

# **D. Y. Patil College of Engineering and Technology, Kolhapur**



**Department of Computer Science and Engineering**

**Class: T.Y. (A, B, C)**

**Lab Manual**

Name of the Subject

**Database Engineering**

**2022-2023**



**D. Y. PATIL COLLEGE OF ENGINEERING & TECHNOLOGY**  
**Department of Computer Science and Engineering**  
**Academic Year 2022-23**

**Name of the course –Database Engineering Laboratory**

**Class – TY (A, B, C)**

**Course Code: 201CSP307**

**SEM- V**

**Faculty: Prof. DR. B. D. Jitkar, Miss. Shobha B. Patil, Mr. S. S. Kore**

| TEACHING SCHEME       | EXAMINATION SCHEME              |
|-----------------------|---------------------------------|
| Practical: 2 Hrs/Week | ISE Marks:25, ESE-POE Marks: 50 |

**List of Experiments**

| Sr.No. | Title of Experiment   | CO Mapped |
|--------|---|-----------|
| 1      | ER Diagram of an Organization   | CO2       |
| 2      | Conversion of ER Diagram to Tables  | CO2       |
| 3      | DDL Statements  | CO1, CO3  |
| 4      | DML Statements  | CO1, CO3  |
| 5      | SQL Character functions, String functions                                       | CO3       |
| 6      | Aggregate functions and Group by, having, between, Order by clauses             | CO3       |
| 7      | Join operations and set operations  | CO3       |
| 8      | Views, Constraints and Subqueries   | CO3       |
| 9      | Demonstrate PLSQL Functions and Procedures.                                     | CO3       |
| 10     | Demonstrate Cursors, and triggers using PL/SQL.                                 | CO3       |
| 11     | Database Connectivity   | CO3       |
| 12     | Normalize any database from first normal form to Boyce-Codd Normal Form (BCNF). | CO4       |

**Prepared by:**  
**Course Coordinator**

**Checked by:**  
**Module Coordinator**

**Verified by:**  
**Program Coordinator**

**Approved by:**  
**HODCSE**

## Experiment No. 1

**Title:** ER Diagram of an Organization.

**Objective:** Draw an E-R Diagram for any organization like Insurance Company, Library systems, College Management systems, Hospital Management systems etc.

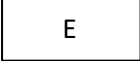

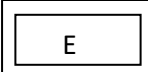

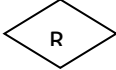

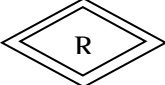
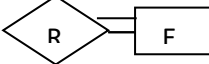
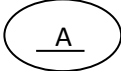
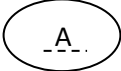
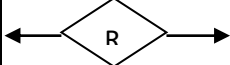

Use data modelling tools like Oracle SQL developer, Tode, ERDPlus etc. to draw ER diagram.

### **Theory:**

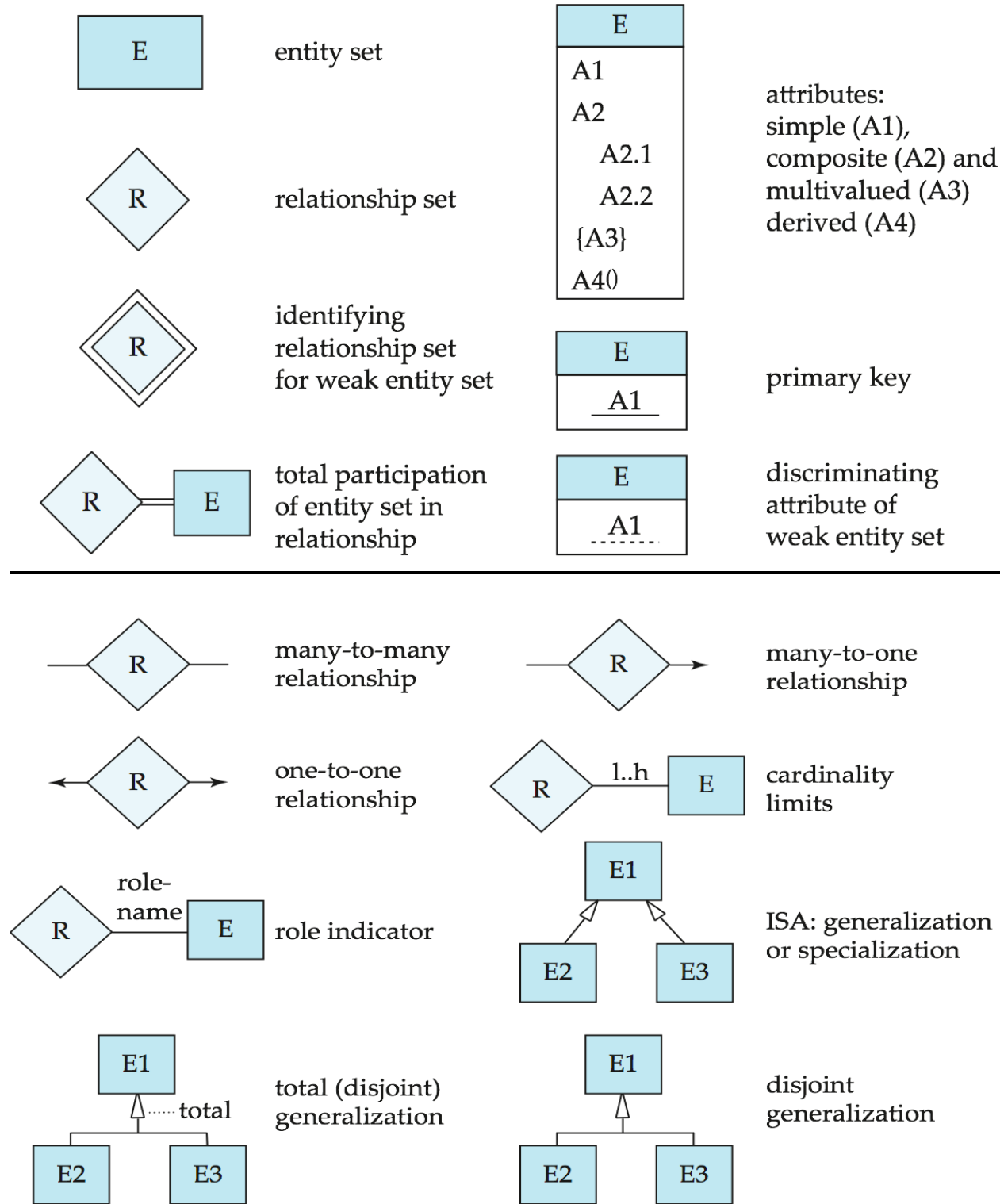
1. **Entity:** It is a “thing” or “object” in the real world that is distinguishable from all other Objects.

2. **Relation:** A relation is an association among several entities.

