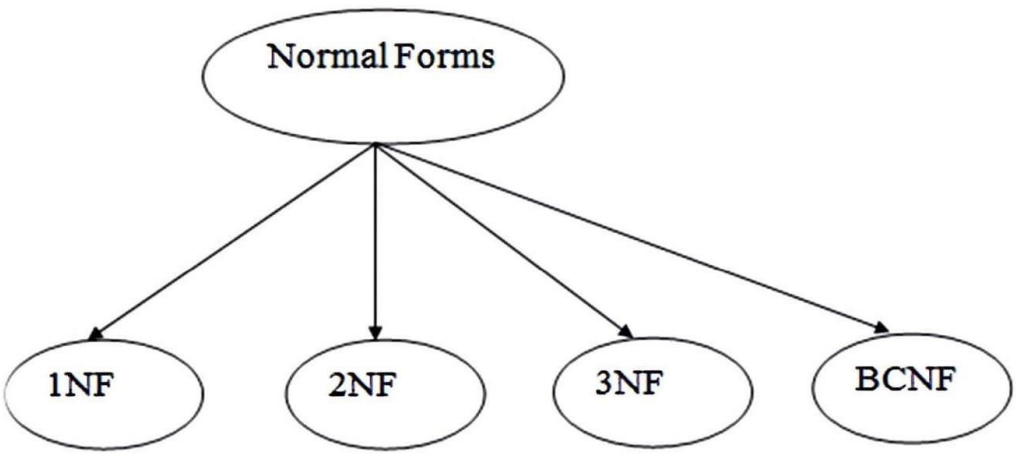
Experiment no: - 4

Objective: - Normalization

Theory: -

- Normalization is the process of organizing the data in the database.
- Normalization is used to minimize the redundancy from a relation or set of relations. It is also used to eliminate the undesirable characteristics like Insertion, Update and Deletion Anomalies.
- Normalization divides the larger table into the smaller table and links them using relationship.
- The normal form is used to reduce redundancy from the database table.

Types of Normalization are:-



1. First Normal Form

If a relation contain composite or multi-valued attribute, it violates first normal form or a relation is in first normal form if it does not contain any composite or multi-valued attribute. A relation is in first normal form if every attribute in that relation is **singled valued attribute**.

- **Example 1 –** Relation STUDENT in table 1 is not in 1NF because of multi-valued attribute STUD_PHONE. Its decomposition into 1NF has been shown in table 2.

STUD_NO	STUD_NAME	STUD_PHONE	STUD_STATE	STUD_COUNTRY
1	RAM	9716271721, 9871717178	HARYANA	INDIA
2	RAM	9898297281	PUNJAB	INDIA
3	SURESH		PUNJAB	INDIA

Table 1

Conversion to first normal form

STUD_NO	STUD_NAME	STUD_PHONE	STUD_STATE	STUD_COUNTRY
1	RAM	9716271721	HARYANA	
1	RAM	9871717178	HARYANA	INDIA
2	RAM	9898297281	PUNJAB	INDIA
3	SURESH		PUNJAB	INDIA

Table 2

2. Second Normal Form

To be in second normal form, a relation must be in first normal form and relation must not contain any partial dependency. A relation is in 2NF if it has **No Partial Dependency,** i.e., no non-prime attribute (attributes which are not part of any candidate key) is dependent on any proper subset of any candidate key of the table.

Partial Dependency – If the proper subset of candidate key determines non-prime attribute, it is called partial dependency.

Example 1 – Consider table-3 as following below.

STUD_NO	COURSE_NO	COURSE_FEE
1	C1	1000
2	C2	1500
1	C4	2000
4	C3	1000
4	C1	1000
2	C5	2000

{Note that, there are many courses having the same course fee. }

Here,
COURSE_FEE cannot alone decide the value of COURSE_NO or STUD_NO;
COURSE_FEE together with STUD_NO cannot decide the value of COURSE_NO;
COURSE_FEE together with COURSE_NO cannot decide the value of STUD_NO;
Hence,

COURSE_FEE would be a non-prime attribute, as it does not belong to the one only candidate key {STUD_NO, COURSE_NO} ;
But, COURSE_NO -> COURSE_FEE , i.e., COURSE_FEE is dependent on COURSE_NO, which is a proper subset of the candidate key. Non-prime attribute COURSE_FEE is dependent on a proper subset of the candidate key, which is a partial dependency and so this relation is not in 2NF.

