z;
WHILE (C1%FOUND) LOOP
DBMS_OUTPUT.PUT_LINE (Z.SID || ' ' || Z.RATING);
FETCH C1 INTO Z;
END LOOP;
CLOSE C1;
END;
/
```

```
SID SRATING
22 8
29 3
31 9
32 9
58 10
64 8
71 10
74 10
85 5
23 8
28 10

Statement processed.
```

Display records according to rating with proper heading.

```
DECLARE
I INTEGER:=1;
CURSOR C1 IS
SELECT * FROM SAILORS ORDER BY RATING;
Z C1%ROWTYPE;
BEGIN
WHILE(I<=10)LOOP
DBMS_OUTPUT.PUT_LINE('SAILORS WITH RATING'|| I);
FOR Z IN C1 LOOP
IF(Z.RATING=I)THEN
DBMS_OUTPUT.PUT_LINE(Z.SID||'_'||Z.SNAME||'_ '||Z.AGE||'_ '||Z.RATING);
END IF;
END LOOP;
I:=I+1;
END LOOP;
END;
/
```

```
29_BRUTUS_ 33_ 3
85_ART_ 25.5_ 5
22_DUSTIN_ 45_ 8
64_HORATIO_ 35_ 8
23_DUSTIN_ 45_ 8
31_LUBBER_ 55.5_ 9
32_ANDY_ 25.5_ 9
28_kanth_ 41_ 10
71_ZORBA_ 16_ 10
74_HARTIO_ 40_ 10
58_RUSTY_ 35_ 10

Statement processed.

0.01 seconds
```

Multiple cursors in a program: - We can use multiple cursors in a program.

Ex: - To display details of particular table sailors, boats, reserves according to users choice.

```
DECLARE
INPUT VARCHAR2(30):=:INPUT;
CURSOR C1 IS SELECT * FROM SAILORS;
CURSOR C2 IS SELECT * FROM BOATS;
CURSOR C3 IS SELECT * FROM RESERVES;
BEGIN
IF(INPUT='SAILORS') THEN
DBMS_OUTPUT.PUT_LINE('SAILORS INFORMATION:');
FOR Z IN C1 LOOP
DBMS_OUTPUT.PUT_LINE(Z.SID||' '||Z.SNAME||' '||Z.AGE||'  '||Z.RATING);
END LOOP;
ELSIF(INPUT='BOATS')THEN
DBMS_OUTPUT.PUT_LINE('BOATS INFORMATION:');
FOR X IN C2 LOOP
DBMS_OUTPUT.PUT_LINE(X.BID||'  '||X.BNAME||'  '||X.COLOR);
END LOOP;
ELSIF(INPUT='RESERVES')THEN
DBMS_OUTPUT.PUT_LINE('RESERVES INFORMATION:');
FOR Y IN C3 LOOP
DBMS_OUTPUT.PUT_LINE(Y.SID||'  '||Y.BID||'  '||Y.DAY);
END LOOP;
ELSE
DBMS_OUTPUT.PUT_LINE('_NO SUCH TABLE EXISTS');
END IF;
END;
/
```

INPUT:RESERVES
OUTPUT:

```
RESERVES INFORMATION:
22  101  10-OCT-98
22  102  10-OCT-98
22  103  08-OCT-98
22  104  07-OCT-98
31  102  10-NOV-98
31  103  06-NOV-98
31  104  12-NOV-98
64  101  05-SEP-98
64  102  08-SEP-98
74  103  08-SEP-98
```

```
Statement processed.

SAILORS INFORMATION:
22 DUSTIN 45  8
29 BRUTUS 33  3
31 LUBBER 55.5  9
32 ANDY 25.5  9
58 RUSTY 35  10
64 HORATIO 35  8
71 ZORBA 16  10
74 HARTIO 40  10
85 ART 25.5  5
23 DUSTIN 45  8
28 kanth 41  10

Statement processed.

BOATS INFORMATION:
101  INTERLAKE  BLUE
102  INTERLAKE  RED
103  CLIPPER  GREEN
104  MARINE  RED

Statement processed.
```

0.02 seconds

Updating the Records :- Similar to inserting the values as well as selecting the values we can use the PL/SQL programming for updating the records in the given table.

Ex:- To update rating of sailors by 2 if rating is less than 5, by 1 if rating is >5 and doesn‘t change the rating if it is equal to 10.

```
DECLARE
CURSOR C1 IS SELECT * FROM SAILORS;
Z C1%ROWTYPE;
BEGIN
FOR Z IN C1 LOOP
IF (Z.RATING<5) THEN
UPDATE SAILORS SET RATING=RATING+2 WHERE SID=Z.SID;
ELSIF (Z.RATING>5 AND Z.RATING<10) THEN
UPDATE SAILORS SET RATING=RATING+1 WHERE SID=Z.SID;
END IF;
END LOOP;
FOR Z IN C1 LOOP
DBMS_OUTPUT.PUT_LINE (Z.SID||'_' ||Z.RATING);
END LOOP;
END;
/
```

OUTPUT:

```
22_8
29_3
31_9
32_9
58_10
64_8
71_10
74_10
85_5
28_10

Statement processed.
```

0.01 seconds

EXCEPTIONS:

Exceptions in PL/SQL. An exception is an error condition during a program execution.

PL/SQL supports programmers to catch such conditions using **EXCEPTION** block in the program and an appropriate action is taken against the error condition.

There are two types of exceptions –

- System-defined or Pre-defined exceptions
- User-defined exceptions

Syntax for Exception Handling
The general syntax for exception handling is as follows. Here you can list down as many exceptions as you can handle. The default exception will be handled using ***WHEN others THEN*** –

systax :

```
DECLARE
<declarations section>
BEGIN
<executable command(s)>
EXCEPTION
<exception handling goes here >
WHEN exception1 THEN
exception1-handling-statements
WHEN exception2 THEN
exception2-handling-statements
WHEN exception3 THEN
```

```
exception3-handling-statements
........
WHEN others THEN
exception3-handling-statements
END;
```

```
DECLARE
SID VARCHAR2 (10);
BEGIN
SELECT SID INTO SID FROM SAILORS WHERE
SNAME='&SNAME';
DBMS_OUTPUT.PUT_LINE (SID);
EXCEPTION
WHEN NO_DATA_FOUND THEN
DBMS_OUTPUT.PUT_LINE ('_No Sailors with given SID found');
WHEN TOO_MANY_ROWS THEN
DBMS_OUTPUT.PUT_LINE ('_More than one Sailors with same name
found');
END;
/
```

Out Put:

```
_No Sailors with given SID found

Statement processed.
```

0.01 seconds

```
DECLARE
SID VARCHAR2(10);
BEGIN
SELECT 22 INTO SID FROM SAILORS WHERE SNAME='DUSTIN';
DBMS_OUTPUT.PUT_LINE (SID);
EXCEPTION
WHEN NO_DATA_FOUND THEN
DBMS_OUTPUT.PUT_LINE ('_No Sailors with given SID found');
WHEN TOO_MANY_ROWS THEN
DBMS_OUTPUT.PUT_LINE ('_More than one Sailors with same name
found');
END;
```

/

OUTPUT:

```
22

Statement processed.
```

0.00 seconds

SID	SNAME	RATING	AGE
22	DUSTIN	8	45
29	BRUTUS	3	33
31	LUBBER	9	55.5
32	ANDY	9	25.5
58	RUSTY	10	35
64	HORATIO	8	35
71	ZORBA	10	16
74	HARTIO	10	40
85	ART	5	25.5
28	kanth	10	41

10 rows returned in 0.00 seconds

```
DECLARE
RATING VARCHAR2 (10);
BEGIN
SELECT 8 INTO RATING FROM SAILORS WHERE
SNAME='DUSTIN';
DBMS_OUTPUT.PUT_LINE (RATING);
EXCEPTION
WHEN NO_DATA_FOUND THEN
DBMS_OUTPUT.PUT_LINE ('_No Sailors with given SID found');
WHEN TOO_MANY_ROWS THEN
DBMS_OUTPUT.PUT_LINE ('_More than one Sailors with same name
found');
END;
/
```

```
_More than one Sailors with same name found

Statement processed.
```

0.02 seconds.

Composite Data types

VARRAY

```
DECLARE
   TYPE t_name_type IS VARRAY(2)
      OF VARCHAR2(20) NOT NULL;
   t_names t_name_type  := t_name_type('John','Jane');
   t_enames t_name_type := t_name_type();
BEGIN
   -- initialize to an empty array
   dbms_output.put_line('The number of elements in t_enames '| |t_enames.COUNT);

   -- initialize to an array of a elements
   dbms_output.put_line('The number of elements in t_names '|| t_names.COUNT);
END;
/
```

output:

```
The number of elements in t_enames 0
The number of elements in t_names  2

Statement processed.
```

0.00 seconds

Nested Table

```
DECLARE
   TYPE names_table IS TABLE OF VARCHAR2(10);
   TYPE grades IS TABLE OF INTEGER;
   names names_table;
   marks grades;
   total integer;
BEGIN
   names := names_table('Kavita', 'Pritam', 'Ayan', 'Rishav', 'Aziz');
   marks:= grades(98, 97, 78, 87, 92);
   total := names.count;
   dbms_output.put_line('Total '|| total || ' Students');
   FOR i IN 1 .. total LOOP
      dbms_output.put_line('Student:'||names(i)||', Marks:' || marks(i));
   end loop;
END;
/
```

output :

```
Total 5 Students
Student:Kavita, Marks:98
Student:Pritam, Marks:97
Student:Ayan, Marks:78
Student:Rishav, Marks:87
Student:Aziz, Marks:92

Statement processed.
```

0.02 seconds

Associative Array or Index by table

```
DECLARE
   TYPE salary IS TABLE OF NUMBER INDEX BY VARCHAR2(20);
   salary_list salary;
   name   VARCHAR2(20);
BEGIN
   -- adding elements to the table
   salary_list('Rajnish') := 62000;
   salary_list('Minakshi') := 75000;
   salary_list('Martin') := 100000;
   salary_list('James') := 78000;
      -- printing the table
   name := salary_list.FIRST;
   WHILE name IS NOT null LOOP
      dbms_output.put_line
      ('Salary of ' || name || ' is ' || TO_CHAR(salary_list(name)));
      name := salary_list.NEXT(name);
   END LOOP;
END;
/
```

output:

```
Salary of James is 78000
Salary of Martin is 100000
Salary of Minakshi is 75000
Salary of Rajnish is 62000

Statement processed.
```

0.01 seconds

Record

```
DECLARE
   type books is record
      (title varchar(50),
      author varchar(50),
      subject varchar(100),
      book_id number);
   book1 books;
   book2 books;
BEGIN
   -- Book 1 specification
   book1.title  := 'C Programming';
   book1.author := 'Nuha Ali ';
   book1.subject := 'C Programming Tutorial';
   book1.book_id := 6495407;
   -- Book 2 specification
   book2.title := 'Telecom Billing';
   book2.author := 'Zara Ali';
   book2.subject := 'Telecom Billing Tutorial';
   book2.book_id := 6495700;
   -- Print book 1 record
   dbms_output.put_line('Book 1 title : '|| book1.title);
   dbms_output.put_line('Book 1 author : '|| book1.author);
   dbms_output.put_line('Book 1 subject : '|| book1.subject);
   dbms_output.put_line('Book 1 book_id : ' || book1.book_id);
     -- Print book 2 record
   dbms_output.put_line('Book 2 title : '|| book2.title);
   dbms_output.put_line('Book 2 author : '|| book2.author);
   dbms_output.put_line('Book 2 subject : '|| book2.subject);
   dbms_output.put_line('Book 2 book_id : '|| book2.book_id);
END;
/
```

output:

```
Book 1 title : C Programming
Book 1 author : Nuha Ali
Book 1 subject : C Programming Tutorial
Book 1 book_id : 6495407
Book 2 title : Telecom Billing
Book 2 author : Zara Ali
Book 2 subject : Telecom Billing Tutorial
Book 2 book_id : 6495700

Statement processed.
```

0.02 seconds

14. Write a PL/SQL Code using Procedures, Functions, and Packages FORMS?

PL/SQL - PROCEDURES

A **subprogram** is a program unit/module that performs a particular task. These subprograms are combined to form larger programs. This is basically called the 'Modular design'. A subprogram can be invoked by another subprogram or program which is called the **calling program**.

A subprogram can be created −

- At the schema level
- Inside a package
- Inside a PL/SQL block

At the schema level, subprogram is a **standalone subprogram**. It is created with the CREATE PROCEDURE or the CREATE FUNCTION statement. It is stored in the database and can be deleted with the DROP PROCEDURE or DROP FUNCTION statement.

A subprogram created inside a package is a **packaged subprogram**. It is stored in the database and can be deleted only when the package is deleted with the DROP PACKAGE statement. We will discuss packages in the chapter **'PL/SQL - Packages'**.

PL/SQL subprograms are named PL/SQL blocks that can be invoked with a set of parameters. PL/SQL provides two kinds of subprograms −

- **Functions** − These subprograms return a single value; mainly used to compute and return a value.
- **Procedures** − These subprograms do not return a value directly; mainly used to perform an action.

This chapter is going to cover important aspects of a **PL/SQL procedure**. We will discuss **PL/SQL function** in the next chapter.

Parts of a PL/SQL Subprogram

Each PL/SQL subprogram has a name, and may also have a parameter list. Like anonymous PL/SQL blocks, the named blocks will also have the following three parts −

S.No	Parts & Description
1	**Declarative Part** It is an optional part. However, the declarative part for a subprogram does not start with the DECLARE keyword. It contains declarations of types, cursors, constants, variables, exceptions, and nested subprograms. These items are local to the

	subprogram and cease to exist when the subprogram completes execution.
2	**Executable Part** This is a mandatory part and contains statements that perform the designated action.
3	**Exception-handling** This is again an optional part. It contains the code that handles run-time errors.

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 −

```
CREATE [OR REPLACE] PROCEDURE procedure_name
[(parameter_name [IN | OUT | IN OUT] type [, ...])]
{IS | AS}
BEGIN
  < procedure_body >
END procedure_name;
```

Where,

- *procedure-name* specifies the name of the procedure.
- [OR REPLACE] option allows the modification of an existing procedure.
- The optional parameter list contains name, mode and types of the parameters. **IN** represents the value that will be passed from outside and **OUT** represents the parameter that will be used to return a value outside of the procedure.
- *procedure-body* contains the executable part.
- The AS keyword is used instead of the IS keyword for creating a standalone procedure.

Example

The following example creates a simple procedure that displays the string 'Hello World!' on the screen when executed.

