MCA-II SEMESTER
Title :DBMS LAB
PAPER-II

DBMS lab index

1. Create marketing company by using following Tables:

→ client_master

→ product_master

→ salesman_master

→ sales_order

→ sales_order_details

→ challan_details

2. Create manufacturing deals with various parts and various suppliers supply these parts by using following tables:

   → supplier (S)

   → parts(P)

   → projects(J)

   →SPJ

3. Create Airline System by using the following tables:

   → Flights

   → Aircraft

   → Certified

   → Employees

4. Create enterprise database system (departments and each department consists of employees) using the following tables:

   →DEPT

   → EMP

5. PL/SQL programs:

a) To check the given number is strong or not.

b) To check the given string is palindrome or not.

c) To swap two numbers without using third variable.

d) To generate multiplication tables for 2, 4, 6.

e) To display sum of even numbers and sum of odd numbers in the given range.

f) To check the given number is palindrome or not

g) The HRD manager has decided to raise the employee salary by 15%.write a pl/sql block to accept the employee number and update the salary of that employee. Display appropriate message based on the existence of the record in emp table.

h) To display top 10 rows in emp table based on their job and salary.

i) To raise the employee salary by 10%, for department number 30 people and also maintain the raised details in the raised table.

j) To update the salary of employee, who are not getting commission by 10%

k) write a pl/sql procedure to prepare an electricity bill

l) write a pl/sql procedure to prepare an telephone bill

m) write a pl/sql program to raise the employee salary by 10%, who are completed there 25 years of service and store the details at ppropriate tables (define the retair_emp table)

n) write a pl/sql procedure to evaluate the grade of a student

o) create an varray, which holds the employee phone numbers (at least three numbers)

p) Create an object to describe the details of address type data.

q) write a pl/sql procedure to read the data into the table as per the following description

ORACLE INRODUCTION:

It is a very large and multi-user database management system. Oracle is a relational database management system developed by 'Oracle Corporation'.

Oracle works to efficiently manage its resource, a database of information, among the multiple clients requesting and sending data in the network.

It is an excellent database server choice for client/server computing. Oracle supports all major operating systems for both clients and servers, including MSDOS, NetWare, UnixWare, OS/2 and most UNIX flavors.

History:

Oracle began in 1977 and celebrating its 32 wonderful years in the industry (from 1977 to 2009).

- 1977 - Larry Ellison, Bob Miner and Ed Oates founded Software Development Laboratories to undertake development work.
- 1979 - Version 2.0 of Oracle was released and it became first commercial relational database and first SQL database. The company changed its name to Relational Software Inc. (RSI).
- 1981 - RSI started developing tools for Oracle.
- 1982 - RSI was renamed to Oracle Corporation.
- 1983 - Oracle released version 3.0, rewritten in C language and ran on multiple platforms.

- 1984 - Oracle version 4.0 was released. It contained features like concurrency control - multi-version read consistency, etc.
- 1985 - Oracle version 4.0 was released. It contained features like concurrency control - multi-version read consistency, etc.
- 2007 - Oracle has released Oracle11g. The new version focused on better partitioning, easy migration etc.

Features:

- Concurrency
- Read Consistency
- Locking Mechanisms
- Quiesce Database
- Portability
- Self-managing database
- SQL*Plus
- ASM
- Scheduler
- Resource Manager
- Data Warehousing
- Materialized views
- Bitmap indexes
- Table compression

- Parallel Execution
- Analytic SQL
- Data mining
- Partitioning

<u>How to Write and execute sql, pl/sql commands/programs</u>:

1). Open your oracle application by the following navigation

Start->allprograms->oracleorahome.->application development->sql.

2). You will be asked for user name, pass word and host string

You have to enter user name, pass word and host string as given by the administrator. It will be different from one user to another user.

3). Upon successful login you will get SQL prompt (SQL>).

In two ways you can write your programs:

a). directly at SQL prompt

b). or in sql editor.

If you type your programs at sql prompt then screen will look like follow:

```
SQL> SELECT ename,empno,
2 sal from
3 emp;
```

where 2 and 3 are the line numbers and rest is the command /program……

to execute above program/command you have to press '/' then enter.

Here editing the program is somewhat difficult; if you want to edit the previous command then you have to open sql editor (by default it displays the sql buffer contents). By giving 'ed' at sql prompt.(this is what I mentioned as a second method to type/enter the program).

in the sql editor you can do all the formatting/editing/file operations directly by selecting menu options provided by it.

To execute the program which saved; do the following

```
SQL> @ programname.sql
```

Or

```
SQL> Run programname.sql
```

Then press '\' key and enter.

This how we can write, edit and execute the sql command and programs.

Always you have to save your programs in your own logins.

Background Theory

Oracle workgroup or server is the largest selling RDBMS product.it is estimated that the combined sales of both these oracle database product account for aroud 80% of the RDBMSsystems sold worldwide.

These products are constantly undergoing change and evolving. The natural language of this RDBMS product is ANSI SQL,PL/SQL a superset of ANSI SQL.oracle 8i and 9i also under stand SQLJ.

Oracle corp has also incorporated a full-fledged java virtual machine into its database engine.since both executable share the same memory space the JVM can communicate With the database engine with ease and has direct access to oracle tables and their data.

Oracle has many tools such as SQL * PLUS, Oracle Forms, Oracle Report Writer, Oracle Graphics etc.

- **SQL * PLUS**: The SQL * PLUS tool is made up of two distinct parts. These are
- **Interactive SQL:** Interactive SQL is designed for create, access and manipulate data structures like tables and indexes.
- **PL/SQL:** PL/SQL can be used to developed programs for different applications.
- **Oracle Forms:** This tool allows you to create a data entry screen along with the suitable menu objects. Thus it is the oracle forms tool that handles data gathering and data validation in a commercial application.
- **Report Writer:** Report writer allows programmers to prepare innovative reports using data from the oracle structures like tables, views etc. It is the report writer tool that handles the reporting section of commercial application.
- **Oracle Graphics:** Some of the data can be better represented in the form of pictures. The oracle graphics tool allows programmers to prepare graphs using data from oracle structures like tables, views etc.

SQL INRODUCTION:

SQL is followed by unique set of rules and guidelines called Syntax. This material gives you a quick start with SQL by listing all the basic SQL Syntax:

All the SQL statements start with any of the keywords like SELECT, INSERT, UPDATE, DELETE, ALTER, DROP, CREATE, USE, SHOW and all the statements end with a semicolon (;).

Important point to be noted is that SQL is **case insensitive**, which means SELECT and select have same meaning in SQL statements, but MySQL makes difference in table names. So if you are working with MySQL, then you need to give table names as they exist in the database.

SQL Commands:

SQL commands are instructions used to communicate with the database to perform specific task that work with data. SQL commands can be used not only for searching the database but also to perform various other functions like, for example, you can create tables, add data to tables, or modify data, drop the table, set permissions for users. SQL commands are grouped into four major categories depending on their functionality:

- **Data Definition Language (DDL)** - These SQL commands are used for creating, modifying, and dropping the structure of database objects. The commands are CREATE, ALTER, DROP, RENAME, and TRUNCATE.
- **Data Manipulation Language (DML)** - These SQL commands are used for storing, retrieving, modifying, and deleting data. These commands are SELECT, INSERT, UPDATE, and DELETE.
- **Transaction Control Language (TCL)** - These SQL commands are used for managing changes affecting the data. These commands are COMMIT, ROLLBACK, and SAVEPOINT.
- **Data Control Language (DCL)** - These SQL commands are used for providing security to database objects. These commands are GRANT and REVOKE.

SQL SELECT Statement:

```
SELECT column1, column2....columnN
FROM   table_name;
```

SQL DISTINCT Clause:

```
SELECT DISTINCT column1, column2....columnN
FROM   table_name;
```

SQL WHERE Clause:

```
SELECT column1, column2....columnN
FROM   table_name
WHERE CONDITION;
```

SQL AND/OR Clause:

```
SELECT column1, column2....columnN
FROM   table_name
WHERE CONDITION-1 {AND|OR} CONDITION-2;
```

SQL IN Clause:

```
SELECT column1, column2....columnN
FROM   table_name
WHERE column_name IN (val-1, val-2,...val-N);
```

SQL BETWEEN Clause:

```
SELECT column1, column2....columnN
FROM   table_name
WHERE column_name BETWEEN val-1 AND val-2;
```

SQL LIKE Clause:

```
SELECT column1, column2....columnN
FROM   table_name
WHERE column_name LIKE {PATTERN};
```

SQL ORDER BY Clause:

```
SELECT column1, column2....columnN
FROM   table_name
WHERE CONDITION
ORDER BY column_name {ASC|DESC};
```

SQL GROUP BY Clause:

```
SELECT SUM (column_name)
FROM   table_name
WHERE CONDITION
GROUP BY column_name;
```

SQL COUNT Clause:

```
SELECT COUNT (column_name)
FROM   table_name
WHERE CONDITION;
```

SQL HAVING Clause:

```
SELECT SUM (column_name)
FROM   table_name
WHERE CONDITION
GROUP BY column_name
HAVING (arithmetic function condition);
```

SQL CREATE TABLE Statement:

```
CREATE TABLE table_name(
column1 datatype,
column2 datatype,
column3 datatype,
.....
columnN datatype,
PRIMARY KEY( one or more columns )
);
```

SQL DROP TABLE Statement:

```
DROP TABLE table_name;
```

SQL CREATE INDEX Statement :

```
CREATE UNIQUE INDEX index_name
ON table_name ( column1, column2,...columnN);
```

SQL DROP INDEX Statement :

```
ALTER TABLE table_name
DROP INDEX index_name;
```

SQL DESC Statement :

```
DESC table_name;
```

SQL TRUNCATE TABLE Statement:

```
TRUNCATE TABLE table_name;
```

SQL ALTER TABLE Statement:

```
ALTER TABLE table_name {ADD|DROP|MODIFY} column_name
{data_ype};
```

SQL ALTER TABLE Statement (Rename) :

```
ALTER TABLE table_name RENAME TO new_table_name;
```

SQL INSERT INTO Statement:

```
INSERT INTO table_name( column1, column2....columnN)
```

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

SQL UPDATE Statement:

```
UPDATE table_name
SET column1 = value1, column2 = value2....columnN=valueN
[ WHERE  CONDITION ];
```

SQL DELETE Statement:

```
DELETE FROM table_name
WHERE  {CONDITION};
```

SQL CREATE DATABASE Statement:

```
CREATE DATABASE database_name;
```

SQL DROP DATABASE Statement:

```
DROP DATABASE database_name;
```

SQL USE Statement:

```
USE database_name;
```

SQL COMMIT Statement:

```
COMMIT;
```

SQL ROLLBACK Statement:

```
ROLLBACK;
```

Different data types in SQL

Exact Numeric Data Types:

DATA TYPE	FROM	TO
bigint	-9,223,372,036,854,775,808	9,223,372,036,854,775,807
int	-2,147,483,648	2,147,483,647
smallint	-32,768	32,767
tinyint	0	255
bit	0	1
decimal	-10^38 +1	10^38 -1
numeric	-10^38 +1	10^38 -1
money	-922,337,203,685,477.5808	+922,337,203,685,477.5807
smallmoney	-214,748.3648	+214,748.3647

Approximate Numeric Data Types:

DATA TYPE	FROM	TO
float	-1.79E + 308	1.79E + 308
real	-3.40E + 38	3.40E + 38

Date and Time Data Types:

DATA TYPE	FROM	TO
datetime	Jan 1, 1753	Dec 31, 9999
smalldatetime	Jan 1, 1900	Jun 6, 2079
date	Stores a date like June 30, 1991	
time	Stores a time of day like 12:30 P.M.	

Note: Here, datetime has 3.33 milliseconds accuracy where as smalldatetime has 1 minute accuracy.

Character Strings Data Types:

DATA TYPE	FROM	TO
char	char	Maximum length of 8,000 characters.(Fixed length non-Unicode characters)
varchar	varchar	Maximum of 8,000 characters.(Variable-length non-Unicode data).
varchar(max)	varchar(max)	Maximum length of 231characters, Variable-length non-Unicode data (SQL Server 2005 only).
text	text	Variable-length non-Unicode data

		with a maximum length of 2,147,483,647 characters.

Unicode Character Strings Data Types:

DATA TYPE	Description
nchar	Maximum length of 4,000 characters.(Fixed length Unicode)
nvarchar	Maximum length of 4,000 characters.(Variable length Unicode)
nvarchar(max)	Maximum length of 231characters (SQL Server 2005 only).(Variable length Unicode)
ntext	Maximum length of 1,073,741,823 characters. (Variable length Unicode)

Binary Data Types:

DATA TYPE	Description
binary	Maximum length of 8,000 bytes(Fixed-length binary data)
varbinary	Maximum length of 8,000 bytes.(Variable length binary data)
varbinary(max)	Maximum length of 231 bytes (SQL Server 2005 only). (Variable length Binary data)

image	Maximum length of 2,147,483,647 bytes. (Variable length Binary Data)

What is an Operator in SQL?

An operator is a reserved word or a character used primarily in an SQL statement's WHERE clause to perform operation(s), such as comparisons and arithmetic operations.

Operators are used to specify conditions in an SQL statement and to serve as conjunctions for multiple conditions in a statement.

- Arithmetic operators
- Comparison operators
- Logical operators
- Operators used to negate conditions

SQL Arithmetic Operators:

Assume variable a holds 10 and variable b holds 20, then:

Show Examples

Operator	Description	Example
+	Addition - Adds values on either side of the operator	a + b will give 30
-	Subtraction - Subtracts right hand operand from left hand operand	a - b will give -10

*	Multiplication - Multiplies values on either side of the operator	a * b will give 200
/	Division - Divides left hand operand by right hand operand	b / a will give 2
%	Modulus - Divides left hand operand by right hand operand and returns remainder	b % a will give 0

SQL Comparison Operators:

Assume variable a holds 10 and variable b holds 20, then:

Show Examples

Operator	Description	Example
=	Checks if the values of two operands are equal or not, if yes then condition becomes true.	(a = b) is not true.
!=	Checks if the values of two operands are equal or not, if values are not equal then condition becomes true.	(a != b) is true.
<>	Checks if the values of two operands are equal or not, if values are not equal then condition becomes true.	(a <> b) is true.

>	Checks if the value of left operand is greater than the value of right operand, if yes then condition becomes true.	(a > b) is not true.
<	Checks if the value of left operand is less than the value of right operand, if yes then condition becomes true.	(a < b) is 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.	(a >= b) is not 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.	(a <= b) is true.
!<	Checks if the value of left operand is not less than the value of right operand, if yes then condition becomes true.	(a !< b) is false.
!>	Checks if the value of left operand is not greater than the value of right operand, if yes then condition becomes true.	(a !> b) is true.

SQL Logical Operators:

Here is a list of all the logical operators available in SQL.

Show Examples

Operator	Description
ALL	The ALL operator is used to compare a value to all values in another value set.
AND	The AND operator allows the existence of multiple conditions in an SQL statement's WHERE clause.
ANY	The ANY operator is used to compare a value to any applicable value in the list according to the condition.
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.
EXISTS	The EXISTS operator is used to search for the presence of a row in a specified table that meets certain criteria.
IN	The IN operator is used to compare a value to a list of literal values that have been specified.
LIKE	The LIKE operator is used to compare a value to similar values using wildcard operators.
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.**
OR	The OR operator is used to combine multiple conditions in an SQL statement's WHERE clause.
IS NULL	The NULL operator is used to compare a value with a NULL value.
UNIQUE	The UNIQUE operator searches every row of a specified table for uniqueness (no duplicates).

CONSTRAINTS:

Constraints are the rules enforced on data columns on table. These are used to limit the type of data that can go into a table. This ensures the accuracy and reliability of the data in the database.

Constraints could be column level or table level. Column level constraints are applied only to one column, whereas table level constraints are applied to the whole table.

Following are commonly used constraints available in SQL.

- **NOT NULL Constraint:** Ensures that a column cannot have NULL value.
- **DEFAULT Constraint:** Provides a default value for a column when none is specified.
- **UNIQUE Constraint:** Ensures that all values in a column are different.

- **PRIMARY Key:** Uniquely identified each rows/records in a database table.
- **FOREIGN Key:** Uniquely identified a rows/records in any another database table.
- **CHECK Constraint:** The CHECK constraint ensures that all values in a column satisfy certain conditions.
- **INDEX:** Use to create and retrieve data from the database very quickly.

Constraints can be specified when a table is created with the CREATE TABLE statement or you can use ALTER TABLE statement to create constraints even after the table is created.

DROPPING CONSTRAINTS:

Any constraint that you have defined can be dropped using the ALTER TABLE command with the DROP CONSTRAINT option.

For example, to drop the primary key constraint in the EMPLOYEES table, you can use the following command:

```
ALTER TABLE EMPLOYEES DROP CONSTRAINT
EMPLOYEES_PK;
```

Some implementations may provide shortcuts for dropping certain constraints. For example, to drop the primary key constraint for a table in Oracle, you can use the following command:

```
ALTER TABLE EMPLOYEES DROP PRIMARY KEY;
```

Some implementations allow you to disable constraints. Instead of permanently dropping a constraint from the database, you may want to temporarily disable the constraint and then enable it later.

INTEGRITY CONSTRAINTS:

Integrity constraints are used to ensure accuracy and consistency of data in a relational database. Data integrity is handled in a relational database through the concept of referential integrity.

There are many types of integrity constraints that play a role in referential integrity (RI). These constraints include Primary Key, Foreign Key, Unique Constraints and other constraints mentioned above.

SQL JOIN TYPES:

There are different types of joins available in SQL:

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

Cycle - I

Aim: Marketing Company wishes to computerize their operations by using following tables.

1. Table Name**:** Client_Master

Description: This table stores the information about the clients.

<u>Creation:</u>

```
SQL>Create table Client_Master(
        client_no varchar2(6) primary key check(client_no like'c%'),
        name varchar2(10) not null,
        address1 varchar2(10),
        address2 varchar2(10),
        city varchar2(10),
        state varchar2(10),
        pincode number(6) not null,
        bal_due number(10,2));
Table created
SQL>desc Client_master;
```

Output:

Table	Column	Data Type	Length	Precision	Scale	Primary Key	Nullable	Default	Comment
CLIENT_MASTER	CLIENT_NO	Varchar2	6	-	-	1	-	-	-
	NAME	Varchar2	10	-	-	-	-	-	-
	ADDRESS1	Varchar2	10	-	-	-	✓	-	-
	ADDRESS2	Varchar2	10	-	-	-	✓	-	-
	CITY	Varchar2	10	-	-	-	✓	-	-
	STATE	Varchar2	10	-	-	-	✓	-	-
	PINCODE	Number	-	6	0	-	-	-	-
	BAL_DUE	Number	-	10	2	-	✓	-	-
									1 - 8

Insertion:

SQL>insert into Client_Master values (‘c00001’,’IvanBayros’,’ward’,’chruch’,’Bombay’,’Maharastra’,400054, 15000);

1 row created.

SQL>insert into Client_Master values ('c00002','Vandana','ward','chruch','Madras','Tamilnadu',780001,0);

1 row created.

```
SQL>insert into Client_Master values
('c00003','Basu','ward','chruch','Delhi','Delhi',400054,3000);
1 row created.
SQL>insert into Client_Master values
('c00004','Vijay','ward','chruch','Chennai','Tamilnadu',400074,2000);
1 row created.
SQL>insert into Client_Master values
('c00005','Raju','ward','chruch','vijayawada','A P',500074,4000);
1 row created.
SQL>insert into Client_Master values
('c00006','Ravi','delhi','noda','delhi','delhi',100001,2000);
1 row created.
SQL>insert into Client_Master values
('c00007','Rukmini','kaharagen','kaharagen','bombay','maharashtra',4
00050,null);
1 row created.

SQL> select * from client_master;
```

CLIENT_NO	NAME	ADDRESS1	ADDRESS2	CITY	STATE	PINCODE	BAL_DUE
c00001	IvanBayros	ward	chruch	Bombay	Maharastra	400054	15000
c00002	Vandana	ward	chruch	Madras	Tamilnadu	780001	0
c00003	Basu	ward	chruch	Delhi	Delhi	400054	3000
c00004	Vijay	ward	chruch	Chennai	Tamilnadu	400074	2000

c00005	Raju	ward	chruch	vijayawada	A P	500074	4000
c00006	Ravi	delhi	noda	delhi	delhi	100001	2000
c00007	Rukmini	kaharagen	kaharagen	bombay	maharashtr	400050	-

2. Table Name: Product_Master

Description: This table stores the information about products.

Creation:

```
SQL>create table Product_Master(
          product_no varchar2(6) primary key check(product_no
like'p%'),
          description varchar2(10) not null,
          profit_percent number(2,2) not null,
          unit_measure varchar2(10),
          qty_on_hand number(8),
          record_lvl number(8),
          sell_price number(8,2) not null check(sell_price<>0),
          cost_price number(8,2) not null check(cost_price<>0));
Table created
SQL>desc PRODUCT_MASTER;
```

Output:

Table	Column	Data Type	Length	Precision	Scale	Primary Key	Nullable	Default	Comment
PRODUCT_MASTER	PRODUCT_NO	Varchar2	6	-	-	1	-	-	-
	DESCRIPTION	Varchar2	10	-	-	-	-	-	-
	PROFIT_PERCENT	Number	-	2	2	-	-	-	-
	UNIT_MEASURE	Varchar2	10	-	-	-	✓	-	-
	QTY_ON_HAND	Number	-	8	0	-	✓	-	-
	RECORD_LVL	Number	-	8	0	-	✓	-	-
	SELL_PRICE	Number	-	8	2	-	-	-	-
	COST_PRICE	Number	-	8	2	-	-	-	-
									1 - 8

Insertion:

SQL>insert into Product_Master

values('p00001','1.44Flopys',0.50,'piece',100,20,525,500);

1 row created.

SQL>insert into Product_Master

values('p00002','Monitors',0.60,'piece',10,3,12000,11280);

1 row created.

```
SQL>insert into Product_Master
values('p00003','540HDD',0.40,'piece',10,3,8400,8000);
1 row created.
SQL>insert into Product_Master values('p00004','1.44
Drive',0.50,'piece',100,3,1050,500);
1 row created.
SQL>insert into Product_Master values('p00005','CD
Drive',0.50,'piece',10,3,1050,800);
1 row created.
SQL>insert into Product_Master
values('p03453','monitors',0.6,'piece',10,3,12000,11280);
1 row created.
SQL>insert into Product_Master
values('p06734','mouse',0.5,'piece',20,5,1050,1000);
1 row created.
```

SQL>Select * from Product_Master;

Output:

PRODUCT_NO	DESCRIPTION	PROFIT_PERCENT	UNIT_MEASURE	QTY_ON_HAND	RECORD_LVL	SELL_PRICE	COST_PRICE
p00001	1.44Flopys	.5	piece	100	20	525	500
p00002	Monitors	.6	piece	10	3	12000	11280
p00003	540HDD	.4	piece	10	3	8400	8000
p00004	1.44 Drive	.5	piece	100	3	1050	500

p00005	CD Drive	.5	piece	10	3	1050	800
p03453	monitors	.6	piece	10	3	12000	11280
p06734	mouse	.5	piece	20	5	1050	1000

3. Table Name: Salesman_Master

Description: This table stores the salesmen working in the company

Creation:

```
create table Salesman_Master(
salesman_id varchar2(6) primary key check(salesman_id like's%'),
name varchar2(10) not null,
address1 varchar2(10),
address2 varchar2(10),
city varchar2(10),
state varchar2(10),
pincode number(6) not null,
sal_amt number(8,2) not null check(sal_amt<>0),
target_amt number(6,2) not null check(target_amt<>0),
remarks varchar2(10));
Table created
SQL>desc Salesman_Master;
```

Output:

Table	Column	Data Type	Length	Precision	Scale	Primary Key	Nullable	Default	Comment
SALESMAN_MASTER	SALESMAN_ID	Varchar2	6	-	-	1	-	-	-
	NAME	Varchar2	10	-	-	-	-	-	-
	ADDRESS1	Varchar2	10	-	-	-	✓	-	-
	ADDRESS2	Varchar2	10	-	-	-	✓	-	-
	CITY	Varchar2	10	-	-	-	✓	-	-
	STATE	Varchar2	10	-	-	-	✓	-	-
	PINCODE	Number	-	6	0	-	-	-	-
	SAL_AMT	Number	-	8	2	-	-	-	-
	TARGET_AMT	Number	-	6	2	-	-	-	-
	REMARKS	Varchar2	10	-	-	-	✓	-	-
									1 - 10

Insertion:

```
SQL>insert into Salesman_Master values
('s00001','Kiran','A/14','Worli','Bombay','Maharastra',400002,3000,10
0,'Good');
1 row Created
```

SQL>insert into Salesman_Master values

('s00002','Manish','65','Nariman','Bombay','Maharastra',400001,3000,
200,'Good');

1 row Created

SQL>insert into Salesman_Master values

('s00003','Ravi','P-
7','Bandra','Bombay','Maharastra',4000032,3000,200,'Good');

1 row Created

SQL>insert into Salesman_Master values

('s00004','ashish','A/5','juhu','Bombay','Maharastra',400044,3500,200,
'Good');

1 row Created

SQL>insert into Salesman_Master values

('s00005','BALAJI','A/7','RAGA','Bombay','Maharastra',400020,4500,2
00,'Good');

1 row Created

SQL>Select * from Salesman_Master;

Output:

SALESMAN_ID	NAME	ADDRESS1	ADDRESS2	CITY	STATE	PINCODE	SAL_AMT	TARGET_AMT	REMARKS
s00001	Kiran	A/14	Worli	Bombay	Maharastra	400002	3000	100	Good

s00002	Manish	65	Nariman	Bombay	Maharastra	400001	3000	200	Good
s00003	Ravi	P-7	Bandra	Bombay	Maharastra	400032	3000	200	Good
s00004	ashish	A/5	juhu	Bombay	Maharastra	400044	3500	200	Good
s00005	BALAJI	A/7	RAGA	Bombay	Maharastra	400020	4500	200	Good

4. Table Name: Sales_order

Description: This table stores the information about orders

Creation:

SQL>create table Sales_Order(

s_order_no varchar2(6) primary key check(s_order_no like 'o%'),

s_order_date date,

client_no varchar2(6) references client_master(client_no),

delve_address varchar2(20),

salesman_no varchar2(6) references salesman_master(salesman_id),

delve_type varchar2(1) default 'f' check(delve_type in('p','f')),

billed_yn char(1),

delve_date date,

order_status varchar2(10) check(order_status

in('InProcess','Fulfilled', 'BackOrder',

'Cancelled')),

check(delve_date>s_order_date));