To convert the above relation to 2NF,
we need to split the table into two tables such as :
Table 1: STUD_NO, COURSE_NO
Table 2: COURSE_NO, COURSE_FEE

Table 1	
STUD_NO	COURSE_NO
1	C1
2	C2
1	C4
3	C3
4	C3
4	C1
2	C5

Table 2	
COURSE_NO	COURSE_FEE
C1	1000
C2	1500
C3	1000
C4	2000
C5	2000

NOTE: 2NF tries to reduce the redundant data getting stored in memory. For instance, if there are 100 students taking C1 course, we don't need to store its Fee as 1000 for all the 100 records, instead once we can store it in the second table as the course fee for C1 is 1000.

3. Third Normal Form –

A relation is in third normal form, if there is **no transitive dependency** for non-prime attributes as well as it is in second normal form.
A relation is in 3NF if **at least one of the following condition holds** in every non-trivial function dependency X –> Y

1. X is a super key.
2. Y is a prime attribute (each element of Y is part of some candidate key).

Example:

EMPLOYEE_DETAIL table:

EMP_ID	EMP_NAME	EMP_ZIP	EMP_STATE	EMP_CITY

222	Harry	201010	UP	Noida
333	Stephan	02228	US	Boston
444	Lan	60007	US	Chicago
555	Katharine	06389	UK	Norwich
666	John	462007	MP	Bhopal

Super key in the table above:

1. {EMP_ID}, {EMP_ID, EMP_NAME}, {EMP_ID, EMP_NAME, EMP_ZIP}....so on

   Candidate key: {EMP_ID}

   Non-prime attributes: In the given table, all attributes except EMP_ID are non-prime.

   Here, EMP_STATE & EMP_CITY dependent on EMP_ZIP and EMP_ZIP dependent on EMP_ID. The non-prime attributes (EMP_STATE, EMP_CITY) transitively dependent on super key(EMP_ID). It violates the rule of third normal form.

   That's why we need to move the EMP_CITY and EMP_STATE to the new <EMPLOYEE_ZIP> table, with EMP_ZIP as a Primary key.

   EMPLOYEE table:

EMP_ID	EMP_NAME	EMP_ZIP
222	Harry	201010
333	Stephan	02228
444	Lan	60007
555	Katharine	06389
666	John	462007

EMPLOYEE_ZIP table:

EMP_ZIP	EMP_STATE	EMP_CITY
201010	UP	Noida
02228	US	Boston
60007	US	Chicago
06389	UK	Norwich
462007	MP	Bhopal

4. Boyce-Codd Normal Form (BCNF)

A relation R is in BCNF if R is in Third Normal Form and for every FD, LHS is super key. A relation is in BCNF iff in every non-trivial functional dependency X –> Y, X is a super key.

Example: Let's assume there is a company where employees work in more than one department.

EMPLOYEE table:

EMP_ID	EMP_COUNTRY	EMP_DEPT	DEPT_TYPE	EMP_DEPT_NO
264	India	Designing	D394	283
264	India	Testing	D394	300
364	UK	Stores	D283	232
364	UK	Developing	D283	549

In the above table Functional dependencies are as follows:

1. EMP_ID → EMP_COUNTRY
2. EMP_DEPT → {DEPT_TYPE, EMP_DEPT_NO}

Candidate key: {EMP-ID, EMP-DEPT}

The table is not in BCNF because neither EMP_DEPT nor EMP_ID alone are keys.

To convert the given table into BCNF, we decompose it into three tables:

EMP_COUNTRY table:

EMP_ID	EMP_COUNTRY
264	India
264	India

EMP_DEPT table:

EMP_DEPT	DEPT_TYPE	EMP_DEPT_NO
Designing	D394	283
Testing	D394	300
Stores	D283	232
Developing	D283	549

EMP_DEPT_MAPPING table:

EMP_ID	EMP_DEPT
D394	283
D394	300
D283	232
D283	549

Functional dependencies:

1. EMP_ID → EMP_COUNTRY
2. EMP_DEPT → {DEPT_TYPE, EMP_DEPT_NO}

Candidate keys:

For the first table: EMP_ID
For the second table: EMP_DEPT
For the third table: {EMP_ID, EMP_DEPT}

Now, this is in BCNF because left side part of both the functional dependencies is a key.

Experiment no: - 5

Objective: - Creating cursor

Theory:-

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

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

- Implicit cursors
- Explicit cursors

Implicit Cursors

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

Whenever a DML statement (INSERT, UPDATE and DELETE) is issued, an implicit cursor is associated with this statement. For INSERT operations, the cursor holds the data that needs to be inserted. For UPDATE and DELETE operations, the cursor identifies the rows that would be affected.