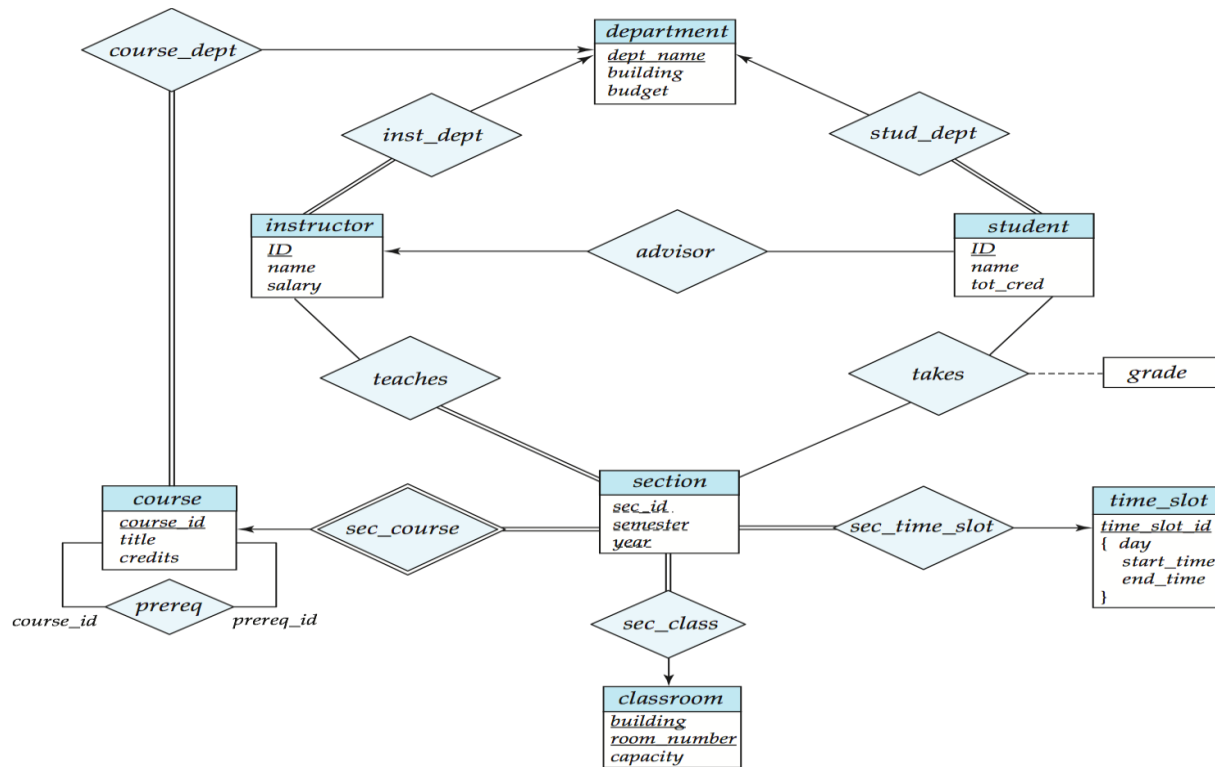
### 3. Symbols used in E-R diagram:

|   |   |
|---|---|
|  Entity Set                  |  Attribute                 |
|  Weak Entity Set             |  Multi valued Attribute    |
|  Relationship set            |  Derived attribute         |
|  Identified Relationship set |  Total Participation      |
|  Primary Key                 |  Discriminator             |
|  One to One Relationship     |  Many to One Relationship |

## Alternative ER Notations:



## E-R Diagram for a University Enterprise



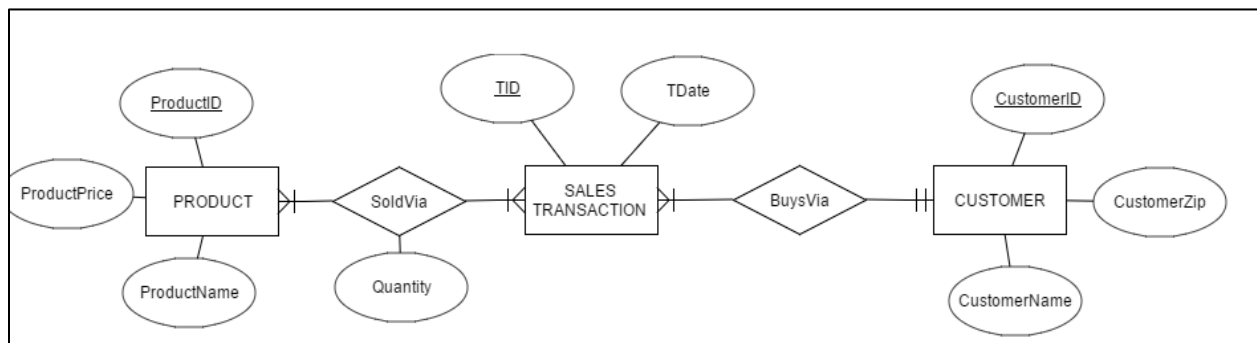
**ERDPlus** is a web-based database modeling tool that lets you quickly and easily create

- Entity Relationship Diagrams (ERDs)
- Relational Schemas (Relational Diagrams)
- Star Schemas (Dimensional Models)

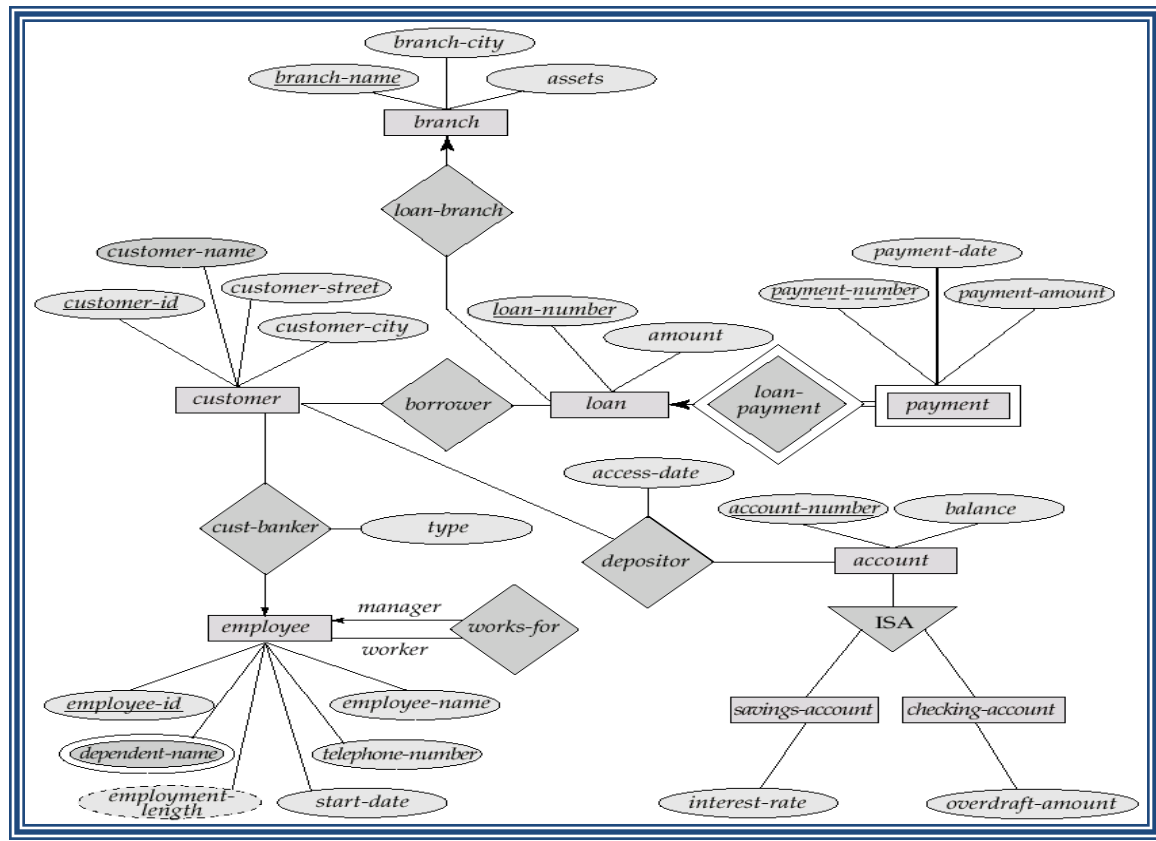
ERDPlus enables drawing standard ERD components.

