Experiment-I

Aim: Create a database schema and ERD for any application.

COLLEGE DATABASE:

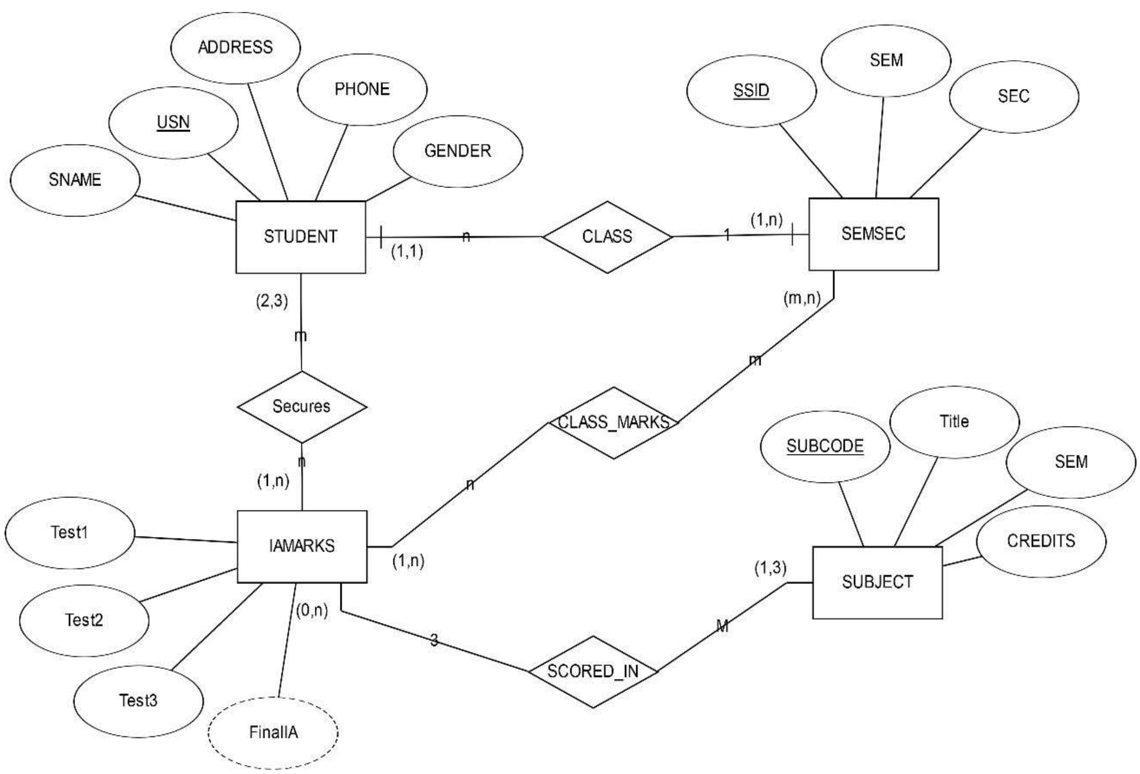
STUDENT (USN, SName, Address, Phone, Gender) SEMSEC (SSID, Sem, Sec)

CLASS (USN, SSID)

SUBJECT (Subcode, Title, Sem, Credits)

IAMARKS (USN, Subcode, SSID, Test1, Test2, Test3, FinalIA)

Entity Relationship Diagram for the College Database is:



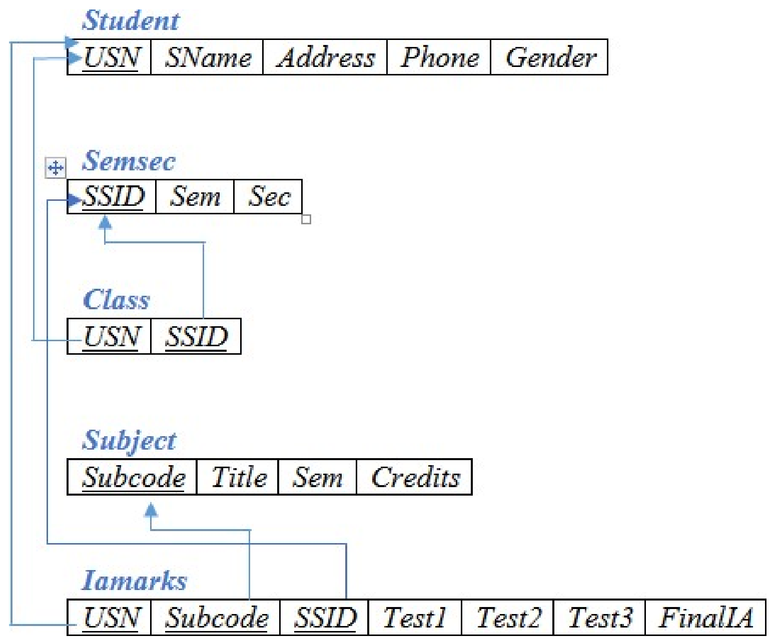


Fig 1.1: Mapping entities and relationships to relation table (Schema Diagram)

Conclusion:

Experiment-II

Aim: Creating tables, Renaming tables, Data constraints (Primary key, Foreign key, Not Null), Data Insertion into a table.

#SQL supports a number of different integrity constraints. In this section, we discuss only a few of them:

primary key (Aj1, Aj2, . . . , Ajm): The primary-key specification says that attributes Aj1, Aj2, . . . , Ajm form the primary key for the relation. The primarykey attributes are required to be nonnull and unique.

foreign key (Ak1, Ak2, . . . , Akn) references s: The foreign key specification says that the values of attributes (Ak1, Ak2, . . . , Akn) for any tuple in the relation must correspond to values of the primary key attributes of some tuple in relation s.

not null: The not null constraint on an attribute specifies that the null value is not allowed for that attribute; in other words, the constraint excludes the null value from the domain of that attribute.

