NPR
College of Engineering & Technology
Approved by AICTE, Affiliated to Anna University,
Accredited by NAAC WITH 'A' GRADE | Recognized by UGC under 2 (f)
Natham, Dindigul - 624 401. Web: www.nprcet.org

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

CS3481 – DATABASE MANAGEMENT SYSTEMS LAB

STUDENT MANUAL

Semester IV

Regulation 2021

TABLE OF CONTENTS

NPR
Group of Institutions
Reach the Star

NPR
College of Engineering & Technology
Approved by AICTE, Affiliated to Anna University,
Accredited by NAAC WITH 'A' GRADE|Recognized by UGC under 2 (f)
Natham, Dindigul - 624 401. Web: www.nprcet.org

A
GRADE
ACCREDITED BY
NAAC

INSTITUTE -VISION AND MISSION

Vision

To develop students with intellectual curiosity and technical expertise to meet the global needs.

Mission

M1: To achieve academic excellence by offering quality technical education using best teaching techniques.

M2 : To improve Industry – Institute interactions and expose industrial atmosphere.

M3 : To develop interpersonal skills along with value based education in a dynamic learning environment.

M4 : To explore solutions for real time problems in the society.

Department of Computer Science and Engineering

Department Vision:

To produce globally competent technical professionals for digitized society.

Department Mission:

M1: To establish conducive academic environment by imparting quality education and value added training.

M2 : To encourage students to develop innovative projects to optimally resolve the challenging social problems.

Program Educational Objectives (PEOs):

Graduates of Computer Science and Engineering Program will be able to

PEO1: Develop into the most knowledgeable professional to pursue higher education and research or have a successful career in industries.

PEO2: Successfully carry forward domain knowledge in computing and allied areas to solve complex and real world engineering problems.

PEO3: Meet the technological revolution they are continuously upgraded with the technical knowledge.

PEO4: Serve the humanity with social responsibility combined with ethics.

Program Outcomes (POs) for Computer Science and Engineerin

Program Outcomes (POs):		
PO1	**Engineering knowledge**	Apply the knowledge of mathematics, science, engineering fundamentals, and an engineering specialization to the solution of complex engineering problems.
PO2	**Problem analysis**	Identify, formulate, review research literature, and analyze complex engineering problems reaching substantiated conclusions using first principles of mathematics, natural sciences, and engineering sciences.
PO3	**Design/development of solutions**	Design solutions for complex engineering problems and design system components or processes that meet the specified needs with appropriate consideration for the public health and safety, and the cultural, societal, and environmental considerations.
PO4	**Conduct investigations of complex problems**	Use research-based knowledge and research methods including design of experiments, analysis and interpretation of data, and synthesis of the information to provide valid conclusions.
PO5	**Modern tool usage**	Create, select, and apply appropriate techniques, resources, and modern engineering and IT tools including prediction and modeling to complex engineering activities with an understanding of the limitations.
PO6	**The engineer and society**	Apply reasoning informed by the contextual knowledge to assess societal, health, safety, legal and cultural issues and the consequent responsibilities relevant to the professional engineering practice.
PO7	**Environment and sustainability**	Understand the impact of the professional engineering solutions in societal and environmental contexts, and demonstrate the knowledge of, and need for sustainable development.
PO8	**Ethics**	Apply ethical principles and commit to professional ethics and responsibilities and norms of the engineering practice.
PO9	**Individual and team work**	Function effectively as an individual, and as a member or leader in diverse teams, and in multidisciplinary settings.

PO9	**Individual and team work**	Function effectively as an individual, and as a member or leader in diverse teams, and in multidisciplinary settings.
PO10	**Communication**	Communicate effectively on complex engineering activities with the engineering community and with society at large, such as, being able to comprehend and write effective reports and design documentation, make effective presentations, and give and receive clear instructions.
PO11	**Project management and finance**	Demonstrate knowledge and understanding of the engineering and management principles and apply these to one's own work, as a member and leader in a team, to manage projects and in multidisciplinary environments.
PO12	**Life-long learning**	Recognize the need for, and have the preparation and ability to engage in independent and life-long learning in the broadest context of technological change.

Program Specific Outcomes (PSOs) for Computer Science and Engineering

At the end of the program students will be able to

PSO1	Deal with real time problems by understanding the evolutionary changes in computing, applying standard practices and strategies in software project development using open-ended programming environments.
PSO2	Employ modern computer languages , environments and platforms in creating innovative career paths by inculcating moral values and ethics.
PSO3	Achieve additional expertise through add-on and certificate programs.

CS3481 DATABASE MANAGEMENT SYSTEMS LABORATORY L T P C

0 0 3 1.5

OBJECTIVES:

- To learn and implement important commands in SQL.
- To learn the usage of nested and joint queries.
- To understand functions, procedures and procedural extensions of databases.
- To understand design and implementation of typical database applications.
- To be familiar with the use of a frontend tool for GUI based application development.

LIST OF EXPERIMENTS:

1. Create a database table, add constraints (primary key, unique, check, Not null), insert rows, update and delete rows using SQL DDL and DML commands.
2. Create a set of tables, add foreign key constraints and incorporate referential integrity.
3. Query the database tables using different 'where' clause conditions and also implement aggregate functions.
4. Query the database tables and explore sub queries and simple join operations.
5. Query the database tables and explore natural, equi and outer joins.
6. Write user defined functions and stored procedures in SQL.
7. Execute complex transactions and realize DCL and TCL commands.
8. Write SQL Triggers for insert, delete, and update operations in a database table.
9. Create View and index for database tables with a large number of records.
10. Create an XML database and validate it using XML schema.
11. Create Document, column and graph-based data using NOSQL database tools.
12. Develop a simple GUI based database application and incorporate all the above-mentioned features
13. Case Study using any of the real-life database applications from the following list
    a) Inventory Management for a EMart Grocery Shop
    b) Society Financial Management
    c) Cop Friendly App – Eseva
    d) Property Management – eMall
    e) Star Small and Medium Banking and Finance
    - Build Entity Model diagram. The diagram should align with the business and functional goals stated in the application
    - Apply Normalization rules in designing the tables in scope.
    - Prepared applicable views, triggers (for auditing purposes), functions for enabling enterprise grade features.
    - Build PL SQL / Stored Procedures for Complex Functionalities, ex EOD Batch Processing for calculating the EMI for Gold Loan for each eligible Customer.
    - Ability to showcase ACID Properties with sample queries with appropriate settings

List of Equipments:(30 Students per Batch)
MYSQL / SQL : 30 Users

TOTAL: 45 PERIODS

OUTCOMES:

At the end of this course, the students will be able to:

CO1: Create databases with different types of key constraints.

CO2: Construct simple and complex SQL queries using DML and DCL commands.

CO3: Use advanced features such as stored procedures and triggers and incorporate in GUI based application development.

CO4: Create an XML database and validate with meta-data (XML schema).

CO5: Create and manipulate data using NOSQL database.

CS3481 DATABASE MANAGEMENT SYSTEMS LABORATORY

Course Outcomes

After completion of the course, Students are able to learn the listed Course Outcomes.

Cos	Course Code	Course Outcomes	Knowledge Level
CO1	C217.1	Create databases with different types of key constraints.	K4
CO2	C217.2	Construct simple and complex SQL queries using DML and DCL commands.	K3
CO3	C217.3	Use advanced features such as stored procedures and triggers and incorporate in GUI based application development	K3
CO4	C217.4	Create an XML database and validate with meta-data (XML schema). .	K3
CO5	C217.5	Create and manipulate data using NOSQL database.	K3

List of Experiments with COs, POs and PSOs

Exp.No.	Name of the Experiment	COs	POs	PSOs
1.	DATA DEFINITION IN SQL (DDL)	CO1,2	PO1,2,3	PSO1,2
2.	DATA MANIPULATION IN SQL (DML)	CO1,2	PO1,2,3	PSO1,2
3.	DATA CONTROL IN SQL (DCL)	CO1,2	PO1,2,3	PSO1,2
4.	CONSTRAINTS	CO1,2	PO1,2,3	PSO1,2
5.	JOINS	CO1,2	PO1,2,3	PSO1,2
6.	VIEWS	CO1,2	PO1,2,3	PSO1,2
7.	NESTED QUERIES	CO1,2	PO1,2,3	PSO1,2

