NEHRU COLLEGE OF ENGINEERING AND RESEARCH CENTRE

(NAACAccredited)

(Approved by AICTE , Affiliated to APJ Abdul Kalam Technological University, Kerala)

Pampady, Thiruvilwamala(PO), Thrissur(DT), Kerala 680 588

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

LAB MANUAL

CS333 APPLICATION SOFTWARE DEVELOPMENT LAB

VISION OF THE INSTITUTION

To mould true citizens who are millennium leaders and catalysts of change through excellence in education.

MISSION OF THE INSTITUTION

NCERC is committed to transform itself into a center of excellence in Learning and Research in Engineering and Frontier Technology and to impart quality education to mould technically competent citizens with moral integrity, social commitment and ethical values.

We intend to facilitate our students to assimilate the latest technological know-how and to imbibe discipline, culture and spiritually, and to mould them in to technological giants, dedicated research scientists and intellectual leaders of the country who can spread the beams of light and happiness among the poor and the underprivileged.

ABOUT THE DEPARTMENT

- Established in: 2002
- Course offered : B.Tech in Computer Science and Engineering

  M.Tech in Computer Science and Engineering

  M.Tech in Cyber Security
- Approved by AICTE New Delhi and Accredited by NAAC
- Certified by ISO 9001-2015
- Affiliated to A P J Abdul Kalam Technological University, Kerala.

DEPARTMENT MISSION

Producing Highly Competent, Innovative and Ethical Computer Science and Engineering Professionals to facilitate continuous technological advancement.

DEPARTMENT MISSION

1. To Impart Quality Education by creative Teaching Learning Process
2. To Promote cutting-edge Research and Development Process to solve real world problems with emerging technologies.
3. To Inculcate Entrepreneurship Skills among Students.
4. To cultivate Moral and Ethical Values in their Profession.

PROGRAMME EDUCATIONAL OBJECTIVES

PEO1: Graduates will be able to Work and Contribute in the domains of Computer Science and Engineering through lifelong learning.

PEO2: Graduates will be able to Analyse, design and development of novel Software Packages, Web Services, System Tools and Components as per needs and specifications.

PEO3: Graduates will be able to demonstrate their ability to adapt to a rapidly changing environment by learning and applying new technologies.

PEO4: Graduates will be able to adopt ethical attitudes, exhibit effective communication skills, Teamworkand leadership qualities.

PROGRAM OUTCOMES (POs)

Engineering Graduates will be able to:

1. **Engineering knowledge**: Apply the knowledge of mathematics, science, engineering fundamentals, and an engineering specialization to the solution of complex engineering problems.
2. **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.
3. **Design/development of solutions**: Design solutions for complex engineering problems and design system components or processes that meet the specified needs with appropriate consideration for the public health and safety, and the cultural, societal, and environmental considerations.
4. **Conduct investigations of complex problems**: Use research-based knowledge and research methods including design of experiments, analysis and interpretation of data, and synthesis of the information to provide valid conclusions.
5. **Modern tool usage**: Create, select, and apply appropriate techniques, resources, and modern engineering and IT tools including prediction and modeling to complex engineering activities with an understanding of the limitations.
6. **The engineer and society**: Apply reasoning informed by the contextual knowledge to assess societal, health, safety, legal and cultural issues and the consequent responsibilities relevant to the professional engineering practice.
7. **Environment and sustainability**: Understand the impact of the professional engineering solutions in societal and environmental contexts, and demonstrate the knowledge of, and need for sustainable development.
8. **Ethics**: Apply ethical principles and commit to professional ethics and responsibilities and norms of the engineering practice.
9. **Individual and team work**: Function effectively as an individual, and as a member or leader in diverse teams, and in multidisciplinary settings.
10. **Communication**: Communicate effectively on complex engineering activities with the engineering community and with society at large, such as, being able to comprehend and write effective reports and design documentation, make effective presentations, and give and receive clear instructions.
11. **Project management and finance**: Demonstrate knowledge and understanding of the engineering and management principles and apply these to one's own work, as a member and leader in a team, to manage projects and in multidisciplinary environments.
12. **Life-long learning**: Recognize the need for, and have the preparation and ability to engage in independent and life-long learning in the broadest context of technological change.

PROGRAM SPECIFIC OUTCOMES (PSO)

PSO1: Ability to Formulate and Simulate Innovative Ideas to provide software solutions for Real-time Problems and to investigate for its future scope.

PSO2: Ability to learn and apply various methodologies for facilitating development of high quality System Software Tools and Efficient Web Design Models with a focus on performance optimization.

PSO3: Ability to inculcate the Knowledge for developing Codes and integrating hardware/software products in the domains of Big Data Analytics, Web Applications and Mobile Apps to create innovative career path and for the socially relevant issues.

COURSE OUTCOME

C333.1	Design a database for a given problem using database design principles.
C333.2	Implement database for a given problem.
C333.3	Apply stored programming concepts (PL-SQL) using Cursors and Triggers.
C333.4	Use graphical user interface , Event Handling and Database connectivity to develop and deploy applications.
C333.5	Use graphical user interface, Event Handling and Database connectivity to develop and deploy applets.
C333.6	Develop medium-sized project in a team.

MAPPING OF COURSE OUTCOMES WITH PROGRAM OUTCOMES

CO'S	PO1	PO2	PO3	PO4	PO5	PO6	PO7	PO8	PO9	PO10	PO11	PO12
C333.1			3	3								
C333.2			3									
C333.3		3	3	3	2							
C333.4		3	3	3								
C333.5		3	3	3								
C333.6			3	3					3			
C333	-	3	3	3	2	-	-	-	3	-	-	-

MAPPING OF COURSE OUTCOMES WITH PROGRAM SPECIFIC OUTCOMES

CO'S	PSO1	PSO2	PSO3
C333.1		3	
C333.2			3
C333.3	3		
C333.4	3		
C333.5	3		
C333.6			3
C333	3	3	3

Note: H-Highly correlated=3, M-Medium correlated=2, L-Less correlated=1

PREPARATION FOR THE LABORATORY SESSION
GENERAL INSTRUCTIONS TO STUDENTS

1. Read carefully and understand the description of the experiment in the lab manual. You may go to the lab at an earlier date to look at the experimental facility and understand it better. Consult the appropriate references to be completely familiar with the concepts and hardware.

2. Make sure that your observation for previous week experiment is evaluated by the faculty member and you have transferred all the contents to your record before entering to the lab/workshop.

3. At the beginning of the class, if the faculty or the instructor finds that a student is not adequately prepared, they will be marked as absent and not be allowed to perform the experiment.

4. Bring necessary material needed (writing materials, graphs, calculators, etc.) to perform the required preliminary analysis. It is a good idea to do sample calculations and as much of the analysis as possible during the session. Faculty help will be available. Errors in the procedure may thus be easily detected and rectified.

5. Please actively participate in class and don't hesitate to ask questions. Please utilize the teaching assistants fully. To encourage you to be prepared and to read the lab manual before coming to the laboratory, unannounced questions may be asked at any time during the lab.

6. Carelessness in personal conduct or in handling equipment may result in serious injury to the individual or the equipment. Always be on the alert for strange sounds

7. Students must follow the proper dress code inside the laboratory.

9. Maintain silence, order and discipline inside the lab. Don't use cell phones inside the laboratory.

10. Any injury no matter how small must be reported to the instructor immediately.

11. Check with faculty members one week before the experiment to make sure that you have the handout for that experiment and all the apparatus.

AFTER THE LABORATORY SESSION

1. Clean up your work area.

2. Check with the technician before you leave.

3. Make sure you understand what kind of report is to be prepared and due submission of record is next lab class.

4. Do sample calculations and some preliminary work to verify that the experiment was successful

MAKE-UPS AND LATE WORK

Students must participate in all laboratory exercises as scheduled. They must obtain permission from the faculty member for absence, which would be granted only under justifiable

circumstances. In such an event, a student must make arrangements for a make-up laboratory, which will be scheduled when the time is available after completing one cycle. Late submission will be awarded less mark for record and internals and zero in worst cases.

LABORATORY POLICIES

1. Food, beverages & mobile phones are not allowed in the laboratory at any time.

2. Do not sit or place anything on instrument benches.

3. Organizing laboratory experiments requires the help of laboratory technicians and staff. Be punctual.

SYLLABUS

Course code	Course Name	L-T-P - Credits	Year of Introduction
CS333	APPLICATION SOFTWARE DEVELOPMENT LAB	0-0-3-1	2016

Pre-requisite : CS208 Principles of Database Design

Course Objectives

- To introduce basic commands and operations on database.
- To introduce stored programming concepts (PL-SQL) using Cursors and Triggers .
- To familiarize front end tools of database.

List of Exercises/Experiments: (Exercises/experiments marked with * are mandatory. Total 12 Exercises/experiments are mandatory)

1. Creation of a database using DDL commands and writes DQL queries to retrieve information from the database.
2. Performing DML commands like Insertion, Deletion, Modifying, Altering, and Updating records based on conditions.
3. Creating relationship between the databases. *
4. Creating a database to set various constraints. *
5. Practice of SQL TCL commands like Rollback, Commit, Savepoint.
6. Practice of SQL DCL commands for granting and revoking user privileges.
7. Creation of Views and Assertions *
8. Implementation of Build in functions in RDBMS *
9. Implementation of various aggregate functions in SQL *
10. Implementation of Order By, Group By& Having clause. *
11. Implementation of set operators, nested queries and Join queries *
12. Implementation of various control structures using PL/SQL *
13. Creation of Procedures and Functions *
14. Creation of Packages *
15. Creation of database Triggers and Cursors *
16. Practice various front-end tools and report generation.
17. Creating Forms and Menus

18. Mini project (Application Development using Oracle/ MySQL using Database connectivity)*
    a. Inventory Control System.
    b. Material Requirement Processing.
    c. Hospital Management System.
    d. Railway Reservation System.
    e. Personal Information System.
    f. Web Based User Identification System.
    g. Timetable Management System.
    h. Hotel Management System.

Expected Outcome

The students will be able to

i. Design and implement a database for a given proble\\\\m using database design principles.
ii. Apply stored programming concepts (PL-SQL) using Cursors and Triggers.
iii. Use graphical user interface, Event Handling and Database connectivity to develop and deploy applications and applets.
iv. Develop medium-sized project in a team.