#Create University_Database with following relations

We define an SQL relation by using the create table command.

department (dept_name, building, budget)
instructor(ID, name, dept_name, salary)
course(course_id, title, dept_name, credits)
section (course_id, sec_id, semester, year, building, room_number, time_slot_id)
teaches (ID, course_id, sec_id, semester, year)
student (ID, name, dept_name, tot_cred)
takes (ID, course_id, sec_id, semester, year, grade)

#Queries to create tables are as follows:

```
create table department (dept_name varchar (20), Building varchar (15),
          Budget numeric (12,2), primary key (dept_name));

create table course (course_id varchar (7), title varchar (50),
          dept_name varchar (20), credits numeric (2,0),
          primary key (course_id),
          foreign key (dept_name) references department(dept_name));

create table instructor (ID varchar (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));

create table section(course_id varchar (8), sec_id varchar (8),
          semester varchar (6), year numeric (4,0), building varchar (15),
          room_number varchar (7), time_slot_id varchar (4),
          primary key (course_id, sec_id, semester, year),
          foreign key (course_id) references course(course_id));

create table teaches (ID varchar (5), course_id varchar (8), sec_id
          varchar (8), semester varchar (6), year numeric (4,0),
          primary key (ID, course_id, sec_id, semester, year),
          foreign key (course_id, sec_id, semester, year) references
          section (course_id, sec_id, semester, year),
          foreign key (ID) references instructor(ID));

create table student (ID varchar (5), name varchar (20) not null,
          dept_name varchar (20), tot_cred int, primary key (ID),
          foreign key (dept_name) references department(dept_name));

create table takes (ID varchar (5), course_id varchar (8),
          sec_id varchar (8), semester varchar (6),
          year numeric (4,0),  grade varchar (4),
          foreign key (course_id, sec_id, semester, year) references
```

section(course_id, sec_id, semester, year),
foreign key (ID) references student(ID));

#Insert the records into table as given below:

We can use the insert command to load data into the relation

Example: *Insert into department (dept_name, building, budget)*
values ('Biology','building','budget');

dept_name	*building*	*budget*
Biology	Watson	90000
Comp. Sci.	Taylor	100000
Elec. Eng.	Taylor	85000
Finance	Painter	120000
History	Painter	50000
Music	Packard	80000
Physics	Watson	70000

Fig 2.1: The department relation.

ID	*name*	*dept_name*	*salary*
10101	Srinivasan	Comp. Sci.	65000
12121	Wu	Finance	90000
15151	Mozart	Music	40000
22222	Einstein	Physics	95000
32343	El Said	History	60000
33456	Gold	Physics	87000
45565	Katz	Comp. Sci.	75000
58583	Califieri	History	62000
76543	Singh	Finance	80000
76766	Crick	Biology	72000
83821	Brandt	Comp. Sci.	92000
98345	Kim	Elec. Eng.	80000

Figure 2.2 The instructor relation.

course_id	*title*	*dept_name*	*credits*
BIO-101	Intro. to Biology	Biology	4
BIO-301	Genetics	Biology	4
BIO-399	Computational Biology	Biology	3
CS-101	Intro. to Computer Science	Comp. Sci.	4
CS-190	Game Design	Comp. Sci.	4
CS-315	Robotics	Comp. Sci.	3
CS-319	Image Processing	Comp. Sci.	3
CS-347	Database System Concepts	Comp. Sci.	3
EE-181	Intro. to Digital Systems	Elec. Eng.	3
FIN-201	Investment Banking	Finance	3
HIS-351	World History	History	3
MU-199	Music Video Production	Music	3
PHY-101	Physical Principles	Physics	4

Figure 2.3 : The course relation.

course_id	*sec_id*	*semester*	*year*	*building*	*room_number*	*time_slot_id*
BIO-101	1	Summer	2009	Painter	514	B
BIO-301	1	Summer	2010	Painter	514	A
CS-101	1	Fall	2009	Packard	101	H
CS-101	1	Spring	2010	Packard	101	F
CS-190	1	Spring	2009	Taylor	3128	E
CS-190	2	Spring	2009	Taylor	3128	A
CS-315	1	Spring	2010	Watson	120	D
CS-319	1	Spring	2010	Watson	100	B
CS-319	2	Spring	2010	Taylor	3128	C
CS-347	1	Fall	2009	Taylor	3128	A
EE-181	1	Spring	2009	Taylor	3128	C
FIN-201	1	Spring	2010	Packard	101	B
HIS-351	1	Spring	2010	Painter	514	C
MU-199	1	Spring	2010	Packard	101	D
PHY-101	1	Fall	2009	Watson	100	A

Figure 2.4 : The section relation.

ID	course_id	sec_id	semester	year
10101	CS-101	1	Fall	2009
10101	CS-315	1	Spring	2010
10101	CS-347	1	Fall	2009
12121	FIN-201	1	Spring	2010
15151	MU-199	1	Spring	2010
22222	PHY-101	1	Fall	2009
32343	HIS-351	1	Spring	2010
45565	CS-101	1	Spring	2010
45565	CS-319	1	Spring	2010
76766	BIO-101	1	Summer	2009
76766	BIO-301	1	Summer	2010
83821	CS-190	1	Spring	2009
83821	CS-190	2	Spring	2009
83821	CS-319	2	Spring	2010
98345	EE-181	1	Spring	2009

Figure 2.5 :The teaches relation.

ID	name	dept_name	tot_cred
00128	Zhang	Comp. Sci.	102
12345	Shankar	Comp. Sci.	32
19991	Brandt	History	80
23121	Chavez	Finance	110
44553	Peltier	Physics	56
45678	Levy	Physics	46
54321	Williams	Comp. Sci.	54
55739	Sanchez	Music	38
70557	Snow	Physics	0
76543	Brown	Comp. Sci.	58
76653	Aoi	Elec. Eng.	60
98765	Bourikas	Elec. Eng.	98
98988	Tanaka	Biology	120

Figure 2.6 The student relation.