8.	AGGREGATE FUNCTIONS	CO1,2	PO1,2,3	PSO1,2
9.	SET OPERATIONS	CO1,2	PO1,2,3	PSO1,2
10.	CURSORS	CO1,2	PO1,2,3	PSO1,2
11.	PROCEDURES	CO3	PO1,2,3	PSO1,2
12.	FUNCTIONS	CO3	PO1,2,3	PSO1,2
13.	CONTROL STRUCTURES	CO3	PO1,2,3	PSO1,2
14.	TRIGGERS	CO3	PO1,2,3	PSO1,2
15	XML DATABASE	CO4	PO1,2,3	PSO1,2
16	DOCUMENT DATA USING NOSQL DATABASE TOOLS	CO5	PO5	PSO1,2
17	BANKING SYSTEM	CO1,2,3	PO1,2,3,4	PSO1,2

Program Outcomes

1. Engineering Knowledge
2. Problem Analysis
3. Design/Development of Solutions
4. Conduct Investigations of Complex Problems
5. Modern Tool Usage
6. The Engineer and Society
7. Environment and Sustainability
8. Ethics
9. Individual and Team Work
10. Communication
11. Project Management and Finance
12. Life-long Learning

Program Specific Outcomes

At the end of the program students will be able to

- Deal with real time problems by understanding the evolutionary changes in computing, applying standard practices and strategies in software project development using open-ended programming environments.
- Employ modern computer languages, environments and platforms in creating innovative career paths by inculcating moral values and ethics.
- Achieve additional expertise through add-on and certificate programs.

SQL – AN INTRODUCTION

What is Database?

A database is a separate application that stores a collection of data. Each database has one or more distinct application programming interfaces (API) for creating, accessing, managing, searching and replicating the data it holds.

Other kinds of data stores can be used, such as files on the file system or large hash tables in memory but data fetching and writing would not be so fast and easy with those types of systems.

So nowadays, we use relational database management systems (RDBMS) to store and manage huge volume of data. This is called relational database because all the data is stored into different tables and relations are established using primary keys or other keys known as foreign keys.

A **Relational DataBase Management System (RDBMS)** is a software that:

- Enables you to implement a database with tables, columns and indexes.
- Guarantees the Referential Integrity between rows of various tables.
- UpDates the indexes automatically.
- Interprets an SQL query and combines information from various tables.

RDBMS Terminology:

Before we proceed to explain SQL database system, let's revise few definitions related to database.

- **Database:** A database is a collection of tables, with related data
- **Table:** A table is a matrix with data. A table in a database looks like a simple spreadsheet
- **Column:** One column (data element) contains data of one and the same kind, for example the column postcode
- **Row:** A row (= tuple, entry or record) is a group of related data, for example the data of one subscription
- **Redundancy:** Storing data twice, redundantly to make the system faster
- **Primary Key:** A primary key is unique. A key value cannot occur twice in one table. With a key, you can find at most one row
- **Foreign Key:** A foreign key is the linking pin between two tables.
- **Compound Key:** A compound key (composite key) is a key that consists of multiple columns, because one column is not sufficiently unique.
- **Index:** An index in a database resembles an index at the back of a book.
- **Referential Integrity:** Referential Integrity makes sure that a foreign key value always points to an existing row.

TABLES:

In relational database systems (DBS) data are represented using tables (relations). A query issued against the DBS also results in a table.

A table has the following structure:

Column 1	**Column 2**	**. . .**	**Column n**	
				<– Tuple (or Record)
. . .	. . .	. . .	. . .	

A table is uniquely identified by its name and consists of rows that contain the stored information, each row containing exactly one tuple (or record). A table can have one or more columns.

A column is made up of a column name and a data type, and it describes an attribute of the tuples. The structure of a table, also called relation schema, thus is defined by its attributes.The type of information to be stored in a table is defined by the data types of the attributes at table creation time.

SQL uses the terms table, row, and column for relation, tuple, and attribute, respectively.

SQL offers the following basic data types:

• **char(n):** Fixed-length character data (string), n characters long. The maximum size for n is 255 bytes (2000 in Oracle8). Note that a string of type char is always padded on right with blanks to full length of n. (+ can be memory consuming).
Example: char(40)

• **varchar2(n):** Variable-length character string. The maximum size for n is 2000 (4000 in Oracle8). Only the bytes used for a string require storage.
Example: varchar2(80)

• **number(o, d):** Numeric data type for integers and reals. o = overall number of digits, d = number of digits to the right of the decimal point.
Maximum values: o =38, d= −84 to +127. Examples: number(8), number(5,2)
• Date Date data type for storing Date and time.
The default format for a Date is: ‘YYYY-MM-DD’. Examples: ’1997-10-10’;

• Long: Character data up to a length of 2GB. Only one long column is allowed per table.

Further properties of tables are:

• the order in which tuples appear in a table is not relevant (unless a query requires an explicit sorting).

• a table has no duplicate tuples (depending on the query, however, duplicate tuples can appear in the query result).

A database schema is a set of relation schemas. The extension of a database schema at database run-time is called a database instance or database.

EX.NO:1 **DATA DEFINITION IN SQL (DDL)**

AIM:

To implement data definition commands using SQL.

DESCRIPTION:

Creating Tables

TheSQL command for creating an empty table has the following form:

```
create table <table> (
<column 1><data type> [not null] [unique] [<column constraint>],
. . . . . . . . .
<column n><data type> [not null] [unique] [<column constraint>],
[<table constraint(s)>]
);
```

For each column, a name and a data type must be specified and the column name must be unique within the table definition. Column definitions are separated by comma. There is no difference between names in lower case letters and names in upper case letters. In fact, the only place where upper and lower case letters matter are strings comparisons.

A **not null** constraint is directly specified after the data type of the column and the constraint requires defined attribute values for that column, different from null.
Unless the condition not null is also specified for this column, the attribute value null is allowed and two tuples having the attribute value null for this column do not violate the constraint.

The keyword **unique** specifies that no two tuples can have the same attribute value for this column.

Checklist for Creating Tables

The following provides a small checklist for the issues that need to be considered before creating a table.

- What are the attributes of the tuples to be stored? What are the data types of the attributes? Should varchar2 be used instead of char?
- Which columns build the primary key?
- Which columns do (not) allow null values? Which columns do (not) allow duplicates?
- Are there default values for certain columns that allow null values?

Modifying Table and Column Definitions:

It is possible to modify the structure of a table (the relation schema) even if rows have already been inserted into this table.
A column can be added using the alter table command

```
alter table <table>
add(<column><data type> [default <value>] [<column constraint>]);
```

If more than only one column should be added at one time, respective add clauses need to be separated by colons. A table constraint can be added to a table using

```
alter table <table> add (<table constraint>);
```

Note:

A column constraint is a table constraint, too. not null and primary key constraints can only be added to a table if none of the specified columns contains a null value. Table definitions can be modified in an analogous way. This is useful, e.g., when the size of strings that can be stored needs to be increased. The syntax of the command for modifying a column is

```
alter table <table>
modify(<column> [<data type>] [default <value>] [<column constraint>]);
```

Note:
In earlier versions of Oracle it is not possible to delete single columns from a table definition. A workaround is to create a temporary table and to copy respective columns and rows into this new table.

Renaming a Table

A table can be renamed using the rename command.

```
rename <old table name> to <new table name>;
```

Deleting a Table

A table and its rows can be deleted by issuing the command

```
drop table <table> [cascade constraints];
```

OUTPUT:

Data Definition Language (DDL)

1.**Creating Tables**

SQL> create table student(sno number(9), stu_name varchar(9) not null, rollno number(9) not null dob date, phone_no number(10));

Table description

SQL> desc student;

Modifying Table and Column Definitions

SQL> alter table student add address varchar(9);

MODIFY: -

SQL> alter table student modify stu_name varchar(20);

Table description

SQL> desc student;

Renaming a Table

SQL> rename student to studentdetail;

SQL> desc student;

Table description

```
SQL> desc studentdetail;
```

```
SQL> drop table studentdetail;
```

Table description:

```
SQL> desc studentdetail;
```

RESULT:

Thus the data definition commands have been used to create , alter , and drop table using SQL queries.

EX NO:2 DATA MANIPULATION IN SQL(DML)

AIM:

To implement data manipulation using SQL queries.

DESCRIPTION:

After a table has been created using the create table command, tuples can be inserted into the table, or tuples can be deleted or modified.

Insertions

The most simple way to insert a tuple into a table is to use the insert statement

```
insert into <table> [(<column i, . . . , column j>)]
values (<value i, . . . , value j>);
```

For each of the listed columns, a corresponding (matching) value must be specified. Thus an insertion does not necessarily have to follow the order of the attributes as specified in the create table statement. If a column is omitted, the value null is inserted instead. If no column list is given, however, for each column as defined in the create table statement a value must be given.