INDEX

Exp.No 1	**Familiarization of Database with DDL and DML Commands.**	**Date……**

AIM

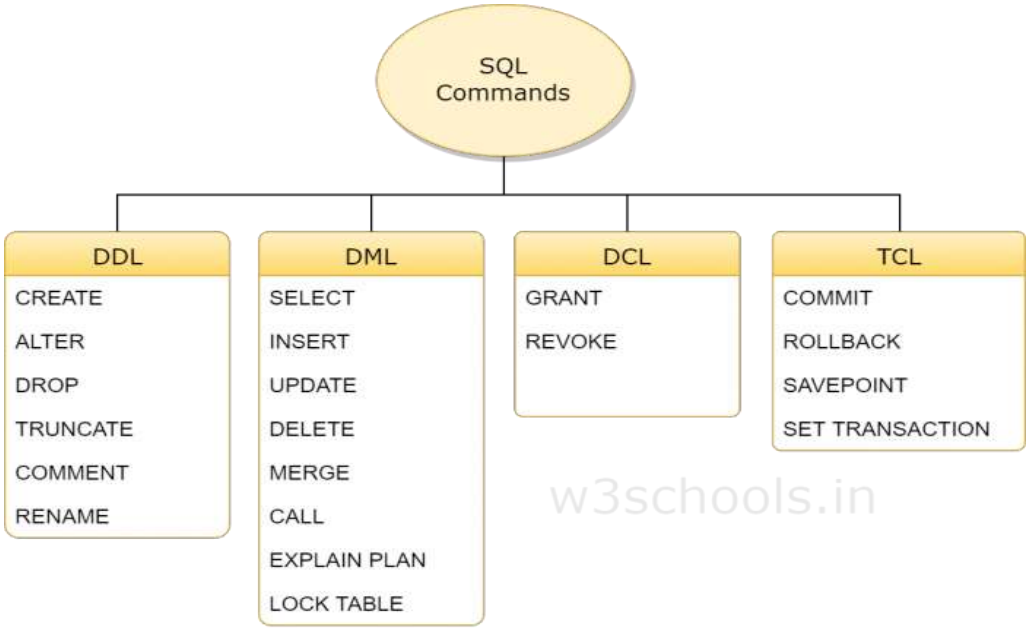
To familiarize Database with DDL and DML Commands.

THEORY

Database is a collection of related data and data is a collection of facts and figures that can be processed to produce information. Mostly data represents recordable facts. Data aids in producing information, which is based on facts

A **database management system** stores data in such a way that it becomes easier to retrieve, manipulate, and produce information. SQL is a standard language for storing, manipulating and retrieving data in databases. SQL softwares are MySQL, SQL Server, MS Access, Oracle etc.,

SQL commands can be divided into three subgroups, DDL, DML and DCL.



1. DDL:

DDL is short name of **Data Definition Language,** which deals with database schemas and descriptions, of how the data should reside in the database.

- CREATE – to create database and its objects like (table, index, views, store procedure, function and triggers)
- ALTER – alters the structure of the existing database
- DROP – delete objects from the database
- TRUNCATE – remove all records from a table, including all spaces allocated for the records are removed
- COMMENT – add comments to the data dictionary
- RENAME – rename an object

2. DML:

DML is short name of **Data Manipulation Language** which deals with data manipulation, and includes most common SQL statements such SELECT, INSERT, UPDATE, DELETE etc, and it is used to store, modify, retrieve, delete and update data in database.

- SELECT – retrieve data from the a database
- INSERT – insert data into a table
- UPDATE – updates existing data within a table
- DELETE – Delete all records from a database table
- MERGE – UPSERT operation (insert or update)
- CALL – call a PL/SQL or Java subprogram
- EXPLAIN PLAN – interpretation of the data access path
- LOCK TABLE – concurrency Control

ALGORITHM

Step 1: Start

Step 2: Create a database using DDL commands.

Step 3: Create a table with required attributes.

Step 4: Insert values into the Table using DML commands.

Step 5: Retrieve any rows using SELECT with restrictions.

Step 6: Update, Alter and Delete any rows.

Step 7: Stop

QUERIES

1. Create Database

mysql>create database employee;

2. Create Table:

mysql>create table customers (empid int(10) primary key, fname varchar(20),age int(10),address varchar(20),salary int(10));

3. Insert:

mysql>Insert into employee values(1,'anu',23,'palakkad',20000);

4. Select Commands:

mysql>Select * from employees;

mysql>Select distinct fname from employees;

mysql>Select empid, fname, age from employees;

mysql>Select fname from employees where empid=01;

mysql>Select fname, lname from employees where salary='30000';

mysql>Select fname from employees where start_date='11-05-05';

mysql>Select fname, lname from employees where salary>30000;

mysql>Select fname, lname from employees where salary<30000;

mysql>Select fname, lname from employees where salary>=30000;

mysql>Select fname, lname from employees where salary<=30000;

mysql>Select fname, lname from employees where salary<>30000;

mysql>Select lname from employees where salary between 25000 and 35000;

mysql>Select fname from employees where salary in (20000, 30000, 35000);

mysql>Select fname from employees where fname like 'a%';

5. Update:

mysql>update employee set lname='lalu' where eno=1;

6. Alter:

mysql>Alter employee add addr varchar(20);

7. Drop:

mysql>Drop table employee;

EXPECTED OUTCOME

Thus the data DDL and DML commands were performed and implemented successfully.

VIVA QUESTIONS

1. What are the categories of SQL command?
2. What is the difference between Drop, Delete and Truncate statements in SQL Server?
3. Explain DCL and TCL commands?

Exp.No 2	Creating relationship between the databases.	Date……

AIM

To create Databases and implement relationship between databases.

THEORY

A join is an SQL operation performed to establish a connection between two or more database tables based on matching columns, thereby creating a relationship between the tables. Most complex queries in an SQL database management system involve join commands. **Types of Joins:**

1. ***INNER JOIN OR EQUIJOIN:***

The most important and frequently used of the joins is the **INNER JOIN**. They are also referred to as an **EQUIJOIN**.

The INNER JOIN creates a new result table by combining column values of two tables (table1 and table2) based upon the join-predicate. The query compares each row of table1 with each row of table2 to find all pairs of rows which satisfy the join-predicate. When the join-predicate is satisfied, column values for each matched pair of rows of A and B are combined into a result row.

Syntax:

The basic syntax of the **INNER JOIN** is as follows.

```
SELECT table1.column1, table2.column2...
FROM table1
INNER JOIN table2
ON table1.common_field = table2.common_field;
```

2. ***LEFT JOIN:***

The SQL **LEFT JOIN** returns all rows from the left table, even if there are no matches in the right table. This means that if the ON clause matches 0 (zero) records in the right table; the join will still return a row in the result, but with NULL in each column from the right table.

This means that a left join returns all the values from the left table, plus matched values from the right table or NULL in case of no matching join predicate.

Syntax:

The basic syntax of a **LEFT JOIN** is as follows.

```
SELECT table1.column1, table2.column2...
FROM table1
LEFT JOIN table2
ON table1.common_field = table2.common_field;
```

3. RIGHT JOIN:

The SQL **RIGHT JOIN** returns all rows from the right table, even if there are no matches in the left table. This means that if the ON clause matches 0 (zero) records in the left table; the join will still return a row in the result, but with NULL in each column from the left table.

This means that a right join returns all the values from the right table, plus matched values from the left table or NULL in case of no matching join predicate.

Syntax:

The basic syntax of a **RIGHT JOIN** is as follow.

```
SELECT table1.column1, table2.column2...
FROM table1
RIGHT JOIN table2
ON table1.common_field = table2.common_field;
```

4. FULL JOIN:

The SQL **FULL JOIN** combines the results of both left and right outer joins. The joined table will contain all records from both the tables and fill in NULLs for missing matches on either side.

Syntax:

The basic syntax of a **FULL JOIN** is as follows −

```
SELECT table1.column1, table2.column2...
FROM table1
FULL JOIN table2
ON table1.common_field = table2.common_field;
```

5. ***SELF JOIN:***

The SQL **SELF JOIN** is used to join a table to itself as if the table were two tables; temporarily renaming at least one table in the SQL statement.

Syntax:

The basic syntax of SELF JOIN is as follows −

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

6. ***CARTESIAN JOIN OR CROSS JOIN:***

The CARTESIAN JOIN or CROSS JOIN returns the Cartesian product of the sets of records from two or more joined tables. Thus, it equates to an inner join where the join-condition always evaluates to either True or where the join-condition is absent from the statement.

Syntax:

The basic syntax of the **CARTESIAN JOIN** or the **CROSS JOIN** is as follows −

```
SELECT table1.column1, table2.column2...
FROM  table1, table2 [, table3 ]
```

ALGORITHM

STEP 1: Start the program.

STEP 2: Create two different tables with its essential attributes.

STEP 3: Insert attribute values into the table.

STEP 4: Create the table object for easy reference.

STEP 5: Join two tables by using JOIN operator.

STEP 6: Display the result of the result table.

STEP 7: Stop the program.

QUERIES

mysql> create table customers (id int(10) primary key,name varchar(20),age int(10),address varchar(20),salary int(10));

mysql> insert into customers values(1,'anu',25,'pkd',25000);

mysql> insert into customers values(2,'sanu',25,'pkd',25500);

mysql> insert into customers values(3,'vinu',25,'pkd',25500);

mysql> select * from customers;

mysql> create table orders(oid int(10),date date,custid int(10),amount int(10));

mysql> insert into orders values(100,'2015-11-2',2,300);

mysql> insert into orders values(200,'2015-10-5',1,500);

mysql> insert into orders values(200,'2015-6-5',4,600);

mysql> select * from orders;

1. Inner Join:

mysql> select id,name,Date,amount from customers inner join orders on customers.id=orders.custid;

2. Left Join:

mysql> select id,name,amount,date from customers left join orders on customers.id=orders.custid;

3.Right Join:

mysql> select id,name,amount,date from customers right join orders on customers.id=orders.custid;

4.Cartesian Join:

mysql> select id,name,amount,date from customers,orders;

5.Self Join:

mysql> select a.id,b.name,a.salary from customers a,customers b where a.salary<b.salary;

6.Full Join:

mysql> select id,name,date,amount from customers full join orders on orders.custid;

EXPECTED OUTCOME

Students are able to know the relationship of databases and important of joins**.**

VIVA QUESTIONS

1. What is the use of join in SQL?

2. What is a cross join?

3. What is the difference between inner join and left join?

4. What is self join in sql?

Exp.No 3	Creating a database to set various constraints	Date……

AIM

To create a database and set various constraints.

THEORY

Constraints are the rules enforced on the data columns of a table. These are used to limit the type of data that can go into a table. This ensures the accuracy and reliability of the data in the database. Constraints could be either on a column level or a table level. The column level constraints are applied only to one column, whereas the table level constraints are applied to the whole table.

- **NOT NULL Constraint**: Ensures that a column cannot have a NULL value.
- **DEFAULT Constraint**: Provides a default value for a column when none is specified.
- **UNIQUE Constraint**: Ensures that all values in a column are different.
- **PRIMARY Key**: Uniquely identifies each row/record in a database table.
- **FOREIGN Key**: Uniquely identifies row/record in any of the given database tables.
- **CHECK Constraint**: The CHECK constraint ensures that all the values in a column
  satisfies certain conditions.
- **INDEX**: Used to create and retrieve data from the database very quickly.