- Entities, Attributes, Relationships

The notation supports drawing regular and weak entities, various types of attributes (regular, unique, multi-valued, derived, composite, and optional), and all possible cardinality constraints of relationships (mandatory-many, optional-many, mandatory-one and optional-one).



## ER Diagram for Banking Enterprise:



### Procedure:

- 1) Consider an enterprise of your choice.
- 2) Identify the entities and their attributes.
- 3) Identify the primary key of each entity.
- 4) Find the relationship between the entities. Name the relationship.
- 5) Find the cardinality of the relations & specify in ER Diagram

### Examples to solve:

1. Construct an E-R diagram for a car-insurance company whose customers own one or more cars each. Each car has associated with it zero to any number of recorded accidents.
2. Construct an E-R diagram for a hospital with a set of patients and a set of medical doctors. Associate with each patient a log of the various tests and examinations conducted.

**Outcome:** Students are able to draw an ER diagram for any organizations like hospital, University Database, Insurance company etc.

## Experiment No. 2

**Title:** Conversion of ER Diagram to Table

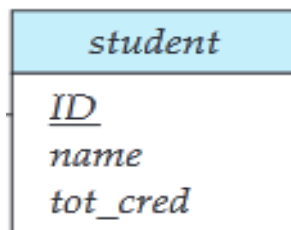
**Objective:** To identify the entities & their relationship and then convert these into corresponding records in the form of table.

### **Theory:**

#### **1. Tabular Representation of Strong Entity Sets**

Let  $E$  be a strong entity set with descriptive attributes  $a_1, a_2, \dots, a_n$ . This entity set is represented by a table called  $E$  with  $n$  distinct columns, each of which corresponds to one of the attributes of  $E$ . Each row in this table corresponds to one entity of the entity set  $E$ .

- A strong entity set reduces to a schema with the same attributes  
e.g. *student* (ID, *name*, *tot\_cred*)

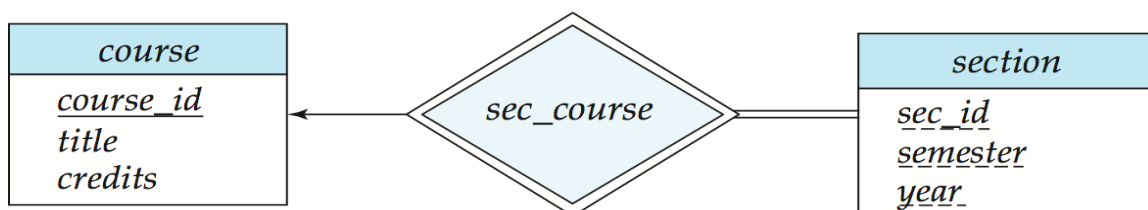


#### **2. Tabular Representation of Weak Entity Sets**

Let  $A$  be a weak entity set with attributes  $a_1, a_2, \dots, a_m$ . Let  $B$  be the strong entity set on which  $A$  depends. Let the primary key of  $B$  consist of attributes  $b_1, b_2, \dots, b_n$ . The entity set  $A$  is represented by a table called  $A$  with one column for each attribute of the set:

$$\{a_1, a_2, \dots, a_m\} \cup \{b_1, b_2, \dots, b_n\}$$

- A weak entity set becomes a table that includes a column for the primary key of the identifying strong entity set  
e.g. *section* ( *course\_id*, *sec\_id*, *sem*, *year* )



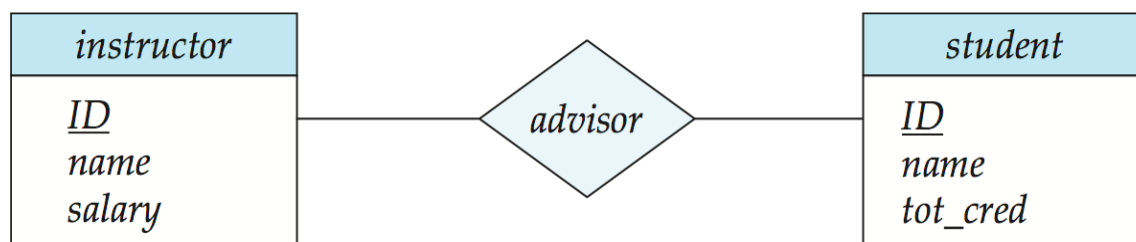
### 3. Tabular Representation of Relationship Sets

Let  $R$  be a relationship set, let  $a_1, a_2, \dots, a_m$  be the set of attributes formed by the union of the primary keys of each of the entity sets participating in  $R$ , and let the descriptive attributes (if any) of  $R$  be  $b_1, b_2, \dots, b_n$ . This relationship set is represented by a table called  $R$  with one column for each attribute of the set:

$$\{a_1, a_2, \dots, a_m\} \cup \{b_1, b_2, \dots, b_n\}$$

- Example: schema for relationship set *advisor*

e.g. *advisor* = (*s\_id*, *i\_id*)



#### **Procedure:**

- 1) Consider an ER Diagrams drawn in Experiment No: 1
- 2) Identify the entities with primary key & relationship sets.
- 3) Convert each entity set into respective tabular form using attributes defined in the definition entity.
- 4) Similarly as in step 3, convert relationship into tabular form.

**Outcome:** Students are able to design relational table for a given ER Diagram.



## Experiment No. 3

**Title:** DDL statements.

**Objective:** Study of DDL statements like create, alter, truncate, rename & drop.

**Theory:**

**1. Create table:**

create table tablename (column1 data type, ....., column n data type);

**2. Alter table :**

1. alter table tablename add columnname datatype;
2. alter table tablename modify columnname newdatatype;
3. alter table tablename drop column columnname;
4. alter table tablename ADD CONSTRAINT <constraint name> PRIMARY KEY (<attribute list>);

**3. Rename:**

1. Alter table tablename rename to new\_table\_name;
2. Alter table tablename rename column oldname to newname;
3. ALTER TABLE table\_name CHANGE old\_column\_name new\_col\_name Data Type;

**4. Drop:**

1. Drop table tablename;
2. Alter table tablename Drop Column Columnname;

**5. Truncate:**

Truncate table tablename;

Consider the student schema as shown below:

Student (Roll no, Name, Address, class)

Department(Id, Name, No. of students)

Elective\_Course(Name, Course\_Id)

Write the correct syntax statements to do the following:

1. Create relation/table for Student, Department, Elective\_Course.

2. Use the Alter command to change the table structure e.g. Change the width of address type, no. of student, etc.
3. Add one column Establishment\_year in Department table.
4. Use different form of drop statement to perform following.
  - a. To delete an entire table.
  - b. To observe the effect of truncate statement.
4. Rename Elective\_Course table to Elecourse.

