Sri
SAI RAM INSTITUTE OF TECHNOLOGY
An Autonomous Institution | Affiliated to Anna University & Approved by AICTE, New Delhi
Accredited by NBA and NAAC "A+" | An ISO 9001:2015 Certified and MHRD NIRF ranked institution
Sai Leo Nagar, West Tambaram, Chennai - 600 044. www.sairamit.edu.in

LABMANUAL

20CSPL402DATABASE MANAGEMENT SYSTEM LABORATORY

IIYEAR/IVSEMESTERBATCH:

2022-2026

ACADEMICYEAR:2023-2024(EVEN)

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

VISION OFTHEINSTITUTE

To be identified as a "Centre of Excellence" with high standards of Knowledge Disseminationand Research opportunities and to transform the students to imbibe qualities of technicalexpertiseofinternationalstandardsandhighlevelsofethicalvalues,whointurnshallcontribute totheadvancementofsocietyandhumankind.

MISSIONOFTHEINSTITUTE

WeshalldedicateandcommitourselvestoattainandmaintainexcellenceinTechnicalEducation through commitment and continuous improvement of infrastructure and equipmentand provide an inspiring environment for Learning, Research and Innovation for our studentstotransformthemintocomplete humanbeings withethicalandsocialvalues.

VISIONOFTHEDEPARTMENT

To be a centre of excellence in educating and graduating Computer Engineers by providingunique environment that foster research, technological, and social enrichment with intellectualknowledge toacquireinternationalstandards..

MISSIONOFTHEDEPARTMENT

M1:DevelophighqualityComputerScienceandEngineeringgraduateswithtechnicalandProfessionalskills byprovidingmoderninfrastructuretoacquireinternationalstandards.

M2:FosterresearchtosolverealworldproblemswithemergingTechnologies

M3:Establishcenterofexcellencesincollaborationwithindustries,tomeetthechangingneeds ofsociety

M4:InculcatespiritofmoralvaluesthatcontributestosocietalethicsPROGRAMMEEDUCATIONALOBJECTIVE

PEO1: Formulate,analyzeandsolveEngineeringproblemswithstrongfoundationinMathematical, Scientific andEngineeringfundamentals.

PEO2: Analyzetherequirements,realizethetechnicalspecificationanddesigntheEngineeringsolutionsbyapplyingcomputerscience theoryandprinciples.

PEO3: Promotecollaborativelearningandteam work spiritthrough multi-disciplinaryprojectsanddiverseprofessionalactivities.

PEO4: Equipthegraduateswithstrongknowledge,competenceandsoftskillsthatallowsthemtocontributeethicallytotheneedsofsociety.

PEO 5 : Accomplish sustainable progress in the emerging areas of Engineering through life-longlearning.

PROGRAMSPECIFICOUTCOMES

PSO1: Demonstratebasicknowledgeofcomputerapplicationsandapplystandardpracticesinsoftwareprojectdevelopment.

PSO2: Understand,AnalyzeandDevelopcomputerprogramsforefficientdesignofcomputer-basedsystems ofvaryingcomplexity

PROGRAMMEOUTCOMES(POS)

PO1. EngineeringKnowledge:Applytheknowledgeofmathematics,science,engineering fundamentals, and an engineering specialization to the solution of complexengineeringproblems.

PO2. Problem Analysis:Identify,formulate,reviewresearchliterature,andanalyzecomplex engineering problems reaching substantiated conclusions using first principles ofmathematics,naturalsciences,andengineeringsciences.

PO3. Design/DevelopmentofSolutions:Designsolutionsforcomplexengineeringproblems and design system components or processes that meet the specified needs withappropriate consideration for the public health and safety, and the cultural, societal, andenvironmentalconsiderations.

PO 4. Conduct Investigations of Complex Problems: Use research-based knowledge andresearch methods including design of experiments, analysis and interpretation of data, andsynthesisoftheinformationtoprovidevalidconclusions.

PO 5. Modern Tool Usage: Create, select, and apply appropriate techniques, resources,andmodern engineeringandIT toolsincludingprediction andmodelingtocomplexengineeringactivitieswithanunderstandingofthelimitations.

PO 6. The Engineer and Society: Apply reasoning informed by the contextual knowledgetoassesssocietal,health,safety,legalandculturalissuesandtheconsequentresponsibilitie srelevanttothe professionalengineeringpractice.

PO7. EnvironmentandSustainability:Understandtheimpactoftheprofessionalengineeringsolution sinsocietalandenvironmentalcontexts,anddemonstratetheknowledge of,andneedforsustainable development.

PO8. Ethics:Applyethicalprinciplesandcommittoprofessionalethicsandresponsibilitiesandnorms oftheengineeringpractice.

PO 9. Individual and Team Work: Function effectively as an individual, and as a memberorleaderindiverse teams,andinmultidisciplinarysettings.

PO 10. Communication: Communicate effectively on complex engineering activities withthe engineering community and with society at large, such as, being able to comprehendand write effective reports and design documentation, make effective presentations, andgive andreceive clearinstructions.

PO 11. Project Management and Finance: Demonstrate knowledge and understanding ofthe engineering and management principles and apply these to one's own work, as amemberandleaderinateam,tomanageprojectsandinmultidisciplinaryenvironments.

PO12. Life-longLearning:Recognizetheneedfor, andhavethepreparationandabilityto engage in independent and life-long learning in the broadest context of technologicalchange

LIST OF EXPERIMENTS

1. Creation of a database and writing SQL queries to retrieve information from thedatabase.
   Data Definition Language(DDL).
   a. CREATE
   b. ALTER
   c. DROP
   d. TRUNCATE
   e.RENAME
   f. COMMENT
   Data Manipulation Language(DML)
   a. INSERT
   b. UPDATE
   c. DELETE
   d. SELECT
   Transaction ControlLanguage(TCL)
   a. COMMIT
   b. ROLLBACK
   c. SAVEPOINT
2. Database Querying -Simple queries, Nested queries, Sub queries andJoins
   Implementation of simplequeries
   Implementation of Nested Queries / Subqueries
   Implementation the JoinOperations
3. Views, Sequences,Synonyms
   Implementation ofViews.
   Implementation ofSequences
   Implementation ofSynonyms
4. Study of PL/SQLblock.
5. Implicit and ExplicitCursors
   ImplicitCursor
   ExplicitCursor
6. Procedures
   Implementation of Procedures and its application
7. Creation of databasetriggers
8. Creation of databasefunctions
9. Write a PL/SQL block that handles all types ofexceptions.
10. Database Design using ER modeling, normalization andImplementation
11. Database Connectivity with Front EndTools
12. MiniProject
    a Inventory ControlSystem.
    b Material RequirementProcessing.
    c Hospital ManagementSystem.
    d Railway Reservation system.
    e Web Based User Identification System.
    f Hotel Management System

Creation of a database and writing SQL queries to retrieve information from the database.

Ex:No:01(1.1) **DATA DEFINITION LANGUAGE(DDL)**

__:__: __

AIM:

To execute the various Data Definition Language commands in RDBMS.

OBJECTIVE:

After completing the exercise the students can able to Understand how to create a table with list of fields, Modify a row using where clause, Drop a table, Delete the unwanted rows in a table.

DATA DEFINITION LANGUAGE

It is used to communicate with database. DDL is used to:

Create an object
Alter the structure of an object

To drop the object created.

ALGORITHM:

Step 1: Start the program

Step 2: Go to SQL.

Step 3: Enter the user name andpassword.

Step 4: Connect to thedatabase.

Step 5: Type the commands for creating tables and perform various operations on thetables.

Step 6: The output isdisplayed.

Step 7: Stop the program

DDL COMMAND:

CREATE

ALTER

DROP

TRUNCATE

COMMENT

RENAME

QUERY: 01

Q1: Write a query to create a table employee with empno, ename, designation, and salary.

Syntax: It is used to create a table

SQL: CREATE <OBJ.TYPE><OBJ.NAME> (COLUMN NAME.1 <DATATYPE> (SIZE),

COLUMN NAME.2 <DATATYPE> (SIZE) ……);

Command:

SQL>**CREATE TABLE** EMP (EMPNO **NUMBER (4),**ENAME **VARCHAR2 (10),**
DESIGNATIN **VARCHAR2 (10),**SALARY **NUMBER (8, 2));**

Table created.

Constraints with Table Creation:

Constraints are condition for the data item to be stored into a database. There are two types of Constraints viz., Column Constraints and Table Constraints.

Syntax:

[CONSTRAINT constraint name]

{[NOT] NULL / UNIQUE / PRIMARY

KEY}(Column[,column]..) FOREIGN KEY (column [, colum]…)

REFERENCES table

[ON DELETE CASCADE]

[CHECK (condition)]

TABLE DESCRIPTION

It is used to view the table structure to confirm whether the table was created correctly.

QUERY: 02

Q2: Write a query to display the column name and data type of the table employee.

Syntax: This is used to view the structure of the table.

SQL: DESC <TABLE NAME>;

Command:

SQL> DESC EMP;

Name Null?	**Type**
----------	**----------**
EMPNO	NUMBER(4)
ENAME	VARCHAR2(10)
DESIGNATIN	VARCHAR2(10)

SALARY NUMBER(8,2)

QUERY: 03

Q3: Write a query for create a from an existing table with all the fields

Syntax: syntax for create a table from an existing table with all fields.

SQL> CREATE TABLE <TRAGET TABLE NAME> SELECT * FROM<SOURCE TABLE NAME>;

Command:

SQL> CREATE TABLE **EMP1** AS SELECT * FROM **EMP**;

Table created.

Command:

SQL> DESC EMP1

NameNull?	Type
EMPNO	NUMBER(4)
ENAME	VARCHAR2(10)
DESIGNATIN	VARCHAR2(10)
SALARY	NUMBER(8,2)

QUERY: 04

Q4: Write a query for create a from an existing table with selected fields

Syntax: Syntax for create a from an existing table with selected fields.

SQL> CREATE TABLE <TRAGET TABLE NAME> AS SELECT EMPNO, ENAMEFROM <SOURCE TABLE NAME>;

Command:

SQL> CREATE TABLE **EMP2** AS SELECT **EMPNO**, **ENAME** FROM **EMP**;

Table created.

Command:

SQL> DESC EMP2

NameNull?	Type
EMPNO	NUMBER (4)
ENAME	VARCHAR2(10)

QUERY: 05

Q5: Write a query for create a new table from an existing table without any record:

Syntax: The syntax for create a new table from an existing table without any record.

SQL> CREATE TABLE <TRAGET TABLE NAME> AS SELECT * FROM<SOURCE TABLE NAME>

WHERE <FALSE CONDITION>;

Command:

SQL> CREATE TABLE EMP3 AS SELECT * FROM EMP WHERE1>2;

Table created.

Command:

SQL> DESC EMP3;

NameNull?	Type
EMPNO	NUMBER(4)
ENAME	VARCHAR2(10)
DESIGNATIN	VARCHAR2(10)
SALARY	NUMBER(8,2);

ALTER & MODIFICATION ON TABLE

To modify structure of an already existing table to add one more columns and also modify the existing columns.

Alter command is used to:

1. Add a newcolumn.
2. Modify the existing columndefinition.
3. To include or drop integrityconstraint.

QUERY: 06

Q6: Write a Query to Alter the column EMPNO NUMBER (4) TO EMPNO NUMBER (6).
Syntax: The syntax for alter & modify on a single column.
SQL > ALTER <TABLE NAME> MODIFY <COLUMN NAME><DATATYPE>(SIZE);

Command:

SQL>ALTER TABLE **EMP** MODIFY **EMPNO** NUMBER (6);
Table altered.

Command:

SQL> DESC EMP;

NameNull?	Type
EMPNO	NUMBER(6)
ENAME	VARCHAR2(10)
DESIGNATIN	VARCHAR2(10)
SALARY	NUMBER(8,2)

QUERY: 07

Q7. Write a Query to Alter the table employee with multiple columns (EMPNO,ENAME.)

Syntax: To alter table with multiple column.

SQL > ALTER <TABLE NAME> MODIFY <COLUMN NAME1><DATATYPE>(SIZE),

MODIFY<COLUMNNAME2><DATATYPE>(SIZE)..... ;

Command:

SQL>ALTER TABLE EMP MODIFY (EMPNO NUMBER (7), ENAMEVARCHAR2(12)); Tablealtered.

Command:

SQL> DESC EMP;

NameNull?	Type
EMPNO	NUMBER(7)
ENAME	VARCHAR2(12)
DESIGNATIN	VARCHAR2(10)
SALARY	NUMBER(8,2);

QUERY: 08

Q8. Write a query to add a new column in to employee

Syntax: To add a new column.

SQL> ALTER TABLE <TABLE NAME> ADD (<COLUMN NAME><DATATYPE><SIZE>);

Command:

SQL> ALTER TABLE EMP ADD QUALIFICATION VARCHAR2(6); Table altered.

SQL> DESC EMP;

NameNull?	Type
EMPNO	NUMBER(7)
ENAME	VARCHAR2(12)
DESIGNATIN	VARCHAR2(10)
SALARY	NUMBER(8,2)
QUALIFICATION	VARCHAR2(6)

QUERY: 09

Q9: Write a query to add multiple columns in to employee **Syntax:** Syntax for add a new column.

SQL> ALTER TABLE <TABLE NAME> ADD (<COLUMN NAME1><DATATYPE><SIZE>, (<COLUMN NAME2><DATA TYPE><SIZE>…);

Command:

SQL>ALTER TABLE EMP ADD (DOB DATE, DOJ DATE); Table altered.

SQL> DESC EMP;

NameNull?	Type
EMPNO	NUMBER(7)
ENAME	VARCHAR2(12)
DESIGNATIN	VARCHAR2(10)
SALARY	NUMBER(8,2)
QUALIFICATION	VARCHAR2(6)
DOB	DATE
DOJ	DATE

REMOVE / DROP

It will delete the table structure provided the table should be empty.

QUERY: 10

Q10. Write a query to drop a column from an existing table employee

Syntax: syntax for add a new column.

SQL> ALTER TABLE <TABLE NAME> DROP COLUMN <COLUMN NAME>;

Command:

SQL> ALTER TABLE EMP DROP COLUMN DOJ;
Table altered.

SQL> DESC EMP;

NameNull?	Type
EMPNO	NUMBER(7)
ENAME	VARCHAR2(12)
DESIGNATIN	VARCHAR2(10)
SALARY	NUMBER(8,2)
QUALIFICATION	VARCHAR2(6)
DOB	DATE

QUERY: 11

Q10. Write a query to drop multiple columns from employee

Syntax: The Syntax for add a new column.
SQL> ALTER TABLE <TABLE NAME> DROP<COLUMNNAME1>,<COLUMNNAME2>,… .....;

Command:

SQL> ALTER TABLE EMP DROP (DOB, QUALIFICATION); Table altered.

SQL> DESC EMP;

NameNull?	Type
EMPNO	NUMBER(7)
ENAME	VARCHAR2(12)
DESIGNATIN	VARCHAR2(10)
SALARY	NUMBER(8,2)

RENAME

QUERY: 12

Q10. Write a query to rename table emp to employee

Syntax: The Syntax for add a new column.

SQL> ALTER TABLE RENAME <OLD NAME> TO <NEW NAME>

Command:

SQL> ALTER TABLE RENAME EMP TO EMPLOYEE;

SQL> DESC EMPLOYEE;

NameNull?	Type
EMPNO	NUMBER(7)
ENAME	VARCHAR2(12)
DESIGNATIN	VARCHAR2(10)
SALARY	NUMBER(8,2)

TRUNCATE TABLE

If there is no further use of records stored in a table and the structure has to be retained then the records alone can be deleted.

Syntax:

TRUNCATE TABLE <TABLE NAME>;

Example:

Truncate table EMP;

DROP:

To remove a table along with its structure and data.

Syntax: The Syntax for add a new column.

SQL> Drop table<table name>;

Command:

SQL> drop table employee;

RESULT:

Thus the SQL commands for DDL commands in RDBMS has been verified and executed successfully.

DATA MANIPULATION LANGUAGE (DML) COMMANDS IN RDBMS

Ex: No:1.2

__:__: __

AIM:

To execute and verify the DML commands are the most frequently used SQL commands and is used to query and manipulate the existing databaseobjects.

DML (DATA MANIPULATION LANGUAGE)

SELECT

INSERT

DELETE

UPDATE

ALGORITHM:

STEP 1: Start the DBMS.

STEP 2: Create the table with its essential attributes.

STEP 3: Insert the record into table

STEP 4: Update the existing records into the table

STEP 5: Delete the records in to the table

STEP 6: use save point if any changes occur in any portion of the record to undo its original state.

STEP 7: use rollback for completely undo therecords

STEP 8: use commit for permanently save therecords

INSERT

The SQL INSERT INTO Statement is used to add new rows of data to a table in the database.

Insert a record from an existing table:

QUERY: 01

Q1. Write a query to insert the records in to employee.

Syntax: syntax for insert records in to a table

SQL :> INSERT INTO <TABLE NAME> VALUES< VAL1, 'VAL2',…..);

Command:

SQL>INSERT INTO EMP VALUES (101,'NAGARAJAN','LECTURER',15000);

1 row created.

Insert A Record Using Substitution Method:

QUERY: 03

Q3. Write a query to insert the records in to employee using substitution method.

Syntax: syntax for insert records into the table.

SQL :> INSERT INTO <TABLE NAME> VALUES< '&column name', '&column name 2', ...);

Command:

SQL> INSERT INTO EMP
VALUES(&EMPNO,'&ENAME','&DESIGNATIN','&SALARY'); Enter value for empno:102

Enter value for ename: SARAVANAN
Enter value for designatin: LECTURER
Enter value for salary: 15000
1 row created.

old 1: INSERT INTO EMP VALUES(&EMPNO,'&ENAME','&DESIGNATIN','&SALARY')
new 1: INSERT INTO EMP VALUES(102,'SARAVANAN','LECTURER','15000')
SQL> /

Enter value for empno: 103

Enter value for ename: PANNERSELVAM

Enter value for designatin: ASST. PROF

Enter value for salary: 20000

1 row created.

old 1: INSERT INTO EMP VALUES(&EMPNO,'&ENAME','&DESIGNATIN','&SALARY')

new 1: INSERT INTO EMP VALUES(103,'PANNERSELVAM','ASST.PROF','20000')
SQL> /

Enter value for empno: 104

Enter value for ename: CHINNI

Enter value for designatin: HOD,

PROF Enter value for salary: 45000

1 row created.

old 1: INSERT INTO EMP VALUES(&EMPNO,'&ENAME','&DESIGNATIN','&SALARY')

new 1: INSERT INTO EMP VALUES(104,'CHINNI','HOD, PROF','45000')

SQL> SELECT * FROM EMP;

EMPNO	ENAME	DESIGNATIN	SALARY
101	NAGARAJAN	LECTURER	15000
102	SARAVANAN	LECTURER	15000
103	PANNERSELVAM	ASST. PROF	20000
104	CHINNI	HOD, PROF	45000

SELECT

SELECT Statement is used to fetch the data from a database table which returns data in the form of result table. These result tables are calledresult-sets.

Display the EMP table:

QUERY: 02

Q3. Write a query to display the records from employee.

Syntax: Syntax for select Records from the table.

SQL> SELECT * FROM <TABLE NAME>;

Command:

SQL> SELECT * FROM EMP;

EMPNO	ENAME	DESIGNATIN	SALARY
101	NAGARAJAN	LECTURER	15000

UPDATE

The SQL **UPDATE** Query is used to modify the existing records in a table. You can use **WHERE** clause with **UPDATE** query to update selected rows, otherwise all the rows would be affected.

QUERY: 04

Q1. Write a query to update the records from employee.

Syntax: syntax for update records from the table.

SQL> UPDATE <<TABLE NAME> SET <COLUMNANE>=<VALUE> WHERE <COLUMN NAME=<VALUE>;

Command:

SQL> UPDATE EMP SET SALARY=16000 WHERE EMPNO=101;
1 row updated.

SQL> SELECT * FROM EMP;

EMPNO	ENAME	DESIGNATIN	SALARY
101	NAGARAJAN	LECTURER	16000
102	SARAVANAN	LECTURER	15000
103	PANNERSELVAM	ASST. PROF	20000
104	CHINNI	HOD,PROF	45000

Update Multiple Columns:

QUERY: 05

Q5. Write a query to update multiple records from employee.

Syntax: syntax for update multiple records from the table.

SQL> UPDATE <<TABLE NAME> SET <COLUMNANE>=<VALUE> WHERE <COLUMN NAME=<VALUE>;

Command:

SQL>UPDATE EMP SET SALARY = 16000, DESIGNATIN='ASST. PROF' WHERE EMPNO=102;

1 row updated.

SQL> SELECT * FROM EMP;

EMPNO	ENAME	DESIGNATIN	SALARY
101	NAGARAJAN	LECTURER	16000
102	SARAVANAN	ASST. PROF	16000
103	PANNERSELVAM	ASST. PROF	20000
104	CHINNI	HOD, PROF	45000

DELETE
The **SQL DELETE** Query is used to delete the existing records from a table. You can use **WHERE** clause with **DELETE** query to delete selected rows, otherwise all the records would be deleted.

QUERY: 06

Q5. Write a query to delete records from employee.
Syntax: Syntax for delete Records from the table:
SQL> DELETE <TABLE NAME> WHERE <COLUMN NAME>=<VALUE>;
Command:
SQL> DELETE EMP WHERE EMPNO=103;
1 row deleted.
SQL> SELECT * FROM EMP;

EMPNO	ENAME	DESIGNATIN	SALARY
101	NAGARAJAN	LECTURER	16000
102	SARAVANAN	ASST. PROF	16000
104	CHINNI	HOD, PROF	45000

RESULT:

Thus the SQL commands for DML has been verified and executed successfully.

TRANSACTION CONTROL LANGUAGE (TCL) COMMANDS IN RDBMS

Ex: No:1.3

__:__: __

COMMIT command

COMMIT command is used to permanently save any transaction into the database.

Following is commit command's syntax,

COMMIT;

ROLLBACK command

This command restores the database to last commited state. It is also used with SAVEPOINT command to jump to a savepoint in an ongoing transaction.

Following is rollback command's syntax,

ROLLBACK TO savepoint_name;

SAVEPOINT command

SAVEPOINT command is used to temporarily save a transaction so that you can rollback to that point whenever required.