ID	course_id	sec_id	semester	year	grade
00128	CS-101	1	Fall	2009	A
00128	CS-347	1	Fall	2009	A-
12345	CS-101	1	Fall	2009	C
12345	CS-190	2	Spring	2009	A
12345	CS-315	1	Spring	2010	A
12345	CS-347	1	Fall	2009	A
19991	HIS-351	1	Spring	2010	B
23121	FIN-201	1	Spring	2010	C+
44553	PHY-101	1	Fall	2009	B-
45678	CS-101	1	Fall	2009	F
45678	CS-101	1	Spring	2010	B+
45678	CS-319	1	Spring	2010	B
54321	CS-101	1	Fall	2009	A-
54321	CS-190	2	Spring	2009	B+
55739	MU-199	1	Spring	2010	A-
76543	CS-101	1	Fall	2009	A
76543	CS-319	2	Spring	2010	A
76653	EE-181	1	Spring	2009	C
98765	CS-101	1	Fall	2009	C-
98765	CS-315	1	Spring	2010	B
98988	BIO-101	1	Summer	2009	A
98988	BIO-301	1	Summer	2010	*null*

Figure 2.7 The takes relation.

#We can use the delete command to delete tuples from a relation

delete from teaches;

#To remove a relation from an SQL database, we use the drop table command.

drop table r;

#We use the alter table command to add attributes to an existing relation.

alter table r add A D;

#We can drop attributes from a relation by the command

alter table r drop A;

#Queries on a Single Relation

- Let us consider a simple query using our university example, “Find the names of all instructors.”
  select name from instructor;

Output:

- In those cases where we want to force the elimination of duplicates, we insert the keyword distinct after select. We can write the query as:
  select distinct dept_name
  from instructor;

Output:

- SQL allows us to use the keyword all to specify explicitly that duplicates are not removed:
  select all dept_name
  from instructor;

Output:

- The select clause may also contain arithmetic expressions involving the operators +, −, ∗, and / operating on constants or attributes of tuples. For example, the query:
  *select ID, name, dept_name, salary * 1.1*
  from instructor;

Output:

- Consider the query “Find the names of all instructors in the Computer Science department who have salary greater than $70,000.” This query can be written in SQL as:
  select name
  from instructor
  where dept_name = ’Comp. Sci.’ and salary > 70000;

Output:

#SQL provides a way of renaming the attributes of a result relation. It uses the as clause, taking the form:

old-name as new-name

For example, if we want the attribute name *name* to be replaced with the name Instructor_name, we can write the query as:

select name as instructor_name, course_id from instructor, teaches
where instructor.ID= teaches.ID;

Output:

#The *as* clause is particularly useful in renaming relations.

To illustrate, we rewrite the query "For all instructors in the university who have taught some course, find their names and the course ID of all courses they taught."

select T.name, S.course_id
from instructor as T, teaches as S
where T.ID= S.ID;

Output:

Conclusion:

Experiment-III

***Aim:* Execution of SET operations, aggregate functions**

#The SQL operations union, intersect, and except operate on relations and correspond to the mathematical set-theory operations ∪, ∩, and −.

#The Union Operation

- To find the set of all courses taught either in Fall 2009 or in Spring 2010, or both, we write:

  (select course_id from section
  where semester = 'Fall' and year= 2009)
  union
  (select course_id from section
  where semester = 'Spring' and year= 2010);

Output:

- The union operation automatically eliminates duplicates, unlike the select clause. If we want to retain all duplicates, we must write union all in place of union:

  (select course_id from section
  where semester = 'Fall' and year= 2009)
  union all
  (select course_id from section
  where semester = 'Spring' and year= 2010);

Output:

#The Intersect Operation

- To find the set of all courses taught in the Fall 2009 as well as in Spring 2010 we write:

  (select course_id from section
  where semester = 'Fall' and year= 2009)
  intersect

(select course_id from section
where semester = 'Spring' and year= 2010);

Since MySQL does not provide support for the INTERSECT operator. However, we can use the INNER JOIN and IN clause to emulate this operator. Query using distinct keyword and IN clause can be written as:

select distinct course_id
from section
where semester = 'Fall' and year= 2009 and
*course_id **in** (select course_id*
from section
where semester = 'Spring' and year= 2010);

Output:

- If we want to retain all duplicates, we must write intersect all in place of intersect:

(select course_id from section
where semester = 'Fall' and year= 2009)
intersect all
(select course_id from section
where semester = 'Spring' and year= 2010);

Since MySQL does not provide support for the INTERSECT ALL operator. However, we can use the INNER JOIN and IN clause to emulate this operator. Query using IN clause can be written as:

select course_id
from section
where semester = 'Fall' and year= 2009 and
*course_id **in** (select course_id*
from section
where semester = 'Spring' and year= 2010);

Output:

#The Except Operation

- To find all courses taught in the Fall 2009 semester but not in the Spring 2010 semester, we write:

  (select course_id from section
  where semester = 'Fall' and year= 2009)
  except
  (select course_id from section
  where semester = 'Spring' and year= 2010);

Since MySQL does not provide support for the EXCEPT operator. However, we can use the LEFT JOIN, NOT IN and EXISTS clause to emulate this operator. Query using distinct keyword and NOT IN clause can be written as:

```
select distinct course_id
from section
where semester = 'Fall' and year= 2009 and
course_id  not in (select course_id
                   from section
                   where semester = 'Spring' and year= 2010);
```

Output:

#Aggregate Functions

Aggregate functions are functions that take a collection (a set or multiset) of values as input and return a single value. SQL offers five built-in aggregate functions:

- Average: avg
- Minimum: min
- Maximum: max
- Total: sum
- Count: count

- Consider the query "Find the average salary of instructors in the Computer Science department." We write this query as follows:

  select avg (salary)
  from instructor

where dept_name= 'Comp. Sci.';

Output:

- There are cases where we must eliminate duplicates before computing an aggregate function. If we do want to eliminate duplicates, we use the keyword distinct in the aggregate expression. An example arises in the query "Find the total number of instructors who teach a course in the Spring 2010 semester."

```
select count(distinct ID)
from teaches
where semester = 'Spring' and year = 2010;
```

Output:

- We use the aggregate function count frequently to count the number of tuples in a relation. The notation for this function in SQL is count (*). Thus, to find the number of tuples in the course relation, we write

```
select count(*)
from course;
```

Output:

Conclusion:

Experiment-IV

Aim: On Created database perform Grouping of data

#group by clause.