**Outcome:** Students are able to create alter and remove tables from the database.

## Experiment No. 4

**Title:** DML statements

**Objective:** To study select, from, where clauses & update, delete, insert statements.

**Theory:**

**1. Select :**

select columnlist from tablename [where condition ] ;

**2. Insert :**

Insert into tablename [(column1, ... , column n)] values ( value1, .... , value n);

**3. Update:**

update tablename set column1= new value[,column2 = new value,...][where condition];

**4. Delete:**

delete [from] tablename [where condition];

Consider the following schema

employee(emp\_id, name, salary, designation)

Execute the following queries:

- 1) Insert at least 10 records with meaningful data.
- 2) Find the name of the employee along with their id.
- 3) Find name of the employees whose salary is >10,000.
- 4) Find name of the employees whose salary is <15,000.
- 5) Update designation of employee 11 to 'Asst.Prof.'
- 6) Delete the employees having designation supervisor.
- 7) Increment the salary of employees by 5%.
- 8) Find the name of employees having salary between 18000 and 22000.

**Outcome:** Students are able to manipulate the database using DML statements.



## Experiment No. 5

**Title:** SQL character functions, String functions & Pattern matching.

**Objective:** To understand the use of standard SQL functions for processing the text data.

**Theory:**

**Character functions** are of the following two types:

1. Case-Manipulative Functions (LOWER, UPPER and INITCAP)
2. Character-Manipulative Functions (CONCAT, LENGTH, SUBSTR, INSTR, LPAD, RPAD, TRIM and REPLACE)

| Name      | Description   |
|-----------|---|
| CONCAT    | Concatenate two or more strings into a single string                      |
| INSTR     | Return the position of the first occurrence of a substring in a string    |
| LENGTH    | Get the length of a string in bytes and in characters                     |
| LEFT      | Get a specified number of leftmost characters from a string               |
| LOWER     | Convert a string to lowercase   |
| LTRIM     | Remove all leading spaces from a string                                   |
| REPLACE   | Search and replace a substring in a string                                |
| RIGHT     | Get a specified number of rightmost characters from a string              |
| RTRIM     | Remove all trailing spaces from a string                                  |
| SUBSTRING | Extract a substring starting from a position with a specific length.      |
| TRIM      | Remove unwanted characters from a string.                                 |
| FORMAT    | Format a number with a specific locale, rounded to the number of decimals |
| UPPER     | Convert a string to uppercase   |

**Like operator:**

LIKE operator is used in a WHERE clause to search for a specified pattern in a column.

There are two wildcards often used in conjunction with the LIKE operator:

1. % - The percent sign represents zero, one, or multiple characters
2. \_ - The underscore represents a single character

```
SELECT column1, column2, ...
```

```
FROM table_name
```

```
WHERE columnN LIKE pattern;
```

Some examples showing different LIKE operators with '%' and '\_' wildcards:

| LIKE Operator                  | Description  |
|--------------------------------|--|
| WHERE CustomerName LIKE 'a%'   | Finds any values that start with "a"   |
| WHERE CustomerName LIKE '%a'   | Finds any values that end with "a"   |
| WHERE CustomerName LIKE '%or%' | Finds any values that have "or" in any position                              |
| WHERE CustomerName LIKE '_r%'  | Finds any values that have "r" in the second position                        |
| WHERE CustomerName LIKE 'a__%' | Finds any values that start with "a" and are at least 3 characters in length |
| WHERE ContactName LIKE 'a%o'   | Finds any values that start with "a" and ends with "o"                       |

Consider the following schema.

customer (customer-name, customer-street, customer-city)

Execute the following queries:

1. Find the names of customers whose names starts with 's' letter and contain 5 letters.
2. Find the customers whose contains substring 'pur'
3. Find the customers whose street address contains substring 'main'.
4. Display first character of customer-name as capital.
5. Display customer cities having 6 letters.

**Outcome:** Students are able to use standard SQL character functions, String functions

## Experiment No. 6

**Title:** Aggregate functions and Group by, having, between, Order by clauses

**Objective:** To understand the use of Aggregate functions and various clauses like group by, order by, having, between clause, etc.

### **Theory:**

1. **Aggregate functions:** sum, min, max, avg, count.

Syntax: select groupfunction (column)  
from tablename  
[where condition(s)]

2. **Group by clause:**

select column , groupfunction (column)  
from tablename  
[where condition(s)]  
[group by column/exp]

3. **Having clause:** To apply condition on group, having clause is used.

select column , groupfunction (column)  
from tablename  
[where condition(s)]  
[group by column/exp]  
[having groupcondition]

4. **Between clause:** The BETWEEN operator selects values within a given range. The values can be numbers, text, or dates. The BETWEEN operator is inclusive: begin and end values are included.

Syntax:   SELECT column\_name(s)  
          FROM table\_name  
          WHERE column\_name BETWEEN value1 AND value2;

5. **Order by clause:**

select columnlist  
from tablename  
[where condition(s)]  
[ order by column/exp[asc/desc]];

Consider the following schema.

account ( account-number, branch-name, balance )

Execute the following queries:

1. Find the balance of account numbers in between '111' and '120'.
  2. List entire account relation in ascending order by loan number.
  3. Find the total account balance at each branch.
  4. Find those branches' having total balance is greater than 20000.
  5. Find the average, minimum, maximum, total account balance.
  6. Find the average, minimum, maximum, total account balance at each branch.
  7. Find the number of depositors at each branch.
  8. Find those branches where average account balance is more than 1200.
- Aggregate functions are used to create summaries of the data in the relation.

**Outcome:** Students will be able to use aggregate functions and above clauses in their mini projects & day-to-day problems solving.



## Experiment No. 7

**Title:-** Join operations and set operations.

**Objective:** To study database joins and set operations.

**Theory:**

### 1. Joins:

| Join types       | Join condition        |
|------------------|-----------------------|
| Inner join       | Natural               |
| Left outer join  | On <predicate>        |
| Right outer join | Using (A1, .... , An) |
| Full outer join  |                       |

- JOINS allow us to combine data from more than one table into a single result set.
- JOINS have better performance compared to sub queries
- INNER JOINS only return rows that meet the given criteria.
- OUTER JOINS can also return rows where no matches have been found. The unmatched rows are returned with the NULL keyword.
- The frequently used clause in JOIN operations is "ON". "USING" clause requires that matching columns be of the same name.
- JOINS can also be used in other clauses such as GROUP BY, WHERE, SUB QUERIES, AGGREGATE FUNCTIONS etc.

#### **Inner Join:**

```
SELECT columns  
FROM tableA  
INNER JOIN tableB  
ON tableA.column = tableB.column;
```

#### **Natural join**

```
SELECT columns  
FROM tableA  
NATURAL JOIN tableB
```

## 2. Set operations: union, intersect, minus

### Syntax:

```
select query1  
set operator  
select query2;
```

Consider the following schema.

borrower (customer-name, loan-number)  
account (account-number, branch-name, balance )  
depositor (customer-name, account-number)

Execute the following queries:

1. Find all customers having either loan, an account or both at the bank.
2. Find all customers having an account but not loan at the bank.
3. Find all customers having both an account and loan at the bank.
4. Find the customers who have balance more than 10000.
5. List in alphabetic order, customers who have account at 'Shahupuri' branch.
6. Find all customers who have either an account or loan (but not both) at the bank.

- MySQL supports union and intersect operation.