If there are already some data in other tables, these data can be used for insertions into a new table. For this, we write a query whose result is a set of tuples to be inserted. Such an insert statement has the form

```
insert into <table> [(<column i, . . . , column j>)] <query>
```

UpDates

For modifying attribute values of (some) tuples in a table, we use the upDate statement:

```
upDate<table> set
<column i> = <expression i>, . . . , <column j> = <expression j>
[where <condition>];
```

An expression consists of either a constant (new value), an arithmetic or string operation, or
an SQL query.

Note: that the new value to assign to <column i> must a matching data type.
An upDate statement without a where clause results in changing respective attributes of all
tuples in the specified table. Typically, however, only a (small) portion of the table requires an upDate.

Deletions

All or selected tuples can be deleted from a table using the delete command:

```
delete from <table> [where <condition>];
```

If the where clause is omitted, all tuples are deleted from the table. An alternative command for deleting all tuples from a table is the truncate table <table> command. However, in this case, the deletions cannot be undone.

OUTPUT:

```
SQL>  create table EMP (EMPNO number(4) not null, ENAME varchar2(30) not null,
  JOB varchar2(10), MGR number(4), HIREDATE Date, SAL number(7,2), DEPTNO number(2));
```

INSERT

Type 1

```
SQL> insert into emp (empno,ename,job,mgr,hireDate,sal,deptno)values (3737,'Priya','Analyst','7777','07-mar-2022',34000,07);
```

Type 2

```
SQL> insert into emp values (2323,'Anitha','Programmer',5454,'09-jan-23',42000,09);

SQL> insert into emp values (7575,'Karthi',' Developer ',3337,'07-jul-2022',72000, 07);

SQL> insert into emp values (5352,'Retish','Secretary','5555','09-jun-2022',20000,08);

SQL> insert into emp values (5332,'Rocky','Assistant','5555','08-jan-2020',13000,07);

SQL> select * from emp;
SQL> update emp set job = 'Manager', deptno=20, sal = sal +3000 where ename = 'Karthi';
SQL> select * from emp;
SQL> update emp set sal = sal * 1.5  where deptno = 7;
```

TUPLE DELETION

```
SQL> delete from emp where empno=2323;

SQL> select * from emp;
```

RESULT:

Thus data manipulation language queries have been implemented and verified using SQL.

EX.NO:3 DATA CONTROL IN SQL (DCL)

AIM:

To implement data control language using SQL.

DESCRIPTION:

Selections

Selecting Columns

The columns to be selected from a table are specified after the keyword select. This operation is also called ***projection.***

The query

```
select <column i, . . . , column j> from <table>;
```

lists only the attribute values of specified columns for each tuple from the denoted relation.

If all columns should be selected, the asterisk symbol "*"can be used to denote all attributes.

The query

```
select * from <table>;
```

retrieves all tuples with all columns from the table.

Instead of an attribute name, the select clause may also contain arithmetic expressions involving arithmetic operators etc.

The query

```
Select <column I >* 1.55 from <table>;
```

retrieves all tuples with column specified as a product with the specified number from the table.

For the different data types supported in Oracle, several operators and functions are provided:

- for numbers: abs, cos, sin, exp, log, power, mod, sqrt, +,−, _, /, . . .
- for strings: chr, concat(string1, string2), lower, upper, replace(string, search string, replacement string), translate, substr(string, m, n), length, to Date, . . .
- for the Date data type: add month, month between, next day, to char, . . .

Inserting the keyword ***distinct*** after the keyword select, duplicate result tuples are automatically eliminated.

```
Select  distinct <column i> from <table>;
```

Order by

It is also possible to specify a sorting order in which the result tuples of a query are displayed. For this the ***order by clause*** is used and which has one or more attributes listed in the select clause as parameter, ***desc***specifies a descending order and ***asc*** specifies an ascending order (this is also the default order).

The query

```
Select <column i, . . . , column j>
From <table>
Order by <column i> [asc], <column j> desc;
```

displays the result in an ascending order by the attribute <column i>. If two tuples have the same attribute value, the sorting criteria is a descending order by the attribute values of
<column j>.

Selection of Tuples

Where

To conditionally select the datas from a table, we use the ***where*** keyword.

The syntax is as follows:

```
Select <column name>
From <table>
Where "condition"
```

And Or

the where keyword can be used to conditionally select data from a table. This condition can be a simple condition (like the one presented in the previous section), or it can be a compound condition. Compound conditions are made up of multiple simple conditions connected by ***AND*** or ***OR***. There is no limit to the number of simple conditions that can be present in a single SQL statement.

The syntax for a compound condition is as follows:

```
Select "column_name"
From "table_name"
Where "simple condition"
{[AND|OR] "simple condition"}+
```

The {}+ means that the expression inside the bracket will occur one or more times. Note that ***AND*** and ***OR*** can be used interchangeably. In addition, we may use the parenthesis sign () to indicate the order of the condition.

In

The **IN** keyword, when used with where in this context, we know exactly the value of the returned values we want to see for at least one of the columns. The syntax for using the **IN** keyword is as follows:

```
Select <column name>
From <table>
Where <column name> IN ('value1', ‘value2’ ...)
```

The number of values in the parenthesis can be one or more, with each values separated by comma. Values can be numerical or characters. If there is only one value inside the parenthesis, this commend is equivalent to

```
Where <column name> = 'value1'
```

Also

```
Select <column name>
From <table>
Where <column name> NOT IN ('value1', ‘value2’ ...)
```

Between

Whereas the IN keyword help people to limit the selection criteria to one or more discrete values, the *BETWEEN* keyword allows for selecting a range.

The syntax for the BETWEEN clause is as follows:

```
Select <column name>
From <table>
Where <column name> BETWEEN 'value1' AND 'value2'
```

This will select all rows whose column has a value between 'value1' and 'value2'.

For all data types, the comparison operators =, != or <>,<, >,<=, => are allowed in the conditions of a where clause.

For a tuple to be selected there must (not) exist a defined value for this column.

```
value: <column> is [not] null
```

Note: The operations = null and ! = null are not defined!
Domain conditions: <column> [not] between<lower bound> and <upper bound

OUTPUT:

SELECT

```
SQL> select * from emp;
SQL> select ename from emp;
SQL> select sal * 2 from emp;
SQL> select ename,sal * 2 from emp;
SQL> select mgr from emp;
```

DISTINCT

```
SQL> select distinct mgr from emp;
```

ORDER BY

```
SQL> select * from emp order by ename;
SQL> select * from emp order by ename asc;
SQL>  select * from emp order by ename desc;
SQL> select * from emp order by ename desc, empno asc;
SQL> select * from emp order by empno desc, ename;
```

OR

```
SQL> select ename from emp where ( mgr = 5555 or deptno = 07);
```

AND

```
SQL> select ename from emp where ( mgr = 5555 and deptno = 07);
```

IN

SQL> select ename,sal from emp where empno in (2323,5555);

SQL> select ename,salfrom emp where empno in (2323,7575);

BETWEEN

SQL> select ename,sal from emp where sal between 25000 and 50000;

RESULT:

Thus data query languages have been implemented and verified using SQL.

EX.NO:4 CONSTRAINTS

AIM:

To implement various types of constrains using SQL.

DESCRIPTION:

Constraints are used to limit the type of data that can go into a table. Such constraints can be specified when the table is first created via the **CREATE TABLE** statement, or after the table is already created via the **ALTER TABLE** statement.

Common types of constraints include the following:

- **NOT NULL Constraint**: Ensures that a column cannot have NULL value.
- **DEFAULT Constraint**: Provides a default value for a column when none is specified.
- **UNIQUE Constraint:** Ensures that all values in a column are different.
- **CHECK Constraint**: Makes sure that all values in a column satisfy certain criteria.
- **PRIMARY KEY Constraint**: Used to uniquely identify a row in the table.
- **FOREIGN KEY Constraint**: Used to ensure referential integrity of the data.

NOT NULL Constraint

By default, a column can hold NULL. If you not want to allow NULL value in a column, you will want to place a constraint on this column specifying that NULL is now not an allowable value.