In PL/SQL, we can refer to the most recent implicit cursor as the **SQL cursor**, which always has attributes such as **%FOUND, %ISOPEN, %NOTFOUND**, and **%ROWCOUNT**. The SQL cursor has additional attributes, **%BULK_ROWCOUNT** and **%BULK_EXCEPTIONS**, designed for use with the **FORALL** statement. The following table provides the description of the most used attributes −

S.No	Attribute & Description
1	**%FOUND** Returns TRUE if an INSERT, UPDATE, or DELETE statement affected one or more rows or a SELECT INTO statement returned one or more rows. Otherwise, it returns FALSE.
2	**%NOTFOUND** The logical opposite of %FOUND. It returns TRUE if an INSERT, UPDATE, or DELETE statement affected no rows, or a SELECT INTO statement returned no rows. Otherwise, it returns FALSE.

3	**%ISOPEN** Always returns FALSE for implicit cursors, because Oracle closes the SQL cursor automatically after executing its associated SQL statement.
4	**%ROWCOUNT** Returns the number of rows affected by an INSERT, UPDATE, or DELETE statement, or returned by a SELECT INTO statement.

Any SQL cursor attribute will be accessed as **SQL%attribute_name** as shown below in the example.

Example

Select * from customers;

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

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

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

```
/
```

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

6 customers selected

PL/SQL procedure successfully completed.

If you check the records in customers table, you will find that the rows have been updated −

Select * from customers;

ID	NAME	AGE	ADDRESS	SALARY
1	Ramesh	32	Ahmedabad	2500.00
2	Khilan	25	Delhi	2000.00
3	Kaushik	23	Kota	2500.00
4	Chaitali	25	Mumbai	7000.00
5	Hardik	27	Bhopal	9000.00
6	Komal	22	MP	5000.00

Explicit Cursors

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

The syntax for creating an explicit cursor is −

```
CURSOR cursor_name IS select_statement;
```

Working with an explicit cursor includes the following steps −

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

Declaring the Cursor

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

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

Opening the Cursor

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

```
OPEN c_customers;
```

Fetching the Cursor

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

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

Closing the Cursor

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

```
CLOSE c_customers;
```

Example

Following is a complete example to illustrate the concepts of explicit cursors &minua;

```
DECLARE
   c_id customers.id%type;
   c_name customerS.No.ame%type;
   c_addr customers.address%type;
   CURSOR c_customers is
      SELECT id, name, address FROM customers;
BEGIN
   OPEN c_customers;
   LOOP
   FETCH c_customers into c_id, c_name, c_addr;
      EXIT WHEN c_customers%notfound;
      dbms_output.put_line(c_id || ' ' || c_name || ' ' || c_addr);
   END LOOP;
   CLOSE c_customers;
END;
/
```

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

```
1 Ramesh Ahmedabad
2 Khilan Delhi
3 kaushik Kota
```

4 Chaitali Mumbai
5 Hardik Bhopal
6 Komal MP

SQL procedure successfully completed.

Experiment no: - 6

Objective: - Creating procedure and functions.

Theory:-

Procedures

Create and drop procedures in MySQL with syntax and examples.

In MySQL, a procedure is a stored program that you can pass parameters into. It does not return a value like a function does.

Syntax

The syntax to create a procedure in MySQL is:

```
CREATE PROCEDURE procedure_name [ (parameter datatype [, parameter datatype]) ]


BEGIN
  declaration_section
  executable_section
END;
```

procedure_name

The name to assign to this procedure in MySQL.

parameter

Optional. One or more parameters passed into the procedure. When creating a procedure, there are three types of parameters that can be declared:

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

declaration_section

The place in the procedure where you declare local variables.

executable_section

The place in the procedure where you enter the code for the procedure.

Example

Let's look at an example that shows how to create a procedure in MySQL:

```
DELIMITER //

CREATE procedure CalcIncome ( OUT ending_value INT )

BEGIN

  DECLARE income INT;

  SET income = 50;

  label1: WHILE income <= 3000 DO
    SET income = income * 2;
  END WHILE label1;

  SET ending_value = income;

END; //

DELIMITER ;
```

You could then reference your new procedure as follows:

```
CALL CalcIncome (@variable_name);

SELECT @variable_name;
```

Drop procedure

Once you have created your procedure in MySQL, you might find that you need to remove it from the database.