Table Created

SQL>desc Sales_Order;

Output:

Table	Column	Data Type	Length	Precision	Scale	Primary Key	Nullable	Default	Comment
SALES_ORDER	S_ORDER_NO	Varchar2	6	-	-	1	-	-	-
	S_ORDER_DATE	Date	7	-	-	-	✓	-	-
	CLIENT_NO	Varchar2	6	-	-	-	✓	-	-
	DELVE_ADDRESS	Varchar2	20	-	-	-	✓	-	-
	SALESMAN_NO	Varchar2	6	-	-	-	✓	-	-
	DELVE_TYPE	Varchar2	1	-	-	-	✓	'f'	-
	BILLED_YN	Char	1	-	-	-	✓	-	-
	DELVE_DATE	Date	7	-	-	-	✓	-	-

	ORDER_STATUS	Varchar2	10	-	-	-	✓	-	-
								1 - 9	

Insertion:

```
SQL>insert into Sales_Order values
('o19001','12-Jan-09','c00001','vjei block1','s00001','f','N','20-Jan-
09','InProcess');
1 row Created
SQL>insert into Sales_Order values
('o19002','25-Jan-09','c00002','vjei block2','s00002','p','N','27-Jan-
09','Cancelled');
1 row Created
SQL>insert into Sales_Order values
('o19003', '18-feb-09','c00003','vjei block3','s00003','f','Y','20-feb-
09','Fulfilled');
1 row Created
SQL>insert into Sales_Order values
('o19004', '03-apr-09','c00004','vjei block4','s00004','f','Y','07-apr-
09','Fulfilled');
1 row Created
SQL>insert into Sales_Order values
```

('o19005','25-MAR-15','c00005','vjei block2','s00005','p','N','27-MAR-15','Cancelled');

1 row Created

SQL> select * from Sales_Order;

Output:

S_ORDER_NO	S_ORDER_DATE	CLIENT_NO	DELVE_ADDRESS	SALESMAN_NO	DELVE_TYPE	BILLED_YN	DELVE_DATE	ORDER_STATUS
o19001	12-JAN-09	c00001	vjei block1	s00001	f	N	20-JAN-09	InProcess
o19002	25-JAN-09	c00002	vjei block2	s00002	p	N	27-JAN-09	Cancelled
o19003	18-FEB-09	c00003	vjei block3	s00003	f	Y	20-FEB-09	Fulfilled
o19004	03-APR-09	c00004	vjei block4	s00004	f	Y	07-APR-09	Fulfilled
o19005	25-MAR-15	c00005	vjei block2	s00005	p	N	27-MAR-15	Cancelled

5. Table Name: Sales_order_details

Description: This table stores the information about products ordered

Creation:

SQL>create table Sales_Order_Details(

```
s_order_no varchar2(6) references sales_order(s_order_no),
product_no varchar2(6) references product_master(product_no),
qty_ordered number(8),
qty_disp number(8),
product_rate number(10,2),
primary key(s_order_no,product_no));
Table Created
SQL>desc sales_order_details;
```

Output:

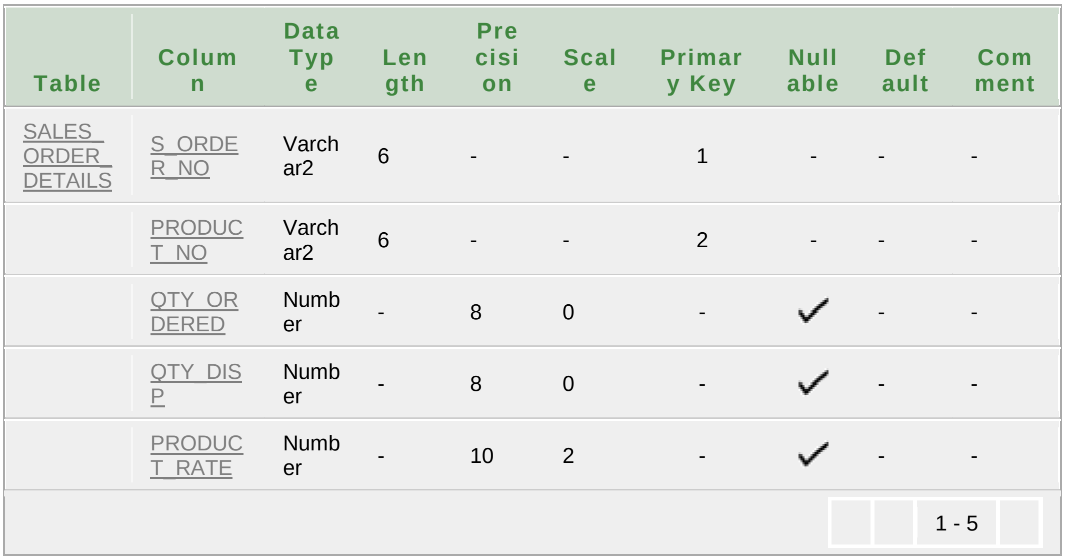


Table	Column	Data Type	Length	Precision	Scale	Primary Key	Nullable	Default	Comment
SALES_ORDER_DETAILS	S_ORDER_NO	Varchar2	6	-	-	1	-	-	-
	PRODUCT_NO	Varchar2	6	-	-	2	-	-	-
	QTY_ORDERED	Number	-	8	0	-	✓	-	-
	QTY_DISP	Number	-	8	0	-	✓	-	-
	PRODUCT_RATE	Number	-	10	2	-	✓	-	-

1 - 5

Insertion:

```
SQL>insert into Sales_Order_Details
values('o19001','p00001',4,4,525);
1 row Created
SQL>insert into Sales_Order_Details
values('o19002','p00002',2,1,8400);
1 row Created
SQL>insert into Sales_Order_Details
values('o19003','p00003',2,1,5250);
1 row Created
SQL>insert into Sales_Order_Details
values('o19004','p00004',4,4,525);
1 row Created
SQL>insert into Sales_Order_Details
values('o19005','p00005',5,5,5225);
1 row Created
SQL> select * from Sales_Order_Details;
```

Output:

S_ORDER_NO	PRODUCT_NO	QTY_ORDERED	QTY_DISP	PRODUCT_RATE
o19001	p00001	4	4	525
o19002	p00002	2	1	8400
o19003	p00003	2	1	5250

o19004	p00004	4	4	525
o19005	p00005	5	5	5225

6. Table Name: Challan_Master

Description: This table stores the information about challans made for orders.

Creation:

SQL>create table Challan_Master(

challan_no varchar2(6) primary key check(challan_no like'ch%'),

s_order_no varchar2(6) references sales_order(s_order_no),

challan_date date,

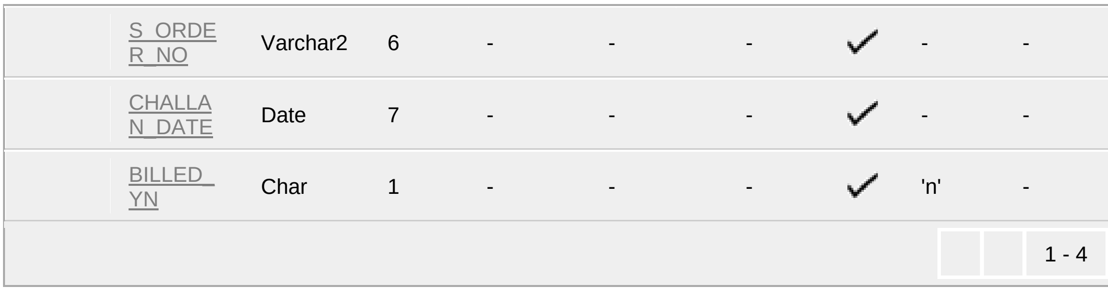
billed_yn char(1) default 'n' check(billed_yn in('y','n'))) ;

Table created

SQL>desc challan_master;.

Output:

Table	Column	Data Type	Length	Precision	Scale	Primary Key	Nullable	Default	Comment
CHALLAN_MASTER	CHALLAN_NO	Varchar2	6	-	-	1	-	-	-



	S_ORDER_NO	Varchar2	6	-	-	-	✓	-	-	
	CHALLAN_DATE	Date	7	-	-	-	✓	-	-	
	BILLED_YN	Char	1	-	-	-	✓	'n'	-	
										1 - 4

Insertion:

SQL>insert into Challan_Master values('ch0001','o19001','12-Jan-96','y');

1 row Created

SQL>insert into Challan_Master values('ch0002','o19002','12-Jan-96','y');

1 row Created

SQL>insert into Challan_Master values('ch0003','o19003','12-Jan-96','y');

1 row Created

SQL>insert into Challan_Master values('ch0004','o19003','12-Jan-96','y');

1 row Created

SQL> select * from Challan_Master;

Output:

CHALLAN_NO	S_ORDER_NO	CHALLAN_DATE	BILLED_YN
ch0001	o19001	12-JAN-96	y
ch0002	o19002	12-JAN-96	y
ch0003	o19003	12-JAN-96	y
ch0004	o19003	12-JAN-96	y

7. Table Name: Challan_Details

Description: This table stores the information about challan details.

Creation:

SQL>create table Challan_Details(

challan_no varchar2(6) references challan_master(challan_no),

product_no varchar2(6) references product_master(product_no),

qty_disp number(4,2) not null,

primary key(challan_no,product_no)) ;

Table created

SQL>desc challan_details;

Output:

Table	Column	Data Type	Length	Precision	Scale	Primary Key	Nullable	Default	Comment
CHALLAN_DETAILS	CHALLAN_NO	Varchar2	6	-	-	1	-	-	-
	PRODUCT_N	Varch	6	-	-	2	-	-	-

	O	ar2							
	QTY_D ISP	Num ber	-	4	2	-	-	-	-
								1 - 3	

Insertion:

SQL>insert into Challan_Details values('ch0001','p00001',4);

1 row created

SQL>insert into Challan_Details values('ch0002','p00002',3);

1 row created

SQL>insert into Challan_Details values('ch0003','p00003',2);

1 row created

SQL>insert into Challan_Details values('ch0004','p00004',1);

1 row created

SQL> select * from Challan_Details;

Output:

CHALLAN_NO	PRODUCT_NO	QTY_DISP
ch0001	p00001	4
ch0002	p00002	3
ch0003	p00003	2
ch0004	p00004	1

Cycle – I
Queries

The following queries by using above tables.

1. Retrieve the list of names and cities of all the clients.

```
SQL>Select name, city from CLIENT_MASTER;
```

Output:

NAME	CITY
IvanBayros	Bombay
Vandana	Madras
Basu	Delhi
Vijay	Chennai
Raju	vijayawada
Ravi	delhi
Rukmini	bombay

2. List the various products available from product_master.

```
SQL>Select product_no from PRODUCT_MASTER;
```

Output:

PRODUCT_NO
p00001
p00002
p00003
p00004
p00005

p03453
p06734

3. Find out the clients who stay in a city whose second letter is 'a'.

SQL>Select city from CLIENT_MASTER where city like '_a%';

Output:

CITY
Madras

4. Find the list of all clients who stay in the city ' CHENNAI' or 'DELHI'

SQL> Select client_no,name,city from CLIENT_MASTER where city in('delhi','chennai');

Output:

CLIENT_NO	NAME	CITY
c00006	Ravi	delhi

5. List all the clients located at 'CHENNAI'.
SQL>Select client_no,name,city from CLIENT_MASTER where city='chennai';

Output:

CLIENT_NO	NAME	CITY
c00004	Vijay	Chennai

6. Print the information from sales order as the order the places in the month of January.
SQL>SELECT S_ORDER_NO,CLIENT_NO,SALESMAN_NO FROM SALES_ORDER WHERE S_ORDER_DATE LIKE '%-JAN-%';

Output:

S_ORDER_NO	CLIENT_NO	SALESMAN_NO
o19001	c00001	s00001
o19002	c00002	s00002

7. Find the products with description as 'Floppy Drive' and 'Pen drive'.
SQL>SELECT PRODUCT_NO,DESCRIPTION FROM PRODUCT_MASTER WHERE DESCRIPTION LIKE '%Flopys'

Output:

PRODUCT_NO	DESCRIPTION
p00001	1.44Flopys

8. Find the products whose selling price is grater than 2000 and less than or equal to 5000.
SQL>Select product_no,description,sell_price,cost_price from PRODUCT_MASTER where sell_price>2000;

Output:

PRODUCT_NO	DESCRIPTION	SELL_PRICE	COST_PRICE
p00002	Monitors	12000	11280
p00003	540HDD	8400	8000
p03453	monitors	12000	11280

9. Find the products whose selling price is more than 1500 and also find the new selling price as original selling price *15.

SQL>Select sell_price,sell_price*15 as new_sell_price from PRODUCT_MASTER where sell_price>1500;

Output:

SELL_PRICE	NEW_SELL_PRICE
12000	180000
8400	126000
12000	180000

10. Find the products in the sorted order of their description.
SQL>Select product_no,description from PRODUCT_MASTER order by description;

Output:

PRODUCT_NO	DESCRIPTION
p00004	1.44 Drive
p00001	1.44Flopys
p00003	540HDD
p00005	CD Drive
p00002	Monitors
p03453	monitors
p06734	mouse

11. Divide the cost of product '540 HDD' by difference between its price and 100.

SQL>Select sell_price-100 as sell_price from PRODUCT_MASTER where description='540HDD';

Output:

SELL_PRICE
8300

12. List the product number, description, sell price of products whose description begin with letter 'M'.

    SQL>Select product_no,description,sell_price from PRODUCT_MASTER where description like 'm%';

    Output:

PRODUCT_NO	DESCRIPTION	SELL_PRICE
p03453	monitors	12000
p06734	mouse	1050

13. List all the orders that were cancelled in the month of March.

    SQL>Select s_order_no,order_status from sales_order where s_order_date like '%MAR%' and order_status='Cancelled';

    Output:

S_ORDER_NO	ORDER_STATUS
o19005	Cancelled

14. Count the total number of orders.

SQL>Select count(*) from sales_order;

Output:

COUNT(*)
5

15. Calculate the average price of all the products.

SQL>Select avg(sell_price) from PRODUCT_MASTER;

Output:

AVG(SELL_PRICE)
5153.571428571428571428571428571428571428571428571432857143

16. Determine the maximum and minimum product prices.

SQL>Select max(sell_price),min(sell_price) from PRODUCT_MASTER;

Output:

MAX(SELL_PRICE)	MIN(SELL_PRICE)
12000	525

17. Count the number of products having price grater than or equal to 1500.

SQL>Select count(product_no) from PRODUCT_MASTER where sell_price>=1500;

Output:

COUNT(PRODUCT_NO)
3

18. Find all the products whose quantity on hand is less than reorder level.

SQL>Select product_no,descrIption from PRODUCT_MASTER where qty_on_hand<record_lvl;

Output:

NO ROWS SELECTED

19. Find out the challan details whose quantity dispatch is high.

SQL>Select challan_no from challan_details where qty_disp=(select max(qty_disp) from challan_details);

Output:

CHALLAN_NO
ch0001

20. Find out the order status of the sales order, whose order delivery is maximum in the month of March.

    SQL>Select order_status from sales_order s1, sales_order_details s where s1.s_order_no=s.s_order_no and s1.s_order_date like '%MAR%' and qty_disp=(select max(qty_disp) from sales_order_details);

Output:

ORDER_STATUS
Cancelled

21. Find out the total sales made by the each salesman.

    SQL>Select sum(qty_disp * product_rate), s.salesman_no from sales_order s,sales_order_details s1 where s.s_order_no= s1 .s_order_no group by s.salesman_no;

Output:

SUM(QTY_DISP*PRODUCT_RATE)	SALESMAN_NO
26125	s00005
2100	s00001
8400	s00002
5250	s00003
2100	s00004

22. Find the total revenue gained by the each product sales in the period of Q1 and Q2 of year 2006.

SQL>Select p.product_no,sum(qty_disp*sell_price-cost_price) from PRODUCT_MASTER p, sales_order s, sales_order_details s1 where p.product_no=s1.product_no and s1.s_order_no=s.s_order_no and to_char(s_order_date,'q') in(1,2) group by p.product_no;

Output:

PRODUCT_NO	SUM(QTY_DISP*SELL_PRICE-COST_PRICE)
p00003	400
p00001	1600
p00002	720
p00004	3700

p00005	4450

23. Print the description and total qty sold for each product.

```
SQL>Select description,sum(qty_disp) from PRODUCT_MASTER
p,sales_order_details s where p.product_no=s.product_no group by
p.description;
```

Output:

DESCRIPTION	SUM(QTY_DISP)
540HDD	1
CD Drive	5
1.44Flopys	4
1.44 Drive	4
Monitors	1

24. Find the value of each product sold.

```
SQL>Select sum(product_rate),product_no from sales_order_details
group by product_no;
```

Output:

SUM(PRODUCT_RATE)	PRODUCT_NO
5250	p00003
525	p00001
8400	p00002
525	p00004
5225	p00005

25. Calculate the average qty sold for each client that has a maximum order value of 1,50,000.

SQL>Select max(qty_disp*product_rate),product_no from sales_order_details group by product_no;

Output:

MAX(QTY_DISP*PRODUCT_RATE)	PRODUCT_NO
5250	p00003
2100	p00001
8400	p00002
2100	p00004
26125	p00005

26. List the products which has highest sales.

SQL>Select s_order_no, avg(qty_disp), sum(qty_ordered*product_rate) total_sales from sales_order_details group by s_order_no having max(qty_ordered * product_rate)>10000;

Output:

S_ORDER _NO	AVG(QTY_D ISP)	TOTAL_SA LES
o19002	1	16800
o19003	1	10500
o19005	5	26125

27. Find out the products and their quantities that will have to deliver in the current month.

SQL>Select p.product_no,s1.delve_date from product_master p,sales_order_details s,sales_order s1 where p.product_no=s.product_no and s.s_order_no=s1.s_order_no and s1.delve_date between '1-FEB-09' and '28-FEB-09';

Output:

PRODUCT_NO	DELVE_DATE
p00003	20-FEB-09

28. Find the product number and descriptions of moving products.

    SQL>Select distinct p.product_no,description from PRODUCT_MASTER p,sales_order_details s where p.product_no=s.product_no;

    Output:

PRODUCT_NO	DESCRIPTION
p00005	CD Drive
p00002	Monitors
p00003	540HDD
p00004	1.44 Drive
p00001	1.44Flopys

29. Find the names of clients who have purchased ‘CD DRIVE’.

    SQL>select name from client_master c,product_master p, sales_order s,sales_order_details s1 where p.product_no=s1.product_no and s1.s_order_no=s.s_order_no and s.client_no=c.client_no and p.description='CD Drive';

Output:

NAME
Raju

30. List the product numbers and sales order numbers of customers having quantity ordered less than 5 from the order details for the product ‘1.44 Floppies’.

SQL>Select p.product_no,s.s_order_no from sales_order s, PRODUCT_MASTER p,sales_order_details s1 where s.s_order_no=s1.s_order_no and p.product_no=s1.product_no and s1.qty_ordered<5 and p.description='1.44Flopys';

Output:

PRODUCT_NO	S_ORDER_NO
p00001	o19001

31. Find the product numbers and descriptions of non-moving products.

SQL>Select distinct p.product_no from PRODUCT_MASTER p,sales_order_details s where p.product_no not in (select product_no from sales_order_details);

Output:

PRODUCT_NO
p03453
p06734

32. Find the customer names and address for the clients, who placed the order ‘019001’.

SQL>Select c.client_no,name,address1 from CLIENT_MASTER c,sales_order s where s.client_no=c.client_no and s_order_no='o19001';

Output:

CLIENT_NO	NAME	ADDRESS1
c00001	IvanBayros	ward

33. Find the client names who have placed orders before the month of May, 2009.

SQL>Select name,c.client_no from CLIENT_MASTER c,sales_order s where c.client_no=s.client_no and s_order_date between '1-JAN-09' and '30-APR-09';

Output:

NAME	CLIENT_NO
IvanBayros	c00001
Vandana	c00002
Basu	c00003
Vijay	c00004

34. Find the names of clients who have placed orders worth of 1000 or more.
    SQL>Select c.client_no,name from CLIENT_MASTER c,sales_order s,sales_order_details s1 where c.client_no=s.client_no and s.s_order_no=s1.s_order_no and s1.product_rate>=1000;

Output:

CLIENT_NO	NAME
c00002	Vandana
c00003	Basu
c00005	Raju

35. Find out if the product is '1.44 drive' is ordered by any client and print the client number, name to whom it is sold.
    SQL>Select c.client_no,c.name from CLIENT_MASTER c,sales_order s,PRODUCT_MASTER p,sales_order_details s1

```
where p.product_no=s1.product_no and
s1.s_order_no=s.s_order_no and s.client_no=c.client_no and
p.description='1.44 Drive';
```

Output:

CLIENT_NO	NAME
c00004	Vijay

Cycle-II

Aim: A Manufacturing Company deals with various parts and various suppliers supply these parts. It consists of three tables to record its entire information. Those are as follows

S(SNO,SNAME,CITY,STATUS)

P(PNO,PNAME,COLOR,WEIGTH,CITY,COST)

SP(SNO,PNO,QTY)

J(JNO,JNAME,CITY)

SPJ(SNO,PNO,JNO,QTY)

1. Table Name**:** Suppliers (S)

Description: This table stores the information about the suppliers.

Creation:

```
SQL>create table S(
sno varchar2(5) primary key,
sname varchar2(15),
status number(2),
city varchar2(10));
Table created
SQL>desc S;
```

Output:

Table	Column	Data Type	Length	Precision	Scale	Primary Key	Nullable	Default	Comment
S	SNO	Varchar2	5	-	-	1	-	-	-
	SNAME	Varchar2	15	-	-	-	✓	-	-
	STATUS	Number	-	2	0	-	✓	-	-
	CITY	Varchar2	10	-	-	-	✓	-	-

<u>Insertion:</u>

SQL>insert into s values('S1','Smith',20,'London');

1 row created.

SQL>insert into s values('S2','Jones',10,'Paris');

1 row created.

SQL>insert into s values('S3','Blake',30,'Paris') ;

1 row created.

SQL>insert into S values('S4','Clark',20,'London');

1 row created.

SQL>insert into S values('S5','Adams',30,'Athens');

1 row created.

SQL>insert into S values('S6','raj',20,'paris');

```
1 row created.
```

```
SQL>select * from S;
```

Output:

SNO	SNAME	STATUS	CITY
S1	Smith	20	London
S2	Jones	10	Paris
S3	Blake	30	Paris
S4	Clark	20	London
S5	Adams	30	Athens
S6	raj	20	paris

6 rows
returned

2. Table Name**:** PART (P)

Description: This table stores the information about the parts.

<u>Creation:</u>

```
create table P(
pno varchar2(5) primary key,
pname varchar2(10),
color varchar2(10),
weight number(6,2),
city varchar2(10),
cost   number(5));
Table created
SQL>desc P;
```

Output:

Table	Column	Data Type	Length	Precision	Scale	Primary Key	Nullable	Default	Comment
P	PNO	Varchar2	5	-	-	1	-	-	-
	PNAME	Varchar2	10	-	-	-	✓	-	-
	COLOR	Varchar2	10	-	-	-	✓	-	-
	WEIGHT	Number	-	6	2	-	✓	-	-
	CITY	Varchar2	10	-	-	-	✓	-	-
	Cost	Number	-	5	-	-	✓	-	-

<u>Insertion:</u>

```
SQL>insert into p values('P1','Nut','Red',12,'London',50);
1 row created.
SQL>insert into p values('P2','Bolt','Green',17,'Paris',70);
1 row created.
SQL>insert into p values('P3','Screw','Blue',17,'Rome',80) ;
```

```
1 row created.
SQL>insert into p values('P4','Screw','Red',14,'London',80);
1 row created.
SQL>insert into p values('P5','Cam','Blue',12,'Paris',90);
1 row created.
SQL>insert into p values('P6','Cog','Red',19,'London',70);
1 row created.
SQL>select * from P;
```

Output:

PNO	PNAME	COLOR	WEIGHT	CITY	cost
P1	Nut	Red	12	London	50
P2	Bolt	Green	17	Paris	70
P3	Screw	Blue	17	Rome	80
P4	Screw	Red	14	London	80
P5	Cam	Blue	12	Paris	90
P6	Cog	Red	19	London	70

6 rows returned

3. Table Name**: SUPPLIERS_PART (SP)**

Description: This table stores the information about the suppliers_parts.