The syntax to be used is

```
Create table <table> (
<column 1><data type> [NOT NULL] [<column constraint>],
. . . . . . . . .
<column n><data type> [NOT NULL] [<column constraint>],
);
```

DEFAULT Constraint

The DEFAULT constraint provides a default value to a column when the INSERT INTO statement does not provide a specific value.
The syntax used is

```
Create table <table> (
<column 1><data type>,
. . . . . . . . .
<column n><data type>,
<column n><data type> DEFAULT value
);
```

UNIQUE Constraint

The UNIQUE constraint ensures that all values in a column are distinct.

The syntax used is

```
Create table <table> (
<column 1><data type> UNIQUE,
. . . . . . . . .
<column n><data type>
);
```

CHECK Constraint

The CHECK constraint ensures that all values in a column satisfy certain conditions. Once defined, the database will only insert a new row or upDate an existing row if the new value satisfies the CHECK constraint. The CHECK constraint is used to ensure data quality.

The syntax used is

```
Create table <table> (
<column 1><data type> CHECK <condition>,
. . . . . . . . .
<column n><data type>
);
```

PRIMARY KEY Constraint

A primary key is used to uniquely identify each row in a table. It can either be part of the actual record itself, or it can be an artificial field (one that has nothing to do with the actual record). A primary key can consist of one or more fields on a table. When multiple fields are used as a primary key, they are called a composite key.

Primary keys can be specified either when the table is created (using **CREATE TABLE**) or by changing the existing table structure (using **ALTER TABLE**).

The syntax used is

```
Create table <table> (
<column 1><data type> PRIMARY KEY,
. . . . . . . . .
<column n><data type>
);
```

Or

```
Create table <table> (
<column 1><data type>,
. . . . . . . . .
<column n><data type>
);

Alter table Customer ADD PRIMARY KEY <column name>;
```

Note: Before using the ALTER TABLE command to add a primary key, you'll need to make sure that the field is defined as 'NOT NULL' -- in other words, NULL cannot be an accepted value for that field.

FOREIGN KEY Constraint

A foreign key is a field (or fields) that points to the primary key of another table. The purpose of the foreign key is to ensure referential integrity of the data. In other words, only values that are supposed to appear in the database are permitted.

The syntax used is

```
Create table <table> (
<column 1><data type> PRIMARY KEY,
. . . . . . . . .
<column n><data type>
);

Create table <table1> (
<column 1><data type> PRIMARY KEY,
. . . . . . . . .
<column m><data type> REFERENCES <table>
);
```

Or

```
Create table <table> (
<column 1><data type> PRIMARY KEY,
. . . . . . . . .
<column n><data type>
);

Create table <table1> (
<column 1><data type> PRIMARY KEY,
. . . . . . . . .
<column m><data type>
);

Alter table <table1>
ADD  (CONSTRAINT fk_<table1>)FOREIGN KEY <column m>REFERENCES <table>(column n);
```

OUTPUT:

CONSTRAINTS

UNIQUE

```
SQL> CREATE TABLE Customer
  (SID integer Unique,
  Last_Name varchar (30),
  First_Name varchar(30));
SQL> insert into customer values ('23','rad','ree');
SQL> insert into customer ('24','ram','pri');
SQL> insert into customer values ('23','kek','kak');
```

NOT NULL

```
SQL> CREATE TABLE Customer
  (SID integer NOT NULL,
  Last_Name varchar (30) NOT NULL,
  First_Name varchar(30));
SQL> desc customer;
SQL> insert into customer values(23333,'ram','priya');
SQL> insert into customer values(12222,",'jai');
SQL> insert into customer values(12222,'raj',");
SQL> select * from customer;
```

CHECK

```
SQL> CREATE TABLE Customer
  (SID integer CHECK (SID > 100),
  Last_Name varchar (30),
  First_Name varchar(30));
SQL> insert into customer values ('244','ram ','pri');
SQL> insert into customer values ('23','uma','ram');
```

PRIMARY KEY

```
SQL> CREATE TABLE Customer (SID integer, Last_Name varchar(30), First_Name varchar(30),
PRIMARY KEY (SID));
SQL> desc customer
SQL> insert into customer values('34','ram','pri');
SQL> insert into customer values('54','oop','raj');
SQL> insert into customer values('54','tem','temp');
 ERROR at line 1:
ORA-00001: unique constraint (SCOTT.SYS_C00602) violated
SQL> insert into customer values(",'trim','trimi');
```

PRIMARY KEY USING ALTER

```
SQL> CREATE TABLE Customer
  (SID integer,
  Last_Name varchar(30),
  First_Name varchar(30));
SQL> desc customer;
SQL> ALTER TABLE Customer ADD PRIMARY KEY (SID);
SQL> desc customer;
```

FOREIGN KEY

```
SQL> desc customer;
SQL> CREATE TABLE ORDERS (Order_ID integer primary key, Order_DateDate, Customer_SID
integer references CUSTOMER(SID),  amount number(8,2));
SQL> desc orders;
SQL> insert into orders values('545','30-sep-2021','567','9000');
SQL> insert into orders values ('506','11-aug-2020',",'3000');
SQL> insert into orders values('555','19-nov-2022','789','3000');
SQL> select * from orders;
SQL> select * from customer;
SQL> delete from customer where last_name='sam';
SQL> select * from customer:
SQL> delete from customer where last_name='raj';
SQL> delete from orders where CUSTOMER_SID=345;
SQL> select * from orders;
SQL> select * from customer;
SQL> delete from customer where last_name='raj';
SQL> select * from customer;
```

ON DELETE CASCADE

```
SQL> desc customer
SQL> select * from customer;
SQL> CREATE TABLE ORDERS1
  (Order_ID integer,
  Order_DateDate,
  Customer_SID integer,
  amount number(8,2),
  Primary Key (Order_ID),
  Foreign Key (Customer_SID) references CUSTOMER(SID) on delete cascade);
SQL> select * from orders1;
SQL> delete from orders1 where amount < 5000;
SQL> select * from orders1;
SQL> delete from customer where last_name='raz';
SQL> select * from customer;
SQL> delete from orders1 where customer_sid=123;
SQL> select * from orders1;
SQL> delete from customer where sid=234;
SQL> select * from customer;
```

RESULT:

Thus, various types of constrains have been implemented and verified using SQL.

EX.NO:5 JOINS

AIM:

To implement joins using SQL.

DESCRIPTION:

This is a binary operation that allows two relations to combine certain selections and cartesian product into one resulting relation.

TYPES OF JOIN

- Inner join
- Outer join

INNER - JOIN

Here, join operation forms a cartesian product of two relation's arguments, performs a selection forcing equality on those attributes that appear in both relation schemes and finally removes duplicate attributes. Also, this is referred as inner join.

The query

```
Select *
From <table1>,<table2>
Where table1.column i = table2.column i;
```

OUTER – JOIN

The outer-join operation is an extension of the join operation to deal with missing information.

There are three forms of outer – join

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

Left Outer – Join

This takes all tuples in the left relation that did not match with any tuple in the right relation, pads the tuples with null values for all other attributes from the right relation and adds them to the result of the join operation.

The query

```
Select *
From <table1>,<table2>
Where table1.column i(+) = table2.column i;
```

Right Outer – Join

This takes all tuples in the right relation that did not match with any tuple in the left relation, pads the tuples with null values for all other attributes from the left relation and adds them to the result of the join operation.

```
Select *
From <table1>,<table2>
Where table1.column i = table2.column i(+);
```

Full Outer – Join

This pads tuples from the left relation that did not match any from the right relation, as well as tuples from the right relation that did not match any from the left relation and adds them to the resultant relation.

Note

The syntax for performing an outer join in SQL is database-dependent.
In Oracle, we will place an "(+)" in the **WHERE** clause on the other side of the table for which we want to include all the rows.

OUTPUT:

JOINS

Table description

```
SQL> desc table1;
```

Table values

```
SQL> select * from table1;
SQL> select * from table1;
```

Table description

```
SQL>desc table2;
```

Table values

```
SQL> select *from table2;
```

INNER JOIN

```
SQL> select *from table1,table2 where table1.id=table2.id;
```

OUTER JOIN

LEFT OUTER JOIN

```
SQL>  select * from table1,table2 where table1.id(+)=table2.id;

SQL> select table1.name,table2.salary from table1,table2 where table1.id(+)=table2.id;NAME    SALARY
```

RIGHT OUTER JOIN

```
SQL>  select * from table1,table2 where table1.id=table2.id(+);

SQL> select table1.name,table2.salary from table1,table2 where table1.id=table2.id(+);
```

RESULT:

Thus joins have been implemented and verified using SQL.