There are circumstances where we would like to apply the aggregate function not only to a single set of tuples, but also to a group of sets of tuples; we specify this wish in SQL using the group by clause. The attribute or attributes given in the group by clause are used to form groups. Tuples with the same value on all attributes in the group by clause are placed in one group.

- As an illustration, consider the query "Find the average salary in each department." We write this query as follows:

select dept_name, avg (salary) as avg_salary
from instructor
group by dept_name;

Output:

- As another example of aggregation on groups of tuples, consider the query "Find the number of instructors in each department who teach a course in the Spring 2010 semester." Information about which instructors teach which course sections in which semester is available in the teaches relation. However, this information has to be joined with information from the instructor relation to get the department name of each instructor. Thus, we write this query as follows:

select dept_name, count (distinct ID) as instr_count
from instructor natural join teaches
where semester = 'Spring' and year = 2010
group by dept_name;

Output:

#The Having Clause

- At times, it is useful to state a condition that applies to groups rather than to tuples. For example, we might be interested in only those departments where the average salary of the instructors is more than $42,000. This condition does not apply to a single tuple; rather, it applies to each group constructed by the group by clause. To express such a query, we use the having clause of SQL. SQL applies predicates in the having clause after groups have been formed, so aggregate functions may be used. We express this query in SQL as follows:

```
select dept_name, avg (salary) as avg_salary
from instructor
group by dept_name
having avg (salary) > 42000;
```

Output:

- To illustrate the use of both a having clause and a where clause in the same query, we consider the query "For each course section offered in 2009, find the average total credits (tot_cred) of all students enrolled in the section, if the section had at least 2 students."

```
select course_id, semester, year, sec_id, avg (tot_cred)
from takes natural join student
where year = 2009
group by course_id, semester, year, sec_id
having count (ID) >= 2;
```

Output:

#Ordering the Display of Tuples

- SQL offers the user some control over the order in which tuples in a relation are displayed. The order by clause causes the tuples in the result of a query to appear in sorted order. To list in alphabetic order all instructors in the Physics department, we write:

```
select name
from instructor
where dept_name = 'Physics'
order by name;
```

Output:

- Suppose that we wish to list the entire instructor relation in descending order of salary. If several instructors have the same salary, we order them in ascending order by name. We express this query in SQL as follows:

```
select *
from instructor
order by salary desc, name asc;
```

Conclusion:

Experiment-V

Aim: Perform My-SQL Built-In functions (mathematical, character functions)

#SQL Function

MySQL comes bundled with a number of built in functions. Built in functions are simply functions come already implemented in the MySQL server. These functions allow us to perform different types of manipulations on the data. The built in functions can be basically categorized into the following most used categories.

- Strings functions - operate on string data types
- Numeric functions - operate on numeric data types
- Date functions - operate on date data types

Strings Function

Function	Description
`UPPER(str)`	Converts a string to upper-case
`UCASE(str)`	Converts a string to upper-case
`LOWER(str)`	Converts a string to lower-case
`LCASE(str)`	Converts a string to lower-case
`LENGTH(str)`	Returns the length of a string (in bytes)
`CHAR_LENGTH(str)`	Returns the length of a string (in characters)
`REVERSE(str)`	Reverses a string and returns the result
`SPACE(n)`	Returns a string of the specified number of space characters
`REPEAT(str,count)`	Repeats a string as many times as specified
`SUBSTRING(string, start, length)`	Extracts a substring from a string (starting at any position)
`MID(str, startindex,len )`	Extracts a substring from a string (starting at any position)
`LEFT(string, number)`	Extracts a number of characters from a string (starting from left)
`RIGHT(str ,len )`	Returns the rightmost len characters from the string
`SUBSTRING_INDEX( str,delim,count )`	Returns the substring from string before count occurrences of the delimiter delim.
`REPLACE(str, fromstr,tostr )`	Returns the string by replacing fromstr with tostr.
`RPAD( str , len , padstr )`	Returns the string right-padded with the string padstr to a length of len characters.

LPAD(str , len , padstr)	Returns the string str, left-padded with the string padstr to a length of len characters.
LTRIM(str)	Returns the string str with leading space characters removed.
RTRIM(str)	Returns the string str with trailing space characters removed.

#Numeric Function

Function	Description
ABS(number)	Returns the absolute value of a number
AVG(expression)	Returns the average value of an expression
CEIL(number)	Returns the smallest integer value that is >= to a number
COUNT(expression)	Returns the number of records returned by a select query
DEGREES(number)	Converts a value in radians to degrees
EXP(number)	Returns e raised to the power of a specified number
FLOOR(number)	Returns the largest integer value that is <= to a number
GREATEST(arg1, arg2, arg3, ...)	Returns the greatest value of the list of arguments
LEAST(arg1, arg2, arg3, ...)	Returns the smallest value of the list of arguments
MAX(expression)	Returns the maximum value in a set of values
MIN(expression)	Returns the minimum value in a set of values
MOD(x, y)	Returns the remainder of a number divided by another number
PI()	Returns the value of PI
POW(x, y)	Returns the value of a number raised to the power of another number
ROUND(number, decimals)	Rounds a number to a specified number of decimal places
SIGN(number)	Returns the sign of a number
SQRT(number)	Returns the square root of a number
SUM(expression)	Calculates the sum of a set of values
TRUNCATE(number, decimals)	Truncates a number to the specified number of decimal places

Output:

Output of the queries using above string and numeric function are as follows:

Conclusion:

Experiment-VI

Aim: Implementation of Views and Joins in database

natural join only requires values to match on specified attributes. SQL supports another form of join, in which an arbitrary join condition can be specified.