<u>**Creation:**</u>

create table SP(

sno varchar2(5)references S(sno),

pno varchar2(5) references P(pno),

qty number(5),

primary key(sno,pno));

Table created

SQL>desc SP;

Output:

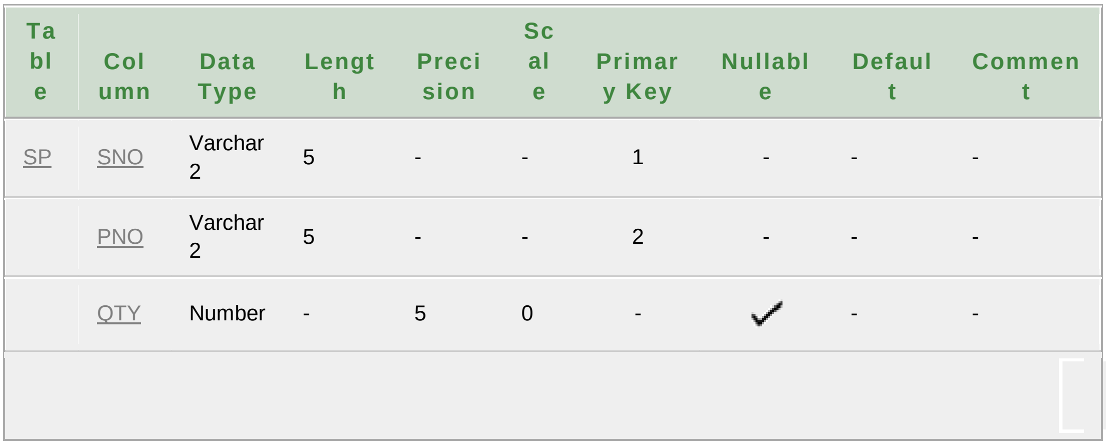


Table	Column	Data Type	Length	Precision	Scale	Primary Key	Nullable	Default	Comment
SP	SNO	Varchar 2	5	-	-	1	-	-	-
	PNO	Varchar 2	5	-	-	2	-	-	-
	QTY	Number	-	5	0	-	✓	-	-

Insertion:

```
SQL>insert into SP values('S1','P1',100);

1 row created.

SQL>insert into SP values('S1','P2',200);

1 row created.

SQL>insert into SP values('S2','P1',100);

1 row created.

SQL>insert into SP values('S3','P2',200);

1 row created.

SQL>insert into SP values('S4','P3',100);

1 row created.

SQL>insert into SP values('S3','P4',400);

1 row created.

SQL>insert into SP values('S4','P4',500);

1 row created.

SQL>insert into SP values('S6','P4',100);

1 row created.

SQL>insert into SP values('S2','P3',400);

1 row created.

SQL>insert into SP values('S2','P5',100);

1 row created.

SQL>insert into SP values('S3','P3',200);
```

```
1 row created.
SQL>insert into SP values('S4','P6',300);
1 row created.
SQL>insert into SP values('S5','P2',200);
1 row created.
SQL>insert into SP values('S5','P5',500);
1 row created.
SQL>insert into SP values('S5','P6',200);
1 row created.
SQL>insert into SP values('S5','P1',100);
1 row created.
SQL>insert into SP values('S5','P3',200);
1 row created.
SQL>insert into SP values('S5','P4',800);
1 row created.
SQL>select * from SP;
```

Output:

SNO	PNO	QTY
S1	P1	100
S1	P2	200

S2	P1	100
S3	P2	200
S4	P3	100
S3	P4	400
S4	P4	500
S6	P4	100
S2	P3	400
S2	P5	100
S3	P3	200
S4	P6	300
S5	P2	200
S5	P5	500
S5	P6	200
S5	P1	100
S5	P3	200
S5	P4	800

18 rows
returned

4. Table Name**:** PROJECTS (J)

Description: This table stores the information about the projects.

Creation:

```
create table J(
jno varchar2(5) primary key,
jname varchar2(10),
city varchar2(10));
Table created
SQL>desc J;
```

Output:

Table	Column	Data Type	Length	Precision	Scale	Primary Key	Nullable	Default	Comment
J	JNO	Varchar2	5	-	-	1	-	-	-
	JNAME	Varchar2	10	-	-	-	✓	-	-
	CITY	Varchar2	10	-	-	-	✓	-	-

Insertion:

```
SQL>insert into J values('J1','Sorter','Paris');
1 row created.
SQL>insert into J values('J2','Display','Rome');
1 row created.
```

```
SQL>insert into J values('J3','OCR','Athens');
1 row created.
SQL>insert into J values('J4','Console','Athens');
1 row created.
SQL>insert into J values('J5','RAID','London');
1 row created.
SQL>insert into J values('J6','EDS','Oslo');
1 row created.
SQL>insert into J values('J7','Tape','London');
1 row created.
SQL>select * from J;
```

Output:

JNO	JNAME	CITY
J1	Sorter	Paris
J2	Display	Rome
J3	OCR	Athens
J4	Console	Athens
J5	RAID	London
J6	EDS	Oslo
J7	Tape	London

7 rows returned

5. Table Name**:** Suppliers_Part_proJects (SPJ)

Description: This table stores the information about the suppliers_part_projects .

Creation:

create table SPJ(

sno varchar2(5)references S(sno),

pno varchar2(5) references P(pno),

jno varchar2(5) references J(jno),

qty number(5),

primary key(sno,pno,jno));

Table created

SQL>desc SPJ;

Output:

Table	Column	Data Type	Length	Precision	Scale	Primary Key	Nullable	Default	Comment
SPJ	SNO	Varchar2	5	-	-	1	-	-	-
	PNO	Varchar2	5	-	-	2	-	-	-
	JNO	Varchar2	5	-	-	3	-	-	-
	QTY	Number	-	5	0	-	✓	-	-

Insertion:

```
SQL>insert into SPJ values('S1','P1','J1',200);

1 row created.

SQL>insert into SPJ values('S1','P1','J4',700);

1 row created.

SQL>insert into SPJ values('S2','P3','J1',400);

1 row created.

SQL>insert into SPJ values('S2','P3','J2',200);

1 row created.

SQL>insert into SPJ values('S2','P3','J3',200);

1 row created.

SQL>insert into SPJ values('S2','P3','J4',500);

1 row created.

SQL>insert into SPJ values('S2','P3','J5',600);

1 row created.

SQL>insert into SPJ values('S2','P3','J6',400);

1 row created.

SQL>insert into SPJ values('S2','P3','J7',800);

1 row created.
```

```
SQL>insert into SPJ values('S2','P5','J2',100);
1 row created.
SQL>insert into SPJ values('S3','P3','J1',200);
1 row created.
SQL>insert into SPJ values('S3','P4','J2',500);
1 row created.
SQL>insert into SPJ values('S4','P6','J3',300);
1 row created.
SQL>insert into SPJ values('S4','P6','J7',300);
1 row created.
SQL>insert into SPJ values('S5','P2','J2',200);
1 row created.
SQL>insert into SPJ values('S5','P2','J4',100);
1 row created.
SQL>insert into SPJ values('S5','P5','J5',500);
1 row created.
SQL>insert into SPJ values('S5','P5','J7',100);
1 row created.
SQL>insert into SPJ values('S5','P6','J2',200);
1 row created.
SQL>insert into SPJ values('S5','P1','J4',100);
1 row created.
```

```
SQL>insert into SPJ values('S5','P3','J4',200);
1 row created.
SQL>insert into SPJ values('S5','P4','J4',800);
1 row created.
SQL>insert into SPJ values('S5','P5','J4',400);
1 row created.
SQL>insert into SPJ values('S5','P6','J4',500);
1 row created.
```

```
SQL>select * from SPJ;
```

Output:

SNO	PNO	JNO	QTY
S1	P1	J1	200
S1	P1	J4	700
S2	P3	J1	400
S2	P3	J2	200
S2	P3	J3	200
S2	P3	J4	500
S2	P3	J5	600
S2	P3	J6	400

S2	P3	J7	800
S2	P5	J2	100
S3	P3	J1	200
S3	P4	J2	500
S4	P6	J3	300
S4	P6	J7	300
S5	P2	J2	200
S5	P2	J4	100
S5	P5	J5	500
S5	P5	J7	100
S5	P6	J2	200
S5	P1	J4	100
S5	P3	J4	200
S5	P4	J4	800
S5	P5	J4	400
S5	P6	J4	500

24 rows returned

Cycle-II
Queries

The following queries by using above tables.

36. Get Suppliers Names for Suppliers who supply at least one red part.

```
SQL>select sname from s where sno in(select sno from sp where
pno in(select pno from p where color='Red'));
```

Output:

SNAME
Blake
Clark
Adams
raj
Jones
Smith

37. Get Suppliers Names for Suppliers who do not supply part ‘P2’

```
SQL>select sname from s where sno in(select sno from spj where
pno in(select pno from p where pno!='P2'));
```

Output:

SNAME
Smith
Jones
Blake
Clark
Adams

38. Using Group by with Having Clause, Get the part numbers for all the parts supplied by more than one supplier.

SQL> select pno,count(sno) from sp group by pno having count(sno)>1;

Output:

PNO	COUNT(SNO)
P4	4
P1	3
P2	3
P3	4
P6	2
P5	2

39. Get supplier numbers for suppliers with status value less the current max status value.

SQL>select sno from s where status<(select max(status) from s);

Output:

SNO
S1
S2
S4
S6

4 rows returned

40. Get the total quantity of the part 'P2' supplied.

SQL> select sum(qty)as Total from sp where pno='P2';

Output:

TOTAL
600

41. Get the part color, supplied by the supplier 'S1'

SQL> select color from P, SP where P.PNO=SP.PNO AND

SNO='S1';

Output:

COLOR
Red
Green

42. Get the names of the parts supplied by the supplier ‘Smith’ and “Black”

SQL> select p.pname from p where p.pno in (select distinct sp.pno FROM sp ,s WHERE sp.sno=s.sno and s.sname='Blake' INTERSECT select distinct sp.pno FROM sp ,s WHERE sp.sno=s.sno and s.sname='Smith');

Output:

PNAME
Bolt

43. Get the Project numbers, whose parts are not in Red Color, from London.

SQL>select jno from j where jno not in (select jno from spj where pno in (select pno from P where color='Red')) and city='London';

Output:

JNO
J5

44. Get the suppliers located from the same city.
SQL>Select distinct e.sno,e.city from s e,s s1 where e.city>s1.city;
Output:

SNO	CITY
S2	Paris
S3	Paris
S1	London
S4	London

45. Get the suppliers, who does not supply any part.

SQL>select sno,sname from s where sno not in(select sno from spj);
Output:

SNO	SNAME
S6	raj

46. Find the pnames of parts supplied by London Supplier and by no one else.

SQL>select pname from p where pno in (select pno from spj where sno in(select sno from s where city='London'));

Output:

PNAME
Nut
Cog

47. Find the sno's of suppliers who charge more for some part than the average cost of that part.

SQL>select sno from s where sno in (select sno from spj where pno in (select pno from p where cost>(select avg(cost) from p)));

Output:

SNO
S3
S5
S2

48. Find the sid's of suppliers who supply only red parts.

SQL>select sno from spj minus (select sno from spj where pno not in (select pno from p where color='Red'));

Output:

SNO
S1
S4

49. Find the sid’s of suppliers who supply a red and a green part.

    SQL> select distinct sp.sno FROM sp ,p WHERE sp.pno=p.pno and p.color='Green' INTERSECT select distinct sp.sno FROM sp ,p WHERE sp.pno=p.pno and p.color='Red';

Output:

SNO
S1
S3
S5

50. Find the sid’s of suppliers who supply a red or green part.

    SQL> select distinct sp.sno FROM sp ,p WHERE sp.pno=p.pno and p.color='Green' UNION select distinct sp.sno FROM sp ,p WHERE sp.pno=p.pno and p.color='Red';

Output:

SNO
S1
S2
S3
S4
S5
S6

6 rows returned

Cycle – III

Aim: An Airline System would like to keep track their information by using the following relations.

1. Table Name**:** Flights

Description: This table stores the information about the Flights.

Creation:

```
SQL>create table flights(
flno number(4,0) primary key,
origin varchar2(20),
destination varchar2(20),
distance number(6,0),
departs date,
arrives date,
price number(7,2));

Table created
SQL>desc flights;
```

Output:

Table	Column	Data Type	Length	Precision	Scale	Primary Key	Nullable	Default	Comment
FLIGHTS	FLNO	Number	-	4	0	1	-	-	-
	ORIGIN	Varchar2	20	-	-	-	✓	-	-
	DESTINATION	Varchar2	20	-	-	-	✓	-	-
	DISTANCE	Number	-	6	0	-	✓	-	-
	DEPARTS	Date	7	-	-	-	✓	-	-
	ARRIVES	Date	7	-	-	-	✓	-	-
	PRICE	Number	-	7	2	-	✓	-	-
									1 - 7

Insertion:

SQL> insert into flights values(99,'Los Angeles','Washington D.C.', 2308,to_date('2005/04/12 09:30 ','yyyy/mm/dd hh24:mi'),to_date('2005/04/12 21:40','yyyy/mm/dd hh24:mi'),235.98)

1 row created.

```
SQL> insert into flights values(13,'Los Angeles','Chicago.',
1749,to_date('2005/04/12 09:30 ','yyyy/mm/dd
hh24:mi'),to_date('2005/04/12 21:40','yyyy/mm/dd hh24:mi'),220.98)
1 row created.

SQL> insert into flights values(346,'Los Angeles','Dallas.',
1251,to_date('2005/04/12 09:30 ','yyyy/mm/dd
hh24:mi'),to_date('2005/04/12 21:40','yyyy/mm/dd hh24:mi'),225.43)
1 row created.

SQL> insert into flights values(387,'Los Angeles',’Boston.',
2606,to_date('2005/04/12 09:30 ','yyyy/mm/dd
hh24:mi'),to_date('2005/04/12 21:40','yyyy/mm/dd hh24:mi'),261.56)
1 row created.

SQL> insert into flights values(7,'Los
Angeles','Sydney.',7487,to_date('2005/04/12 09:30 ','yyyy/mm/dd
hh24:mi'),to_date('2005/04/12 21:40','yyyy/mm/dd hh24:mi'),1278.56)
1 row created.
```

```
SQL> insert into flights values(2,'Los Angeles',’Tokya', 5478,to_date('2005/04/12 09:30 ','yyyy/mm/dd hh24:mi'),to_date('2005/04/12 21:40','yyyy/mm/dd hh24:mi'),780.99)
1 row created.

SQL> insert into flights values(33,'Los Angeles',’Honolulu’, 2551,to_date('2005/04/12 09:30 ','yyyy/mm/dd hh24:mi'),to_date('2005/04/12 21:40','yyyy/mm/dd hh24:mi'),375.23)
1 row created.

SQL> insert into flights values(34,'Los Angeles',’Honolulu’, 2551,to_date('2005/04/12 09:30 ','yyyy/mm/dd hh24:mi'),to_date('2005/04/12 21:40','yyyy/mm/dd hh24:mi'),425.98)
1 row created.

SQL> insert into flights values(76,'Chicago',’ Los Angeles’, 1749,to_date('2005/04/12 09:30 ','yyyy/mm/dd hh24:mi'),to_date('2005/04/12 21:40','yyyy/mm/dd hh24:mi'),220.98)
1 row created.
```

```
SQL> insert into flights values(68,'Chicago',’New York’, 802,to_date('2005/04/12 09:30 ','yyyy/mm/dd hh24:mi'),to_date('2005/04/12 21:40','yyyy/mm/dd hh24:mi'),202.45)
1 row created.

SQL> insert into flights values(7789,'Madison',’ Detroit’, 319,to_date('2005/04/12 09:30 ','yyyy/mm/dd hh24:mi'),to_date('2005/04/12 21:40','yyyy/mm/dd hh24:mi'),120.33)
1 row created.

SQL> insert into flights values(701,'Detroit',’ New York’, 470,to_date('2005/04/12 09:30 ','yyyy/mm/dd hh24:mi'),to_date('2005/04/12 21:40','yyyy/mm/dd hh24:mi'),180.56)
1 row created.

SQL> insert into flights values(702,' Madison ',’ New York’, 789,to_date('2005/04/12 09:30 ','yyyy/mm/dd hh24:mi'),to_date('2005/04/12 21:40','yyyy/mm/dd hh24:mi'),202.34)
1 row created.
```

```
SQL> insert into flights values(4884,' Madison ',’ Chicago’, 84,to_date('2005/04/12 09:30 ','yyyy/mm/dd hh24:mi'),to_date('2005/04/12 21:40','yyyy/mm/dd hh24:mi'),112.45)
1 row created.

SQL> insert into flights values(2223,' Madison ',’ Pittsburgh’, 517,to_date('2005/04/12 09:30 ','yyyy/mm/dd hh24:mi'),to_date('2005/04/12 21:40','yyyy/mm/dd hh24:mi'),189.98)
1 row created.

SQL> insert into flights values(5694,' Madison ',’ Minneapolis’, 247,to_date('2005/04/12 09:30 ','yyyy/mm/dd hh24:mi'),to_date('2005/04/12 21:40','yyyy/mm/dd hh24:mi'),120.11)
1 row created.

SQL> insert into flights values(304,' ‘ Minneapolis’,’ New York’, 991,to_date('2005/04/12 09:30 ','yyyy/mm/dd hh24:mi'),to_date('2005/04/12 21:40','yyyy/mm/dd hh24:mi'),101.56)
1 row created
```

SQL> insert into flights values(149,' ‘Pittsburgh’,’ New York’, 303,to_date('2005/04/12 09:30 ','yyyy/mm/dd hh24:mi'),to_date('2005/04/12 21:40','yyyy/mm/dd hh24:mi'),116.5)

1 row created

SQL>select * from flights;

Output:

FLNO	ORIGIN	DESTINATION	DISTANCE	DEPARTS	ARRIVES	PRICE
99	Los Angeles	Washington D.C.	2308	12-APR-05	12-APR-05	235.98
13	Los Angeles	Chicago.	1749	12-APR-05	12-APR-05	220.98
346	Los Angeles	Dallas.	1251	12-APR-05	12-APR-05	225.43
387	Los Angeles	Boston	2606	12-APR-05	12-APR-05	261.56
7	Los Angeles	Sydney.	7487	12-APR-05	12-APR-05	1278.56
2	Los Angeles	Tokya	5478	12-APR-05	12-APR-05	780.99
33	Los Angeles	Honolulu	2551	12-APR-05	12-APR-05	375.23

34	Los Angeles	Honolulu	2551	12-APR-05	12-APR-05	425.98
76	Chicago	Los Angeles	1749	12-APR-05	12-APR-05	220.98
68	Chicago	New York	802	12-APR-05	12-APR-05	202.45
7789	Madison	Detroit	319	12-APR-05	12-APR-05	120.33
701	Detroit	New York	470	12-APR-05	12-APR-05	180.56
702	Madison	New York	789	12-APR-05	12-APR-05	202.34
4884	Madison	Chicago	84	12-APR-05	12-APR-05	112.45
2223	Madison	Pittsburgh	517	12-APR-05	12-APR-05	189.98
5694	Madison	Minneapolis	247	12-APR-05	12-APR-05	120.11
304	Minneapolis	New York	991	12-APR-05	12-APR-05	101.56
149	Pittsburgh	New York	303	12-APR-05	12-APR-05	116.5

18 rows returned

2. Table Name**:** Aircraft

Description: This table stores the information about the Aircraft.

Creation:

```
SQL>create table Aircraft(

aid number(9,0) primary key,

aname varchar2(30),

crusing_range number(6,0));


Table created

SQL>desc Aircraft;
```

Output:

Table	Column	Data Type	Length	Precision	Scale	Primary Key	Nullable	Default	Comment
AIRCRAFT	AID	Number	-	9	0	1	-	-	-
	ANAME	Varchar2	30	-	-	-	✓	-	-
	CRUSING_RANGE	Number	-	6	0	-	✓	-	-
									1 - 3

Insertion:

```
SQL>insert into Aircraft values(1,'Boeing 747-400',8430);
1 row created

SQL>insert into Aircraft values(2,'Boeing 737-800',3383);
1 row created

SQL>insert into Aircraft values(3,'Airbus A340-300',7120);
1 row created

SQL>insert into Aircraft values(4,'British Aerospace Jetstream 41',1502);
1 row created

SQL>insert into Aircraft values(5,'Embraer ERJ-145',1530);
1 row created

SQL>insert into Aircraft values(6,'SAAB 340',2128);
1 row created
```

```
SQL>insert into Aircraft values(7,'Piper Archer III',520);
1 row created

SQL>insert into Aircraft values(8,'Tupolev 154',4103);
1 row created

SQL>insert into Aircraft values(19,'Schwitzer 2-33',30);
1 row created

SQL>insert into Aircraft values(9,'Lockheed L1011',6900);
1 row created

SQL>insert into Aircraft values(10,'Boeing 757-300',4010);
1 row created

SQL>insert into Aircraft values(11,'Boeing 777-300',6441);
1 row created

SQL>insert into Aircraft values(12,'Boeing 767-400ER',6475);
1 row created
```

```
SQL>insert into Aircraft values(13,'Airbus A320',2605);

1 row created


SQL>insert into Aircraft values(14,'Airbus A319',1805);

1 row created


SQL>insert into Aircraft values(15,'Boeing 727',1504);

1 row created


SQL> select * from Aircraft;
```

Output:

AID	ANAME	CRUSING_RANGE
1	Boeing 747-400	8430
2	Boeing 737-800	3383
3	Airbus A340-300	7120
4	British Aerospace Jetstream 41	1502
5	Embraer ERJ-145	1530
6	SAAB 340	2128

7	Piper Archer III	520
8	Tupolev 154	4103
19	Schwitzer 2-33	30
9	Lockheed L1011	6900
10	Boeing 757-300	4010
11	Boeing 777-300	6441
12	Boeing 767-400ER	6475
13	Airbus A320	2605
14	Airbus A319	1805
15	Boeing 727	1504

16 rows returned

3. Table Name**:** Employees

Description: This table stores the information about theEmployees.

Creation:

```
SQL>create table Employees(
eid number(9,0) primary key,
ename varchar2(30),
salary number(10,2) );
```

Table created

SQL>desc Employees;

Output:

Table	Column	Data Type	Length	Precision	Scale	Primary Key	Nullable	Default	Comment
EMPLOYEES	EID	Number	-	9	0	1	-	-	-
	ENAME	Varchar2	30	-	-	-	✓	-	-
	SALARY	Number	-	10	2	-	✓	-	-
									1 - 3

Insertion:

SQL> insert into employees values(242518965,'James Smith',120433);

```
1 row created.

SQL> insert into employees values(141582651,'Mary Johnson',178345);

1 row created.

SQL> insert into employees values(11564812,'John Williams',153972);

1 row created.


SQL> insert into employees values(567354612,'Lisa Walker',256481);

1 row created.


SQL> insert into employees values(552455318,'Larry West',101745);

1 row created.

SQL> insert into employees values(550156548,'Karen Scott',205187);

1 row created.


SQL> insert into employees values(390487451,'Lawrence Sperry',212156);
```

```
1 row created.

SQL> insert into employees values(274878974,'Michael Miller',99890);
1 row created.

SQL> insert into employees values(254099823,'Patricia Jones',24450);
1 row created.

SQL> insert into employees values(356187925,'Robert Brown',44740);
1 row created.

SQL> insert into employees values(355548984,'Angela Martinez',212156);
1 row created.

SQL> insert into employees values(310454876,'Joseph Thompson',212156);
```

```
1 row created.

SQL> insert into employees values(489456522,'Linda Davis',127984);
1 row created.

SQL> insert into employees values(489221823,'Richard Jackson',23980);
1 row created.

SQL> insert into employees values(548977562,'William Ward',84476);
1 row created.

SQL> insert into employees values(310454877,'Chad Stewart',46);
1 row created.

SQL> insert into employees values(142519864,'Betty Adams',227489);
1 row created.
```

```
SQL> insert into employees values(269734834,'George Wright',289950);

1 row created.

SQL> insert into employees values(287321212,'Michael Miller',48090);

1 row created.


SQL> insert into employees values(552455348,'Dorthy Lewis',92013);

1 row created.


SQL> insert into employees values(248965255,'Barbara Wilson',43723);

1 row created.


SQL> insert into employees values(159542516,'William Moore',48250);

1 row created.
```

```
SQL> insert into employees values(348121549,'Haywood Kelly',32899);
1 row created.

SQL> insert into employees values(90873519,'lizabeth Taylor',32021);
1 row created.

SQL> insert into employees values(486512566,'David Anderson',743001);
1 row created.

SQL> insert into employees values(619023588,'Jennifer Thomas',54921);
1 row created.

SQL> insert into employees values(15645489,'Donald King',18050);
1 row created.
```

```
SQL> insert into employees values(556784565,'Mark Young',205187);

1 row created.

SQL> insert into employees values(573284895,'Eric Cooper',114323);

1 row created.

SQL> insert into employees values(574489456,'William Jones',105743);

1 row created.

SQL> insert into employees values(574489457,'Milo Brooks',20);

1 row created.

SQL> insert into employees values(242518965,'James Smith',120433);

1 row created

SQL> select * from employees;
```

Output:

EID	ENAME	SALARY
242518965	James Smith	120433
141582651	Mary Johnson	178345
11564812	John Williams	153972
567354612	Lisa Walker	256481
552455318	Larry West	101745
550156548	Karen Scott	205187
390487451	Lawrence Sperry	212156
274878974	Michael Miller	99890
254099823	Patricia Jones	24450
356187925	Robert Brown	44740
355548984	Angela Martinez	212156
310454876	Joseph Thompson	212156
489456522	Linda Davis	127984
489221823	Richard Jackson	23980
548977562	William Ward	84476
310454877	Chad Stewart	46
142519864	Betty Adams	227489

269734834	George Wright	289950
287321212	Michael Miller	48090
552455348	Dorthy Lewis	92013
248965255	Barbara Wilson	43723
159542516	William Moore	48250
348121549	Haywood Kelly	32899
90873519	lizabeth Taylor	32021
486512566	David Anderson	743001
619023588	Jennifer Thomas	54921
15645489	Donald King	18050
556784565	Mark Young	205187
573284895	Eric Cooper	114323
574489456	William Jones	105743
574489457	Milo Brooks	20

31 rows returned

4. Table Name**:** Certified

Description: This table stores the information about the Certified.