Following is savepoint command's syntax,

SAVEPOINT savepoint_name;

Example:

```
INSERT INTO class VALUES(5, 'Rahul');

COMMIT;

UPDATE class SET name = 'Abhijit' WHERE id = '5';

SAVEPOINTA;

INSERT INTO class VALUES(6, 'Chris');

SAVEPOINTB;

INSERT INTO class VALUES(7, 'Bravo');

SAVEPOINTC;

SELECT * FROM class;
```

The resultant table will look like,

id	name
1	Abhi
2	Adam
4	Alex
5	Abhijit
6	Chris
7	Bravo

```
ROLLBACK TO B;

SELECT * FROM class;
```

id	name
1	Abhi
2	Adam
4	Alex
5	Abhijit
6	Chris

```
ROLLBACK TO A;

SELECT * FROM class;
```

id	name
1	Abhi
2	Adam
4	Alex
5	Abhijit

RESULT:

Thus the SQL commands for TCL has been verified and executed successfully.

Database Querying -Simple queries, Nested queries, Sub queries and Joins

Ex: No: 02(2.1) Implementation of Simple Queries

__:__: __

AIM:

To performing insertion, deletion, modifying, altering, updating and viewing records based onconditions.

ALGORITHM:

STEP 1: Start the DBMS
STEP 2: Connect to the database (DB)
STEP 3: Create the table with its essential attributes.
STEP 4: Insert the record into table based on some condition using WHERE CLAUSE
STEP 5: Update the existing records into the table based on some condition
STEP 6: Delete the records in to the table based on some condition
STEP 7: Use commit for permanently save the records
STEP 8: Stop the program

DRL-DATA RETRIEVAL IMPLEMENTING ON SELECT COMMANDS

Command:

```
SQL> CREATE TABLE EMP(
                        EMPNO      NUMBER (4),
                        ENAME      VARCHAR2 (10),
                        JOB        VARCHAR2(20),
                        MGR        NUMBER(4),
                        HIREDATE   DATE,
                        SAL        NUMBER(8,2),
                        DEPTNO     NUMBER(3)
                    );
```

Table created.

SQL> INSERT INTO EMP VALUES(7369,'SMITH','CLERK',5001,'17-DEC-80','8000',200); 1 row created.

SQL> INSERT INTO EMP VALUES(7499,'ALLEN','SALESMAN',5002,'20-FEB-80','3000',300); 1 row created.

SQL> INSERT INTO EMP VALUES(7521,'WARD','SALESMAN',5003,'22-FEB-80','5000',500); 1 row created.

SQL> INSERT INTO EMP VALUES(7566,'JONES','MANAGER',5002,'02-APR-85','75000',200);
1 row created.

SQL> INSERT INTO EMP VALUES(7566,'RAJA','OWNER',5000,'30-APR-75',NULL,100);
1 row created.

SQL> INSERT INTO EMP VALUES(7566,'KUMAR','COE',5002,'12-JAN-87','55000',300);
1 row created.

SQL> INSERT INTO EMP VALUES(7499,'RAM KUMAR','SR.SALESMAN',5003,'22-JAN-87','12000.55',200);
1 row created.

SQL> INSERT INTO EMP VALUES(7521,'SAM KUMAR','SR.SALESMAN',5003,'22-JAN-75','22000',300); 1 row created.

THE SELECT STATEMENT SYNTAX WITH ADDITIONAL CLAUSES:

Select [Distinct / Unique] (*columnname [As alias}, ….]
From tablename
[where condition]
[Group BY group _by_expression]
[Having group_condition]
[ORDER BY {*col(s)*|*expr*|*numeric_pos*} [ASC|DESC] [NULLS FIRST|LAST]];

SQL> SELECT * FROM EMP;

EMPNO	ENAME	JOB	MGR	HIREDATE	SAL	DEPTNO
7369	SMITH	CLERK	5001	17-DEC-80	8000	200
7499	ALLEN	SALESMAN	5002	20-FEB-80	3000	300
7521	WARD	SALESMAN	5003	22-FEB-80	5000	500
7566	JONES	MANAGER	5002	02-APR-85	75000	200
7566	RAJA	OWNER	5000	30-APR-75		100
7566	KUMAR	COE	5002	12-JAN-87	55000	300
7499	RAM KUMAR	SR.SALESMAN	5003	22-JAN-87	12000.55	200
7521	SAM KUMAR	SR.SALESMAN	5003	22-JAN-75	22000	300

8 rows selected.

BY USING SELECTED COLUNMS

SQL> SELECT EMPNO,ENAME,JOB,SAL FROM EMP;

EMPNO	ENAME	JOB	SAL
7369	SMITH	CLERK	8000
7499	ALLEN	SALESMAN	3000
7521	WARD	SALESMAN	5000
7566	JONES	MANAGER	75000
7566	RAJA	OWNER	
7566	KUMAR	COE	55000
7499	RAM KUMAR	SR.SALESMAN	12000.55
7521	SAM KUMAR	SR.SALESMAN	22000

8 rows selected.

SQL> SELECT EMPNO,ENAME,JOB,SAL FROM EMP WHERE SAL=5000;

EMPNO	ENAME	JOB	SAL
7521	WARD	SALESMAN	5000

BY USING BETWEEN / NOT / IN / NULL / LIKE

BETWEEN Syntax:

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

SQL> SELECT EMPNO,ENAME,JOB,SAL FROM EMP WHERE SAL **BETWEEN** 10000 AND 30000;

EMPNO	ENAME	JOB	SAL
7499	RAM KUMAR	SR.SALESMAN	12000.55
7521	SAM KUMAR	SR.SALESMAN	22000

SQL> SELECT EMPNO,ENAME,JOB,SAL FROM EMP WHERE SAL **NOT BETWEEN** 10000 **AND** 30000;

EMPNO	ENAME	JOB	SAL
7369	SMITH	CLERK	8000
7499	ALLEN	SALESMAN	3000
7521	WARD	SALESMAN	5000
7566	JONES	MANAGER	75000
7566	KUMAR	COE	55000

IN Syntax

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

SQL> SELECT EMPNO,ENAME,JOB,SAL FROM EMP WHERE SAL **IN** (1000.5,75000);

EMPNO	ENAME	JOB	SAL
7566	JONES	MANAGER	75000

SQL> SELECT EMPNO,ENAME,JOB,SAL FROM EMP WHERE SAL **NOT IN** (1000.5,75000);

EMPNO	ENAME	JOB	SAL
7369	SMITH	CLERK	8000
7499	ALLEN	SALESMAN	3000
7521	WARD	SALESMAN	5000
7566	KUMAR	COE	55000
7499	RAM KUMAR	SR.SALESMAN	12000.55
7521	SAM KUMAR	SR.SALESMAN	22000

6 rows selected.

SQL> SELECT EMPNO,ENAME,JOB,SAL FROM EMP WHERE SAL **IS NULL**;

EMPNO	ENAME	JOB	SAL
7566	RAJA	OWNER	

SQL> SELECT EMPNO,ENAME,JOB,SAL FROM EMP WHERE SAL **IS NOT NULL**;

EMPNO	ENAME	JOB	SAL

7369 SMITH CLERK 8000
7499 ALLEN SALESMAN 3000
7521 WARD SALESMAN 5000
7566 JONES MANAGER 75000
7566 KUMAR COE 55000
7499 RAM KUMAR SR.SALESMAN 12000.55
7521 SAM KUMAR SR.SALESMAN 22000
7 rowsselected.

LIKESyntax:

SELECT *column_name(s)*
FROM *table_name*
WHERE *column_name* LIKE *pattern*;

SQL> SELECT EMPNO,ENAME,JOB,SAL FROM EMP WHERE SAL **LIKE** 55000;

EMPNC	ENAME	JOB	SAL
7566	KUMAR	COE	55000

SQL> SELECT EMPNO,ENAME,JOB,SAL FROM EMP WHERE ENAME **LIKE 'S%';**

EMPNC	ENAME	JOB	SAL
7369	SMITH	CLERK	8000
7521	SAMKUMARSR.SALESMAN		22000

SQL> SELECT EMPNO,ENAME,JOB,SAL FROM EMP WHERE ENAME **LIKE '%R';**

EMPNC	ENAME	JOB	SAL
7566	KUMAR	COE	55000
7499	RAM KUMAR SR.SALESMAN		12000.55
7521	SAMKUMARSR.SALESMAN		22000

SQL> SELECT EMPNO,ENAME,JOB,SAL FROM EMP WHERE ENAME **LIKE '%U%';**

EMPNC	ENAME	JOB	SAL
7566	KUMAR	COE	55000
7499	RAM KUMAR	SR.SALESMAN	12000.55
7521	SAM KUMAR	SR.SALESMAN	22000

SQL> SELECT EMPNO,ENAME,JOB,SAL FROM EMP WHERE ENAME **LIKE '%A%';**

EMPNC	ENAME	JOB	SAL
7499	ALLEN	SALESMAN	3000
7521	WARD	SALESMAN	5000
7566	RAJA	OWNER	
7566	KUMAR	COE	55000
7499	RAM KUMAR	SR.SALESMAN	12000.55
7521	SAM KUMAR	SR.SALESMAN	22000

6 rowsselected.

SQL> SELECT EMPNO,ENAME,JOB,SAL FROM EMP WHERE ENAME **LIKE '%LL%';**

EMPNO	ENAME	JOB	SAL
7499	ALLEN	SALESMAN	3000

SQL> SELECT EMPNO,ENAME,JOB,SAL FROM EMP WHERE ENAME **LIKE '%E%';**

EMPNO	ENAME	JOB	SAL
7499	ALLEN	SALESMAN	3000

7566 JONES MANAGER 75000

SQL> SELECT EMPNO,ENAME,JOB,SAL FROM EMP WHERE ENAME **LIKE '%U%A%';**

EMPNO	ENAME	JOB	SAL
7566	KUMAR	COE	55000
7499	RAM KUMAR	SR.SALESMAN	12000.55
7521	SAM KUMAR	SR.SALESMAN	22000

SQL> SELECT EMPNO,ENAME,JOB,SAL FROM EMP WHERE ENAME**LIKE'R____';//3_**

EMPNO	ENAME	JOB	SAL
7566	RAJA	OWNER	

SQL> SELECT EMPNO,ENAME,JOB,SAL FROM EMP WHERE ENAME **LIKE** '**R_J_';**

EMPNO	ENAME	JOB	SAL
7566	RAJAOWNER		

SQL> SELECT EMPNO,ENAME,JOB,SAL FROM EMP WHERE ENAME **LIKE** '**_M%';**

EMPNO	ENAME	JOB	SAL
7369	SMITH	CLERK	8000

SQL> SELECT EMPNO,ENAME,JOB,SAL FROM EMP WHERE ENAME **LIKE** '**_M';**

no rows selected

SQL> SELECT EMPNO,ENAME,JOB,SAL FROM EMP WHERE ENAME**LIKE'_____R'; // 4_**

EMPNO	ENAME	JOB	SAL
7566	KUMAR	COE	55000

SQL> SELECT EMPNO,ENAME,JOB,SAL FROM EMP WHERE ENAME**LIKE'K____R'; // 3_**

EMPNO	ENAME	JOB	SAL
7566	KUMAR	COE	55000

SQL> SELECT EMPNO,ENAME,JOB,SAL FROM EMP WHERE ENAME NOT **LIKE** '**R_J_';**

EMPNO	ENAME	JOB	SAL
7369	SMITH	CLERK	8000
7499	ALLEN	SALESMAN	3000
7521	WARD	SALESMAN	5000
7566	JONES	MANAGER	75000
7566	KUMAR	COE	55000
7499	RAM KUMAR	SR.SALESMAN	12000.55
7521	SAM KUMAR	SR.SALESMAN	22000

7 rowsselected.

RELATIONAL OPERATOR

```
SQL> SELECT EMPNO,ENAME,JOB,SAL FROM EMP WHERE SAL=55000;
EMPNO      ENAME      JOB                  SAL
---------- ---------- -------------------- ----------
7566       KUMAR      COE                  55000

SQL> SELECT EMPNO,ENAME,JOB,SAL FROM EMP WHERE SAL!=55000;
EMPNO      ENAME      JOB                  SAL
---------- ---------- -------------------- ----------
7369       SMITH      CLERK                8000
7499       ALLEN      SALESMAN             3000
7521       WARD       SALESMAN             5000
7566       JONES      MANAGER              75000
7499       RAM KUMAR  SR.SALESMAN          12000.55
7521       SAM KUMAR  SR.SALESMAN          22000
6 rows selected.

SQL> SELECT EMPNO,ENAME,JOB,SAL FROM EMP WHERE SAL<>55000;
EMPNO ENAMEJOB                  SAL
---------- ---------- -------------------- ----------
7369 SMITH     CLERK                8000
7499 ALLEN     SALESMAN             3000
7521 WARD      SALESMAN             5000
7566 JONES     MANAGER              75000
7499 RAM KUMARSR.SALESMAN           12000.55
7521 SAM KUMARSR.SALESMAN           22000
6 rows selected.

SQL> SELECT EMPNO,ENAME,JOB,SAL FROM EMP WHERE SAL>55000;
EMPNO      ENAME      JOB                  SAL
---------- ---------- -------------------- ----------
7566       JONES      MANAGER              75000

SQL> SELECT EMPNO,ENAME,JOB,SAL FROM EMP WHERE SAL<55000;
EMPNO      ENAME      JOB                  SAL
---------- ---------- -------------------- ----------
7369       SMITH      CLERK                8000
7499       ALLEN      SALESMAN             3000
7521       WARD       SALESMAN             5000
7499       RAM KUMARSR.SALESMAN            12000.55
7521       SAM KUMARSR.SALESMAN            22000

SQL> SELECT EMPNO,ENAME,JOB,SAL FROM EMP WHERE SAL<=55000;
EMPNO      ENAME      JOB                  SAL
---------- ---------- -------------------- ----------
7369       SMITH      CLERK                8000
7499       ALLEN      SALESMAN             3000
7521       WARD       SALESMAN             5000
7566       KUMAR      COE                  55000
7499       RAM KUMARSR.SALESMAN            12000.55
7521       SAM KUMARSR.SALESMAN            22000
6 rows selected.
SQL> SELECT EMPNO,ENAME,JOB,SAL FROM EMP WHERE SAL>=55000;
EMPNO      ENAME      JOB                  SAL
---------- ---------- -------------------- ----------
7566       JONES      MANAGER              75000
7566       KUMAR      COE                  55000
```

AND / OR

SQL> SELECT EMPNO,ENAME,JOB,SAL FROM EMP WHERE JOB='SR.SALESMAN' **AND** SAL=22000;

EMPNO	ENAME	JOB	SAL
7521	SAM KUMAR	SR.SALESMAN	22000

SQL> SELECT EMPNO,ENAME,JOB,SAL FROM EMP WHERE JOB='SR.SALESMAN' **OR** SAL=22000;

EMPNO	ENAME	JOB	SAL
7499	RAM KUMAR	SR.SALESMAN	12000.55
7521	SAM KUMAR	SR.SALESMAN	22000

SQL> SELECT EMPNO,ENAME,JOB,SAL FROM EMP

WHERE SAL=5000 **AND** (JOB='SR.SALESMAN' **OR** JOB='SALESMAN');

EMPNO	ENAME	JOB	SAL
7521	WARD	SALESMAN	5000

ORDER BY

Syntax:

SELECT *column_name,column_name*
FROM *table_name*
ORDER BY *column_name,column_name* ASC|DESC;

SQL> SELECT EMPNO,ENAME,JOB,SAL FROM EMP **ORDER BY ENAME**;

EMPNO	ENAME	JOB	SAL
7499	ALLEN	SALESMAN	3000
7566	JONES	MANAGER	75000
7566	KUMAR	COE	55000
7566	RAJA	OWNER	
7499	RAM KUMAR	SR.SALESMAN	12000.55
7521	SAM KUMAR	SR.SALESMAN	22000
7369	SMITH	CLERK	8000
7521	WARD	SALESMAN	5000

8 rows selected.

SQL> SELECT EMPNO,ENAME,JOB,SAL FROM EMP ORDER BY ENAME **DESC;**

EMPNO	ENAME	JOB	SAL
7521	WARD	SALESMAN	5000
7369	SMITH	CLERK	8000
7521	SAM KUMAR	SR.SALESMAN	22000
7499	RAM KUMAR	SR.SALESMAN	12000.55
7566	RAJA	OWNER	
7566	KUMAR	COE	55000
7566	JONES	MANAGER	75000
7499	ALLEN	SALESMAN	3000

8 rows selected.

SQL> SELECT EMPNO,ENAME,JOB,SAL FROM EMP ORDER BY ENAME **ASC;**

EMPNO	ENAME	JOB	SAL
7499	ALLEN	SALESMAN	3000
7566	JONES	MANAGER	75000
7566	KUMAR	COE	55000
7566	RAJA	OWNER	
7499	RAM KUMAR	SR.SALESMAN	12000.55
7521	SAM KUMAR	SR.SALESMAN	22000
7369	SMITH	CLERK	8000
7521	WARD	SALESMAN	5000

8 rows selected.

TOP

// **TOP** clause is not in oracle instead of that **ROWNUM**

Syntax

SELECT *column_name(s)*
FROM *table_name*
WHERE ROWNUM <= *number*;

SQL> SELECT EMPNO,ENAME,JOB,SAL FROM EMP WHERE **ROWNUM <4;**

EMPNO	ENAME	JOB	SAL
7369	SMITH	CLERK	8000
7499	ALLEN	SALESMAN	3000
7521	WARD	SALESMAN	5000

SQL>SELECTEMPNO,ENAME,JOB,SALFROMEMPWHERE**ROWNUM<4ORDERBYENAME;**

EMPNO	ENAME	JOB	SAL
7499	ALLEN	SALESMAN	3000
7369	SMITH	CLERK	8000
7521	WARD	SALESMAN	5000

DISTINCT

Syntax: *SELECT DISTINCT* column_name,column_name FROM *table_name*;

Ex:

SQL> SELECT **DISTINCT** JOB FROM EMP;

JOB
CLERK
SALESMAN
SR.SALESMAN
MANAGER
COE
OWNER

6 rows selected.

USING ALTER

This can be used to add or remove columns and to modify the precision of the datatype.

a) ADDING COLUMN

Syntax:
alter table <*table_name*> add <*col datatype*>;

Ex:
```
SQL> DESC EMP;
 Name                    Null?          Type
 ----------------------- ------------   -----------------
 EMPNO                                  NUMBER(4)
 ENAME                                  VARCHAR2(10)
 JOB                                    VARCHAR2(20)
 MGR                                    NUMBER(4)
 HIREDATE                               DATE
 SAL                                    NUMBER(8,2)
 DEPTNO                                 NUMBER(3)
SQL> alter table EMP add TAX number;
Table altered.

SQL> DESC EMP;
 Name                           Null?   Type
 -------------------------------------- --------------------------
 EMPNO                                  NUMBER(4)
 ENAME                                  VARCHAR2(10)
 JOB                                    VARCHAR2(20)
 MGR                                    NUMBER(4)
 HIREDATE                               DATE
 SAL                                    NUMBER(8,2)
 DEPTNO                                 NUMBER(3)
 TAX                                    NUMBER
```

b) REMOVINGCOLUMN

Syntax:
alter table <*table_name*> drop <*col datatype*>;

Ex:

```
SQL> alter table EMP drop column TAX;
Table altered.

SQL> DESC EMP;
 Name                    Null?               Type
 ----------------------- ------------------  -------------------
 EMPNO                                       NUMBER(4)
 ENAME                                       VARCHAR2(10)
 JOB                                         VARCHAR2(20)
 MGR                                         NUMBER(4)
 HIREDATE                                    DATE
 SAL                                         NUMBER(8,2)
 DEPTNO                                      NUMBER(3)
```

c) INCREASING OR DECREASING PRECISION OF A COLUMN

Syntax:

alter table <*table_name*> modify <*col datatype*>;

Ex:

SQL> alter table EMP modify DEPTNO number(5);
Table altered.

SQL> DESC EMP;

Name	Null?	Type
EMPNO		NUMBER(4)
ENAME		VARCHAR2(10)
JOB		VARCHAR2(20)
MGR		NUMBER(4)
HIREDATE		DATE
SAL		NUMBER(8,2)
DEPTNO		NUMBER(5)

* To decrease precision the column should be empty.

d) MAKING COLUMNUNUSED

Syntax:
alter table <*table_name*> set unused column <*col*>;

Ex:
SQL> alter table EMP set unused column DEPTNO;
Table altered.

SQL> DESC EMP;

Name	Null?	Type
EMPNO		NUMBER(4)
ENAME		VARCHAR2(10)
JOB		VARCHAR2(20)
MGR		NUMBER(4)
HIREDATE		DATE
SAL		NUMBER(8,2)

SQL> SELECT * FROM EMP;

EMPNO	ENAME	JOB	MGR	HIREDATE	SAL
7369	SMITH	CLERK	5001	17-DEC-80	8000
7499	ALLEN	SALESMAN	5002	20-FEB-80	3000
7521	WARD	SALESMAN	5003	22-FEB-80	5000

9 rows selected.

Even though the column is unused still it will occupy memory.

d) DROPPING UNUSED COLUMNSSyntax:
alter table <*table_name*> drop unused columns;

Ex:
SQL> alter table EMP drop unused columns;
Table altered.

* You can not drop individual unused columns of a table.

e) RENAMING COLUMNSyntax:
alter table <*table_name*> rename column <*old_col_name*> to <*new_col_name*>;
Ex:
SQL> alter table EMP rename column SAL to SALARY;

Table altered.

SQL> DESC EMP;

Name	Null?	Type
EMPNO		NUMBER(4)
ENAME		VARCHAR2(10)
JOB		VARCHAR2(20)
MGR		NUMBER(4)
HIREDATE		DATE
SALARY		NUMBER(8,2)

INSERT

Method 1

GENERAL INSERT COMMAND:

```
SQL> INSERT INTO EMP(EMPNO,ENAME,JOB,MGR,HIREDATE,SALARY)
        VALUES(1111,'RAMU','SALESMAN',5063,'12-JAN-87','5643.55');
1 row created.
```

Method 2

WITHOUT SPECIFY THE COLUMNS DETAILS

SQL> INSERT INTO Emp VALUES(1111,'RAMU','SALESMAN',5063,'12-JAN-87','5643.55'); 1 row created.