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

1. NOT NULL Constraint

By default, a column can hold NULL values. If you do not want a column to have a NULL value, then you need to define such a constraint on this column specifying that NULL is now not allowed for that column. A NULL is not the same as no data, rather, it represents unknown data.

2.DEFAULT Constraint

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

3.UNIQUE Constraint

The UNIQUE Constraint prevents two records from having identical values in a column. In the CUSTOMERS table, for example, you might want to prevent two or more people from having an identical age.

4.Primary Key

A primary key is a field in a table which uniquely identifies each row/record in a database table. Primary keys must contain unique values. A primary key column cannot have NULL values. A table can have only one primary key, which may consist of single or multiple fields. When multiple fields are used as a primary key, they are called a **composite key**. If a table has a primary key defined on any field(s), then you cannot have two records having the same value of that field(s).

5.Foreign Key

A foreign key is a key used to link two tables together. This is sometimes also called as a referencing key. A Foreign Key is a column or a combination of columns whose values match a Primary Key in a different table.

The relationship between 2 tables matches the Primary Key in one of the tables with a Foreign Key in the second table.

If a table has a primary key defined on any field(s), then you cannot have two records having the same value of that field(s).

6.CHECK Constraint

The CHECK Constraint enables a condition to check the value being entered into a record. If the condition evaluates to false, the record violates the constraint and isn't entered the table.

7.INDEX Constraint

The INDEX is used to create and retrieve data from the database very quickly. An Index can be created by using a single or a group of columns in a table. When the index is created, it is assigned a **ROWID** for each row before it sorts out the data. Proper indexes are good for performance in large databases, but you need to be careful while creating an index. A selection of fields depends on what you are using in your SQL queries.

ALGORITHM

Step 1: Start

Step 2: Create a database.

Step 3: While creating the table, specify constraints along with the specific attribute.

Step 4: Insert some values to the table to check the working of constraints.

Step 5: Stop**.**

QUERIES

1. NOT NULL:

The following SQL query creates a new table called CUSTOMERS and adds five columns, three of which, are ID NAME and AGE, In this we specify not to accept NULLs.

```
CREATE TABLE CUSTOMERS(
ID INT NOT NULL,
NAME VARCHAR (20) NOT NULL,
AGE INT NOT NULL,
ADDRESS CHAR (25) ,
SALARY DECIMAL (18, 2),
PRIMARY KEY (ID)
);
```

If CUSTOMERS table has already been created, then to add a NOT NULL constraint to the SALARY column in Oracle and MySQL, we use the ALTER command.

```
ALTER TABLE CUSTOMERS
MODIFY SALARY DECIMAL (18, 2) NOT NULL;
```

2. DEFAULT:

For example, the following SQL creates a new table called CUSTOMERS and adds five columns. Here, the SALARY column is set to 5000.00 by default, so in case the INSERT INTO statement does not provide a value for this column, then by default this column would be set to 5000.00.

```
CREATE TABLE CUSTOMERS(
ID INT NOT NULL,
NAME VARCHAR (20) NOT NULL,
```

```
AGE INT NOT NULL,
ADDRESS CHAR (25) ,
SALARY DECIMAL (18, 2) DEFAULT 5000.00,
PRIMARY KEY (ID)
)
```

If the CUSTOMERS table has already been created, then to add a DEFAULT constraint to the SALARY column, we use the below syntax.

```
ALTER TABLE CUSTOMERS
MODIFY SALARY DECIMAL (18, 2) DEFAULT 5000.00;
```

To drop a DEFAULT constraint, use the following SQL query

```
ALTER TABLE CUSTOMERS
ALTER COLUMN SALARY DROP DEFAULT;
```

3. UNIQUE:

For example, the following SQL query creates a new table called CUSTOMERS and adds five columns. Here, the AGE column is set to UNIQUE, so that you cannot have two records with the same age.

```
CREATE TABLE CUSTOMERS(
ID INT NOT NULL,
NAME VARCHAR (20) NOT NULL,
AGE INT NOT NULL UNIQUE,
ADDRESS CHAR (25),
SALARY DECIMAL (18, 2),
PRIMARY KEY (ID)
);
```

If the CUSTOMERS table has already been created, then to add a UNIQUE constraint to the AGE column, then we use the below code.

```
ALTER TABLE CUSTOMERS
```

MODIFY AGE INT NOT NULL UNIQUE;

You can also use the following syntax, which supports naming the constraint in multiple columns as well.

```
ALTER TABLE CUSTOMERS
ADD CONSTRAINT myUniqueConstraint UNIQUE(AGE,SALARY);
```

To drop a UNIQUE constraint, use the following SQL query.

```
ALTER TABLE CUSTOMERS
DROP INDEX myUniqueConstraint;
```

4. Primary Key

Here is the syntax to define the ID attribute as a primary key in a CUSTOMERS table.

```
CREATE TABLE CUSTOMERS(
ID INT NOT NULL,
NAME VARCHAR (20) NOT NULL,
AGE INT NOT NULL,
ADDRESS CHAR (25) ,
SALARY DECIMAL (18, 2),
PRIMARY KEY (ID)
);
```

To create a PRIMARY KEY constraint on the "ID" column when the CUSTOMERS table already exists, use the following SQL syntax −

```
ALTER TABLE CUSTOMER ADD PRIMARY KEY (ID);
```

For defining a PRIMARY KEY constraint on multiple columns, use the SQL syntax given below.

```
CREATE TABLE CUSTOMERS(
```

```
ID   INT          NOT NULL,
NAME VARCHAR (20)    NOT NULL,
AGE  INT          NOT NULL,
ADDRESS  CHAR (25) ,
SALARY   DECIMAL (18, 2),
PRIMARY KEY (ID, NAME)
);
```

To create a PRIMARY KEY constraint on the "ID" and "NAMES" columns when CUSTOMERS table already exists, use the following SQL syntax.

```
ALTER TABLE CUSTOMERS
  ADD CONSTRAINT PK_CUSTID PRIMARY KEY (ID, NAME);
```

You can clear the primary key constraints from the table with the syntax given below.

```
ALTER TABLE CUSTOMERS DROP PRIMARY KEY;
```

5. Foreign Key

Consider the structure of the following two tables.

```
CREATE TABLE CUSTOMERS(
ID INT NOT NULL,
NAME VARCHAR (20) NOT NULL,
AGE INT NOT NULL,
ADDRESS CHAR (25) ,
SALARY DECIMAL (18, 2),
PRIMARY KEY (ID)
);

CREATE TABLE ORDERS (
ID INT NOT NULL,
```

```
DATE DATETIME,
CUSTOMER_ID INT references CUSTOMERS(ID),
AMOUNT double,
PRIMARY KEY (ID)
);
```

If the ORDERS table has already been created and the foreign key has not yet been set, the use the syntax for specifying a foreign key by altering a table.

```
ALTER TABLE ORDERS
ADD FOREIGN KEY (Customer_ID) REFERENCES CUSTOMERS (ID);
```

To drop a FOREIGN KEY constraint, use the following SQL syntax.

```
ALTER TABLE ORDERS
DROP FOREIGN KEY;
```

6. CHECK

For example, the following program creates a new table called CUSTOMERS and adds five columns. Here, we add a CHECK with AGE column, so that you cannot have any CUSTOMER who is below 18 years.

```
CREATE TABLE CUSTOMERS(
ID INT NOT NULL,
NAME VARCHAR (20) NOT NULL,
AGE INT NOT NULL CHECK (AGE >= 18),
ADDRESS CHAR (25) ,
SALARY DECIMAL (18, 2),
PRIMARY KEY (ID)
);
```

If the CUSTOMERS table has already been created, then to add a CHECK constraint to AGE column, you would write a statement like the one given below.

```
ALTER TABLE CUSTOMERS
MODIFY AGE INT NOT NULL CHECK (AGE >= 18 );
```

You can also use the following syntax, which supports naming the constraint in multiple columns as well −

```
ALTER TABLE CUSTOMERS
ADD CONSTRAINT myCheckConstraint CHECK(AGE >= 18);
```

To drop a CHECK constraint, use the following SQL syntax. This syntax does not work with MySQL.

```
ALTER TABLE CUSTOMERS
DROP CONSTRAINT myCheckConstraint;
```

7. INDEX Constraint

For example, the following SQL syntax creates a new table called CUSTOMERS and adds five columns in it.

```
CREATE TABLE CUSTOMERS(
ID INT NOT NULL,
NAME VARCHAR (20) NOT NULL,
AGE INT NOT NULL,
ADDRESS CHAR (25) ,
SALARY DECIMAL (18, 2),
PRIMARY KEY (ID)
);
```

Now, you can create an index on a single or multiple columns using the syntax given below.

CREATE INDEX index_name

ON table_name (column1, column2.....);

To create an INDEX on the AGE column, to optimize the search on customers for a specific age, you can use the follow SQL syntax which is given below −

CREATE INDEX idx_age

ON CUSTOMERS (AGE);

To drop an INDEX constraint, use the following SQL syntax.

ALTER TABLE CUSTOMERS

DROP INDEX idx_age;

EXPECTED OUTCOME

Students are able to understand the concept of constraints in detail.

VIVA QUESTIONS

1. What is the purpose of foreign key constraint?
2. How many types of constraints are present in SQL Server?
3. Can a column with Primary Key have Null value?
4. Which key accepts multiple NULL values?
5. Can column with Unique key have duplicate values?

Exp.No 4	Creation of views and assertions	Date……

AIM

To create a view and assertions.

THEORY

A view is nothing more than a SQL statement that is stored in the database with an associated name. A view is actually a composition of a table in the form of a predefined SQL query.

A view 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 a view.

Views, which are a type of virtual tables allow users to do the following:

- Structure data in a way that users or classes of users find natural or intuitive.
- Restrict access to the data in such a way that a user can see and (sometimes) modify exactly what they need and no more.
- Summarize data from various tables which can be used to generate reports.

Creating Views:

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 specific implementation.

The basic **CREATE VIEW** syntax is as follows:

CREATE VIEW view_name AS
SELECT column1, column2.....
FROM table_name
WHERE [condition];

You can include multiple tables in your SELECT statement in a similar way as you use them in a normal SQL SELECT query.

ALGORITHM

Step 1: Start.
Step 2: Create a database and tables with necessary attributes.
Step 3: Create a view with any attributes of above table.
Step 4: Insert values into the view.
Step 5: Retrieve values from the view using select operations.
Step 6: Stop.

QUERIES

Following is an example to create a view from the CUSTOMERS table. This view would be used to have customer name and age from the CUSTOMERS table.

CREATE VIEW CUSTOMERS_VIEW AS
SELECT name, age
FROM CUSTOMERS;

Now, you can query CUSTOMERS_VIEW in a similar way as you query an actual table.

Following is an example for the same.