```
CREATE OR REPLACE PROCEDURE greetings
AS
BEGIN
   dbms_output.put_line('Hello World!');
END;
/
```

When the above code is executed using the SQL prompt, it will produce the following result –

```
Procedure created.
```

Executing a Standalone Procedure

A standalone procedure can be called in two ways −

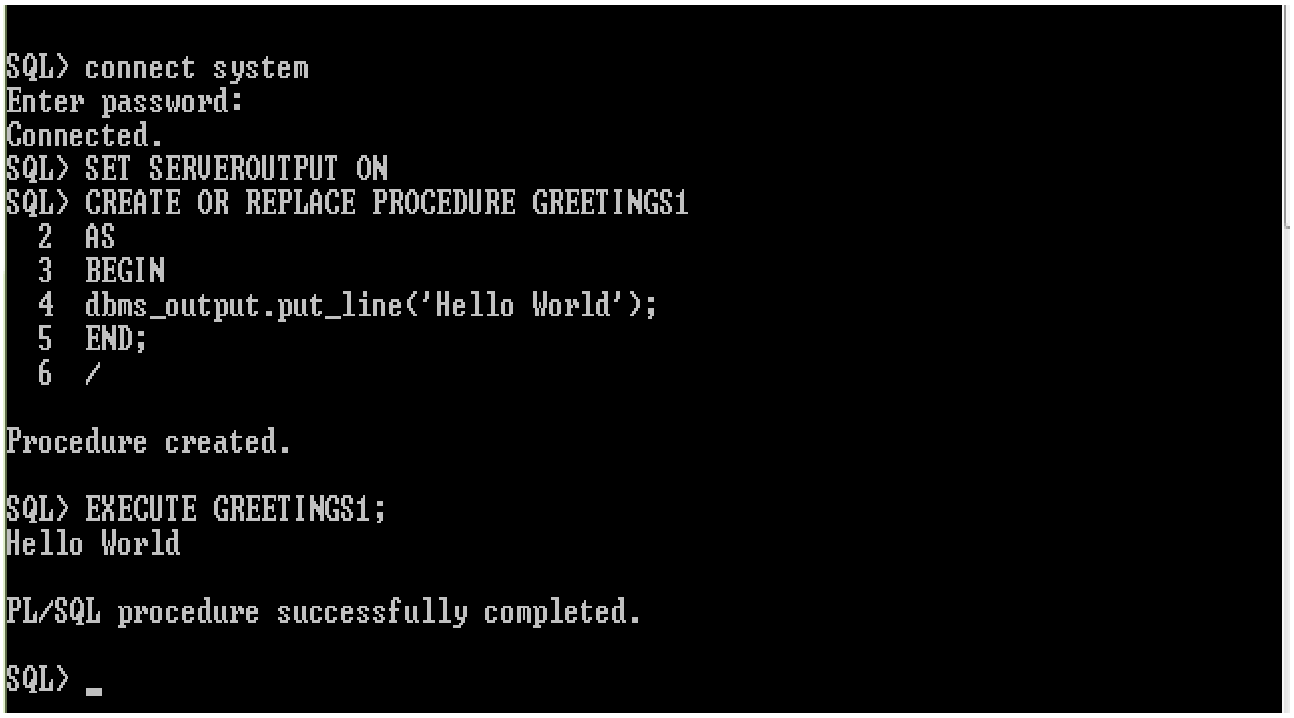
- Using the **EXECUTE** keyword
- Calling the name of the procedure from a PL/SQL block

The above procedure named **'greetings'** can be called with the EXECUTE keyword as −

```
EXECUTE greetings;
```

The above call will display −

```
Hello World

PL/SQL procedure successfully completed.
```



```
SQL> connect system
Enter password:
Connected.
SQL> SET SERVEROUTPUT ON
SQL> CREATE OR REPLACE PROCEDURE GREETINGS1
  2  AS
  3  BEGIN
  4  dbms_output.put_line('Hello World');
  5  END;
  6  /

Procedure created.

SQL> EXECUTE GREETINGS1;
Hello World

PL/SQL procedure successfully completed.

SQL> _
```

The procedure can also be called from another PL/SQL block −

```
BEGIN
   greetings;
END;
/
```

The above call will display −

```
Hello World
PL/SQL procedure successfully completed.
```

Deleting a Standalone Procedure

A standalone procedure is deleted with the **DROP PROCEDURE** statement. Syntax for deleting a procedure is −

```
DROP PROCEDURE procedure-name;
```

You can drop the greetings procedure by using the following statement −

```
DROP PROCEDURE greetings;
```

Parameter Modes in PL/SQL Subprograms

The following table lists out the parameter modes in PL/SQL subprograms −

S.No	Parameter Mode & Description
1	**IN** An IN parameter lets you pass a value to the subprogram. **It is a read-only parameter**. Inside the subprogram, an IN parameter acts like a constant. It cannot be assigned a value. You can pass a constant, literal, initialized variable, or expression as an IN parameter. You can also initialize it to a default value; however, in that case, it is omitted from the subprogram call. **It is the default mode of parameter passing. Parameters are passed by reference**.
2	**OUT** An OUT parameter returns a value to the calling program. Inside the subprogram, an OUT parameter acts like a variable. You can change its value and reference the value after assigning it. **The actual parameter must be variable and it is passed by value**.
3	**IN OUT** An **IN OUT** parameter passes an initial value to a subprogram and returns an updated value to the caller. It can be assigned a value and the value can be read. The actual parameter corresponding to an IN OUT formal parameter must be a variable, not a constant or an expression. Formal parameter must be assigned a value. **Actual parameter is passed by value.**

IN & OUT Mode Example 1

This program finds the minimum of two values. Here, the procedure takes two numbers using the IN mode and returns their minimum using the OUT parameters.

```
DECLARE
   a number;
   b number;
   c number;
PROCEDURE findMin(x IN number, y IN number, z OUT number) IS
BEGIN
   IF x < y THEN
      z:= x;
   ELSE
      z:= y;
   END IF;
END;
BEGIN
   a:= 23;
   b:= 45;
   findMin(a, b, c);
   dbms_output.put_line(' Minimum of (23, 45) : ' || c);
END;
/
```

When the above code is executed at the SQL prompt, it produces the following result −

```
Minimum of (23, 45) : 23

PL/SQL procedure successfully completed.
```

IN & OUT Mode Example 2

This procedure computes the square of value of a passed value. This example shows how we can use the same parameter to accept a value and then return another result.