Syntax

The syntax to a drop a procedure in MySQL is:

```
DROP procedure [ IF EXISTS ] procedure_name;
```

procedure_name

The name of the procedure that you wish to drop.

Example

Let's look at an example of how to drop a procedure in MySQL.

For example:

```
DROP procedure CalcIncome;
```

This example would drop the procedure called *CalcIncome*.

Function

create and drop functions in MySQL with syntax and examples.

In MySQL, a function is a stored program that you can pass parameters into and then return a value.

Create Function

Just as you can create functions in other languages, you can create your own functions in MySQL. Let's take a closer look.**Syntax**

The syntax to create a function in MySQL is:

```
CREATE FUNCTION function_name [ (parameter datatype [, parameter datatype]) ]
RETURNS return_datatype
BEGIN
  declaration_section
  executable_section
END;
```

function_name

The name to assign to this function in MySQL.

parameter

One or more parameters passed into the function. When creating a function, all parameters are considered to be **IN parameters** (not OUT or INOUT parameters) where the parameters can be referenced by the function but can not be overwritten by the function.

return_datatype

The data type of the function's return value.

declaration_section

The place in the function where you declare local variables.

executable_section

The place in the function where you enter the code for the function.

Example

Let's look at an example that shows how to create a function in MySQL:

```
DELIMITER //

CREATE FUNCTION CalcIncome ( starting_value INT )
RETURNS INT

BEGIN

  DECLARE income INT;

  SET income = 0;

  label1: WHILE income <= 3000 DO
    SET income = income + starting_value;
  END WHILE label1;

  RETURN income;

END; //

DELIMITER ;
```

You could then reference your new function as follows:

```
SELECT CalcIncome (1000);
```

Drop Function

Once you have created your function in MySQL, you might find that you need to remove it from the database.

Syntax

The syntax to a drop a function in MySQL is:

```
DROP FUNCTION [ IF EXISTS ] function_name;
```

function_name

The name of the function that you wish to drop.

Example

Let's look at an example of how to drop a function in MySQL.

For example:

```
DROP FUNCTION CalcIncome;
```

This example would drop the function called *CalcIncome*.

Experiment - 7

Objective: -Design and implementation of payroll processing system

Theory

This ER (Entity Relationship) Diagram represents the model of Payroll Management System Entity. The entity-relationship diagram of Payroll Management System shows all the visual instrument of database tables and the relations between Salary, Appraisals, Payroll, and Payments etc. It used structure data and to define the relationships between structured data groups of Payroll Management System functionalities. The main entities of the Payroll Management System are Payroll, Salary, Employee, Appraisals, Working Points and Payments.

Payroll Management System entities and their attributes :

Payroll Entity : Attributes of Payroll are payroll_id, payroll_employee_id, payroll_title, payroll_type, payroll_description

Salary Entity : Attributes of Salary are salary_id, salary_employee_id, salary_amount, salary_total, salary_type, salary_description

Employee Entity : Attributes of Employee are employee_id, employee_name, employee_mobile, employee_email, employee_username, employee_password, employee_address

Appraisals Entity : Attributes of Appraisals are apprasail_id, apprasail_employee_id, apprasail_name, apprasail_type, apprasail_description

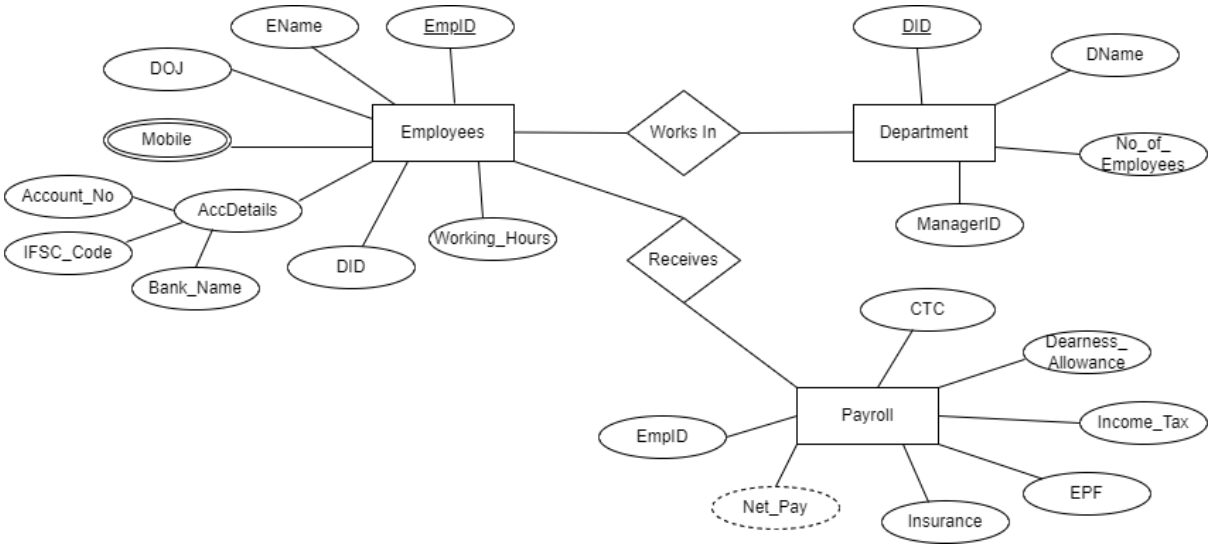
Working Points Entity : Attributes of Working Points are point_id, point_title, point_type, point_description