**Outcome:** Students are able to perform joins and set operations on various relational tables.

## Experiment No. 8

**Title:** Views, Constraints and Subqueries.

**Objective:** To study the implementation of view and different operations on it. Also to study Constraints and Subqueries.

**Theory:**

**View:** Syntax: create view viewname

As

< query expression >

e.g. create view marks60 as select rollno, marks from student where marks >60;

- **Constraints:** primary key, foreign key, not null, check, unique clause, on delete cascade.

Creating table using all constraints:

Syntax: create table tablename

( A1D1, .... , AnDn, <integrity\_constraint1>..... , <integrity\_constraint n>);

Where A1,...., An are attributes, D1,..... , Dn are datatypes.

- **Subqueries:**

**a. Single row subquery:**

Syntax: select columnlist from tablename where columnname operator

(select columnlist from tablename [where condition]) ;

**b. Multiple row subquery:**

Operators: In, All, any/some

**Examples of constraints:**

**//Primary Key**

create table department

(dept\_name varchar(20) primary key,

building varchar(20), budget numeric(15,2));

insert into department values ('cse', 'new' , 200000);

insert into department values ('etx', 'new' , 250000);

### **//Foreign key**

```
create table instructor
(ID char(5), name varchar(20) not null,
dept_name varchar(20),
salary numeric(8,2),
primary key (ID),
foreign key (dept_name) references department(dept_name) on delete cascade on update
cascade)
```

```
insert into instructor values ('10211', 'Ram', 'cse', 66000);
insert into instructor values ('10212', 'Shourya', 'etx', 60000);
```

```
select * from department;
select * from instructor;
```

### **//To see effect of on delete cascade and on update cascade**

```
update department set dept_name='etc' where dept_name='etx';
select * from department;
select * from instructor;
delete from department where dept_name='etc';
```

```
select * from department;
select * from instructor;
```

```
create table student (
    ID varchar(5), name varchar(20) not null, dept_name varchar(20),
    tot_cred numeric(3,0) DEFAULT 5.5, primary key (ID),
    foreign key (dept_name) references department (dept_name));
```

### **//Setting Default value to column**

```
mysql> ALTER TABLE table_name
    ALTER column_name SET DEFAULT default_value;
```

```
mysql> ALTER TABLE table_name
      ALTER column_name DROP DEFAULT;
```

```
mysql> create table s1 (rn int, class char(4) default 'TY', marks float(4,2));
Query OK, 0 rows affected, 1 warning (1.26 sec)
```

```
mysql> insert into s1(rn, marks) values (1,78);
Query OK, 1 row affected (0.11 sec)
```

```
mysql> select * from s1;
+-----+-----+-----+
| rn  | class | marks |
+-----+-----+-----+
|  1 | TY   | 78.00 |
+-----+-----+-----+
1 row in set (0.00 sec)
```

### //Check Constraint

```
create table account1 (accno int, brnm varchar(10), balance numeric(6,1),
check(balance>20000));
```

```
insert into account1 values (11,"shahupuri", 6000);
```

ERROR 3819 (HY000): Check constraint 'account1\_chk\_1' is violated.

### Queries to Solve:

Consider the following schema.

```
loan (loan-number, branch-name, amount)
borrower (customer-name, loan-number)
account (account-number, branch-name, balance )
depositor (customer-name, account-number)
```

Execute the following queries:

1. Create a view 'all-customers' consisting of branch names and the names of customers who have either an account or a loan or both.

2. Using view 'all-customers', list in alphabetic order, the customers of 'downtown' branch.
3. Create a view consisting of sum of the amounts of all the loans for each branch.

Consider the following schema.

branch (branch-name, branch-city, assets)

borrower (customer-name, loan-number)

account (account-number, branch-name, balance )

depositor (customer-name, account-number)

Execute the following queries:

1. Create above tables using all constraints- primary key, foreign key, not null, check
2. Find the customers having same balance as 'Ram'
3. Update balance of 'Smith' by 5%.
4. By using subquery, find the names of customers who have account but not loan.
5. By using subquery, find the names of customers who have both account and loan.
6. Find the names of all branches that have assets greater than those of at least one branch located in 'Kolhapur'.
7. Update those accounts whose balance is greater than average balance by 5%.
8. Find the customers who have a loan at bank & whose names are neither 'Smith' nor 'Jones'.
9. Delete all account tuples at every branch located in 'Kolhapur' city.

**Outcome:** Students are able to create views. Also Students are able to create the tables using primary key, foreign key, not null, check constraints.

## Experiment No. 9

**Title:** PLSQL Functions and Procedures

**Objective:** Demonstrate PLSQL Functions and Procedures.

**Theory:**

**Stored Procedure:**

A stored procedure is a named collection of procedural and SQL statements.

There are two advantages to the use of stored procedures:

1. Stored procedures substantially reduce network traffic and increase performance. Because the procedure is stored at the server, there is no transmission of individual SQL statements over the network. The use of stored procedures improves system performance because all transactions are executed locally on the RDBMS, so each SQL statement does not have to travel over the network.
2. Stored procedures help reduce code duplication by means of code isolation and code sharing (creating unique PL/SQL modules that are called by application programs), thereby minimizing the chance of errors and the cost of application development and maintenance.

**Syntax to create a stored procedure(on Oracle database):**

```
CREATE OR REPLACE PROCEDURE procedure_name [(argument [IN/OUT] data-type, ... )]  
    [IS/AS]  
    [variable_name data type[:=initial_value] ]  
BEGIN  
    PL/SQL or SQL statements;  
    ...  
END;
```

1. *argument* specifies the parameters that are passed to the stored procedure. A stored procedure could have zero or more arguments or parameters.
2. IN/OUT indicates whether the parameter is for input, output, or both.
3. *data-type* is one of the procedural SQL data types used in the RDBMS. The data types normally match those used in the RDBMS table-creation statement.
4. Variables can be declared between the keywords IS and BEGIN. You must specify the variable name, its data type, and (optionally) an initial value.

**For e.g. (Procedures on MySQL databases)**

**1. Counting number of tuples in table t**

```
mysql> delimiter //
```

```
mysql> CREATE PROCEDURE simpleproc (OUT param1 INT)
```

```
-> BEGIN
```

```
-> SELECT COUNT(*) INTO param1 FROM t;
```

```
-> END//
```

Query OK, 0 rows affected (0.00 sec)

```
mysql> delimiter ;
```

```
mysql> CALL simpleproc(@a);
```

Query OK, 0 rows affected (0.00 sec)

```
mysql> SELECT @a;
```

|    |
|----|
| @a |
| 3  |

**2. Displaying information of Student table**

```
DELIMITER $$
```

```
drop procedure if exists getstudent $$
```

```
CREATE PROCEDURE getstudent()
```

```
BEGIN
```

```
SELECT * FROM students;
```

```
END$$
```

```
DELIMITER ;
```

**To call this procedure**

```
call getstudent(); or call getstudent() $$;
```

**3. Displaying information of Student having specific roll number.**

```
DELIMITER $$
```

```
drop procedure if exists getstudent1 $$
```

```
CREATE PROCEDURE getstudent1(in rn int)
```

```
BEGIN
```

```
SELECT * FROM students where rollno=rn;
```

```
END$$
```

```
DELIMITER ;
```



To call this procedure : call getstudent1(1) ;