```
DECLARE
   a number;
PROCEDURE squareNum(x IN OUT number) IS
BEGIN
  x := x * x;
END;
BEGIN
   a:= 23;
   squareNum(a);
   dbms_output.put_line(' Square of (23): ' || a);
END;
/
```

When the above code is executed at the SQL prompt, it produces the following result −

```
Square of (23): 529
PL/SQL procedure successfully completed.
```

PL/SQL - Functions

A function is same as a procedure except that it returns a value. Therefore, all the discussions of the previous chapter are true for functions too.

Creating a Function

A standalone function is created using the **CREATE FUNCTION** statement. The simplified syntax for the **CREATE OR REPLACE PROCEDURE** statement is as follows –

```
CREATE [OR REPLACE] FUNCTION function_name
[(parameter_name [IN | OUT | IN OUT] type [, ...])]
RETURN return_datatype
{IS | AS}
BEGIN
   < function_body >
END [function_name];
```

Where,

- *function-name* specifies the name of the function.
- [OR REPLACE] option allows the modification of an existing function.
- The optional parameter list contains name, mode and types of the parameters. IN represents the value that will be passed from outside and OUT represents the parameter that will be used to return a value outside of the procedure.
- The function must contain a **return** statement.
- The *RETURN* clause specifies the data type you are going to return from the function.
- *function-body* contains the executable part.
- The AS keyword is used instead of the IS keyword for creating a standalone function.

Example

The following example illustrates how to create and call a standalone function. This function returns the total number of CUSTOMERS in the customers table.

We will use the CUSTOMERS table, which we had created in the PL/SQL Variables chapter −

```
create table customers(ID NUMBER, NAME VARCHAR2(20),AGE NUMBER,ADDRESS
VARCHAR2(20),SALARY FLOAT);
INSERT INTO CUSTOMERS VALUES(1,'Ramesh',32,'Ahmedabad',3000.00);
INSERT INTO CUSTOMERS VALUES(2,'Khilan',25,'Delhi',3000.00);
INSERT INTO CUSTOMERS VALUES(3,'kaushik',23,'Kota',3000.00);
INSERT INTO CUSTOMERS VALUES(4,'Chaitali',25,'Mumbai',7500.00);
INSERT INTO CUSTOMERS VALUES(5,'Chaitali',27,'Bhopal',9500.00 );
INSERT INTO CUSTOMERS VALUES(6,'Komal',22,'MP',5500.00);
```

ID	NAME	AGE	ADDRESS	SALARY
1	Ramesh	32	Ahmedabad	3000
2	Khilan	25	Delhi	3000
3	kaushik	23	Kota	3000
4	Chaitali	25	Mumbai	7500
5	Chaitali	27	Bhopal	9500
6	Komal	22	MP	5500

6 rows returned in 0.03 seconds

```
CREATE OR REPLACE FUNCTION totalCustomers
RETURN number IS
   total number(2) := 0;
BEGIN
   SELECT count(*) into total
   FROM customers;

   RETURN total;
END;
/
```

When the above code is executed using the SQL prompt, it will produce the following result.

```
Function created.
```

Calling a Function

While creating a function, you give a definition of what the function has to do. To use a function, you will have to call that function to perform the defined task. When a program calls a function, the program control is transferred to the called function.

A called function performs the defined task and when its return statement is executed or when the **last end statement** is reached, it returns the program control back to the main program.

To call a function, you simply need to pass the required parameters along with the function name and if the function returns a value, then you can store the returned value. Following program calls the function **totalCustomers** from an anonymous block −

```
DECLARE
   c number(2);
BEGIN
   c := totalCustomers();
   dbms_output.put_line('Total no. of Customers: ' || c);
END;
/
```

When the above code is executed at the SQL prompt, it produces the following result –

```
Total no. of Customers: 6
PL/SQL procedure successfully completed.
```

Example

The following example demonstrates Declaring, Defining, and Invoking a Simple PL/SQL Function that computes and returns the maximum of two values.

```
DECLARE
   a number;
   b number;
   c number;
FUNCTION findMax(x IN number, y IN number)
RETURN number
IS
    z number;
BEGIN
   IF x > y THEN
      z:= x;
   ELSE
      Z:= y;
   END IF;
   RETURN z;
END;
BEGIN
   a:= 23;
   b:= 45;
   c := findMax(a, b);
   dbms_output.put_line(' Maximum of (23,45): ' || c);
END;
/
```

When the above code is executed at the SQL prompt, it produces the following result –

```
Maximum of (23,45): 45

PL/SQL procedure successfully completed.
```

PL/SQL Recursive Functions

We have seen that a program or subprogram may call another subprogram. When a subprogram calls itself, it is referred to as a recursive call and the process is known as **recursion**.

To illustrate the concept, let us calculate the factorial of a number. Factorial of a number n is defined as −

```
n! = n*(n-1)!
   = n*(n-1)*(n-2)!
      ...
   = n*(n-1)*(n-2)*(n-3)... 1
```

The following program calculates the factorial of a given number by calling itself recursively.

```
DECLARE
   num number;
   factorial number;

FUNCTION fact(x number)
RETURN number
IS
   f number;
BEGIN
   IF x=0 THEN
      f := 1;
   ELSE
      f := x * fact(x-1);
   END IF;
RETURN f;
END;

BEGIN
   num:= 6;
   factorial := fact(num);
   dbms_output.put_line(' Factorial '|| num || ' is ' || factorial);
END;
/
```

When the above code is executed at the SQL prompt, it produces the following result –

```
Factorial 6 is 720

PL/SQL procedure successfully completed.
```

PL/SQL - PACKAGE

Packages are schema objects that groups logically related PL/SQL types, variables, and subprograms.

A package will have two mandatory parts −

- Package specification
- Package body or definition

Package Specification

The specification is the interface to the package. It just **DECLARES** the types, variables, constants, exceptions, cursors, and subprograms that can be referenced from outside the package. In other words, it contains all information about the content of the package, but excludes the code for the subprograms.

All objects placed in the specification are called **public** objects. Any subprogram not in the package specification but coded in the package body is called a **private** object.

The following code snippet shows a package specification having a single procedure. You can have many global variables defined and multiple procedures or functions inside a package.

```
create table customers(ID NUMBER, NAME VARCHAR2(20),AGE NUMBER,ADDRESS VARCHAR2(20),SALARY FLOAT);

INSERT INTO CUSTOMERS VALUES(1,'Ramesh',32,'Ahmedabad',3000.00);

INSERT INTO CUSTOMERS VALUES(2,'Khilan',25,'Delhi',3000.00);

INSERT INTO CUSTOMERS VALUES(3,'kaushik',23,'Kota',3000.00);

INSERT INTO CUSTOMERS VALUES(4,'Chaitali',25,'Mumbai',7500.00);

INSERT INTO CUSTOMERS VALUES(5,'Chaitali',27,'Bhopal',9500.00 );

INSERT INTO CUSTOMERS VALUES(6,'Komal',22,'MP',5500.00);
```

ID	NAME	AGE	ADDRESS	SALARY
1	Ramesh	32	Ahmedabad	3000
2	Khilan	25	Delhi	3000
3	kaushik	23	Kota	3000
4	Chaitali	25	Mumbai	7500
5	Chaitali	27	Bhopal	9500
6	Komal	22	MP	5500

6 rows returned in 0.03 seconds

```
CREATE PACKAGE cust_sal AS
   PROCEDURE find_sal(c_id customers.id%type);
END cust_sal;
/
```

When the above code is executed at the SQL prompt, it produces the following result −

```
Package created.
```

Package Body

The package body has the codes for various methods declared in the package specification and other private declarations, which are hidden from the code outside the package.

The **CREATE PACKAGE BODY** Statement is used for creating the package body. The following code snippet shows the package body declaration for the ***cust_sal*** package created above. I assumed that we already have CUSTOMERS table created in our database as mentioned in the PL/SQL - Variables chapter.