EX.NO:6 VIEWS

AIM:

To implement views using SQL.

DESCRIPTION:

A view is a virtual table. A view consists of rows and columns just like a table. The difference between a view and a table is that views are definitions built on top of other tables (or views), and do not hold data themselves. If data is changing in the underlying table, the same change is reflected in the view. A view can be built on top of a single table or multiple tables. It can also be built on top of another view.

Views are allowed to use one of the following constructs in the view definition:

- Joins
- Aggregate function such as sum, min, max etc.
- Set-valued subqueries (in, any, all) or test for existence (exists)
- Group by clause or distinct clause

Views are classified as

- Read – Only views
- Updatable views

Read – Only Views

These views are created from multiple relations.

Updatable Views

These views are created from individual relation.

Views offer the following advantages:

1. Ease of use: A view hides the complexity of the database tables from end users. Essentially we can think of views as a layer of abstraction on top of the database tables.

2. Space savings: Views takes very little space to store, since they do not store actual data.

3. Additional data security: Views can include only certain columns in the table so that only the non-sensitive columns are included and exposed to the end user. In addition, some databases allow views to have different security settings, thus hiding sensitive data from prying eyes.

CREATE VIEW

A view can be created using command create.

The syntax

```
CREATE VIEW view_name AS
SELECT column_name(s)
FROM table_name
WHERE condition
```

DELETE VIEW

A view can be deleted using the command drop.

The syntax

```
Drop view <view-name>;
```

OUTPUT:
VIEW

```
SQL> desc emp;
SQL> select * from emp;
SQL> desc empp;
SQL> select * from empp;
```

CREATE VIEW

View created from more than one table leading to 'Read Only' view.

```
SQL> create view view1 as select emp.ename,empp.ph_no from emp,empp where
emp.empno=empp.empno;
SQL> select * from view1;
```

View created from single relation known as 'Updatable' view.

```
SQL> create view view2 asselect ename,sal from emp;
SQL> select * from view2;
```

View can be upDated only on the updatable views.

```
SQL> upDate view2 set sal=23000 where ename='santh';
SQL> drop view view1.
SQL> select * from view1;
```

RESULT:

Thus views have been implemented and verified using SQL.

EX.NO:7 NESTED QUERIES

AIM:
To implement the nested query using SQL.

DESCRIPTION:
A query result can also be used in a condition of a where clause. In such a case the query is called a subquery and the complete select statement is called a nested query.

A respective condition in the where clause then can have one of the following forms:
1. Set-valued subqueries

```
<expression> [not] in (<subquery>)
<expression><comparison operator> [any|all] (<subquery>)
```

An <expression> can either be a column or a computed value.

2. Test for (non)existence

```
[not] exists (<subquery>)
```

In a where clause conditions using sub queries can be combined arbitrarily by using the logical connectives and and or.Conditions of the form <expression><comparison operator> [any|all] <subquery> are usedto compare a given <expression> with each value selected by <subquery>.

• For the clause any, the condition evaluates to true if there exists at least on row selected by the subquery for which the comparison holds. If the subquery yields an empty result set, the condition is not satisfied.

• For the clause all, in contrast, the condition evaluates to true if for all rows selected by the subquery the comparison holds. In this case the condition evaluates to true if the subquery does not yield any row or value.
For all and any, the following equivalences hold:

in , = any
not in , <> all or != all

Often a query result depends on whether certain rows do (not) exist in (other) tables. Such type of queries is formulated using the exists operator.

OUTPUT:

NESTED QUERY (SUB QUERY)

Table Description

```
SQL> SELECT * FROM EMP;

SQL> select * from empp;
```

SUBQUERY USING IN

```
SQL> select * from EMP E1 where DEPTNO in  (select DEPTNO from EMP E where E.DEPTNO =7);
```

SUBQUERY USING ANY

```
SQL> select * from EMP where SAL >= any (select SAL from EMP where DEPTNO =8);


SQL> select * from EMP where SAL >= any (select SAL from EMP where DEPTNO =8)
   and DEPTNO = 7;
```

SUBQUERY USING ALL

```
SQL> select * from EMP where SAL > all (select SAL from EMP where DEPTNO = 7)
         and DEPTNO <> 15;

SQL>  select *  from empp where exists  (select * from EMP where emp.empno=empp.empno);
```

SUBQUERY USING NOT EXISTS

```
SQL> select * from empp where not exists  (select * from EMP  where emp.empno=empp.empno);
```

RESULT:

Thus, the nested queries using SQL has been implemented and verified.

EX.NO:8 **AGGREGATE FUNCTIONS**

AIM:
To implement various aggregate functions using SQL.

DESCRIPTION:
Aggregate functions are statistical functions such as count, min, max etc. They are used to compute a single value from a set of attribute values of a column.

AVG Function

SQL uses the AVG function to calculate the average of a column.

The syntax for using this function is,

```
Select AVG <column name>
from <table>
```

SUM Function

The SUM function is used to calculate the total for a column.

The syntax is,

```
Select SUM<column name>
from <table>
```

MIN Function

SQL uses the MIN function to find the maximum value in a column.

The syntax for using the MIN function is,

```
Select MIN<column name>
from <table>
```

MAX Function

SQL uses the MAX function to find the maximum value in a column.

The syntax for using the MAX function is,

```
Select MAX <column name>
from <table>
```

COUNT Function

Another arithmetic function is COUNT. This allows us to COUNT up the number of row in a certain table.

The syntax is,

```
Select COUNT <column name>
from <table>
```

VARIANCE Function

SQL uses the VARIAVCE function to find the variance value of a column.

The syntax for using the VARIAVCE function is,

```
Select VARIAVCE <column name>
from <table>
```

STANDARD DEVIATION Function

SQL uses the STDDEV function to find the standard deviation of a column.

The syntax for using the STDDEV function is,

```
Select STDDEV <column name>
from <table>
```

GROUP BY Function

Often applications require grouping rows that have certain properties and then applying an aggregate function on one column for each group separately. For this, SQL provides the clause *group by <group column(s)>*. This clause appears after the where clause and must refer to columns of tables listed in the from clause.

The syntax for using the GROUP BY function is,

```
select <column(s)>
from <table(s)>
where <condition>
group by <group column(s)>;
```

HAVING CLAUSE

The **HAVING** clause, which is reserved for aggregate functions. The **HAVING** clause is typically placed near the end of the SQL statement, and a SQL statement with the **HAVING** clause may or may not include the **GROUP BY** clause.

```
select <column(s)>
from <table(s)>
where <condition>
group by <group column(s)>
[having <group condition(s)>];
```

OUTPUT:

AGGREGATE FUNCTIONS

```
SQL> desc emp;
```

```
SQL> select * from emp;
```

AVG

```
SQL> select avg(sal) from emp;
```

SUM

```
SQL> select sum(sal) total_salary from emp;
```

MIN

```
SQL> select min(ename) from emp;
```

MAX

```
SQL> select max(hireDate) from emp;
```

COUNT

```
SQL> select count(mgr) from emp;
```

VARIANCE

```
SQL> select variance(sal) from emp;
```

STANDARD DEVIATION

```
SQL> select stddev(sal) from emp;
```

```
SQL> select avg(sal),sum(sal),min(sal),max(sal),count(sal) from emp  where deptno=20;
```

GROUP BY

```
SQL> select job,sum(sal) from emp group by (job);
```

HAVING CLAUSE

```
SQL>  select job,sum(sal) from emp group by (job)  having sum(sal) > 70000;
```

CHARACTER STRING

These are special commands to work on characters.

```
SQL> select * from emp;
```

INSTR

```
SQL> select ename,instr(ename,'a') from emp;
```

LENGTH

```
SQL> select ename,length(ename) from emp;
SQL> select rpad(ename,12,'*') from emp;
```

LPAD

```
SQL> select lpad(ename,12,'*') from emp;
```

REPLACE

```
SQL> select replace(ename,'ya','thi') from emp;
```

ASCII

```
SQL> select ASCII('priya') from emp where empno=4545;||
```

Function

```
SQL> select 'santh'||'ini' from emp where ename='santh';
```

UPPER

```
SQL> select upper(ename) from emp;
```

LOWER

```
SQL> select lower(ename) from emp;
```

SUBSTR

```
SQL> select ename,substr(ename,3,3) from emp;

SQL> select ename,translate(ename,'a','e') from emp;
```

STRING OPERATION

These are special commands to work on strings.

```
SQL> select ename from emp;
SQL> select ename from emp where ename like 'pri%';

SQL> select ename from emp where ename like 'Pri%';

SQL> select ename from emp where ename like 'Pri_%';

SQL> select ename from emp where ename like '%th';

SQL> select ename from emp where ename like '%th_';

SQL> select ename from emp where ename like '___th

SQL> select ename from emp where ename like '___th%';

SQL> select ename from emp where ename like '_r%';

SQL> select ename from emp where ename like '_a%';

SQL> select ename from emp where ename like '_a_th';
SQL> select ename from emp where ename like '_a_th%';
```

RESULT:

Thus the various aggregate functions using SQL has been implemented and verified.