The *on* condition allows a general predicate over the relations being joined. This predicate is written like a *where* clause predicate except for the use of the keyword *on* rather than where. Like the using condition, the on condition appears at the end of the join expression.

#The natural join

The natural join operation operates on two relations and produces a relation as the result. Unlike the Cartesian product of two relations, which concatenates each tuple of the first relation with every tuple of the second, natural join considers only those pairs of tuples with the same value on those attributes that appear in the schemas of both relations.

- Consider the query "For all instructors in the university who have taught some course, find their names and the course ID of all courses they taught", which we wrote earlier as:

  select name, course_id

  from instructor, teaches

  where instructor.ID= teaches.ID;

This query can be written more concisely using the natural-join operation in SQL as:

select name, course_id

from instructor natural join teaches;

Output:

- suppose we wish to answer the query "List the names of instructors along with the the titles of courses that they teach." The query can be written in SQL as follows:

```
select name, title
from instructor natural join teaches, course
where teaches.course_id= course.course_id;
```

Output

- To provide the benefit of natural join while avoiding the danger of equating attributes erroneously, SQL provides a form of the natural join construct that allows you to specify exactly which columns should be equated. This feature is illustrated by the following query:

  select name, title

  from (instructor natural join teaches) join course using (course_id);

Output:

#Outer Joins

The outer join operation works in a manner similar to the join operations, but preserve those tuples that would be lost in a join, by creating tuples in the result containing null values.

There are in fact three forms of outer join:

- The left outer join preserves tuples only in the relation named before (to the left of) the left outer join operation.
- The right outer join preserves tuples only in the relation named after (to the right of) the right outer join operation.
- The full outer join preserves tuples in both relations.

- Suppose we wish to display a list of all students, displaying their ID, and name, dept_name, and tot_cred, along with the courses that they have taken. The following SQL query may appear to retrieve the required information:

  *select **

  from student natural join takes;

Output:

- Unfortunately, the above query does not work quite as intended. Suppose that there is some student who takes no courses. Then the tuple in the student relation for that particular student would not satisfy the condition of a natural join with any tuple in the takes relation, and that student's data would not appear in the result. But if we use left outer join the students data who have not opted for any course will appear in result.

```
select *
from student natural left outer join takes;
```

Output:

- As another example of the use of the outer-join operation, we can write the query "Find all students who have not taken a course" as:

```
select ID
from student natural left outer join takes
where course_id is null;
```

Output:

- As an example of the use of full outer join, consider the following query: "Display a list of all students in the Comp. Sci. department, along with the course sections, if any, that they have taken in Spring 2009; all course sections from Spring 2009 must be displayed, even if no student from the Comp. Sci. department has taken the course section." This query can be written as:

```
select *
from (select *
      from student
      where dept_name= 'Comp. Sci')
natural full outer join
      (select *
      from takes
      where semester = 'Spring' and year = 2009);
```

Output:

- The on clause can be used with outer joins. The following query is identical to the first query we saw using "student natural left outer join takes," except that the attribute ID appears twice in the result.

```
select *
from student left outer join takes on student.ID= takes.ID;
```

Output:

#inner join

To distinguish normal joins from outer joins, normal joins are called inner joins in SQL. A join clause can thus specify inner join instead of outer join to specify that a normal join is to be used. The keyword inner is, however, optional. The default join type, when the join clause is used without the outer prefix is the inner join.

Thus,

```
select *
from student join takes using (ID);
```

is equivalent to:

```
select *
from student inner join takes using (ID);
```

Similarly, natural join is equivalent to natural inner join.

Output:

Conclusion:

Experiment-VII

Aim: Implementation of Sub-queries in MYSQL

#Nested Subqueries

SQL provides a mechanism for nesting subqueries. A subquery is a select-from-where expression that is nested within another query. A common use of subqueries is to perform tests for set membership, make set comparisons, and determine set cardinality, by nesting subqueries in the where clause.

#Set Membership

SQL allows testing tuples for membership in a relation. The in connective tests for set membership, where the set is a collection of values produced by a select clause. The not in connective tests for the absence of set membership.

- Consider the query "Find all the courses taught in the both the Fall 2009 and Spring 2010 semesters."

  We begin by finding all courses taught in Spring 2010, and we write the subquery.

  (select course_id
  from section
  where semester = 'Spring' and year= 2010)

  We then need to find those courses that were taught in the Fall 2009 and that appear in the set of courses obtained in the subquery. We do so by nesting the subquery in the where clause of an outer query. The resulting query is

  select distinct course_id
  from section
  where semester = 'Fall' and year= 2009 and
  course_id in (select course_id from section
  where semester = 'Spring' and year= 2010);

Output:

- We use the not in construct in a way similar to the in construct. For example, to find all the courses taught in the Fall 2009 semester but not in the Spring 2010 semester, we can write:

```
select distinct course_id
from section
where semester = 'Fall' and year= 2009 and
course_id not in (select course_id
                  from section
                  where semester = 'Spring' and year= 2010);
```

Output:

- The in and not in operators can also be used on enumerated sets. The following query selects the names of instructors whose names are neither “Mozart” nor “Einstein”.

```
select distinct name
from instructor
where name not in ('Mozart', 'Einstein');
```

Output:

- In the preceding examples, we tested membership in a one-attribute relation. It is also possible to test for membership in an arbitrary relation in SQL. For example, we can write the query “find the total number of (distinct) students who have taken course sections taught by the instructor with ID 110011” as follows:

```
select count (distinct ID)
from takes
where (course_id, sec_id, semester, year)
in (select course_id, sec_id, semester, year from teaches
where teaches.ID= 10101);
```

Output:

#Set Comparison

- As an example of the ability of a nested subquery to compare sets, consider the query "Find the names of all instructors whose salary is greater than at least one instructor in the Biology department.", we write this query as follows:

  select distinct T.name
  from instructor as T, instructor as S
  where T.salary > S.salary and S.dept_name = 'Biology';

Output:

- SQL does, however, offer an alternative style for writing the preceding query. The phrase "greater than at least one" is represented in SQL by > some. This construct allows us to rewrite the query in a form that resembles closely our formulation of the query in English.

  select name
  from instructor
  where salary > some (select salary from instructor
  where dept_name = 'Biology');

Output:

- Now we modify our query slightly. Let us find the names of all instructors that have a salary value greater than that of each instructor in the Biology department. The construct > all corresponds to the phrase "greater than all." Using this construct, we write the query as follows:

  select name
  from instructor
  where salary > all (select salary from instructor
  where dept_name = 'Biology');

Output:

- As another example of set comparisons, consider the query "Find the departments that have the highest average salary." We begin by writing a query to find all average salaries, and then nest it as a subquery of a larger query that finds those departments for which the average salary is greater than or equal to all average salaries:

```
select dept_name
from instructor
group by dept_name
having avg (salary) >= all (select avg (salary)
                              from instructor
                              group by dept_name);
```

Output:

#Test for Empty Relations

- SQL includes a feature for testing whether a subquery has any tuples in its result. The exists construct returns the value true if the argument subquery is nonempty. Using the exists construct, we can write the query "Find all courses taught in both the Fall 2009 semester and in the Spring 2010 semester" in still another way:

```
select course_id
from section as S
where semester = 'Fall' and year= 2009 and
exists (select * from section as T
        where semester = 'Spring' and year= 2010 and
              S.course_id= T.course_id);
```

Output:

- The above query also illustrates a feature of SQL where a correlation name from an outer query (S in the above query), can be used in a subquery in the where clause. A subquery that uses a correlation name from an outer query is called a correlated subquery.

- To illustrate the not exists operator, consider the query “Find all students who have taken all courses offered in the Biology department.” Using the except construct, we can write the query as follows:

```
select distinct S.ID, S.name
from student as S
where not exists ((select course_id from course
                    where dept_name = 'Biology')
                    except (select T.course_id
                    from takes as T where S.ID = T.ID));
```

Output:

#Test for the Absence of Duplicate Tuples

- SQL includes a boolean function for testing whether a subquery has duplicate tuples in its result. The unique construct9 returns the value true if the argument subquery contains no duplicate tuples. Using the unique construct, we can write the query “Find all courses that were offered at most once in 2009” as follows:

```
select T.course_id
from course as T
where unique (select R.course_id
from section as R
where T.course_id= R.course_id and R.year = 2009);
```

Output:

- We can test for the existence of duplicate tuples in a subquery by using the not unique construct. To illustrate this construct, consider the query “Find all courses that were offered at least twice in 2009” as follows:

```
select T.course_id
from course as T
where not unique (select R.course_id from section as R
                    where T.course_id= R.course_id
                    and R.year = 2009);
```

Output:

#Subqueries in the From Clause

- SQL allows a subquery expression to be used in the from clause. The key concept applied here is that any select-from-where expression returns a relation as a result and, therefore, can be inserted into another select-from-where anywhere that a relation can appear.
- Consider the query "Find the average instructors' salaries of those departments where the average salary is greater than $42,000."

```
select dept_name, avg_salary
from (select dept_name, avg (salary) as avg_salary
        from instructor
        group by dept_name)
        where avg_salary > 42000;
```

Output:

- We can give the subquery result relation a name, and rename the attributes, using the as clause, as illustrated below.

```
select dept_name, avg_salary
from (select dept_name, avg (salary)
        from instructor
        group by dept_name)
        as dept_avg (dept_name, avg_salary)
        where avg_salary > 42000;
```

Output:

- As another example, suppose we wish to find the maximum across all departments of the total salary at each department. The having clause does not help us in this task, but we can write this query easily by using a subquery in the from clause, as follows:

```
select max (tot_salary)
from (select dept_name, sum(salary)
        from instructor
```

```
group by dept_name) as dept_total (dept_name, tot_salary);
```

Output:

#The with Clause

- The with clause provides a way of defining a temporary relation whose definition is available only to the query in which the with clause occurs. Consider the following query, which finds those departments with the maximum budget.

```
with max_budget (value) as
    (select max(budget)
    from department)
    select budget from department, max_budget
    where department.budget = max_budget.value;
```

Output:

- For example, suppose we want to find all departments where the total salary is greater than the average of the total salary at all departments. We can write the query using the with clause as follows.

```
with dept_total (dept_name, value) as
    (select dept_name, sum(salary)
    from instructor
    group by dept_name),
    dept_total_avg(value) as
        (select avg(value) from dept_total)
            select dept_name
            from dept_total, dept_total_avg
             where dept_total.value >= dept_total_avg.value;
```

Output:

#Scalar Subqueries

- SQL allows subqueries to occur wherever an expression returning a value is permitted, provided the subquery returns only one tuple containing a single attribute; such subqueries are called scalar subqueries. For example, a subquery can be used in the select clause as illustrated in the following example that lists all departments along with the number of instructors in each department:

```
select dept_name,
        (select count(*)
        from instructor
        where department.dept_name = instructor.dept_name)
        as num_instructors
from department;
```

Output:

Conclusion:

Experiment-VIII

Aim: Apply Triggers (Creation of insert trigger, delete trigger, update trigger)

#Triggers

A Trigger is a user defined SQL command that is invoked automatically in response to an event such as insert, delete, or update.