SELECT * FROM CUSTOMERS_VIEW;

The following code block has an example to update the age of Ramesh.

```
UPDATE CUSTOMERS_VIEW
SET AGE = 35
WHERE name='Ramesh';
```

Rows of data can be deleted from a view. Following is an example to delete a record having AGE = 22. Obviously, where you have a view, you need a way to drop the view if it is no longer needed. The syntax is very simple and is given below:

```
DELETE FROM CUSTOMERS_VIEW
WHERE age = 22;
```

Following is an example to drop the CUSTOMERS_VIEW from the CUSTOMERS table.

```
DROP VIEW view_name;
```

Expected Outcome

Students are able to understand the concept of views and its operations.

VIVA QUESTIONS

1. What is the purpose of views in SQL?
2. Are views stored in Databases?
3. What is the alternate name of views?
4. SQL Server has mainly how many types of views?
5. What is the syntax for creating a view?

Exp.No 5	Implementation of build in functions in RDBMS	Date……

AIM

To implement various build in functions in RDBMS.

THEORY

1. String Functions.
2. Numeric Functions
3. Date Functions.
4. Conversion Functions.
5. Conditional Functions
6. Group Functions

ALGORITHM

Step1: start

Step 2: Create database and tables with necessary attributes.

Step 3: Using dual table we can calculate string and numeric functions.

Step 4: Do all functions for practice.

Step 5: Stop.

QUERIES WITH EXPLANATION:

STRING FUNCTIONS

1. ASCII(str)

Returns the numeric value of the leftmost character of the string str. Returns 0 if str is the empty string. Returns NULL if str is NULL. ASCII() works for characters with numeric values from 0 to 255.

SQL> SELECT ASCII('2');

SQL> SELECT ASCII('dx');

2. BIT_LENGTH(str)

Returns the length of the string str in bits.

SQL> SELECT BIT_LENGTH('text');

3. CHAR(N,... [USING charset_name])

CHAR() interprets each argument N as an integer and returns a string consisting of the characters given by the code values of those integers. NULL values are skipped.

SQL> SELECT CHAR(77,121,83,81,'76');

4. CHAR_LENGTH(str)

Returns the length of the string str measured in characters. A multi-byte character counts as a single character. This means that for a string containing five two-byte characters, LENGTH() returns 10, whereas CHAR_LENGTH() returns 5.

SQL> SELECT CHAR_LENGTH("text");

5. CONCAT(str1,str2,...)

Returns the string that results from concatenating the arguments. May have one or more arguments. If all arguments are non-binary strings, the result is a non-binary string. If the arguments include any binary strings, the result is a binary string. A numeric argument is converted to its equivalent binary string form; if you want to avoid that, you can use an explicit type cast, as in this example −

SQL> SELECT CONCAT('My', 'S', 'QL');

6. CONCAT_WS(separator,str1,str2,...)

CONCAT_WS() stands for Concatenate With Separator and is a special form of CONCAT(). The first argument is the separator for the rest of the arguments. The separator is added between the strings to be concatenated. The separator can be a string, as can the rest of the arguments. If the separator is NULL, the result is NULL.

SQL> SELECT CONCAT_WS(',','First name','Last Name');

7. FIND_IN_SET(str,strlist)

Returns a value in the range of 1 to N if the string str is in the string list strlist consisting of N substrings.

SQL> SELECT FIND_IN_SET('b','a,b,c,d');

8. HEX(N_or_S)

If N_or_S is a number, returns a string representation of the hexadecimal value of N, where N is a longlong (BIGINT) number. This is equivalent to CONV(N,10,16).

If N_or_S is a string, returns a hexadecimal string representation of N_or_S where each character in N_or_S is converted to two hexadecimal digits.

SQL> SELECT HEX(255);

SQL> SELECT 0x616263;

9. INSERT(str,pos,len,newstr)

Returns the string str, with the substring beginning at position pos and len characters long replaced by the string newstr. Returns the original string if pos is not within the length of the string. Replaces the rest of the string from

position pos if len is not within the length of the rest of the string. Returns NULL if any argument is NULL.

SQL> SELECT INSERT('Quadratic', 3, 4, 'What');

10. INSTR(str,substr)

Returns the position of the first occurrence of substring substr in string str. This is the same as the two-argument form of LOCATE(), except that the order of the arguments is reversed.

SQL> SELECT INSTR('foobarbar', 'bar');

11. LEFT(str,len)

Returns the leftmost len characters from the string str, or NULL if any argument is NULL.

SQL> SELECT LEFT('foobarbar', 5);

12. LENGTH(str)

Returns the length of the string str, measured in bytes. A multi-byte character counts as multiple bytes. This means that for a string containing five two-byte characters, LENGTH() returns 10, whereas CHAR_LENGTH() returns 5.

SQL> SELECT LENGTH('text');

13. LOWER(str)

Returns the string str with all characters changed to lowercase according to the current character set mapping.

SQL> SELECT LOWER('QUADRATICALLY');

14. UPPER(str)

Returns the string str with all characters changed to uppercase according to the current character set mapping.

SQL> SELECT UPPER('Allah-hus-samad');

15. STRCMP(str1, str2)

Compares two strings and returns 0 if both strings are equal, it returns -1 if the first argument is smaller than the second according to the current sort order otherwise it returns 1.

SQL> SELECT STRCMP('MOHD', 'MOHD');

NUMERIC FUNCTIONS

1. ABS(X)

The ABS() function returns the absolute value of X. Consider the following example −

SQL> SELECT ABS(2);

2. GREATEST(n1,n2,n3,.....)

The GREATEST() function returns the greatest value in the set of input parameters (n1, n2, n3, a nd so on). The following example uses the GREATEST() function to return the largest number from a set of numeric values −

SQL>SELECT GREATEST(3,5,1,8,33,99,34,55,67,43);

3. LEAST(N1,N2,N3,N4,.....)

The LEAST() function is the opposite of the GREATEST() function. Its purpose is to return the least-valued item from the value list (N1, N2, N3, and

so on). The following example shows the proper usage and output for the LEAST() function −

SQL>SELECT LEAST(3,5,1,8,33,99,34,55,67,43);

4. MOD(N,M)

This function returns the remainder of N divided by M. Consider the following example −

SQL>SELECT MOD(29,3);

5. PI()

This function simply returns the value of pi. SQL internally stores the full double-precision value of pi.

SQL>SELECT PI();

6. POWER(X,Y)

These two functions return the value of X raised to the power of Y.

SQL> SELECT POWER(3,3);

7. ROUND(X), ROUND(X,D)

This function returns X rounded to the nearest integer. If a second argument, D, is supplied, then the function returns X rounded to D decimal places. D must be positive or all digits to the right of the decimal point will be removed. Consider the following example −

SQL>SELECT ROUND(5.693893);

SQL>SELECT ROUND(5.693893,2);

8. SQRT(X)

This function returns the non-negative square root of X. Consider the following example −

SQL>SELECT SQRT(49);

9. TRUNCATE(X,D)

This function is used to return the value of X truncated to D number of decimal places. If D is 0, then the decimal point is removed. If D is negative, then D number of values in the integer part of the value is truncated. Consider the following example −

SQL>SELECT TRUNCATE(7.536432,2);

DATE FUNCTIONS

1. ADDDATE(date,INTERVAL expr unit), ADDDATE(expr,days)

When invoked with the INTERVAL form of the second argument, ADDDATE() is a synonym for DATE_ADD(). The related function SUBDATE() is a synonym for DATE_SUB(). For information on the INTERVAL unit argument, see the discussion for DATE_ADD().

mysql> SELECT DATE_ADD('1998-01-02', INTERVAL 31 DAY);

mysql> SELECT ADDDATE('1998-01-02', INTERVAL 31 DAY);

mysql> SELECT ADDDATE('1998-01-02', 31);

2. ADDTIME(expr1,expr2)

ADDTIME() adds expr2 to expr1 and returns the result. The expr1 is a time or datetime expression, while the expr2 is a time expression.

mysql> SELECT ADDTIME('1997-12-31 23:59:59.999999','1 1:1:1.000002');

3. CURDATE()

Returns the current date as a value in 'YYYY-MM-DD' or YYYYMMDD format, depending on whether the function is used in a string or in a numeric context.

mysql> SELECT CURDATE();

4. CURTIME()

Returns the current time as a value in 'HH:MM:SS' or HHMMSS format, depending on whether the function is used in a string or in a numeric context. The value is expressed in the current time zone.

mysql> SELECT CURTIME();

5. DATEDIFF(expr1,expr2)

DATEDIFF() returns **expr1 . expr2** expressed as a value in days from one date to the other. Both expr1 and expr2 are date or date-and-time expressions. Only the date parts of the values are used in the calculation.

mysql> SELECT DATEDIFF('1997-12-31 23:59:59','1997-12-30');

6. DAYNAME(date)

Returns the name of the weekday for date.

mysql> SELECT DAYNAME('1998-02-05');

7. DAYOFMONTH(date)

Returns the day of the month for date, in the range 0 to 31.

mysql> SELECT DAYOFMONTH('1998-02-03');

8. DAYOFWEEK(date)

Returns the weekday index for date (1 = Sunday, 2 = Monday, ., 7 = Saturday). These index values correspond to the ODBC standard.

mysql> SELECT DAYOFWEEK('1998-02-03');

9. DAYOFYEAR(date)

Returns the day of the year for date, in the range 1 to 366.

mysql> SELECT DAYOFYEAR('1998-02-03');

10. HOUR(time)

Returns the hour for time. The range of the return value is 0 to 23 for time-of-day values. However, the range of TIME values actually is much larger, so HOUR can return values greater than 23.

mysql> SELECT HOUR('10:05:03');

11. LAST_DAY(date)

Takes a date or datetime value and returns the corresponding value for the last day of the month. Returns NULL if the argument is invalid.

mysql> SELECT LAST_DAY('2003-02-05');

12. MAKEDATE(year,dayofyear)

Returns a date, given year and day-of-year values. The **dayofyear** value must be greater than 0 or the result will be NULL.

mysql> SELECT MAKEDATE(2001,31), MAKEDATE(2001,32);

13. MAKETIME(hour,minute,second)

Returns a time value calculated from the hour, minute and second arguments.

mysql> SELECT MAKETIME(12,15,30);

14. MINUTE(time)

Returns the minute for time, in the range 0 to 59.

mysql> SELECT MINUTE('98-02-03 10:05:03');

15. MONTH(date)

Returns the month for date, in the range 0 to 12.

mysql> SELECT MONTH('1998-02-03')

16. MONTHNAME(date)

Returns the full name of the month for a date.

mysql> SELECT MONTHNAME('1998-02-05');

17. NOW()

Returns the current date and time as a value in 'YYYY-MM-DD HH:MM:SS' or

YYYYMMDDHHMMSS format, depending on whether the function is used in a string or numeric context. This value is expressed in the current time zone.

mysql> SELECT NOW();

18. SYSDATE()

Returns the current date and time as a value in 'YYYY-MM-DD HH:MM:SS' or

YYYYMMDDHHMMSS format, depending on whether the function is used in a string or in a numeric context.

mysql> SELECT SYSDATE();

CONVERSION FUNCTIONS:

1. TO_CHAR function

TO_CHAR function is used to typecast a numeric or date input to character type with a format model (optional).

Syntax

TO_CHAR(number1, [format], [nls_parameter])

```
SELECT first_name,
    TO_CHAR (hire_date, 'MONTH DD, YYYY') HIRE_DATE,
        TO_CHAR (salary, '$99999.99') Salary
FROM employees
WHERE rownum < 5;
```

2. TO_NUMBER function

The TO_NUMBER function converts a character value to a numeric datatype. If the string being converted contains nonnumeric characters, the function returns an error.

Syntax

TO_NUMBER (string1, [format], [nls_parameter])

```
SELECT  TO_NUMBER('121.23', '9G999D99')  FROM DUAL
```

3. TO_DATE function

The function takes character values as input and returns formatted date equivalent of the same. The TO_DATE function allows users to enter a date in any format, and then it converts the entry into the default format.

Syntax:

TO_DATE(string1, [format_mask], [nls_language])

SELECT TO_DATE('January 15, 1989, 11:00 A.M.', 'Month dd, YYYY, HH:MI A.M.', 'NLS_DATE_LANGUAGE = American') FROM DUAL;

CONDITIONAL FUNCTIONS:

1. CASE expression

CASE expressions works on the same concept as DECODE but differs in syntax and usage.

SYNTAX:

```
CASE  [ expression ]
  WHEN condition_1 THEN result_1
  WHEN condition_2 THEN result_2
  ...
  WHEN condition_n THEN result_n
  ELSE result
END
```

```
SELECT first_name, CASEWHEN salary < 200 THEN 'GRADE 1'
                        WHEN salary > 200 AND salary < 5000 THEN 'GRADE 2'
                        ELSE 'GRADE 3'
                   END CASE
```

FROM employees;

2. The DECODE function

The function is the SQL equivalence of IF..THEN..ELSE conditional procedural statement. DECODE works with values/columns/expressions of all data types.

SYNTAX:

DECODE (expression, search, result [, search, result]... [, default])

```
SELECT DECODE(NULL,NULL,'EQUAL','NOT EQUAL')
FROM DUAL;
SELECT first_name, salary, DECODE (hire_date, sysdate,'NEW JOINEE','EMPLOYEE')
    FROM employees;
```

EXPECTED OUTCOME

Students are able to study the various build in functions in SQL.

VIVA QUESTIONS

1. What are the uses of Bulid in functions in SQL?
2. What are the types of build in functions in SQL?
3. Explain the String comparison function?
4. Explain the necessary of DATE function in SQL?
5. What is the build in function for find the square root of a number?
6. What is the build in function for change the string into initial capital format?

Exp.No 6	Implementation of various aggregate functions in SQL	Date……

AIM

To implement various aggregate functions in SQL.

THEORY

An aggregate function allows you to perform calculation on a set of values to return a single scalar value. We often use aggregate functions with the GROUP BY and HAVING clauses of the SELECT statement.

1. COUNT

MySQL COUNT function is the simplest function and very useful in counting the number of records, which are expected to be returned by a SELECT statement**.**

2. MAX

MySQL **MAX** function is used to find out the record with maximum value among a record set.

3. MIN

MySQL **MIN** function is used to find out the record with minimum value among a record set.

4. AVG

MySQL **AVG** function is used to find out the average of a field in various records.

5. SUM

MySQL **SUM** function is used to find out the sum of a field in various records.

THEORY

Step 1: Start.

Step 2: Create database and tables.

Step 3: Use aggregate functions, find the MIN, MAX, AVG, COUNT and SUM of attributes.

Step 4: Stop.

QUERIES:

1.COUNT

To understand COUNT function, consider an employee_tbl table, which is having the following records −

mysql> SELECT * FROM employee_tbl;

```
+------+------+------------+-------------------+
| id  | name | work_date  | daily_typing_pages |
+------+------+------------+-------------------+
|  1  | John | 2007-01-24 |       250         |
|  2  | Ram  | 2007-05-27 |       220         |
|  3  | Jack | 2007-05-06 |       170         |
|  3  | Jack | 2007-04-06 |       100         |
|  4  | Jill | 2007-04-06 |       220         |
|  5  | Zara | 2007-06-06 |       300         |
|  5  | Zara | 2007-02-06 |       350         |
+------+------+------------+-------------------+
```

Now, suppose based on the above table you want to count total number of rows in this table, then you can do it as follows −

mysql>SELECT COUNT(*) FROM employee_tbl ;

```
+----------+
| COUNT(*) |
+----------+
```

```
|   7    |
+----------+
```

Similarly, if you want to count the number of records for Zara, then it can be done as follows −

mysql>SELECT COUNT(*) FROM employee_tbl WHERE name = "Zara";

```
+----------+
| COUNT(*) |
+----------+
|   2    |
+----------+
```

2. MAX

Now, suppose based on the above table you want to fetch maximum value of daily_typing_pages, then you can do so simply using the following command –

mysql> SELECT MAX(daily_typing_pages) FROM employee_tbl;

```
+-----------------------+
| MAX(daily_typing_pages) |
+-----------------------+
|          350          |
+-----------------------+
```

You can find all the records with maximum value for each name using GROUP BY clause as follows −

mysql> SELECT id, name, MAX(daily_typing_pages) FROM employee_tbl GROUP BY name;

```
+------+------+-----------------------+
|  id  | name | MAX(daily_typing_pages) |
```

```
+------+------+-----------------------+
|  3  | Jack |         170          |
|  4  | Jill |         220          |
|  1  | John |         250          |
|  2  | Ram  |         220          |
|  5  | Zara |         350          |
+------+------+-----------------------+
```

You can use MIN Function along with MAX function to find out minimum value as well. Try out the following example −

mysql> SELECT MIN(daily_typing_pages) least, MAX(daily_typing_pages) max FROM employee_tbl;

```
+-------+------+
| least |  max |
+-------+------+
|  100  |  350 |
+-------+------+
```

3. MIN

Now, suppose based on the above table you want to fetch minimum value of daily_typing_pages, then you can do so simply using the following command –

mysql> SELECT MIN(daily_typing_pages) FROM employee_tbl;

```
+-----------------------+
| MIN(daily_typing_pages) |
+-----------------------+
|         100          |
+-----------------------+
```

You can find all the records with minimum value for each name using GROUP BY clause as follows −

mysql>SELECT id, name, MIN(daily_typing_pages) FROM employee_tbl GROUP BY name;

```
+------+------+-----------------------+
| id  | name | MIN(daily_typing_pages) |
+------+------+-----------------------+
|  3  | Jack |          100          |
|  4  | Jill |          220          |
|  1  | John |          250          |
|  2  | Ram  |          220          |
|  5  | Zara |          300          |
+------+------+-----------------------+
```

You can use MIN Function along with MAX function to find out minimum value as well. Try out the following example −

mysql> SELECT MIN(daily_typing_pages) least, MAX(daily_typing_pages) max FROM employee_tbl;

```
+-------+------+
| least |  max |
+-------+------+
|  100  | 350  |
+-------+------+
```

4. AVG

Now, suppose based on the above table you want to calculate average of all the dialy_typing_pages, then you can do so by using the following command –

mysql> SELECT AVG(daily_typing_pages) FROM employee_tbl;

```
+------------------------+
| AVG(daily_typing_pages) |
+------------------------+
|        230.0000        |
+------------------------+
```

You can take average of various records set using GROUP BY clause. Following example will take average all the records related to a single person and you will have average typed pages by every person.

mysql> SELECT name, AVG(daily_typing_pages) FROM employee_tbl GROUP BY name;

```
+------+------------------------+
| name | AVG(daily_typing_pages) |
+------+------------------------+
| Jack |        135.0000        |
| Jill |        220.0000        |
| John |        250.0000        |
| Ram  |        220.0000        |
| Zara |        325.0000        |
+------+------------------------+
```

5. SUM

Now, suppose based on the above table you want to calculate total of all the dialy_typing_pages, then you can do so by using the following command −

mysql> SELECT SUM(daily_typing_pages) FROM employee_tbl;

```
+------------------------+
| SUM(daily_typing_pages) |
+------------------------+
```

```
|        1610          |
+------------------------+
```

You can take sum of various records set using GROUP BY clause. Following example will sum up all the records related to a single person and you will have total typed pages by every person.

mysql> SELECT name, SUM(daily_typing_pages) FROM employee_tbl GROUP BY name;

```
+------+------------------------+
| name | SUM(daily_typing_pages) |
+------+------------------------+
| Jack |          270           |
| Jill |          220           |
| John |          250           |
| Ram  |          220           |
| Zara |          650           |
+------+------------------------+
```

EXPECTED OUTCOME

Students are able to understand the concept of aggregate functions.

VIVA QUESTIONS

1. What is the use of aggregate functions?
2. What are the types of aggregate functions?
3. What is the syntax for each aggregate function?

Exp.No 7	Implementation of order by, group by& having clause.	Date……

AIM

To implement Order By, Group By and Having Clauses.

THEORY

1. ORDER BY

The SQL **ORDER BY** clause is used to sort the data in ascending or descending order, based on one or more columns. Some databases sort the query results in an ascending order by default.

Syntax

The basic syntax of the ORDER BY clause is as follows:

SELECT column-list

FROM table_name

[WHERE condition]

[ORDER BY column1, column2, .. columnN] [ASC | DESC];

You can use more than one column in the ORDER BY clause. Make sure whatever column you are using to sort that column should be in the column-list.

2. GROUP BY

The SQL **GROUP BY** clause is used in collaboration with the SELECT statement to arrange identical data into groups. This GROUP BY clause follows the WHERE clause in a SELECT statement and precedes the ORDER BY clause.

Syntax:

The basic syntax of a GROUP BY clause is shown in the following code block. The GROUP BY clause must follow the conditions in the WHERE clause and must precede the ORDER BY clause if one is used.

```
SELECT column1, column2
FROM table_name
WHERE [ conditions ]
GROUP BY column1, column2
ORDER BY column1, column2
```

3. HAVING CLAUSE

The **HAVING Clause** enables you to specify conditions that filter which group results appear in the results. The WHERE clause places conditions on the selected columns, whereas the HAVING clause places conditions on groups created by the GROUP BY clause.

Syntax

The following code block shows the position of the HAVING Clause in a query.

```
SELECT
FROM
WHERE
GROUP BY
HAVING
ORDER BY
```

The HAVING clause must follow the GROUP BY clause in a query and must also precede the ORDER BY clause if used. The following code block has the syntax of the SELECT statement including the HAVING clause:

SELECT column1, column2

FROM table1, table2

WHERE [conditions]

GROUP BY column1, column2

HAVING [conditions]

ORDER BY column1, column2

ALGORITHM

Step 1: Start

Step 2: Create database and tables with necessary attributes.

Step 3: Using Order By clause, make data's in ascending or descending order.

Step 4: Using Group By Clause, arrange data's in any particular group.

Step 5: Using Having Clause, retrieve data's with some conditions.

Step 6: Stop.

QUERIES

1. ORDER BY

The following code block has an example, which would sort the result in an ascending order by the NAME and the SALARY.

```
SQL> SELECT * FROM CUSTOMERS
ORDER BY NAME, SALARY;
```

The following code block has an example, which would sort the result in the descending order by NAME.

```
SQL> SELECT * FROM CUSTOMERS
ORDER BY NAME DESC;
```

2. GROUP BY

If you want to know the total amount of the salary on each customer, then the GROUP BY query would be as follows.

```
SQL> SELECT NAME, SUM(SALARY) FROM CUSTOMERS
GROUP BY NAME;
```

3. HAVING CLAUSE

Following is an example, which would display a record for a similar age count that would be more than or equal to 2.

```
SQL > SELECT ID, NAME, AGE, ADDRESS, SALARY
FROM CUSTOMERS
GROUP BY age
HAVING COUNT(age) >= 2;
```

EXPECTED OUTCOME

Students are able to understand the concept of Order By, Group By and Having Clauses.

VIVA QUESTIONS

1. What is the meaning of "GROUP BY" clause in Mysql?
2. Is "GROUP BY" clause is similar to "ORDER BY" clause? Justify your answer.
3. Is Null values in GROUP BY fields are omitted?

Exp.No 8	Implementation of set operators, nested queries.	Date……

AIM

To implement set operators.

THEORY

Set operators are used to join the results of two (or more) SELECT statements. The SET operators are UNION, UNION ALL, INTERSECT, and MINUS.

The UNION set operator returns the combined results of the two SELECT statements. Essentially, it removes duplicates from the results i.e. only one row will be listed for each duplicated result. To counter this behavior, use the UNION ALL set operator which retains the duplicates in the final result. INTERSECT lists only records that are common to both the SELECT queries; the MINUS set operator removes the second query's results from the output if they are also found in the first query's results. INTERSECT and MINUS set operations produce unduplicated results.

1. UNION

When multiple SELECT queries are joined using UNION operator, Oracle displays the combined result from all the compounded SELECT queries, after removing all duplicates and in sorted order (ascending by default), without ignoring the NULL values.

UNION ALL

UNION and UNION ALL are similar in their functioning with a slight difference. But UNION ALL gives the result set without removing

duplication and sorting the data. For example,in above query UNION is replaced by UNION ALL to see the effect.

Consider the query demonstrated in UNION section. Note the difference in the output which is generated without sorting and deduplication.

2. INTERSECT

Using INTERSECT operator, Oracle displays the common rows from both the SELECT statements, with no duplicates and data arranged in sorted order (ascending by default).

3. MINUS

Minus operator displays the rows which are present in the first query but absent in the second query, with no duplicates and data arranged in ascending order by default.

ALGORITHM

Step 1: Start

Step 2: Create database and tables.

Step 3: Using UNION, INTERSECT and MINUS, find the relation between tables.

Step 4: Stop.

QUERIES

1. UNION

Consider the below five queries joined using UNION operator. The final combined result set contains value from all the SQLs. Note the duplication removal and sorting of data.

```
SELECT 1 NUM FROM DUAL
UNION
SELECT 5 FROM DUAL
UNION
SELECT 3 FROM DUAL
UNION
SELECT 6 FROM DUAL
UNION
SELECT 3 FROM DUAL;

SELECT employee_id, first_name, salary
FROM employees
WHERE department_id=10
UNION
SELECT employee_id, first_name, salary
FROM employees
WHERE department_id=20
ORDER BY 3;
```

2. UNION ALL

Consider the below five queries joined using UNION ALL operator. The final combined result set contains value from all the SQLs. Note the duplication removal and sorting of data

```
SELECT 1 NUM FROM DUAL
UNION ALL
SELECT 5 FROM DUAL
UNION ALL
SELECT 3 FROM DUAL
UNION ALL
```

```
SELECT 6 FROM DUAL
UNION ALL
SELECT 3 FROM DUAL;
```

3. INTERSECT

For example, the below SELECT query retrieves the salary which are common in department 10 and 20.

```
SELECT SALARY
FROM employees
WHERE DEPARTMENT_ID = 10
INTRESECT
SELECT SALARY
FROM employees
WHERE DEPARTMENT_ID = 20
```

4. MINUS

```
SELECT JOB_ID
FROM employees
WHERE DEPARTMENT_ID = 10
MINUS
SELECT JOB_ID
FROM employees
WHERE DEPARTMENT_ID = 20;
```

AIM

To implement nested queries.

THEORY

A Subquery or Inner query or a Nested query is a query within another SQL query and embedded within the WHERE clause. A subquery is used to return data that will be used in the main query as a condition to further restrict the data to be retrieved. Subqueries can be used with the SELECT, INSERT, UPDATE, and DELETE statements along with the operators like =, <, >, >=, <=, IN, BETWEEN, etc.

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])
```

Subqueries with the UPDATE Statement

The subquery can be used in conjunction with the UPDATE statement. Either single or multiple columns in a table can be updated when using a subquery with the UPDATE statement. The basic syntax is as follows.

```
UPDATE table
SET column_name = new_value
[ WHERE OPERATOR [ VALUE ]
  (SELECT COLUMN_NAME
  FROM TABLE_NAME)
  [ WHERE) ]
```

Subqueries with the DELETE Statement

The subquery can be used in conjunction with the DELETE statement like with any other statements mentioned above. The basic syntax is as follows.

```
DELETE FROM TABLE_NAME
[ WHERE OPERATOR [ VALUE ]
  (SELECT COLUMN_NAME
  FROM TABLE_NAME)
  [ WHERE) ]
```

Subqueries with the INSERT Statement

Subqueries also can be used with INSERT statements. The INSERT statement uses the data returned from the subquery to insert into another table. The selected data in the subquery can be modified with any of the character, date or number functions.

```
INSERT INTO table_name [ (column1 [, column2 ]) ]
SELECT [ *|column1 [, column2 ]
FROM table1 [, table2 ]
[ WHERE VALUE OPERATOR ]
```

ALGORITHM

Step 1: Start

Step 2: Create database and tables.

Step 3: create subqueries using the syntax and check the insert, update and delete commands.

Step 4: Stop.

QUERY

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 |
+----+----------+-----+-----------+----------+
```

Now, let us check the following subquery with a SELECT statement.

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

Subqueries with the UPDATE Statement:

Assuming, we have CUSTOMERS_BKP table available which is backup of CUSTOMERS table. The following example updates SALARY by 0.25 times in the CUSTOMERS table for all the customers whose AGE is greater than or equal to 27.

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

Subqueries with the DELETE Statement:

Assuming, we have a CUSTOMERS_BKP table available which is a backup of the CUSTOMERS table. The following example deletes the records from the CUSTOMERS table for all the customers whose AGE is greater than or equal to 27.

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

Subqueries with the INSERT Statement

Consider a table CUSTOMERS_BKP with similar structure as CUSTOMERS table. Now to copy the complete CUSTOMERS table into the CUSTOMERS_BKP table, you can use the following syntax.

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

EXPECTED OUTCOME

Students are able to understand the concept of nested queries.

VIVA QUESTIONS

1. Explain Nested Queries?
2. What is the syntax of nested query?

Exp.No 9	Implementation of various control structures using PL/SQL	**Date……**

Q1 : Write PL/SQL block which will calculate some of two numbers and display the output?
Q2: Write a PL/SQL block which accepts employee number and increment is salary by 1000?
Q3: Write a PL/SQL block which empno and delete that row from the emp table?
Q4: PL/SQL for reversing the given stringQ3: Write a PL/SQL block which empno and delete that row from the emp table?

Theory
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 and loops.
If-then-
else Case
o Case with no else
o Labeled case
o Searched
case Simple loop
While loop
For loop
Goto and Labels

Conditional control in PL/SQL-
Syntax:
IF <condition> THEN
<Action>
ELSEIF<condition>
<Action>
ELSE
<Action>
ENDIF;

The WHILE LOOP:
Syntax:
WHILE <condition>

LOOP
<Action>
END LOOP;

The FOR LOOP statement:
Syntax:
FOR variable IN [REVERSE] start—end
LOOP
<Action>
END LOOP;

The GOTO statement: The goto statement allows you to change the flow of control
within a PL/SQL Block.

Queries :
Q1 : Write PL/SQL block which will calculate some of two numbers and display the output?
DECLARE

A number(2);
B number(2);
C number(3);
BEGIN
A := 10;
B := 20;
C := A + B;
DBMS_OUTPUT.PUT_LINE(C);
DBMS_OUTPUT.PUT_LINE('sum of two numbers' || C);
END;
/
Output:
30
sum of two numbers 30
PL/SQL procedure successfully completed.
Q2: Write a PL/SQL block which accepts employee number and increment is salary by 1000?
DECLARE
A number(4);
A := &Empno;

```
Update emp set sal = sal + 1000 where Empno = A;
END;
/
```

Q3: Write a PL/SQL block which empno and delete that row from the emp table?

```
DECLARE
A number(4);
BEGIN
A := &Empno;
Delete from emp where Empno = A;
END;
/
```

Q4: PL/SQL for reversing the given string

Algorithm:

1. Get the input string.
2. Find the length of the string.
3. Extract the characters one by one from the end of the string.
4. Concatenate the extracted characters.
5. Display the concatenated reversed string.
6. Stop the program.

Program:

```
declare
b varchar2(10) := '&b';
c varchar2(10);
l number(2);
i number(2);
g number(2);
d varchar2(10);
begin
l:=length(b);
g:=l;
for i in 1..l

loop
c:=substr(b,g,1);
g := g - 1;
d := d ||
c; end loop;
dbms_output.put_line('revised string
is'); dbms_output.put_line(d);
```

```
end;
```

OUTPUT:

```
Enter value for b: ramu
old 2: b varchar2(10) := '&b';
new 2: b varchar2(10) := 'ramu';
revised string is
umar
PL/SQL procedure successfully completed.
```

Expected Output

studied and implemented of various control structures using PL/SQL

Exp.No 10	CREATION OF PROCEDURES AND FUNCTIONS	Date……

Aim :Creation of Procedures and Functions

Q1 : Create a procedure to insert record
Q2 : Create a function to find factorial

Theory

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

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

At the schema level, subprogram is a **standalone subprogram**. It is created with the CREATE PROCEDURE or the CREATE FUNCTION statement. It is stored in the database and can be deleted with the DROP PROCEDURE or DROP FUNCTION statement.
A subprogram created inside a package is a **packaged subprogram**. It is stored in the database and can be deleted only when the package is deleted with the DROP PACKAGE statement. We will discuss packages in the chapter **'PL/SQL - Packages'**.
PL/SQL subprograms are named PL/SQL blocks that can be invoked with a set of parameters. PL/SQL provides two kinds of subprograms −

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

BEGIN
< procedure_body >
END procedure_name;
Where,

•*procedure-name* specifies the name of the procedure.

•[OR REPLACE] option allows the modification of an existing procedure.

•The optional parameter list contains name, mode and types of the parameters. **IN** represents the value that will be passed from outside and OUT represents the parameter that will be used to return a value outside of the procedure.

•*procedure-body* contains the executable part.

•The AS keyword is used instead of the IS keyword for creating a standalone procedure.

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

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

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

```
Procedure created.
```

Executing a Standalone Procedure
A standalone procedure can be called in two ways −

•Using the **EXECUTE** keyword

•Calling the name of the procedure from a PL/SQL block

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

```
EXECUTE greetings;
```

The above call will display −

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

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

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

The above call will display −

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

Deleting a Standalone Procedure

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

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

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

```
DROP PROCEDURE greetings;
```

Parameter Modes in PL/SQL Subprograms

IN & OUT Mode Example 1

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

```
DECLARE
a number;
b number;
c number;
PROCEDURE findMin(x IN number, y IN number, z OUT number) IS
BEGIN
IF x < y THEN
z:= x;
ELSE
z:= y;
END IF;
END;
BEGIN
```

```
a:= 23;
b:= 45;
findMin(a, b, c);
dbms_output.put_line(' Minimum of (23, 45) : ' || c);
END;
/
```

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

```
Minimum of (23, 45) : 23
PL/SQL procedure successfully completed.
```

IN & OUT Mode Example 2

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

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

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

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

Methods for Passing Parameters

Actual parameters can be passed in three ways −

- Positional notation
- Named notation
- Mixed notation

PROCEDURE TO INSERT NUMBER

```
SQL> create table emp1(id number(3),First_name varchar2(20));
Table created.
SQL> insert into emp1 values(101,'Nithya');
1 row created.
SQL> insert into emp1 values(102,'Maya');
1 row created.
SQL> select * from emp1;
ID FIRST_NAME
101 Nithya
102 Maya
SQL> set serveroutput on;
SQL> create or replace
2 procedure insert_num(p_num number)is
3 begin
4 insert into emp1(id,First_name) values(p_num,user);
5 end insert_num;
6 /
Procedure created.
SQL> exec insert_num(3);
PL/SQL procedure successfully completed.
SQL> select * from emp1;
ID FIRST_NAME
--------- --------------------
101 Nithya
102 Maya
103 SCOTT
```

FUNCTION TO FIND FACTORIAL

```
SQL> create or replace function fact(n number)
2 return number is
3 i number(10);
4 f number:=1;
5 begin
6 for i in 1..N loop
7 f:=f*i;
8 end loop;
```

```
9 return f;
10 end;
11 /
Function created.
SQL> select fact(5) from dual;
FACT(5)
---------
120
```

Creating a Function

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

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

Where,

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

•The AS keyword is used instead of the IS keyword for creating a standalone function.

Example

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

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

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

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

```
Function created.
```

Calling a Function

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

A called function performs the defined task and when its return statement is executed or when the **last end statement** is reached, it returns the program control back to the main program.
To call a function, you simply need to pass the required parameters along with the function name and if the function returns a value, then you can store the returned value. Following program calls the function **totalCustomers** from an anonymous block −

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

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

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

Expected Output
Created procedures and functions specified above.

Exp.No 11	CREATION OF PACKAGES	Date……

Aim :

Creation of Packages

Theory

The package might include a set of procedures that forms an API, or a pool of type definitions and variable declarations. The package is compiled and stored in the database, where its contents can be shared by many applications.
A package is a schema object that groups logically related PL/SQL types, variables, and subprograms. Packages usually have two parts, a specification (spec) and a body; sometimes the body is unnecessary. The specification is the interface to the package. It declares the types, variables, constants, exceptions, cursors, and subprograms that can be referenced from outside the package. The body defines the queries for the cursors and the code for the subprograms.

To create package specs, use the SQL statement CREATE PACKAGE. A CREATE PACKAGE BODY statement defines the package body.
The body holds implementation details and private declarations, which are hidden from code outside the package.
Following the declarative part of the package body is the optional initialization part, which holds statements that initialize package variables and do any other one-time setup steps.

The AUTHID clause determines whether all the packaged subprograms execute with the privileges of their definer (the default) or invoker, and whether their unqualified references to schema objects are resolved in the schema of the definer or invoker.
A call spec lets you map a package subprogram to a Java method or external C function. The call spec maps the Java or C name, parameter types, and return type to their SQL counterparts.
The following is contained in a PL/SQL package:

•Get and Set methods for the package variables, if you want to avoid letting other procedures read and write them directly.

•Cursor declarations with the text of SQL queries. Reusing exactly the same query text in multiple locations is faster than retyping the same query each time with slight differences. It is also easier to maintain if you need to change a query that is used in many places.

•Declarations for exceptions. Typically, you need to be able to reference these from different procedures, so that you can handle exceptions within called subprograms.

•Declarations for procedures and functions that call each other. You do not need to worry about compilation order for packaged procedures and functions, making them more convenient than standalone stored procedures and functions when they call back and forth to each other.

•Declarations for overloaded procedures and functions. You can create multiple variations of a procedure or function, using the same names but different sets of parameters.

•Variables that you want to remain available between procedure calls in the same session. You can treat variables in a package like global variables.

•Type declarations for PL/SQL collection types. To pass a collection as a parameter between stored procedures or functions, you must declare the type in a package so that both the calling and called subprogram can refer to it.

The spec lists the package resources available to applications. All the information your application needs to use the resources is in the spec. For example, the following declaration shows that the function named factorial takes one argument of type INTEGER and returns a value of type INTEGER:
FUNCTION factorial (n INTEGER) RETURN INTEGER; -- returns n!
That is all the information you need to call the function. You need not consider its underlying implementation (whether it is iterative or recursive for example).

If a spec declares only types, constants, variables, exceptions, and call specs, the package body is unnecessary. Only subprograms and cursors have an

underlying implementation. In Example, the package needs no body because it declares types, exceptions, and variables, but no subprograms or cursors. Such packages let you define global variables, usable by stored procedures and functions and triggers, that persist throughout a session.

CREATE PACKAGE trans_data AS -- bodiless package TYPE TimeRec IS RECORD (minutes SMALLINT, hours SMALLINT); TYPE TransRec IS RECORD (category VARCHAR2(10), account INT, amount REAL, time_of TimeRec); minimum_balance CONSTANT REAL := 10.00; number_processed INT; insufficient_funds EXCEPTION; END trans_data; /

Referencing Package Contents

To reference the types, items, subprograms, and call specs declared within a package spec, use dot notation:
package_name.type_name
package_name.item_name
package_name.subprogram_name

You can reference package contents from database triggers, stored subprograms, 3GL application programs, and various Oracle tools. For example, you can call package procedures as shown in Example 9-3.
The following example calls the hire_employee procedure from an anonymous block in a Pro*C program. The actual parameters emp_id, emp_lname, and emp_fname are host variables.
EXEC SQL EXECUTE
BEGIN
emp_actions.hire_employee(:emp_id,:emp_lname,:emp_fname, ...);

Restrictions
You cannot reference remote packaged variables, either directly or indirectly. For example, you cannot call the a procedure through a database link if the procedure refers to a packaged variable.
Inside a package, you cannot reference host variables.

Creating the emp_admin Package
Using -- create the audit table to track changes
CREATE TABLE emp_audit(date_of_action DATE,user_id VARCHAR2(20), package_name VARCHAR2(30));

CREATE OR REPLACE PACKAGE emp_admin AS TYPE EmpRecTyp IS RECORD (emp_id NUMBER, sal NUMBER);

-- Declare externally visible types, cursor, exception

CURSOR desc_salary RETURN EmpRecTyp;
invalid_salary EXCEPTION;

-- Declare externally callable subprograms

FUNCTION hire_employee (last_name VARCHAR2, first_name VARCHAR2, email VARCHAR2, phone_number VARCHAR2, job_id VARCHAR2, salary NUMBER, commission_pct NUMBER, manager_id NUMBER, department_id NUMBER) RETURN NUMBER;
PROCEDURE fire_employee (emp_id NUMBER);

-- overloaded subprogram

PROCEDURE fire_employee (emp_email VARCHAR2);

-- overloaded subprogram

PROCEDURE raise_salary(emp_id NUMBER, amount NUMBER);
FUNCTION nth_highest_salary (n NUMBER) RETURN EmpRecTyp;
END emp_admin; /
CREATE OR REPLACE PACKAGE BODY emp_admin AS number_hired NUMBER;

-- visible only in this package
-- Fully define cursor specified in package

CURSOR desc_salary RETURN EmpRecTyp IS SELECT employee_id, salary FROM employees ORDER BY salary DESC;

-- Fully define subprograms specified in package

FUNCTION hire_employee(last_name VARCHAR2, first_name VARCHAR2, email VARCHAR2, phone_number VARCHAR2, job_id VARCHAR2, salary NUMBER, commission_pct NUMBER, manager_id

```
NUMBER, department_id NUMBER) RETURN NUMBER IS new_emp_id
NUMBER;
BEGIN SELECT employees_seq.NEXTVAL INTO new_emp_id FROM
dual;
INSERT INTO employees VALUES (new_emp_id, last_name, first_name,
email, phone_number, SYSDATE, job_id, salary, commission_pct,
manager_id, department_id); number_hired := number_hired + 1;
DBMS_OUTPUT.PUT_LINE('The number of employees hired is ' ||
TO_CHAR(number_hired) ); RETURN new_emp_id;
END hire_employee;
PROCEDURE fire_employee (emp_id NUMBER) IS BEGIN DELETE
FROM employees WHERE
employee_id = emp_id; END fire_employee;
PROCEDURE fire_employee (emp_email VARCHAR2)
IS BEGIN DELETE FROM employees WHERE email = emp_email;
END fire_employee;

-- Define local function, available only inside package
FUNCTION sal_ok (jobid VARCHAR2, sal NUMBER) RETURN
BOOLEAN IS min_sal NUMBER;
max_sal NUMBER; BEGIN SELECT MIN(salary), MAX(salary) INTO
min_sal, max_sal FROM employees WHERE job_id = jobid;
RETURN (sal >= min_sal) AND (sal <= max_sal);
END sal_ok;
PROCEDURE raise_salary (emp_id NUMBER, amount NUMBER) IS sal
NUMBER(8,2);
jobid VARCHAR2(10); BEGIN SELECT job_id, salary INTO jobid, sal
FROM employees WHERE employee_id = emp_id;
IF sal_ok(jobid, sal + amount)
THEN UPDATE employees SET salary = salary + amount WHERE
employee_id = emp_id;

ELSE RAISE invalid_salary;
END IF;
EXCEPTION

-- exception-handling part starts here
WHEN invalid_salary THEN DBMS_OUTPUT.PUT_LINE('The salary is
out of the specified range.');
END raise_salary;
```

```
FUNCTION nth_highest_salary (n NUMBER) RETURN EmpRecTyp IS emp_rec EmpRecTyp; BEGIN OPEN desc_salary;
FOR i IN 1..n LOOP FETCH desc_salary INTO emp_rec;
END LOOP;
CLOSE desc_salary;
RETURN emp_rec;
END nth_highest_salary;
BEGIN

--initialization part starts here
INSERT INTO emp_audit VALUES (SYSDATE, USER, 'EMP_ADMIN'); number_hired := 0; END emp_admin; /

-- calling the package procedures
DECLARE new_emp_id NUMBER(6);
BEGIN new_emp_id := emp_admin.hire_employee('Belden', 'Enrique', 'EBELDEN', '555.111.2222', 'ST_CLERK', 2500, .1, 101, 110);
DBMS_OUTPUT.PUT_LINE('The new employee id is ' || TO_CHAR(new_emp_id) );
EMP_ADMIN.raise_salary(new_emp_id, 100);
DBMS_OUTPUT.PUT_LINE('The 10th highest salary is '|| TO_CHAR(emp_admin.nth_highest_salary(10).sal) || ', belonging to employee: ' ||
TO_CHAR(emp_admin.nth_highest_salary(10).emp_id) );
emp_admin.fire_employee(new_emp_id);

-- you could also delete the newly added employee as follows: -- emp_admin.fire_employee('EBELDEN');
END; /
```

Expected Output

Created package mentioned above.

Exp.No 12	CREATION OF DATABASE TRIGGERS AND CURSORS	Date……

Q1 : Trigger for displaying grade of the student
Q2 : Program to indicate invalid condition using trigger
Q3 : Cursor program for electricity bill calculation

Theory

Cursor– We have seen how oracle executes an SQL statement. Oracle DBA uses a work area for its internal processing. This work area is private to SQL's operation and is called a cursor.
The data that is stored in the cursor is called the Active Data set. The size of the cursor in memory is the size required to hold the number of rows in the Active Data Set.

Explicit Cursor- You can explicitly declare a cursor to process the rows individually. A cursor declared by the user is called Explicit Cursor. For Queries that return more than one row, You must declare a cursor explicitly.
The data that is stored in the cursor is called the Active Data set. The size of the cursor in memory is the size required to hold the number of rows in the Active

Why use an Explicit Cursor- Cursor can be used when the user wants to process data one row at a time.

Explicit Cursor Management- The steps involved in declaring a cursor and manipulating data in the active data set are:-

- Declare a cursor that specifies the SQL select statement that you want to process.
- Open the Cursor.
- Fetch the data from the cursor one row at a time.
- Close the cursor.

Explicit Cursor Attributes- Oracle provides certain attributes/ cursor variables to
control the execution of the cursor. Whenever any cursor(explicit or implicit) is opened

and used Oracle creates a set of four system variables via which Oracle keeps track of the
'Current' status of the cursor.

- declare a cursor that specifies the SQL select statement that you want to process.
- Open the Cursor.
- Fetch the data from the cursor one row at a time.
- Close the cursor.

How to Declare the Cursor:-
The General Syntax to create any particular cursor is as follows:-
Cursor <Cursorname> is Sql Statement;

How to Open the Cursor:-
The General Syntax to Open any particular cursor is as follows:-
Open Cursorname;

How to apply DataBase Triggers:-
A trigger has three basic parts:-
1. A triggering event or statement.
2. A trigger restriction
3. A trigger action.
Types of Triggers:-
Using the various options , four types of triggers can be created:-
1. Before Statement Trigger:- Before executing the triggering statement, the trigger action is executed.
2. Before Row Trigger:- Before modifying the each row affected by the triggering statement and before appropriate integrity constraints, the trigger is executed if the trigger restriction either evaluated to TRUE or was not included.'
3. After Statement Trigger:- After executing the triggering statement and applying any deferred integrity constraints, the trigger action is executed.
4. After row Trigger:- After modifying each row affected by the triggering statement and possibly applying appropriate integrity constraints, the trigger action is executed for the current row if the trigger restriction either evaluates to TRUE or was not included.

Syntax For Creating Trigger:-
The syntax for Creating the Trigger is as follows:-

Create or replace Trigger<Triggername> {Before,After} {Delete, Insert, Update } On
<Tablename> For Each row when Condition
Declare
<Variable declarations>;
<Constant Declarations>;
Begin
<PL/SQL> Subprogram Body;
Exception
Exception Pl/SQL block;
End;
How to Delete a Trigger:-
The syntax for Deleting the Trigger is as follows:-
Drop Trigger <Triggername>;

Queries :

Q1 : Trigger for displaying grade of the student

```
SQL> create table stdn(rollno number(3),name varchar(2),m1 number(3),m2 number(3),m3 number(3),tot number(3),avrg number(3),result varchar(10));
Table created.
SQL> create or replace trigger t1 before insert on stdn
2 for each row
3 begin
4 :new.tot:=:new.m1+:new.m2+:new.m3;
5 :new.avrg:=:new.tot/3;
6 if(:new.m1>=50 and :new.m2>=50 and :new.m3>=50) then
7 :new.result:='pass';
8 else
9 :new.result:='Fail';
10 end if;
11 end;
12 /
Trigger created.
SQL> insert into stdn values(101,'SM',67,89,99,",",");
1 row created.
SQL> select * from stdn;
ROLLNO NA M1 M2 M3 TOT AVRG Expected Outcome
--------- -- --------- --------- --------- --------- --------- ----------
101 SM 67 89 99 255 85 pass
```

Q2 : Program to indicate invalid condition using trigger

```
SQL> create table emp (name varchar(10),empno number(3),age number(3));
Table created.
SQL>
1 create or replace trigger t2 before insert on emp
2 for each row
3 when(new.age>100)
4 begin
5 RAISE_APPLICATION_ERROR(-20998,'INVALID ERROR');
6* end;
SQL> /
Trigger created.
SQL> insert into emp values('nithya',101,24);
1 row created.
SQL> insert into emp values('nithya',101,103);
insert into emp values('nithya',101,103)
*
ERROR at line 1:
ORA-20998: INVALID ERROR
ORA-06512: at "SCOTT.T2", line 2
ORA-04088: error during execution of trigger 'SCOTT.T2'
```

Q3 : Cursor program for electricity bill calculation

```
SQL> create table bill(name varchar2(10), address varchar2(20), city varchar2(20), unit
number(10));
Table created.
SQL> insert into bill values('&name','&addess','&city','&unit');
Enter value for name: yuva
Enter value for addess: srivi
Enter value for city: srivilliputur
Enter value for unit: 100
old 1: insert into bill values('&name','&addess','&city','&unit')
new 1: insert into bill values('yuva','srivi','srivilliputur','100')
1 row created.
SQL> /
Enter value for name: nithya
Enter value for addess: Lakshmi nagar
Enter value for city: sivakasi
Enter value for unit: 200
old 1: insert into bill values('&name','&addess','&city','&unit')
new 1: insert into bill values('nithya','Lakshmi nagar','sivakasi','200')
```

```
1 row created.
SQL> /
Enter value for name: maya
Enter value for addess: housing board
Enter value for city: sivakasi
Enter value for unit: 300
old 1: insert into bill values('&name','&addess','&city','&unit')
new 1: insert into bill values('maya','housing board','sivakasi','300')
1 row created.
SQL> /
Enter value for name: jeeva
Enter value for addess: RRR nagar
Enter value for city: sivaganagai
Enter value for unit: 400
old 1: insert into bill values('&name','&addess','&city','&unit')
new 1: insert into bill values('jeeva','RRR nagar','sivaganagai','400')
1 row created.
SQL> select * from bill;
NAME ADDRESS CITY UNIT
---------- ------------------- ------------------- ---------
yuva srivi srivilliputur100
nithya Lakshmi nagar sivakasi 200
maya housing board sivakasi 300
jeeva RRR nagar sivaganagai400
SQL> declare
2 cursor c is select * from bill;
3 b bill %ROWTYPE;
4 begin
5 open c;
6 dbms_output.put_line('Name Address city Unit Amount');
7 loop
8 fetch c into b;
9 if(c % notfound) then
10 exit;
11 else
12 if(b.unit<=100) then
13 dbms_output.put_line(b.name||' '||b.address||'
'||b.city||'
'||b.unit||'
'||b.uni
```

```
t*1);
14 elsif(b.unit>100 and b.unit<=200) then
15 dbms_output.put_line(b.name||'
'||b.address||'
'||b.city||' '||b.unit||' '||b.
'||b.city||' '||b.unit||' '||b.
unit*2);
16 elsif(b.unit>200 and b.unit<=300) then
17 dbms_output.put_line(b.name||'
'||b.address||'
unit*3);
18 elsif(b.unit>300 and b.unit<=400) then
19 dbms_output.put_line(b.name||' '||b.address||' '||b.city||'
'||b.unit||' '||b.unit*
'||b.unit||' '||b.unit*
4);
20 else
21 dbms_output.put_line(b.name||' '||b.address||' '||b.city||'
5);
22 end if;
23 end if;
24 end loop;
25 close c;
26 end;
27 /
Name Addresscity UnitAmount
yuva srivi srivilliputur100100
nithyaLakshmi nagarsivakasi200400
mayahousing board sivakasi300900
jeeva RRR nagar sivaganagai4001600
PL/SQL procedure successfully completed.
```

Expected Output

Created database Triggers and Cursors as mentioned above.

Exp.No 13	MINI PROJECT	Date……

Aim :Mini project (Application Development using Oracle/ MySQL using Database Connectivity)

Expected Output

Any mini project related to the following areas :
a.Inventory Control System.
b. Material Requirement Processing.
c. Hospital Management System.
d. Railway Reservation System.
e. Personal Information System.
f. Web Based User Identification System.
g. Timetable Management System.
h. Hotel Management System.

Contents of PROJECT REPORT

1.Project Title

2.Certificate

3.Acknowledgement

4.System Overview

5.-- Current system

6.-- Objectives of the proposed system

7.Advantages of the Proposed system (over current)

8.E.R.Diagram

9.-- Entities

10.-- Relationships

11.--Mapping Constraints

12.Database Schema/Table Definition

13.-- Table Name

14.-- Field Name

15.-- Datatype

16.-- Field size

17.-- Constraint (e.g. autogenerated, primary key, foreign key)

18.-- Validation (e.g. not null, default value)

19.Implementation

20.Output

21.Future Enhancements of the system

22.Bibliography