```
CREATE OR REPLACE PACKAGE BODY cust_sal AS

   PROCEDURE find_sal(c_id customers.id%TYPE) IS
   c_sal customers.salary%TYPE;
   BEGIN
      SELECT salary INTO c_sal
      FROM customers
      WHERE id = c_id;
      dbms_output.put_line('Salary: '|| c_sal);
   END find_sal;
END cust_sal;
/
```

When the above code is executed at the SQL prompt, it produces the following result −

```
Package body created.
```

Using the Package Elements

The package elements (variables, procedures or functions) are accessed with the following syntax −

```
package_name.element_name;
```

Consider, we already have created the above package in our database schema, the following program uses the ***find_sal*** method of the ***cust_sal*** package −

```
DECLARE
   code customers.id%type := &cc_id;
BEGIN
   cust_sal.find_sal(code);
END;
/
```

When the above code is executed at the SQL prompt, it prompts to enter the customer ID and when you enter an ID, it displays the corresponding salary as follows −

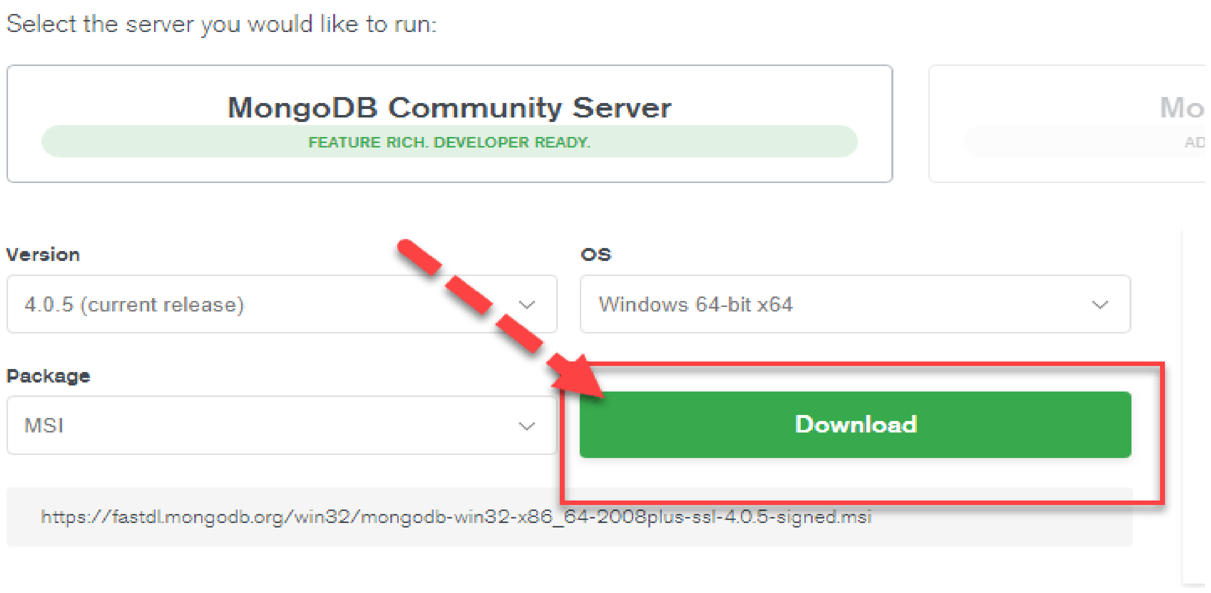
```
Enter value for cc_id: 1
Salary: 3000