Payments Entity : Attributes of Payments are payment_id, payment_customer_id, payment_date, payment_amount, payment_description

Description of Payroll Management System Database :

The details of Payroll is store into the Payroll tables respective with all tables Each entity (Payments, Employee, Working Points, Salary, Payroll) contains primary key and unique keys. The entity Employee, Working Points has binded with Payroll, Salary entities with foreign key There is one-to-one and one-to-many relationships available between Working Points, Appraisals, Payments, Payroll All the entities Payroll, Working Points, Employee, Payments are normalized and reduce duplicity of records We have implemented indexing on each tables of Payroll Management System tables for fast query execution.

ER DIAGRAM OF PAYROLL MANAGEMENT SYSTEM



Experiment no: - 8

Objective: - Design and implementation of Library information system.

Theory:-

This ER (Entity Relationship) Diagram represents the model of Library Information System Entity. The entity-relationship diagram of Library Information System shows all the visual instrument of database tables and the relations between Book Maintenance, Membership Management, Student, Member etc. It used structure data and to define the relationships between structured data groups of Library Information System functionalities. The main entities of the Library Information System are Student, Book Maintenance, Fees Collection, Membership Management, Facilities and Member.

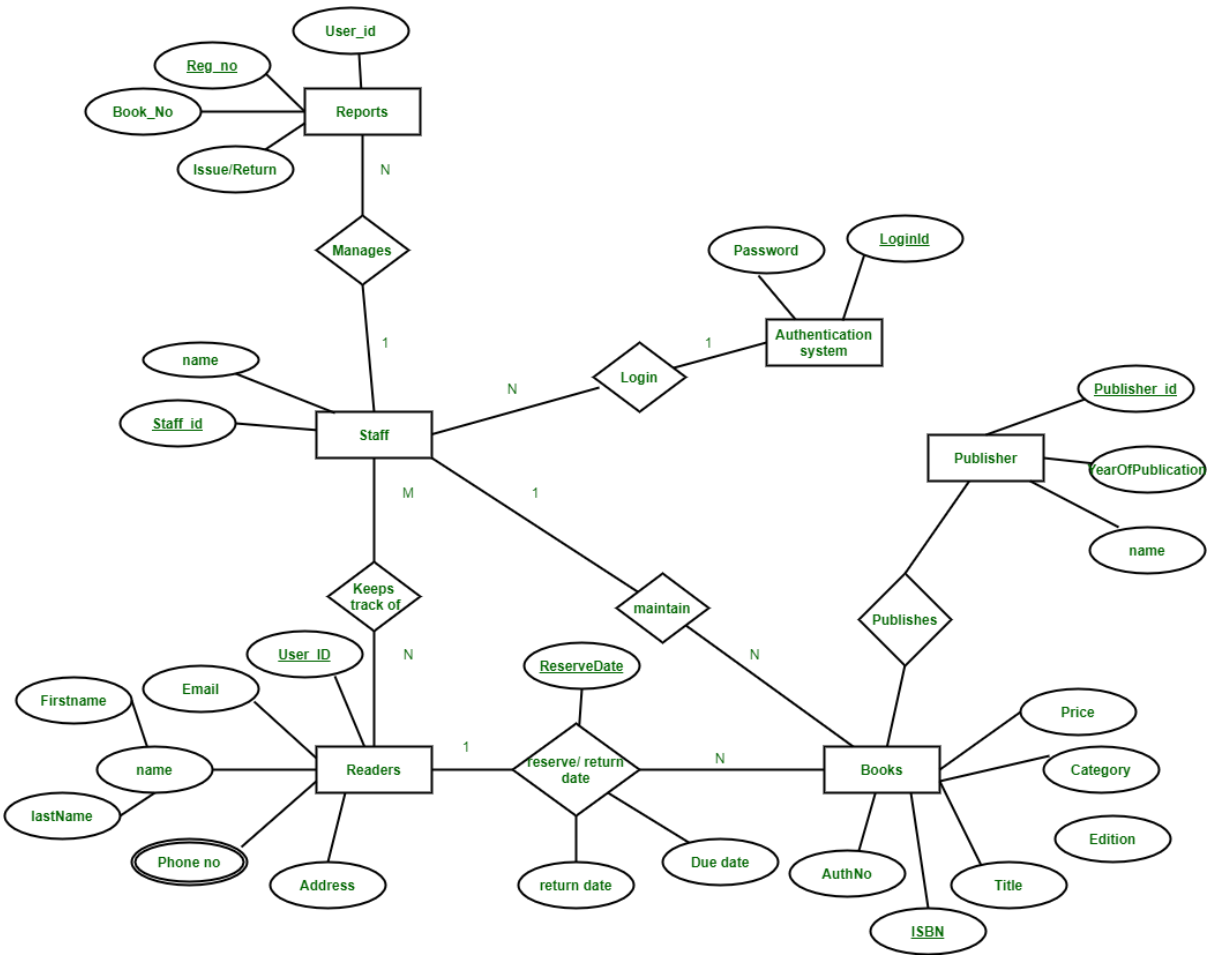
Library Information System entities and their attributes:

- **Student Entity** : Attributes of Student are student_id, student_college_id, student_name, student_mobile, student_email, student_username, student_password, student_address
- **Book Maintenance Entity** : Attributes of Book Maintenance are maintenance_id, maintenance_type, maintenance_date, maintenance_description, maintenance_number
- **Fees Collection Entity** : Attributes of Fees Collection are fees_id, fees_amount, fees_type, fees_description
- **Membership Management Entity** : Attributes of Membership Management are membership_id, membership_student_id, membership_name, membership_type, membership_description
- **Facilities Entity** : Attributes of Facilities are Facility_id, Facility_name, Facility_type, Facility_description
- **Member Entity** : Attributes of Member are memeber_id, memeber_name, memeber_mobile, memeber_email, memeber_username, memeber_password, memeber_address

Description of Library Information System Database :

- The details of Student is store into the Student tables respective with all tables
- Each entity (Member, Fees Collection, Facilities, Book Maintenance, Student) contains primary key and unique keys.
- The entity Fees Collection, Facilities has binded with Student, Book Maintenance entities with foreign key
- There is one-to-one and one-to-many relationships available between Facilities, Membership Management, Member, Student
- All the entities Student, Facilities, Fees Collection, Member are normalized and reduce duplicity of records
- We have implemented indexing on each tables of Library Information System tables for fast query execution.

ER DIAGRAM OF LIBRARY MANAGEMENT SYSTEM



Experiment no: - 9

Objective: - Design and implement Student information system

Theory:-

This ER (Entity Relationship) Diagram represents the model of Student Information System Entity. The entity-relationship diagram of Student Information System shows all the visual instrument of database tables and the relations between Administrator, Student, Course, Department, Exam etc. It used structure data and to define the relationships between structured data groups of Student Information System functionalities. The main entities of the Student Information System are Administrator, Student, Course, Department, Attendance, Section and Exam.