EX.NO:9 **SET OPERATIONS**

AIM:

To implement set operations using SQL.

DESCRIPTION:

Sometimes it is useful to combine query results from two or more queries into a single result.

SQL supports three set operators which have the pattern:
<query 1><set operator><query 2>

The set operators are:

- union [all] returns a table consisting of all rows either appearing in the result of <query1> or in the result of <query 2>.
  Duplicates are automatically eliminated unless the clause all is used.

- intersect returns all rows that appear in both results <query 1> and <query 2>.

- minus returns those rows that appear in the result of <query 1> but not in the result of <query 2>.

OUTPUT:

SET OPERATIONS

```
SQL> desc tab1;
SQL> select * from tab1;
SQL> desc tab2;
SQL> select * from tab2;
```

UNION

```
SQL> select rollno from tab1 union select rollno from tab2;
```

UNION ALL

```
SQL> select rollno from tab1  union all select rollno from tab2;
```

NTERSECT

```
SQL> select rollno from tab1  intersect select rollno from tab2;
```

MINUS

```
SQL> select rollno from tab1  minus select rollno from tab2;
```

CARTESIAN PRODUCT

```
SQL> select tab1.rollno,tab2.rollno from tab1,tab2;
```

RESULT:

Thus, the set operations using SQL has been implemented and verified.

EX.NO:10 CURSORS

AIM:
To execute PL/SQL code using cursor.

DESCRIPTION:

A cursor is a SELECT statement that is defined within the *declaration* section of your PL/SQL code. We'll take a look at three different syntaxes for cursors.

A cursor must be declared and opened before it can be used, and it must be closed to deactivate it after it is no longer required. Once the cursor has been opened, the rows of the query result can be retrieved one at a time using a FETCH statement.

DECLARE CURSOR

The DECLARE CURSOR statement defines the specific SELECT to be performed and associates a cursor name with the query.

Cursor without parameters

The basic syntax for a cursor without parameters is:

```
CURSOR cursor_name
IS
SELECT_statement;
```

Cursor with parameters

The basic syntax for a cursor with parameters is:

```
CURSOR cursor_name (parameter_list)
IS
SELECT_statement;
```

OPEN CURSOR

The OPEN statement executes the query and identifies all the rows that satisfy the query search condition, and positions the cursor before the first row of this result table.

Syntax

```
OPEN <cursor name>;
```

FETCH CURSOR

The FETCH statement retrieves the next row of the active set.Syntax

```
FETCH <cursor name>

INTO {host variable [indicator var i],[]}
```

CLOSE CURSOR

The CLOSE statement is used to close the cursor that is currently open.

Syntax

```
CLOSE <cursor name>
```

OUTPUT:

CURSORS

```
SQL> desc salary
SQL> select * from salary;
SQL> set serverout on;
SQL> declare
  e_no number(6);
  e_name varchar2(10);
  net_salary number(8,2);
  cursor cur_salary is select emp_no,
  emp_name,basic+da_percent*basic/100+ma+other_allowances-deduction from salary;
  begin
  dbms_output.put_line('Emp no '||' Name '||' Net salary');
  dbms_output.put_line('---------------------------------------------');
 open cur_salary;
 loop
 fetch cur_salary into e_no,e_name,net_salary;
 exit when cur_salary%notfound;
 dbms_output.put_line(rpad(e_no,5,' ')||rpad(e_name,10,' ')||net_salary);
 end loop;
 close cur_salary;
 end;18 /
```

RESULT:

Thus the cursors using PL/SQL has been implemented and verified.

EX.NO:11

PROCEDURES

AIM:

To execute PL/SQL program with procedure.

DESCRIPTION:

The syntax for a procedure is:

```
CREATE [OR REPLACE] PROCEDURE procedure name
   [ (parameter [,parameter]) ]
IS
   [declaration_section]
BEGIN
   executable_section
[EXCEPTION
   exception_section]
END [procedure_name];
```

When you create a procedure or function, you may define parameters. There are three types of parameters that can be declared:

1. **IN** - The parameter can be referenced by the procedure or function. The value of the parameter can not be overwritten by the procedure or function.
2. **OUT** - The parameter can not be referenced by the procedure or function, but the value of the parameter can be overwritten by the procedure or function.
3. **IN OUT** - The parameter can be referenced by the procedure or function and the value of the parameter can be overwritten by the procedure or function.

The optional clause or replace re-creates the procedure. A procedure can be deleted using the command drop procedure <procedure name>.

OUTPUT:

PROCEDURES

```
SQL> CREATE OR REPLACE PROCEDURE greetings
 AS
 BEGIN
 dbms_output.put_line('Hello World!');
 END;6 /
SQL> set serverout on;
SQL> BEGIN
 greetings;
 END;4 /
SQL>  DECLARE
 a number;
 b number;
 c number;
 PROCEDURE findMin(x IN number, y IN number, z OUT number)IS
 BEGIN
 IF x < y THEN8
 z:= x;
 ELSE
 z:=
y;END IF;
END;
BEGIN
a:=&a;
b:=&b;
findMin(a, b, c);
dbms_output.put_line(' Minimum of'||a||'and'||b||'is'||c);
end;
 /
Enter value for a: 5
old  14: a:=&a;
new 14:  a:=5;
Enter value for b: 2
old  15: b:=&b;
new 15: b:=2;
```

RESULT:

Thus, the procedure using PL/SQL has been implemented and verified.

EX.NO:12 **FUNCTIONS**

AIM:

To execute PL/SQL program with function.

DESCRIPTION:

The syntax for a function is:

```
CREATE [OR REPLACE] FUNCTION function_name
   [ (parameter [,parameter]) ]
   RETURN return_datatype
IS | AS
   [declaration_section]
BEGIN
   executable_section
[EXCEPTION
   exception_section]
END [function_name]
```

When you create a procedure or function, you may define parameters. There are three types of parameters that can be declared:

1. **IN** - The parameter can be referenced by the procedure or function. The value of the parameter can not be overwritten by the procedure or function.
2. **OUT** - The parameter can not be referenced by the procedure or function, but the value of the parameter can be overwritten by the procedure or function.
3. **IN OUT** - The parameter can be referenced by the procedure or function and the value of the parameter can be overwritten by the procedure or function.

The optional clause or replace re-creates the function. A function can be deleted using the command drop function <function name>.

OUTPUT:

FUNCTIONS

```
SQL> desc phonebook;
SQL> select * from phonebook;
SQL> create or replace function findAddress(phone in number) return varchar2 as address varchar2(100);
 SQL>select username||','||doorno ||','||street ||','||place||','||pincode  into address from  phonebook where
 phone_no=phone;
 SQL>return address;
SQL> declare
 address varchar2(100);
 begin
 address:=findaddress(25301);
 dbms_output.put_line(address);
 end;7 /
```

```
SQL> declare
 address varchar2(100);
 begin
 address:=findaddress(25601);
 dbms_output.put_line(address);
 end;7 /
```

RESULT:

Thus the functions has been implemented and verified using PL/SQL.

EX.NO:13

CONTROL STRUCTURES

AIM:
To implement control structures using PL/SQL.

DESCRIPTION:

IF STATEMENTS

The structure of the PL/SQL IF statement is similar to the structure of IF statements in other procedural languages. It allows PL/SQL to perform actions selectively based on conditions.

There are three forms of IF statements:

- IF-THEN-END IF

  Syntax:

```
IF condition THEN
     Statements;
END IF;
```

- IF-THEN-ELSE-END IF

  Syntax:

```
IF condition THEN
     Statement 1;
ELSE
     Statement 2;
END IF;
```

- IF-THEN-ELSIF-END IF

  Syntax:

```
IF condition THEN
     Statement 1;
[ELSIF condition THEN
     Statement 2;]
……
[ELSIF
     Statements n;]
END IF;
```

If the condition is FALSE or NULL, PL/SQL ignores the statements in the IF block. In either case, control resumes at the next statement in the program following the END IF.