PL/SQL procedure successfully completed.
```

TO DROP PROCUDURE USE THE FOLLOWING COMMAND :
DROP PACKAGE cust_sal;

Part-B

1. Install and start MongoDB:

Step1:-Go to link and Download MongoDB Community Server. We will install the 64-bit version for Windows.



Select the server you would like to run:

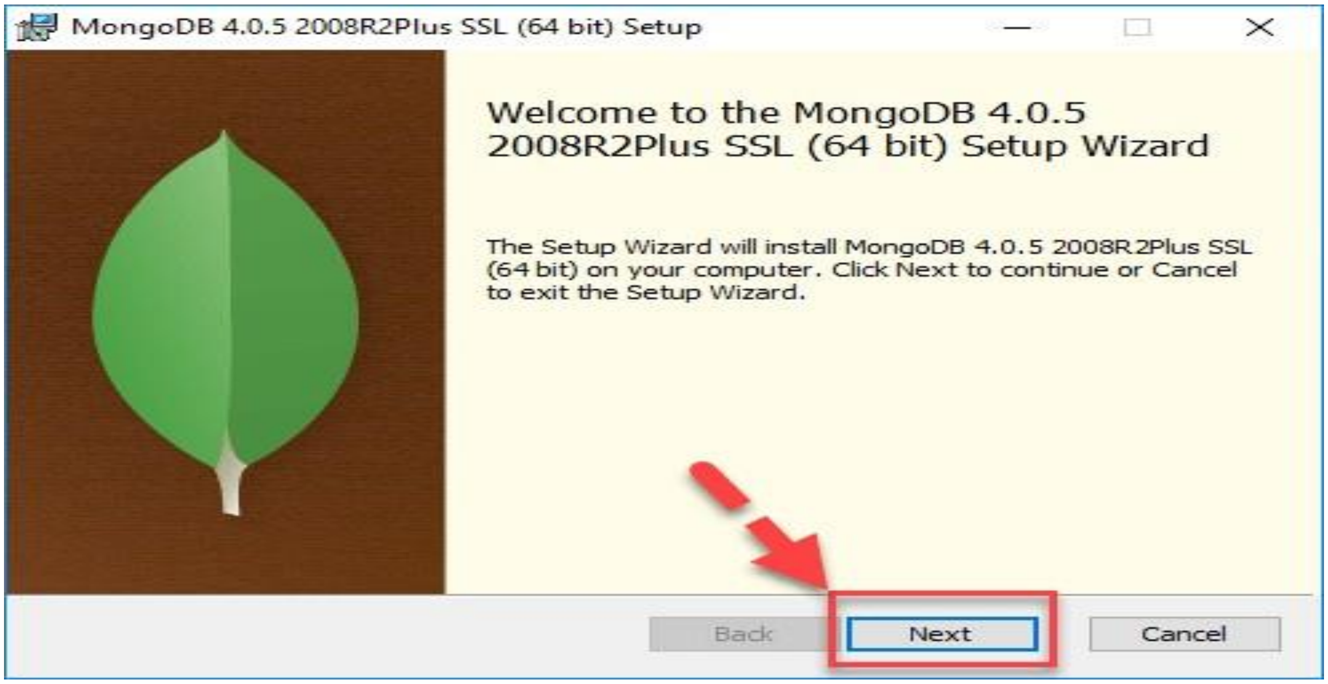
MongoDB Community Server
FEATURE RICH. DEVELOPER READY.

Version: 4.0.5 (current release)
OS: Windows 64-bit x64
Package: MSI

Download

https://fastdl.mongodb.org/win32/mongodb-win32-x86_64-2008plus-ssl-4.0.5-signed.msi

Step2:-Once download is complete open the msi file. Click Next in the startup screen

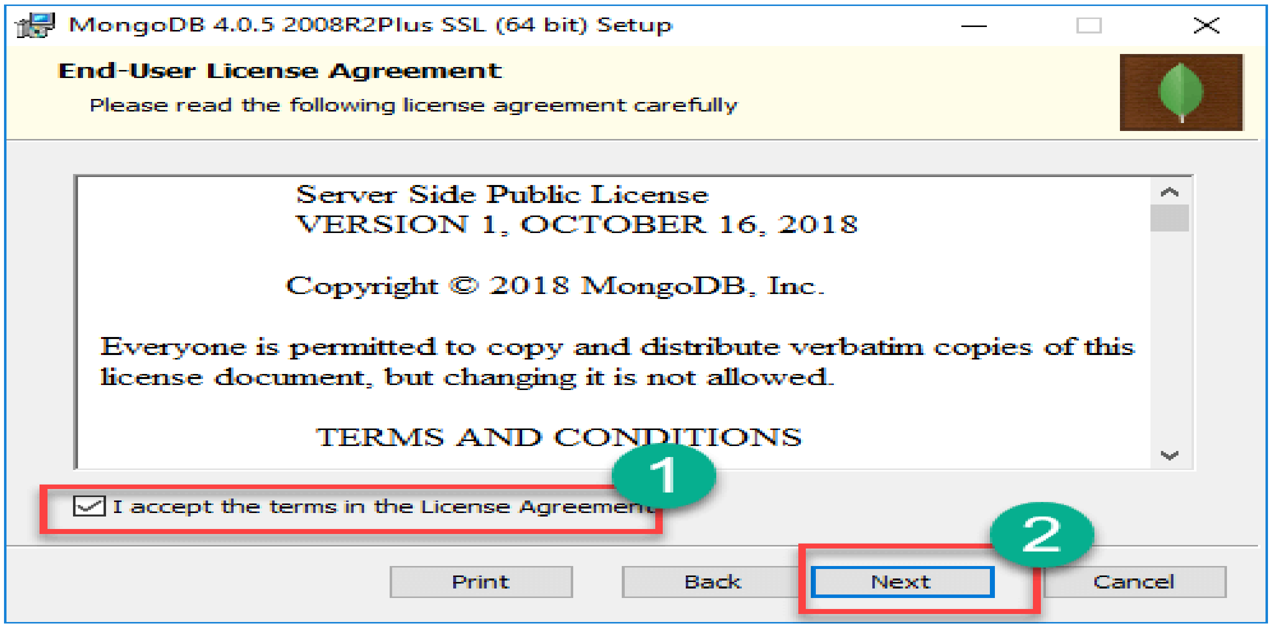


MongoDB 4.0.5 2008R2Plus SSL (64 bit) Setup

Welcome to the MongoDB 4.0.5 2008R2Plus SSL (64 bit) Setup Wizard

The Setup Wizard will install MongoDB 4.0.5 2008R2Plus SSL (64 bit) on your computer. Click Next to continue or Cancel to exit the Setup Wizard.

Step3:- 1. Accept the End-User License Agreement.
2. Click Next.


MongoDB 4.0.5 2008R2Plus SSL (64 bit) Setup

End-User License Agreement

Please read the following license agreement carefully

Server Side Public License
VERSION 1, OCTOBER 16, 2018

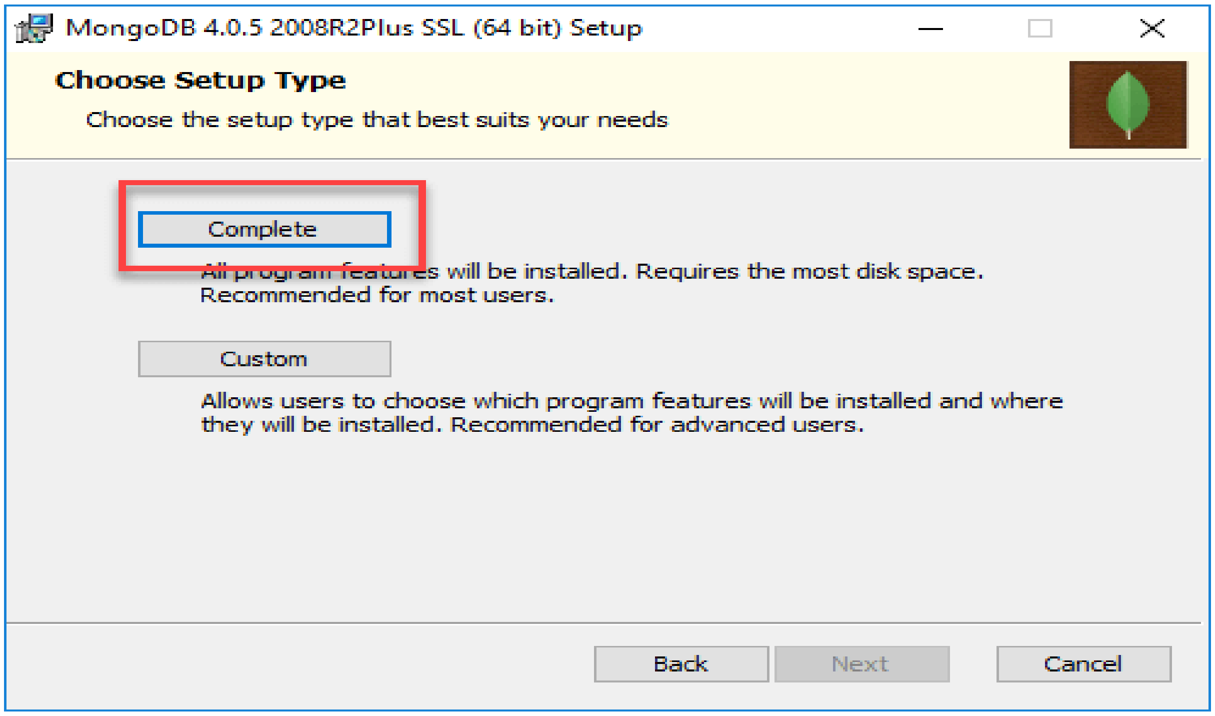
Copyright © 2018 MongoDB, Inc.

Everyone is permitted to copy and distribute verbatim copies of this license document, but changing it is not allowed.

TERMS AND CONDITIONS

[✓] I accept the terms in the License Agreement

Step4. Click on the "complete" button to install all of the components. The custom option can be used to install selective components or if you want to change the location of the installation.


MongoDB 4.0.5 2008R2Plus SSL (64 bit) Setup

Choose Setup Type

Choose the setup type that best suits your needs

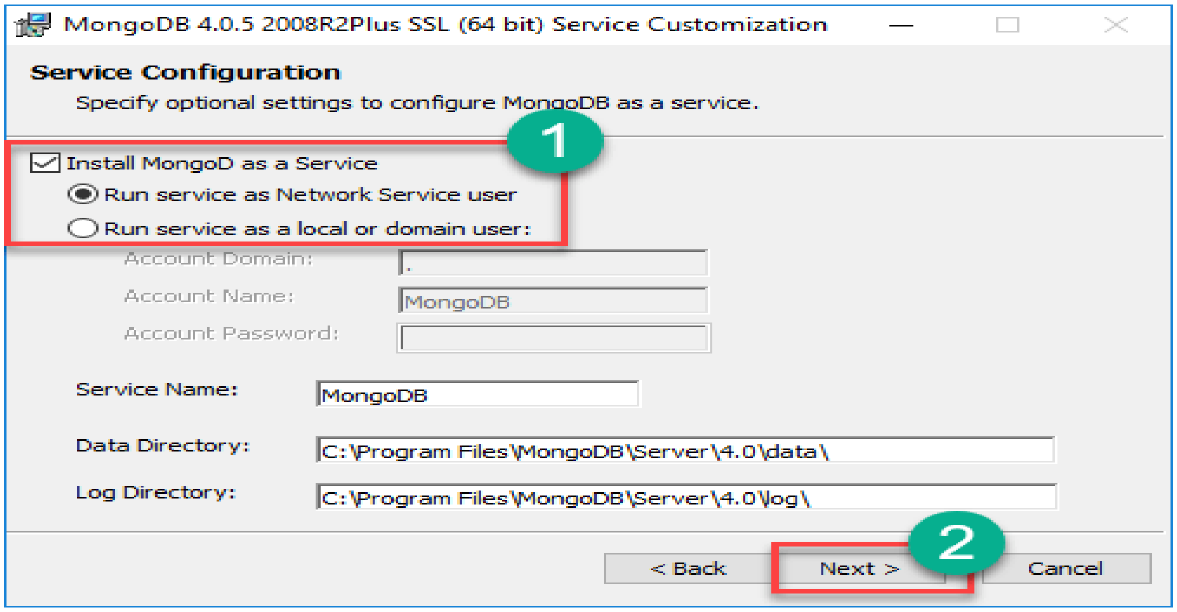
Complete

All program features will be installed. Requires the most disk space. Recommended for most users.

Custom