#### **4. Procedure to find given number is prime or not.**

```
DELIMITER $$
DROP PROCEDURE IF EXISTS getprime_number $$
CREATE PROCEDURE getprime_number(in num int)
BEGIN
    DECLARE x INT;
    DECLARE flag INT;
    SET x = 2; SET flag=1;
    ll:
    WHILE x<num DO
    IF ((num mod x)=0) THEN
    SET flag=0;
    LEAVE ll;
    ELSE
    SET x = x + 1;
    END IF;
    END WHILE;
    IF (flag=1) THEN
    select num as Number, "Prime number" as Result;
    ELSE
    select num as Number, "Not Prime number" as Result;
    END IF;
    END$$
DELIMITER ;
```

#### **5. Procedure to find Factorial of given number.**

```
Delimiter //
DROP PROCEDURE IF EXISTS fact//
CREATE PROCEDURE fact(IN x INT)
BEGIN
    DECLARE result INT;
    DECLARE i INT;
    SET result = 1; SET i = 1;
    WHILE i <= x DO
    SET result = result * i;
    SET i = i + 1;
    END WHILE;
```

```
SELECT x AS Number, result as Factorial;
END//
```

**Functions:** function always returns a value back to a calling block.

**Syntax: (Oracle database)**

```
Create [or replace] procedure procedurename
[[parameter 1[, parameter 2, ..... , ]]
return data type
is
[constant / variable declaration]
begin
executable statements
[Exception exception handling statements return returnvalue]
End [functionname];
```

### **1. Function to find Average of given numbers**

```
DELIMITER $$
DROP FUNCTION IF EXISTS AVERAGE1$$
CREATE FUNCTION AVERAGE1(n1 INT, n2 INT, n3 INT, n4 INT)
RETURNS INT

BEGIN
DECLARE avg INT;
SET avg = (n1+n2+n3+n4)/4;
RETURN avg;
END$$
DELIMITER ;
```

To call this function

```
select AVERAGE1(1,2,3,4) ;
or
select AVERAGE1(1,2,3,4) ;
$$;
```

### **2. Function to find greater from three numbers**

```
DELIMITER $$
DROP FUNCTION IF EXISTS greater$$
CREATE FUNCTION greater(n1 INT, n2 INT, n3 INT)
RETURNS INT

BEGIN
DECLARE gr INT;
```

```
if(n1 > n2 && n1>n3) then
SET gr=n1;
else if(n2 > n1 && n2>n3) then
SET gr=n2;
else
SET gr=n3;
end if;
end if;
RETURN gr;
END$$
DELIMITER ;
```

**Conclusion:** Students will able to implement Functions, Procedures using PL/SQL.

## Experiment No.10

**Title:** Cursors, and triggers using PL/SQL

**Objective:** Demonstrate Cursors, and triggers using PL/SQL.

**Theory:**

**Cursor:**

A cursor is a pointer to this context area. PL/SQL controls the context area through a Cursor. A cursor is a temporary work area created in the system memory when a SQL statement is executed. A cursor contains information on a select statement and the rows of data accessed by it. Therefore, cursors are used as to speed the processing time of queries in large databases.

**Cursors can be of two types:**

- **Implicit cursors**
- **Explicit cursors**

Implicit cursors are automatically created by Oracle whenever an SQL statement is executed when there is no explicit cursor defined for the statement.

**Cursor attributes like:**

- **%FOUND,**
- **%ISOPEN,**
- **%NOTFOUND,**
- **%ROWCOUNT.**

**Following image describes these attributes briefly:**

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

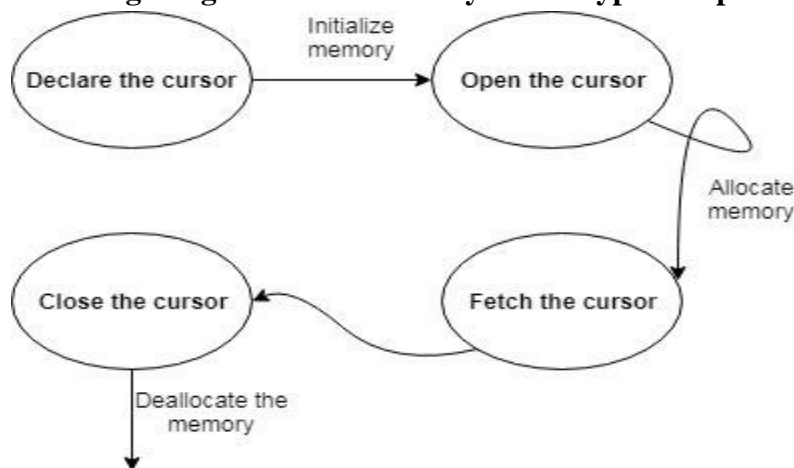
**Write a cursor that will increase salary by 1000 of those whose age is less than 30.**

```
DECLARE
total_rows number(2);
BEGIN
UPDATE Employee
SET salary = salary + 1000
where age < 30;
IF sql%notfound THEN
    dbms_output.put_line('No employees found for under 30 age');
ELSIF sql%found THEN
    total_rows := sql%rowcount;
    dbms_output.put_line( total_rows || ' employees updated ');
END IF;
END;
```

**Explicit cursors:**

**CURSOR cursor\_name IS select\_statement;**

**Following image denotes the life cycle of a typical explicit cursor:**



**e.g. Write a cursor that will increase salary by 10% if salary is > 10000 otherwise by 5%.**

```
declare
e_id employee1.emp_id%type;
e_name employee1.emp_name%type;
e_salary employee1.emp_salary%type;
cursor s1 is
select emp_id,emp_name,emp_salary from employee1;
begin
```

```

open s1;
loop
fetch s1 into e_id,e_name,e_salary;
if(e_salary>100)
then e_salary:=e_salary*1.10;
else
e_salary:=e_salary*1.05;
end if;
update employee1 set emp_salary=e_salary where e_id=emp_id;
exit when s1 % notfound;
end loop;
end;

```

### **Trigger:**

A trigger is a statement that the system executes automatically as a side effect of a modification to the Database.

A trigger is invoked before or after a data row is inserted, updated, or deleted.

1. A trigger is associated with a database table.
2. Each database table may have one or more triggers.
3. A trigger is executed as part of the transaction that triggered it.

Triggers are critical to proper database operation and management. For example:

1. Triggers can be used to enforce constraints that cannot be enforced at the DBMS design and implementation levels.
2. Triggers add functionality by automating critical actions and providing appropriate warnings and suggestions for remedial action. In fact, one of the most common uses for triggers is to facilitate the enforcement of referential integrity.
3. Triggers can be used to update table values, insert records in tables, and call other stored procedures.

Oracle recommends triggers for:

1. Auditing purposes (creating audit logs).
2. Automatic generation of derived column values.
3. Enforcement of business or security constraints.
4. Creation of replica tables for backup purposes.