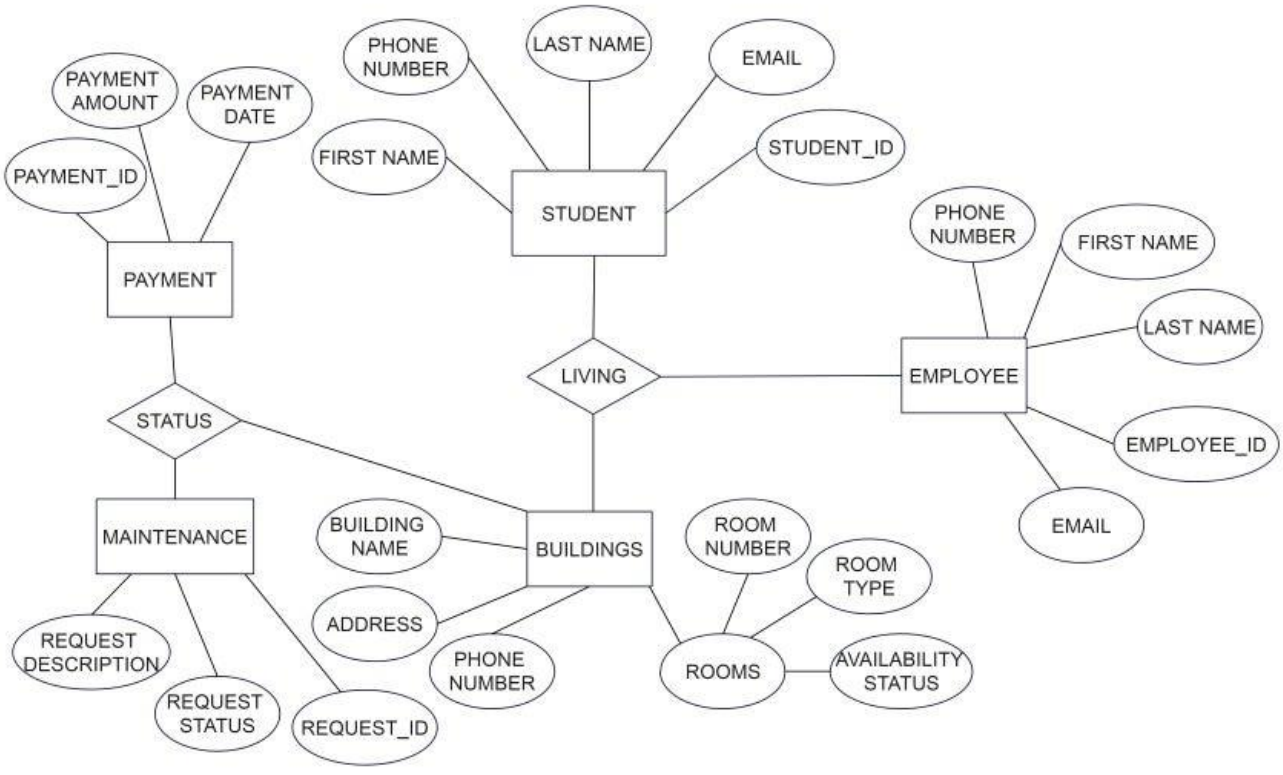
Student Information System entities and their attributes:

- **Administrator Entity :** Attributes of administrator are admin_id, admin_name, admin_password
- **Student Entity :** Attributes of Student are roll_no, s_name, s_address, contact_no
- **Course Entity :** Attributes of Course are couse_id, c_name, Qualification
- **Department Entity:** Attributes of Department are dept_id, dept_name
- **Attendance Entity:** Attributes of Attendance are roll_no, Student_name, Course, Percentage
- **Section Entity:** Attributes of Section are section_id, section_name
- **Exam Entity:** Attributes of Exam are roll_no, Course, Student_name, Student_marks

Description of Strudent Information System Database :

- The details of Student is store into the Student tables respective with all tables
- Each entity (Administrator, Student, Course, Department, Attendance, Section and Exam) contains primary key and unique keys.
- There is one-to-one and one-to-many relationships available between administrator, Department, Student, Exam, Section
- All the entities Administrator, Student, Course, Department, Attendance, Section and Exam are normalized and reduce duplicity of records
- We have implemented indexing on each tables of Student Information System tables for fast query execution.

Design the ER diagram for Student Information System



Experiment: - 10

Objective: - Creating packages and triggers.

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

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

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

Creating Triggers

The syntax for creating a trigger is –

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

Example of the following

To start with, we will be using the CUSTOMERS table we had created

Select * from customers;

ID	Name	Age	Address	Salary
1	Ramesh	32	Ahmedabad	2000.0
2	Khilan	25	Delhi	1500.0
3	Kaushik	23	Kota	2000.0
4	Chaitali	25	Mumbai	6500.0
5	Hardik	27	Bhopal	8500.0
6	Komal	22	Mp	4500.0

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

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

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

Trigger created.