Allows users to choose which program features will be installed and where they will be installed. Recommended for advanced users.

Step 5. Select "Run service as Network Service user". make a note of the data directory, we'll need this later. Click Next


MongoDB 4.0.5 2008R2Plus SSL (64 bit) Service Customization

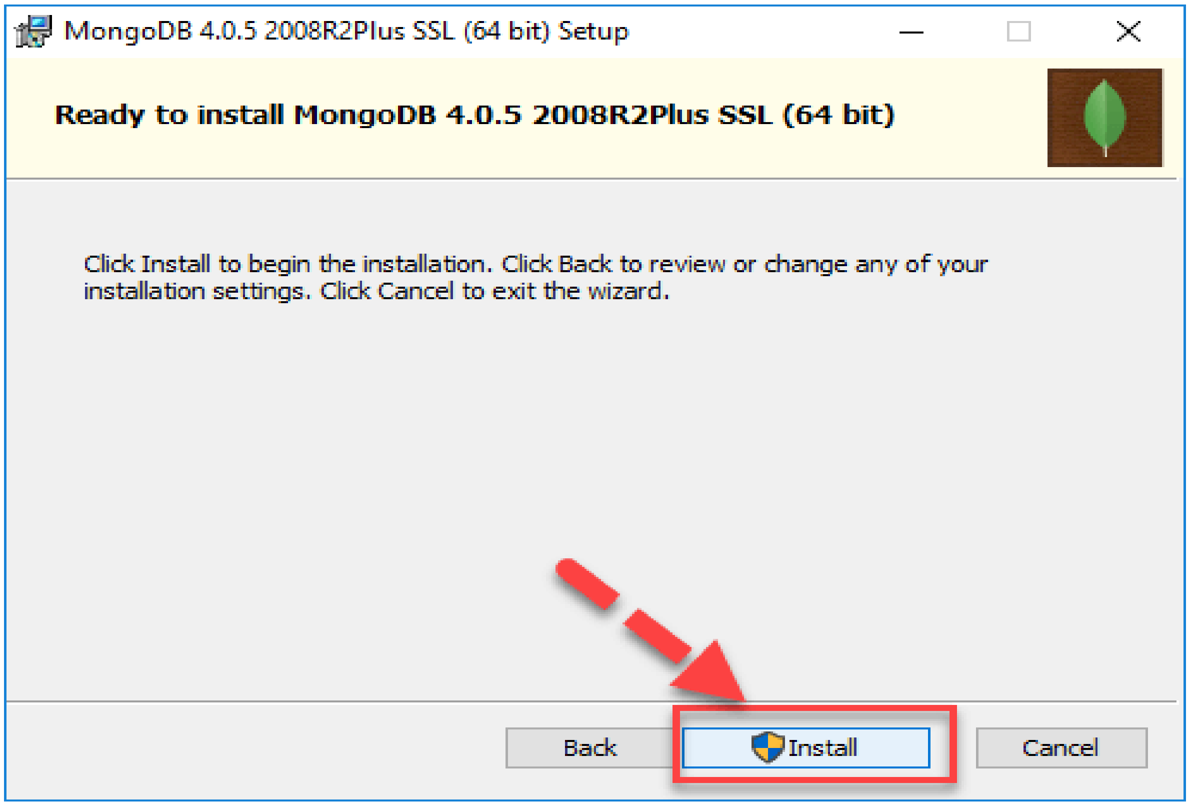
Service Configuration

Specify optional settings to configure MongoDB as a service.

- [✓] Install MongoD as a Service
  - [●] Run service as Network Service user
  - [ ] Run service as a local or domain user:
    - Account Domain: .
    - Account Name: MongoDB
    - Account Password:

Service Name:	MongoDB
Data Directory:	C:\Program Files\MongoDB\Server\4.0\data\
Log Directory:	C:\Program Files\MongoDB\Server\4.0\log\

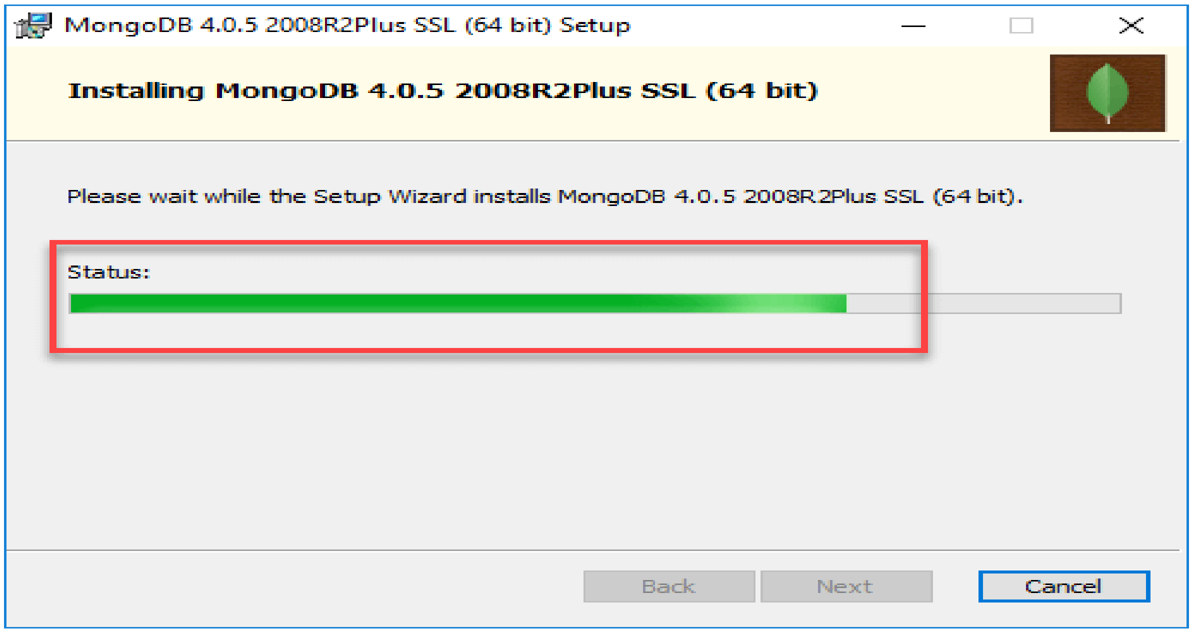
Step6:- Click on the Install button to start the installation.


MongoDB 4.0.5 2008R2Plus SSL (64 bit) Setup

Ready to install MongoDB 4.0.5 2008R2Plus SSL (64 bit)

Click Install to begin the installation. Click Back to review or change any of your installation settings. Click Cancel to exit the wizard.

Step7:- Installation begins. Click Next once completed



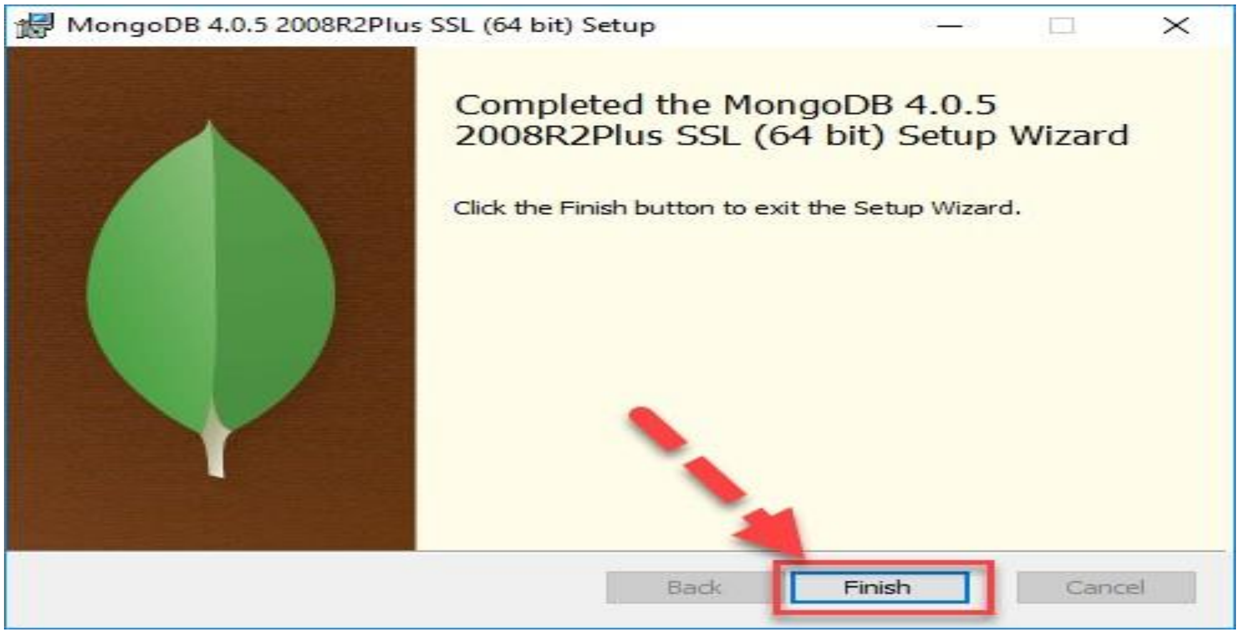
MongoDB 4.0.5 2008R2Plus SSL (64 bit) Setup

Installing MongoDB 4.0.5 2008R2Plus SSL (64 bit)

Please wait while the Setup Wizard installs MongoDB 4.0.5 2008R2Plus SSL (64 bit).

Status:

Step8:- Click on the Finish button to complete the installation



MongoDB 4.0.5 2008R2Plus SSL (64 bit) Setup

Completed the MongoDB 4.0.5 2008R2Plus SSL (64 bit) Setup Wizard

Click the Finish button to exit the Setup Wizard.

2. Create and drop database and collection?

RDBMS	MongoDB
Database	Database
Table	Collection
Record	Document
Columns	Fields/key value pairs
Index	Index
Joins	Embedded Documents
Primary Key	Primary key(_id is a identifier)

	Oracle	MongoDB
Database Server	Oracle	Mongod
Database Client	SQL PLUS	mongo

CREATE DATABASE:

USE DATABASE_NAME
EXAMPE: use kanthDB