A trigger is a statement that the system executes automatically as a side effect of a modification to the database. To design a trigger mechanism, we must meet two requirements:

1. Specify when a trigger is to be executed. This is broken up into an event that causes the trigger to be checked and a condition that must be satisfied for trigger execution to proceed.
2. Specify the actions to be taken when the trigger executes.

#Need for Triggers

Triggers can be used to implement certain integrity constraints that cannot be specified using the constraint mechanism of SQL. Triggers are also useful mechanisms for alerting humans or for starting certain tasks automatically when certain conditions are met.

Syntax:

```
CREATE TRIGGER
TRIGGER_NAME TRIGGER_TIME
TRIGGER_EVENT
ON TABLE_NAME FOR EACH ROW
BEGIN
---------------
----------------
END;
```

WHERE

TRIGGER_TIME → BEFORE, AFTER

TRIGGER_EVENT → INSERT, UPDATE,DELETE.

When an event happens, trigger do something. It checks data, handle errors, auditing tables.

Lets create Employees table by executing the following statement:

```
CREATE TABLE EMPLOYEES
        ( EMPLOYEE_ID INT PRIMARY KEY,
          FIRST_NAME VARCHAR(20),
          LAST_NAME VARCHAR(20),
          HOURLY_PAY NUMERIC(6,2),
          SALARY NUMERIC(10,2),
          JOB VARCHAR(10)
        );

INSERT INTO EMPLOYEES
VALUES (1,'JOHN','KARL',25.5,53040.00,'MANAGER'),
        (2,'JAMES','SMITH',15.00,31200.00,'CASHIER'),
        (3,'JACK','CLINTON',12.5,26000.00,'COOK');
```

#TRIGGER BEFORE UPDATION EVENT

```
CREATE TRIGGER BEFORE_HOURLY_PAY_UPDATE
BEFORE UPDATE ON EMPLOYEES
FOR EACH ROW
SET NEW.SALARY = (NEW.HOURLY_PAY * 2080);
SELECT * FROM EMPLOYEES;
```

OUTPUT:

```
UPDATE EMPLOYEES
SET HOURLY_PAY = HOURLY_PAY + 1;
SELECT * FROM EMPLOYEES;
```

OUTPUT:

 # TRIGGER BEFORE INSERTION EVENT

```
CREATE TRIGGER BEFORE_HOURLY_PAY_INSERT
BEFORE INSERT ON EMPLOYEES
FOR EACH ROW
SET NEW.SALARY = (NEW.HOURLY_PAY * 2080);

INSERT INTO EMPLOYEES
VALUES(4,'SHELDON','PLANKTON',10,NULL,'JANITOR');

SELECT * FROM EMPLOYEES;
```

OUTPUT:

#Now we create another table EXPENSES as follows:

```
CREATE TABLE EXPENSES
(EXPENSE_ID INT PRIMARY KEY,
EXPENSE_NAME VARCHAR(50),
EXPENSE_TOTAL DECIMAL(10,2));

INSERT INTO EXPENSES VALUES
    (1,'SALARIES',0),
    (2,'SUPPLIES',0),
    (3,'TAXES',0);

SELECT * FROM EXPENSES;
```

OUTPUT:

```
UPDATE EXPENSES
SET EXPENSE_TOTAL = (SELECT SUM(SALARY) FROM EMPLOYEES)
WHERE EXPENSE_NAME ='SALARIES';

SELECT * FROM EXPENSES;
```

OUTPUT:

#TRIGGER FOR AFTER DELETION EVENT

```
CREATE TRIGGER AFTER_SALARY_DELETE
AFTER DELETE ON EMPLOYEES
FOR EACH ROW
UPDATE EXPENSES
SET EXPENSE_TOTAL = EXPENSE_TOTAL - OLD.SALARY
WHERE EXPENSE_NAME = 'SALARIES';

DELETE FROM EMPLOYEES
WHERE EMPLOYEE_ID =4;

SELECT * FROM EXPENSES;
```

OUTPUT:

 # TRIGGER FOR AFTER INSERT EVENT

```
CREATE TRIGGER AFTER_SALARY_INSERT
AFTER INSERT ON EMPLOYEES
FOR EACH ROW
UPDATE EXPENSES
SET EXPENSE_TOTAL = EXPENSE_TOTAL + NEW.SALARY
WHERE EXPENSE_NAME = 'SALARIES';

INSERT INTO EMPLOYEES
VALUES(5,'BLACK','SMITH',10,NULL,'JANITOR');
SELECT * FROM EXPENSES;
```

OUTPUT:

 # TRIGGER FOR AFTER UPDATION EVENT

```
CREATE TRIGGER AFTER_SALARY_UPDATE
AFTER UPDATE ON EMPLOYEES
FOR EACH ROW
UPDATE EXPENSES
SET EXPENSE_TOTAL = EXPENSE_TOTAL + (NEW.SALARY - OLD.SALARY)
WHERE EXPENSE_NAME = 'SALARIES';
```

```
UPDATE EMPLOYEES
SET HOURLY_PAY =100
WHERE EMPLOYEE_ID =1;

SELECT * FROM EXPENSES;
```

OUTPUT:

Conclusion:

Experiment-IX

Aim: Apply Normal Forms: First, Second, Third and Boyce Codd Normal Forms on Application database.

Normalization

A large database defined as a single relation may result in data duplication. This repetition of data may result in:

- Making relations very large.
- It isn't easy to maintain and update data as it would involve searching many records in relation. Wastage and poor utilization of disk space and resources.
- The likelihood of errors and inconsistencies increases.

So to handle these problems, we should analyze and decompose the relations with redundant data into smaller, simpler, and well-structured relations that are satisfy desirable properties. Normalization is a process of decomposing the relations into relations with fewer attributes.

What is Normalization?