The following points need to be considered here −

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

Triggering a Trigger

Let us perform some DML operations on the CUSTOMERS table. Here is one INSERT statement, which will create a new record in the table −

```
INSERT INTO CUSTOMERS (ID,NAME,AGE,ADDRESS,SALARY)
VALUES (7, 'Kriti', 22, 'HP', 7500.00 );
```

When a record is created in the CUSTOMERS table, the above create trigger, **display_salary_changes** will be fired and it will display the following result −

Old salary:
New salary: 7500
Salary difference:

Because this is a new record, old salary is not available and the above result comes as null. Let us now perform one more DML operation on the CUSTOMERS table. The UPDATE statement will update an existing record in the table −

```
UPDATE customers
SET salary = salary + 500
WHERE id = 2;
```

When a record is updated in the CUSTOMERS table, the above create trigger, **display_salary_changes** will be fired and it will display the following result –

```
Old salary: 1500
New salary: 2000
Salary difference: 500
```

Previous Page Print Page

Creating a packages

A package will have two mandatory parts –

- Package specification
- Package body or definition

Package Specification

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

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

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

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

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

```
Package created.
```

Package Body

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

The **CREATE PACKAGE BODY** Statement is used for creating the package body. The following code snippet shows the package body declaration for the ***cust_sal*** package created above.

```
CREATE OR REPLACE PACKAGE BODY cust_sal AS
```

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

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

```
Package body created.
```

Using the Package Elements

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

```
package_name.element_name;
```

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

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

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

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

PL/SQL procedure successfully completed.
```

Example

Select * from customers;

ID	Name	Age	Address	Salary
1	Ramesh	32	Ahmedabad	2000.0
2	Khilan	25	Delhi	1500.0
3	Kaushik	23	Kota	2000.0
4	Chaitali	25	Mumbai	6500.0
5	Hardik	27	Bhopal	8500.0
6	Komal	22	Mp	4500.0

The Package Specification

```
CREATE OR REPLACE PACKAGE c_package AS
   -- Adds a customer
```

```
   PROCEDURE addCustomer(c_id   customers.id%type,
   c_name  customerS.No.ame%type,
   c_age  customers.age%type,
   c_addr customers.address%type,
   c_sal  customers.salary%type);

   -- Removes a customer
   PROCEDURE delCustomer(c_id  customers.id%TYPE);
   --Lists all customers
   PROCEDURE listCustomer;

END c_package;
/
```

When the above code is executed at the SQL prompt, it creates the above package and displays the following result −

```
Package created.
```

Creating the Package Body

```
CREATE OR REPLACE PACKAGE BODY c_package AS
   PROCEDURE addCustomer(c_id  customers.id%type,
      c_name customerS.No.ame%type,
      c_age  customers.age%type,
      c_addr  customers.address%type,
      c_sal   customers.salary%type)
   IS
   BEGIN
      INSERT INTO customers (id,name,age,address,salary)
         VALUES(c_id, c_name, c_age, c_addr, c_sal);
   END addCustomer;

   PROCEDURE delCustomer(c_id   customers.id%type) IS
   BEGIN
      DELETE FROM customers
      WHERE id = c_id;
   END delCustomer;

   PROCEDURE listCustomer IS
   CURSOR c_customers is
      SELECT  name FROM customers;
   TYPE c_list is TABLE OF customers.Name%type;
   name_list c_list := c_list();
   counter integer :=0;
   BEGIN
      FOR n IN c_customers LOOP
      counter := counter +1;
      name_list.extend;
      name_list(counter) := n.name;
      dbms_output.put_line('Customer(' ||counter||
')'||name_list(counter));
      END LOOP;
   END listCustomer;
```

```
END c_package;
/
```

The above example makes use of the **nested table**. When the above code is executed at the SQL prompt, it produces the following result –

```
Package body created.
```

Using The Package

The following program uses the methods declared and defined in the package *c_package*.

```
DECLARE
   code customers.id%type:= 8;
BEGIN
   c_package.addcustomer(7, 'Rajnish', 25, 'Chennai', 3500);
   c_package.addcustomer(8, 'Subham', 32, 'Delhi', 7500);
   c_package.listcustomer;
   c_package.delcustomer(code);
   c_package.listcustomer;
END;
/
```

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

```
Customer(1): Ramesh
Customer(2): Khilan
Customer(3): kaushik
Customer(4): Chaitali
Customer(5): Hardik
Customer(6): Komal
Customer(7): Rajnish
Customer(8): Subham
Customer(1): Ramesh
Customer(2): Khilan
Customer(3): kaushik
Customer(4): Chaitali
Customer(5): Hardik
Customer(6): Komal
Customer(7): Rajnish

PL/SQL procedure successfully completed
```