Method 3

INSERTING DATA INTO SPECIFIEDCOLUMNS

SQL> INSERT INTO EMP(EMPNO,ENAME,JOB)
VALUES(1111,'RAMU','SALESMAN'); 1 rowcreated.

Method 4

BY CHANGE THE ORDER OF COLUMNS

SQL> INSERT INTO EMP(salary,EMPNO,ENAME,JOB)
VALUES(35000,1111,'RAMU','SALESMAN'); 1 row created.

SQL> select * from emp;

EMPNO	ENAME	JOB	MGR	HIREDATE	SALARY
7369	SMITH	CLERK	5001	17-DEC-80	8000
7499	ALLEN	SALESMAN	5002	20-FEB-80	3000
7521	WARD	SALESMAN	5003	22-FEB-80	5000
7566	JONES	MANAGER	5002	02-APR-85	75000
7566	RAJA	OWNER	5000	30-APR-75	
7566	KUMAR	COE	5002	12-JAN-87	55000
7499	RAM KUMAR	SR.SALESMAN	5003	22-JAN-87	12000.55
7521	SAM KUMAR	SR.SALESMAN	5003	22-JAN-75	22000
7521	SAM KUMAR	SR.SALESMAN	5003	22-JAN-75	22000
1111	RAMU	SALESMAN	5063	12-JAN-87	5643.55
1111	RAMU	SALESMAN	5063	12-JAN-87	5643.55
1111	RAMU	SALESMAN			
1111	RAMU	SALESMAN			35000

13 rows selected.

Method 5

INSERT WITH SELECT

Using this we can insert existing table data to another table in a single trip. But the table structure should be same.

Syntax:

Insert into <*table1*> select * from <*table2*>;

Ex:

SQL> DESC EMP

Name	Null?	Type
EMPNO		NUMBER(4)
ENAME		VARCHAR2(10)
JOB		VARCHAR2(20)
MGR		NUMBER(4)
HIREDATE		DATE
SALARY		NUMBER(8,2)

SQL> create table EMPLOYEE(EMP_NO,EMP_NAME,EMP_JOB,HR,HIREDATE,SALARY) as select * from EMP where 1 = 2;
Table created.

SQL> DESC EMPLOYEE

Name	Null?	Type
EMP_NO		NUMBER(4)
EMP_NAME		VARCHAR2(10)
EMP_JOB		VARCHAR2(20)
HR		NUMBER(4)
HIREDATE		DATE
SALARY		NUMBER(8,2)

SQL> SELECT * FROM
EMPLOYEE; no rows selected

SQL> insert into EMPLOYEE select * from
EMP; 13 rows created.

SQL> SELECT * FROM EMPLOYEE;

EMP_NO	EMP_NAME	EMP_JOB	HR	HIREDATE	SALARY
7369	SMITH	CLERK	5001	17-DEC-80	8000
7499	ALLEN	SALESMAN	5002	20-FEB-80	3000
7521	WARD	SALESMAN	5003	22-FEB-80	5000
7566	JONES	MANAGER	5002	02-APR-85	75000
7566	RAJA	OWNER	5000	30-APR-75	
7566	KUMAR	COE	5002	12-JAN-87	55000
7499	RAM KUMAR	SR.SALESMAN	5003	22-JAN-87	12000.55
7521	SAM KUMAR	SR.SALESMAN	5003	22-JAN-75	22000
7521	SAM KUMAR	SR.SALESMAN	5003	22-JAN-75	22000
1111	RAMU	SALESMAN	5063	12-JAN-87	5643.55
1111	RAMU	SALESMAN	5063	12-JAN-87	5643.55
1111	RAMU	SALESMAN			
1111	RAMU	SALESMAN			35000

13 rows selected.

GROUP BY

Using group by, we can create groups of related information. Columns used in select must be used with group by; otherwise it was not a group by expression.

Ex:

SQL> select * from emp;

EMPNO	ENAME	JOB	MGR	HIREDATE	SAL	DEPTNO
7369	SMITH	CLERK	5001	17-DEC-80	8000	200
7499	ALLEN	SALESMAN	5002	20-FEB-80	3000	300
7521	WARD	SALESMAN	5003	22-FEB-80	5000	500
7499	RAM KUMAR	SR.SALESMAN	5003	22-JAN-87	12000.55	200
7566	JONES	MANAGER	5002	02-APR-85	75000	200
7521	SAM KUMAR	SR.SALESMAN	5003	22-JAN-75	22000	300

6 rows selected.

SQL> select job from EMP group by job;

JOB

CLERK
SALESMAN
SR.SALESMAN
MANAGER

SQL> select job,SUM(SAL) from EMP group by job;

JOB	SUM(SAL)
CLERK	8000
SALESMAN	8000
SR.SALESMAN	34000.55
MANAGER	75000

HAVING

This will work as where clause which can be used only with group by because of absence of where clause in group by.

SQL> select deptno,job,sum(sal) Total_Salary_Of_Each_Dept
from emp group by deptno,job having sum(sal) > 3000;

DEPTNO	JOB	TOTAL_SALARY_OF_EACH_DEPT

```
200         MANAGER          75000
200         SR.SALESMAN      12000.55
200         CLERK            8000
500         SALESMAN         5000
300         SR.SALESMAN      22000
```

```
SQL> select deptno,job,sum(sal) Total_Salary_of_Each_Dept from emp
                                group by deptno,job
                                having sum(sal) > 3000
                                order by job;
DEPTNO      JOB              TOTAL_SALARY_OF_EACH_DEPT
----------  ---------------  -------------------------
200         CLERK            8000
200         MANAGER          75000
500         SALESMAN         5000
200         SR.SALESMAN      12000.55
300         SR.SALESMAN      22000
```

USING DELETE

```
SQL> select * from EMP;
EMPNO       ENAME       JOB              MGR      HIREDATE        SAL      DEPTNO
----------  ----------  ---------------  -------  --------------  -------  ----------
1001        RAM         CLERK            5001     17-DEC-84       8000     301
1002        SAM         MANAGER          5001     11-JAN-81       85000    301
1003        SAMU        SALESMAN         5003     09-FEB-82       8000     302
1004        RAMU        SR.SALESMAN      5002      22-JUN-85      45000    303

SQL> DELETE EMP WHERE ENAME='SAM';
1 rowdeleted.

SQL> select * from EMP;
EMPNO       ENAME       JOB              MGR      HIREDATE        SAL      DEPTNO
----------  ----------  ---------------  -------  --------------  -------  ----------
1001        RAM         CLERK            5001     17-DEC-84       8000     301
1003        SAMU        SALESMAN         5003     09-FEB-82       8000     302
1004        RAMU        SR.SALESMAN      5002     22-JUN-85       45000    303

SQL> DELETE EMP WHERE ENAME LIKE 'R';
1 rowdeleted.

SQL> select * from EMP;
EMPNO       ENAME       JOB              MGR      HIREDATE        SAL      DEPTNO
----------  ----------  ---------------  -------  --------------  -------  ----------
1003        SAMU        SALESMAN         5003     09-FEB-82       8000     302
1004        RAMU        SR.SALESMAN      5002     22-JUN-85       45000    303

SQL> DELETE FROM EMP WHERE
ENAME='SAMU'; 1 row deleted.
```

TO DELETE ALL RECORDS

```
SQL> DELETE FROM
EMP; 1 row deleted.
```

DELETE DUPLICATE ROWS

```
SQL> SELECT * FROM myTBL;
NAME MARK
---------- ----------
RAM    101
RAM    101
SAM    102
```

```
SAM     102
RAMU
RAMU
SAMU  103
SAMU  103
SAMU  103
TAM
RAJA    555
KAJA    123

12 rows selected.

SQL> delete from myTBL t1
where t1.rowid > (select min(t2.rowID) from myTBL
t2 where t1.name = t2.name and t1.mark = t2.mark);
4 rows deleted.
SQL> SELECT * FROM myTBL;

NAME            MARK
---------       ----------
RAM             101
SAM             102
RAMU
SAMU            103
TAM
RAJA            555
KAJA            123

8 rows selected.
```

Using UPDATE

```
SQL> select * from EMP;
EMPNO          ENAME          JOB            MGR    HIREDATE       SAL      DEPTNO
-------------- -------------- -------------- ------ ------------------ ---------- ----------
1001           RAM            CLERK          5001   17-DEC-84      8000     301
1002           SAM            MANAGER        5001   11-JAN-81      85000    301
1003           SAMU           SALESMAN       5003   09-FEB-82      8000     302
SQL> UPDATE EMP SET SAL = 55555,JOB = 'SR.MANAGER' WHERE ENAME
LIKE 'R'; 1 row updated.

SQL> select * from EMP;
EMPNO          ENAME          JOB            MGR    HIREDATE       SAL      DEPTNO
-------------- -------------- -------------- ------ ------------------ ---------- ----------
1001           RAM            SR.MANAGER     5001   17-DEC-84      55555    301
1002           SAM            MANAGER        5001   11-JAN-81      85000    301
1003           SAMU           SALESMAN       5003   09-FEB-82      8000     302

SQL> UPDATE EMP SET SAL = 55555,JOB = 'SR.MANAGER';
3 rows updated.

SQL> select * from EMP;
EMPNO          ENAME          JOB            MGR    HIREDATE       SAL      DEPTNO
-------------- -------------- -------------- ------ ------------------ ---------- ----------
1001           RAM            SR.MANAGER     5001   17-DEC-84      55555    301
1002           SAM            SR.MANAGER     5001   11-JAN-81      55555    301
1003           SAMU           SR.MANAGER     5003   09-FEB-82      55555    302
```

RESULT:

Thus the SQL commands for simple queries using Insertion, Deletion, Modifying, Altering, Updating and Viewing records based on conditions has been verified and executedsuccessfully.

2.1 Implementation of Subqueries

A Subquery or Inner query or a Nested query is a query within another SQL query and embedded within the WHERE clause.

Subqueries are most frequently used with the SELECT statement. The basic syntax is as follows −

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

Consider the CUSTOMERS table having the following records −

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

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

This would produce the following result.

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

Subqueries with the INSERT Statement

```
INSERT INTO CUSTOMERS_BKP
  SELECT * FROM CUSTOMERS
  WHERE ID IN (SELECT ID
  FROM CUSTOMERS) ;
```

Subqueries with the UPDATE Statement

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

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

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

Subqueries with the DELETE Statement

```
DELETE FROM CUSTOMERS
  WHERE AGE IN (SELECT AGE FROM CUSTOMERS_BKP
    WHERE AGE >= 27 );
```

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

```
+----+----------+-----+---------+----------+
| ID|NAME      | AGE | ADDRESS | SALARY|
+----+----------+-----+---------+----------+
|  2 | Khilan   |  25 | Delhi   |  1500.00|
|  3 | kaushik  |  23|Kota      |  2000.00|
| 4 | Chaitali | 25 | Mumbai | 6500.00 |
|  6|Komal     |  22|MP        | 4500.00|
|  7|Muffy     | 24 | Indore | 10000.00|
+----+----------+-----+---------+----------+
```

RESULT:

Thus the SQL commands for sub queries based on conditions has been verified and executed successfully.

Implementation of Joins

A JOIN is a means for combining fields from two tables by using values common to each.

Consider the following two tables −

Table 1 − CUSTOMERS Table

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

Table 2 − ORDERS Table

```
+-----+---------------------+-------------+--------+
|OID  |DATE                 | CUSTOMER_ID | AMOUNT|
+-----+---------------------+-------------+--------+
| 102 | 2009-10-0800:00:00 |           3 |   3000|
| 100 | 2009-10-0800:00:00 |           3 |   1500|
| 101 | 2009-11-2000:00:00 |           2 |   1560|
| 103 | 2008-05-2000:00:00 |           4 |   2060|
+-----+---------------------+-------------+--------+
```

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

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

This would produce the following result.

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

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 SQLstatement.
CARTESIAN JOIN− returns the Cartesian product of the sets of records from the two or more joined tables.

1. **INNER JOIN**

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

This would produce the following result.

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

2. **LEFT JOIN**

```
SELECT ID, NAME, AMOUNT, DATE
FROMCUSTOMERS
LEFT JOINORDERS
ON CUSTOMERS.ID = ORDERS.CUSTOMER_ID;
```

This would produce the following result −

```
+---+---------+--------+---------------------+
| ID|NAME     | AMOUNT|DATE                 |
+---+---------+--------+---------------------+
|  1 | Ramesh   |   NULL|NULL                 |
|  2 | Khilan   |   1560 | 2009-11-20 00:00:00|
|  3 | kaushik  |   3000 | 2009-10-08 00:00:00|
|  3 | kaushik  |   1500 | 2009-10-08 00:00:00|
| 4 | Chaitali | 2060 | 2008-05-20 00:00:00 |
|  5 | Hardik   |   NULL|NULL                 |
|  6|Komal     |   NULL|NULL                 |
|  7|Muffy     |   NULL|NULL                 |
```

3. **RIGHTJOIN**

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

This would produce the following result −

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

4. **FULLJOIN**

```
SELECT ID, NAME, AMOUNT, DATE
  FROMCUSTOMERS
  FULL JOINORDERS
  ON CUSTOMERS.ID = ORDERS.CUSTOMER_ID;
```

This would produce the following result

```
| ID |NAME       | AMOUNT|DATE                 |

+----+-------+--------+---------------------+
|  1 | Ramesh   |   NULL|NULL                 |
|  2 | Khilan   |  1560 | 2009-11-20 00:00:00|
|  3 | kaushik  |  3000 | 2009-10-08 00:00:00|
|  3 | kaushik  |  1500 | 2009-10-08 00:00:00|
|  4 | Chaitali | 2060 | 2008-05-20 00:00:00|
|  5 | Hardik   |  NULL| NULL                 |
|  6|Komal     |  NULL| NULL                  |
|  7|Muffy     |  NULL|NULL                   |
|  3 | kaushik  |  3000 | 2009-10-08 00:00:00|
|  3 | kaushik  |  1500 | 2009-10-08 00:00:00|
|  2 | Khilan   |  1560 | 2009-11-20 00:00:00|
|  4 | Chaitali |  2060 | 2008-05-20 00:00:00|
+----+-------+--------+---------------------+
```

5. **SELF JOIN**

6.

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

This would produce the following result

```
| ID|NAME      | SALARY|
+---+----------+--------+
|  2 | Ramesh   | 1500.00|
| 2 | kaushik | 1500.00 |
|  1 | Chaitali | 2000.00|
|  2 | Chaitali | 1500.00|
|  3 | Chaitali | 2000.00|
|  6 | Chaitali | 4500.00|
|  1 | Hardik   | 2000.00|
|  2 | Hardik   | 1500.00|
```

```
|  3 | Hardik   | 2000.00|
|  4 | Hardik   | 6500.00|
|  6 | Hardik   | 4500.00|
|  1|Komal      | 2000.00|
|  2|Komal      | 1500.00|
|  3|Komal      | 2000.00|
|  1|Muffy      | 2000.00|
|  2|Muffy      | 1500.00|
|  3|Muffy      | 2000.00|
|  4|Muffy      | 6500.00|
|  5|Muffy      | 8500.00|
|  6|Muffy      | 4500.00|
+---+-----------+----------+
```

RESULT:

Thus the SQL commands for joins based on different types has been verified and executed successfully.

Creation of Views, Synonyms, Sequence

Ex: No:03(3.1) **VIEWS**

__:__: __

AIM:

To create the view, execute and verify the various operations on views.

OBJECTIVE:

- Views Helps to encapsulate complex query and make itreusable.
- Provides user security on each view - it depends on your data policysecurity.
    - Using view to convert units - if you have a financial data in US currency, you can create view to convert them into Euro for viewing in Eurocurrency.
- A view is nothing more than a SQL statement that is stored in the database with an associated name. A view is actually a composition of a table in the form of a predefined SQLquery.
- A view can contain all rows of a table or select rows from a table. A view can be created from one or many tables which depends on the written SQL query to create aview.
- Views, which are kind of virtual tables, allow users to do thefollowing:
- Structure data in a way that users or classes of users find natural orintuitive.
    - Restrict access to the data such that a user can see and (sometimes) modify exactly what they need and nomore.

ALGORITHM:

STEP 1: Start the DMBS.

STEP 2: Connect to the existing database(DB)

STEP 3: Create the table with its essentialattributes.

STEP 4: Insert record values into thetable.

STEP 5: Create the view from the above created table.

STEP 6: Display the data presented on the VIEW.

STEP 7: Insert the records into the VIEW,

STEP 8: Check the database object table and view the inserted values presented

STEP 9: Execute different Commands and extract information from the View.

STEP 10: Stop the DBMS.

COMMANDS EXECUTION
CREATION OF TABLE:

Database views are created using the **CREATE VIEW** statement. Views can be created from a single table, multiple tables, or another view. To create a view, a user must have the appropriate system privilege according to the specificimplementation.

```
SQL> CREATE TABLE EMPLOYEE (
                                    EMPLOYEE_NAME VARCHAR2(10),
                                    EMPLOYEE_NO     NUMBER(8),
                                    DEPT_NAME       VARCHAR2(10),
                                    DEPT_NO         NUMBER(5),
                                    DATE_OF_JOIN    DATE
                                );
Table created.
```

TABLE DESCRIPTION:

```
SQL> DESC EMPLOYEE;
        NAMENULL?                TYPE
        ------------------------- ----------------------
        EMPLOYEE_NAME            VARCHAR2(10)
        EMPLOYEE_NO              NUMBER(8)
        DEPT_NAME                VARCHAR2(10)
        DEPT_NO                  NUMBER(5)
        DATE_OF_JOIN             DATE
```

CREATE VIEW
SUNTAX FOR CREATION OF VIEW

CREATE [OR REPLACE] [FORCE] VIEW viewname [(column-name, column-name)] AS Query [with check option];

CREATION OF VIEW

```
SQL> CREATE VIEW EMPVIEW AS SELECT EMPLOYEE_NAME,
EMPLOYEE_NO,
DEPT_NAME,
DEPT_NO,
DATE_OF_JOIN FROM EMPLOYEE;
View Created.
```

DESCRIPTION OF VIEW

```
SQL> DESC EMPVIEW;

NAMENULL?                TYPE
------------------------ -------------------------------
EMPLOYEE_NAME            VARCHAR2(10)
EMPLOYEE_NO              NUMBER(8)
DEPT_NAME                VARCHAR2(10)
DEPT_NO                  NUMBER(5)
```

DISPLAY VIEW:

```
SQL> SELECT * FROM EMPVIEW;
EMPLOYEE_N          EMPLOYEE_NO          DEPT_NAME          DEPT_NO
----------          -----------          ----------         ----------
RAVI                124                  ECE                89
VIJAY               345                  CSE                21
RAJ                 98                   IT                 22
GIRI                100                  CSE                67
```

INSERTION OF VALUES INTO VIEW

Rows of data can be inserted into a view. The same rules that apply to the UPDATE command also apply to the INSERT command. Here, we can not insert rows in CUSTOMERS_VIEW because we have not included all the NOT NULL columns in this view, otherwise you can insert rows in a view in similar way as you insert them in a table.

INSERT STATEMENT SYNTAX:

```
SQL> INSERT INTO <VIEW_NAME> (COLUMN NAME1, …) VALUES(VALUE1,….);
```

COMMAND:

```
SQL> INSERT INTO EMPVIEW VALUES ('SRI', 120,'CSE', 67,'16-NOV-1981');
     1 ROW CREATED.

SQL> SELECT * FROM EMPVIEW;
     EMPLOYEE_N          EMPLOYEE_NO          DEPT_NAMEDEPT_NO
     ----------          -----------          ----------         ----------
     RAVI                124                  ECE                89
     VIJAY               345                  CSE                21
     RAJ                 98                   IT                 22
     GIRI                100                  CSE                67
     SRI                 120                  CSE                67

SQL> SELECT * FROM EMPLOYEE;
     EMPLOYEE_N          EMPLOYEE_NO          DEPT_NAMEDEPT_NO         DATE_OF_J
     ----------          -----------          ----------         ----------  ----------
     RAVI                124                  ECE                89          15-JUN-05
     VIJAY               345                  CSE                21          21-JUN-06
     RAJ                 98                   IT                 22          30-SEP-06
     GIRI                100                  CSE                67          14-NOV-81
     SRI                 120                  CSE                67          16-NOV-81
```

DELETION OF VIEW:

Rows of data can be deleted from a view. The same rules that apply to the UPDATE and INSERT commands apply to the DELETEcommand.

DELETE STATEMENT SYNTAX:

SQL> DELETE <VIEW_NMAE>WHERE <COLUMN NMAE> ='VALUE';

Command:

SQL> DELETE FROM EMPVIEW WHERE
EMPLOYEE_NAME='SRI'; 1 row deleted.

SQL> SELECT * FROM EMPVIEW;

EMPLOYEE_N	EMPLOYEE_NO	DEPT_NAME	DEPT_NO
RAVI	124	ECE	89
VIJAY	345	CSE	21
RAJ	98	IT	22
GIRI	100	CSE	67

UPDATE STATEMENT:

A view can be updated under certain conditions:

- The SELECT clause may not contain the keywordDISTINCT.
- The SELECT clause may not contain summaryfunctions.
- The SELECT clause may not contain setfunctions.
- The SELECT clause may not contain setoperators.
- The SELECT clause may not contain an ORDER BY clause.
  The FROM clause may not contain multipletables.
- The WHERE clause may not contain subqueries.
- The query may not contain GROUP BYor
- HAVING. Calculated columns may not beupdated.
- All NOT NULL columns from the base table must be included in the view in order for the INSERT query tofunction.
- **SYNTAX:**

SQL>UPDATE <VIEW_NAME> SET< COLUMN NAME> = <COLUMN NAME>
+<VIEW> WHERE <COLUMN NAME>=VALUE;

Command:

SQL> UPDATE EMPKAVIVIEW SET EMPLOYEE_NAME='KAVI' WHERE
EMPLOYEE_NAME='RAVI'; 1 row updated.

SQL> SELECT * FROM EMPKAVIVIEW;

EMPLOYEE_N	EMPLOYEE_NO	DEPT_NAME	DEPT_NO
KAVI	124	ECE	89
VIJAY	345	CSE	21
RAJ	98	IT	22
GIRI	100	CSE	67

DROP A VIEW:

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

SYNTAX:

SQL> DROP VIEW <VIEW_NAME>

EXAMPLE

SQL>DROP VIEW
EMPVIEW; view droped

CREATE A VIEW WITH SELECTED FIELDS:

SYNTAX:

SQL>CREATE [OR REPLACE] VIEW <VIEW NAME>AS SELECT <COLUMN NAME1>…..FROM <TABLE ANME>;

EXAMPLE-2:

SQL> CREATE OR REPLACE VIEW EMPL_VIEW1 AS SELECT EMPNO, ENAME,SALARY FROM EMPL; SQL> SELECT * FROM EMPL_VIEW1;

EXAMPLE-3:

SQL> CREATE OR REPLACE VIEW EMPL_VIEW2 AS SELECT * FROM EMPL WHERE DEPTNO=10; SQL> SELECT * FROM EMPL_VIEW2;

Note:

Replace is the keyboard to avoid the error “ora_0095:name is already used by an existing abject”.

CHANGING THE COLUMN(S) NAME M THE VIEW DURING AS SELECT STATEMENT:

TYPE-1:

SQL>CREATE OR REPLACE VIEW EMP_TOTSAL(EID,NAME,SAL) AS SELECT EMPNO, ENAME,SALARY FROM EMPL;
View created.

EMPNO	ENAME S	ALARY
7369	SMITH	1000
7499	MARK 1050	
7565	WILL	1500
7678	JOHN	1800
7578	TOM	1500
7548	TURNER	1500

6 rows selected.
View created.

SQL> SELECT * FROM EMP_TOTSAL;

EMPNO	ENAME	SALARY	MGRNO	DEPTNO
7578	TOM	1500	7298	10
7548	TURNER	1500	7298	10

View created.

TYPE-2:

SQL> CREATE OR REPLACE VIEW EMP_TOTSAL AS SELECT EMPNO "EID",
ENAME "NAME", SALARY "SAL" FROM EMPL;

SQL> SELECT * FROM EMP_TOTSAL;

EXAMPLE FOR JOIN VIEW:

TYPE-3:

SQL> CREATE OR REPLACE VIEW DEPT_EMP AS SELECT A.EMPNO "EID",
A.ENAME "EMPNAME",
EPTNO "DNO",
B.DNAME
"D_NAME",
B.LOC"D_LOC
"
FROM EMPL A,DEPMT B WHERE A.DEPTNO=B.DEPTNO;

SQL> SELECT * FROM DEPT_EMP;

EID	NAMESAL	
7369	SMITH	1000
7499	MARK 1050	
7565	WILL	1500
7678	JOHN	1800
7578	TOM	1500
7548	TURNER	1500

6 rows selected.
Viewcreated.

EID	NAMESAL	
7369	SMITH	1000
7499	MARK 1050	
7565	WILL	1500
7678	JOHN	1800
7578	TOM	1500
7548	TURNER	1500

6 rows selected.

Viewcreated.

EID	EMPNAME	DNO	D_NAME	D_LOC
7578	TOM	10	ACCOUNT	NEW YORK
7548	TURNER	10	ACCOUNT	NEW YORK
7369	SMITH	20	SALES	CHICAGO
7678	JOHN	20	SALES	CHICAGO
7499	MARK 30	RESEARCHZURICH		
7565	WILL	30	RESEARCH	ZURICH

VIEW READ ONLY AND CHECK OPTION:

READ ONLY CLAUSE:

You can create a view with read only option which enable other to only query .no DML operation can be performed to this type of a view.

EXAMPLE-4:

SQL>CREATE OR REPLACE VIEW EMP_NO_DML AS SELECT * FROM EMPL WITH READ ONLY;

WITH CHECK OPTION CLAUSE:

EXAMPLE-4:

SQL> CREATE OR REPLACE VIEW EMP_CK_OPTION AS SELECT EMPNO, ENAME, SALARY, DEPTNO FROM EMPL WHERE DEPTNO=10 WITH CHECK OPTION;

SQL> SELECT * FROM EMP_CK_OPTION;

JOIN VIEW:

EXAMPLE-5:

```
SQL> CREATE OR REPLACE VIEW DEPT_EMP_VIEW AS SELECT A.EMPNO,
                                                    NAME,
                                                    A.DEPTNO,
                                                    B.DNAME,
                                                    B.LOC
                                                    FROM EMPL A, DEPMT B
                                                    WHERE A.DEPTNO=B.DEPTNO;
```

SQL> SELECT * FROM DEPT_EMP_VIEW;

View created.

EMPNO	ENAME	SALARY	DEPTNO
7578	TOM	1500	10
7548	TURNER	1500	10

View created.

EMPNO	ENAME	DEPTNO	DNAME	LOC
7578	TOM	10	ACCOUNT	NEW YORK
7548	TURNER	10	ACCOUNT	NEW YORK
7369	SMITH	20	SALES	CHICAGO
7678	JOHN	20	SALES	CHICAGO
7499	MARK	30	RESEARCH	ZURICH
7565	WILL	30	RESEARCH	ZURICH

6 rows selected.

FORCE VIEW:

EXAMPLE-6:

SQL> CREATE OR REPLACE FORCE VIEW MYVIEW AS SELECT * FROM XYZ;

SQL> SELECT * FROM MYVIEW;

SQL> CREATE TABLE XYZ AS SELECT EMPNO,ENAME,SALARY,DEPTNO FROM EMPL;

SQL> SELECT * FROM XYZ;

SQL> CREATE OR REPLACE FORCE VIEW MYVIEW AS SELECT * FROM XYZ;

SQL> SELECT * FROM MYVIEW;

Warning: View created with compilation errors.

SELECT * FROM MYVIEW

*

ERROR at line 1:

ORA-04063: view "4039.MYVIEW" has errors

Table created.

EMPNO	ENAME	SALARY	DEPTNO
7369	SMITH	1000	20
7499	MARK 1050	30	
7565	WILL	1500	30
7678	JOHN	1800	20
7578	TOM	1500	10
7548	TURNER	1500	10

6 rows selected.

View created.

EMPNO	ENAME	SALARY	DEPTNO
7369	SMITH	1000	20
7499	MARK 1050	30	
7565	WILL	1500	30
7678	JOHN	1800	20
7578	TOM	1500	10
7548	TURNER	1500	10

6 rows selected

COMPILING A VIEW:

SYNTAX:

ALTER VIEW <VIEW_NAME> COMPILE;

EXAMPLE:

SQL> ALTER VIEW MYVIEW COMPILE;

RESULT:

Thus the SQL commands for View has been verified and executed successfully.

Synonyms

Ex: No: 03 (3.2)

__:__: __

AIM:

To create the Synonyms and verify the various operations on Synonyms

OBJECTIVE:

A **synonym** is an alias for any table, view, materialized view, sequence, procedure, function, package, type, Java class schema object, user-defined object type, or another synonym. Because a synonym is simply an alias, it requires no storage other than its definition in the data dictionary.

Synonyms are often used for security and convenience. For example, they can do thefollowing:

Mask the name and owner of anobject

Provide location transparency for remote objects of a distributed database Simplify SQL statements for databaseusers

Enable restricted access similar to specialized views when exercising fine-grained access control

You can create both public and private synonyms. A **public** synonym is owned by the special user group named PUBLIC and every user in a database can access it. A **private** synonym is in the schema of a specific user who has control over its availability toothers.

ALGORITHM:

STEP 1: Start the DMBS.

STEP 2: Connect to the existing database(DB)

STEP 3: Create the table with its essential attributes.

STEP 4: Insert record values into thetable.

STEP 5: Create the synonyms from the above created table or any data object.

STEP 6: Display the data presented on the synonyms.

STEP 7: Insert the records into the synonyms,

STEP 8: Check the database object table and view the inserted values presented

STEP 9: Stop the DBMS.

Example:

Syntax:

SQL>CREATE SYNONYM synonymName **FOR** object;

OR

SQL>CREATE SYNONYM **tt** for **test1;**

Dependent Oject - **tt** (SYNONYM NAME)

Referenced Object - **test1** (TABLE NAME)

USAGE:

Using emp you can perform DML operation as you have create sysnonm for table object
If employees table is dropped then status of emp will beinvalid.

Local Dependencies are automatically managed by oracle server.

COMMANDS:

CREATE THE TABLE

SQL> CREATE TABLE student_table(Reg_No number(5),NAME varchar2(5),MARK number(3));
Table created.

INSERT THE VALUES INTO THE TABLE

SQL> insert into student_table
values(10001,'ram',85); 1 rowcreated.

SQL> insert into student_table
values(10002,'sam',75); 1 rowcreated.

SQL> insert into student_table
values(10003,'samu',95); 1 rowcreated.

SQL> select * from STUDENT_TABLE;

REG_NO	NAME	MARK
10001	ram	85
10002	sam	75
10003	samu	95

CREATE THE SYNONYM FROM TABLE

SQL> CREATE SYNONYM STUDENT_SYNONYM FOR STUDENT_TABLE;

Synonym created.

DISPLAY THE VALUES OF TABLE BY USING THE SYSNONYM

SQL> select * from STUDENT_SYNONYM;

REG_NO	NAME	MARK
10001	ram	85
10002	sam	75
10003	samu	95

INSERT THE VALUES TO THE SYNONYM

SQL> insert into student_SYNONYM values(10006,'RAJA',80);

1 row created.

DISPLAY THE VALUES IN BOTH TABLE AND SYNONYM

SQL> select * from STUDENT_TABLE;

REG_NO	NAME	MARK
10001	ram	85
10002	sam	75
10003	samu	95
10006	RAJA	80

SQL> select * from STUDENT_SYNONYM;

REG_NO	NAME	MARK
10001	ram	85
10002	sam	75
10003	samu	95
10006	RAJA	80

YOU CAN UPDATE THE TABLE BY USING SYNONYM

SQL> UPDATE STUDENT_SYNONYM SET MARK=100 WHERE REG_NO=10006;

1 rowupdated.

SQL> select * from STUDENT_SYNONYM;

REG_NO	NAME	MARK
10001	ram	85
10002	sam	75
10003	samu	95
10006	RAJA	100

SQL> select * from STUDENT_TABLE;

REG_NO	NAME	MARK
10001	ram	85
10002	sam	75
10003	samu	95
10006	RAJA	100

TO DROP SYSNONYM

SQL> DROP SYNONYM STUDENT_SYNONYM;

Synonym dropped.

BUT WE CAN USE THE TABLE

SQL> select * from STUDENT_TABLE;

REG_NO	NAME	MARK
10001	ram	85
10002	sam	75
10003	samu	95
10006	RAJA	100

RESULT:

Thus the SQL commands for creation and various operation on Synonyms has been verified and executed successfully.

Sequence

Ex: No: 03 (3.3)

__:___: __

AIM:

To create the Sequence and verify the various operations on Sequence to get the incrementednumber.

OBJECTIVE:

The sequence generator provides a sequential series of numbers. The sequence generator is especially useful in multiuser environments for generating unique sequential numbers without the overhead of disk I/O or transaction locking

Sequence numbers are integers of up to 38 digits defined in the database. A sequence definition indicates general information, such as thefollowing:

The name of the sequence

Whether the sequence ascends or descends The interval betweennumbers

Whether Oracle Database should cache sets of generated sequence numbers in memory

ALGORITHM:

Step 1: Start the DMBS.

Step 2: Connect to the existing database (DB)

Step 3: Create the sequence with its essential optional parameter.

Step 4: Display the data presented on the sequence by using pseudo column.

Step 5: Alter the sequence with different optional parameter.

Step 6: Drop the sequence

Step 7: Stop the DBMS.

Creating a Sequence

You create a sequence using the CREATE SEQUENCE statement, which has the following.

SYNTAX:

SQL>CREATE SEQUENCE sequence_name

[**START WITH** start_num]

[**INCREMENT BY** increment_num]

[{ **MAXVALUE** maximum_num | **NOMAXVALUE** }] [

{ **MINVALUE** minimum_num | **NOMINVALUE** }]

[{ **CYCLE** | **NOCYCLE** }]

[{ **CACHE** cache_num | **NOCACHE** }] [

{ **ORDER** | **NOORDER** }];

Where

sequence_name is the name of the sequence.

start_num is the integer to start the sequence. The default start number is 1.

increment_num is the integer to increment the sequence by. The default increment number is 1. The absolute value of *increment_num* must be less than the difference between *maximum_num* and *minimum_num*.

maximum_num is the maximum integer of the sequence; *maximum_num* must be greater than or equal to *start_num*, and *maximum_num* must be greater than *minimum_num*.

NOMAXVALUE specifies the maximum is 1027 for an ascending sequence or –1 for a descending sequence. NOMAXVALUE is the default.

minimum_num is the minimum integer of the sequence; *minimum_num* must be less than or equal to *start_num*, and *minimum_num* must be less than*maximum_num*.

NOMINVALUE specifies the minimum is 1 for an ascending sequence or –1026 for a descending sequence. NOMINVALUE is the default.

CYCLE means the sequence generates integers even after reaching its maximum or minimum value. When an ascending sequence reaches its maximum value, the next value generated is the minimum. When a descending sequence reaches its minimum value, the next value generated is the maximum.

NOCYCLE means the sequence cannot generate any more integers after reaching its maximum or minimum value. NOCYCLE is the default.

cache_num is the number of integers to keep in memory. The default number of integers to cache is 20. The minimum number of integers that may be cached is 2. The maximum integers that may be cached is determined by the formula CEIL(*maximum_num* - *minimum_num*)/ABS(*increment_num*).

NOCACHE means no caching. This stops the database from pre-allocating values for the sequence, which prevents numeric gaps in the sequence but reduces performance. Gaps occur because cached values are lost when the database is shut down. If you omit CACHE and NOCACHE, the database caches 20 sequence numbers by default.

ORDER guarantees the integers are generated in the order of the request. You typically use ORDER when using Real Application Clusters, which are set up and managed by database administrators.

NOORDER doesn't guarantee the integers are generated in the order of the request. NOORDER is the default.

Example: 1

Command:

```
SQL> CREATE SEQUENCE seq1
INCREMENT BY 1
START with 1
MAXVALUE5
MINVALUE0;
```

Sequence created.

TO DISPLAY THE VALUES OF SEQUENCES

After creating sequence use nextval as nextval is used to generate sequence values SQL> select seq1.nextval from dual;

```
NEXTVAL
-----------
      1
```

SQL> select seq1.nextval from dual;

```
NEXTVAL
-----------
      2
```

SQL> select seq1.nextval from dual;

```
NEXTVAL
-----------
      3
```

SQL> select seq1.currval from dual;

```
CURRVAL
-----------
      3
```

The following is the list of available pseudo columns in Oracle.

PseudoColumn		**Meaning**
CURRVAL	-	Returns the current value of a sequence.
NEXTVAL	-	Returns the next value of a sequence.
NULL	-	Return a null value.
ROWID	-	Returns the ROWID of a row. See ROWID section below.
ROWNUM	-	Returns the number indicating in which order Oracle selects rows. First row selected will be ROWNUM of 1 and second row ROWNUM of 2 and so on.
SYSDATE	-	Returns current date and time.
USER	-	Returns the name of the current user.
UID	-	Returns the unique number assigned to the current user.

TO ALTER THE SEQUENCES

```
alter SEQUENCE
seq1 maxvalue 25
INCREMENT BY
2 cycle
cache 2
drop SEQUENCE seq1;
```

EXAMPLE: 2

```
CREATE SEQUENCE seq2
INCREMENT BY
1 start with 1
maxvalue5
minvalue0
cycle
CACHE 4;
```

EXAMPLE: 3

CREATE SEQUENCE seq3
INCREMENT BY -
1 start with 2
maxvalue5
minvalue0;

EXAMPLE: 4

CREATE SEQUENCE seq3
INCREMENT BY -
1 start with 2
maxvalue5
minvalue0
cycle
cache 4;

EXAMPLE: 5

CREATE SEQUENCE seq1
INCREMENT BY
1 start with1
maxvalue10
minvalue 0;

EXAMPLE: 6

create table test1(a number primary key);

TO INSERT THE VALUES FROM SEQUENCES TO TABLE:

insert into test1 values(seq1.nextval)

TO DROP SEQUENCES

drop sequence sequenceNAme

RESULT:

Thus the SQL commands for creation and various operations on Sequence has been verified and executedsuccessfully.

4. Study of PL/SQL block.

Ex: No: 04

__:___: __

AIM:

To write a PL/SQL block using different control (if, if else, for loop, while loop,…) statements.

OBJECTIVE:

PL/SQL Control Structure provides conditional tests, loops, flow control and branches that let to produce well-structuredprograms

PL/SQL

PL/SQL is Oracle's procedural language extension to SQL. PL/SQL allows you to mix SQL statements with procedural statements like IF statement, Looping structuresetc.

It is extension of SQL the following or advantages of PL/SQL.

1. We can use programming features like if statement loopsetc.
2. PL/SQL helps in reducing networktraffic.
3. We can have user defined error massages by using concept of exceptionhandling.
4. We can perform related actions by using concept ofTriggers.
5. We can save the source code permanently for repeatedexecution.

PL/SQL Block:

A PL/SQL programs called as PL/SQL block.

PL/SQL Block:

```
DECLARE
                    Declaration of variable
                    Declarationofcursor----------    (OPTIONAL)
                    Declaration ofexception
BEGIN
                     Executablecommands---------    (MANDATORY)
EXCEPTION
                     Exceptionhandlers----------    (OPTIONAL)
END;
/                    To execute the program /command
```

Declare:

This section is used to declare local variables, cursors, Exceptions and etc. This section is optional.

Executable Section:

This section contains lines of code which is used to complete table. It is mandatory.

Exception Section:

This section contains lines of code which will be executed only when exception is raised. This section is optional.

Simplest PL/SQL Block:

```
Begin
------
------
------
END;
```

SERVEROUTPUT

This will be used to display the output of the PL/SQL programs. By default this will be off.

Syntax:

Set serveroutput on | off

Ex:

SQL> set serveroutput on

BLOCK TYPES

Anonymous blocks
Named blocks
Labeled blocks
Subprograms
Triggers

ANONYMOUS BLOCKS

Anonymous blocks implies basic block structure.

Ex:

Q : program to display the string “”

```
BEGIN
    DBMS_OUTPUT.PUT_LINE('My first program'):
END;
/
```

LABELED BLOCKS

Labeled blocks are anonymous blocks with a label which gives a name to the block.

Ex:

```
<<my_bloock>>

BEGIN
    Dbms_output.put_line('My first program'):
END;
```

SUBPROGRAMS

Subprograms are procedures and functions. They can be stored in the database as stand-alone objects, as part of package or as methods of an object type.

TRIGGERS

Triggers consist of a PL/SQL block that is associated with an event that occurs in the database.

NESTED BLOCKS

A block can be nested within the executable or exception section of an outer block.

IDENTIFIERS

Identifiers are used to name PL/SQL objects, such as variables, cursors, types and subprograms.

Identifiers consists of a letter, optionally followed by any sequence of characters, including letters, numbers, dollar signs, underscores, and pound signs only. The maximum length for an identifier is 30characters.

QUOTED IDENTIFIERS

If you want to make an identifier case sensitive, include characters such as spaces or use a reserved word, you can enclose the identifier in double quotationmarks.

Ex:

```
DECLARE
    "a" number := 5;
    "A" number := 6;
BEGIN
    dbms_output.put_line('a = ' || a);
    dbms_output.put_line('A = ' || A);
END;
/
```

Output:

```
a = 6
A = 6
```

COMMENTS

Comments improve readability and make your program more understandable. They are ignored by the PL/SQLcompiler.

There are two types of comments available.

Single line comments

Multiline comments

SINGLE LINE COMMENTS

A single-line comment can start any point on a line with two dashes and continues until the end of the line.

Ex:

```
BEGIN
    Dbms_output.put_line('hello');          -- sampleprogram
END;
```

/

MULTILINE COMMENTS

Multiline comments start with the /* delimiter and ends with */ delimiter.

Ex:

```
BEGIN
    Dbms_output.put_line('hello');                    /* sample program*/
END;
/
```

VARIABLE DECLERATIONS

Variables can be declared in declarative section of the block;

Ex:

```
DECLARE
    a number;
    b number := 5;
    c number default 6;
```

DECLARATIONS

To declare a constant, you include the CONSTANT keyword, and you must supply a default value.

Ex:

```
DECLARE
    b constant number := 5;
    c constant number default 6;
```

NOT NULL CLAUSE

You can also specify that the variable must be not null.

Ex:

```
DECLARE
    b constant number not null:=
    5; c number not null default 6;
```

ANCHORED DECLARATIONS

PL/SQL offers two kinds of anchoring.

Scalar anchoring

Record anchoring

SCALAR ANCHORING

Use the %TYPE attribute to define your variable based on table's column of some other PL/SQL scalar variable.

Ex:

DECLARE

dno **dept.deptno%type;**

Subtype t_number is number;

a t_number;

Subtype t_sno is **student.sno%type;**

V_sno t_sno;

RECORD ANCHORING

Use the %ROWTYPE attribute to define your record structure based on a table.

Ex:

DECLARE

V_dept **dept%rowtype;**

Benefits of Anchored Declarations

Synchronization with database columns.

Normalization of local variables.

PROGRAMMER-DEFINED TYPES

With the SUBTYPE statement, PL/SQL allows you to define your own subtypes or aliases of predefined datatypes, sometimes referred to as abstractdatatypes.

There are two kinds of subtypes.

Constrained

Unconstrained

CONSTRAINED SUBTYPE

A subtype that restricts or constrains the values normally allowd by the datatype itself.

Ex:

Subtype positive is binary_integer range 1..2147483647;

In the above declaration a variable that is declared as positive can store only ingeger greater than zero even though binary_integer ranges from-2147483647..+2147483647.

UNCONSTRAINED SUBTYPE

A subtype that does not restrict the values of the original datatype in variables declared with the subtype.

Ex:

Subtype float is number;

DATATYPE CONVERSIONS

PL/SQL can handle conversions between different families among the datatypes.

Conversion can be done in two ways.

Explicit conversion

Implicit conversion

EXPLICIT CONVERSION

This can be done using the built-in functions available.

IMPLICIT CONVERSION

PL/SQL will automatically convert between datatype families when possible.

Ex:

```
DECLARE
    a varchar(10);
BEGIN
    select deptno into a from dept where dname='ACCOUNTING';
END;
```

In the above variable a is char type and deptno is number type even though, oracle will automatically converts the numeric data into char type assigns to the variable.

PL/SQL can automatically convert between

Characters and numbers

Characters and dates

VARIABLE SCOPE AND VISIBILITY

The scope of a variable is the portion of the program in which the variable can be accessed. For PL/SQL variables, this is from the variable declaration until the end of the block. When a variable goes out of scope, the PL/SQL engine will free the memory used to store thevariable.

The visibility of a variable is the portion of the program where the variable can be accessed without having to qualify the reference. The visibility is always within the scope. If it is out of scope, it is not visible.

Ex1:

```
DECLARE
        a number; -- scope of a
BEGIN
------.
                DECLARE
                        b number; -- scope of b
                BEGIN
                ----.
                END;
----
END;
```

Ex2:

```
        DECLARE
                a number;
                b number;
        BEGIN
                -- a , b available here
                DECLARE
                        b char(10);
                BEGIN
                        -- a and char type b is available here
                END;
        ----.
        END;
```

Ex3:

```
<<my_block>>
DECLARE
        a number;
        b number;
BEGIN
        -- a , b available here
```

```
        DECLARE
            b char(10);
        BEGIN
        -- a and char type b is available here
        -- number type b is available using <<my_block>>.b
        END;
- - - -
END;
```

PL/SQL CONTROL STRUCTURES

PL/SQL has a variety of control structures that allow you to control the behaviour of the block as it runs. These structures include conditional statements andloops.

If-then-
else Case

- Case with noelse
- Labeledcase
- Searched

case Simpleloop

While loop
For loop

Goto and Labels

IF-THEN-ELSE

Syntax:

```
If <condition1> then
        Sequence of statements;
Elseif <condition1> then
        Sequence of statements;
        ……
Else
        Sequence of statements;
End if;
```

Ex:

```
DECLARE
    dno number(2);
BEGIN
    select deptno into dno from dept where dname =
        'ACCOUNTING'; if dno = 10 then
            dbms_output.put_line('Location is NEW
        YORK'); elseif dno = 20 then
            dbms_output.put_line('Location is
        DALLAS'); elseif dno = 30 then
            dbms_output.put_line('Location is CHICAGO');
```

```
        else
            dbms_output.put_line('Location is BOSTON');
        end if;
END;
```

Output:

Location is NEW YORK

CASE

Syntax:

Case ***test-variable***

When ***value1*** then *sequence ofstatements;*

When ***value2*** then *sequence ofstatements;*

.
.

When ***valuen*** then *sequence ofstatements;*

Else *sequence of statements;*

End case;

Ex:

```
DECLARE
    dno number(2);
BEGIN
    select deptno into dno from dept where dname =
    'ACCOUNTING'; case dno
        when 10 then
            dbms_output.put_line('Location is NEW
        YORK'); when 20 then
            dbms_output.put_line('Location is
        DALLAS'); when 30 then
            dbms_output.put_line('Location is CHICAGO');
        else
            dbms_output.put_line('Location is BOSTON');
    end case;
```

END;

Output:

Location is NEW YORK

CASE WITHOUT ELSE

Syntax:

Case ***test-variable***

When ***value-1*** then *sequence ofstatements;*

When ***value-2*** then *sequence ofstatements;*

……

When ***value-n*** then *sequence of statements;*

End case;

Ex:

```
DECLARE
    dno number(2);
BEGIN
    select deptno into dno from dept where dname =
    'ACCOUNTING'; case dno
        when 10 then
            dbms_output.put_line('Location is NEW
        YORK'); when 20 then
            dbms_output.put_line('Location is
        DALLAS'); when 30 then
            dbms_output.put_line('Location is
        CHICAGO'); when 40 then
            dbms_output.put_line('Location is BOSTON');
    end case;
END;
```

Output:

Location is NEW YORK

LABELED CASE

Syntax:

```
<<label>>
    Case test-variable
        When value1 then sequence of statements;
        When value2 then sequence of statements;
        ……
        When valuen then sequence of statements;
    End case;
```

Ex:

```
DECLARE
    dno number(2);
BEGIN
select deptno into dno from dept where dname=
        'ACCOUNTING';<<my_case>>
            case dno
                when 10 then
                    dbms_output.put_line('Location is NEW
                YORK'); when 20 then
                    dbms_output.put_line('Location is
                DALLAS'); when 30 then
                    dbms_output.put_line('Location is
                CHICAGO'); when 40 then
                    dbms_output.put_line('Location is BOSTON');
            end case
        my_case;
END;
```

Output:

Location is NEW YORK

SEARCHED CASE

Syntax:

Case

When <***condition-1***> then *sequence ofstatements;*

When <***condition-2***> then *sequence ofstatements;*

……

When <***condition-n***> then *sequence of statements;*

End case;

Ex:

```
DECLARE
    dno number(2);
BEGIN
    select deptno into dno from dept where dname = 'ACCOUNTING';
        case dno
            when dno = 10 then
                dbms_output.put_line('Location is NEW
            YORK'); when dno = 20 then
                dbms_output.put_line('Location is
            DALLAS'); when dno = 30 then
                dbms_output.put_line('Location is
            CHICAGO'); when dno = 40 then
dbms_output.put_line('Location is BOSTON');
        end case;
END;
```

Output:

Location is NEW YORK

SIMPLE LOOP

Syntax:

Loop

Sequence of statements;

Exit when <*condition*>;

End loop;

In the syntax exit when <*condition*> is equivalent to

If <*condition*> then

Exit;

End if;

Ex:

```
DECLARE
    i number := 1;
BEGIN
    loop
        dbms_output.put_line('i = ' ||
        i); i := i + 1;
        exit when i > 5;
    end loop;
END;
```

Output:

```
i = 1
i = 2
i = 3
i = 4
i = 5
```

WHILE LOOP

Syntax:

While <condition> loop

Sequence of statements;

End loop;

Ex:

```
DECLARE
    i number := 1;
BEGIN
    While i <= 5 loop
        dbms_output.put_line('i = ' || i);
        i := i + 1;
    end loop;
END;
```

Output:

i = 1

i = 2

i = 3

i = 4

i = 5

FOR LOOP

Syntax:

For <*loop_counter_variable*> in *low_bound..high_bound* loop

Sequence of statements;

End loop;

Ex1:

```
BEGIN
        For i in 1..5 loop
                dbms_output.put_line('i = ' || i);
        end loop;
END;
```

Output:

i = 1

i = 2

i = 3

i = 4

i = 5

Ex2:

```
BEGIN

        For i in reverse 1..5 loop

                dbms_output.put_line('i = ' || i);

        end loop;

END;
```

Output:

i = 5
i = 4
i = 3
i = 2
i = 1

GOTO AND LABELS

Syntax:

Goto *label*;

Where *label* is a label defined in the PL/SQL block. Labels are enclosed in double angle brackets.

When a goto statement is evaluated, control immediately passes to the statement identified by the label.

Ex:

```
BEGIN
    For i in 1..5 loop
        dbms_output.put_line('i = ' || i);
        if i = 4 then
            goto exit_loop;
        end if;
    end loop;
    <<exit_loop>>
    Null;
END;
```

Output:

```
i = 1
i = 2
i = 3
i = 4
```

Restrictions on GOTO

It is illegal to branch into an inner block, loop.

At least one executable statement must follow.

It is illegal to branch into an if statement.

It is illegal to branch from one if statement to another if statement. It is illegal to branch from exception block to the current block.

RESULT:

Thus the Study of PL/SQL block has been implemented by various control structure are verified and executed successfully.

Implicit and Explicit Cursors

Ex: No: 05

__:__: __

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.

ImplicitCursors

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.

Attributes such as

%FOUND,
%ISOPEN,
%NOTFOUND
%ROWCOUNT.

Example

```
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 will update the table and increase the salary of each customer by 500 and use the **SQL%ROWCOUNT** attribute to determine the number of rows affected −

```
DECLARE
   total_rows number(2);
BEGIN
   UPDATE customers
   SET salary = salary + 500;
   IF sql%notfound THEN
      dbms_output.put_line('no customers selected');
   ELSIF sql%found THEN
      total_rows :=sql%rowcount;
      dbms_output.put_line( total_rows || ' customers selected ');
   END IF;
END;
/
```

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

```
6 customers selected

PL/SQL procedure successfully completed.
```

ExplicitCursors

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 includes the following steps −

- Declaring the cursor for initializing thememory

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

- Opening the cursor for allocating thememory

```
OPEN c_customers;
```

- Fetching the cursor for retrieving thedata

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

- Closing the cursor to release the allocatedmemory

```
CLOSE c_customers;
```

EXAMPLE

```
DECLARE
   c_id customers.id%type;
   c_namecustomerS.No.ame%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;
      EXIT WHEN c_customers%notfound;
      dbms_output.put_line(c_id || ' ' || c_name || ' ' || c_addr);
   END LOOP;
   CLOSE c_customers;
END;
/When the above code is executed at the SQL prompt, it produces the following result –
1 RameshAhmedabad
2 KhilanDelhi
3 kaushikKota
4 ChaitaliMumbai
5 HardikBhopal
6 KomalMP

PL/SQL procedure successfully completed.
```

Creation of Procedures.

Ex: No: 06

__:__: __

AIM:

To write a PL/SQL block to display the student name, marks whose average mark is above 60%.

ALGORITHM:

STEP1: Start the program.

STEP2: Create a table with table name stud_exam

STEP3: Insert the values into the table and Calculate total and average of each student

STEP4: Execute the procedure function the student who get above 60%.

STEP5: Display the total and average of student

STEP6: Terminate the application

SYNTAX

CREATE [OR REPLACE] PROCEDURE name [(parameter[,parameter, ...])]

{IS|AS}

[local declarations]

BEGIN

executable statements

[EXCEPTION

exception handlers]

END [name];

EXECUTION:

SQL> SET SERVEROUTPUT ON

I) PROGRAM:

PROCEDURE USING POSITIONAL PARAMETERS:

SETTING SERVEROUTPUT ON:

```
SQL> SET SERVEROUTPUT ON

SQL> CREATE OR REPLACE PROCEDURE PROC1 AS
        BEGIN
                DBMS_OUTPUT.PUT_LINE('Hello from procedure...');
        END;
        /
```

Output:

```
Procedure created.

SQL> EXECUTE PROC1

        Hello from procedure...


PL/SQL procedure successfully completed.
```

II) PROGRAM:

Q: PROCEDURE USING NOTATIONAL PARAMETERS:

```
SQL> CREATE OR REPLACE PROCEDURE PROC2(N1 IN NUMBER,N2 IN NUMBER,TOT OUT NUMBER)
IS
        BEGIN
                TOT := N1 + N2;
        END;
        /
```

Output:

```
Procedure created.

SQL> VARIABLE T NUMBER
SQL> EXEC PROC2(33,66,:T)
PL/SQL procedure successfully completed.

SQL> PRINT T

        T

        -------

        99
```

PROCEDURE FOR GCD NUMBERS

III) PROGRAM:

```
SQL> create or replace procedure
        pro is
                a number(3);
                b number(3);
                c number(3);
                d number(3);
        begin
                a:=&a;
                b:=&b;
                        if(a>b) then
                                c:=mod(a,b);
                                if(c=0) then
                                        dbms_output.put_line('GCD is');
                                        dbms_output.put_line(b);
                                else
                                        dbms_output.put_line('GCD is');
                                        dbms_output.put_line(c);
                                end if;
                        else
                                d:=mod(b,a);
                                if(d=0) then
                                        dbms_output.put_line('GCD is');
                                        dbms_output.put_line(a);
                                else
                                        dbms_output.put_line('GCD is');
                                        dbms_output.put_line(d);
                                end if;
                        end if;
        end;
/
```

Out put:

Enter value for a:
8 old 8: a:=&a;
new 8: a:=8;

Enter value for b:
16 old 9: b:=&b;
new 9: b:=16;

Procedure created.

SQL> set serveroutput on;

SQL> execute pro;
GCD is
8

RESULT:

Thus the implementation of PL/SQL procedure has been verified and executed successfully.

Implementation of Triggers and its Application

Ex: No: 7

__:__: __

AIM:

To the Implementation of Triggers for the purpose of monitor the database object(table..etc) for any modification by the query user and/or the Application program.

ALGORITHM:

1. Start the program.
2. Create the table with its essentialattributes.
3. Insert attribute values into thetable.
4. Create the trigger for a particulartable
5. Specify when the trigger is to be fired - before orafter
6. Specify DML statement that invokes the trigger - UPDATE, DELETE, orINSERT
7. Specify the type of triger whether row-level trigger ornot
8. Condition to filterrows.
9. PL/SQL block that is to be executed when trigger isfired.
10. Modify the specified table to fire thetrigger.
11. Display the trigger message for the particular kind ofmodification
12. Stop theprogram.

DATABASE TRIGGERS

Triggers are similar to procedures or functions in that they are named PL/SQL blocks with declarative, executable, and exception handling sections. A trigger is executed implicitly whenever the triggering event happens. The act of executing a trigger is known as firing the trigger.

USE OF TRIGGERS

Maintaining complex integrity constraints not possible through declarative constraints enable at tablecreation.

Auditing information in a table by recording the changes made and who made them.

Automatically signaling other programs that action needs to take place when chages are made to atable.

Perform validation on changes being made to tables.

Automate maintenance of the database.

RESTRICTIONS ON TRIGGERES

Like packages, triggers must be stored as stand-alone objects in the database and cannot be local to a block orpackage.

A trigger does not accept arguments.

TYPES OF TRIGGERS

DML Triggers

Instead of Triggers

DDL Triggers

System Triggers

Suspend Triggers

CATEGORIES

Timing -- Before or After

Level -- Row or Statement

Row level trigger fires once for each row affected by the triggering statement. Row level trigger is identified by the FOR EACH ROWclause.

Statement level trigger fires once either before or after the statement.

TRIGGER SYNTAX

CREATE [OR REPLACE] TRIGGER

trigername {BEFORE | AFTER}

{DELETE | INSERT | UPDATE [OF columns]}

[OR {DELETE | INSERT |UPDATE [OF columns]}]...

ON table

[FOR EACH ROW [WHEN condition]]

[REFERENCING [OLD AS old] [NEW AS new]]

PL/SQL block

DML TRIGGERS

A DML trigger is fired on an INSERT, UPDATE, or DELETE operation on a database table. It can be fired either before or after the statement executes, and can be fired once per affected row, or once perstatement.

The combination of these factors determines the types of the triggers. These are a total of 12 possible types (3 statements * 2 timing * 2levels).

ORDER OF DML TRIGGER FIRING

Before statement level

Before row level

After rowlevel

After statement level

Ex:

Suppose we have a follwing table.

SQL> select * from student;

NO	NAME	MARKS
1	a	100
2	b	200
3	c	300
4	d	400

Also we have triggering_firing_order table with firing_order as the field.

Ex:

```
CREATE OR REPLACE TRIGGER
TRIGGER1 before insert on student
        BEGIN
                insert into trigger_firing_order values('Before Statement Level');
END TRIGGER1;
```

Ex:

```
CREATE OR REPLACE TRIGGER
TRIGGER2 before insert on student
for each row
        BEGIN
                insert into trigger_firing_order values('Before Row Level');
END TRIGGER2;
```

Ex:

```
CREATE OR REPLACE TRIGGER
TRIGGER3 after insert on student
        BEGIN
                insert into trigger_firing_order values('After Statement Level');
END TRIGGER3;
```

Ex:

```
CREATE OR REPLACE TRIGGER
TRIGGER4 after insert on student
for each row
        BEGIN
                insert into trigger_firing_order values('After Row Level');
END TRIGGER4;
```

Output:

```
SQL> select * from
trigger_firing_order; no rows selected
SQL> insert into student
values(5,'e',500); 1 row created.
```

SQL> select * from trigger_firing_order;

FIRING_ORDER

Before Statement
Level Before Row
Level After RowLevel
After Statement Level SQL>

select * from student;

NO	NAME	MARKS
1	a	100
2	b	200
3	c	300
4	d	400
5	e	500

Ex:

Suppose we have a table called marks with fields no, old_marks, new_marks.

```
CREATE OR REPLACE TRIGGER OLD_NEW
before insert or update or delete on student
for each row
        BEGIN
                insert into marks values(:old.no,:old.marks,:new.marks);
END OLD_NEW;
```

Output:

SQL> select * from student;

NO	NAME	MARKS
1	a	100
2	b	200
3	c	300
4	d	400
5	e	500

SQL> select * from
marks; no rows selected

SQL> insert into student
values(6,'f',600); 1 row created.

SQL> select * from student;

NO	NAME	MARKS
1	a	100
2	b	200
3	c	300
4	d	400
5	e	500
6	f	600

SQL> select * from marks;

NO	OLD_MARKS	NEW_MARKS
		600

SQL> update student set marks=555 where no=5; 1 row updated.

SQL> select * from student;

NO	NAME	MARKS
1	a	100
2	b	200
3	c	300
4	d	400
5	e	555
6	f	600

SQL> select * from marks;

NO	OLD_MARKS	NEW_MARKS
		600
5	500	555

SQL> delete student where no = 2; 1 row deleted.

SQL> select * from student;

NO	NAME	MARKS
1	a	100
3	c	300
4	d	400
5	e	555
6	f	600

SQL> select * from marks;

NO	OLD_MARKS	NEW_MARKS
		600
5	500	555
2	200	

REFERENCING CLAUSE

If desired, you can use the REFERENCING clause to specify a different name for :old ane :new. This clause is found after the triggering event, before the WHENclause.

Syntax:

REFERENCING [old as old_name] [new as new_name]

Ex:

```
CREATE OR REPLACE TRIGGER REFERENCE_TRIGGER
before insert or update or delete on student

referencing old as old_student new as
new_student for each row
    BEGIN
        insert into
        marks values(:old_student.no,:old_student.marks,:new_student.marks);
END REFERENCE_TRIGGER;
```

WHEN CLAUSE

WHEN clause is valid for row-level triggers only. If present, the trigger body will be executed only for those rows that meet the condition specified by the WHENclause.

Syntax:

WHEN *trigger_condition*;

Where *trigger_condition* is a Boolean expression. It will be evaluated for each row. The *:new* and *:old* records can be referenced inside *trigger_condition* as well, but like REFERENCING, the colon is not used there. The colon is only valid in the trigger body.

Ex:

```
CREATE OR REPLACE TRIGGERWHEN_TRIGGER
before insert or update or delete on student

referencing old as old_student new as
new_student for each row
    when (new_student.marks > 500)
    BEGIN
        insert into
        marks values(:old_student.no,:old_student.marks,:new_student.marks);
END WHEN_TRIGGER;
```

TRIGGER PREDICATES

There are three Boolean functions that you can use to determine what the operation is. The predicatesare

INSERTING

UPDATING

DELETING

Ex:

```
CREATE OR REPLACE TRIGGER PREDICATE_TRIGGER
before insert or update or delete on student
        BEGIN
        if inserting then
                insert into predicates
        values('I'); elsif updatingthen
                insert into predicates
        values('U'); elsif deletingthen
                insert into predicates values('D');
        end if;
END PREDICATE_TRIGGER;
```

Output:

```
SQL> delete student where
no=1; 1 row deleted.

SQL> select * from predicates;
MSG
----------
D

SQL> insert into student
values(7,'g',700); 1 row created.

SQL> select * from predicates;
MSG
----------
D
I
```

SQL> update student set marks = 777 where
no=7; 1 row updated.

SQL> select * from predicates;

MSG

- - - - - - - - - -

D

I

U

INSTEAD-OF TRIGGERS

Instead-of triggers fire instead of a DML operation. Also, instead-of triggers can be defined only on views. Instead-of triggers are used in two cases:

To allow a view that would otherwise not be modifiable to be modified. To modify the columns of a nested table column in aview.

RESULT:

Thus the Implementation of Triggers and its applications to monitor the modification in database has been verified and executed successfully.

IMPLEMENTATION OF FUNCTIONS AND ITS APPLICATION

Ex: No: 8

__:__: __

AIM:

To write the PL/SQL block for the implementation of functions and its application.

ALGORITHM:

STEP 1: Start the program.

STEP 2: Create the table with its essential attributes.

STEP 3: Insert attribute values into the table.

STEP 4: Create the function with necessary arguments and return data types.

STEP 5: create the PL/SQL block to call / use the function.

STEP 6: Execute the PL/SQL program.

STEP 7: Give the input values

STEP 8: Extract/process the information from the function.

STEP 9: Stop the program.

STORED FUNCTION

A function is similar to procedure, except that it returns a value. The calling program should use the value returned by the function.

CREATE FUNCTION

The **create function** command is used to create a stored function.

SYNTAX:

CREATE [OR REPLACE] FUNCTION **name**
[(parameter[,parameter, ...])]
RETURN datatype
{IS | AS}
[local declarations]
BEGIN
executable statements
RETURN value;
[EXCEPTION
exception handlers]
END [name];

Note:

OR REPLACE is used to create a function even though a function with the same name already exists

RETURN datatype specifies the type of data to be returned by the function.

RETURN statement in the executable part returns the value. The value must be of the same type as the return type specified using RETURN option in the header.

User-defined PL/SQL functions can be used in SQL in the same manner as the standard functions such as ROUND and SUBSTR

Q 1: To write a PL/SQL block to implementation of factorial using function

I) PROGRAM:

```
SQL>create function fnfact(n
        number) return number is
        b number;
                begin
                        b:=1;
                        for i in 1..n
                                loop
                                        b:=b*i;
                                end loop;
                        return b;
                end;
/

SQL>Declare
                n number:=&n;
                y number;
        begin
                dbms_output.put_line(y);
        end;
/
```

Output:

Function created.

Enter value for n: 5

old 2: n

number:=&n; **new 2:**

n number:=5; 120

PL/SQL procedure successfully completed.

Q2:create a function which count total no.of employees having salary less than 6000.

/*function body*/

```
Create or replace function count_emp(esal number)return number as
        Cursor vin_cur as Select empno,sal from emp;
        Xno emp.empno%type;
        Xsal emp.sal%type;
        C number;
Begin
        Open vin_cur;
        C:=0;
        loop
                fetch vin_cur into xno,xsal;
                        if(xsal<esal) then
                                c:=c+1;
                        end if;
                exit when
        vin_cur%notfound; end loop;
        close vin_cur;
        return c;
end;
/
```

Function created.

/*function specification*/

```
Declare
        Ne number;
        Xsal number;
Begin
        Ne:=count_emp(xsal);
        Dbms_output.put_line(xsal);
        Dbma_output.put_line('there are '||ne||;employees');
End;
/
```

OUTPUT

There are 8 employees.

Q2: To write a PL/SQL function to search an address from the given database

II) PROGRAM

```
SQL> create table phonebook (phone_no number (6) primary
        key, username varchar2(30),
        doorno varchar2(10),
        street varchar2(30),
        place varchar2(30),
        pincode char(6));
      table created.


SQL> insert into phonebook values(20312,'vijay','120/5D','bharathi
street','NGOcolony','629002'); 1 row created.


SQL> insert into phonebook values(29467,'vasanth','39D4','RK
bhavan','sarakkal vilai','629002'); 1 rowcreated.


SQL> select * from phonebook;
PHONE_NO  USERNAME        DOORNO     STREET         PLACE         PINCODE
--------  ----------      ---------  -------        -------       ------
20312     vijay           120/5D     bharathi street NGO colony   629002
29467     vasanth         39D4       RK bhavan      sarakkal vilai 629002


SQL> create or replace function findAddress(phone in number) return varchar2 as
        address varchar2(100);begin

              select username||','||doorno ||','||street ||','||place||','||pincode into address
              from phonebook wherephone_no=phone;

                    return address;
                    exception
              when no_data_found then return 'address not found';
        end;
/


Function created.



declare
```

```
        address varchar2(100);
begin
        address:=findaddress(20312);
        dbms_output.put_line(address);
end;
/
```

OUTPUT 1:

Vijay,120/5D,bharathi street,NGO colony,629002

PL/SQL procedure successfully completed.

```
declare
        address
        varchar2(100); begin
                address:=findaddress(23556);
                dbms_output.put_line(address);
        end;
/
```

OUTPUT2:

Address not found

PL/SQL procedure successfully completed.

Result:

Thus the implementation of functions and its applications has been executed successfully.

Write a PL/SQL block that handles all types of exceptions.

Ex: No: 09

__:__: __

AIM:

To Write a PL/SQL block that handles all types of exceptions.

ALGORITHM:

STEP1: Start the program.

STEP2: Create a table with some valid data.

STEP3: write the PL/SQL program that to handle the exception on exception block

STEP4: Execute the PL/SQL program and give the input values or make the error on the table data.

STEP5: Display the PL/SQL program error message.

STEP6: Stop the program.

EXCEPTIONS:

In PL/SQL, errors and warnings are called as exceptions. Whenever a predefined error occurs in the program, PL/SQL raises an exception. For example, if you try to divide a number by zero then PL/SQL raises an exception called ZERO_DIVIDE and if SELECT can not find a record then PL/SQL raises exception NO_DATA_FOUND.

PL/SQL has a collection of predefined exceptions. Each exception has a name. These exceptions are automatically raised by PL/SQL whenever the corresponding error occurs.

In addition to PL/SQL predefined exceptions, user can also create his own exceptions to deal with errors in the applications.

They are three types of Exceptions.

1. ORACLE PredefinedException
2. ORACLE Non PredefinedException
3. USER DefinedException

SYNTAX OF EXCEPTION HANDLING

```
WHEN exception-1 [or exception -2] ...
        THEN statements;
[WHEN exception-3 [or exception-4] ... THEN
        statements; ] ...
[WHEN OTHERS THEN
        statements; ]
```

Q: The following example handles **NO_DATA_FOUND** exception. declare

```
…
begin
    select
… exception
    when no_data_found
        then statements;
end;
```

Output:

Q: The following exception handling part takes the same action when either **NO_DATA_FOUND** or **TOO_MANY_ROWS** exceptions occur.

```
declare
    ...
begin
    select ...
exception
    when no_data_found or
        too_many_rows then
        statements;
end;
```

Output:

Q: The following snippet handles these two exceptions in different ways.

```
declare
        ...
begin
        select ...
exception
        when no_data_found
                then
        statements; when
        too_many_rows
                then statements;
end;
```

Output:

WHEN OTHERS is used to execute statements when an exception other than what are mentioned in exception handler has occurred.

```
declare
        newccode varchar2(5) := null;
begin
        update courses set ccode = newccode where ccode
= 'c'; exception
        when dup_val_on_index then
                dbms_output.put_line('Duplicate course
                code');
        when others then
                dbms_output.put_line(
                sqlerrm);
end;
```

Output:

Predefined exceptions

PL/SQL has defined certain common errors and given names to these errors, which are called as predefined exceptions.

Each exception has a corresponding Oracle error code. The following is the list of predefined exceptions and th

Exception	Oracle Error	SQLCODE Value
ACCESS_INTO_NULL	ORA-06530	-6530
COLLECTION_IS_NULL	ORA-06531	-6531
CURSOR_ALREADY_OPEN	ORA-06511	-6511
DUP_VAL_ON_INDEX	ORA-00001	-1
INVALID_CURSOR	ORA-01001	-1001
INVALID_NUMBER	ORA-01722	-1722
LOGIN_DENIED	ORA-01017	-1017
NO_DATA_FOUND	ORA-01403	+100
NOT_LOGGED_ON	ORA-01012	-1012
PROGRAM_ERROR	ORA-06501	-6501
ROWTYPE_MISMATCH	ORA-06504	-6504
SELF_IS_NULL	ORA-30625	-30625
STORAGE_ERROR	ORA-06500	-6500
SUBSCRIPT_BEYOND_COUNT	ORA-06533	-6533
SUBSCRIPT_OUTSIDE_LIMIT	ORA-06532	-6532
SYS_INVALID_ROWID	ORA-01410	-1410
TIMEOUT_ON_RESOURCE	ORA-00051	-51

NO_DATA_FOUND:

This Exception is Raised if a SELECT INTO statement returns no rows or if you reference an un-initialized row in a PL/SQL table.

Ex:

```
Declare
        L_sal emp.sal%type;
Begin
        DBMS_OUTPUT.PUT_LINE( 'WELCOME' );
        Select sal INTO L_sal from emp where empno =
        &empno; DBMS_OUTPUT.PUT_LINE(L_sal);
        DBMS_OUTPUT.PUT_LINE( 'THANK YOU' );
EXCEPTION
        when NO_DATA_FOUND then
                DBMS_OUTPUT.PUT_LINE( 'INVALID EMPNO');
END;
/
```

TOO_MANY_ROWS:

This Exception is Raised if a SELECT INTO statement returns more than one row.

Ex:

```
Declare
        L_sal emp.sal%type;
Begin
        DBMS_OUTPUT.PUT_LINE( 'WELCOME' );
        Select sal INTO L_sal from emp where deptno =
        30; DBMS_OUTPUT.PUT_LINE(L_sal);
        DBMS_OUTPUT.PUT_LINE( 'THANK YOU' );
EXCEPTION
        when TOO_MANY_ROWS then
                DBMS_OUTPUT.PUT_LINE( 'MORE THEN ONE ROW RETURNED');
END;
/
```

ZERO_DIVIDE:

Raised when your program attempts to divide a number by zero.

Ex:

```
Declare
        A Number;
```

```
Begin
        A :=
5/0; Exception
        when ZERO_DIVIDE then
                DBMS_OUTPUT.PUT_LINE( 'DO NOT DIVIDE BY 0' );
END;
/
```

Note:

This Exception is raised when we try to divided by zero.

VALUE_ERROR:

This Exception is raised when there is miss match with the value and data type of local variable or size of local variables.

Ex 1:

```
Declare
        L_sal emp.sal%type;
Begin
        DBMS_OUTPUT.PUT_LINE( 'WELCOME' );
        Select ename INTO L_sal from emp where empno =
        7521; DBMS_OUTPUT.PUT_LINE(L_sal);
        DBMS_OUTPUT.PUT_LINE( 'THANK YOU' );
EXCEPTION
        when VALUE_ERROR then
                DBMS_OUTPUT.PUT_LINE( 'please check the local variables');
END;
/
```

Ex 2:

```
Declare
        A number(3);
Begin
        A :=
1234; Exception
        when VALUE_ERROR then
                DBMS_OUTPUT.PUT_LINE( 'PLEASE CHECK THE LOCAL VARIABLES' );
END;
/
```

DUP_VAL_ON_INDEX: (duplicate value on index)

This Exception is raised when we try to insert a duplicate value in primary key constraint.

Ex:

```
Begin
        DBMS_OUTPUT.PUT_LINE( 'welcome' );
        Insert into student values(104, 'ARUN',60);
        DBMS_OUTPUT.PUT_LINE( 'Thank you' );
Exception
        when DUP_VAL_ON_INDEX then
                DBMS_OUTPUT.PUT_LINE(' Do not insert duplicates' );
END;
/
```

The above program works on an assumption the table student for if having a primary key SNO column with value 104.

WHEN OTHERS:

When others are a universal Exception angular this can catch all the Exceptions.Declare

```
        L_sal number(4);
        A number;
Begin
        DBMS_OUTPUT.PUT_LINE( 'Welcome' );
        Select sal INTO L_SAL from emp where deptno = &deptno;
        DBMS_OUTPUT.PUT_LINE('The sal is ….'||L_sal);
        A :=10/0;
        DBMS_OUTPUT.PUT_LINE( 'Thank you' );
Exception
        WHEN OTHERS THEN
                DBMS_OUTPUT.PUT_LINE( 'please check the code' );
END;
/
```

ERROR REPORTING FUNCTIONS:

They are two Error Reporting functions.

1. SQLCODE
2. SQLERRM

These error reporting functions are used in when others clause to identified the exception which is raised.

1. **SQLCODE**: It returnsERRORCODE

2. **SQLERRM**: It returns Exception number and Exceptionmessage.

Note: for NO_DATA_FOUND Exception SQLCODE will return 100.

```
Declare
        L_sal number(4);
        A number;
```

```
Begin
        DBMS_OUTPUT.PUT_LINE( 'Welcome' );
        Select sal INTO L_SAL from emp where deptno = &deptno;
        DBMS_OUTPUT.PUT_LINE('The sal is
        ….'||L_sal); A :=15/0;
        DBMS_OUTPUT.PUT_LINE( 'Thank you' );
Exception
        WHEN OTHERS THEN
                DBMS_OUTPUT.PUT_LINE( 'please check the code' );
                DBMS_OUTPUT.PUT_LINE(SQLCODE);
                DBMS_OUTPUT.PUT_LINE(SQLERRM);
END;
/
```

NESTED BLOCK:

```
Declare
        A number := 10;
Begin
        DBMS_OUTPUT.PUT_LINE('HELLO1');
                Declare
                        B number := 20;
                Begin
                        DBMS_OUTPUT.PUT_LINE('HELLO2');
                        DBMS_OUTPUT.PUT_LINE(B);
                        DBMS_OUTPUT.PUT_LINE(A);
                END;
        DBMS_OUTPUT.PUT_LINE('HELLO3');
        DBMS_OUTPUT.PUT_LINE(B); --ERROR
END;
/
```

Note:

outer block variables can be accessed in nested block nested block variables can not be accessed in outer block.

EXCEPTION PROPAGATION:

```
Begin
        DBMS_OUTPUT.PUT_LINE('HELLO1');
        L_SAL EMP.SAL%TYPE;
        Begin
```

```
        DBMS_OUTPUT.PUT_LINE('HELLO2');
        Select sal INTO L_SAL from emp where empno =
        1111;DBMS_OUTPUT.PUT_LINE('HELLO3');
    END;
    DBMS_OUTPUT.PUT_LINE('HELLO4');
EXCEPTION
    WHEN NO_DATA_FOUND THEN
        DBMS_OUTPUT.PUT_LINE('HELLO5');
END;
/
```

ORALE NON PREDEFINEDEXCEPTIONS:

These Exceptions will have only Exception number. But does not have Exception name. Steps to handle non predefinedexceptions.

Syntax:

Step1: Declare the Exception

```
        <EXCEPTION_NAME> EXCEPTION;
```

Step2: Associate the Exception

```
        PRAGMA EXCEPTION_INIT(<EXCEPTION_NAME>,<EXCEPTION
```

NO>); Step3: Catch the Exception

```
        WHEN <EXCEPTION_NAME> THEN
    ----------
    ----------
    ----------
END;
/
```

ORA -2292 is an example of non predefined exception.

This exception is raised when we delete row from a parent table. If the corresponding value existing the childtable.

```
Declare
    MY_EX1 Exception; --step1
    PRAGMA EXCEPTION_INIT(MY_EX1, -2292); --step2
Begin
    DBMS_OUTPUT.PUT_LINE('Welcome');
    Select from student where eno = 102;
EXCEPTION
    WHEN MY_EX1 THEN --step3
    DBMS_OUTPUT.PUT_LINE('do not delete pargma table');
```

```
END;
/
```

Pragma Exception_init is a compiler directive which is used to associated an Exception name to the predefined number.

USER DEFINED EXCEPTION:

These Exceptions are defined and controlled by the user. These Exceptions neither have predefined name nor have predefined number. Steps to handle user definedExceptions.

Step1: Declare the Exception

```
declare
        out_of_stock exception;
begin
        statements;
end;
```

Step2: Raised the Exception

```
if qty < 10 then
    raise out_of_stock;
end if;
```

Step3: Catch the Exception

```
exception
    when out_of_stock then
        -- handle the exception (that is reraised) in outer block
        ...
end;
```

Ex

```
Declare
    MY_EX1 EXCEPTION; --Step1
    L_SAL EMP.SAL%TYPE;
Begin
    DBMS_OUTPUT.PUT_LINE('welcome');
    Select SAL INTO L_SAL from emp where empno =
    &empno; IF L_SAL > 2000 THEN
        RAISE MY_EX1; --Step2
    ENDIF;
    DBMS_OUTPUT.PUT_LINE('The sal is … '||L_sal);
    DBMS_OUTPUT.PUT_LINE('Thank you');
EXCEPTION
    WHEN MY_EX1 THEN --Step3
    DBMS_OUTPUT.PUT_LINE('Sal is two high');
END;
/
```

Note: When others should be the last handler of the exception section other wise we get a compiler ERROR.

RAISE_APPLICATION_ERROR:

RAISE_APPLICATION_ERROR is a procedure which is used to throw one error number and error message to the callingenvironment.

It internally performance rolls back.

ERROR number should be range of -20000 to -20999. ERROR message should be displayed less then or equal to 512characters.

```
Declare
    L_sal emp.sal%TYPE;
Begin
    DBMS_OUTPUT.PUT_LINE('Welcome');
    Insert INTO dept values (08,'arun',70);
    Select sal INTO L_sal from emp where empno =
        7698; IF L_sal > 2000 THEN
            RAISE_APPLICATION_ERROR(-20150, 'SAL IS TOO HIGH');
        END IF;
    DBMS_OUTPUT.PUT_LINE('THE SAL IS…'||L_SAL);
END;
/
```

.

RESULT:

Thus the PL/SQL block that handles all types of exceptions has been verified and executed successfully

DATABASE DESIGN USING ER MODELING, NORMALIZATION AND IMPLEMENTATION FOR ANY APPLICATION

EX: NO: 10

--/--/---

The waterfall model can be applied to database design. The steps can be summarized as follows:

Requirements specification -> Analysis -> Conceptual design -> Implementation Design-> Physical Schema Design and Optimisation

Figure 1: A basic example of a requirements document

The business is a sweet factory called 'Sweets R Us'. The business buys raw ingredients from several suppliers and keeps a monthly record of the purchases. Sweets R Us has several employees who prepare the ingredients or make sweets in three of the company's departments. The products are sold to a number of retail stores, and the sales are recorded in an inventory on a monthly basis.

The company would like to keep track of purchases of raw ingredients and their suppliers, as well as employees and their wages. There are 20 employees in the company, and 3 departments. The employees specialize in different area of the sweet manufacturing process and have different job descriptions. The company sells their products to a number of different retailers in various cities. They would like to keep track of this information.

As far as keeping track of sales and purchases is concerned, the company would like to keep track of the value and date of sales and purchases as well who bought or sold what to them.

The company is loosing a lot of money and they would like to be able to see where costs could be cut. One example would be to examine how much they could save by reducing the highest paid employees wages, with the idea that they could let those employees go, and replace them with lower skilled employees.

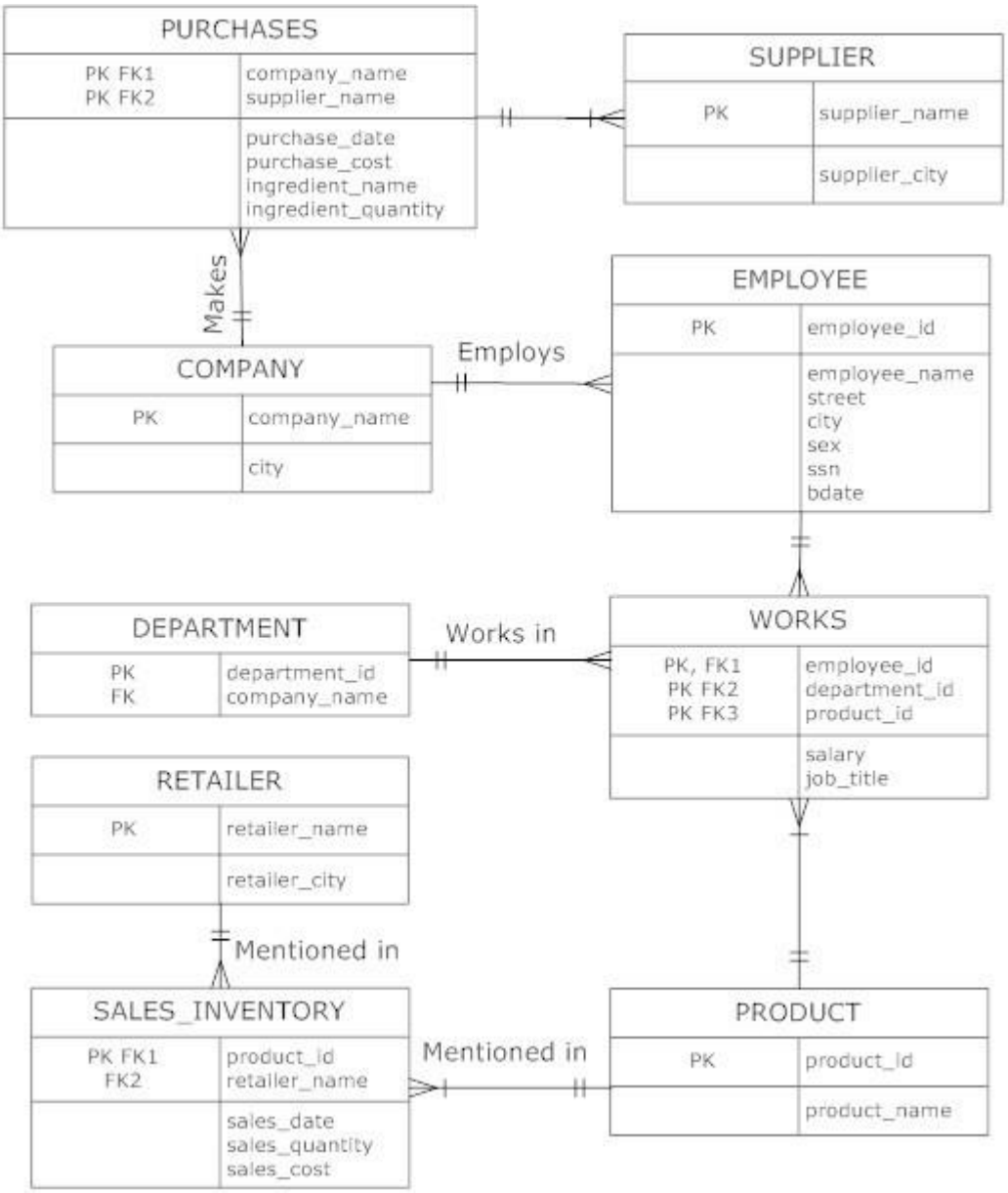
They would also like to keep track of where employees live with the aim of letting go of employees who live too far away. Another request is to be able to locate retailers from various cities so they can reduce costs by having one single delivery system to each city, rather than have multiple deliveries there.

The requirements document can then be analyzed and turned into a basic data set (as shown in Figure 2) which can be converted into a conceptual model. The result of the conceptual design phase is a conceptual data model (Figure 3), which provides little information about how the database system will eventually be implemented. The conceptual data model is merely a high-level overview of the database system.

Figure 2: A Database Data Set is the Result of analyzing the Information from the Requirements Phase. ThePrimary Keys are Underlined.

COMPANY (company_name, city);
PURCHASES (company_name, supplier_name, purchase_date, purchase_cost, ingredient_name, ingredient_quantity)
SUPPLIER (supplier_name, supplier_city)
EMPLOYEE (employee_id, employee_name, street, city, sex, ssn, bdate)
WORKS (employee_id, department_id, product_id, job_title, salary)
DEPARTMENT (department_id, company_name)
PRODUCT (product_id, product_name)
SALES (product_id, retailer_name, sales_date, sales_quantity, sales_cost)
RETAILER (retailer_name, retailer_city)

Figure 3: A Normalized Entity-Relationship model (ERD) in Crow's Foot Notation is an Example of aConceptual Data Model and provides no information of how the database system will eventually beimplemented



In the implementation design phase, the conceptual data model is translated into a 'logical' representation of the database system. The logical data model conveys the "logical functioning and structure" of the database and describes 'how the data is stored' (e.g., what tables are used, what constraints are applied) but is not specific to any DBMS. The logical database model is a lower-level conceptual model, which must be translated to a physical design.

Figure 4: In the implementation design phase, the conceptual data model (ERD) is translated into a 'logical'representation (logical schema) of the database system: a data dictionary.

Snapshot of Data Dictionary

	A	B	C	D	E	F	G	H	I
1	TABLE NAME	ATTRIBUTE NAME	CONTENTS	TYPE	FORMAT	RANGE	REQUIRED	PK/FK	REFERENCE
2	COMPANY	company_name	Company Name	varchar2 (19)	Xxxxxxx		Y	PK	
		city	City where the company is based	varchar2 (19)	Xxxxxxx				
3	PURCHASES	company_name	Company Name	varchar2 (19)	Xxxxxxx			PK FK	COMPANY
		supplier_name	Supplier Company Name	varchar2 (19)	Xxxxxxx	-		PK FK	SUPPLIER
		purchase_date	Date when the purchase was made	date	dd-mm-yy	-			
		purchase_cost	How much the order cost	number (6,2)	9999	1.00 - 9999.99			
		ingredient_name	What was bought	varchar2 (19)	Xxxxxxx	-			
		ingredient_quantit	How much was bought	number (3,0)	999	1 - 999			
4	SUPPLIER	supplier_name	Supplier Company Name	varchar2 (19)	Xxxxxxx		Y	PK	
		supplier_city	City where the supplier is based	varchar2 (19)	Xxxxxxx				
5	EMPLOYEE	employee_id	Employee id number	char (4) not null	999999999	1111 - 9999	Y	PK	
		employee_name	Full name of employee	varchar2 (19)	Xxxxxxx	-			
		street	Employees street address	varchar2 (19)	Xxxxxxx	-			
		city	City where employee lives	varchar2 (19)	Xxxxxxx	-			
		sex	Gender of employee	char (3)	X	-			
		ssn	Social security number	char (5)	999999999	11111 - 99999			
		bdate	Date of birth	date	dd-mm-yy				
6	WORKS	employee_id	Employee id number	char (4) not null	999999999	1111 - 9999	Y	PK FK	EMPLOYEE
		job_title	Title of job	varchar2 (19)	Xxxxxxx	-		-	-
		department_id	Id of department where he works	char (3)	999	111 - 999		PK FK	DEPARTMEI
		product_id	Id number of product he makes	char (5)	99999	11111 - 99999		PK FK	PRODUCT
		salary	Annual salary	number (8,2)	999999.99	0.00 - 999, 999.99			
7	DEPARTMENT	department_id	Id of department	char (3)	999	111 - 999	Y	PK	
		company_name	Name of company	varchar2 (19)	Xxxxxxx				
8	PRODUCT	product_id	Final Product Id	char (5)	99999		Y	PK	
		product_name	Name of final product	varchar2 (19)	Xxxxxxx	11111 - 99999			
9	SALES	product_id	Final Product Id	char (5)	99999	11111 - 99999		PK FK	PRODUCT
		retailer_name	Name of company that bought it	varchar2 (19)	Xxxxxxx	-		PK FK	RETAILER
		sales_date	Date when product was sold	date	dd-mm-yy	-			
		sales_quantity	How much was sold to retailer	number (4,0)	9999	1 - 9999			
		sales_cost	Total amount charged	number (6,2)	9999.99	0.00 - 9999.99			
10	RETAILER	retailer_name	Retailer name	varchar2 (19)	Xxxxxxx		Y	PK	
		retailer_city	Retailer's city	varchar2 (19)	Xxxxxxx				

SQL Statements – Implementing the Database

The final step is to physically implement the logical design which was illustrated in Figure 4. To physically implement the database, SQL can be used. These are the main steps in implementing the database:

1. Create the DatabaseTables

The tables come directly from the information contained in the Data Dictionary. The following blocks of code each represent a row in the data dictionary and are executed one after another. The blocks of "create table" code contain the details of all the data items (COMPANY, SUPPLIER, PURCHASES, EMPLOYEE, etc), their attributes (names, ages, costs, numbers, and other details), the Relationships between the data items, the Keys and Data Integrity Rules. All of this information is already detailed in the Data Dictionary, but now we are converting it and implementing it in a physical database system.

```
create table COMPANY (
company_name varchar2(19) not null,
city varchar2(19),
CONSTRAINT COMPANY_company_name_pk PRIMARY KEY(company_name)
);
create table SUPPLIER (
supplier_name varchar2(19) not null,
city varchar2(19),
CONSTRAINT SUPPLIER_supplier_name_pk PRIMARY KEY(supplier_name)
);
create table PURCHASES (
company_name varchar2(19) not null,
supplier_name varchar2(19) not null,
purchase_date date,
purchase_cost number (6,2),
ingredient_name varchar2 (19),
ingredient_quantity number (3,0),
CONSTRAINT PURCHASES_company_name_fk FOREIGN KEY(company_name) REFERENCES
COMPANY(company_name),
CONSTRAINT PURCHASES_supplier_name_fk FOREIGN KEY(supplier_name) REFERENCES
SUPPLIER(supplier_name),
CONSTRAINT PURCHASES_company_name_pk PRIMARY KEY(company_name,
supplier_name)
);
```

```
create table EMPLOYEE (
employee_id char(4) not null,
employee_name  varchar2(19),
street varchar2 (19),
city  varchar2 (19),
sex char (3),
ssn char (5),
bdate date,
CONSTRAINT EMPLOYEE_employee_id_pk PRIMARY KEY(employee_id)
);
create table DEPARTMENT (
department_id  char (3) not null,
company_name varchar2 (19),
CONSTRAINT DEPARTMENT_department_id_pk PRIMARY KEY(department_id),
CONSTRAINT DEPARTMENT_company_fk FOREIGN KEY(company_name) REFERENCES
COMPANY(company_name)
);
create table WORKS (
employee_id  char (4),
job_title varchar2 (19),
department_id  char (3),
product_id  char (5),
salary number (8,2),
CONSTRAINT WORKS_employee_id_fk FOREIGN KEY(employee_id) REFERENCES
EMPLOYEE(employee_id),
CONSTRAINT WORKS_department_id_fk FOREIGN KEY(department_id) REFERENCES
DEPARTMENT(department_id),
CONSTRAINT WORKS_product_id_fk FOREIGN KEY(product_id) REFERENCES
PRODUCT(product_id),
CONSTRAINT WORKS_product_id_pk PRIMARY KEY(employee_id, department_id,
product_id)
);
create table RETAILER (
retailer_name varchar2 (19) not null,
retailer_city varchar2 (19),
CONSTRAINT RETAILER_retailer_name_pk PRIMARY KEY(retailer_name)
);
create table SALES (
product_id  char (5) not null,
retailer_name varchar2 (19) not null,
sales_date date,
sales_quantity number (4,0),
sales_cost number (6,2),
CONSTRAINT SALES_product_id_fk FOREIGN KEY(product_id) REFERENCES
PRODUCT(product_id),
CONSTRAINT SALES_retailer_name_fk FOREIGN KEY(retailer_name) REFERENCES
RETAILER(retailer_name),
CONSTRAINT SALES_product_id_pk PRIMARY KEY(product_id, retailer_name)
);
```

2. Populate thetables

Use SQL statements to populate each table with specific data (such as employee names, ages, wages etc).

3. Query thedatabase.

Write SQL statements to obtain information and knowledge about the company, e.g. how many employees are there, total profit etc.

RESULT:

Thus Database Design using ER modeling, normalization and Implementation for a sweet company was carried out successfully.

EX: NO: 11

--/--/--- **DATABASE CONNECTIVITY WITH FRONT ENDTOOLS**

Aim

To Connect Oracle Database from Visual Basic 6.0

```
Dim conn As ADODB.Connection

' Open a Conn_Dataection using Oracle ODBC.
Set Conn_Data = New ADODB.Connection
Conn_Data.ConnectionString    =    "Driver={Microsoft    ODBC    for    Oracle};"
                              &"UID=user_name;PWD=user_passsword"
Conn_Data.Open

'Open the table as in:

Dim rs_Data As ADODB.Recordset

' Open the table.
Set rs_Data = New ADODB.Recordset
rs_Data.Open "TableName", Conn_Data, adOpenDynamic, adLockOptimistic, adCmdTable

'Enter the user name password and table name as per the database.
'it must be valid one.

'To reads the data from the table and displays the values in a ListBox

' List the data.
Do While Not rs_Data.EOF
    txt = ""
    For Each fld In rs_Data.Fields
        txt = txt & Trim$(fld.Value) & ", "
    Next fld
    If Len(txt) > 0 Then txt = Left$(txt, Len(txt) - 2)
    List1.AddItem txt
    rs_Data.MoveNext
Loop

'Finally close the recordset and close the Conn_Dataection:
rs_Data.Close
Conn_Data.Close
```

RESULT:

Thus Oracle Database was connected successfully from Visual Basic .

Ex.No. 12a INVENTORY CONTROL SYSTEM

AIM:

Create a Inventory Control System in VB and connect the corresponding Table from SQP-PLUS

PROCEDURE:

1. Start the process.
2. Create the table using oracle,SQL and insert some tuples in it.
3. Commit the table and exit.
4. For ODBC connectivity, select control panel -> administrative tools->ODBC data sources, select drivers menu, select Microsoft ODBC driver, click ok.
5. In VB 6.0, choose project menu, under that references.
6. Select Microsoft DAO 3.51 object library and Microsoft DAO 3.6 object library
7. Create forms for the project for the appropriate requirements specified.
8. Execute and terminate the project.

CODING:

GENERAL:

```
Dim con As New ADODB.Connection
Dim res As New ADODB.Recordset
```

VIEW:

```
Private Sub Command1_Click()
con.ConnectionString = "UID=system;PWD=manager;DRIVER={Microsoft ODBC for Oracle};" _
& "SERVER=csedb;"
con.Open
res.Open "select * from  invent ", con, adOpenDynamic, adLockOptimistic
res.MoveFirst
With res
Text1.Text = res(0)
Text2.Text = res(1)
Text3.Text = res(2)
End With
res.Close
con.Close
End Sub
```

INSERT:

```
Private Sub Command2_Click()
con.ConnectionString = "UID=system;PWD=manager;DRIVER={Microsoft ODBC for Oracle};" _
& "SERVER=csedb;"
con.Open
 res.Open "select * from invent ", con, adOpenDynamic, adLockOptimistic
res.AddNew
res(0) = Val(Text1.Text)
res(1) = Val(Text2.Text)
res(2) = Val(Text3.Text)
res.Update
```

MsgBox "1 row inserted"
res.Close
con.Close
End Sub

CLEAR:

Private Sub Command3_Click()
Text1.Text = ""
Text2.Text = ""
Text3.Text = ""
End Sub

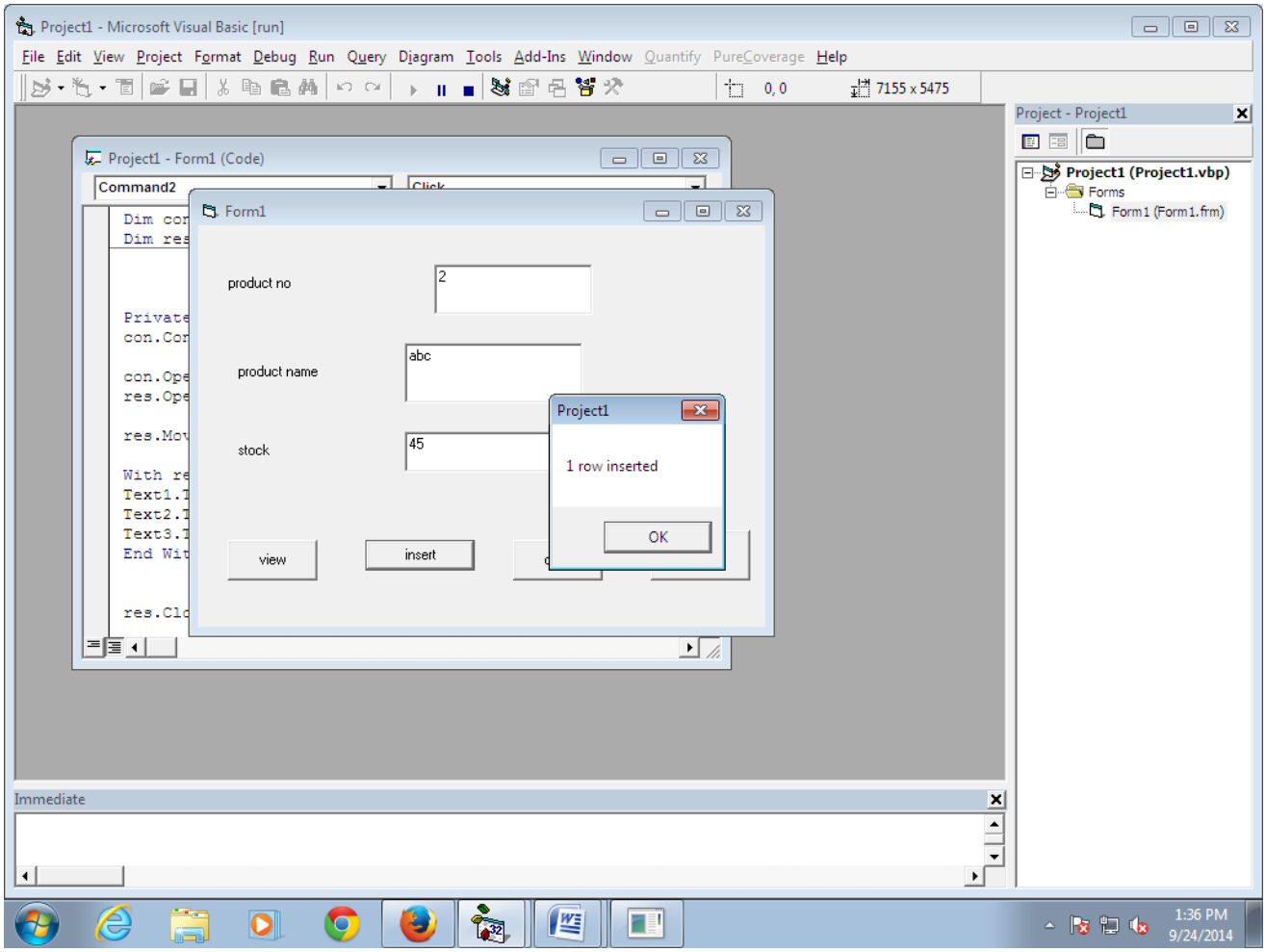
CLOSE:

Private Sub Command4_Click()
End
End Sub

OUTPUT:

SQL> select * from invent;

PRNAME	PRNO	STOCK
abc	245	



RESULT:

Thus the Inventory System is designed in VB and connected to SQL-PLUS with proper ADODB Connection

Ex.No. 12b MATERIAL REQUIREMENT PROCESSING

AIM:

Create a Material Requirement System in VB and connect the corresponding Table from SQP-PLUS

PROCEDURE:

1. Start the process.
2. Create the table using oracle,SQL and insert some tuples in it.
3. Commit the table and exit.
4. For ODBC connectivity, select control panel -> administrative tools->ODBC data sources, select drivers menu, select Microsoft ODBC driver, click ok.
5. In VB 6.0, choose project menu, under that references.
6. Select Microsoft DAO 3.51 object library and Microsoft DAO 3.6 object library
7. Create forms for the project for the appropriate requirements specified.
8. Execute and terminate the project.

CODING:

GENERAL:

```
Dim con As New ADODB.Connection
Dim res As New ADODB.Recordset
```

VIEW:

```
Private Sub Command1_Click()
con.ConnectionString = "UID=system;PWD=manager;DRIVER={Microsoft ODBC for Oracle};" _
& "SERVER=csedb;"
con.Open
res.Open "select * from  material ", con, adOpenDynamic, adLockOptimistic
res.MoveFirst
With res
Text1.Text = res(0)
Text2.Text = res(1)
Text3.Text = res(2)
End With
res.Close
con.Close
End Sub
```

INSERT:

```
Private Sub Command2_Click()
con.ConnectionString = "UID=system;PWD=manager;DRIVER={Microsoft ODBC for Oracle};" _
& "SERVER=csedb;"
con.Open
 res.Open "select * from material ", con, adOpenDynamic, adLockOptimistic
res.AddNew
res(0) = Val(Text1.Text)
res(1) = Val(Text2.Text)
res(2) = Val(Text3.Text)
res.Update
```

MsgBox "1 row inserted"
res.Close
con.Close
End Sub

CLEAR:

Private Sub Command3_Click()
Text1.Text = ""
Text2.Text = ""
Text3.Text = ""
End Sub

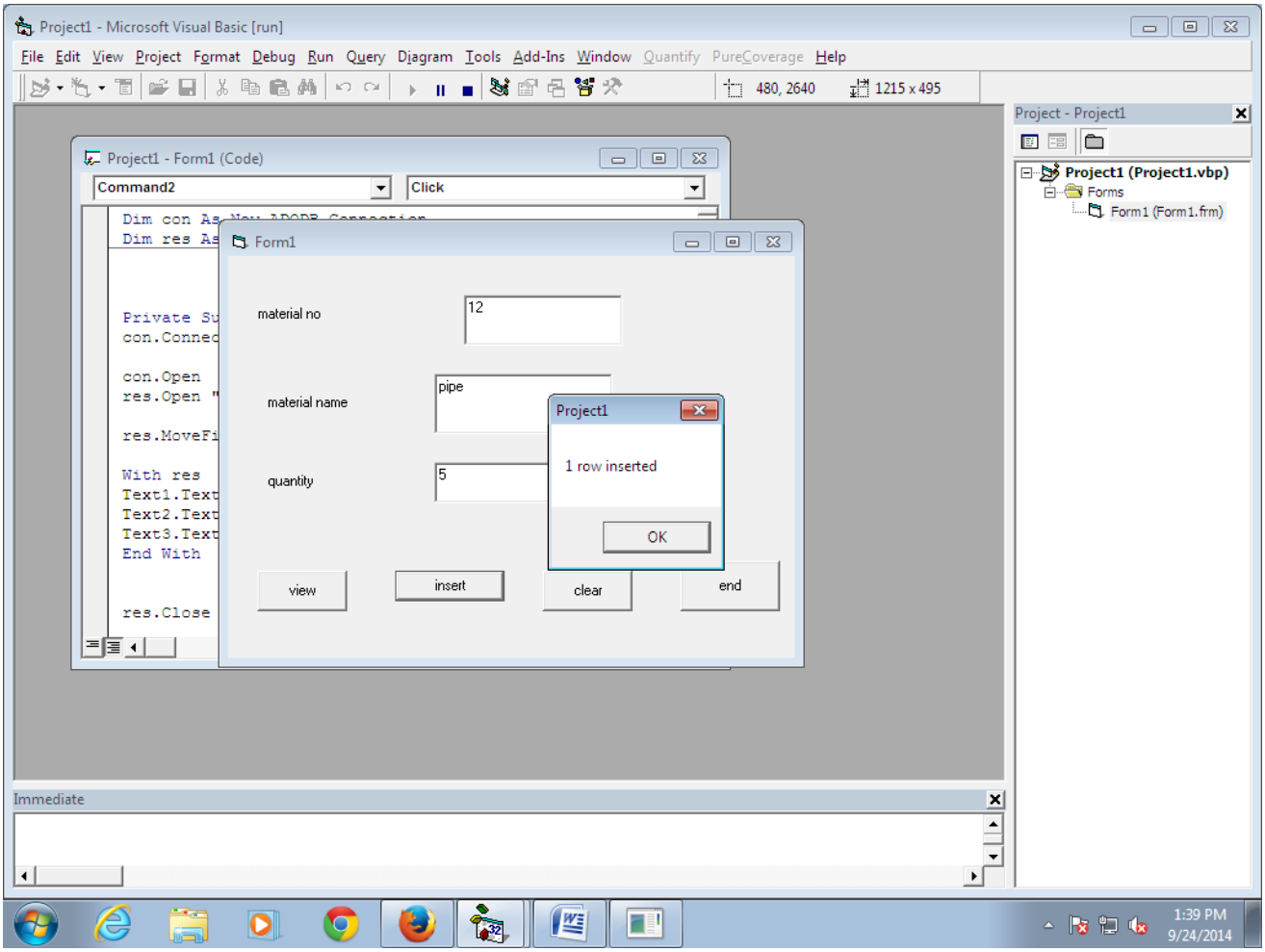
CLOSE:

Private Sub Command4_Click()
End
End Sub

OUTPUT:

SQL> select * from material;

MRNAME	MRNO	QUAN
pipe	125	



RESULT:

Thus the Material Requirement System is designed in VB and executed with the support of SQL-PLUS with proper ADODB Connection

Ex.No. 12c HOSPITAL MANAGEMENT SYSTEM

AIM:

Create a Hospital Management System in VB and connect the corresponding Table from SQP-PLUS

PROCEDURE:

1. Start the process.
2. Create the table using oracle,SQL and insert some tuples in it.
3. Commit the table and exit.
4. For ODBC connectivity, select control panel -> administrative tools->ODBC data sources, select drivers menu, select Microsoft ODBC driver, click ok.
5. In VB 6.0, choose project menu, under that references.
6. Select Microsoft DAO 3.51 object library and Microsoft DAO 3.6 object library
7. Create forms for the project for the appropriate requirements specified.
8. Execute and terminate the project.

CODING:

GENERAL:

```
Dim con As New ADODB.Connection
Dim res As New ADODB.Recordset
```

VIEW:

```
Private Sub Command1_Click()
con.ConnectionString = "UID=system;PWD=manager;DRIVER={Microsoft ODBC for Oracle};" _
& "SERVER=csedb;"
con.Open
res.Open "select * from  hosp ", con, adOpenDynamic, adLockOptimistic
res.MoveFirst
With res
Text1.Text = res(0)
Text2.Text = res(1)
Text3.Text = res(2)
End With
res.Close
con.Close
End Sub
```

INSERT:

```
Private Sub Command2_Click()
con.ConnectionString = "UID=system;PWD=manager;DRIVER={Microsoft ODBC for Oracle};" _
& "SERVER=csedb;"
con.Open
 res.Open "select * from hosp ", con, adOpenDynamic, adLockOptimistic
res.AddNew
```

```
res(0) = Val(Text1.Text)
res(1) = Val(Text2.Text)
res(2) = Val(Text3.Text)
res.Update
MsgBox "1 row inserted"
res.Close
con.Close
End Sub
```

CLEAR:

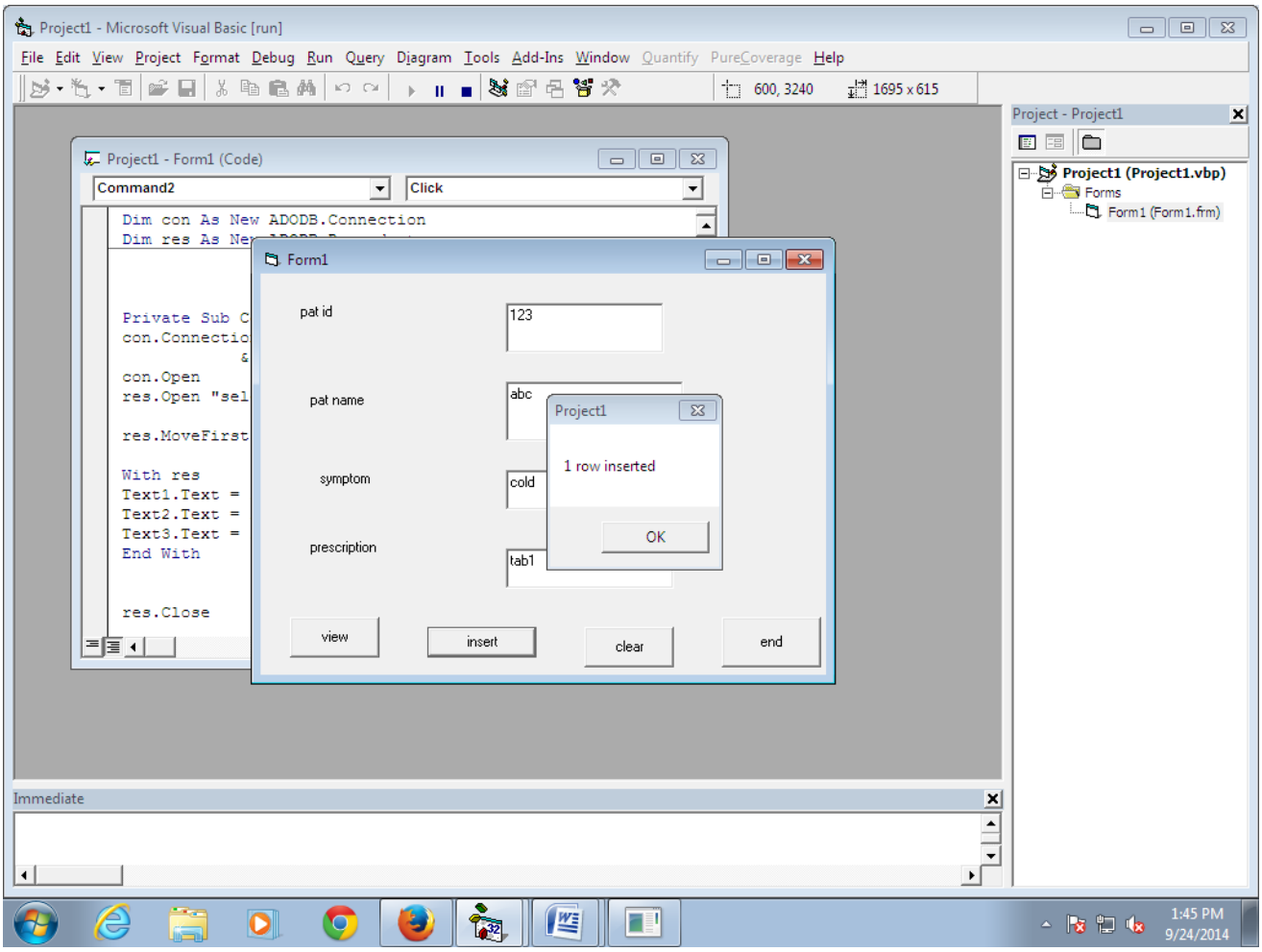
```
Private Sub Command3_Click()
Text1.Text = ""
Text2.Text = ""
Text3.Text = ""
End Sub
```

CLOSE:

```
Private Sub Command4_Click()
End
End Sub
```

OUTPUT:

```
SQL> select * from hosp;
PID     PNAME    SYMPTOM PRESCRIPTION
------  ----------  ----------  ----------
123abccold     Tab1
```



RESULT:

Thus the Hospital Management System is designed in VB and executed with the support of SQL-PLUS with proper ADODB Connection

Ex.No. 12d RAILWAY RESERVATION SYSTEM

AIM:

Create a Railway Reservation System in VB and connect the corresponding Table from SQP-PLUS

PROCEDURE:

1. Start the process.
2. Create the table using oracle,SQL and insert some tuples in it.
3. Commit the table and exit.
4. For ODBC connectivity, select control panel -> administrative tools->ODBC data sources, select drivers menu, select Microsoft ODBC driver, click ok.
5. In VB 6.0, choose project menu, under that references.
6. Select Microsoft DAO 3.51 object library and Microsoft DAO 3.6 object library
7. Create forms for the project for the appropriate requirements specified.
8. Execute and terminate the project.

CODING:

GENERAL:

```
Dim con As New ADODB.Connection
Dim res As New ADODB.Recordset
```

VIEW:

```
Private Sub Command1_Click()
con.ConnectionString = "UID=system;PWD=manager;DRIVER={Microsoft ODBC for Oracle};" _
& "SERVER=csedb;"
con.Open
res.Open "select * from  rail ", con, adOpenDynamic, adLockOptimistic
res.MoveFirst
With res
Text1.Text = res(0)
Text2.Text = res(1)
Text3.Text = res(2)
End With
res.Close
con.Close
End Sub
```

INSERT:

```
Private Sub Command2_Click()
con.ConnectionString = "UID=system;PWD=manager;DRIVER={Microsoft ODBC for Oracle};" _
& "SERVER=csedb;"
con.Open
 res.Open "select * from rail ", con, adOpenDynamic, adLockOptimistic
res.AddNew
res(0) = Val(Text1.Text)
```

```
res(1) = Val(Text2.Text)
res(2) = Val(Text3.Text)
res.Update
MsgBox "1 row inserted"
res.Close
con.Close
End Sub
```

CLEAR:

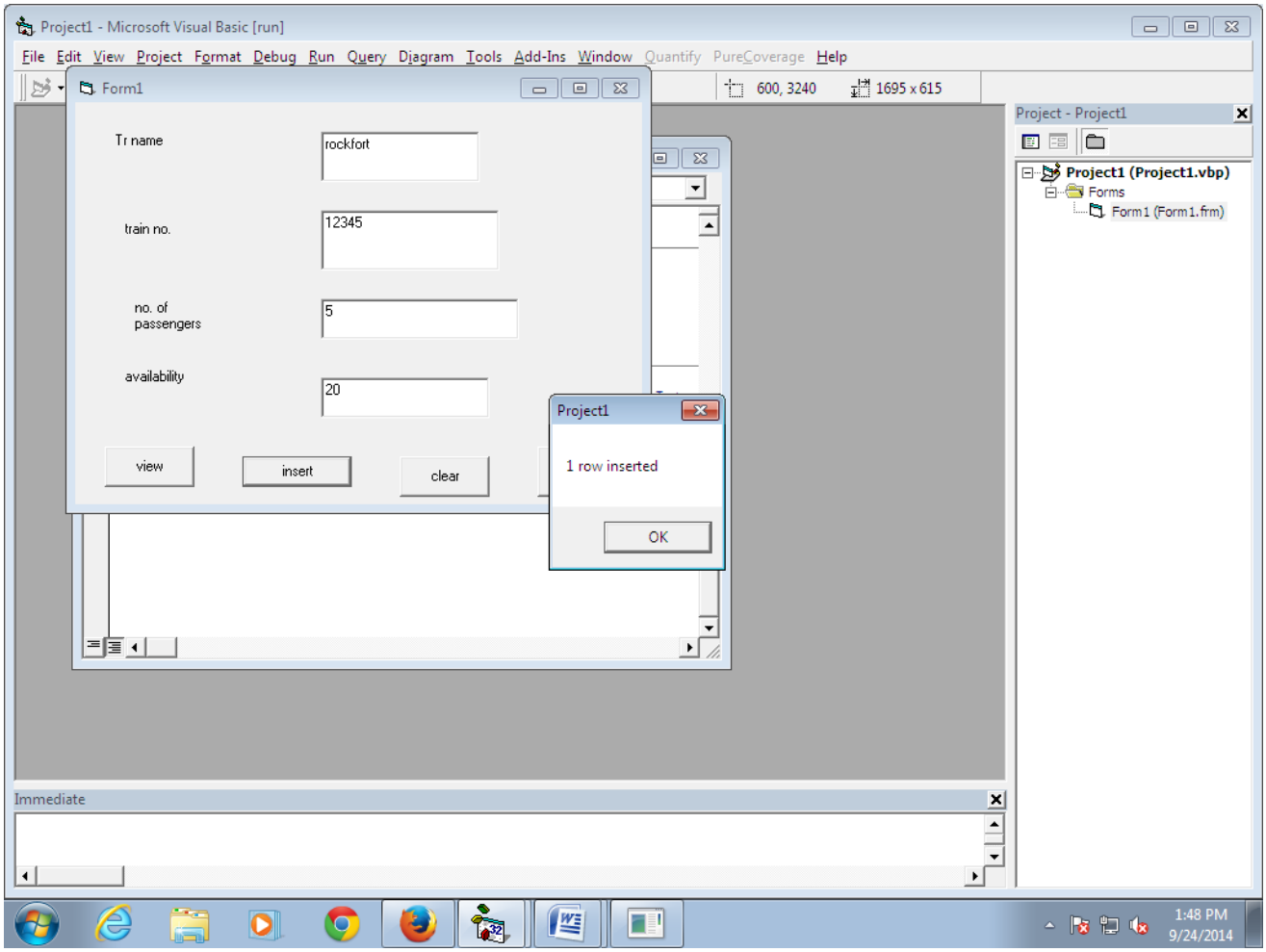
```
Private Sub Command3_Click()
Text1.Text = ""
Text2.Text = ""
Text3.Text = ""
End Sub
```

CLOSE:

```
Private Sub Command4_Click()
End
End Sub
```

OUTPUT:

```
SQL> select * from rail;
TRNAME      TRNO    PASSENGERS        AVAIL
------              ---------- -------------------   ----------
rockfort123455                  20
```



RESULT:

Thus the Railway Booking System is designed in VB and executed with the support of SQL-PLUS with proper ADODB Connection

Ex.No. 12e WEB BASED USER IDENTIFICATION SYSTEM

AIM:

Create a Web Based Identification System in VB and connect the corresponding Table from SQP-PLUS

PROCEDURE:

1. Start the process.
2. Create the table using oracle,SQL and insert some tuples in it.
3. Commit the table and exit.
4. For ODBC connectivity, select control panel -> administrative tools->ODBC data sources, select drivers menu, select Microsoft ODBC driver, click ok.
5. In VB 6.0, choose project menu, under that references.
6. Select Microsoft DAO 3.51 object library and Microsoft DAO 3.6 object library
7. Create forms for the project for the appropriate requirements specified.
8. Execute and terminate the project.

CODING:

GENERAL:

```
Dim con As New ADODB.Connection
Dim res As New ADODB.Recordset
```

VIEW:

```
Private Sub Command1_Click()
con.ConnectionString = "UID=system;PWD=manager;DRIVER={Microsoft ODBC for Oracle};" _
& "SERVER=csedb;"
con.Open
res.Open "select * from  web ", con, adOpenDynamic, adLockOptimistic
res.MoveFirst
With res
Text1.Text = res(0)
Text2.Text = res(1)
Text3.Text = res(2)
End With
res.Close
con.Close
End Sub
```

INSERT:

```
Private Sub Command2_Click()
con.ConnectionString = "UID=system;PWD=manager;DRIVER={Microsoft ODBC for Oracle};" _
& "SERVER=csedb;"
con.Open
 res.Open "select * from web ", con, adOpenDynamic, adLockOptimistic
res.AddNew
res(0) = Val(Text1.Text)
```

```
res(1) = Val(Text2.Text)
res(2) = Val(Text3.Text)
res.Update
MsgBox "1 row inserted"
res.Close
con.Close
End Sub
```

CLEAR:

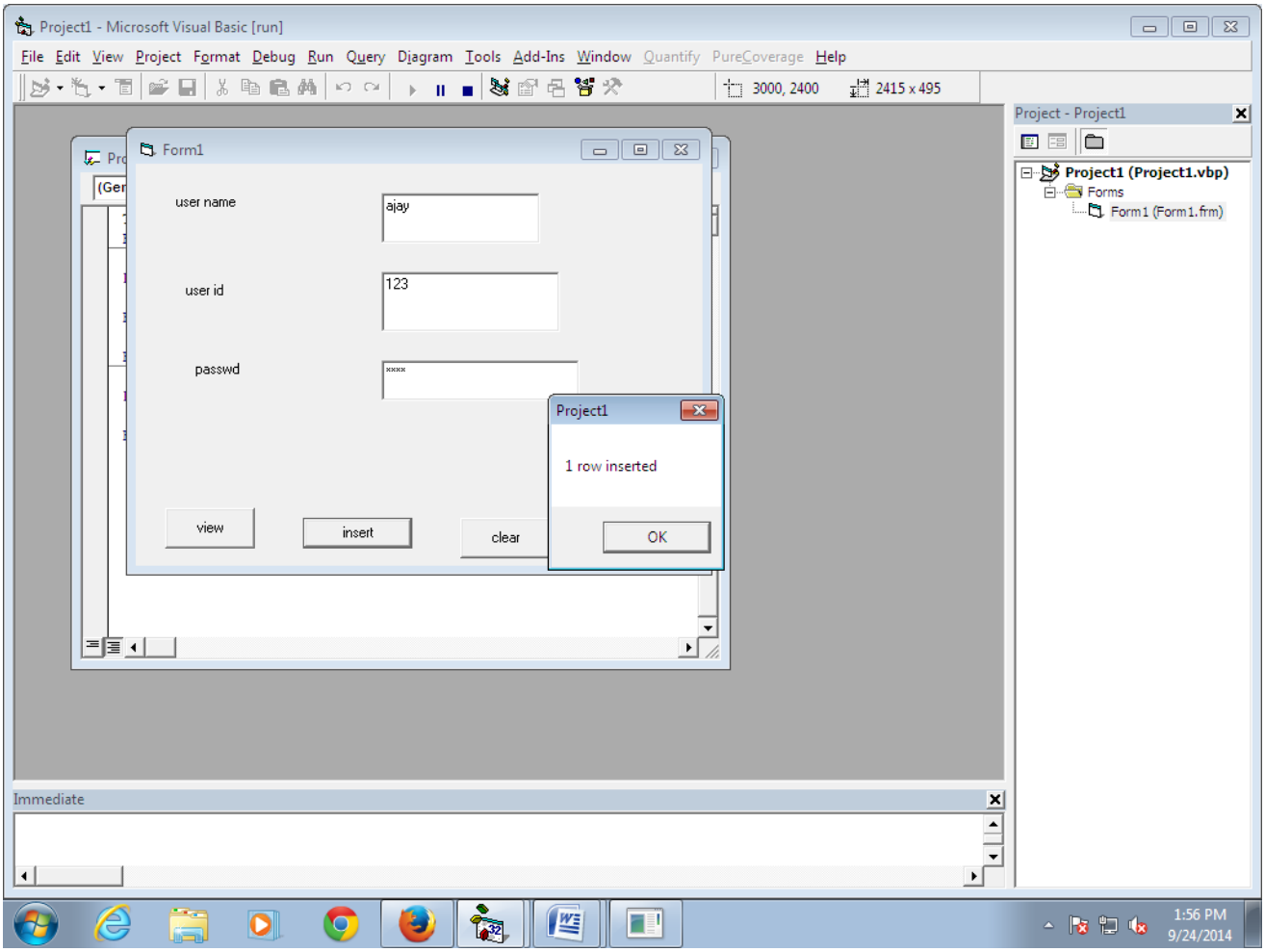
```
Private Sub Command3_Click()
Text1.Text = ""
Text2.Text = ""
Text3.Text = ""
End Sub
```

CLOSE:

```
Private Sub Command4_Click()
End
End Sub
```

OUTPUT:

```
SQL> select * from web;
USERNAME        USERID      PASSWD
------          --------- -      ------------------
ajay            123         ajay
```



RESULT:

Thus the Web Based Application System is designed in VB and executed with the support of SQL-PLUS with proper ADODB Connection

Ex.No. 12f HOTEL MANAGEMENT SYSTEM

AIM:

Create a Hotel Management System in VB and connect the corresponding Table from SQP-PLUS

PROCEDURE:

1. Start the process.
2. Create the table using oracle,SQL and insert some tuples in it.
3. Commit the table and exit.
4. For ODBC connectivity, select control panel -> administrative tools->ODBC data sources, select drivers menu, select Microsoft ODBC driver, click ok.
5. In VB 6.0, choose project menu, under that references.
6. Select Microsoft DAO 3.51 object library and Microsoft DAO 3.6 object library
7. Create forms for the project for the appropriate requirements specified.
8. Execute and terminate the project.

CODING:

GENERAL:

```
Dim con As New ADODB.Connection
Dim res As New ADODB.Recordset
```

VIEW:

```
Private Sub Command1_Click()
con.ConnectionString = "UID=system;PWD=manager;DRIVER={Microsoft ODBC for Oracle};" _
& "SERVER=csedb;"
con.Open
res.Open "select * from  hotel ", con, adOpenDynamic, adLockOptimistic
res.MoveFirst
With res
Text1.Text = res(0)
Text2.Text = res(1)
Text3.Text = res(2)
End With
res.Close
con.Close
End Sub
```

INSERT:

```
Private Sub Command2_Click()
con.ConnectionString = "UID=system;PWD=manager;DRIVER={Microsoft ODBC for Oracle};" _
& "SERVER=csedb;"
con.Open
 res.Open "select * from hotel ", con, adOpenDynamic, adLockOptimistic
res.AddNew
res(0) = Val(Text1.Text)
```

```
res(1) = Val(Text2.Text)
res(2) = Val(Text3.Text)
res.Update
MsgBox "1 row inserted"
res.Close
con.Close
End Sub
```

CLEAR:

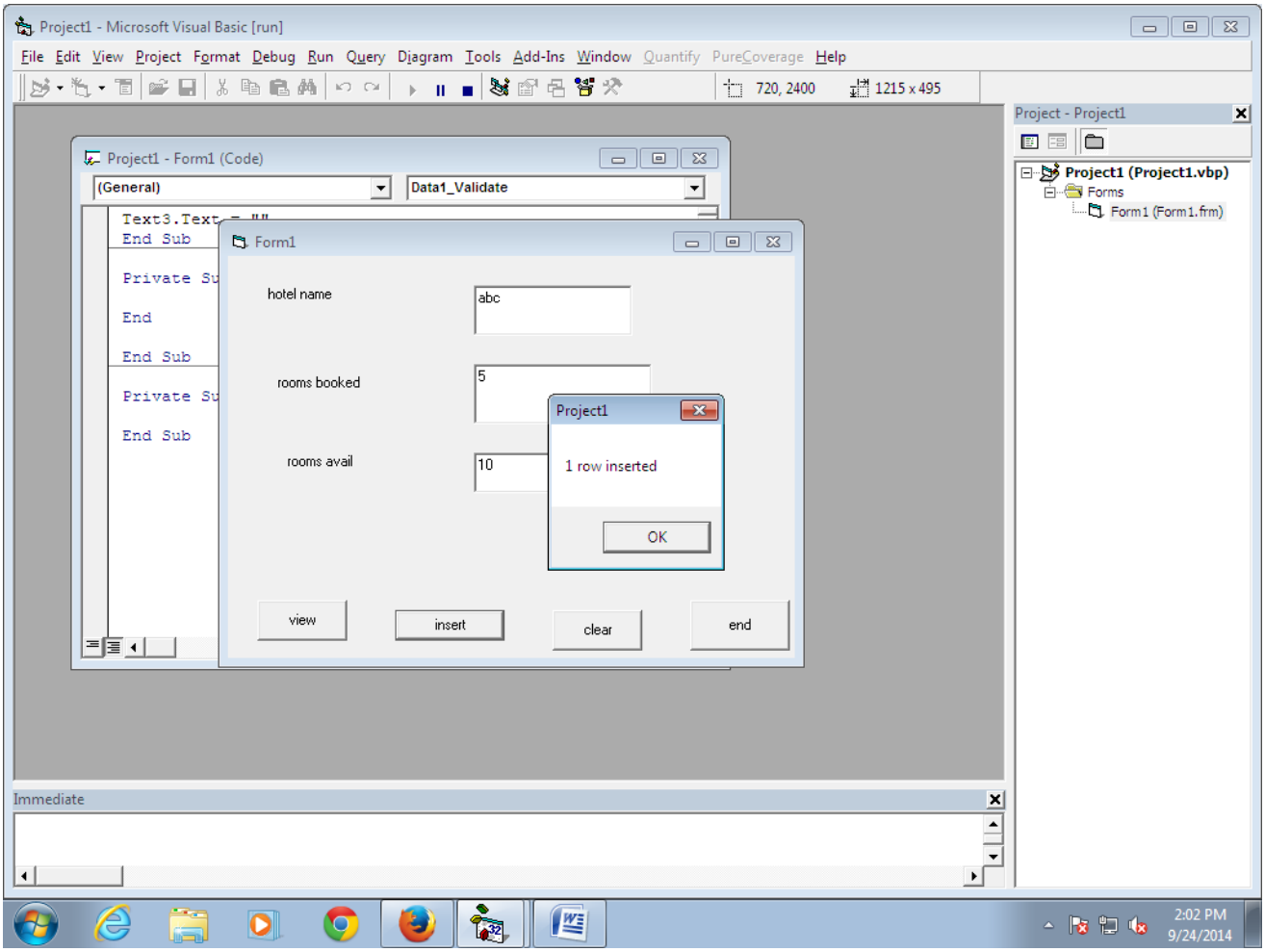
```
Private Sub Command3_Click()
Text1.Text = ""
Text2.Text = ""
Text3.Text = ""
End Sub
```

CLOSE:

```
Private Sub Command4_Click()
End
End Sub
```

OUTPUT:

```
SQL> select * from hotel;
HOTELNAME          ROOMS BOOKED    ROOMS AVAIL
------             ---------- -    ------------------
abc                5               10
```



RESULT:

Thus the Hotel Management System is designed in VB and executed with the support of SQL-PLUS with proper ADODB Connection