### Syntax:

```
CREATE OR REPLACE TRIGGER trigger_name
[BEFORE / AFTER] [DELETE / INSERT / UPDATE OF column_name] ON table_name
[FOR EACH ROW]
[DECLARE]
[variable_name data type[:=initial_value] ]
BEGIN
PL/SQL instructions;
.....
END;
```

### A trigger definition contains the following parts

- **The triggering timing:** BEFORE or AFTER. This timing indicates when the trigger's PL/SQL code executes; in this case, before or after the triggering statement is completed.
- **The triggering event:** the statement that causes the trigger to execute (INSERT, UPDATE, or DELETE).
- **The triggering level:** There are two types of triggers: statement-level triggers and row-level triggers.
  - **A statement-level trigger** is assumed if you omit the FOR EACH ROW keywords. This type of trigger is executed once, before or after the triggering statement is completed. This is the default case.
  - **A row-level trigger** requires use of the FOR EACH ROW keywords. This type of trigger is executed once for each row affected by the triggering statement. (In other words, if you update 10 rows, the trigger executes 10 times.)
- **The triggering action:** The PL/SQL code enclosed between the BEGIN and END keywords. Each statement inside the PL/SQL code must end with a semicolon “;”.

e.g. **mysql> CREATE TABLE account (acct\_num INT, amount DECIMAL(10,2));**

**Query OK, 0 rows affected (0.03 sec)**

**mysql> CREATE TRIGGER ins\_sum BEFORE INSERT ON account  
FOR EACH ROW SET @sum = @sum + NEW.amount;**

**Query OK, 0 rows affected (0.01 sec)**

```

mysql> SET @sum = 0;
mysql> INSERT INTO account VALUES(137,14.98),(141,1937.50),(97,-100.00);
mysql> SELECT @sum AS 'Total amount inserted';
+-----+
| Total amount inserted |
+-----+
|          1852.48      |
+-----+

mysql> delimiter //
mysql> CREATE TRIGGER upd_check BEFORE UPDATE ON account
-> FOR EACH ROW
-> BEGIN
->   IF NEW.amount < 0 THEN
->     SET NEW.amount = 0;
->   ELSEIF NEW.amount > 100 THEN
->     SET NEW.amount = 100;
->   END IF;
-> END;//
mysql> delimiter ;

```

#### **Example of Before insert trigger**

```

CREATE TABLE people (age INT, name varchar(150));
delimiter //
CREATE TRIGGER agecheck BEFORE INSERT ON people FOR EACH ROW IF NEW.age < 0
THEN SET NEW.age = 0; END IF;//
delimiter ;
INSERT INTO people VALUES (-20, 'aaa'), (30, 'bbb');
select * from people;

```

**Conclusion:** Students will be able to use Cursors, and triggers using PL/SQL.



## Experiment No.11

**Title:** Database Connectivity-To connect frontend with the backend

**Objective:** Write a program to perform Database connectivity using JDB- ODBC components & MySQL database on Ubuntu

### **Procedure:**

#### **Setting up MySQL or JDBC driver on Ubuntu:**

- sudo apt-get install libmysql-java
- export CLASSPATH=\$CLASSPATH:/usr/share/java/mysql-connector-java.jar

#### **Simple Example JDBC-MySQL Connection**

```
import java.sql.*;

class testdb{

public static void main(String args[]){

int rn=0;

String name;

try{

Class.forName("com.mysql.jdbc.Driver");

Connection con=DriverManager.getConnection(

"jdbc:mysql://localhost/stud","root","root");

//here stud is database name, first root is username and second root is password

Statement stmt=con.createStatement();

ResultSet rs=stmt.executeQuery("select * from student");

while(rs.next())

{   rn= rs.getInt(1);

    name= rs.getString(2);

    System.out.println("Rollno:"+rn+ "\t" + "Name:"+nm);

}

con.close();

}catch(Exception e){ System.out.println(e);}

} }
```

**Steps:**

1. Open Terminal
2. Open MySQL prompt using

```
mysql -u root -p
```

Give password
3. create database stud;
4. use database stud;
5. create table student(rollno int, name varchar(20));
6. insert into student values(1, "Ram");  
Insert into student values (2, "Krishna");
7. Open another Terminal
8. Install MySQL Library using following command

```
sudo apt-get install libmysql-java
```
9. Set Path to MySQL Library

```
export CLASSPATH=$CLASSPATH:/usr/share/java/mysql-connector-java.jar
```
10. Compile java program

```
javac testdb.java
```
11. Execute java program

```
java testdb
```

**Conclusion:** With database connectivity standards like JDBC, it is possible to connect front-end (java) with the backend (MySQL database)

**Outcome:** Students are able to create application which will access database from frontend (java)

## Experiment No. 12

**Title:** Normalization

**Objective:** Normalize any database from first normal form to Boyce-Codd Normal Form (BCNF).

**Theory:**

**Normalization:**

- Normalization is the process of efficiently organizing data in a database.
- Normalization of data can be considered as a process of analyzing the given relation schemas based on their FDs and primary keys to achieve the desirable properties of
  - (1) minimizing redundancy and
  - (2) minimizing the insertion, deletion, and update anomalies

**Normal forms:**

**1. First normal form (1NF):**

- Disallows **multivalued** attributes, **composite** attributes and their combinations.
- Only attribute values permitted by 1NF are single atomic (or indivisible) values.

**2. Second normal form (2NF):**

A relation schema R is in **2NF** if every nonprime attribute A in R is **fully functionally** dependent on the **primary key** of R.

**3. Boyce Codd Normal Form (BCNF):** Relation schema R is in BCNF w.r.t. set F if all functional dependencies in  $F^+$  of the form  $\alpha \rightarrow \beta$  where  $\alpha, \beta$  are subsets of R, at least one of the following holds:

- a.  $\alpha \rightarrow \beta$  is a trivial functional dependency.
- b.  $\alpha$  is a superkey for schema R.

**4. Third Normal Form (3NF):** Relation schema R is in 3NF w.r.t. set F if all functional dependencies in  $F^+$  of the form  $\alpha \rightarrow \beta$  where  $\alpha, \beta$  are subsets of R, at least one of the following holds:

- a.  $\alpha \rightarrow \beta$  is a trivial functional dependency.
- b.  $\alpha$  is a superkey for schema R.
- c. Each attribute A in  $\beta - \alpha$  is contained in a candidate key for R.

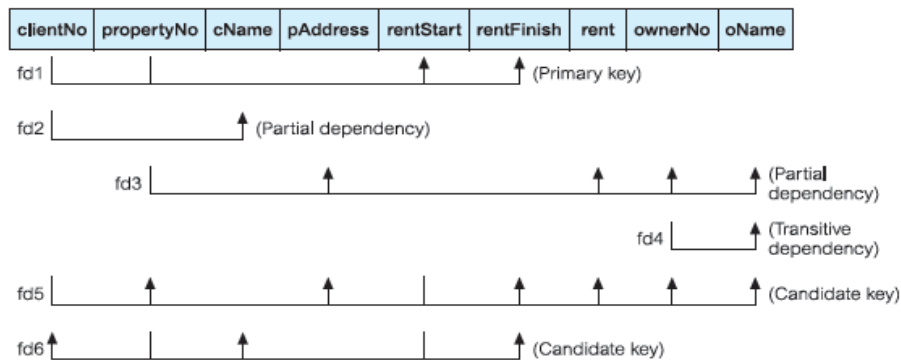
A relation that is in first and second normal form and in which no non-candidate key attribute is transitively dependent on any candidate key that relation is in 3NF.