<u>Creation:</u>

```
SQL> create table Certified(
  2  eid number(9,0),
  3  aid number(9,0),
  4  primary key(eid,aid),
  5  foreign key(eid) references employees,
  6  foreign key(aid) references aircraft);
```

```
Table created.
```

```
SQL> desc Certified;
```

Output:

Table	Column	Data Type	Length	Precision	Scale	Primary Key	Nullable	Default	Comment
CERTIFIED	EID	Number	-	9	0	1	-	-	-
	AID	Number	-	9	0	2	-	-	-

Insertion:

```
SQL> insert into Certified values(11564812,2);
1 row created.

SQL> insert into Certified values(11564812,10);
1 row created.

SQL> insert into Certified values(90873519,6);
1 row created.

SQL> insert into Certified values(141582651,2);
1 row created.

SQL> insert into Certified values(141582651,10);
1 row created.

SQL> insert into Certified values(141582651,12);
1 row created.
```

```
SQL> insert into Certified values(142519864,1);
1 row created.

SQL> insert into Certified values(142519864,2);
1 row created.

SQL> insert into Certified values(142519864,3);
1 row created.

SQL> insert into Certified values(142519864,7);
1 row created.

SQL> insert into Certified values(142519864,10);
1 row created.

SQL> insert into Certified values(142519864,11);
1 row created.

SQL> insert into Certified values(142519864,12);
1 row created.
```

```
SQL> insert into Certified values(142519864,13);

1 row created.


SQL> insert into Certified values(159542516,5);

1 row created.


SQL> insert into Certified values(159542516,7);

1 row created.


SQL> insert into Certified values(242518965,2);

1 row created.


SQL> insert into Certified values(242518965,10);

1 row created.


SQL> insert into Certified values(269734834,1);

1 row created.


SQL> insert into Certified values(269734834,2);

1 row created.
```

```
SQL> insert into Certified values(269734834,3);

1 row created.


SQL> insert into Certified values(269734834,4);

1 row created.


SQL> insert into Certified values(269734834,5);

1 row created.


SQL> insert into Certified values(269734834,6);

1 row created.


SQL> insert into Certified values(269734834,7);

1 row created.


SQL> insert into Certified values(269734834,8);

1 row created.


SQL> insert into Certified values(269734834,9);

1 row created.
```

```
SQL> insert into Certified values(269734834,10);
1 row created.

SQL> insert into Certified values(269734834,11);
1 row created.

SQL> insert into Certified values(269734834,12);
1 row created.

SQL> insert into Certified values(269734834,13);
1 row created.

SQL> insert into Certified values(269734834,14);
1 row created.

SQL> insert into Certified values(269734834,15);
1 row created.

SQL> insert into Certified values(274878974,10);
1 row created.
```

```
SQL> insert into Certified values(274878974,12);

1 row created.


SQL> insert into Certified values(310454876,8);

1 row created.


SQL> insert into Certified values(310454876,9);

1 row created.


SQL> insert into Certified values(355548984,8);

1 row created.


SQL> insert into Certified values(355548984,9);

1 row created.


SQL> insert into Certified values(356187925,6);

1 row created.


SQL> insert into Certified values(390487451,3);

1 row created.
```

```
SQL> insert into Certified values(390487451,13);

1 row created.


SQL> insert into Certified values(390487451,14);

1 row created.


SQL> insert into Certified values(548977562,7);

1 row created.


SQL> insert into Certified values(550156548,1);

1 row created.


SQL> insert into Certified values(550156548,12);

1 row created.


SQL> insert into Certified values(552455318,2);

1 row created.


SQL> insert into Certified values(552455318,7);

1 row created.
```

```
SQL> insert into Certified values(552455318,14);
1 row created.

SQL> insert into Certified values(556784565,2);
1 row created.

SQL> insert into Certified values(556784565,3);
1 row created.

SQL> insert into Certified values(556784565,5);
1 row created.

SQL> insert into Certified values(567354612,1);
1 row created.

SQL> insert into Certified values(567354612,2);
1 row created.

SQL> insert into Certified values(567354612,3);
1 row created.
```

```
SQL> insert into Certified values(567354612,4);

1 row created.


SQL> insert into Certified values(567354612,5);

1 row created.


SQL> insert into Certified values(567354612,7);

1 row created.


SQL> insert into Certified values(567354612,9);

1 row created.


SQL> insert into Certified values(567354612,10);

1 row created.


SQL> insert into Certified values(567354612,11);

1 row created.
```

```
SQL> insert into Certified values(567354612,12);

1 row created.


SQL> insert into Certified values(567354612,15);

1 row created.


SQL> insert into Certified values(573284895,3);

1 row created.


SQL> insert into Certified values(573284895,4);

1 row created.


SQL> insert into Certified values(573284895,5);

1 row created.


SQL> insert into Certified values(574489456,6);

1 row created.


SQL> insert into Certified values(574489456,8);

1 row created.
```

```
SQL> insert into Certified values(574489457,7);

1 row created.


SQL> select * from Certified;
```

Output:

EID	AID
11564812	2
11564812	10
90873519	6
141582651	2
141582651	10
141582651	12
142519864	1
142519864	2
142519864	3
142519864	7
142519864	10
142519864	11

142519864	12
142519864	13
159542516	5
159542516	7
242518965	2
242518965	10
269734834	1
269734834	2
269734834	3
269734834	4
269734834	5
269734834	6
269734834	7
269734834	8
269734834	9
269734834	10
269734834	11
269734834	12
269734834	13
269734834	14
269734834	15

274878974	10
274878974	12
310454876	8
310454876	9
355548984	8
355548984	9
356187925	6
390487451	3
390487451	13
390487451	14
548977562	7
550156548	1
550156548	12
552455318	2
552455318	7
552455318	14
556784565	2
556784565	3
556784565	5
567354612	1
567354612	2

567354612	3
567354612	4
567354612	5
567354612	7
567354612	9
567354612	10
567354612	11
567354612	12
567354612	15
573284895	3
573284895	4
573284895	5
574489456	6
574489456	8
574489457	7

69 rows returned

Cycle-III
Queries

The following queries by using above tables.

1. For each pilot who is certified for more than three aircraft, find the eid's and the maximum cruising range of the aircraft that he (or She) certified for.

   SQL>select c.eid,max(a.crusing_range) from certified c,aircraft a where c.aid=a.aid group by c.eid having count(eid)>3;

   Output:

EID	MAX(A.CRUSING_RANGE)
269734834	8430
142519864	8430
567354612	8430

2. Find the names of pilots whose salary is less than the price of the cheapest routefrom Los Angeles to Honolulu.

   SQL>select ename from employees where salary<(select min(price) from flights where origin='Los Angeles' and destination='Honolulu');

Output:

ENAME
Chad Stewart
Milo Brooks

3. Find the name of the pilots certified from some Boeing aircraft.

SQL**>** select ename from employees where eid in (select eid from certified where aid in(select aid from aircraft where aname like 'Boeing%'));

Output:

ENAME
James Smith
Mary Johnson
John Williams
Lisa Walker
Larry West
Karen Scott
Michael Miller
Betty Adams
George Wright

Mark Young

10 rows returned

4. For all aircraft with cruising range over 1,000 miles, find the name of the aircraft and the average salary of all pilots certified for this aircraft.

   SQL**>** select a.aname,avg(e.salary) from employees e,aircraft a, certified c where a.aid=c.aid and c.eid= e.eid and a.crusing_range>1000 group by a.aname;

   Output:

ANAME	AVG(E.SALARY)
Boeing 737-800	191700.25
Tupolev 154	205001.25
Airbus A340-300	217597.666666666666666666666666666667
Embraer ERJ-145	182838.2
SAAB 340	118113.5

British Aerospace Jetstream 41	220251.33333333333333333333333333333333
Boeing 757-300	189508.57142857142857142857142857 1428571
Boeing 777-300	257973.33333333333333333333333333333333
Airbus A319	201283.66666666666666666666666666666667
Boeing 747-400	244776.75
Airbus A320	243198.33333333333333333333333333333333
Lockheed L1011	242685.75
Boeing 767-400ER	209557
Boeing 727	273215.5

14 rows returned

5. Find the aid's of all aircraft than can be used from Los Angels to Chicago.

```
SQL> select aid from aircraft where crusing_range>(select
min(distance) from flights where origin='Los Angeles' and
destination='Chicago.');
```

Output:

AID
1
2
3
6
8
9
10
11
12
13
14

6. Print the enames of pilots who can operate planes with cruising range greater than 3,000 miles, but are not certified by Boeing aircraft.

```
SQL> select ename from employees where eid in
  (select eid from employees where eid in
    (select eid from certified where aid in
    (select aid from aircraft where crusing_range>3000))
    Minus
    (select eid from employees where eid in
    (select eid from certified where aid in
    (select aid from aircraft where aname like 'Boeing%'))));
```

Output:

ENAME
Lawrence Sperry
Angela Martinez
Joseph Thompson
Eric Cooper
William Jones

5 rows returned

7. Find the total amount paid to employees as salaries.

SQL**>** select sum(salary) from employees;

Output:

SUM(SALARY)
4113877

8. Find the eid's of employees who are certified for exactly three aircrafts.

SQL**>** select eid from certified group by eid having count(aid)=3;

Output:

EID
141582651
390487451
552455318
556784565
573284895

5 rows returned

9. Find the eid's of employee who make second highest salary.

SQL**>** select eid,salary from employees e where 1=(select count(e1.salary) from employees e1 where salary>e.salary)

Output:

EID	SALARY
269734834	289950

10. Find the aid's of all than can be used on non-stop flights from Los Angeles to Boston.

    SQL**>**select a.aid from aircraft a,flights f where f.origin='Los Angeles'and f.destination='Boston' and a.crusing_range>f.distance;

Output:

AID
1
2
3
8
9
10
11
12

Cycle – IV
Employee Database

Aim: An enterprise wishes to maintain a database to automate its operations. Enterprise divided into two certain departments and each department consists of employees. The following two tables describes the automation schemas

1. Table Name: Dept

   Description: This table stores the information about the department.

   Creation:

```
create table Dept (

deptno number(2) not null,

dname char(14),

loc char(13),

constraint dept_primary_key primary key (deptno));

Table created.

SQL>desc Dept;
```

Output:

Table	Column	Data Type	Length	Precision	Scale	Primary Key	Nullable	Default	Comment
DEPT	DEPTNO	Number	-	2	0	1	-	-	-
	DNAME	Char	14	-	-	-	✓	-	-
	LOC	Char	13	-	-	-	✓	-	-

1 - 3

Insertion:

```
SQL> insert into Dept values (10,'accounting','new york');
1 row created.

SQL> insert into Dept values (20,'research','dallas');
1 row created.

SQL> insert into Dept values (30,'sales','chicago');
1 row created.
```

```
SQL> insert into Dept values (40,'operations','boston');

1 row created.


SQL> select * from Dept;
```

Output:

DEPTNO	DNAME	LOC
10	accounting	new york
20	research	dallas
30	sales	chicago
40	operations	boston

4 rows returned

2. Table Name: EMP

Description: This table stores the information about the EMPLOYEE.

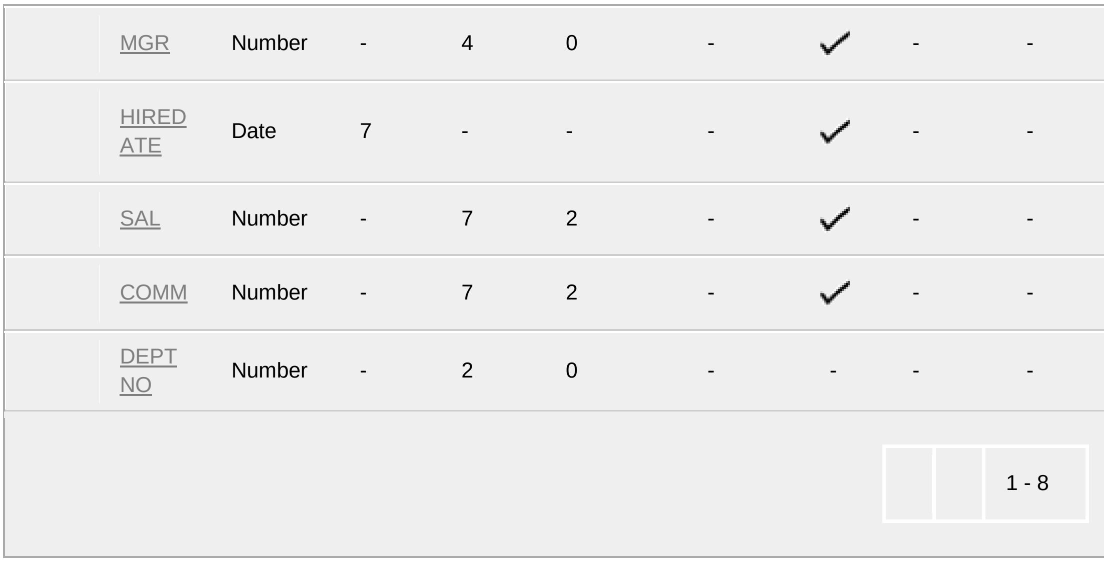
Creation:

```
SQL> create table Emp
   (
   empno number(4) not null,
   ename char(10),
    job char(9),
    mgr number(4) constraint emp_self_key references emp (empno),
    hiredate date,
    sal number(7,2),
    comm number(7,2),
   deptno number(2) not null,
   constraint emp_foreign_key foreign key (deptno) references dept
(deptno),
   constraint emp_primary_key primary key (empno)
   );
Table created.
```

SQL>desc Emp;

Output:

Table	Column	Data Type	Length	Precision	Scale	Primary Key	Nullable	Default	Comment
EMP	EMPNO	Number	-	4	0	1	-	-	-
	ENAME	Char	10	-	-	-	✓	-	-
	JOB	Char	9	-	-	-	✓	-	-



	MGR	Number	-	4	0	-	✓	-	-
	HIREDATE	Date	7	-	-	-	✓	-	-
	SAL	Number	-	7	2	-	✓	-	-
	COMM	Number	-	7	2	-	✓	-	-
	DEPTNO	Number	-	2	0	-	-	-	-

1 - 8

Insertion:

SQL> insert into Emp values (7839,'king','president',null,'17-nov-81',5000,null,10);

1 row created.

SQL> insert into Emp values (7698,'blake','manager',7839,'01-may-81',2850,null,30);

1 row created.

```
SQL> insert into Emp values (7782,'clark','manager',7839,'09-jun-81',2450,null,10);
1 row created.

SQL> insert into Emp values (7566,'jones','manager',7839,'02-apr-81',2975,null,20);
1 row created.

SQL> insert into Emp values (7654,'martin','salesman',7698,'28-sep-81',1250,1400,30);
1 row created.

SQL> insert into Emp values (7499,'allen','salesman',7698,'20-feb-81',1600,300,30);
1 row created.

SQL> insert into Emp values (7844,'turner','salesman',7698,'08-sep-81',1500,0,30);
1 row created.
```

```
SQL> insert into Emp values (7900,'james','clerk',7698,'03-dec-
81',950,null,30);
1 row created.

SQL> insert into Emp values (7521,'ward','salesman',7698,'22-feb-
81',1250,500,30);
1 row created.

SQL> insert into Emp values (7902,'ford','analyst',7566,'03-dec-
81',3000,null,20);
1 row created.

SQL> insert into Emp values (7369,'smith','clerk',7902,'17-dec-
80',800,null,20);
1 row created.

SQL> insert into Emp values (7788,'scott','analyst',7566,'09-dec-
82',3000,null,20);
1 row created.
```

```
SQL>  insert into Emp values (7876,'adams','clerk',7788,'12-jan-
83',1100,null,20);
1 row created.

SQL> insert into Emp values (7934,'miller','clerk',7782,'23-jan-
82',1300,null,10);
1 row created.



SQL> select * from emp;
```

Output:

EMPNO	ENAME	JOB	MGR	HIREDATE	SAL	COMM	DEPTNO
7839	king	president	-	17-NOV-81	5000	-	10
7698	blake	manager	7839	01-MAY-81	2850	-	30
7782	clark	manager	7839	09-JUN-81	2450	-	10
7566	jones	manager	7839	02-APR-81	2975	-	20

7654	martin	salesman	7698	28-SEP-81	1250	1400	30
7499	allen	salesman	7698	20-FEB-81	1600	300	30
7844	turner	salesman	7698	08-SEP-81	1500	0	30
7900	james	clerk	7698	03-DEC-81	950	-	30
7521	ward	salesman	7698	22-FEB-81	1250	500	30
7902	ford	analyst	7566	03-DEC-81	3000	-	20
7369	smith	clerk	7902	17-DEC-80	800	-	20
7788	scott	analyst	7566	09-DEC-82	3000	-	20
7876	adams	clerk	7788	12-JAN-83	1100	-	20
7934	miller	clerk	7782	23-JAN-82	1300	-	10

Cycle-IV

Queries

The following queries by using above tables.

1. Create a view, which contain employee names and their manager names working in sales department.

   SQL> create view v1 as (select e.ename,n.ename as managername from emp e,emp n,dept d where e.mgr=n.empno and e.deptno=d.deptno and d.dname='sales');

   Output:

   View created

   SQL> select * from v1;

   Output:

ENAME	MANAGERNAME
blake	king
martin	blake
allen	blake
turner	blake

james	blake
ward	blake

6 rows returned

2. Determine the names of employee, who earn more than their managers.

   SQL> select e. ename,e. sal, s.sal from emp e,emp s where e.mgr=s.empno and e.sal>s.sal;

   Output:

ENAME	SAL	SAL
ford	3000	2975
scott	3000	2975

2 rows returned

3. Determine the name of employees, who take the highest salary in their departments.

   SQL> select ename,deptno,sal from emp where sal in(select max(sal) from emp group by deptno);

Output:

ENAME	DEPTNO	SAL
king	10	5000
blake	30	2850
ford	20	3000
scott	20	3000

4 rows returned

4. Determine the employees, who located at the same place.

SQL> select emp.ename,emp.job,emp.deptno,dept.dname,dept.loc from emp

,dept where emp.deptno=dept.deptno order by dept.loc;

Output:

ENAM E	JOB	DEPTN O	DNAME	LOC
turner	salesma n	30	sales	chicag o
ward	salesma n	30	sales	chicag o
james	clerk	30	sales	chicag o
allen	salesma	30	sales	chicag

	n			o
martin	salesman	30	sales	chicago
blake	manager	30	sales	chicago
scott	analyst	20	research	dallas
smith	clerk	20	research	dallas
ford	analyst	20	research	dallas
jones	manager	20	research	dallas
adams	clerk	20	research	dallas
miller	clerk	10	accounting	new york
clark	manager	10	accounting	new york
king	president	10	accounting	new york

14 rows returned

5. Determine the employees whose total salary is like the minimum salary of any department.

   SQL> select * from emp where nvl(sal,comm) in(select min(sal) from emp group by deptno);

Output:

EMPNO	ENAME	JOB	MGR	HIREDATE	SAL	COMM	DEPTNO
7900	james	clerk	7698	03-DEC-81	950	-	30
7369	smith	clerk	7902	17-DEC-80	800	-	20
7934	miller	clerk	7782	23-JAN-82	1300	-	10

3 rows returned

6. Update the employee salary by 25%, whose experience is greater than 10 years.

SQL> update emp set sal=sal+(sal*0.25) where round(months_between(sysdate,hiredate)/12)>10;

Output:

14 rows updated

SQL> select * from emp;

Output:

EMPNO	ENAME	JOB	MGR	HIREDATE	SAL	COMM	DEPTNO
7839	king	president	-	17-NOV-81	6250	-	10
7698	blake	manager	7839	01-MAY-81	3562.5	-	30
7782	clark	manager	7839	09-JUN-81	3062.5	-	10
7566	jones	manager	7839	02-APR-81	3718.75	-	20
7654	martin	salesman	7698	28-SEP-81	1562.5	1400	30
7499	allen	salesman	7698	20-FEB-81	2000	300	30
7844	turner	salesman	7698	08-SEP-81	1875	0	30
7900	james	clerk	7698	03-DEC-81	1187.5	-	30
7521	ward	salesman	7698	22-FEB-81	1562.5	500	30
7902	ford	analyst	7566	03-DEC-81	3750	-	20
7369	smith	clerk	7902	17-DEC-80	1000	-	20
7788	scott	analyst	7566	09-DEC-82	3750	-	20
7876	adams	clerk	7788	12-JAN-83	1375	-	20
7934	miller	clerk	7782	23-JAN-82	1625	-	10

14 rows
returned

7. Delete the employees, who completed 34 years of service.

   SQL> delete from emp where
   round(months_between(sysdate,hiredate)/12)>34;

   Output:

   ```
   no data found
   ```

8. Determine the minimum salary of an employee and his details, who join on the same date.

   SQL> select * from emp where sal in(select min(sal) from emp group by hiredate having count(hiredate)>=2);

   Output:

EMPNO	ENAME	JOB	MGR	HIREDATE	SAL	COMM	DEPTNO
7900	james	clerk	7698	03-DEC-81	950	-	30

1 rows returned

9. Determine the count of employees, who are taking commission and not taking commission.

   SQL> select count(comm) nottakingcommission,count(empno)-count(comm) as
   takingcommission from emp;

Output:

NOTTAKINGCOMMISSION	TAKINGCOMMISSION
4	10

1 rows returned

10. Determine the department does not contain any employees.

    SQL> select deptno from dept minus select deptno from emp;

Output:

DEPTNO
40

11. Find out the details of top 5 earners of company.

    (Note: Employee Salaries should not be duplicate like 5k, 4k, 4k, 3k, 2k)

    SQL> select * from (select * from emp order by sal desc) where rownum <= 5;

Output:

EMPNO	ENAME	JOB	MGR	HIREDATE	SAL	COMM	DEPTNO
7839	king	president	-	17-NOV-81	5000	-	10
7902	ford	analyst	7566	03-DEC-81	3000	-	20
7788	scott	analyst	7566	09-DEC-82	3000	-	20
7566	jones	manager	7839	02-APR-81	2975	-	20
7698	blake	manager	7839	01-MAY-81	2850	-	30

5 rows returned

12. Display those managers name whose salary is more than an average salary of his employees.

SQL> select m.ename,m.sal,avg(e.sal) from emp e,emp m where e.mgr=m.empno group by m.ename, m.sal having m.sal>avg(e.sal);

Output:

ENAME	SAL	AVG(E.SAL)
ford	3000	800
king	5000	2758.33333333333333333333333333333333333
scott	3000	1100
blake	2850	1310
clark	2450	1300

5 rows returned

13. Display the names of the managers who is having maximum number of employees

working under him?

SQL> select * from emp where empno in (select mgr from emp group by mgr having

count(*)=(select max(count(*)) from emp group by mgr));

Output:

EMPNO	ENAME	JOB	MGR	HIREDATE	SAL	COMM	DEPTNO
7698	blake	manager	7839	01-MAY-81	2850	-	30

1 rows returned

14. In which year did most people join the company? Display the year and number of employees.

SQL> select to_char(hiredate,'yyyy') "YEAR",count(EMPNO) "NO. OF Employees" from emp group by to_char(hiredate,'yyyy') having count(empno)=(select max(count(empno)) from emp group by to_char(hiredate,'yyyy'));

Output:

YEAR	NO. OF Employees
1981	10

1 rows returned

15. Display ename, dname even if there no employees working in a particular department(use outer join).

SQL> select d.dname,e.ename from dept d

```
left join emp e on d.deptno = e.deptno where e.deptno is null;
```

Output:

DNAME	ENAME
operations	-

1 rows returned

PL/SQL INRODUCTION:

The PL/SQL programming language was developed by Oracle Corporation in the late 1980s as procedural extension language for SQL and the Oracle relational database. Following are notable facts about PL/SQL:

- PL/SQL is a completely portable, high-performance transaction-processing language.
- PL/SQL provides a built-in interpreted and OS independent programming environment.
- PL/SQL can also directly be called from the command-line SQL*Plus interface.
- Direct call can also be made from external programming language calls to database.
- PL/SQL's general syntax is based on that of ADA and Pascal programming language.

- Apart from Oracle, PL/SQL is available in TimesTen in-memory database and IBM DB2.

Text Editor

Running large programs from command prompt may land you in inadvertently losing some of the work. So a better option is to use command files. To use the command files:

- Type your code in a text editor, like Notepad, Notepad+, or EditPlus, etc.
- Save the file with the .sql extension in the home directory.
- Launch SQL*Plus command prompt from the directory where you created your PL/SQL file.
- Type @file_name at the SQL*Plus command prompt to execute your program.

If you are not using a file to execute PL/SQL scripts, then simply copy your PL/SQL code and then right click on the black window having SQL prompt and use **paste** option to paste complete code at the command prompt. Finally, just press enter to execute the code, if it is not already executed.

PL/SQL - Basic Syntax

PL/SQL is a block-structured language, meaning that PL/SQL programs are divided and written in logical blocks of code. Each block consists of three sub-parts:

S.N.	Sections & Description
1	**Declarations** This section starts with the keyword **DECLARE**. It is an optional section and defines all variables, cursors, subprograms, and other elements to be used in the program.
2	**Executable Commands** This section is enclosed between the keywords **BEGIN** and **END** and it is a mandatory section. It consists of the executable PL/SQL statements of the program. It should have at least one executable line of code, which may be just a NULL command to indicate that nothing should be executed.
3	**Exception Handling** This section starts with the keyword **EXCEPTION**. This

	section is again optional and contains exception(s) that handle errors in the program.

Every PL/SQL statement ends with a semicolon **(;)**. PL/SQL blocks can be nested within other PL/SQL blocks using **BEGIN** and **END**. Here is the basic structure of a PL/SQL block:

```
DECLARE
   <declarations section>
BEGIN
   <executable command(s)>
EXCEPTION
   <exception handling>
END;
```

The 'Hello World' Example:

```
DECLARE
   message  varchar2(20):= 'Hello, World!';
BEGIN
   dbms_output.put_line(message);
END;
/
```

The **end;** line signals the end of the PL/SQL block. To run the code from SQL command line, you may need to type **/** at the beginning of the first blank line after the last line of the code. When the above code is executed at SQL prompt, it produces following result:

```
Hello World
```

PL/SQL procedure successfully completed.

PL/SQL - Data Types

PL/SQL variables, constants and parameters must have a valid data type which specifies a storage format, constraints, and valid range of values. This tutorial will take you through **SCALAR** and **LOB** data types available in PL/SQL and other two data types will be covered in other chapters.

Category	Description
Scalar	Single values with no internal components, such as a NUMBER, DATE, or BOOLEAN.
Large Object (LOB)	Pointers to large objects that are stored separately from other data items, such as text, graphic images, video clips, and sound waveforms.
Composite	Data items that have internal components that can be accessed individually. For example, collections and records.
Reference	Pointers to other data items.

PL/SQL Scalar Data Types and Subtypes

PL/SQL Scalar Data Types and Subtypes come under the following categories:

Date Type	Description
Numeric	Numeric values on which arithmetic operations are performed.
Character	Alphanumeric values that represent single characters or strings of characters.
Boolean	Logical values on which logical operations are performed.
Datetime	Dates and times.

PL/SQL provides subtypes of data types. For example, the data type NUMBER has a subtype called INTEGER. You can use subtypes in your PL/SQL program to make the data types compatible with data types in other programs while embedding PL/SQL code in another program, such as a Java program.

PL/SQL Numeric Data Types and Subtypes

Following is the detail of PL/SQL pre-defined numeric data types and their sub-types:

Data Type	Description
PLS_INTEGER	Signed integer in range -2,147,483,648 through 2,147,483,647, represented in 32 bits

BINARY_INTEGER	Signed integer in range -2,147,483,648 through 2,147,483,647, represented in 32 bits
BINARY_FLOAT	Single-precision IEEE 754-format floating-point number
BINARY_DOUBLE	Double-precision IEEE 754-format floating-point number
NUMBER(prec, scale)	Fixed-point or floating-point number with absolute value in range 1E-130 to (but not including) 1.0E126. A NUMBER variable can also represent 0.
DEC(prec, scale)	ANSI specific fixed-point type with maximum precision of 38 decimal digits.
DECIMAL(prec, scale)	IBM specific fixed-point type with maximum precision of 38 decimal digits.
NUMERIC(pre, secale)	Floating type with maximum precision of 38 decimal digits.
DOUBLE PRECISION	ANSI specific floating-point type with maximum precision of 126 binary digits (approximately 38 decimal digits)
FLOAT	ANSI and IBM specific floating-point type with maximum precision of 126 binary digits (approximately 38 decimal digits)

INT	ANSI specific integer type with maximum precision of 38 decimal digits
INTEGER	ANSI and IBM specific integer type with maximum precision of 38 decimal digits
SMALLINT	ANSI and IBM specific integer type with maximum precision of 38 decimal digits
REAL	Floating-point type with maximum precision of 63 binary digits (approximately 18 decimal digits)

Following is a valid declaration:

```
DECLARE
   num1 INTEGER;
   num2 REAL;
   num3 DOUBLE PRECISION;
BEGIN
   null;
END;
/
```

When the above code is compiled and executed, it produces the following result:

```
PL/SQL procedure successfully completed
```

PL/SQL Character Data Types and Subtypes

Following is the detail of PL/SQL pre-defined character data types and their sub-types:

Data Type	Description
CHAR	Fixed-length character string with maximum size of 32,767 bytes
VARCHAR2	Variable-length character string with maximum size of 32,767 bytes
RAW	Variable-length binary or byte string with maximum size of 32,767 bytes, not interpreted by PL/SQL
NCHAR	Fixed-length national character string with maximum size of 32,767 bytes
NVARCHAR2	Variable-length national character string with maximum size of 32,767 bytes
LONG	Variable-length character string with maximum size of 32,760 bytes
LONG RAW	Variable-length binary or byte string with maximum size of 32,760 bytes, not interpreted by PL/SQL
ROWID	Physical row identifier, the address of a row in an ordinary table
UROWID	Universal row identifier (physical, logical, or foreign row identifier)

PL/SQL Boolean Data Types

The BOOLEAN data type stores logical values that are used in logical operations. The logical values are the Boolean values TRUE and FALSE and the value NULL.

However, SQL has no data type equivalent to BOOLEAN. Therefore, Boolean values cannot be used in:

- SQL statements
- Built-in SQL functions (such as TO_CHAR)
- PL/SQL functions invoked from SQL statements

PL/SQL Datetime and Interval Types

The DATE datatype to store fixed-length datetimes, which include the time of day in seconds since midnight. Valid dates range from January 1, 4712 BC to December 31, 9999 AD.

The default date format is set by the Oracle initialization parameter NLS_DATE_FORMAT. For example, the default might be 'DD-MON-YY', which includes a two-digit number for the day of the month, an abbreviation of the month name, and the last two digits of the year, for example, 01-OCT-12.

Each DATE includes the century, year, month, day, hour, minute, and second. The following table shows the valid values for each field:

Field Name	**Valid Datetime Values**	**Valid Interval Values**
YEAR	-4712 to 9999 (excluding year 0)	Any nonzero integer
MONTH	01 to 12	0 to 11
DAY	01 to 31 (limited by the values of MONTH and YEAR, according to the rules of the calendar for the locale)	Any nonzero integer
HOUR	00 to 23	0 to 23
MINUTE	00 to 59	0 to 59
SECOND	00 to 59.9(n), where 9(n) is the precision of time fractional seconds	0 to 59.9(n), where 9(n) is the precision of interval fractional seconds
TIMEZONE_HOUR	-12 to 14 (range accommodates daylight savings time changes)	Not applicable
TIMEZONE_MINUTE	00 to 59	Not applicable
TIMEZONE_REGION	Found in the dynamic	Not applicable

	performance view V$TIMEZONE_NAMES	
TIMEZONE_ABBR	Found in the dynamic performance view V$TIMEZONE_NAMES	Not applicable

PL/SQL Large Object (LOB) Data Types

Large object (LOB) data types refer large to data items such as text, graphic images, video clips, and sound waveforms. LOB data types allow efficient, random, piecewise access to this data. Following are the predefined PL/SQL LOB data types:

Data Type	Description	Size
BFILE	Used to store large binary objects in operating system files outside the database.	System-dependent. Cannot exceed 4 gigabytes (GB).
BLOB	Used to store large binary objects in the database.	8 to 128 terabytes (TB)
CLOB	Used to store large blocks of character data in the database.	8 to 128 TB

NCLOB	Used to store large blocks of NCHAR data in the database.	8 to 128 TB

PL/SQL User-Defined Subtypes

A subtype is a subset of another data type, which is called its base type. A subtype has the same valid operations as its base type, but only a subset of its valid values.

PL/SQL predefines several subtypes in package STANDARD. For example, PL/SQL predefines the subtypes CHARACTER and INTEGER as follows:

```
SUBTYPE CHARACTER IS CHAR;
SUBTYPE INTEGER IS NUMBER(38,0);
```

You can define and use your own subtypes. The following program illustrates defining and using a user-defined subtype:

```
DECLARE
   SUBTYPE name IS char(20);
   SUBTYPE message IS varchar2(100);
   salutation name;
   greetings message;
BEGIN
   salutation := 'Reader ';
   greetings := 'Welcome to the World of PL/SQL';
   dbms_output.put_line('Hello ' || salutation || greetings);
END;
```

```
/
```

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

```
Hello Reader Welcome to the World of PL/SQL

PL/SQL procedure successfully completed.
```

NULLs in PL/SQL

PL/SQL NULL values represent missing or unknown data and they are not an integer, a character, or any other specific data type. Note that NULL is not the same as an empty data string or the null character value '\0'. A null can be assigned but it cannot be equated with anything, including itself.

PL/SQL - Variables

The name of a PL/SQL variable consists of a letter optionally followed by more letters, numerals, dollar signs, underscores, and number signs and should not exceed 30 characters. By default, variable names are not case-sensitive. You cannot use a reserved PL/SQL keyword as a variable name.

Variable Declaration in PL/SQL

PL/SQL variables must be declared in the declaration section or in a package as a global variable. When you declare a variable, PL/SQL allocates memory for the variable's value and the storage location is identified by the variable name.

The syntax for declaring a variable is:

```
variable_name [CONSTANT] datatype [NOT NULL] [:= | DEFAULT
initial_value]
```

Where, *variable_name* is a valid identifier in PL/SQL, *datatype* must be a valid PL/SQL data type or any user defined data type which we already have discussed in last chapter. Some valid variable declarations along with their definition are shown below:

```
sales number(10, 2);
pi CONSTANT double precision := 3.1415;
name varchar2(25);
address varchar2(100);
```

When you provide a size, scale or precision limit with the data type, it is called a**constrained declaration**. Constrained declarations require less memory than unconstrained declarations. For example:

```
sales number(10, 2);
name varchar2(25);
address varchar2(100);
```

Initializing Variables in PL/SQL

Whenever you declare a variable, PL/SQL assigns it a default value of NULL. If you want to initialize a variable with a value other than the NULL value, you can do so during the declaration, using either of the following:

- The **DEFAULT** keyword
- The **assignment** operator

For example:

```
counter binary_integer := 0;
greetings varchar2(20) DEFAULT 'Have a Good Day';
```

You can also specify that a variable should not have a **NULL** value using the **NOT NULL**constraint. If you use the NOT NULL constraint, you must explicitly assign an initial value for that variable.

It is a good programming practice to initialize variables properly otherwise, sometime program would produce unexpected result. Try the following example which makes use of various types of variables:

```
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;
/
```

When the above code is executed, it produces the following result:

```
Value of c: 30
Value of f: 23.333333333333333333

PL/SQL procedure successfully completed.
```

Variable Scope in PL/SQL

PL/SQL allows the nesting of Blocks, i.e., each program block may contain another inner block. If a variable is declared within an inner block, it is not accessible to the outer block. However, if a variable is declared and accessible to an outer Block, it is also accessible to all nested inner Blocks. There are two types of variable scope:

- **Local variables** - variables declared in an inner block and not accessible to outer blocks.
- **Global variables** - variables declared in the outermost block or a package.

Following example shows the usage of **Local** and **Global** variables in its simple form:

```
DECLARE
   -- Global variables
   num1 number := 95;
   num2 number := 85;
BEGIN
   dbms_output.put_line('Outer Variable num1: ' || num1);
   dbms_output.put_line('Outer Variable num2: ' || num2);
   DECLARE
```

```
      -- Local variables
      num1 number := 195;
      num2 number := 185;
   BEGIN
      dbms_output.put_line('Inner Variable num1: ' || num1);
      dbms_output.put_line('Inner Variable num2: ' || num2);
   END;
END;
/
```

When the above code is executed, it produces the following result:

```
Outer Variable num1: 95
Outer Variable num2: 85
Inner Variable num1: 195
Inner Variable num2: 185

PL/SQL procedure successfully completed.
```

PL/SQL - Constants and Literals

A constant holds a value that once declared, does not change in the program. A constant declaration specifies its name, data type, and value, and allocates storage for it. The declaration can also impose the NOT NULL constraint.

Declaring a Constant

A constant is declared using the CONSTANT keyword. It requires an initial value and does not allow that value to be changed. For example:

```
PI CONSTANT NUMBER := 3.141592654;
DECLARE
   -- constant declaration
   pi constant number := 3.141592654;
   -- other declarations
   radius number(5,2);
   dia number(5,2);
   circumference number(7, 2);
   area number (10, 2);
BEGIN
   -- processing
   radius := 9.5;
   dia := radius * 2;
   circumference := 2.0 * pi * radius;
   area := pi * radius * radius;
   -- output
   dbms_output.put_line('Radius: ' || radius);
   dbms_output.put_line('Diameter: ' || dia);
   dbms_output.put_line('Circumference: ' || circumference);
   dbms_output.put_line('Area: ' || area);
END;
/
```

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

```
Radius: 9.5
Diameter: 19
Circumference: 59.69
Area: 283.53

Pl/SQL procedure successfully completed.
```

The PL/SQL Literals

A literal is an explicit numeric, character, string, or Boolean value not represented by an identifier. For example, TRUE, 786, NULL, 'tutorialspoint' are all literals of type Boolean, number, or string. PL/SQL, literals are case-sensitive. PL/SQL supports the following kinds of literals:

- Numeric Literals
- Character Literals
- String Literals
- BOOLEAN Literals
- Date and Time Literals

The following table provides examples from all these categories of literal values.

Literal Type	Example:

Numeric Literals	050 78 -14 0 +32767 6.6667 0.0 -12.0 3.14159 +7800.00 6E5 1.0E-8 3.14159e0 -1E38 -9.5e-3
Character Literals	'A' '%' '9' ' ' 'z' '('
String Literals	'Hello, world!' 'Tutorials Point' '19-NOV-12'
BOOLEAN Literals	TRUE, FALSE, and NULL.

Date and Time Literals	DATE '1978-12-25';
	TIMESTAMP '2012-10-29 12:01:01';

To embed single quotes within a string literal, place two single quotes next to each other as shown below:

```
DECLARE
  message  varchar2(20):= ''That''s tutorialspoint.com!'';
BEGIN
  dbms_output.put_line(message);
END;
/
```

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

```
That's tutorialspoint.com!
```

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

PL/SQL - Operators

An operator is a symbol that tells the compiler to perform specific mathematical or logical manipulation. PL/SQL language is rich in built-in operators and provides the following type of operators:

- Arithmetic operators
- Relational operators
- Comparison operators
- Logical operators
- String operators

This tutorial will explain the arithmetic, relational, comparison and logical operators one by one. The String operators will be discussed under the chapter: **PL/SQL - Strings**.

Arithmetic Operators

Following table shows all the arithmetic operators supported by PL/SQL. Assume variable A holds 10 and variable B holds 5 then:

Operator	Description	Example
+	Adds two operands	A + B will give 15

-	Subtracts second operand from the first	A - B will give 5
*	Multiplies both operands	A * B will give 50
/	Divides numerator by de-numerator	A / B will give 2
**	Exponentiation operator, raises one operand to the power of other	A ** B will give 100000

Relational Operators

Relational operators compare two expressions or values and return a Boolean result. Following table shows all the relational operators supported by PL/SQL. Assume variable A holds 10 and variable B holds 20, then:

Operator	**Description**	**Example**
=	Checks if the values of two operands are equal or not, if yes then condition becomes true.	(A = B) is not true.
!= <> ~=	Checks if the values of two operands are equal or not, if values are not equal then condition becomes true.	(A != B) is true.

>	Checks if the value of left operand is greater than the value of right operand, if yes then condition becomes true.	(A > B) is not true.
<	Checks if the value of left operand is less than the value of right operand, if yes then condition becomes true.	(A < B) is 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.	(A >= B) is not 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.	(A <= B) is true.

Comparison Operators

Comparison operators are used for comparing one expression to another. The result is always either TRUE, FALSE OR NULL.

Operator	Description	Example
LIKE	The LIKE operator compares a character, string, or CLOB value to a pattern and returns TRUE if the value matches the pattern and FALSE if it does not.	If 'Zara Ali' like 'Z% A_i' returns a Boolean true, whereas, 'Nuha Ali' like 'Z% A_i' returns a Boolean false.
BETWEEN	The BETWEEN operator tests	If x = 10 then, x between

	whether a value lies in a specified range. x BETWEEN a AND b means that x >= a and x <= b.	5 and 20 returns true, x between 5 and 10 returns true, but x between 11 and 20 returns false.
IN	The IN operator tests set membership. x IN (set) means that x is equal to any member of set.	If x = 'm' then, x in ('a', 'b', 'c') returns boolean false but x in ('m', 'n', 'o') returns Boolean true.
IS NULL	The IS NULL operator returns the BOOLEAN value TRUE if its operand is NULL or FALSE if it is not NULL. Comparisons involving NULL values always yield NULL.	If x = 'm', then 'x is null' returns Boolean false.

Logical Operators

Following table shows the Logical operators supported by PL/SQL. All these operators work on Boolean operands and produces Boolean results. Assume variable A holds true and variable B holds false, then:

Operator	Description	Example
and	Called logical AND operator. If both the operands are true then condition becomes true.	(A and B) is false.

or	Called logical OR Operator. If any of the two operands is true then condition becomes true.	(A or B) is true.
not	Called logical NOT Operator. Used to reverse the logical state of its operand. If a condition is true then Logical NOT operator will make it false.	not (A and B) is true.

PL/SQL Operator Precedence

Operator precedence determines the grouping of terms in an expression. This affects how an expression is evaluated. Certain operators have higher precedence than others; for example, the multiplication operator has higher precedence than the addition operator:

For example x = 7 + 3 * 2; here, x is assigned 13, not 20 because operator * has higher precedence than +, so it first gets multiplied with 3*2 and then adds into 7.

Here operators with the highest precedence appear at the top of the table, those with the lowest appear at the bottom. Within an expression, higher precedence operators will be evaluated first.

Operator	**Operation**
**	exponentiation

+, -	identity, negation
*, /	multiplication, division
+, -, ||	addition, subtraction, concatenation
=, <, >, <=, >=, <>, !=, ~=, ^=, IS NULL, LIKE, BETWEEN, IN	comparison
NOT	logical negation
AND	conjunction
OR	inclusion

PL/SQL - Conditions

Decision-making structures require that the programmer specify one or more conditions to be evaluated or tested by the program, along with a statement or statements to be executed if the condition is determined to be true, and optionally, other statements to be executed if the condition is determined to be false.

IF-THEN Statement

It is the simplest form of **IF** control statement, frequently used in decision making and changing the control flow of the program execution.

The **IF statement** associates a condition with a sequence of statements enclosed by the keywords **THEN** and **END IF**. If the condition is **TRUE**, the statements get executed, and if the condition is **FALSE** or **NULL**, then the **IF** statement does nothing.

Syntax:

Syntax for IF-THEN statement is:

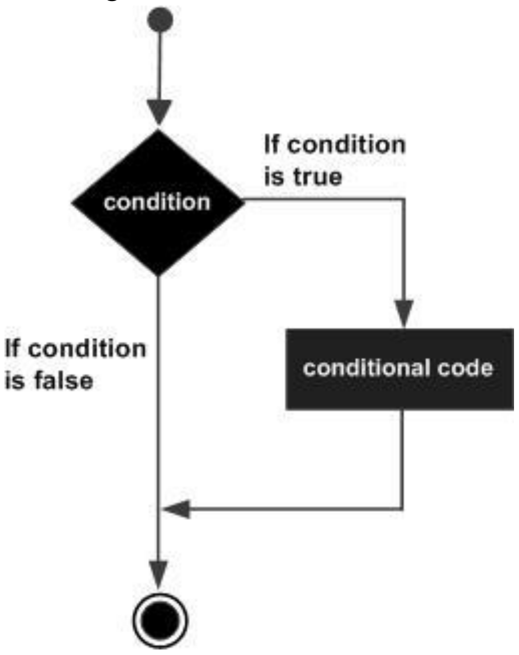
```
IF condition THEN
   S;
END IF;
```

Where *condition* is a Boolean or relational condition and *S* is a simple or compound statement. Example of an IF-THEN statement is:

```
IF (a <= 20) THEN
   c:= c+1;
END IF;
```

If the Boolean expression *condition* evaluates to true, then the block of code inside the if statement will be executed. If Boolean expression evaluates to false, then the first set of code after the end of the if statement (after the closing end if) will be executed.

Flow Diagram:



IF-THEN-ELSE Statement

A sequence of **IF-THEN** statements can be followed by an optional sequence of **ELSE**statements, which execute when the condition is **FALSE**.

Syntax:

Syntax for the IF-THEN-ELSE statement is:

```
IF condition THEN
   S1;
ELSE
```

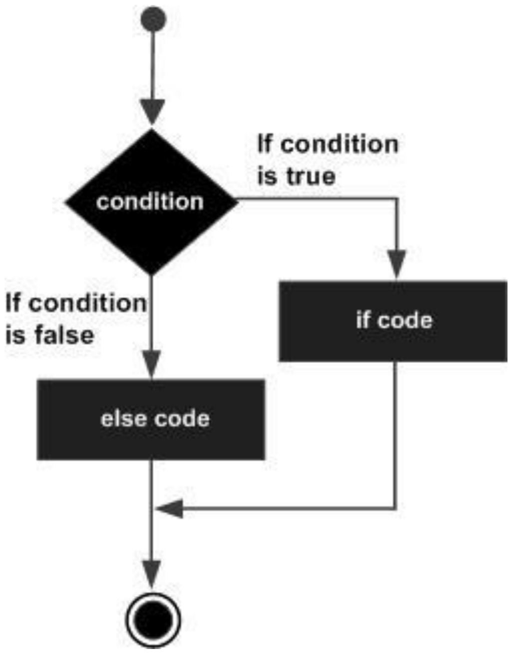
```
   S2;
END IF;
```

Where, *S1* and *S2* are different sequence of statements. In the IF-THEN-ELSE statements, when the test *condition* is TRUE, the statement *S1* is executed and *S2* is skipped; when the test *condition* is FALSE, then *S1* is bypassed and statement *S2* is executed. For example,

```
IF color = red THEN
  dbms_output.put_line('You have chosen a red car')
ELSE
  dbms_output.put_line('Please choose a color for your car');
END IF;
```

If the Boolean expression *condition* evaluates to true, then the if-then block of code will be executed, otherwise the else block of code will be executed.

Flow Diagram:



IF-THEN-ELSIF Statement

The **IF-THEN-ELSIF** statement allows you to choose between several alternatives. An **IF-THEN** statement can be followed by an optional **ELSIF...ELSE** statement. The **ELSIF** clause lets you add additional conditions.

When using **IF-THEN-ELSIF** statements there are few points to keep in mind.

- It's ELSIF, not ELSEIF

- An IF-THEN statement can have zero or one ELSE's and it must come after any ELSIF's.
- An IF-THEN statement can have zero to many ELSIF's and they must come before the ELSE.
- Once an ELSIF succeeds, none of the remaining ELSIF's or ELSE's will be tested.

Syntax:

The syntax of an IF-THEN-ELSIF Statement in PL/SQL programming language is:

```
IF(boolean_expression 1)THEN
   S1; -- Executes when the boolean expression 1 is true
ELSIF( boolean_expression 2) THEN
   S2;  -- Executes when the boolean expression 2 is true
ELSIF( boolean_expression 3) THEN
   S3; -- Executes when the boolean expression 3 is true
ELSE
   S4; -- executes when the none of the above condition is true
END IF;
```

CASE Statement

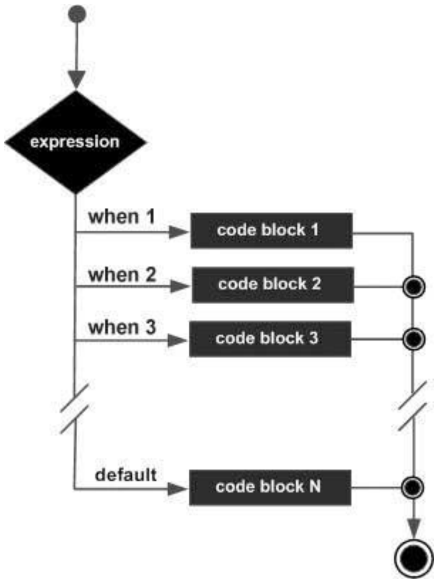
Like the **IF** statement, the **CASE statement** selects one sequence of statements to execute. However, to select the sequence, the **CASE** statement uses a selector rather than multiple Boolean expressions. A selector is an expression, whose value is used to select one of several alternatives.

Syntax:

The syntax for case statement in PL/SQL is:

```
CASE selector
   WHEN 'value1' THEN S1;
   WHEN 'value2' THEN S2;
   WHEN 'value3' THEN S3;
   ...
   ELSE Sn;  -- default case
END CASE;
```

Flow Diagram:



Searched CASE Statement

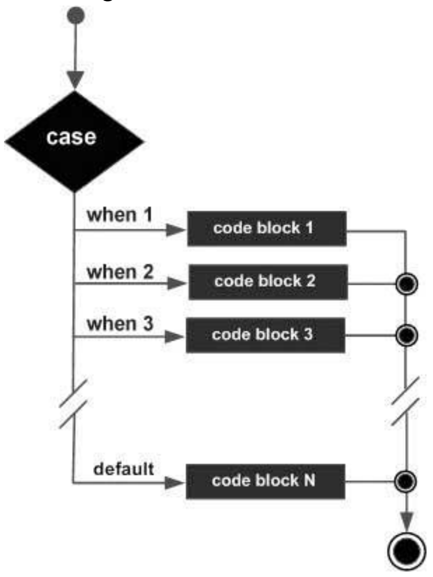
The searched **CASE** statement has no selector and its **WHEN** clauses contain search conditions that give Boolean values.

Syntax:

The syntax for searched case statement in PL/SQL is:

```
CASE
   WHEN selector = 'value1' THEN S1;
   WHEN selector = 'value2' THEN S2;
   WHEN selector = 'value3' THEN S3;
   ...
   ELSE Sn;  -- default case
END CASE;
```

Flow Diagram:



Nested IF-THEN-ELSE Statements

It is always legal in PL/SQL programming to nest **IF-ELSE** statements, which means you can use one **IF** or **ELSE IF** statement inside another **IF** or **ELSE IF** statement(s).

Syntax:

```
IF( boolean_expression 1)THEN
    -- executes when the boolean expression 1 is true
    IF(boolean_expression 2) THEN
      -- executes when the boolean expression 2 is true
      sequence-of-statements;
   END IF;
```

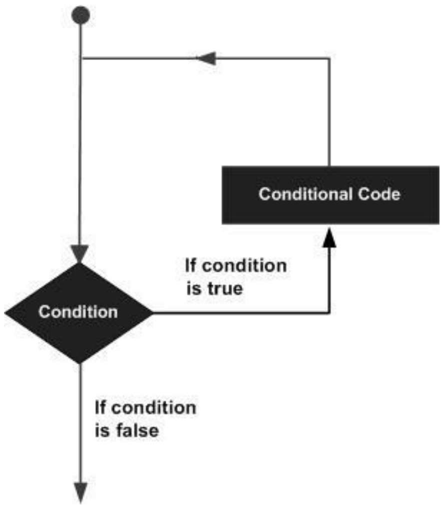
```
ELSE
   -- executes when the boolean expression 1 is not true
  else-statements;
END IF;
```

PL/SQL - Loops

There may be a situation when you need to execute a block of code several number of times. In general, statements are executed sequentially: The first statement in a function is executed first, followed by the second, and so on.

Programming languages provide various control structures that allow for more complicated execution paths.

A loop statement allows us to execute a statement or group of statements multiple times and following is the general form of a loop statement in most of the programming languages:



Basic Loop Statement

Basic loop structure encloses sequence of statements in between the **LOOP** and **END LOOP**statements. With each iteration, the sequence of statements is executed and then control resumes at the top of the loop.

Syntax:

The syntax of a basic loop in PL/SQL programming language is:

```
LOOP
  Sequence of statements;
END LOOP;
```

Here, sequence of statement(s) may be a single statement or a block of statements. An EXIT statement or an EXIT WHEN statement is required to break the loop.

WHILE LOOP Statement

A **WHILE LOOP** statement in PL/SQL programming language repeatedly executes a target statement as long as a given condition is true.

Syntax:

```
WHILE condition LOOP
   sequence_of_statements
END LOOP;
```

FOR LOOP Statement

A **FOR LOOP** is a repetition control structure that allows you to efficiently write a loop that needs to execute a specific number of times.

Syntax:

```
FOR counter IN initial_value .. final_value LOOP
   sequence_of_statements;
END LOOP;
```

Following are some special characteristics of PL/SQL for loop:

- The *initial_value* and *final_value* of the loop variable or *counter* can be literals, variables, or expressions but must evaluate to numbers. Otherwise, PL/SQL raises the predefined exception VALUE_ERROR.
- The *initial_value* need not to be 1; however, the **loop counter increment (or decrement) must be 1**.
- PL/SQL allows determine the loop range dynamically at run time.

Nested Loops

PL/SQL allows using one loop inside another loop. Following section shows few examples to illustrate the concept.

The syntax for a nested basic LOOP statement in PL/SQL is as follows:

```
LOOP
   Sequence of statements1
   LOOP
      Sequence of statements2
   END LOOP;
END LOOP;
```

The syntax for a nested FOR LOOP statement in PL/SQL is as follows:

```
FOR counter1 IN initial_value1 .. final_value1 LOOP
```

```
  sequence_of_statements1
  FOR counter2 IN initial_value2 .. final_value2 LOOP
    sequence_of_statements2
  END LOOP;
END LOOP;
```

The syntax for a nested WHILE LOOP statement in Pascal is as follows:

```
WHILE condition1 LOOP
  sequence_of_statements1
  WHILE condition2 LOOP
    sequence_of_statements2
  END LOOP;
END LOOP;
```

EXIT Statement

The **EXIT** statement in PL/SQL programming language has following two usages:

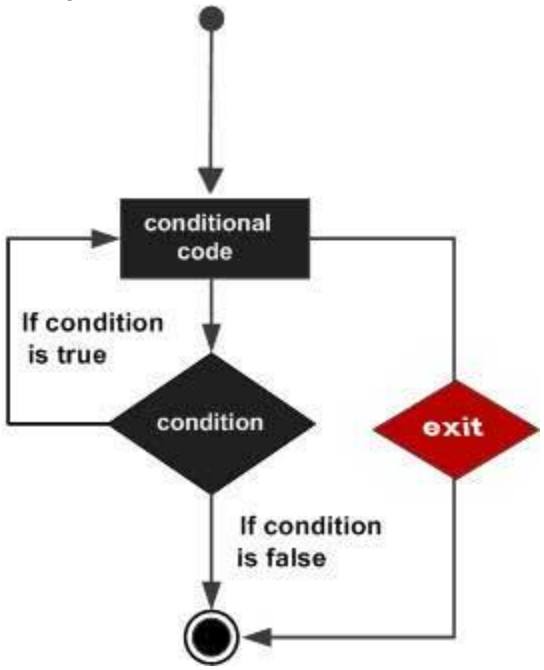
- When the EXIT statement is encountered inside a loop, the loop is immediately terminated and program control resumes at the next statement following the loop.
- If you are using nested loops (i.e. one loop inside another loop), the EXIT statement will stop the execution of the innermost loop and start executing the next line of code after the block.

Syntax:

The syntax for a EXIT statement in PL/SQL is as follows:

```
EXIT;
```

Flow Diagram:



CONTINUE Statement

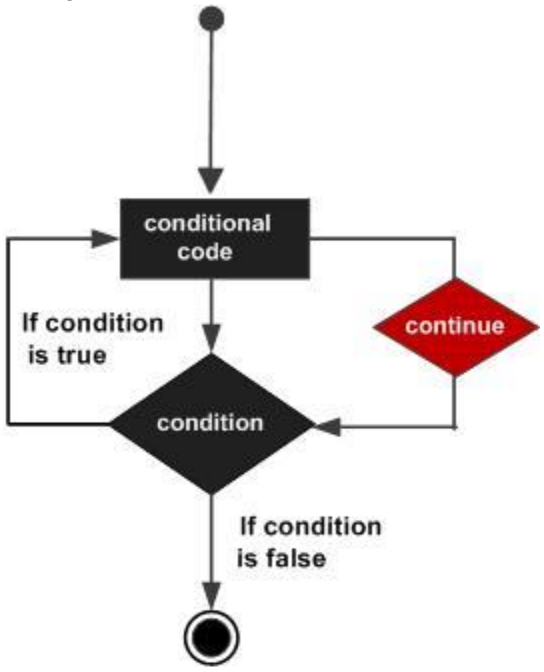
The **CONTINUE** statement causes the loop to skip the remainder of its body and immediately retest its condition prior to reiterating. In other words, it forces the next iteration of the loop to take place, skipping any code in between.

Syntax:

The syntax for a CONTINUE statement is as follows:

```
CONTINUE;
```

Flow Diagram:



GOTO Statement

A **GOTO** statement in PL/SQL programming language provides an unconditional jump from the GOTO to a labeled statement in the same subprogram.

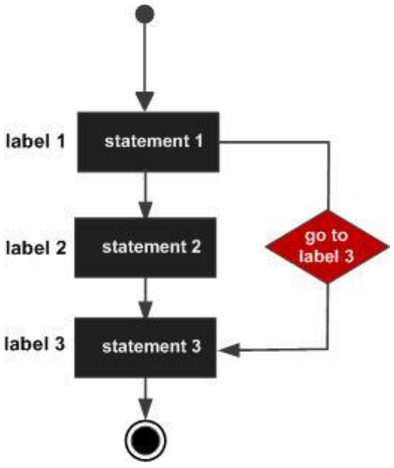
NOTE: Use of GOTO statement is highly discouraged in any programming language because it makes difficult to trace the control flow of a program, making the program hard to understand and hard to modify. Any program that uses a GOTO can be rewritten so that it doesn't need the GOTO.

Syntax:

The syntax for a GOTO statement in PL/SQL is as follows:

```
GOTO label;
..
..
<< label >>
statement;
```

Flow Diagram:



PL/SQL - Strings

The string in PL/SQL is actually a sequence of characters with an optional size specification. The characters could be numeric, letters, blank, special characters or a combination of all. PL/SQL offers three kinds of strings:

- **Fixed-length strings**: In such strings, programmers specify the length while declaring the string. The string is right-padded with spaces to the length so specified.
- **Variable-length strings**: In such strings, a maximum length up to 32,767, for the string is specified and no padding takes place.
- **Character large objects (CLOBs)**: These are variable-length strings that can be up to 128 terabytes.

PL/SQL strings could be either variables or literals. A string literal is enclosed within quotation marks. For example,

```
'This is a string literal.' Or 'hello world'
```

To include a single quote inside a string literal, you need to type two single quotes next to one another, like:

```
'this isn''t what it looks like'
```

Declaring String Variables

Oracle database provides numerous string datatypes , like, CHAR, NCHAR, VARCHAR2, NVARCHAR2, CLOB, and NCLOB. The datatypes prefixed with an 'N' are 'national character set' datatypes, that store Unicode character data.

If you need to declare a variable-length string, you must provide the maximum length of that string. For example, the VARCHAR2 data type. The following example illustrates declaring and using some string variables:

```
DECLARE
   name varchar2(20);
   company varchar2(30);
   introduction clob;
   choice char(1);
BEGIN
   name := 'John Smith';
   company := 'Infotech';
   introduction := ' Hello! I''m John Smith from Infotech.';
   choice := 'y';
   IF choice = 'y' THEN
      dbms_output.put_line(name);
      dbms_output.put_line(company);
      dbms_output.put_line(introduction);
   END IF;
END;
/
```

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

```
John Smith
Infotech Corporation
Hello! I'm John Smith from Infotech.

PL/SQL procedure successfully completed
```

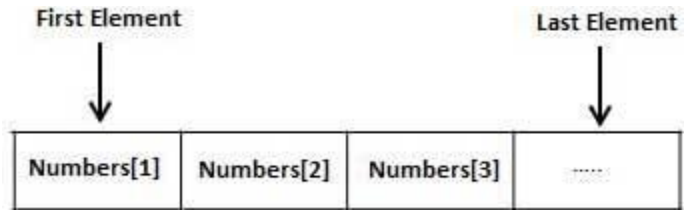
To declare a fixed-length string, use the CHAR datatype. Here you do not have to specify a maximum length for a fixed-length variable. If you leave off the length constraint, Oracle Database automatically uses a maximum length required. So following two declarations below are identical:

```
 red_flag CHAR(1) := 'Y';
 red_flag CHAR    := 'Y';
```

PL/SQL - Arrays

PL/SQL programming language provides a data structure called the VARRAY, which can store a fixed-size sequential collection of elements of the same type. A varray is used to store an ordered collection of data, but it is often more useful to think of an array as a collection of variables of the same type.

All varrays consist of contiguous memory locations. The lowest address corresponds to the first element and the highest address to the last element.



An array is a part of collection type data and it stands for variable-size arrays. We will study other collection types in a later chapter 'PL/SQL Collections'.

Each element in a varray has an index associated with it. It also has a maximum size that can be changed dynamically.

Creating a Varray Type

A varray type is created with the CREATE TYPE statement. You must specify the maximum size and the type of elements stored in the varray.

The basic syntax for creating a VRRAY type at the schema level is:

```
CREATE OR REPLACE TYPE varray_type_name IS VARRAY(n) of
<element_type>
```

Where,

- *varray_type_name* is a valid attribute name,

- *n* is the number of elements (maximum) in the varray,
- *element_type* is the data type of the elements of the array.

Maximum size of a varray can be changed using the ALTER TYPE statement.

For example,

```
CREATE Or REPLACE TYPE namearray AS VARRAY(3) OF
VARCHAR2(10);
/

Type created.
```

The basic syntax for creating a VRRAY type within a PL/SQL block is:

```
TYPE varray_type_name IS VARRAY(n) of <element_type>
```

For example:

```
TYPE namearray IS VARRAY(5) OF VARCHAR2(10);
Type grades IS VARRAY(5) OF INTEGER;
```

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

A schema level subprogram is a **standalone subprogram**. It is created with the CREATE PROCEDURE or 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** and we will cover**PL/SQL function** in next chapter.

Parts of a PL/SQL Subprogram

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

S.N.	Parts & Description
1	**Declarative Part** It is an optional part. However, the declarative part for a subprogram does n DECLARE keyword. It contains declarations of types, cursors, const exceptions, and nested subprograms. These items are local to the subprogra exist when the subprogram completes execution.
2	**Executable Part** This is a mandatory part and contains statements that perform the designate
3	**Exception-handling** This is again an optional part. It contains the code that handles run-time erro

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 modifying an existing procedure.
- The optional parameter list contains name, mode and types of the parameters. IN represents that value will be passed from outside and OUT represents that this parameter 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.

PL/SQL - Functions

A PL/SQL function is same as a procedure except that it returns a value.

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

PL/SQL - Cursors

Oracle creates a memory area, known as context area, for processing an SQL statement, which contains all information needed for processing the statement, for example, number of rows processed, etc.

A cursor is a pointer to this context area. PL/SQL controls the context area through a cursor. A cursor holds the rows (one or more) returned by a SQL statement. The set of rows the cursor holds is referred to as the **active set**.

You can name a cursor so that it could be referred to in a program to fetch and process the rows returned by the SQL statement, one at a time. There are two types of cursors:

- Implicit cursors
- Explicit cursors

Implicit Cursors

Implicit cursors are automatically created by Oracle whenever an SQL statement is executed, when there is no explicit cursor for the statement. Programmers cannot control the implicit cursors and the information in it.

Whenever a DML statement (INSERT, UPDATE and DELETE) is issued, an implicit cursor is associated with this statement. For

INSERT operations, the cursor holds the data that needs to be inserted. For UPDATE and DELETE operations, the cursor identifies the rows that would be affected.

In PL/SQL, you can refer to the most recent implicit cursor as the **SQL cursor**, which always has the attributes like %FOUND, %ISOPEN, %NOTFOUND, and %ROWCOUNT. The SQL cursor has additional attributes, %BULK_ROWCOUNT and %BULK_EXCEPTIONS, designed for use with the FORALL statement.

Explicit Cursors

Explicit cursors are programmer defined cursors for gaining more control over the **context area**. An explicit cursor should be defined in the declaration section of the PL/SQL Block. It is created on a SELECT Statement which returns more than one row.

The syntax for creating an explicit cursor is :

```
CURSOR cursor_name IS select_statement;
```

Working with an explicit cursor involves four steps:

- Declaring the cursor for initializing in the memory
- Opening the cursor for allocating memory
- Fetching the cursor for retrieving data
- Closing the cursor to release allocated memory

Declaring the Cursor

Declaring the cursor defines the cursor with a name and the associated SELECT statement. For example:

```
CURSOR c_customers IS
   SELECT id, name, address FROM customers;
```

Opening the Cursor

Opening the cursor allocates memory for the cursor and makes it ready for fetching the rows returned by the SQL statement into it. For example, we will open above-defined cursor as follows:

```
OPEN c_customers;
```

Fetching the Cursor

Fetching the cursor involves accessing one row at a time. For example we will fetch rows from the above-opened cursor as follows:

```
FETCH c_customers INTO c_id, c_name, c_addr;
```

Closing the Cursor

Closing the cursor means releasing the allocated memory. For example, we will close above-opened cursor as follows:

```
CLOSE c_customers;
```

Example:

Following is a complete example to illustrate the concepts of explicit cursors:

```
DECLARE
   c_id customers.id%type;
   c_name customers.name%type;
   c_addr customers.address%type;
   CURSOR c_customers is
      SELECT id, name, address FROM customers;
BEGIN
   OPEN c_customers;
   LOOP
      FETCH c_customers into c_id, c_name, c_addr;
      dbms_output.put_line(c_id || ' ' || c_name || ' ' || c_addr);
      EXIT WHEN c_customers%notfound;
   END LOOP;
   CLOSE c_customers;
END;
/
```

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

```
1 Ramesh Ahmedabad
2 Khilan Delhi
3 kaushik Kota
4 Chaitali Mumbai
5 Hardik Bhopal
6 Komal MP

PL/SQL procedure successfully completed.
```

PL/SQL - Records

A PL/SQL **record** is a data structure that can hold data items of different kinds. Records consist of different fields, similar to a row of a database table.

PL/SQL can handle the following types of records:

- Table-based
- Cursor-based records
- User-defined records

Table-Based Records

The %ROWTYPE attribute enables a programmer to create **table-based** and **cursor-based**records.

The following example would illustrate the concept of **table-based** records. We will be using the CUSTOMERS table we had created and used in the previous chapters:

```
DECLARE
   customer_rec customers%rowtype;
BEGIN
   SELECT * into customer_rec
   FROM customers
   WHERE id = 5;

   dbms_output.put_line('Customer ID: ' || customer_rec.id);
   dbms_output.put_line('Customer Name: ' || customer_rec.name);
   dbms_output.put_line('Customer Address: ' ||
customer_rec.address);
```

```
   dbms_output.put_line('Customer Salary: ' || customer_rec.salary);
END;
/
```

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

```
Customer ID: 5
Customer Name: Hardik
Customer Address: Bhopal
Customer Salary: 9000

PL/SQL procedure successfully completed.
```

Cursor-Based Records

The following example would illustrate the concept of **cursor-based** records. We will be using the CUSTOMERS table we had created and used in the previous chapters:

```
DECLARE
   CURSOR customer_cur is
      SELECT id, name, address
      FROM customers;
   customer_rec customer_cur%rowtype;
BEGIN
   OPEN customer_cur;
   LOOP
      FETCH customer_cur into customer_rec;
      EXIT WHEN customer_cur%notfound;
      DBMS_OUTPUT.put_line(customer_rec.id || ' ' ||
customer_rec.name);
   END LOOP;
END;
/
```

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

```
1 Ramesh
2 Khilan
3 kaushik
4 Chaitali
5 Hardik
6 Komal

PL/SQL procedure successfully completed.
```

User-Defined Records

PL/SQL provides a user-defined record type that allows you to define different record structures. Records consist of different fields. Suppose you want to keep track of your books in a library. You might want to track the following attributes about each book:

- Title
- Author
- Subject
- Book ID

Defining a Record

The record type is defined as:

```
TYPE
type_name IS RECORD
  ( field_name1  datatype1  [NOT NULL]  [:= DEFAULT
EXPRESSION],
   field_name2   datatype2   [NOT NULL]  [:= DEFAULT
EXPRESSION],
   ...
   field_nameN  datatypeN  [NOT NULL]  [:= DEFAULT
EXPRESSION);
record-name  type_name;
```

Here is the way you would declare the Book record:

```
DECLARE
TYPE books IS RECORD
(title  varchar(50),
    author  varchar(50),
    subject varchar(100),
    book_id   number);
book1 books;
book2 books;
```

Accessing Fields

To access any field of a record, we use the dot (.) operator. The member access operator is coded as a period between the record variable name and the field that we wish to access. Following is the example to explain usage of 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;
/
```

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

```
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

PL/SQL procedure successfully completed.
```

PL/SQL - Exceptions

An error condition during a program execution is called an exception in PL/SQL. 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 exceptions
- User-defined exceptions

Syntax for Exception Handling

The General Syntax for exception handling is as follows. Here, you can list down as many as exceptions you want to handle. The default exception will be handled using *WHEN others THEN*:

```
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;
```

Example

Let us write some simple code to illustrate the concept. We will be using the CUSTOMERS table we had created and used in the previous chapters:

```
DECLARE
   c_id customers.id%type := 8;
   c_name  customers.name%type;
   c_addr customers.address%type;
BEGIN
   SELECT  name, address INTO  c_name, c_addr
   FROM customers
   WHERE id = c_id;

   DBMS_OUTPUT.PUT_LINE ('Name: '||  c_name);
   DBMS_OUTPUT.PUT_LINE ('Address: ' || c_addr);
EXCEPTION
   WHEN no_data_found THEN
      dbms_output.put_line('No such customer!');
   WHEN others THEN
      dbms_output.put_line('Error!');
END;
/
```

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

```
No such customer!
```

```

PL/SQL procedure successfully completed.
```

The above program displays the name and address of a customer whose ID is given. Since there is no customer with ID value 8 in our database, the program raises the run-time exception **NO_DATA_FOUND** which is captured in **EXCEPTION** block.

PL/SQL - Triggers

Triggers are stored programs, which are automatically executed or fired when some events occur. Triggers are, in fact, written to be executed in response to any of the following events:

- A database manipulation (DML) statement (DELETE, INSERT, or UPDATE).
- A database definition (DDL) statement (CREATE, ALTER, or DROP).
- A database operation (SERVERERROR, LOGON, LOGOFF, STARTUP, or SHUTDOWN).

Triggers could be defined on the table, view, schema, or database with which the event is associated.

Creating Triggers

The syntax for creating a trigger is:

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

Where,

- CREATE [OR REPLACE] TRIGGER trigger_name : Creates or replaces an existing trigger with the *trigger_name*.
- {BEFORE | AFTER | INSTEAD OF}: This specifies when the trigger would be executed. The INSTEAD OF clause is used for creating trigger on a view.
- {INSERT [OR] | UPDATE [OR] | DELETE}: This specifies the DML operation.

- [OF col_name]: This specifies the column name that would be updated.
- [ON table_name]: This specifies the name of the table associated with the trigger.
- [REFERENCING OLD AS o NEW AS n]: This allows you to refer new and old values for various DML statements, like INSERT, UPDATE, and DELETE.
- [FOR EACH ROW]: This specifies a row level trigger, i.e., the trigger would be executed for each row being affected. Otherwise the trigger will execute just once when the SQL statement is executed, which is called a table level trigger.
- WHEN (condition): This provides a condition for rows for which the trigger would fire. This clause is valid only for row level triggers.

Example:

To start with, we will be using the CUSTOMERS table:

```
Select * from customers;

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

The following program creates a **row level** trigger for the customers table that would fire for INSERT or UPDATE or DELETE operations performed on the CUSTOMERS table. This trigger will display the salary difference between the old values and new values:

```
CREATE OR REPLACE TRIGGER display_salary_changes
BEFORE DELETE OR INSERT OR UPDATE ON customers
FOR EACH ROW
WHEN (NEW.ID > 0)
DECLARE
   sal_diff number;
BEGIN
   sal_diff := :NEW.salary  - :OLD.salary;
   dbms_output.put_line('Old salary: ' || :OLD.salary);
   dbms_output.put_line('New salary: ' || :NEW.salary);
   dbms_output.put_line('Salary difference: ' || sal_diff);
END;
/
```

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

```
Trigger created.
```

Here following two points are important and should be noted carefully:

- OLD and NEW references are not available for table level triggers, rather you can use them for record level triggers.
- If you want to query the table in the same trigger, then you should use the AFTER keyword, because triggers can query the table or change it again only after the initial changes are applied and the table is back in a consistent state.
- Above trigger has been written in such a way that it will fire before any DELETE or INSERT or UPDATE operation on the table, but you can write your trigger on a single or multiple operations, for example BEFORE DELETE, which will fire whenever a record will be deleted using DELETE operation on the table.

PL/SQL - Packages

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

When the above code is executed at 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 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**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 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 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 SQL prompt, it prompt to enter 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.
```

PL/SQL - Collections

A collection is an ordered group of elements having the same data type. Each element is identified by a unique subscript that represents its position in the collection.

PL/SQL provides three collection types:

- Index-by tables or Associative array
- Nested table
- Variable-size array or Varray

Index-By Table

An **index-by** table (also called an associative array) is a set of **key-value** pairs. Each key is unique, and is used to locate the corresponding value. The key can be either an integer or a string.

An index-by table is created using the following syntax. Here we are creating an index-by table named **table_name** whose keys will be of *subscript_type* and associated values will be of *element_type*

```
TYPE type_name IS TABLE OF element_type [NOT NULL] INDEX
BY subscript_type;

table_name type_name;
```

Example:

Following example how to create a table to store integer values along with names and later it prints the same list of names.

```
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;
/
```

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

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

PL/SQL procedure successfully completed.
```

Nested Tables

A **nested table** is like a one-dimensional array with an arbitrary number of elements. However, a nested table differs from an array in the following aspects:

- An array has a declared number of elements, but a nested table does not. The size of a nested table can increase dynamically.
- An array is always dense i.e., it always has consecutive subscripts. A nested array is dense initially, but it can become sparse when elements are deleted from it.

An **nested table** is created using the following syntax:

```
TYPE type_name IS TABLE OF element_type [NOT NULL];

table_name type_name;
```

This declaration is similar to declaration of an **index-by** table, but there is no INDEX BY clause.

A nested table can be stored in a database column and so it could be used for simplifying SQL operations where you join a single-column table with a larger table. An associative array cannot be stored in the database.

Example:

The following examples illustrate the use of 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;
/
```

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

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

PL/SQL procedure successfully completed.
```

PL/SQL - Transactions

A database **transaction** is an atomic unit of work that may consist of one or more related SQL statements. It is called atomic because the database modifications brought about by the SQL statements that constitute a transaction can collectively be either committed, i.e., made permanent to the database or rolled back (undone) from the database.

A successfully executed SQL statement and a committed transaction are not same. Even if an SQL statement is executed successfully, unless the transaction containing the statement is committed, it can be rolled back and all changes made by the statement(s) can be undone.

Starting an Ending a Transaction

A transaction has a **beginning** and an **end**. A transaction starts when one of the following events take place:

- The first SQL statement is performed after connecting to the database.
- At each new SQL statement issued after a transaction is completed.

A transaction ends when one of the following events take place:

- A COMMIT or a ROLLBACK statement is issued.

- A DDL statement, like CREATE TABLE statement, is issued; because in that case a COMMIT is automatically performed.
- A DCL statement, such as a GRANT statement, is issued; because in that case a COMMIT is automatically performed.
- User disconnects from the database.
- User exits from SQL*PLUS by issuing the EXIT command, a COMMIT is automatically performed.
- SQL*Plus terminates abnormally, a ROLLBACK is automatically performed.
- A DML statement fails; in that case a ROLLBACK is automatically performed for undoing that DML statement.

Committing a Transaction

A transaction is made permanent by issuing the SQL command COMMIT. The general syntax for the COMMIT command is:

```
COMMIT;
```

Rolling Back Transactions

Changes made to the database without COMMIT could be undone using the ROLLBACK command.

The general syntax for the ROLLBACK command is:

```
ROLLBACK [TO SAVEPOINT < savepoint_name>];
```

When a transaction is aborted due to some unprecedented situation, like system failure, the entire transaction since a commit is automatically rolled back. If you are not using**savepoiny**, then simply use the following statement to rollback all the changes:

```
ROLLBACK;
```

Savepoints

Savepoints are sort of markers that help in splitting a long transaction into smaller units by setting some checkpoints. By setting savepoints within a long transaction, you can roll back to a checkpoint if required. This is done by issuing the SAVEPOINT command.

The general syntax for the SAVEPOINT command is:

```
SAVEPOINT < savepoint_name >;
```

Automatic Transaction Control

To execute a COMMIT automatically whenever an INSERT, UPDATE or DELETE command is executed, you can set the AUTOCOMMIT environment variable as:

```
SET AUTOCOMMIT ON;
```

You can turn-off auto commit mode using the following command:

```
SET AUTOCOMMIT OFF;
```

PL/SQL - Date & Time

PL/SQL provides two classes of date and time related data types:

- Datetime data types
- Interval data types

The Datetime data types are:

- DATE
- TIMESTAMP
- TIMESTAMP WITH TIME ZONE
- TIMESTAMP WITH LOCAL TIME ZONE

The Interval data types are:

- INTERVAL YEAR TO MONTH
- INTERVAL DAY TO SECOND

The Datetime Data Types

Following are the Datetime data types:

- **DATE** - it stores date and time information in both character and number datatypes. It is made of information on century, year, month, date, hour, minute, and second. It is specified as:
- **TIMESTAMP** - it is an extension of the DATE datatype. It stores the year, month, and day of the DATE datatype, along with hour, minute, and second values. It is useful for storing precise time values.

- **TIMESTAMP WITH TIME ZONE** - it is a variant of TIMESTAMP that includes a time zone region name or a time zone offset in its value. The time zone offset is the difference (in hours and minutes) between local time and UTC. This datatype is useful for collecting and evaluating date information across geographic regions.
- **TIMESTAMP WITH LOCAL TIME ZONE** - it is another variant of TIMESTAMP that includes a time zone offset in its value.

Examples:

```
SELECT SYSDATE FROM DUAL;
```

Output:

```
08/31/2012 5:25:34 PM
SELECT TO_CHAR(CURRENT_DATE, 'DD-MM-YYYY HH:MI:SS')
FROM DUAL;
```

Output:

```
31-08-2012 05:26:14
SELECT ADD_MONTHS(SYSDATE, 5) FROM DUAL;
```

Output:

```
01/31/2013 5:26:31 PM
SELECT LOCALTIMESTAMP FROM DUAL;
```

Output:

```
8/31/2012 5:26:55.347000 PM
```

The Interval Data Types

Following are the Interval data types:

- INTERVAL YEAR TO MONTH - it stores a period of time using the YEAR and MONTH datetime fields.
- INTERVAL DAY TO SECOND - it stores a period of time in terms of days, hours, minutes, and seconds.

PL/SQL - DBMS Output

The **DBMS_OUTPUT** is a built-in package that enables you to display output, display debugging information, and send messages from PL/SQL blocks, subprograms, packages, and triggers.

Example:

```
DECLARE
   lines dbms_output.chararr;
   num_lines number;
BEGIN
   -- enable the buffer with default size 20000
   dbms_output.enable;

   dbms_output.put_line('Hello Reader!');
   dbms_output.put_line('Hope you have enjoyed the tutorials!');
   dbms_output.put_line('Have a great time exploring pl/sql!');

   num_lines := 3;
   dbms_output.get_lines(lines, num_lines);

   FOR i IN 1..num_lines LOOP
      dbms_output.put_line(lines(i));
   END LOOP;
END;
/
```

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

```
Hello Reader!
Hope you have enjoyed the tutorials!
Have a great time exploring pl/sql!

PL/SQL procedure successfully completed.
```

PL/SQL - Object Oriented

PL/SQL allows defining an object type, which helps in designing object-oriented database in Oracle. An object type allows you to crate composite types. Using objects allow you implementing real world objects with specific structure of data and methods for operating it. Objects have attributes and methods. Attributes are properties of an object and are used for storing an object's state; and methods are used for modeling its behaviors.

Objects are created using the CREATE [OR REPLACE] TYPE statement. Below is an example to create a simple **address** object consisting of few attributes:

```
CREATE OR REPLACE TYPE address AS OBJECT
(house_no varchar2(10),
 street varchar2(30),
 city varchar2(20),
 state varchar2(10),
 pincode varchar2(10)
);
/
```

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

```
Type created.
```

Let's create one more object **customer** where we will wrap **attributes** and **methods**together to have object oriented feeling:

```
CREATE OR REPLACE TYPE customer AS OBJECT
(code number(5),
 name varchar2(30),
 contact_no varchar2(12),
 addr address,
 member procedure display
);
/
```

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

```
Type created.
```

Instantiating an Object

Defining an object type provides a blueprint for the object. To use this object, you need to create instances of this object. You can access the attributes and methods of the object using the instance name and **the access operator (.)** as follows:

```
DECLARE
   residence address;
BEGIN
   residence := address('103A', 'M.G.Road', 'Jaipur',
'Rajasthan','201301');
```

```
   dbms_output.put_line('House No: '|| residence.house_no);
   dbms_output.put_line('Street: '|| residence.street);
   dbms_output.put_line('City: '|| residence.city);
   dbms_output.put_line('State: '|| residence.state);
   dbms_output.put_line('Pincode: '|| residence.pincode);
END;
/
```

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

```
House No: 103A
Street: M.G.Road
City: Jaipur
State: Rajasthan
Pincode: 201301

PL/SQL procedure successfully completed.
```

PL/SQL PROGRAMS

1. WRITE A PL/SQL PROGRAM TO CHECK THE GIVEN NUMBER IS STRONG OR NOT.

```
SQL> Declare
          n number:=&n;
          n1 number:=n;
          fact number;
          sum1 number;
          r number;
          i number;
          Begin
          sum1:=0;
          while(n>0) loop
          r:=mod(n,10);
          n:=trunc(n/10);
          i:=1;
          fact:=1;
          while(i<=r)loop
          fact:=fact*i;
          i:=i+1;
          end loop;
```

```
sum1:=sum1+fact;
end loop;
if(sum1=n1) then
dbms_output.put_line(n1||' is strong number');
else
dbms_output.put_line(n1||' is not a strong number');
end if;
END;
/
```

Output1:

```
Enter value for n: 145
old   2: n number:=&n;
new   2: n number:=145;
145 is strong number
PL/SQL procedure successfully completed.
```

Output2:

```
Enter value for n: 121
old   2: n number:=&n;
new   2: n number:=121;
121 is not a strong number
```

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

2. WRITE A PL/SQL PROGRAM TO CHECK THE GIVEN STRING IS PALINDROME OR NOT.

```
SQL> Declare
len number;
s1 varchar2(20) := '&s1';
s2 varchar2(20);
Begin
len := length(s1);
for i in REVERSE 1..len loop
s2 := s2||substr(s1,i,1);
end loop;
if s2= s1 then
dbms_output.put_line('Given String '||s1||' is a PALINDROME');
else
dbms_output.put_line('Given String '||s1||' is not a PALINDROME');
end if;
END;
/
```

Output1:

```
Enter value for s1: madam
old   3: s1 varchar2(20) := '&s1';
new   3: s1 varchar2(20) := 'madam';
Given String madam is a PALINDROME

PL/SQL procedure successfully completed.
```

Output2:

```
Enter value for s1: moam
old   3: s1 varchar2(20) := '&s1';
new   3: s1 varchar2(20) := 'moam';
Given String moam is not a PALINDROME

PL/SQL procedure successfully completed.
```

3. WRITE A PL/SQL PROGRAM TO SWAP TWO NUMBERS WITHOUT USING THIRD VARIABLE.

```
SQL> Declare
a number:=&a;
b number:=&b;
Begin
```

```
dbms_output.put_line('Before Swapping :'||a||' ' ||b);
a:=a+b;
b:=a-b;
a:=a-b;
dbms_output.put_line('After Swapping:'||a||' '||b);
END;
/
```

Output:

```
Enter value for a: 10
old   2: a number:=&a;
new   2: a number:=10;
Enter value for b: 30
old   3: b number:=&b;
new   3: b number:=30;
Before Swapping :10 30
After Swapping:30 10

PL/SQL procedure successfully completed.
```

4. WRITE A PL/SQL PROGRAM TO GENERATE MULTIPLICATION TABLES FOR 2,4,6

```
SQL> Declare
i number:=2;
Begin
dbms_output.put_line('- - - - - - - - - -');
while (i<=6) loop
dbms_output.put_line(i||' Multiplication table');
dbms_output.put_line('- - - - - - - - - -');
for j in 1 .. 10 loop
dbms_output.put_line(i||'*'||j||'='||(i*j));
end loop;
i:=i+2;
dbms_output.put_line('- - - - - - - - - -');
end loop;
END;
/
```

<u>Output:</u>

```
- - - - - - - - - -
2 Multiplication table
```

```
- - - - - - - - - -

2*1=2

2*2=4

2*3=6

2*4=8

2*5=10

2*6=12

2*7=14

2*8=16

2*9=18

2*10=20

- - - - - - - - - -

4 Multiplication table

- - - - - - - - - -

4*1=4

4*2=8

4*3=12

4*4=16

4*5=20

4*6=24

4*7=28
```

```
4*8=32

4*9=36

4*10=40

- - - - - - - - - -

6 Multiplication table

- - - - - - - - - -

6*1=6

6*2=12

6*3=18

6*4=24

6*5=30

6*6=36

6*7=42

6*8=48

6*9=54

6*10=60

- - - - - - - - - -


PL/SQL procedure successfully completed.
```

5. WRITE A PL/SQL PROGRAM TO DISPLAY SUM OF EVEN NUMBERS AND
SUM OF ODD NUMBERS IN THE GIVEN RANGE.

```
SQL> Declare
n number;
i number;
evensum number;
oddsum number;
Begin
--dbms_output.put_line('Enter a number to find even sum and odd
sum');
n:=&n;
i:=1;
oddsum:=0;
evensum:=0;
while i<=n loop
if(mod(i,2)=0) then
evensum:=evensum+i;
else
oddsum:=oddsum+i;
end if;
```

```
i:=i+1;
end loop;
dbms_output.put_line('Even sum= '||evensum);
dbms_output.put_line('odd sum= '||oddsum);
END;
/
```

Output:

```
Enter value for n: 5
old   8: n:=&n;
new   8: n:=5;
Even sum= 6
odd sum= 9
PL/SQL procedure successfully completed.
```

6. WRITE A PL/SQL PROGRAM TO CHECK THE GIVEN NUMBER IS PALLINDROME OR NOT.

```
SQL>  Declare
n number;
sum1 number:=0;
r number;
k number;
```

```
Begin
n:=&n;
k:=n;
while k>0 loop
r:=k mod 10;
sum1:=sum1*10+r;
k:=trunc(k/10);
end loop;
if(sum1=n) then
dbms_output.put_line('Given Number' ||n||' is Palindrome');
else
dbms_output.put_line('Given Number' ||n||' is not a Palindrome');
end if;
END;.
/
```

Output1:

```
Enter value for n: 141
old   7: n:=&n;
new   7: n:=141;
Given Number141 is Palindrome
PL/SQL procedure successfully completed.
```

Output2:

```
Enter value for n: 121
old   7: n:=&n;
new   7: n:=121;
Given Number121 is Palindrome
PL/SQL procedure successfully completed.
```

7. The hrd manager has decided to raise the employee salary by 20%. Write a Pl/Sql block to accept the employee number and update the salary of that employee. Display appropriate message based on the existence of the record in emp table.

```
SQL>declare
e emp%rowtype;
no emp.empno %type;
sa emp.sal%type;
begin
no:=&no;
select * into e from emp where empno=no;
sa:=e.sal+(15/100)*e.sal;
update emp set sal=sa where empno=no;
```

```
dbms_output.put_line('employee record is modified');
dbms_output.put_line('employee number is '||e.empno);
dbms_output.put_line('employee name is '||e.ename);
dbms_output.put_line('employee job is '||e.job);
dbms_output.put_line('employee sal is '||sa);
end;
/
```

Output:

```
Enter value for no: 7499
old   6: no:=&no;
new   6: no:=7499;
employee record is modified
employee number is 7499
employee name is allen
employee job is salesman
employee sal is 1840
PL/SQL procedure successfully completed.
```

8. WRITE A PL/SQL PROGRAM TO DISPLAY TOP 10 ROWS IN EMP TABLE BASED ON THEIR JOB AND SALARY.

```
SQL> Declare cursor c is select * from emp e where 10> (select
count(distinct sal) from emp a where a.sal>e.sal);
        Begin
        for rec in c loop
        dbms_output.put_line(rec.empno||' '||rec.ename||' '||rec.sal);
        end loop;
        END;
        /
```

Output:

```
        7839    king       5000
        7698    blake       2850
        7782    clark      2450
        7566    jones       2975
        7654    martin      1250
        7499    allen      1600
        7844    turner      1500
        7521    ward        1250
        7902    ford       3000
        7788    scott      3000
        7876   adams       1100
```

```
7934     miller     1300
PL/SQL procedure successfully completed.
```

9. Write a Pl/Sql program to raise the employee salary by 10%, for department number 30 people and also maintain the raised details in the raise table.

```
SQL> create table raise_emp
(empid number(4) primary key,
name varchar2(10),
desig varchar2(9),
mgr number(4),
doj date,
salary number(7,2),
comm number(7,2),
dno number(2));
```

Output:

```
Table created.
```

```
SQL> declare
     cursor c2 is select * from emp where deptno=30 for update;
     e emp%rowtype;
```

```
begin
open c2;
loop
fetch c2 into e;
exit when c2%notfound;
update emp set sal=e.sal+(10/100)*e.sal where current of c2;
end loop;
dbms_output.put_line('employee details are stored in the
raise_emp table');
insert into raise_emp(select * from emp where deptno=30);
close c2;
end;
/
employee details are stored in the raise_emp table
PL/SQL procedure successfully completed.
```

Output:

```
SQL> select * from raise_emp;
```

EMPID	NAME	DESIG	MGR	DOJ	SALARY	COMM	DNO

7698	blake	manager	7839	01-MAY-81	3135	-	30
7654	martin	salesman	7698	28-SEP-81	1375	1400	30
7499	allen	salesman	7698	20-FEB-81	1760	300	30
7844	turner	salesman	7698	08-SEP-81	1650	0	30
7900	james	clerk	7698	03-DEC-81	1045	-	30

7521	ward	salesman	7698	22-FEB-81	1375	500	30

6 rows returned

10. WRITE A PROCEDURE TO UPDATE THE SALARY OF EMPLOYEE, WHO ARE NOT GETTING COMMISSION 10%.

```
SQL> create or replace procedure raise_comm as
begin
update emp set sal=sal+(10/100)*sal where comm is null or comm=0;
end;
/
```

Output:

```
  Procedure created.
```

```
SQL> begin
            raise_comm();
```

```
          end;
          /
```

PL/SQL procedure successfully completed.

11 records updated

Output:

SQL> select * from emp;

EMPNO	ENAME	JOB	MGR	HIREDATE	SAL	COMM	DEPTNO
7839	king	president	-	17-NOV-81	5500	-	10
7698	blake	manager	7839	01-MAY-81	3135	-	30
7782	clark	manager	7839	09-JUN-81	2695	-	10
7566	jones	manager	7839	02-APR-81	3272.5	-	20
7654	marti	sales	769	28-	125	1400	30

	n	man	8	SEP-81	0		
7499	allen	salesman	7698	20-FEB-81	1600	300	30
7844	turner	salesman	7698	08-SEP-81	1650	0	30
7900	james	clerk	7698	03-DEC-81	1045	-	30
7521	ward	salesman	7698	22-FEB-81	1250	500	30
7902	ford	analyst	7566	03-DEC-81	3300	-	20
7369	smith	clerk	7902	17-DEC-80	880	-	20
7788	scott	analyst	7566	09-DEC-82	3300	-	20
7876	adams	clerk	7788	12-JAN-83	1210	-	20

7934	miller	clerk	7782	23-JAN-82	1430	-	10

14 rows returned

11. WRITE A PL/SQL PROCEDURE TO PREPARE AN ELECTRICITY BILL BY USING FOLLOWING TABLE

TABLE USED: ELECT

```
NAME                    NULL?       TYPE


MNO                   NOT NULL      NUMBER(3)
CNAME                                VARCHAR2(20)
CUR_READ                             NUMBER(5)
PREV_READ                              NUMBER(5)
  NO_UNITS                          NUMBER(5)
AMOUNT                         NUMBER(8,2)
SER_TAX                        NUMBER(8,2)
NET_AMT                         NUMBER(9,2)
```

```
SQL> create table elect (mno number(3) primary key, cname
varchar2(20), cur_read number(5), prev_read number(5), no_units
number(5), amount number(8,2), ser_tax number(8,2), net_amt
number(9,2) );
```

Output:

```
Table created.
```

```
SQL> desc empr;
```

Output:

Table	Column	Data Type	Length	Precision	Scale	Primary Key	Nullable	Default	Comment
EMPR	ENO	Number	-	3	0	-	✓	-	-
	ENAME	Varchar2	20	-	-	-	✓	-	-
	PHONE	Emp_Phone	2317	-	-	-	✓	-	-
									1 - 3

```
SQL> Insert into elect(mno,cname,cur_read,prev_read)
  2  Values(&mno,'&cname',&cur_read,&prev_read);
Enter value for mno: 1
Enter value for cname: raju
Enter value for cur_read: 2000
Enter value for prev_read: 1600
old   2: Values(&mno,'&cname',&cur_read,&prev_read)
new   2: Values(1,'raju',2000,1600)

1 row created.
SQL> /
Enter value for mno: 2
Enter value for cname: rani
Enter value for cur_read: 3000
Enter value for prev_read: 2000
old   2: Values(&mno,'&cname',&cur_read,&prev_read)
new   2: Values(2,'rani',3000,2000)

1 row created.
SQL> /
Enter value for mno: 4
Enter value for cname: rajulu

Enter value for cur_read: 300

Enter value for prev_read: 200

old   2: Values(&mno,'&cname',&cur_read,&prev_read)

new   2: Values(4,'rajulu',300,200)


1 row created.

SQL> select * from elect;
```

MNO	CNAME	CUR_READ	PREV_READ	NO_UNITS	AMOUNT	SER_TAX	NET_AMT
1	raju	2000	1600	-	-	-	-
2	rani	3000	2000	-	-	-	-
4	rajulu	300	200	-	-	-	-

```
3 rows returned


SQL> create or replace procedure powerbill

As no_un elect.no_units%type;

amt elect.amount%type;
```

```
stax elect.ser_tax%type;
net elect.net_amt%type;
e elect%rowtype;
cursor c9 is select * from elect for update;
begin
open c9;
loop
fetch c9 into e;
exit when c9%notfound;
no_un:=e.cur_read-e.prev_read;
amt:=no_un*1.5;
stax:=(5/100)*amt;
net:=amt+stax;
update elect set no_units=no_un,amount=amt,
ser_tax=stax,net_amt=net where current of c9;
dbms_output.put_line('customer meter number'||e.mno);
dbms_output.put_line('customer name'||e.cname);
dbms_output.put_line('total amount is '||net);
end loop;
close c9;
end;
```

```
/
```

Output:

```
Procedure created

SQL> exec powerbill;

PL/SQL procedure successfully completed.
SQL> commit;
Commit complete.
```

MNO	CNAME	CUR_READ	PREV_READ	NO_UNITS	AMOUNT	SER_TAX	NET_AMT
1	raju	2000	1600	400	600	30	630
2	rani	3000	2000	1000	1500	75	1575
4	rajulu	300	200	100	150	7.5	157.5

12. WRITE A PL/SQL PROCEDURE TO PREPARE AN TELEPHONE BILL BY USING FOLLOWING TABLE. AND PRINT THE MOTHLY BILLS FOR EACH CUSTOMER

TABLE USED : PHONE.

```
NAME                    NULL?          TYPE
     --------------------------- --------         ---------
 TEL_NO          NOT NULL            NUMBER(6)
      CNAME                                VARCHAR2(20)
      CITY                                 VARCHAR2(10)
      PR_READ                              NUMBER(5)
      CUR_READ                             NUMBER(5)
      NET_UNITS                            NUMBER(5)
      TOT_AMT                     NUMBER(8,2)
```

```
SQL>Create table phone
(
Telno number(6) primary key,
Cname varchar2(20),
City varchar2(10),
Pr_read number(5),
Cur_read number(5),
Net_units number(5),
Tot_amt number(8,2)
);
```

Output:

```
Table created.
SQL> desc phone;
```

Output:

Table	Column	Data Type	Length	Precision	Scale	Primary Key	Nullable	Default	Comment
PHONE	TELNO	Number	-	6	0	1	-	-	-
	CNAME	Varchar2	20	-	-	-	✓	-	-
	CITY	Varchar2	10	-	-	-	✓	-	-
	PR_READ	Number	-	5	0	-	✓	-	-
	CUR_READ	Number	-	5	0	-	✓	-	-

	NET_U NITS	Number	-	5	0	-	✓	-	-
	TOT_A MT	Number	-	8	2	-	✓	-	-

```
SQL> insert into phone(telno,cname,city,pr_read,cur_read)
  2  values('&telno','&cname','&city',&pr_read,&net_units);
Enter value for telno: 222222
Enter value for cname: scott
Enter value for city: vijayawada
Enter value for pr_read: 200
Enter value for net_units: 350
old   2: values('&telno','&cname','&city',&pr_read,&net_units)
new   2: values('222222','scott','vijayawada',200,350)

1 row created.

SQL> /
Enter value for telno: 333333
Enter value for cname: tiger
Enter value for city: guntur
Enter value for pr_read: 500
Enter value for net_units: 650
old   2: values('&telno','&cname','&city',&pr_read,&net_units)
new   2: values('333333','tiger','guntur',500,650)

1 row created.

SQL> /
Enter value for telno: 444444
Enter value for cname: system
Enter value for city: anu
Enter value for pr_read: 4000
Enter value for net_units: 5500
old   2: values('&telno','&cname','&city',&pr_read,&net_units)
new   2: values('444444','system','anu',4000,5500)

1 row created.

SQL> select * from phone;
```

Output:

TELNO	CNAME	CITY	PR_READ	CUR_READ	NET_UNI TS	TOT_AMT
222222	scott	vijayawada	200	350	-	-
333333	tiger	guntur	500	650	-	-
444444	system	anu	4000	5500	-	-

Procedure:

```
SQL> create or replace procedure phonebill
   as
   nunit phone.net_units%type;
   tamt phone.tot_amt%type;
   p phone%rowtype;
   cursor c is select * from phone for update;
```

```
 begin
 open c;
 loop
fetch c into p;
exit when c%notfound;
nunit:=p.cur_read-p.pr_read;
tamt:=nunit*0.5;
update phone set net_units=nunit where current of c;
update phone set tot_amt=tamt where current of c;
dbms_output.put_line('customer phone number'||p.telno);
dbms_output.put_line('customer name'||p.cname);
dbms_output.put_line('city'||p.city);
dbms_output.put_line('total amount'||tamt);
end loop;
close c;
end;
/
Procedure created.
SQL> exec phonebill;
```

Output:

PL/SQL procedure successfully completed

SQL> commit;

Commit complete.

SQL> select * from phone;

Output:

TELNO	CNAME	CITY	PR_READ	CUR_READ	NET_UNITS	TOT_AMT
222222	scott	vijayawada	200	350	150	75
333333	tiger	guntur	500	650	150	75
444444	system	anu	4000	5500	1500	750

3 rows

returned

13. WRITE A PL/SQL PROGRAM TO RAISE THE EMPLOYEE SALARY BY 10%, WHO ARE COMPLETED THERE 25 YEARS OF SERVICE AND STORE THE DETAILS AT PPROPRIATE TABLES (DEFINE THE RETAIR_EMP TABLE).

SQL> insert into Emp values (7856,'raju','clerk',7839,'23-jan-92',1300,null,10);

1 row created.

SQL> select * from emp;

EMPNO	ENAME	JOB	MGR	HIREDATE	SAL	COMM	DEPTNO
7839	king	president	-	17-NOV-81	5000	-	10
7698	blake	manager	7839	01-MAY-81	2850	-	30
7782	clark	manager	7839	09-JUN-81	2450	-	10
7566	jones	manager	7839	02-APR-81	2975	-	20
7654	martin	salesman	7698	28-SEP-81	1250	1400	30

7499	allen	salesman	7698	20-FEB-81	1600	300	30
7844	turner	salesman	7698	08-SEP-81	1500	0	30
7900	james	clerk	7698	03-DEC-81	950	-	30
7521	ward	salesman	7698	22-FEB-81	1250	500	30
7902	ford	analyst	7566	03-DEC-81	3000	-	20
7369	smith	clerk	7902	17-DEC-80	800	-	20
7788	scott	analyst	7566	09-DEC-82	3000	-	20
7876	adams	clerk	7788	12-JAN-83	1100	-	20
7934	miller	clerk	7782	23-JAN-82	1300	-	10
7856	raju	clerk	7839	23-JAN-92	1300	-	10

15 rows
returned

SQL> Create table retair_emp(empno number NOT NULL, ename varchar2(20),job varchar2(20),hiredate date,sal number,deptno number);

Table created.

SQL>desc retair_emp;

Table	Column	Data Type	Length	Precision	Scale	Primary Key	Nullable	Default	Comment
RETAIR_EMP	EMPNO	Number	-	-	-	-	-	-	-
	ENAME	Varchar2	20	-	-	-	✓	-	-
	JOB	Varchar2	20	-	-	-	✓	-	-
	HIREDATE	Date	7	-	-	-	✓	-	-
	SAL	Number	-	-	-	-	✓	-	-
	DEPTNO	Number	-	-	-	-	✓	-	-

```
SQL> declare
cursor c5 is select * from emp where
extract(year from sysdate)-extract(year from hiredate)>25 for update;
e emp%rowtype;
begin
open c5;
loop
fetch c5 into e;
exit when c5%notfound;
update emp set sal=e.sal+(30/100)*e.sal where current of c5;
insert into retair_emp(empno,ename,job,hiredate,sal,deptno) select
empno,ename,job,hiredate,sal,deptno from emp where
empno=e.empno;
end loop;
close c5;
end;
/
```

Output:

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

Output:

```
SQL> select * from retair_emp;
```

EMPNO	ENAME	JOB	HIREDATE	SAL	DEPTNO
7839	king	president	17-NOV-81	6500	10
7698	blake	manager	01-MAY-81	3705	30
7782	clark	manager	09-JUN-81	3185	10
7566	jones	manager	02-APR-81	3867.5	20
7654	martin	salesman	28-SEP-81	1625	30
7499	allen	salesman	20-FEB-81	2080	30
7844	turner	salesman	08-SEP-81	1950	30
7900	james	clerk	03-DEC-	123	30

			81	5	
7521	ward	sales man	22-FEB-81	1625	30
7902	ford	analyst	03-DEC-81	3900	20
7369	smith	clerk	17-DEC-80	1040	20
7788	scott	analyst	09-DEC-82	3900	20
7876	adams	clerk	12-JAN-83	1430	20
7934	miller	clerk	23-JAN-82	1690	10

14 rows returned

14. WRITE A PL/SQL PROCEDURE TO EVALUATE THE GRADE OF A STUDENT WITH FOLLOWING CONDITIONS:

FOR PASS: ALL MARKS > 40

FOR I CLASS: TOTAL%>59

FOR II CLASS: TOTAL% BETWEEN >40 AND <60

FOR III CLASS: TOTAL% =40

AND ALSO MAINTAIN THE DETAILS IN ABSTRACT TABLE.

TABLES USED

1. TABLE STD

```
NAME                          NULL?                    TYPE
 ------------------------------ --------                ----
 NO                       NOT NULL                    NUMBER
 NAME                                            VARCHAR2(10)
INTNO
NUMBER
CLASS                     NOT NULL               VARCHAR2(10)
M1                                                NUMBER
 M2                                              NUMBER
M3                                                NUMBER
 M4                                               NUMBER
 M5                                                          NUMBER
```

2. TABLE ABSTRACT

```
NAME                          NULL?                TYPE
 ------------------------------     --------               ----
 STDNO                                           NUMBER
 STDNAME                                          VARCHAR2(10)
CLASS                                            VARCHAR2(10)
 MONTH                                           VARCHAR2(10)
 INTNO (INTERNAL NUMBER)                    NUMBER
TOT                                              NUMBER
GRADE                                            VARCHAR2(10)
PERCENT                                          NUMBER
  DAT_ENTER                                      DATE
```

```
SQL> create table std(no number primary key,name
varchar2(10),intno number,class varchar2(10)NOT NULL,M1
number,M2 number,M3 number,M4 number,M5 number);

Table created
```

SQL> desc std;

Output:

Table	Column	Data Type	Length	Precision	Scale	Primary Key	Nullable	Default	Comment
STD	NO	Number	-	-	-	1	-	-	-
	NAME	Varchar2	10	-	-	-	✓	-	-
	INTNO	Number	-	-	-	-	✓	-	-
	CLASS	Varchar2	10	-	-	-	-	-	-
	M1	Number	-	-	-	-	✓	-	-
	M2	Number	-	-	-	-	✓	-	-
	M3	Number	-	-	-	-	✓	-	-
	M4	Number	-	-	-	-	✓	-	-
	M5	Number	-	-	-	-	✓	-	-

```
SQL> insert into std
values(&no,'&name',&intno,'&class',&m1,&m2,&m3,&m4,&m5);
Enter value for no: 1
Enter value for name: raju
Enter value for intno: 1
Enter value for class: MCA
Enter value for m1: 40
Enter value for m2: 45
Enter value for m3: 68
Enter value for m4: 75
Enter value for m5: 55
old   1: insert into std
values(&no,'&name',&intno,'&class',&m1,&m2,&m3,&m4,&m5)

new   1: insert into std values(1,'raju',1,'MCA',40,45,68,75,55)

1 row created.

SQL> /
Enter value for no: 2
Enter value for name: rani
Enter value for intno: 1
Enter value for class: MCA
Enter value for m1: 35
Enter value for m2: 68
Enter value for m3: 44
Enter value for m4: 48
Enter value for m5: 88
old   1: insert into std
values(&no,'&name',&intno,'&class',&m1,&m2,&m3,&m4,&m5)

new   1: insert into std values(2,'rani',1,'MCA',35,68,44,48,88)

1 row created.

SQL> /
Enter value for no: 3
Enter value for name: basu
Enter value for intno: 2
Enter value for class: MCA
Enter value for m1: 45
Enter value for m2: 45
Enter value for m3: 65
Enter value for m4: 25
Enter value for m5: 77
old   1: insert into std
values(&no,'&name',&intno,'&class',&m1,&m2,&m3,&m4,&m5)

new   1: insert into std values(3,'basu',2,'MCA',45,45,65,25,77)

1 row created.

SQL> /
Enter value for no: 4
Enter value for name: horse
Enter value for intno: 2
Enter value for class: MCA
Enter value for m1: 45
Enter value for m2: 24
Enter value for m3: 24
Enter value for m4: 55
Enter value for m5: 85
```

```
old   1: insert into std
values(&no,'&name',&intno,'&class',&m1,&m2,&m3,&m4,&m5)

new   1: insert into std values(4,'horse',2,'MCA',45,24,24,55,85)

1 row created.

SQL> /
Enter value for no: 5
Enter value for name: girl
Enter value for intno: 1
Enter value for class: MCA
Enter value for m1: 45
Enter value for m2: 65
Enter value for m3: 78
Enter value for m4: 87
Enter value for m5: 98
old   1: insert into std
values(&no,'&name',&intno,'&class',&m1,&m2,&m3,&m4,&m5)

new   1: insert into std values(5,'girl',1,'MCA',45,65,78,87,98)
```

```
1 row created.

SQL> /
Enter value for no: 6
Enter value for name: tiger
Enter value for intno: 2
Enter value for class: MCA
Enter value for m1: 45
Enter value for m2: 85
Enter value for m3: 98
Enter value for m4: 78
Enter value for m5: 95
old   1: insert into std
values(&no,'&name',&intno,'&class',&m1,&m2,&m3,&m4,&m5)

new   1: insert into std values(6,'tiger',2,'MCA',45,85,98,78,95)

1 row created.
SQL> select * from std;
```

Output:

NO	NAME	INTNO	CLASS	M1	M2	M3	M4	M5
1	raju	1	MCA	40	45	68	75	55
2	rani	1	MCA	35	68	44	48	88
3	basu	2	MCA	45	45	65	25	77
4	horse	2	MCA	45	24	24	55	85
5	girl	1	MCA	45	65	78	87	98
6	tiger	2	MCA	45	85	98	78	95

6 rows returned

```
SQL> create table abstract(stdno number NOT NULL,stdname
varchar2(10), class varchar2(10),month varchar2(10),intno
number,tot number,grade varchar2(10) percent number,dat_enter
date);

Table created

SQL> desc abstract;
```

Output:

Table	Column	Data Type	Length	Precision	Scale	Primary Key	Nullable	Default	Comment
ABSTRACT	STDNO	Number	-	-	-	-	-	-	-

	STDNAME	Varchar2	10	-	-	-	✓	-	-
	CLASS	Varchar2	10	-	-	-	✓	-	-
	MONTH	Varchar2	10	-	-	-	✓	-	-
	INTNO	Number	-	-	-	-	✓	-	-
	TOT	Number	-	-	-	-	✓	-	-
	GRADE	Varchar2	10	-	-	-	✓	-	-
	PERCENT	Number	-	-	-	-	✓	-	-
	DAT_ENTER	Date	7	-	-	-	✓	-	-

```
SQL> create or replace function gradeall return varchar2 is
cursor c is select m1,m2,m3,m4,m5 from std;
mark1 integer;
mark2 integer;
mark3 integer;
```

```
mark4 integer;
mark5 integer;
average number(5,2);
tot integer;
begin
open c;
loop
fetch c into mark1,mark2,mark3,mark4,mark5;
exit when c%notfound;
tot:= mark1+mark2+mark3+mark4+mark5;
average:=tot/5;
if(mark1>40 and mark2>40 and mark3>40 and mark4>40 and
mark5>40) then
return('Pass');
elsif(average>59) then return('I Class');
elsif(average>40 and average<60) then return('II Class');
elsif(average=40) then return('III Class');
else return('FAIL');
end if;
end loop;
close c;
```

```
end;
/
```

Output:

```
Function created.
```

```
SQL> create or replace procedure Grade( stdno in
abstract.stdno%type, stdname in abstract.stdname%type, class in
abstract.class%type, month in abstract.month%type ,intno in
abstract.intno%type, tot in abstract.tot%type,g in
abstract.grade%type,percent in abstract.percent%type, dat_enter in
abstract.dat_enter%type ) as
 begin
 insert into abstract
values(stdno,stdname,class,month,intno,tot,g,percent,dat_enter);
 end;
/
```

Output:

```
Procedure created.
```

```
SQL> declare
cursor c is select no,name,intno,class, M1,M2,M3,M4,M5 From std;
```

```
mark1 integer;
mark2 integer;
mark3 integer;
mark4 integer;
mark5 integer;
stdno  abstract.stdno%type;
 stdname abstract.stdname%type;
 class  abstract.class%type;
 month  abstract.month%type:= to_char(sysdate, 'MON') ;
intno  abstract.intno%type;
 tot  abstract.tot%type;
g  abstract.grade%type;
percent  abstract.percent%type;
 dat_enter  abstract.dat_enter%type:=sysdate();
begin
open c;
loop
fetch c into stdno,stdname,intno,class, mark1, mark2, mark3, mark4,
mark5;
exit when c%notfound;
tot:= mark1+ mark2+ mark3+ mark4+ mark5;
```

```
percent:=tot/5;
g:=gradeall();
grade(stdno , stdname , class , month ,intno , tot ,g ,percent ,
dat_enter );
end loop;
close c;
end;
/
```

Output:

```
PL/SQL procedure successfully completed.

SQL> select * from abstract;
```

Output:

STDNO	STDNAME	CLASS	MONTH	INTNO	TOT	GRADE	PERCENT	DAT_ENTER
1	raju	MCA	MAY	1	283	II Class	56.6	06-MAY-15
2	rani	MCA	MAY	1	283	II Class	56.6	06-MAY-15
3	basu	MCA	MAY	2	257	II Class	51.4	06-MAY-15
4	horse	MCA	MAY	2	233	II Class	46.6	06-MAY-15
5	girl	MCA	MAY	1	373	II Class	74.6	06-MAY-15
6	tiger	MCA	MAY	2	401	II Class	80.2	06-MAY-15

15. CREATE AN VARRAY, WHICH HOLDS THE EMPLOYEE PHONE NUMBERS (AT LEAST THREE NUMBERS)

```
SQL> create type emp_phone is varray(100)of number(7);
   /
Type created
```

```
SQL> desc emp_phone;
```

<u>Output:</u>

Name	Null?	Type
emp_phone		VARRAY(100) OF NUMBER(7)

```
SQL> create table emps(eno number(3),ename varchar2(20),phone
emp_phone);

Table created.

SQL> desc emps;
```

Output:

Table	Column	Data Type	Length	Precision	Scale	Primary Key	Nullable	Default	Comment
EMPS	ENO	Number	-	3	0	-	✓	-	-
	ENAME	Varchar2	20	-	-	-	✓	-	-
	PHONE	Emp_Phone	2317	-	-	-	✓	-	-

```
SQL> insert into emps values(100,'sai',emp_phone(12,13))
1 row created.
```

```
SQL> insert into emps
values(200,'rani',emp_phone(4444444,121212,131313,141414));
1 row created.
SQL> select * from emps;
```

Output:

EMPNO	ENAME	PHONE
100	sai	EMP_PHONE(12, 13)
200	rani	EMP_PHONE(4444444, 121212, 131313, 141414)

16. CREATE AN OBJECT TO DESCRIBE THE DETAILS OF ADDRESS TYPE DATA.

```
SQL> create type address as object (street_address
varchar2(35),city varchar2(13) ,state varchar2(30),zipcode
varchar2(20));

Type created.

SQL> desc address;
```

Output:

Name	Null?	Type
STREET_ADDRESS		VARCHAR2(35)
CITY		VARCHAR2(13)
STATE		VARCHAR2(30)
ZIPCODE		VARCHAR2(20)

SQL> create table emps1(empno number(2),ename varchar2(20),eaddress address);

Table created

SQL> desc emps1;

Output:

Table	Column	Data Type	Length	Precision	Scale	Primary Key	Nullable	Default	Comment
EMP	EMPNO	Numbe	-	2	0	-	✓	-	-

S1		r							
	ENAME	Varchar 2	20	-	-	-	✓	-	-
	EADDRESS	Addres s	1	-	-	-	✓	-	-

```
SQL> insert into emps1 values(01,'sai',address('as','gnt','ap',432));
1 row created.

SQL> insert into emps1 values(02,'rani',address('sat','bza','ap',520));

1 row created.
SQL> select * from emps1;
```

Output:

EMPNO	ENAME	EADDRESS(STREET_ADDRESS, CITY, STATE, ZIPCODE)
1	sai	ADDRESS('as', 'gnt', 'ap', '432')

2	rani	ADDRESS('sat', 'bza', 'ap', '520')

17. WRITE A PL/SQL PROCEDURE TO READ THE DATA INTO THE TABLE AS PER THE FOLLOWING DESCRIPTION

Attribute Name	Data Type	DETAILS
EMPLOYEE NUMBER	**NUMBER**	
EMPLOYEE NAME	**CHARACTER**	
ADDRESS	**OBJECT**	**STREET NUMBER**
		STREET NAME
		TOWN
		DIST AND STATE
QUALIFICATION	**CHARACTER**	
PHONE NUMBER	**OBJECT-**	**HOLDS THREE PHONE**

	VARRAY	**NUMBER**

SQL> create table emp_obj(eno number(5),ename

varchar2(20),address1 address,qual

ification varchar2(20),ph_no emp_phone);

Table created.

SQL> desc emp_obj;

Table	**Column**	**Data Type**	**Length**	**Precision**	**Scale**	**Primary Key**	**Nullable**	**Default**	**Comment**
EMP_OBJ	ENO	Number	-	5	0	-	✓	-	-
	ENAME	Varchar2	20	-	-	-	✓	-	-

	ADDRESS1	Address	1	-	-	-	✓	-	-
	QUALIFICATION	Varchar2	20	-	-	-	✓	-	-
	PH_NO	Emp_Phone	2317	-	-	-	✓	-	-

```
SQL>    create or replace procedure edetails( eno in
emp_obj.eno%type,
  2   ename in emp_obj.ename%type,
  3   address1 in emp_obj.address1%type,
  4   qualification in emp_obj.qualification%type,
  5   phno in emp_obj.ph_no%type
  6   ) as
  7   begin
  8   insert into emp_obj
values(eno,ename,address1,qualification,phno);
  9   end;
 10  /
```

Output:

```
Procedure created.
SQL> declare
  2   eno emp_obj.eno%type:='&eno';
  3    ename emp_obj.ename%type:='&ename';
  4    street_address varchar2(35):='&street_address';
  5    city varchar2(13):='&city';
  6    state varchar2(20):='&state';
  7    zipcode varchar2(20):='&zipcode';
  8    qualification emp_obj.qualification%type:='&qualification';
  9    phno1 number(7):='&phno1';
 10     phno2 number(7):='&phno2';
 11    phno3 number(7):='&phno3';
 12    begin
 13
edetails(eno,ename,address(street_address,city,state,zipcode),qualif
ication,emp_phone(phno1,phno2,phno3));
 14   end;
 15  /
Enter value for eno: 1
old   2:  eno emp_obj.eno%type:='&eno';
new   2:  eno emp_obj.eno%type:='1';
Enter value for ename: raju
old   3:   ename emp_obj.ename%type:='&ename';
new   3:   ename emp_obj.ename%type:='raju';
Enter value for street_address: S.N.Puram
old   4:   street_address varchar2(35):='&street_address';
new   4:   street_address varchar2(35):='S.N.Puram';
Enter value for city: guntur
old   5:   city varchar2(13):='&city';
new   5:   city varchar2(13):='guntur';
Enter value for state: A.P
old   6:   state varchar2(20):='&state';
new   6:   state varchar2(20):='A.P';
```

```
Enter value for zipcode: 5200
old   7:   zipcode varchar2(20):='&zipcode';
new   7:   zipcode varchar2(20):='5200';
Enter value for qualification: 1
old   8:   qualification emp_obj.qualification%type:='&qualification';
new   8:   qualification emp_obj.qualification%type:='1';
Enter value for phno1: 11111
old   9:   phno1 number(7):='&phno1';
new   9:   phno1 number(7):='11111';
Enter value for phno2: 22222
old  10:   phno2 number(7):='&phno2';
new  10:   phno2 number(7):='22222';
Enter value for phno3: 333333
old  11:  phno3 number(7):='&phno3';
new  11:  phno3 number(7):='333333';
```

Output:

```
PL/SQL procedure successfully completed.

SQL>select * from emp_obj;
```

Output:

ENO	ENAME	ADDRESS1(STREET_ADDRESS, CITY, STATE, ZIPCODE)	QUALIFICATION	PH_NO
1	raju	ADDRESS('S.N.Puram', 'guntur', 'A.P', '5200')	1	EMP_PHONE (11111, 22222, 333333)

VIVA VOCE

1. What is database?

A database is a collection of data in an organized manner.

2. What is DBMS?

DBMS is a software that is used to perform operations on a database. These operations may include reading, writing, modifying of data and even provide control over accessing of data when multiple users were accessing the data at the same time or even at different times.

3. Advantages of DBMS?

- ✓ Redundancy is controlled.
- ✓ Unauthorized access is restricted.
- ✓ Providing multiple user interfaces.
- ✓ Enforcing integrity constraints.
- ✓ Providing backup and recovery.

4. Disadvantage in File Processing System?

- ✓ Data redundancy & inconsistency.
- ✓ Difficult in accessing data.
- ✓ Data isolation.
- ✓ Data integrity.
- ✓ Concurrent access is not possible.
- ✓ Security Problems.

5. Describe the three levels of data abstraction?

Three levels of abstraction:

- Physical level: The lowest level of abstraction describes how data are stored.
- Logical level: The next higher level of abstraction, describes what data are stored in database and what relationship among those data.
- View level: The highest level of abstraction describes only part of entire database.

6. Define the "integrity rules"

There are two Integrity rules.

- Entity Integrity: States that? Primary key cannot have NULL value?
- Referential Integrity: States that? Foreign Key can be either a NULL value or should be Primary Key value of other relation.

7. What is Data Independence?

Data independence means, the ability to modify the schema definition in one level should not affect the schema definition in the next higher level.

Two types of Data Independence:

- Physical Data Independence: Modification in physical level should not affect the logical level.
- Logical Data Independence: Modification in logical level should affect the view level.

NOTE: Logical Data Independence is more difficult to achieve

8. What is a view? How it is related to data independence?

A view may be thought of as a virtual table, that is, a table that does not really exist in its own right but is instead derived from one or more underlying base tables. Growth and restructuring of base tables is not reflected in views. Thus the view can insulate users from the effects of restructuring and growth in the database. Hence accounts for logical data independence.

9. What is Data Model?

A collection of conceptual tools for describing data, data relationships, data semantics and constraints is called Data Model.

10. What is E-R model?

This data model is based on real world that consists of basic objects called entities and of relationship among these objects. Entities are described in a database by a set of attributes.

11. What is Object Oriented model?

This model is based on collection of objects. An object contains values stored in instance variables within the object. An object also contains bodies of code that operate on the object. These bodies of code are called methods. Objects that contain same types of values and the same methods are grouped together into classes.

12. What is an Entity?

It is a 'thing' in the real world with an independent existence.

13. What is an Entity type?

It is a collection (set) of entities that have same attributes.

14. What is an Entity set?

It is a collection of all entities of particular entity type in the database.

15. What is Weak Entity set?

An entity set may not have sufficient attributes to form a primary key, and its primary key compromises of its partial key and primary key of its parent entity, then it is said to be Weak Entity set.

16. What is an attribute?

It is a particular property, which describes the entity.

17. What is a Relation Schema and a Relation?

A relation Schema denoted by R (A1, A2,…?, An) is made up of the relation name R and the list of attributes Ai that it contains. A relation is defined as a set of tuples. Let r be the relation which contains set tuples (t1, t2, t3... tn). Each tuple is an ordered list of n-values t= (v1, v2... vn).

18. What is degree of a Relation?

It is the number of attributes of its relation schema.

19. What is Relationship?

It is an association among two or more entities.

20. What is Relationship set?

The collection (or set) of similar relationships.

21. What is Relationship type?

Relationship type defines a set of associations or a relationship set among a given set of entity types.

22. What is DML Compiler?

It translates DML statements in a query language into low-level instruction that the query evaluation engine can understand.

23. What is Query evaluation engine?

It executes low-level instruction generated by compiler.

24. What is DDL Interpreter?

It interprets DDL statements and record them in tables containing metadata.

25. What is Record-at-a-time?

The Low level or Procedural DML can specify and retrieve each record from a set of records. This retrieve of a record is said to be Record-at-a-time.

26. What is Relational Algebra?

It is procedural query language. It consists of a set of operations that take one or two relations as input and produce a new relation.

27. What is normalization?

It is a process of analysing the given relation schemas based on their Functional Dependencies (FDs) and primary key to achieve the properties

- Minimizing redundancy
- Minimizing insertion, deletion and update anomalies.

28. What is Functional Dependency?

A Functional dependency is denoted by X Y between two sets of attributes X and Y that are subsets of R specifies a constraint on the possible tuple that can form a relation state r of R. The constraint is for any two tuples t1 and t2 in r if t1[X] = t2[X] then they have t1[Y] = t2[Y]. This means the value of X component of a tuple uniquely determines the value of component Y.

29. When is a functional dependency F said to be minimal?

- Every dependency in F has a single attribute for its right hand side.
- We cannot replace any dependency X A in F with a dependency Y A where Y is a proper subset of X and still have a set of dependency that is equivalent to F.
- We cannot remove any dependency from F and still have set of dependency that is equivalent to F.

30. What is Lossless join property?

It guarantees that the spurious tuples generation does not occur with respect to relation schemas after decomposition.

31. What is Fully Functional dependency?

It is based on concept of full functional dependency. A functional dependency X Y is full functional dependency if removal of any attribute A from X means that the dependency does not hold any more.

32. What is 1 NF (Normal Form)?

The domain of attribute must include only atomic (simple, indivisible) values.

33. What is 2NF?

A relation schema R is in 2NF if it is in 1NF and every non-prime attribute A in R is fully functionally dependent on primary key.

34. What is 3NF?

A relation schema R is in 3NF if it is in 2NF and for every FD X A either of the following is true

- X is a Super-key of R.
- A is a prime attribute of R.

In other words, if every non prime attribute is non-transitively dependent on primary key.

35. What is BCNF (Boyce-Codd Normal Form)?

A relation schema R is in BCNF if it is in 3NF and satisfies an additional constraint that for every FD X A, X must be a candidate key.

36. What is 4NF?

A relation schema R is said to be in 4NF if for every multivalued dependency X Y that holds over R, one of following is true

? X is subset or equal to (or) XY = R.

? X is a super key.

37. What is 5NF?

A Relation schema R is said to be 5NF if for every join dependency {R1, R2... Rn} that holds R, one the following is true

? Ri = R for some i.

? The join dependency is implied by the set of FD, over R in which the left side is key of R.

38. What is Domain-Key Normal Form?

A relation is said to be in DKNF if all constraints and dependencies that should hold on the constraint can be enforced by simply enforcing the domain constraint and key constraint on the relation.

39. What is indexing and what are the different kinds of indexing?

Indexing is a technique for determining how quickly specific data can be found.

Types:

- Binary search style indexing
- B-Tree indexing
- Inverted list indexing
- Memory resident table
- Table indexing

40. What is meant by query optimization?

The phase that identifies an efficient execution plan for evaluating a query that has the least estimated cost is referred to as query optimization.

41. What do you mean by atomicity and aggregation?

<u>Atomicity:</u> Either all actions are carried out or none are. Users should not have to worry about the effect of incomplete transactions. DBMS ensures this by undoing the actions of incomplete transactions.

<u>Aggregation:</u> A concept which is used to model a relationship between a collection of entities and relationships. It is used when we need to express a relationship among relationships.

42. What is a checkpoint and when does it occur?

A Checkpoint is like a snapshot of the DBMS state. By taking checkpoints, the DBMS can reduce the amount of work to be done during restart in the event of subsequent crashes.

43. What are the different phases of transaction?

Different phases are Analysis phase Redo Phase Undo phase

44. What do you mean by flat file database?

It is a database in which there are no programs or user access languages. It has no cross-file capabilities but is user-friendly and provides user-interface management.

45. What is "transparent DBMS"?

It is one, which keeps its Physical Structure hidden from user.

46. Brief theory of Network, Hierarchical schemas and their properties

Network schema uses a graph data structure to organize records while a hierarchical schema uses a tree data structure.

47. What is a query?

A query with respect to DBMS relates to user commands that are used to interact with a data base.

48. What do you mean by Correlated sub query?

Sub queries, or nested queries, are used to bring back a set of rows to be used by the parent query. Depending on how the sub query is written, it can be executed once for the parent query or it can be executed once for each row returned by the parent query. If the sub query is executed for each row of the parent, this is called a correlated sub query.

E.g. Select * From CUST Where '10/03/1990' IN (Select ODATE from ORDER Where CUST.CNUM = ORDER.CNUM)

49. What are the primitive operations common to all record management systems?

Addition, deletion and modification.

50. Name the buffer in which all the commands that are typed in are stored

? Edit? Buffer

51. What are the unary operations in Relational Algebra?

PROJECTION and SELECTION.

52. Are the resulting relations of PRODUCT and JOIN operation the same?

No.

PRODUCT: Concatenation of every row in one relation with every row in another.

JOIN: Concatenation of rows from one relation and related rows from another.

53. What is RDBMS KERNEL?

Two important pieces of RDBMS architecture are the kernel, which is the software, and the data dictionary, which consists of the system-level data structures used by the kernel to manage the database

54. Which part of the RDBMS takes care of the data dictionary? How?

Data dictionary is a set of tables and database objects that is stored in a special area of the database and maintained exclusively by the kernel.

55. What is the job of the information stored in data-dictionary?

The information in the data dictionary validates the existence of the objects, provides access to them, and maps the actual physical storage location.

56. How do you communicate with an RDBMS?

You communicate with an RDBMS using Structured Query Language (SQL)

57. Define SQL and state the differences between SQL and other conventional programming Languages

SQL is a nonprocedural language that is designed specifically for data access operations on normalized relational database structures. The primary difference between SQL and other conventional programming languages is that SQL statements specify what data operations should be performed rather than how to perform them.

58. What is an Oracle Instance?

The Oracle system processes, also known as Oracle background processes, provide functions for the user processes.

Oracle database-wide system memory is known as the SGA, the SYSTEM GLOBAL AREA or SHARED GLOBAL AREA.

The combination of the SGA and the Oracle background processes is known as an Oracle instance

59. What are the four Oracle system processes that must always be up and running for the database to be useable

DBWR (Database Writer), LGWR (Log Writer), SMON (System Monitor), and PMON (Process Monitor).

60. What are database files, control files and log files? How many of these files should a database have at least? Why?

Database Files: The database files hold the actual data and are typically the largest in size. The database files are fixed in size and never grow bigger than the size at which they were created. Depending on their sizes, the tables (and other objects) for all the user accounts can go in one database file? But that's not an ideal situation because it does not make the database structure very flexible for controlling access to storage for different users, putting the database on different disk drives, or backing up and restoring just part of the database.

Control Files: The control file records the name of the database, the date and time it was created, the location of the database and redoes logs, and the synchronization information to ensure that all three sets of files are always in step. Every time you add a new database or redo log file to the database, the information is recorded in the control files.

Redo Logs: The redo logs record all changes to the user objects or system objects. If any type of failure occurs, the changes recorded in the redo logs can be used to bring the database to a consistent state without losing any committed transactions. The redo log files are fixed in size and never grow dynamically from the size at which they were created.

61. What is ROWID?

The ROWID is a unique database-wide physical address for every row on every table. Once assigned it never changes until the row is deleted or the table is dropped. The ROWID is used internally in indexes as a quick means of retrieving rows with a particular key value.

62. What is database Trigger?

A database trigger is a PL/SQL block that can be defined to perform certain actions automatically when a condition is encountered. For example: a trigger can be defined to deny any database insertions into a database on a specified system date.

63. Name two utilities that Oracle provides, which are use for backup and recovery.

Along with the RDBMS software, Oracle provides two utilities that you can use to back up and restore the database. These utilities are Export and Import.

The Export utility dumps the definitions and data for the specified part of the database to an operating system binary file. The Import utility reads the file produced by an export, recreates the definitions of objects, and inserts the data

64. What are stored-procedures? And what are the advantages of using them?

Stored procedures are database objects that perform a user defined operation by executing the SQL commands and returning the result to the client Stored procedures are used to reduce network traffic.

65. How are exceptions handled in PL/SQL? Give some of the internal exceptions' names.

The exception handler must be defined within a subprogram specification. When an exception is raised control is transferred to the exception-handler block. After the exception handler executes, control returns to the block in which the handler was defined. If there are no more executable statements in the block, control returns to the caller.

<u>PL/SQL Internal exceptions.</u>

Exception Name	Oracle Error
ZERO_DIVIDE	ORA-01476
CURSOR_ALREADY_OPEN	ORA-06511
DUP_VAL_ON_INDEX	ORA-00001
INVALID_NUMBER	ORA-01722
NO_DATA_FOUND	ORA-01403

In addition to this list of exceptions, there is a catch-all exception named OTHERS that traps all errors for which specific error handling has not been established.

66. Does PL/SQL support "overloading"? Explain

The concept of overloading in PL/SQL relates to the idea that you can define procedures and functions with the same name. PL/SQL does not look only at the referenced name, however, to resolve a procedure or function call. The count and data types of formal parameters are also considered.

67. Tables derived from the ERD

→ Are always in 1NF