```
> use kanthDB
switched to db kanthDB

> db
kanthDB
```

DROP Database :

Sysntax to drop database:
Db.dropDatabase();

```
> db.dropDatabase();
{ "ok" : 1 }
```

3. Insert, update , delete, query document

Insert:

```
> db.student.insert({_id:1,studRollno:'S101',studName:'kanth',grade:'I',Hobbies:'BookReading', Doj:'10-oct-
2012'});
WriteResult({ "nInserted" : 1 })
```

Select:

```
> db.student.find()
{ "_id" : 1, "studRollno" : "S101", "studName" : "kanth", "grade" : "I", "Hobbies" : "BookReading", "Doj" :
"10-oct-2012" }
{ "_id" : 2, "studRollno" : "S102", "studName" : "Ajay", "grade" : "II", "Hobbies" : "Hockey", "Doj" : "11-
oct-2012" }
```

Update:

```
db.student.update({studRollno:'S102'},{$set:{Hobbies:'Ice Hockey'}})
> db.student.find()
{ "_id" : 2, "studRollno" : "S102", "studName" : "Ajay", "grade" : "II", "Hobbies" : "Hockey", "Doj" : "11-
oct-2012" }

{ "_id" : 2, "studRollno" : "S102", "studName" : "Ajay", "grade" : "II", "Hobbies" : "Ice Hockey", "Doj" :
"11-oct-2012" }
```

Delete:

```
> db.student.remove({studRollno:'S101'})
{ "_id" : 2, "studRollno" : "S102", "studName" : "Ajay", "grade" : "II", "Hobbies" : "Ice Hockey", "Doj" :
"11-oct-2012" }
```