FOR LOOP

FOR loops have the general structure as the basic loop. In addition they have a control statement before the LOOP keyword to determine the number of iterations that PL/SQL performs.

The syntax for the for loop is

```
FOR counter IN [REVERSE]
          Lower_bound…..upper_bound LOOP
          Statement 1;
          Statement 2;
```

```
END LOOP;
```

COUNTER – An explicitly declared integer whose value automatically increases by 1 on each iteration of the loop until the upper or lower bound is reached.
REVERSE – Causes the counter to decrement with each iteration from the upper bound to the lower bound.
LOWER_BOUND – The lower bound for the range of counter values.
UPPER_BOUND – The upper bound for the range of counter values.

WHILE LOOP

The WHILE loop is used to repeat a sequence of statements until the controlling condition is no longer true. The condition is evaluated at the start of each iteration. The loop terminates when the condition is false. If the condition is false at the start of the loop, then no futher iterations are performed.
The syntax for while loop is

```
WHILE condition LOOP
        Statement 1;
        Statement 2;
END LOOP;
```

OUTPUT:
CONTROL STRUCTURES

```
SQL> set serveroutput on;
SQL> declare
  2  a number;
  3  b number;
  4  c number;
  5  begin
  6 a:=&a;
  7  b:=&b;
  8  c:=a+b;
  9  dbms_output.put_line('sum of'||a||'and'||b||'is'||c);
 10  end;
 11 /

Enter value for a: 34
old  6: a:=&a;
new   6:  a:=34;
Enter value for b: 56
old  7: b:=&b;
new   7:  b:=56;
sum of34and56is90
```

IF…..THEN…..ENDIF

```
SQL> set serveroutput on;
SQL> declare
  2  a number;
  3  b number;
  4  begin
  5  a := &a;
  6 if a > 40 then
  7 b := a - 40;
  8  dbms_output.put_line('Calculated value :: ' || b);
  9  end if;
 10  end;
 11 /
Enter value for a: 45
old   5:  a := &a;
new 5: a := 45;
Calculated value :: 5

SQL> /
Enter value for a: 23
old   5:  a := &a;
new 5: a := 23;
```

<u>IF…..THEN…..ELSE…..ENDIF</u>

```
SQL> set serveroutput on;

SQL> declare
  2  a number;
  3  b number;
  4  begin
  5  a := &a;
  6 if a > 40 then
  7 b := a - 40;
  8  dbms_output.put_line('Calculated value :: ' || b);
  9  else
 10  dbms_output.put_line('No calculated value');
 11  end if;
 12  end;
 13 /
Enter value for a: 67
old   5:  a := &a;
new 5: a := 67;
Calculated value :: 27



SQL> /
Enter value for a: 34
old   5:  a := &a;
new 5: a := 34;
No calculated value
```

<u>ELSIF LADDER</u>

```
SQL> set serveroutput on;
SQL> declare
  2   a number;
  3   b number;
  4   c number;
  5   d number;
  6   begin
  7  a:=&a;
  8  b:=&b;
  9  c:=&b;
```

```
 10   if(a>b)and(a>c) then
 11   dbms_output.put_line('A is maximum');
 12   elsif(b>a)and(b>c)then
 13   dbms_output.put_line('B is maximum');
 14   else
 15   dbms_output.put_line('C is maximum');
 16   end if;
 17   end;
 18 /

Enter value for a: 34
old  7: a:=&a;
new   7:  a:=34;
Enter value for b: 56
old  8: b:=&b;
new   8:  b:=56;
Enter value for b: 45
old  9: c:=&b;
new 9:  c:=45;
B is maximum
```

FOR LOOP

```
SQL> set serveroutput on;

SQL> declare
  2  n number;
  3  sum1 number default 0;
  4  endvalue number;
  5  begin
  6  endvalue:=&endvalue;
  7  n:=1;
  8  for n in 1..endvalue
  9  loop
 10  if mod(n,2)=1
 11  then
 12  sum1:=sum1+n;
 13  end if;
 14  end loop;
 15  dbms_output.put_line('sum ='||sum1);
 16  end;
 17 /
Enter value for endvalue: 5
old  6: endvalue:=&endvalue;
new  6: endvalue:=5;
sum =9


SQL> /
Enter value for endvalue: 56
old  6: endvalue:=&endvalue;
new  6: endvalue:=56;
sum =784
```

WHILE…..LOOP

```
SQL> set serveroutput on;
SQL> declare
  2  n number;
  3  sum1 number default 0;
  4  endvalue number;
  5  begin
  6  endvalue:=&endvalue;
  7  n:=1;
  8  while(n<endvalue)
  9  loop
 10  sum1:=sum1+n;
 11  n:=n+2;
 12  end loop;
 13  dbms_output.put_line('sum of odd no. between 1 and' ||endvalue||'is'||sum1);
 14  end;
 15 /
Enter value for endvalue: 50
old  6: endvalue:=&endvalue;
new  6: endvalue:=50;
sum of odd no. between 1 and50is625 P
```

RESULT:

Thus, the control structures has been implemented using PL/SQL.

EX.NO:14 TRIGGERS

AIM:

To implement PL/SQL program using triggers.

DESCRIPTION:

Complex integrity constraints that refer to several tables and attributes (as they are known as assertions in the SQL standard) cannot be specified within table definitions.

Triggers, in contrast, provide a procedural technique to specify and maintain integrity constraints. Triggers even allow users to specify more complex integrity constraints since a trigger essentially is a PL/SQL procedure. Such a procedure is associated with a table and is automatically called by the database system whenever a certain modification (event) occurs on that table. Modifications on a table may include insert, update, and delete operations.

Structure of Triggers

A trigger definition consists of the following (optional) components:

• trigger name

```
create [or replace] trigger <trigger name>
```

• trigger time point

```
before | after
```

• triggering event(s)

```
insert or update [of <column(s)>] or delete on <table>
```

• trigger type (optional)

```
for each row
```

• trigger restriction (only for each row triggers !)

```
when (<condition>)
```

• trigger body

```
<PL/SQL block>
```

Below is the syntax for creating a trigger in Oracle

```
CREATE [OR REPLACE] TRIGGER <trigger_name>

  {BEFORE|AFTER} {INSERT|DELETE|UPDATE} ON <table_name>

  [REFERENCING [NEW AS <new_row_name>] [OLD AS <old_row_name>]]

  [FOR EACH ROW [WHEN (<trigger condition>)]]

  <trigger_body>
```

Some important points to note:

- You can create only BEFORE and AFTER triggers for tables. (INSTEAD OF triggers are only available for views; typically they are used to implement view upDates.)

- You may specify up to three triggering events using the keyword OR. Furthermore, UPDATE can be optionally followed by the keyword OF and a list of attribute(s) in <table_name>. If present, the OF clause defines the event to be only an upDate of the attribute(s) listed after OF.

Here are some examples:

```
... INSERT ON R ...
   ... INSERT OR DELETE OR UPDATE ON R ...
…UPDATE OF A, B OR INSERT ON R ...
```

- If FOR EACH ROW option is specified, the trigger is row-level; otherwise, the trigger is statement-level.
-

- Only for row-level triggers:
    - The special variables NEW and OLD are available to refer to new and old tuples respectively.

      Note: In the trigger body, NEW and OLD must be preceded by a colon (":"), but in the WHEN clause, they do not have a preceding colon! See example below.

    - The REFERENCING clause can be used to assign aliases to the variables NEW and OLD.
    - A trigger restriction can be specified in the WHEN clause, enclosed by parentheses. The trigger restriction is a SQL condition that must be satisfied in order for Oracle to fire the trigger. This condition cannot contain subqueries. Without the WHEN clause, the trigger is fired for each row.

- <trigger_body> is a PL/SQL block, rather than sequence of SQL statements. Oracle has placed certain restrictions on what you can do in <trigger_body>, in order to avoid situations where one trigger performs an action that triggers a second trigger, which then triggers a third, and so on, which could potentially create an infinite loop. The restrictions on <trigger_body> include:
    - You cannot modify the same relation whose modification is the event triggering the trigger.
    - You cannot modify a relation connected to the triggering relation by another constraint such as a foreign-key constraint.

The clause replace re-creates a previous trigger definition having the same <trigger name>.

The clause replace re-creates a previous trigger definition having the same <trigger name>. The name of a trigger can be chosen arbitrarily, but it is a good programming style to use a trigger name that reflects the table and the event(s), e.g., upd ins EMP. A trigger can be invoked before or after the triggering event. The triggering event specifies before (after) which operations on the table <table> the trigger is executed. A single event is an insert, an
update, or a delete; events can be combined using the logical connective or. If for an update
trigger no columns are specified, the trigger is executed after (before) <table> is updated. If
the trigger should only be executed when certain columns are upDated, these columns must be specified after the event update.

OUTPUT:

TRIGGERS

```
SQL> select * from empa;

SQL> desc empa;
```

TRIGGER AFTER UPDATE

```
SQL> set serverout on;

sql> create or replace trigger trig1

after update or insert or delete on empa

for each row

begin

if updating then

dbms_output.put_line('table updated');

elsif inserting then

dbms_output.put_line('table inserted');

elsif deleting then
```

```
 10  dbms_output.put_line('table deleted');
 11  end if;
 12  end;
 13 /
SQL> insert into empa values(9,'ram',9000,900,909);
SQL> select * from empa;
SQL> update empa set savings=800 where name='nandhu';
SQL> select * from empa;
SQL> delete from empa where name='kumar';
SQL> select * from empa;
```

TRIGGER BEFORE UPDATE

```
SQL> select * from empa;
SQL> set serverout on;
SQL> create or replace trigger trig2
  2  before update or insert or delete on empa
  3  for each row
  4  begin
  5  if updating then
  6  dbms_output.put_line('table updated');
  7  elsif inserting then
  8  dbms_output.put_line('table inserted');
  9  elsif deleting then
 10  dbms_output.put_line('tuple deleted');
 11  end if;
 12  end;
 13 /
SQL> insert into empa values (1,'valli',9999,980,800);
SQL> select * from empa;
SQL> update empa set expence=100 where name='valli';
SQL> select * from empa;
SQL> delete from empa where name='venky';
SQL> select * from empa;
```

RESULT:

Thus, the triggers has been implemented using PL/SQL.

EX.NO:15 **XML DATABAS**

AIM:

To study the XML database and its types

XML

XML stands for **Ex**tensible **M**arkup **L**anguage and is a text-based markup language derived from Standard Generalized Markup Language (SGML).

XML tags identify the data and are used to store and organize the data, rather than specifying how to display it like HTML tags, which are used to display the data. XML is not going to replace HTML in the near future, but it introduces new possibilities by adopting many successful features of HTML.

There are three important characteristics of XML that make it useful in a variety of systems and solutions −

XML is extensible − XML allows you to create your own self-descriptive tags, or language, that suits your application.

XML carries the data, does not present it − XML allows you to store the data irrespective of how it will be presented.

XML is a public standard − XML was developed by an organization called the World Wide Web Consortium (W3C) and is available as an open standard.

XML Database is used to store huge amount of information in the XML format. As the use of XML is increasing in every field, it is required to have a secured place to store the XML documents. The data stored in the database can be queried using **XQuery**, serialized, and exported into a desired format.

XML Database Types

There are two major types of XML databases −

- XML- enabled
- Native XML (NXD)

XML - Enabled Database

XML enabled database is nothing but the extension provided for the conversion of XML document. This is a relational database, where data is stored in tables consisting of rows and columns. The tables contain set of records, which in turn consist of fields.

Native XML Database

Native XML database is based on the container rather than table format. It can store large amount of XML document and data. Native XML database is queried by the **XPath**-expressions.

Native XML database has an advantage over the XML-enabled database. It is highly capable to store, query and maintain the XML document than XML-enabled database.

Example

Following example demonstrates XML database −

```
<?xml version = "1.0"?><contact-info>
  <contact1>
    <name>Thivagar </name>
    <company>TCS</company>
    <phone>(011) 123-4567</phone>
  </contact1><contact2>
    <name>Manisha </name>
    <company>Infosis</company>
    <phone>(011) 789-4567</phone>
  </contact2></contact-info>
```

Here, a table of contacts is created that holds the records of contacts (contact1 and contact2), which in turn consists of three entities − *name, company* and *phone*.

XML Schema is commonly known as **XML Schema Definition (XSD)**. It is used to describe and validate the structure and the content of XML data. XML schema defines the elements, attributes and data types. Schema element supports Namespaces. It is similar to a database schema that describes the data in a database.

Syntax

You need to declare a schema in your XML document as follows −

Example

The following example shows how to use schema −

```
<?xml version = "1.0" encoding = "UTF-8"?><xs:schema xmlns:xs =
"http://www.w3.org/2001/XMLSchema">
  <xs:element name = "contact">
    <xs:complexType>
      <xs:sequence>
        <xs:element name = "name" type = "xs:string" />
        <xs:element name = "company" type = "xs:string" />
        <xs:element name = "phone" type = "xs:int" />
      </xs:sequence>
    </xs:complexType>
  </xs:element></xs:schema>
```

The basic idea behind XML Schemas is that they describe the legitimate format that an XML document can take.

Conclusion :

Thus the XML database and its types were studied.

EX.NO:16 DOCUMENT BASED DATA USING NOSQL DATABASE TOOL

AIM : To study the creation of document based data using Nosql Database Tools

Document based data

A document database is a type of nonrelational database that is designed to store and query data as JSON-like documents. Document databases make it easier for developers to store and query data in a database by using the same document-model format they use in their application code. The flexible, semistructured, and hierarchical nature of documents and document databases allows them to evolve with applications' needs. The document model works well with use cases such as catalogs, user profiles, and content management systems where each document is unique and evolves over time. Document databases enable flexible indexing, powerful ad hoc queries, and analytics over collections of documents.

In the following example, a JSON-like document describes a book.

```
[
 {
  "year" : 2013,
  "title": "Turn It Down, Or Else!",
  "info": {
  "directors": [ "Alice Smith", "Bob Jones"],
 "release_date":"2013-01-18T00:00:00Z",
 "rating" : 6.2,
 "genres" : ["Comedy", "Drama"],
 "image_url" "http://ia.media-
imdb.com/images/N/09ERWAU7FS797AJ7LUSHN09AMUP908RL105JF90EWR7
LJKQ7@@._V1_SX400_.jpg",
}

"plot" "A rock band plays their music at high volumes, annoying the neighbors.",

"actors": ["David Matthewman", "Jonathan G: Neff"]
}
}


{
"year": 2015,
"title": "The Big New Movie",
"info": {
}
"plot": "Nothing happens at all.",
"rating": 0
}
}
}
```

The following is an example of a document that might appear in a document
database like MongoDB. This sample document represents a company contact card,
describing
an employee called Sammy:

```
{
"_id": "sammyshark",
"firstName": "Sammy",
"lastName": "Shark",
"email": "sammy.shark@digitalocean.com",

"department": "Finance"
}
```

Notice that the document is written as a JSON object. JSON is a human-readable data format that has become quite popular in recent years. While many different formats can be used to represent data within a document database, such as XML or YAML, JSON is one of the most common choices. For example, MongoDB adopted JSON as the primary data format to define and manage data.

All data in JSON documents are represented as field-and-value pairs that take form of field: value. In the previous example, the first line shows an _id field with the value sammyshark. The example also includes fields for the employee's first and last names ,their email address as well as as what department they work in.

Field names allow us to understand what kind of data is held within a document just a glance. Documents in document databases are self-describing,which means they contain both the data values as well as the information on what kind of data is being stored. When retrieving a document from the database, we always get the whole picture.

Conclusion :

Thus the document based data using Nosql Database Tools were studied.

EX.NO :17 BANKING SYSTEM

AIM:

To design and implement any one of the module of Banking system with visual basic as front end and sql plus as back end.

PROCEDURE :

1. **Setup an ODBC Data source**
    a. Choose the **Administrative Tools** icon from the Windows **Control Panel**.
    b. Choose the **Data Sources** shortcut.
    c. In the **ODBC Data Source Administrator** dialog box, click **Add**, then select the oracle provider for OLEDB and choose OK.
    d. In the **Setup** dialog box, set option values as necessary and choose **OK**.

2. **Creating an ADO Data Control**

PROGRAM :

```
Private Sub Command1_Click()
Adodc1.Recordset.AddNew
End Sub

Private Sub Command2_Click()
Adodc1.Recordset.Update
MsgBox ("Updated")
End Sub

Private Sub Command3_Click()
Adodc1.Recordset.Delete
MsgBox ("Deleted")
End Sub
Private Sub Command4_Click()
End
End Sub
```

CONCLUSION :

Thus the application program for banking system is done successfully using visual basic as front end and sql plus as back end.