### Example: Normalization of ClientRental database from first normal form to 3NF

**Figure 13.10**  
ClientRental  
unnormalized table.

| clientNo | cName         | propertyNo | pAddress               | rentStart | rentFinish | rent | ownerNo | oName       |
|----------|---------------|------------|------------------------|-----------|------------|------|---------|-------------|
| CR76     | John Kay      | PG4        | 6 Lawrence St, Glasgow | 1-Jul-03  | 31-Aug-04  | 350  | CO40    | Tina Murphy |
|          |               | PG16       | 5 Novar Dr, Glasgow    | 1-Sep-04  | 1-Sep-05   | 450  | CO93    | Tony Shaw   |
| CR56     | Aline Stewart | PG4        | 6 Lawrence St, Glasgow | 1-Sep-02  | 10-June-03 | 350  | CO40    | Tina Murphy |
|          |               | PG36       | 2 Manor Rd, Glasgow    | 10-Oct-03 | 1-Dec-04   | 375  | CO93    | Tony Shaw   |
|          |               | PG16       | 5 Novar Dr, Glasgow    | 1-Nov-05  | 10-Aug-06  | 450  | CO93    | Tony Shaw   |

ClientRental



**Figure 13.12**  
Functional  
dependencies of the  
ClientRental relation.

**Figure 13.13**  
Alternative 1NF  
Client and  
PropertyRentalOwner  
relations.

Client

| clientNo | cName         |
|----------|---------------|
| CR76     | John Kay      |
| CR56     | Aline Stewart |

PropertyRentalOwner

| clientNo | propertyNo | pAddress               | rentStart | rentFinish | rent | ownerNo | oName       |
|----------|------------|------------------------|-----------|------------|------|---------|-------------|
| CR76     | PG4        | 6 Lawrence St, Glasgow | 1-Jul-03  | 31-Aug-04  | 350  | CO40    | Tina Murphy |
| CR76     | PG16       | 5 Novar Dr, Glasgow    | 1-Sep-04  | 1-Sep-05   | 450  | CO93    | Tony Shaw   |
| CR56     | PG4        | 6 Lawrence St, Glasgow | 1-Sep-02  | 10-Jun-03  | 350  | CO40    | Tina Murphy |
| CR56     | PG36       | 2 Manor Rd, Glasgow    | 10-Oct-03 | 1-Dec-04   | 375  | CO93    | Tony Shaw   |
| CR56     | PG16       | 5 Novar Dr, Glasgow    | 1-Nov-05  | 10-Aug-06  | 450  | CO93    | Tony Shaw   |

Client

(clientNo, cName)

PropertyRentalOwner

(clientNo, propertyNo, pAddress, rentStart, rentFinish, rent, ownerNo, oName)

**Figure 13.14**

Second Normal Form relations derived from the ClientRental relation.

**Client**

| clientNo | cName         |
|----------|---------------|
| CR76     | John Kay      |
| CR56     | Aline Stewart |

**Rental**

| clientNo | propertyNo | rentStart | rentFinish |
|----------|------------|-----------|------------|
| CR76     | PG4        | 1-Jul-03  | 31-Aug-04  |
| CR76     | PG16       | 1-Sep-04  | 1-Sep-05   |
| CR56     | PG4        | 1-Sep-02  | 10-Jun-03  |
| CR56     | PG36       | 10-Oct-03 | 1-Dec-04   |
| CR56     | PG16       | 1-Nov-05  | 10-Aug-06  |

**PropertyOwner**

| propertyNo | pAddress               | rent | ownerNo | oName       |
|------------|------------------------|------|---------|-------------|
| PG4        | 6 Lawrence St, Glasgow | 350  | CO40    | Tina Murphy |
| PG16       | 5 Novar Dr, Glasgow    | 450  | CO93    | Tony Shaw   |
| PG36       | 2 Manor Rd, Glasgow    | 375  | CO93    | Tony Shaw   |

The relations have the following form:

Client                    (clientNo, cName)  
Rental                    (clientNo, propertyNo, rentStart, rentFinish)  
PropertyOwner        (propertyNo, pAddress, rent, ownerNo, oName)

### Example 13.11 Third Normal Form (3NF)

The functional dependencies for the Client, Rental, and PropertyOwner relations, derived in Example 13.10, are as follows:

Client

fd2    clientNo → cName                    (Primary key)

Rental

fd1    clientNo, propertyNo → rentStart, rentFinish                    (Primary key)

fd5'    clientNo, rentStart → propertyNo, rentFinish                    (Candidate key)

fd6'    propertyNo, rentStart → clientNo, rentFinish                    (Candidate key)

PropertyOwner

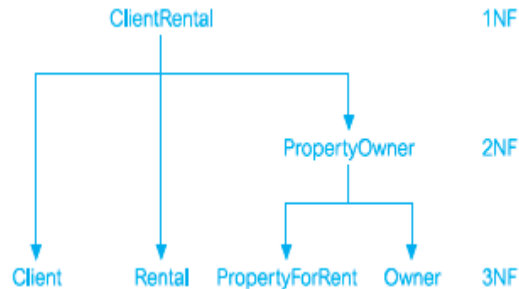
fd3    propertyNo → pAddress, rent, ownerNo, oName                    (Primary key)

fd4    ownerNo → oName                    (Transitive dependency)

**Figure 13.15**  
Third Normal Form  
relations derived  
from the  
PropertyOwner  
relation.

| PropertyForRent |                        |      |         | Owner   |             |
|-----------------|------------------------|------|---------|---------|-------------|
| propertyNo      | pAddress               | rent | ownerNo | ownerNo | oName       |
| PG4             | 6 Lawrence St, Glasgow | 350  | CO40    | CO40    | Tina Murphy |
| PG16            | 5 Novar Dr, Glasgow    | 450  | CO93    | CO93    | Tony Shaw   |
| PG36            | 2 Manor Rd, Glasgow    | 375  | CO93    |         |             |

**Figure 13.16**  
The decomposition  
of the ClientRental  
1NF relation into  
3NF relations.



original 1NF relation is decomposed into the 3NF relations. The resulting 3NF relations have the form:

|                 |  |
|-----------------|--|
| Client          | ( <u>clientNo</u> , cName)                                     |
| Rental          | ( <u>clientNo</u> , <u>propertyNo</u> , rentStart, rentFinish) |
| PropertyForRent | ( <u>propertyNo</u> , pAddress, rent, ownerNo)                 |
| Owner           | ( <u>ownerNo</u> , oName)                                      |

| Client   |               | Rental   |            |           |            |
|----------|---------------|----------|------------|-----------|------------|
| clientNo | cName         | clientNo | propertyNo | rentStart | rentFinish |
| CR76     | John Kay      | CR76     | PG4        | 1-Jul-03  | 31-Aug-04  |
| CR56     | Aline Stewart | CR76     | PG16       | 1-Sep-04  | 1-Sep-05   |
|          |               | CR56     | PG4        | 1-Sep-02  | 10-Jun-03  |
|          |               | CR56     | PG36       | 10-Oct-03 | 1-Dec-04   |
|          |               | CR56     | PG16       | 1-Nov-05  | 10-Aug-06  |

| PropertyForRent |                        |      |         | Owner   |             |
|-----------------|------------------------|------|---------|---------|-------------|
| propertyNo      | pAddress               | rent | ownerNo | ownerNo | oName       |
| PG4             | 6 Lawrence St, Glasgow | 350  | CO40    | CO40    | Tina Murphy |
| PG16            | 5 Novar Dr, Glasgow    | 450  | CO93    | CO93    | Tony Shaw   |
| PG36            | 2 Manor Rd, Glasgow    | 375  | CO93    |         |             |

**Figure 13.17**  
A summary of the  
3NF relations  
derived from the  
ClientRental relation.

**Outcome:** Students are able to normalize any database from 1NF to BCNF.