- 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 undesirable characteristics like Insertion, Update, and Deletion Anomalies.
- Normalization divides the larger table into smaller and links them using relationships. The normal form is used to reduce redundancy from the database table.

Why do we need Normalization?

The main reason for normalizing the relations is removing these anomalies. Failure to eliminate anomalies leads to data redundancy and can cause data integrity and other problems as the database grows. Normalization consists of a series of guidelines that helps to guide you in creating a good database structure.

Data modification anomalies can be categorized into three types:

Insertion Anomaly: Insertion Anomaly refers to when one cannot insert a new tuple into a relationship due to lack of data.

Deletion Anomaly: The delete anomaly refers to the situation where the deletion of data results in the unintended loss of some other important data.

Updation Anomaly: The update anomaly is when an update of a single data value requires multiple rows of data to be updated.

First Normal Form (1NF)

- A relation will be 1NF if it contains an atomic value.
- It states that an attribute of a table cannot hold multiple values. It must hold only single-valued attribute.
- First normal form disallows the multi-valued attribute, composite attribute, and their combinations.

Example: Relation EMPLOYEE is not in 1NF because of multi-valued attribute EMP_PHONE.

EMPLOYEE table:

EMP_ID	EMP_NAME	EMP_PHONE	EMP_STATE
14	John	7272826385, 9064738238	UP
20	Harry	8574783832	Bihar
12	Sam	7390372389, 8589830302	Punjab

The decomposition of the EMPLOYEE table into 1NF has been shown below:

EMP_ID	EMP_NAME	EMP_PHONE	EMP_STATE
14	John	7272826385	UP
14	John	9064738238	UP
20	Harry	8574783832	Bihar
12	Sam	7390372389	Punjab
12	Sam	8589830302	Punjab

Second Normal Form (2NF)

- In the 2NF, relational must be in 1NF.
- In the second normal form, all non-key attributes are fully functional dependent on the primarykey

Example: Let's assume, a school can store the data of teachers and the subjects they teach. In aschool, a teacher can teach more than one subject.

TEACHER table

TEACHER_ID	SUBJECT	TEACHER_AGE
25	Chemistry	30
25	Biology	30
47	English	35
83	Math	38
83	Computer	38

In the given table, non-prime attribute TEACHER_AGE is dependent on TEACHER_ID which is a propersubset of a candidate key. That's why it violates the rule for 2NF.
To convert the given table into 2NF, we decompose it into two tables:

TEACHER_DETAIL table:

TEACHER_ID	TEACHERS AGE
25	30
47	35
83	38

TEACHER_SUBJECT table:

TEACHER_ID	SUBJECT
25	Chemistry
25	Biology
47	English

83	Math
83	Computer

Third Normal Form (3NF)

- A relation will be in 3NF if it is in 2NF and not contain any transitive partial dependency. 3NF is used to reduce the data duplication. It is also used to achieve the data integrity.
- If there is no transitive dependency for non-prime attributes, then the relation must be in third normal form.

A relation is in third normal form if it holds atleast one of the following conditions for every non-trivial function dependency X → Y.

X is a super key.

Y is a prime attribute, i.e., 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:

> {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

Boyce Codd normal form (BCNF)

- BCNF is the advance version of 3NF. It is stricter than 3NF.
- A table is in BCNF if every functional dependency X → Y, X is the super key of the table. For BCNF, the table should be in 3NF, and for every FD, LHS is 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:

EMP_ID → EMP_COUNTRY

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:

EMP_ID → EMP_COUNTRY

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.

Conclusion:

Experiment-X

Aim: Transaction Processing

Transaction Processing Language: TCL is a computer programming language basically a component of SQL which is used to manage transactions in a database. This is used to manage changes made to the data in a table by DML statements. It also allows statements to be grouped together into logical transactions. There are three TCL commands: COMMIT, ROLLBACK, SAVEPOINT

COMMIT: It is used to save the changes made to the table permanently.
ROLLBACK: It is used to get back to the previous permanent status of the table. Similar to UNDO command. Table can be rolled back only if it is temporary. If you committed your changes, it cannot be rolled back.
SAVEPOINT: It is used along with the ROLLBACK command. It is used to mark a transaction in a table. A transaction can be named using this command. It is similar to bookmarks.

Let's create ACCOUNTINFO table by executing the following statement:

```
CREATE TABLE ACCOUNTINFO
    ( ACCOUNT_NUMBER VARCHAR(16) PRIMARY KEY,
     FIRST_NAME VARCHAR(20),
     LAST_NAME VARCHAR(20),
     BRANCH VARCHAR(10),
     BALANCE NUMERIC(10,2)
    );
```

```
#Execute START TENSACTION command to disable auto commit.
    START TRANSACTION;
# Insert rows into ACCOUNTINFO table
    INSERT INTO ACCOUNTINFO
    VALUES('456321456','SUNNY','AHUJA','WARDHA',5000),
            ('456321258','MONTY','KAPOOR','NAGPUR',6000),
            ('456589763','KOMAL','SINGH','AMARAVTI',3000),
```

```
                    ('456345698','ABHILASHA','SHARMA','YAVATMAL',2000);
    COMMIT;
    SELECT * FROM ACCOUNTINFO;
```

Output:

```
    DELETE FROM ACCOUNTINFO WHERE FIRST_NAME='MONTY';
    SELECT * FROM ACCOUNTINFO;
```

Output:

```
    ROLLBACK;
    SELECT * FROM ACCOUNTINFO;
```

Output:

```
# create savepoint
    SAVEPOINT UPD;
    UPDATE ACCOUNTINFO SET BALANCE=10000
    WHERE FIRST_NAME ='KOMAL';
    SELECT * FROM ACCOUNTINFO;
```

Output:

```
# Rollback to savepoint
    ROLLBACK TO UPD;
    SELECT * FROM ACCOUNTINFO;
```

Output:

Conclusion: