Shri Vileparle Kelvani Mandals'
DWARKADAS J. SANGHVI COLLEGE OF ENGINEERING
(Affiliated to the University of Mumbai)

Department of Computer Engineering
Academic year: 2022-23

Class : S.Y. B.Tech

Sem :III

Lab Manual

Subject: Database Management Systems Lab

(DBMS Lab)

CSL503

Experiment No-1

Aim: Identify the case study and detail statement of problem. Design an Entity-Relationship (ER) / Extended Entity-Relationship (EER) Model

Theory:

Entity Relationship Diagrams are a major data modelling tool and will help organize the data in your project into entities and define the relationships between the entities. This process has proved to enable the analyst to produce a good database structure so that the data can be stored and retrieved in a most efficient manner.

By using a graphical format it may help communication about the design between the designer and the user and the designer and the people who will implement it.

Components of an ERD

An ERD typically consists of four different graphical components:

1. Entity.
   A data entity is anything real or abstract about which we want to store data. Entity types fall into five classes: roles, events, locations, tangible things or concepts. E.g. employee, payment, campus, book. Specific examples of an entity are called instances. E.g. the employee John Jones, Mary Smith's payment, etc.
2. Relationship.
   A data relationship is a natural association that exists between one or more entities. E.g. Employees process payments.
3. Cardinality.
   Defines the number of occurrences of one entity for a single occurrence of the related entity. E.g. an employee may process many payments but might not process any payments depending on the nature of her job.
4. Attribute.
   A data attribute is a characteristic common to all or most instances of a particular entity. Synonyms include property, data element, and field. E.g. Name, address, Employee Number, pay rate are all attributes of the entity employee. An attribute or combination of attributes that uniquely identifies one and only one instance of an entity is called a primary key or identifier. E.g. Employee Number is a primary key for Employee.

A Simple Example

The above process will be illustrated by working through the following example.

A company has several departments. Each department has a supervisor and at least one employee. Employees must be assigned to at least one, but possibly more departments.

At least one employee is assigned to a project, but an employee may be on vacation and not assigned to any projects. The important data fields are the names of the departments, projects, supervisors and employees, as well as the supervisor and employee number and a unique project number.

Each of the following sections corresponds to one of the stages above.

Identify entities

In this stage, you look through the information about the system and seek to identify the roles, events, locations, concepts and other tangible things that you wish to store data about. One approach to this is to work through the information and highlight those words which you think correspond to entities.

A **company** has several **departments**. Each department has a **supervisor** and at least one **employee**. Employees must be assigned to at least one, but possibly more departments. At least one employee is assigned to a **project**, but an employee may be on vacation and not assigned to any projects. The important data fields are the names of the departments, projects, supervisors and employees, as well as the supervisor and employee number and a unique project number.

This example is quite simple in that the last couple of lines actually tell you what data is being stored and that makes it somewhat easy to identify the entities.

You may notice that "**company**" has been highlighted. It is **not** an example of an entity. A single company will use the system we are designing to keep track of its departments, projects, supervisors and employees.

A true entity should have more than one instance. Our system will probably contain information about multiple employees, supervisors, projects and departments. But it will only contain one instance of a company.

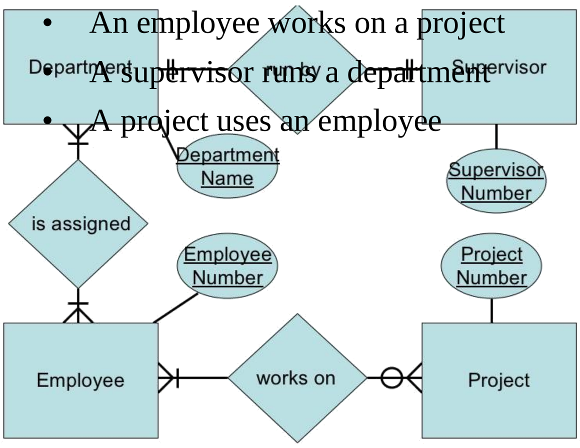
Find Relationships

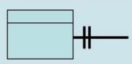
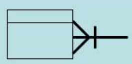
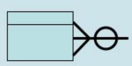
In this step the aim is to identify the associations, the connections between pairs of entities.

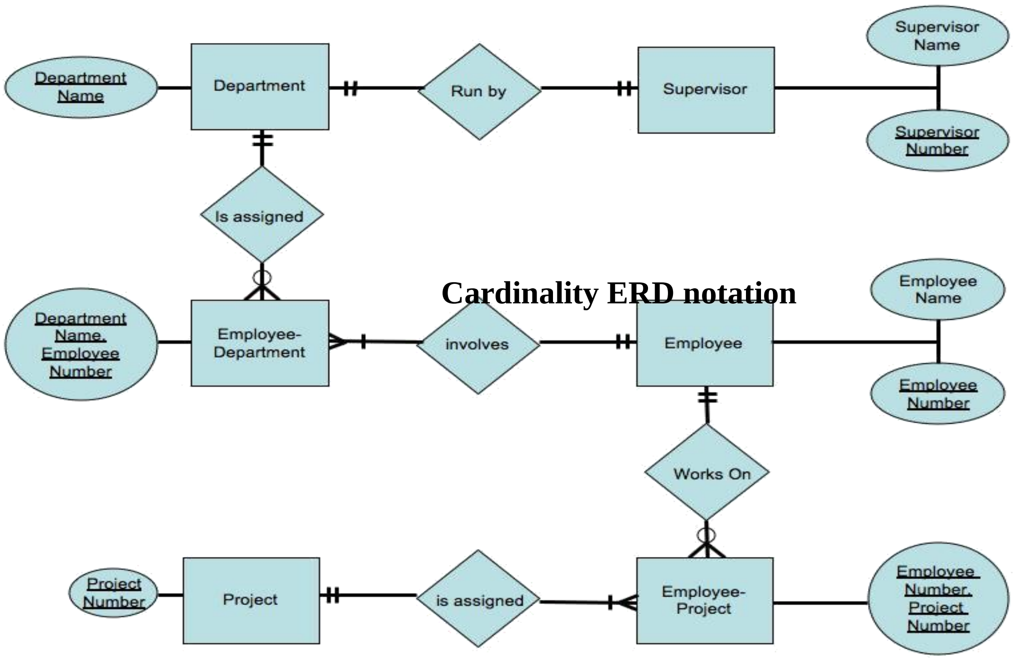
An example complete relationship matrix.

The names placed in the cells are meant to capture/describe the relationships. So you can use them like this

- A Department is assigned an employee
- A Department is run by a supervisor
- An employee belongs to a department
- An employee works on a project
- A supervisor runs a department
- A project uses an employee



Symbol	Meaning
	One and only one
	One or more
	Zero or more
	Zero or one



Cardinality ERD notation

Fully attributed ERD

Conclusion: Entity Relationship Modeling is a graphical approach to database design. It uses Entity/Relationship to represent real world objects. It is better to design E-R model before going for implementation of any project

Experiment No-2

Aim: Mapping ER/EER to Relational schema model.

Theory:

Step 1: Mapping of Regular Entity Types.

- For each regular (strong) entity type E in the ER schema, create a relation R that includes all the simple attributes of E.
- Choose one of the key attributes of E as the primary key for R. If the chosen key of E is composite, the set of simple attributes that form it will together form the primary key of R.

Step 2: Mapping of Weak Entity Types

- For each weak entity type W in the ER schema with owner entity type E, create a relation R and include all simple attributes (or simple components of composite attributes) of W as attributes of R.
- In addition, include as foreign key attributes of R the primary key attribute(s) of the relation(s) that correspond to the owner entity type(s).
- The primary key of R is the *combination of* the primary key(s) of the owner(s) and the partial key of the weak entity type W, if any.

Step 3: Mapping of Binary 1:1 Relation Types

- For each binary 1:1 relationship type R in the ER schema, identify the relations S and T that correspond to the entity types participating in R. There are three possible approaches:
  - Foreign Key approach: Choose one of the relations-S, say-and include a foreign key in S the primary key of T. It is better to choose an entity type with *total participation* in R in the role of S.
  - Merged relation option: An alternate mapping of a 1:1 relationship type is possible by merging the two entity types and the relationship into a single relation. This may be appropriate when *both participations are total.*
  - Cross-reference or relationship relation option: The third alternative is to set up a third relation R for the purpose of cross-referencing the primary keys of the two relations S and T representing the entity types.

Step 4: Mapping of Binary 1:N Relationship Types.

- For each regular binary 1:N relationship type R, identify the relation S that represent the participating entity type at the N-side of the relationship type.
- Include as foreign key in S the primary key of the relation T that represents the other entity type participating in R.
- Include any simple attributes of the 1:N relation type as attributes of S.

Step 5: Mapping of Binary M:N Relationship Types.

- For each regular binary M:N relationship type R, *create a new relation* S to represent R.
- Include as foreign key attributes in S the primary keys of the relations that represent the participating entity types; *their combination will form the primary key* of S.
- Also include any simple attributes of the M:N relationship type (or simple components of composite attributes) as attributes of S.

Step 6: Mapping of Multivalued attributes.

- For each multivalued attribute A, create a new relation R. This relation R will include an attribute corresponding to A, plus the primary key attribute K-as a foreign key in R-of the relation that represents the entity type of relationship type that has A as an attribute.
- The primary key of R is the combination of A and K. If the multivalued attribute is composite, we include its simple components.

Step 7: Mapping Specialization or Generalization.

Convert each specialization with m subclasses $\{S_1, S_2,\ldots,S_m\}$ and generalized superclass C, where the attributes of C are $\{k,a_1,\ldots a_n\}$ and k is the (primary) key, into relational schemas using one of the four following options:

- **Option 8A: Multiple relations-Superclass and subclasses.**
  Create a relation L for C with attributes Attrs(L) = $\{k,a_1,\ldots a_n\}$ and PK(L) = k. Create a relation L_i for each subclass S_i, $1 < i < m$, with the attributesAttrs(L_i) = {k} U {attributes of S_i} and PK(L_i)=k. This option works **for any specialization** (total or partial, disjoint of over-lapping).
- **Option 8B: Multiple relations-Subclass relations only**
  Create a relation L_i for each subclass S_i, $1 < i < m$, with the attributes Attr(L_i) = {attributes of S_i} U $\{k,a_1\ldots,a_n\}$ and PK(L_i) = k. This option only works for a specialization whose subclasses are **total** (every entity in the superclass must belong to (at least) one of the subclasses).
- **Option 8C: Single relation with one type attribute.**
  Create a single relation L with attributes Attrs(L) = $\{k,a_1,\ldots a_n\}$ U {attributes of S_1} U…U {attributes of S_m} U {t} and PK(L) = k. The attribute t is called a type (or **discriminating**) attribute that indicates the subclass to which each tuple belongs(**Disjoint**)
- **Option 8D: Single relation with multiple type attributes.**
  Create a single relation schema L with attributes Attrs(L) = $\{k,a_1,\ldots a_n\}$ U {attributes of S_1} U…U {attributes of S_m} U $\{t_1, t_2,\ldots,t_m\}$ and PK(L) = k. Each t_i, 1 < I < m, is a Boolean type attribute indicating whether a tuple belongs to the subclass S_i.(**Overlap)**

Conclusion: By applying conversion process we converted Entity Relationship/EER Modeling to relational schema.

Experiment No-3

Aim: Create and populate database using Data Definition Language (DDL) and DML Commands for your specified System.

Theory: DDL- Data Definition Language (DDL) statements are used to define the database structure or schema. Data Definition Language understanding with database schemas and describes how the data should consist in the database, therefore language statements like CREATE TABLE or ALTER TABLE belongs to the DDL. DDL is about "metadata".

DDL includes commands such as CREATE, ALTER and DROP statements. DDL is used to CREATE, ALTER OR DROP the database objects (Table, Views, Users).

Data Definition Language (DDL) are used different statements :

1. CREATE - to create objects in the database
2. ALTER - alters the structure of the database
3. DROP - delete objects from the database
4. TRUNCATE - remove all records from a table, including all spaces allocated for the records are removed
5. RENAME - rename an object

Use Following schema to perform the experiment.

Student (<u>stuId</u>, lastName, firstName, major, credits)

Faculty (<u>facId</u>, name, department, rank)

Class (<u>classNumber</u>, facId, schedule, room)

Enroll (<u>classNumber</u>, *stuId*, grade)

NOTE: Underlined Text: Primary Key, Italic Text: Forign Key

- **<u>DDL</u>**

1) **CREATE**
   a) **To create a database**

      CREATE DATABASE *dbname*;
      Eg-

CREATE DATABASE UniversityData;

b) To select existing database

Use dbname;

Eg- USE UniversityData;

c) To create a Table

The create table command defines each column of the table uniquely. Each column has minimum of three attributes.

- Name
- Data type
- Size (column width).

Each table column definition is a single clause in the create table syntax. Each table column definition is separated from the other by a comma. Finally, the SQL statement is terminated with a semicolon.

Syntax: Create table table name(fieldname1 datatype(),fieldname2 datatype()...);

The Structure of Create Table Command

CREATE TABLE Student	(
stuId	CHAR(6),
lastName	CHAR(20) NOT NULL,
firstName	CHAR(20) NOT NULL,
major	(CHAR(10),
credits	SMALLINT DEFAULT 0,
CONSTRAINT Student_stuId_pk PRIMARY KEY (stuId)),	
CONSTRAINT Student_credits_cc CHECK ((CREDITS>=0) AND (credits < 150));	
CREATE TABLE Faculty	(
facId	CHAR(6),
name	CHAR(20) NOT NULL,
department	CHAR(20) NOT NULL,
rank	CHAR(10),
CONSTRAINT Faculty_facId_pk PRIMARY KEY (facId));	
CREATE TABLE Class	(
classNumber	CHAR(8),
facId	CHAR(6) NOT NULL,
schedule	CHAR(8),
room	CHAR(6),
CONSTRAINT Class_classNumber_pk PRIMARY KEY (classNumber),	
CONSTRAINT Class_facId_fk FOREIGN KEY (facId) REFERENCES Faculty (facId) ON DELETE NO ACTION);	
CREATE TABLE Enroll	(
classNumber	CHAR(8),
stuId	CHAR(6),
grade	CHAR(2),
CONSTRAINT Enroll_classNumber_stuId_pk PRIMARY KEY (classNumber, stuId),	
CONSTRAINT Enroll_classNumber_fk FOREIGN KEY (classNumber) REFERENCES Class (classNumber) ON DELETE NO ACTION,	
CONSTRAINT Enroll_stuId_fk FOREIGN KEY (stuId) REFERENCES Student (stuId) ON DELETE CASCADE);	

Note: Create Tables for an extra entities and relationships

2) ALTER

By The use of ALTER TABLE Command we can **modify** our exiting table.

Adding New Columns

Syntax:
ALTER TABLE <table_name>
ADD (<NewColumnName> <Data_Type>(<size>),......n)

```
Example: Add a new column,cTitle, to our Classtable
         ALTER TABLE Class ADD cTitle CHAR(30);
```

The schema of the Class table would then be:
Class(classNumber,facId,schedule,room,cTitle)

Dropping a Column from the Table

Syntax:

ALTER TABLE <table_name> DROP COLUMN <column_name>

Example:

```
Example: Drop the cTitle column from the Class table
ALTER TABLE Class DROP COLUMN cTitle;
```

This command will drop particular column

If we want to add, drop, or change a constraint, we can use the same ALTER TABLE command. For example, if we created the Class table and neglected to make facIda foreign key in Class, we could add the constraint at any time by writing:

```
ALTER TABLE Class ADD CONSTRAINT Class_facId_fk FOREIGN KEY (facId)REFERENCES
Faculty (facId)ON DELETE NO ACTION);
```

We could drop an existing named constraint using the ALTER TABLE command.
For example, to drop the check condition on the creditsattribute of Student that we created earlier, we could write:

```
ALTER TABLE Student DROP CONSTRAINT Student_credits_cc;
```

Modifying Existing Column

Syntax:
ALTER TABLE <table_name> MODIFY <column_name> NewDataType>(<NewSize>)

Example:

```
ALTER TABLE Student MODIFY stuId Varchar(20);
```

Renaming Existing Table

Syntax:

ALTER TABLE <table_name> RENAME <new_table_name>

Example:

```
ALTER TABLE student RENAME new_student;

ALTER TABLE new_student RENAME student; (To revert the change)
```

3) RENAME

Syntax:
RENAME TABLE <OldTableName> TO <NewTableName>

Example:
```
RENAME table visiting TO visiting_staff;
```

4) DROP

Syntax:
DROP TABLE <table_name>

Example:
```
DROP TABLE visiting_staff;
```

5) TRUNCATE

Syntax:
TRUNCATE TABLE <Table_name>

Example:
```
TRUNCATE TABLE visiting_staff;
```

6- SHOW
To check available databases and tables

Syntax

```
SHOW DATABASES;
SHOW TABLES;
```

7- DESCRIBE

To obtain information about table structure or query execution plans.

DESCRIBE < table_name>
DESC < table_name>

Example-

```
DESC Student;
```

- **Apply Integrity Constraints for the specified system**

MySQL CONSTRAINT is used to define rules to allow or restrict what values can be stored in columns. The purpose of inducing constraints is to enforce the integrity of a database.

MySQL CONSTRAINTS are used to limit the type of data that can be inserted into a table.

MySQL CONSTRAINTS can be classified into two types - column level and table level.

The column level constraints can apply only to one column where as table level constraints are applied to the entire table.

MySQL CONSTRAINT is declared at the time of creating a table.

MySQL CONSTRAINTs are :

- NOT NULL
- UNIQUE
- PRIMARY KEY
- FOREIGN KEY
- CHECK
- DEFAULT
- AUTO INCREMENT

CONSTRAINT	**DESCRIPTION**

NOT NULL	In MySQL NOT NULL constraint allows to specify that a column can not contain any NULL value. MySQL NOT NULL can be used to CREATE and ALTER a table.
	
	`Eg. lastName CHAR(20) NOT NULL,`
	`firstName CHAR(20) NOT NULL`

UNIQUE	The UNIQUE constraint in MySQL does not allow to insert a duplicate value in a column. The UNIQUE constraint maintains the uniqueness of a column in a table. More than one UNIQUE column can be used in a table. `Eg. CONSTRAINT Class_schedule_room_uk UNIQUE (schedule, room)`
PRIMARY KEY	A PRIMARY KEY constraint for a table enforces the table to accept unique data for a specific column and this constraint creates a unique index for accessing the table faster. `Eg: CONSTRAINT Faculty_facId_pk PRIMARY KEY (facId));`
FOREIGN KEY	A FOREIGN KEY in MySQL creates a link between two tables by one specific column of both tables. The specified column in one table must be a PRIMARY KEY and referred by the column of another table known as FOREIGN KEY. `Eg. CONSTRAINT Class_facId_fk FOREIGN KEY (facId) REFERENCES Faculty (facId) ON DELETE NO ACTION);`
CHECK	A CHECK constraint controls the values in the associated column. The CHECK constraint determines whether the value is valid or not from a logical expression. `Eg: CONSTRAINT Student_credits_cc CHECK ((credits>=0) AND (credits <150);`
DEFAULT	In a MySQL table, each column must contain a value (including a NULL). While inserting data into a table, if no value is supplied to a column, then the column gets the value set as DEFAULT. `credits SMALLINT DEFAULT 0 CHECK ((credits>=0) AND (credits <150))`
AUTO INCREMENT	Auto-increment allows a unique number to be generated automatically when a new record is inserted into a table. Often this is the primary key field that we would like to be created automatically every time a new record is inserted. `Eg. Field name int AUTO_INCREMENT PRIMARY KEY`

- **DML(Data Manipulation Language)**

A data manipulation language (DML) is a family of computer languages including commands permitting users to manipulate data in a database. This manipulation involves inserting data into database tables, retrieving existing data, deleting data from existing tables and modifying existing data. DML is mostly incorporated in SQL databases.

Use following database to perform the experiment.
Database:

Student

stuId	lastName	firstName	major	credits
S1001	Smith	Tom	History	90
S1002	Chin	Ann	Math	36
S1005	Lee	Perry	History	3
S1010	Burns	Edward	Art	63
S1013	McCarthy	Owen	Math	0
S1015	Jones	Mary	Math	42
S1020	Rivera	Jane	CSC	15

Faculty

facId	name	department	rank
F101	Adams	Art	Professor
F105	Tanaka	CSC	Instructor
F110	Byrne	Math	Assistant
F115	Smith	History	Associate
F221	Smith	CSC	Professor

Class

classNumber	facId	schedule	room
ART103A	F101	MWF9	H221
CSC201A	F105	TuThF10	M110
CSC203A	F105	MThF12	M110
HST205A	F115	MWF11	H221
MTH101B	F110	MTuTh9	H225
MTH103C	F110	MWF11	H225

Enroll

stuId	classNumber	grade
S1001	ART103A	A
S1001	HST205A	C
S1002	ART103A	D
S1002	CSC201A	F
S1002	MTH103C	B
S1010	ART103A	
S1010	MTH103C	
S1020	CSC201A	B
S1020	MTH101B	A

1) INSERT

This command adds one or more records to a database table.

Syntax

INSERT INTO "table_name" ("column1", "column2", ...)
VALUES ("value1", "value2", ...);

Example

1) insert into Student (stuld, lastname, firstname, major, credits) values('S1001','Smith',Tom', 'History', 90);

2) SELECT

The SELECT statement is used to select data from a database.

Syntax

SELECT * FROM *table_name*;

Example-

Select * from Student;

3) UPDATE

The UPDATE statement is used to update existing records in a table.

Syntax

UPDATE *table_name*
SET *column1*=*value1*, *column2*=*value2,...*
WHERE *some_column*=*some_value*;

Example-

1) update Student set Name='JANEE' where stuID=S1020;

4) DELETE

This command removes one or more records from a table according to specified conditions.

Syntax: DELETE
FROM *table_name*
WHERE *some_column*=*some_value*;

Example-1) delete from Student
where stuID=S1020;

Conclusion: Data definition Language is used to create database and apply Integrity Constraints for the project and used DML to populate database.

Experiment No-4

Aim: Perform Simple queries, string manipulation operations

The most commonly used SQL command is SELECT statement. SQL SELECT statement is used to query or retrieve data from a table in the database. A query may retrieve information from specified columns or from all of the columns in the table. To create a simple SQL SELECT Statement, you must specify the column(s) name and the table name. The whole query is called SQL SELECT Statement.

Syntax-

SELECT [DISTINCT|ALL] { * | [fieldExpression [AS newName]} FROM tableName [alias] [WHERE condition][GROUP BY fieldName(s)] [HAVING condition] ORDER BY fieldName(s)

- **SELECT** is the SQL keyword that lets the database know that you want to retrieve data.
- **[DISTINCT | ALL]** are optional keywords that can be used to fine tune the results returned from the SQL SELECT statement. If nothing is specified then ALL is assumed as the default.
- **{*| [fieldExpression [AS newName]}** at least one part must be specified, "*" selected all the fields from the specified table name, fieldExpression performs some computations on the specified fields such as adding numbers or putting together two string fields into one.
- **FROM** tableName is mandatory and must contain at least one table, multiple tables must be separated using commas or joined using the JOIN keyword.
- **WHERE** condition is optional, it can be used to specify criteria in the result set returned from the query.
- **GROUP BY** is used to put together records that have the same field values.
- **HAVING** condition is used to specify criteria when working using the GROUP BY keyword.
- **ORDER BY** is used to specify the sort order of the result set.

Queries:

1. Select all column from a table
   Select * from tablename;
2. Select specific column of list from a table
   Select col1,col2 from tablename;
3. Select ….where clause with various operators (<,>,<=,> =,IN,NOT IN,BETWEEN..AND,NOT BETWEEN… AND)

Select col1 from tablename where col2>value1;
Select * from tablename where col2 in(value1,value2,value3)
Select * from tablename where col2 between value1 and value2
Select * from tablename where col2 NOT between value1 and value2

4. Select ….where clause with multiple conditions
   Select * from tablename where col2=Value1 and/or col3=value2

5. Select ….where clause with string matching
   - Starts with any character
   - ends with any character
   - have a specific substring
   - character at specific position
   - starts with a specific character and having specifically n no of characters
   - starts and ends with different character

6. Query your database by applying aggregate function MIN,MAX,COUNT,AVG,SUM (create alias also)
7. Query your database by applying group by clause using one column and multiple column
8. Query your database by applying order by clause using one column and multiple column
9. Query your database by applying group by, order by and having clause simultaneously
10. Select clause with various date functions

CURDATE(),CURRENT_TIME(),CURRENT_TIMESTAMP(),DATE(),DATEDIFF(),DAY(),DAYNAME(),EXTRACT(),MINUTE(),MONTH(),WEEK(),NOW(),YEAR()

11. Perform set operation on multiple select queries(UNION)

Examples:

Question: Get names, IDs and number of credits of all Math majors.

SQL Query:

```
SELECT    lastName, firstName, stuId, credits
FROM      Student
WHERE     major = 'Math';
```

Result:

lastName	firstName	stuId	credits
Jones	Mary	S1015	42
Chin	Ann	S1002	36
McCarthy	Owen	S1013	9

Question: Get all information about CSC Faculty.

Solution: We want the entire Faculty record of any faculty member whose department is 'CSC'. Since many SQL retrievals require all columns of a single table, there is a short way of expressing "all columns," namely by using an asterisk in place of the column names in the SELECT line.

SQL Query:

```
SELECT    *
FROM      Faculty
WHERE     department = 'CSC';
```

Result:

facId	name	department	rank
F105	Tanaka	CSC	Instructor
F221	Smith	CSC	Professor

OR

```
SELECT    facId, name, department, rank
FROM      Faculty
WHERE     department  = 'CSC';
```

Question: Get the course number of all courses in which students are enrolled.

Solution: We go to the Enroll table rather than the Class table, because it is possible there is a Class record for a planned class in which no one is enrolled. From the Enroll table, we could ask for a list of all the classNumber values, as follows.

SQL Query:

```
SELECT    classNumber
FROM      Enroll;
```

Result:

classNumber
ART103A
CSC201A
CSC201A
ART103A
ART103A
MTHlOlB
HST205A
MTH103C
MTH103C

Notice that there are several duplicates in our result; it is a multi-set. To eliminate the duplicates, we need to use the DISTINCT option in the SELECT line. If we write,

```
SELECT DISTINCT classNumber
FROM Enroll;
```

The result would be:

classNumber
ART103A
CSC201A
MTH101B
HST205A
MTH103C

In any retrieval, especially if there is a possibility of confusion because the same column name appears on two different tables, specify *tablename.colname*. In this example, we could have written:

```
SELECT     DISTINCT Enroll.classNumber
FROM
Enroll;
```

Question: Get all information about all students.

Solution: Because we want all columns of the Student table, we use the asterisk notation. Because we want all the records in the table, we omit the WHERE line.

SQL Query:

```
SELECT     *
FROM       Student;
```

Result: The result is the entire Student table.

Question: Get names and IDs of all Faculty members, arranged in alphabetical order by name. Call the resulting columns Faculty-Name and FacultyNumber.

Solution: The ORDER BY option in the SQL SELECT allows us to order the retrieved records in ascending (ASC—the default) or descending (DESC) order on any field or combination of fields, regardless of whether that field appears in the results. If we order by more than one field, the one named first determines major order, the next minor order, and so on.

SQL Query:

```
SELECT    name AS FacultyName, facId AS
          FacultyNumber
FROM      Faculty
ORDER BY  name;
```

Result:

FacultyName	FacultyNumber
Adams	F101
Byrne	F110
Smith	F202
Smith	F221
Tanaka	F105

The column headings are changed to the ones specified in the AS clause. We can rename any column or columns for display in this way. Note the duplicate name of 'Smith'. Since we did not specify minor order, the system will arrange these two rows in any order it chooses. We could break the "tie" by giving a minor order, as follows:

```
SELECT    name AS FacultyName, facId AS
          FacultyNumber
FROM      Faculty
ORDER BY  name, department;
```

Now the Smith records will be reversed, since F221 is assigned to CSC, which is alphabetically before History. Note also that the field that determines ordering need not be one of the ones displayed.

Question: Get names of all math majors who have more than 30 credits.

Solution: From the Student table, we choose those rows where the major is 'Math' and the number of credits is greater than 30. We express these two conditions by connecting them with 'AND.' We display only the lastName and firstName.

SQL Query:

```
SELECT    lastName, firstName
FROM      Student
WHERE     major = 'Math'
          AND credits > 30;
```

Result:

lastName	firstName
Jones	Mary
Chin	Ann

The predicate can be as complex as necessary by using the standard comparison operators =, <>, <, <=, >, >= and the standard logical operators AND, OR and NOT, with parentheses, if needed or desired, to show order of evaluation.

SELECT Using Multiple Tables

Example 7. Natural Join

Question: Find IDs and names of all students taking ART103A.

Solution:This question requires the use of two tables. We first look in the Enroll table for records where the classNumberis'ART103A.' We then look up the Student table for records with matching stuId values, and join those records into a new table. From this table, we find the lastName and firstName. This is similar to the JOIN operation in relational algebra. SQL allows us to do a natural join, as described in Section 4.6.2, by naming the tables involved and expressing in the predicate the condition that the records should match on the common field.

SQL Query:

```
SELECT    Enroll.stuId, lastName, firstName
FROM      Student, Enroll
WHERE     classNumber = 'ART103A'
          AND Enroll.stuId = Student.stuId;
```

Result:

stuId	lastName	firstName
Sl00l	Smith	Tom
Sl0l0	Burns	Edward
S1002	Chin	Ann

Example 8. Natural Join with Ordering

Question:Find stuIdand gradeof all students taking anycourse taught by the Facultymember whose facIdis F110.Arrange in order by stuId.

Solution:We need to look at the Classtable to find the class-Numberof all courses taught by F110. We then look at the Enrolltable for records with matching classNumbervalues, and get thejoin of the tables. From this we find the corresponding stuIdandgrade. Because we are using two tables, we will write this as a join.

SQL Query:

```
SELECT     stuId,grade
FROM       Class,Enroll
WHERE      facId = 'F110' AND Class.classNumber
           = Enroll.classNumber
ORDER BY   stuId ASC;
```

Result:

stuId	grade
S1002	B
S1010	
S1020	A

Example 9. Natural Join of Three Tables

Question:Find course numbers and the names and majors of all students enrolled in the courses taught by Faculty member F110.

Solution:As in the previous example, we need to start at theClasstable to find the classNumberof all courses taught byF110. We then compare these with classNumbervalues in theEnrolltable to find the stuIdvalues of all students in those6.4 Manipulating the Database: SQL DML307007_33148_06_286-372_1e.qxd 1/28/04 8:01 AM Page 307 courses. Then we look at the Student table to find the names and majors of all the students enrolled in them.

SQL Query

```
SELECT  Enroll.classNumber, lastName,
        firstName, major
FROM    Class, Enroll, Student
WHERE   facId = 'F110'
        AND Class.classNumber =
        Enroll.classNumber
        AND Enroll.stuId = Student.stuId;
```

Result:

classNumber	lastName	firstName	major
MTH101B	Rivera	Jane	CSC
MTH103C	Burns	Edward	Art
MTH103C	Chin	Ann	Math

Example 10. Use of Aliases

Question:Get a list of all courses that meet in the same room, withtheir schedules and room numbers.

Solution:This requires comparing the Classtable with itself, and it would be useful if there were two copies of the table so we could do a natural join. We can pretend that there are two copies of a table by giving it two "aliases," for example, COPY and COPY2, and then treating these names as if they were the names of two distinct tables. We introduce the "aliases" in the FROM line by writing them immediately after the real table names. Then we have the aliases available for use in the other lines of the query.

SQL Query:

```
SELECT COPYl.classNumber, COPYl.schedule, COPYl.room,
COPY2.classNumber, COPY2.schedule
FROM    Class COPYl, Class COPY2
WHERE   COPYl.room = COPY2.room
        AND COPYl.classNumber > COPY2.classNumber ;
```

Result:

COPYl.classNumber	COPYl.schedule	COPYl.room	COPY2.classNumber	COPY2.schedule
ART103A	MWF9	H221	HST205A	MWF11
CSC201A	TUTHF10	M110	CSC203A	MTHF12
MTH101B	MTUTH9	H225	MTH103C	MWF11

Example 11. Join without Equality Condition

Question:Find all combinations of students and Faculty where the student's major is different from the Faculty member's department.

Solution:This unusual request is to illustrate a join in which the condition is not an equality on a common field. In this case, the fields we are examining, major and department, do not even have the same name. However, we can compare them since they have the same domain. Since we are not told which columns to show in the result, we use our judgment.

SQL Query:

```
SELECT   stuId, lastName, firstName, major, facId,
         name, department
FROM     Student, Faculty
WHERE    Student.major <> Faculty.department;
```

Result:

stuId	lastName	firstName	major	facId	name	department
S1001	Smith	Tom	History	F101	Adams	Art
S1001	Smith	Tom	History	F105	Tanaka	CS
S1001	Smith	Tom	History	F110	Byrne	Math
S1001	Smith	Tom	History	F221	Smith	CS
S1010	Burns	Edward	Art	F202	Smith	History
....						
....						
....						
S1013	McCarthy	Owen	Math	F221	Smith	CS

Examples-

1. **Retrieve all attribute values from employee.**
   select * from employee;

2. **Retrieve the cross product of employee and department.**
   select *
   from employee, department;

3. **Retrieve employee id of all employees who work for department no.3**
   select id
   from employee
   where dno=3;

4. **Retrieve employee name and phone numbers of all employees residing at vile parle**.
   select ename, phone_no
   from employee, department
   where location='vile parle' and dnumber=dno;

5. **Retrieve employee name and phone numbers of all employees residing at vile parle**.
   select e.ename, e.phone_no
   from employee as e, department as d
   where d.location='vile parle' and d.dnumber=e.dno;

6. **Retrieve distinct addresses from employee.**
   select distinct address from employee;

7. **Retrieve all addresses from employee.**
   select all address from employee;

8. **Retrieve all employee names whose address is in vile parle.**
   select ename
   from employee
   where address like '%vile%';

9. **Retrieve all employee names whose phone number starts with 99.**
   select ename
   from employee
   where phone_no like '99________';

10. **Retrieve all employee names as employee_name for all employees working for department no.3**
    select ename as employee_name
    from employee
    where dno=3;

11. **Retrieve all employees in department no.1 whose salary is between 30000 and 60000**
    select *
    from employee
    where (salary between 30000 and 60000) and dno=1;

12. **Retrieve employee id, name, phone number and department name of all employees who work for computer department ordered alphabetically in ascending order by employee name and descending order by department name**.

```
select id, ename, phone_no,dname
from employee,department
where dname='comp' and dnumber=dno
order by dname desc, ename asc;
```

13. Retrieve names of all employees who do not get salary.

```
select ename
from employee
where salary is null;
```

14. Retrieve maximum salary offered to employees

```
select max(salary)
from employee;
```

15. Retrieve minimum salary offered to employees

```
select min(salary)
from employee;
```

16. Retrieve department wise average salary of employee.

```
select avg(salary)
from employee
group by dno;
```

17. Retrieve total number of employees.

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

15. Retrieve department name, department number and average salary for all departments having average salary greater than 40000

```
select dname, dnumber, avg(salary)
from employee, department
where dnumber=dno
group by dno
having avg(salary)>40000;
```

18. Retrieve total salary given to employees.

```
select sum(salary)
from employee;
```

Conclusion: Select queries with various clauses like group by, order by and aggregate functions is implemented in mysql**.**

Experiment No-5

Aim: Perform Nested queries and Complex queries

Theory: A Subquery or Inner query or Nested query is a query within another SQL query and embedded within the WHERE clause.

A subquery is used to return data that will be used in the main query as a condition to further restrict the data to be retrieved.

Subqueries can be used with the SELECT, INSERT, UPDATE, and DELETE statements along with the operators like =, <, >, >=, <=, IN, BETWEEN etc.

```
SELECT column_name [, column_name ]
FROM   table1 [, table2 ]
WHERE  column_name OPERATOR
    (SELECT column_name [, column_name ]
    FROM table1 [, table2 ]
    [WHERE])
```

Example 12. Using a Subquery with Equality

Question:Find the numbers of all the courses taught by Byrne of the math department.

Solution:We already know how to do this by using a natural join, but there is another way of finding the solution. Instead of imagining a join from which we choose records with the same facId, we could visualize this as two separate queries. For the first one, we would go to the Faculty table and find the record with name of Byrne and department of Math. We could make a note of the corresponding facId. Then we could take the result of that query, namely Fll0, and search the Class table for records with that value in facId. Once we found them, we would display the classNumber. SQL allows us to sequence these queries so that the result of the first can be used in the second, shown as follows:

SQL Query:

```
SELECT    classNumber
FROM      Class
WHERE     facId =
          (SELECT   facId
          FROM      Faculty
          WHERE     name  = 'Byrne'
                    AND department = 'Math');
```

Result:

<u>classNumber</u>
MTH101B
MTH103C

Note that this result could have been produced by the following SQL query, using a join:

```
SELECT    classNumber
FROM      Class, Faculty
WHERE     name = 'Byrne' AND department = 'Math'
          AND Class.facId = Faculty.facId;
```

A subquery can be used in place of a join, provided the result to be displayed is contained in a single table and the data retrieved from the subquery con-sists of only one column. When you write a subquery involving two tables, you name only one table in each SELECT. The query to be done first, the subquery, is the one in parentheses, following the first WHERE line. The main query is performed using the result of the subquery. Normally you want the value of some field in the table mentioned in the main query to match the value of some field from the table in the subquery. In this example, we knew we would get only one value from the subquery, since facId is the key of Faculty, so a unique value would be produced. Therefore, we were able to use equality as the operator.

Since the sub-query is performed first, the SELECT . . . FROM . . . WHERE of the subquery is actually replaced by the value retrieved, so the main query is changed to the following:

```
SELECT    classNumber
FROM      Class
WHERE     facId = ('F110');
```

Example 13. Subquery Using 'IN'

Question:Find the names and IDs of all Faculty members who teach a class in Room H221.

Solution:We need two tables, Class and Faculty, to answer this question. We also see that the names and IDs both appear on the Faculty table, so we have a choice of a join or a subquery. If we use a subquery, we begin with the Class table to find facId values for any courses that meet in Room H221. We find two such entries, so we make a note of those values. Then we go to the Facultytable and compare the facId value of each record on that table with the two ID values from Class, and display the corresponding facId and name.

SQL Query:

```
SELECT   name, facId
FROM     Faculty
WHERE    facId IN
         (SELECT   facId
         FROM      Class
         WHERE     room = 'H221');
```

Result:

name facId
Adams F10l
Smith F202

In the WHERE line of the main query we used IN, rather than =, because the result of the subquery is a set of values rather than a single value. We are saying we want the facId in Faculty to match any member of the set of values we obtain from the subquery. When the subquery is replacedby the values retrieved, the main query becomes:

```
SELECT   name, facId
FROM     Faculty
WHERE    FACID IN ('F101','F202');
```

The IN is a more general form of subquery than the comparison operator, which is restricted to the case where a single value is produced. We can also use the negative form 'NOT IN', which will evaluate to true if the record has a field value which is not in the set of values retrieved by the subquery.

Example 14. Nested Subqueries

Question:Get an alphabetical list of names and IDs of all students in any class taught by F110.

Solution:We need three tables, Student, Enroll, and Class, to answer this question. However, the values to be displayed appear on one table, Student, so we can use a subquery. First we check the Class table to find the classNumber of all courses taught byF110. We find two values, MTH101B and MTH103C. Next we go to the Enroll table to find the stuId of all students in either of these courses. We find three values, S1020, S1010, and S1002. We nowlook at the Studenttable to find the records with matching stuId values, and display the stuId, lastName, and firstName, in alphabetical order by name.

SQL Query:

```
SELECT    lastName, firstName, stuId
FROM      Student
WHERE     stuId IN
          (SELECT   stuId
          FROM      Enroll
          WHERE     classNumber IN
                    (SELECT   classNumber
                    FROM      Class
                    WHERE     facId = 'F110'))
ORDER BY lastName, firstName ASC;
```

Result:

lastName	firstName	stuId
Burns	Edward	Sl0l0
Chin	Ann	S1002
Rivera	Jane	S1020

In execution, the most deeply nested SELECT is done first, and it is replaced by the values retrieved, so we have:

```
SELECT    lastName, firstName, stuId
FROM      Student
WHERE     stuId IN
          (SELECT   stuId
          FROM      Enroll
          WHERE     classNumber IN
                    ('MTH10lB', 'MTH103C'))
ORDER BY lastName, firstName ASC;
```

Next the subquery on `Enroll` is done, and we get:

```
SELECT    lastName, firstName, stuId
FROM      Student
WHERE     stuId IN
          ('S1020', 'Sl0l0', 'S1002')
ORDER BY lastName, firstName ASC;
```

Finally, the main query is done, and we get the result shown earlier. Note that the ordering refers to the final result, not to any intermediate steps. Also note that we could have performed either part of the operation as a natural join and the other part as a subquery, mixing both methods.

Example 15. Query Using EXISTS

Question:Find the names of all students enrolled in CSC201A.

Solution:We already know how to write this using a join or a sub-query with IN. However, another way of expressing this query isto use the existential quantifier, EXISTS, with a subquery.

SQL Query:

```
SELECT    lastName, firstName
FROM      Student
WHERE     EXISTS
          (SELECT   *
          FROM      Enroll
          WHERE     Enroll.stuId = Student.stuId
          AND       classNumber = 'CSC201A');
```

Result:

lastName	firstName
Rivera	Jane
Chin	Ann

This query could be phrased as "Find the lastName and firstName of all students such that there exists an Enroll record containing their stuId with a classNumberof CSC201A". The test for inclusion is the existence of such a record. If it exists, the "EXISTS (SELECT FROM . . .;" evaluates to true.

Notice we needed to use the name of the main query table (Student) in the subquery to express the condition Student.stuId = Enroll.stuId.In general, we avoid mentioning a table not listed in the FROM for that particular query, but it is necessary and permissible to do so in this case. This form is called a correlated subquery, since the table in the subquery is being compared to the table in the main query.

Example 16. Query Using NOT EXISTS

Question:Find the names of all students who are not enrolled inCSC201A.

Solution:Unlike the previous example, we cannot readily expressthis using a join or an IN subquery. Instead, we will use NOTEXISTS.

SQL Query:

```
SELECT   lastName, firstName
FROM     Student
WHERE    NOT EXISTS
         (SELECT
         FROM     Enroll
         WHERE    Student.stuId = Enroll.stuId
         AND      classNumber = 'CSC201A');
```

Result:

lastName	firstName
Smith	Tom
Burns	Edward
Jones	Mary
McCarthy	Owen

We could phrase this query as "Select student names from the `Student` table such that there is no `Enroll` record containing their STUID values with `classNumber` of CSC201A."

1) Find the customers who are borrowers from the bank and who appear in the list of account holders.

```
SELECT cust_name
from borrower
where cust_name in ( select cust_name
                     from depositor);
```

2) Find the customers who are borrowers from the bank but do not hold an account.

```
SELECT cust_name
from borrower
where cust_name not in ( select cust_name
                         from depositor);
```

3) Find all customers who have balance greater than at least one customer located at branch Mumbai.

```
select cust_name
from depositor
```

```
where balance> some ( select balance
                      from depositor
                      where branch='mumbai');
```

4) Find customers who have balance greater than or equal to that of each customer located at Mumbai branch.

```
select cust_name
from depositor
where salary>=all ( select salary
                    from depositor
                    where branch='mumbai');
```

5) Find the customers who are borrowers from the bank and who appear in the list of account holders using EXISTS.

```
SELECT cust_name
from borrower
where exists ( select *
               from depositor
               where borrower.cust_name=depositor.cust_name);
```

6) Find the customers who are borrowers from the bank but do not hold an account using NOT EXISTS.

```
SELECT cust_name
from borrower
where not exists ( select *
                   from depositor
                  where borrower.cust_name=depositor.cust_name);
```

7) create a backup for borrower table by copying all records of borrower to a new table borrower_bkp.

```
insert into borrower_bkp
   select * from borrower;
```

Conclusion: Database is searched for various nested and correlated queries.

Experiment No-7

Aim: Perform set operations and Join operations
Theory :

SET OPERATIONS

1) UNION

The UNION operator is used to combine the result-set of two or more SELECT statements.

Notice that each SELECT statement within the UNION must have the same number of columns. The columns must also have similar data types. Also, the columns in each SELECT statement must be in the same order.

SQL UNION Syntax

SELECT *column_name(s)* FROM *table1*
UNION
SELECT *column_name(s)* FROM *table2*;

Note: The UNION operator selects only distinct values by default. To allow duplicate values, use the ALL keyword with UNION.

SQL UNION ALL Syntax

SELECT *column_name(s)* FROM *table1*
UNION ALL
SELECT *column_name(s)* FROM *table2*;

2) INTERSECT

The SQL INTERSECT clause/operator is used to combine two SELECT statements, but returns rows only from the first SELECT statement that are identical to a row in the second SELECT statement. This means INTERSECT returns only common rows returned by the two SELECT statements.

Just as with the UNION operator, the same rules apply when using the INTERSECT operator. MySQL does not support INTERSECT operator

3) EXCEPT

The SQL EXCEPT clause/operator is used to combine two SELECT statements and returns rows from the first SELECT statement that are not returned by the second SELECT statement. This means EXCEPT returns only rows, which are not available in second SELECT statement.

Just as with the UNION operator, the same rules apply when using the EXCEPT operator. MySQL does not support EXCEPT operator.

JOINS

INNER JOIN

The INNER JOIN keyword selects all rows from both tables as long as there is a match between the columns in both tables.

SQL INNER JOIN Syntax

SELECT *column_name(s)*
FROM *table1*
INNER JOIN *table2*
ON *table1.column_name=table2.column_name*;

LEFT JOIN

The LEFT JOIN keyword returns all rows from the left table (table1), with the matching rows in the right table (table2). The result is NULL in the right side when there is no match.

SQL LEFT JOIN Syntax

SELECT *column_name(s)*
FROM *table1*
LEFT JOIN *table2*
ON *table1.column_name=table2.column_name*;

RIGHT JOIN

The RIGHT JOIN keyword returns all rows from the right table (table2), with the matching rows in the left table (table1). The result is NULL in the left side when there is no match.

SQL RIGHT JOIN Syntax

SELECT *column_name(s)*
FROM *table1*
RIGHT JOIN *table2*
ON *table1.column_name=table2.column_name*;

Examples-

1) **Retrieve customer names having loan or account or both by eliminating duplicates**

```
select cust_name
from borrower
union
select cust_name
from depositor;
```

2) **Retrieve customer names having loan or account or both without eliminating duplicates**

```
select cust_name
from borrower
union all
select cust_name
from depositor;
```

3) **List customers who have placed order.**

```
select cname, order_number
from customer join order1 on order1.id=customer.customer_id;
```

4) **List all customers whether they placed order or not.**

```
select cname, order_number
from customer left outer join order1 on order1.id=customer.customer_id;
```

5) **List all order numbers whether they are ordered by customers or not.**

```
select cname, order_number
from customer right outer join order1 on order1.id=customer.customer_id;
```

Conclusion: Various queries are executed with set operations and all types of joins.

Experiment No-8

Aim: Create Views and Triggers.

Theory: A view can contain all rows of a table or select rows from a table. A view can be created from one or many tables which depend on the written SQL query to create a view. Views, which are kind of virtual tables, allow users to do the following:

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

Creating Views:

Database views are created using the CREATE VIEW statement. Views can be created from a single table, multiple tables, or another view. To create a view, a user must have the appropriate system privilege according to the specific implementation.

The basic CREATE VIEW syntax is as follows:

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

1) create a view to select employee name and address as well as the department details in which he is working group by department number.

```
create view emp_temp
as select ename, address, salary, dno,dname
from employee, department
where dno=dnumber
group by dno;
```

2) Display all records of a view.

Select * from emp_temp;

3) Delete the view emp_temp.

Drop view emp_temp;

Trigger:

A trigger is a set of actions that are run automatically when a specified change operation (SQL INSERT, UPDATE, or DELETE statement) is performed on a specified table. Triggers are useful for tasks such as enforcing business rules, validating input data, and keeping an audit trail.

A trigger is a named database object that is associated with a table, and it activates when a particular event (e.g. an insert, update or delete) occurs for the table. The statement CREATE TRIGGER creates a new trigger in MySQL.

Syntax

```
CREATE
[DEFINER = { user | CURRENT_USER }]
TRIGGER trigger_name
trigger_time trigger_event
ON tbl_name FOR EACH ROW
trigger_body
trigger_time: { BEFORE | AFTER }
trigger_event: { INSERT | UPDATE | DELETE }
```

Example

```
1) delimiter //
        create trigger depocheck
        before insert on depositor
        FOR EACH ROW
        IF NEW.salary is null
        THEN
         SET NEW.salary = 5000;
        END IF;
 //
2) delimiter //
        create trigger feed_depositor_bkp
        after insert on depositor
        FOR EACH ROW
        insert into depositor_bkp(cust_name,salary,branch) values
        (NEW.cust_name, new.salary,new.branch);
  //
3) delimiter //
```

```
        create trigger employeetrig
        before update on employee
        for each row
        if new.dno is null then
        set new.dno= 01;
        end if;
 //
4) delimiter //
        create trigger total_sal1
        after insert on employee
        for each row
        if new.dno is not null then
        update department
        set total_sal=total_sal+new.salary
        where dnumber=new.dno;
        end if;
        //

5) drop trigger feed_depositor_bkp;
```

creating user and Granting privileges to user:

```
mysql> create database your_db_name;
mysql> grant usage on *.* to your_user@localhost identified by 'your_user_password';
mysql> SELECT User FROM mysql.user;
mysql> grant all privileges on your_db_name.* to your_user@localhost ;
mysql> GRANT SELECT,INSERT,UPDATE,DELETE,CREATE,DROP ON your_db_name.*
   TO your_user@localhost ;
mysql> REVOKE ALL PRIVILEGES, GRANT OPTION FROM your_user@localhost ;
```

Conclusion: Views, trigger performed on database.

Experiment No-9

Aim: Write Functions, cursor and procedure.
Theory: A procedure (often called a stored procedure) is a subroutine like a subprogram in a regular computing language, stored in database. A procedure has a name, a parameter list, and SQL statement(s). All most all relational database system supports stored procedure, MySQL 5 introduce stored procedure.MySQL 5.6 supports "routines" and there are two kinds of routines : stored procedures which you call, or functions whose return values you use in other SQL statements the same way that you use pre-installed MySQL functions like pi(). The major difference is that UDFs can be used like any other expression within SQL statements, whereas stored procedures must be invoked using the CALL statement

Create a procedure:

```
CREATE [DEFINER = { user | CURRENT_USER }]
PROCEDURE sp_name ([proc_parameter[,...]])
[characteristic ...] routine_body
proc_parameter: [ IN | OUT | INOUT ] param_name type
type:
Any valid MySQL data type
characteristic:
COMMENT 'string'
| LANGUAGE SQL
| [NOT] DETERMINISTIC
| { CONTAINS SQL | NO SQL | READS SQL DATA
| MODIFIES SQL DATA }
| SQL SECURITY { DEFINER | INVOKER }
routine_body:
```

Pick a Delimiter : The delimiter is the character or string of characters which is used to complete an SQL statement. By default we use semicolon (;) as a delimiter. But this causes problem in stored procedure because a procedure can have many statements, and everyone must end with a semicolon. So for your delimiter, pick a string which is rarely occur within statement or within procedure. Here we have used double dollar sign i.e. $$.You can use whatever you want. To resume using ";" as a delimiter later, say "DELIMITER ; $$". See here how to change the delimiter :

mysql> DELIMITER $$;
Now the default DELIMITER is "$$". Let execute a simple SQL command :

mysql> SELECT * FROM user $$

Example : MySQL Procedure

```
mysql> DELIMITER $$ ;mysql> CREATE PROCEDURE job_data()
> SELECT * FROM JOBS; $$
```

Valid SQL routine statement

Here we have created a simple procedure called job_data, when we will execute the procedure it will display all the data from "jobs" tables.

```
mysql> DELIMITER $$ ;mysql> CREATE PROCEDURE job_data()
> SELECT * FROM JOBS; $$
```

Call a procedure

The CALL statement is used to invoke a procedure that is stored in a DATABASE. Here is the syntax :

CALL job_data()

MySQL Procedure : Parameter IN example

In the following procedure, we have used a IN parameter 'var1' (type integer) which accept a number from the user. Within the body of the procedure, there is a SELECT statement which fetches rows from 'jobs' table and the number of rows will be supplied by the user. Here is the procedure :

```
mysql> CREATE PROCEDURE my_proc_IN (IN var1 INT)
-> BEGIN
-> SELECT * FROM jobs LIMIT var1;
-> END$$
Query OK, 0 rows affected (0.00 sec)
```

To execute the first 2 rows from the 'jobs' table execute the following command :

```
mysql> CALL my_proc_in(2)$$
+---------+-------------------------------+------------+------------+
| JOB_ID  | JOB_TITLE                     | MIN_SALARY | MAX_SALARY |
+---------+-------------------------------+------------+------------+
| AD_PRES | President                     |      20000 |      40000 |
| AD_VP   | Administration Vice President |      15000 |      30000 |
+---------+-------------------------------+------------+------------+
2 rows in set (0.00 sec)Query OK, 0 rows affected (0.03 sec)
```

Now execute the first 5 rows from the 'jobs' table :

```
mysql>
CALL my_proc_in(5)$$
+------------+-------------------------------+------------+------------+
| JOB_ID     | JOB_TITLE                     | MIN_SALARY | MAX_SALARY |
+------------+-------------------------------+------------+------------+
```

```
| AD_PRES    | President                     |     20000 |     40000 |
| AD_VP      | Administration Vice President |     15000 |     30000 |
| AD_ASST    | Administration Assistant      |      3000 |      6000 |
| FI_MGR     | Finance Manager               |      8200 |     16000 |
| FI_ACCOUNT | Accountant                    |      4200 |      9000 |
+------------+-------------------------------+------------+------------+
5 rows in set (0.00 sec)Query OK, 0 rows affected (0.05 sec)
```

MySQL Procedure : Parameter OUT example

The following example shows a simple stored procedure that uses an OUT parameter. Within the procedure MySQL MAX() function retrieves maximum salary from MAX_SALARY of jobs table.

```
mysql> CREATE PROCEDURE my_proc_OUT (OUT highest_salary INT)
-> BEGIN
-> SELECT MAX(MAX_SALARY) INTO highest_salary FROM JOBS;
-> END$$
Query OK, 0 rows affected (0.00 sec)
```

In the body of the procedure, the parameter will get the highest salary from MAX_SALARY column. After calling the procedure the word OUT tells the DBMS that the value goes out from the procedure. Here highest_salary is the name of the output parameter and we have passed its value to a session variable named @M, in the CALL statement.

```
mysql> CALL my_proc_OUT(@M)$$
Query OK, 1 row affected (0.03 sec)

mysql< SELECT @M$$+-------+
| @M    |
+-------+
| 40000 |
+-------+
1 row in set (0.00 sec)
```

MySQL Procedure : Parameter INOUT example

The following example shows a simple stored procedure that uses an INOUT parameter and an IN parameter. The user will supply 'M' or 'F' through IN parameter (emp_gender) to count a number of male or female from user_details table. The INOUT parameter (mfgender) will return the result to a user. Here is the code and output of the procedure :

```
mysql> CALL my_proc_OUT(@M)$$Query OK, 1 row affected (0.03 sec)mysql> CREATE
PROCEDURE my_proc_INOUT (INOUT mfgender INT, IN emp_gender CHAR(1))
-> BEGIN
```

```
-> SELECT COUNT(gender) INTO mfgender FROM user_details WHERE gender = emp_gender;
-> END$$
Query OK, 0 rows affected (0.00 sec)
```

Now check the number of **male** and **female** users of the said tables :

```
mysql> CALL my_proc_INOUT(@C,'M')$$
Query OK, 1 row affected (0.02 sec)

mysql> SELECT @C$$
+------+
| @C   |
+------+
|    3 |
+------+
1 row in set (0.00 sec)

mysql> CALL my_proc_INOUT(@C,'F')$$
Query OK, 1 row affected (0.00 sec)

mysql> SELECT @C$$
+------+
| @C   |
+------+
|    1 |
+------+
1 row in set (0.00 sec)
```

<u>MYSQL: Creating stored function:</u>

The CREATE FUNCTION statement is used for creating a stored function and user-defined functions. A stored function is a set of SQL statements that perform some operation and return a single value.

Just like Mysql in-built function, it can be called from within a Mysql statement.

By default, the stored function is associated with the default database.

The CREATE FUNCTION statement require CREATE ROUTINE database privilege.

Syntax:

The syntax for CREATE FUNCTION statement in Mysql is:

```
CREATE FUNCTION function_name(func_parameter1, func_parameter2, ..)
      RETURN datatype [characteristics]
          func_body
```

Example:

```
DELIMITER //
```

```
CREATE FUNCTION no_of_years(date1 date) RETURNS int DETERMINISTIC
BEGIN
 DECLARE date2 DATE;
  Select current_date()into date2;
  RETURN year(date2)-year(date1);
END
//
DELIMITER ;
```

Calling of above function:
Select emp_id, fname, lname, no_of_years(start_date) as 'years' from employee;

MySQL: Cursors

A database cursor is a control structure that enables traversal over the records in a database. Cursors are used by database programmers to process individual rows returned by database system queries. Cursors enable manipulation of whole result sets at once. In this scenario, a cursor enables the rows in a result set to be processed sequentially. In SQL procedures, a cursor makes it possible to define a result set (a set of data rows) and perform complex logic on a row by row basis. By using the same mechanics, an SQL procedure can also define a result set and return it directly to the caller of the SQL procedure or to a client application.

MySQL supports cursors inside stored programs. The syntax is as in embedded SQL. Cursors have these properties :
- Asensitive: The server may or may not make a copy of its result table
- Read only: Not updatable
- Nonscrollable: Can be traversed only in one direction and cannot skip rows

To use cursors in MySQL procedures, you need to do the following :
- Declare a cursor.
- Open a cursor.
- Fetch the data into variables.
- Close the cursor when done.

Declare a cursor:

The following statement declares a cursor and associates it with a SELECT statement that retrieves the rows to be traversed by the cursor.

DECLARE cursor_name

CURSOR FOR select_statement

Open a cursor:

The following statement opens a previously declared cursor.

OPEN cursor_name

Fetch the data into variables :

This statement fetches the next row for the SELECT statement associated with the specified cursor (which must be open) and advances the cursor pointer. If a row exists, the fetched columns are stored in the named variables. The number of columns retrieved by the SELECT statement must match the number of output variables specified in the FETCH statement.

```
FETCH [[NEXT] FROM] cursor_name
INTO var_name [, var_name] ...
```

Close the cursor when done :

This statement closes a previously opened cursor. An error occurs if the cursor is not open.

CLOSE cursor_name

Example:

The procedure starts with three variable declarations. Incidentally, the order is important. First, declare variables. Then declare conditions. Then declare cursors. Then, declare handlers. If you put them in the wrong order, you will get an error message.

```
DELIMITER $$
CREATE PROCEDURE my_procedure_cursors(INOUT return_val INT)
BEGIN
DECLARE a,b INT;
DECLARE cur_1 CURSOR FOR
SELECT max_salary FROM jobs;
DECLARE CONTINUE HANDLER FOR NOT FOUNDSET b = 1;
OPEN cur_1;REPEATFETCH cur_1 INTO a;
UNTIL b = 1END REPEAT;
CLOSE cur_1;
SET return_val = a;
END;
$$
```

Now execute the procedure:

```
mysql>
CALL my_procedure_cursors(@R);
```

```
Query OK, 0 rows affected (0.00 sec)

mysql> SELECT @R;
+-------+
| @R    |
+-------+
| 10500 |
+-------+
1 row in set (0.00 sec)
```

Conclusion: Procedure and functions are implemented with cursor.

Experiment No-10

Aim: Implement Transaction and Concurrency control techniques

Theory:

A transaction is a logical unit of processing in a DBMS which entails one or more database access operation. In a nutshell, database transactions represent real-world events of any enterprise.

All types of database access operation which are held between the beginning and end transaction statements are considered as a single logical transaction. During the transaction the database is inconsistent. Only once the database is committed the state is changed from one consistent state to another.

Concurrency control is the procedure in DBMS for managing simultaneous operations without conflicting with each another. Concurrent access is quite easy if all users are just reading data. There is no way they can interfere with one another. Though for any practical database, would have a mix of reading and WRITE operations and hence the concurrency is a challenge.

Concurrency control is used to address such conflicts which mostly occur with a multi-user system. It helps you to make sure that database transactions are performed concurrently without violating the data integrity of respective databases.

Therefore, concurrency control is a most important element for the proper functioning of a system where two or multiple database transactions that require access to the same data, are executed simultaneously.

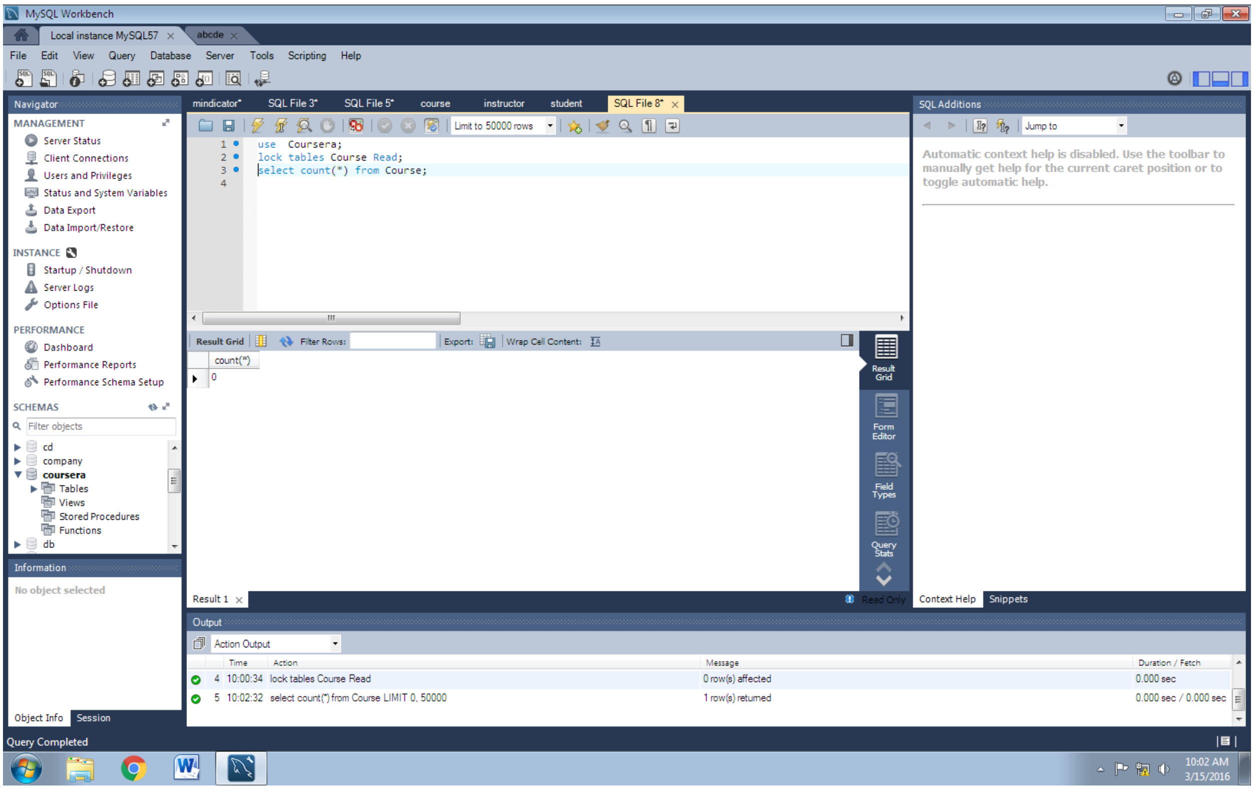
Concurrency Control Protocols

Different concurrency control protocols offer different benefits between the amount of concurrency they allow and the amount of overhead that they impose.

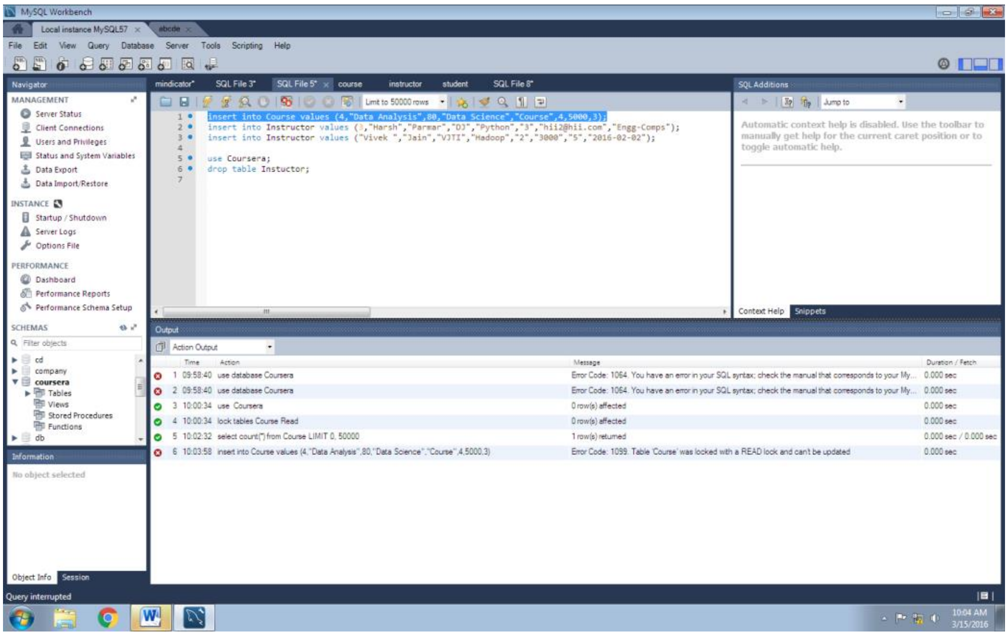
- Lock-Based Protocols
- Two Phase
- Timestamp-Based Protocols
- Validation-Based Protocols

Concurrency control using Mysql Locks:

User 1

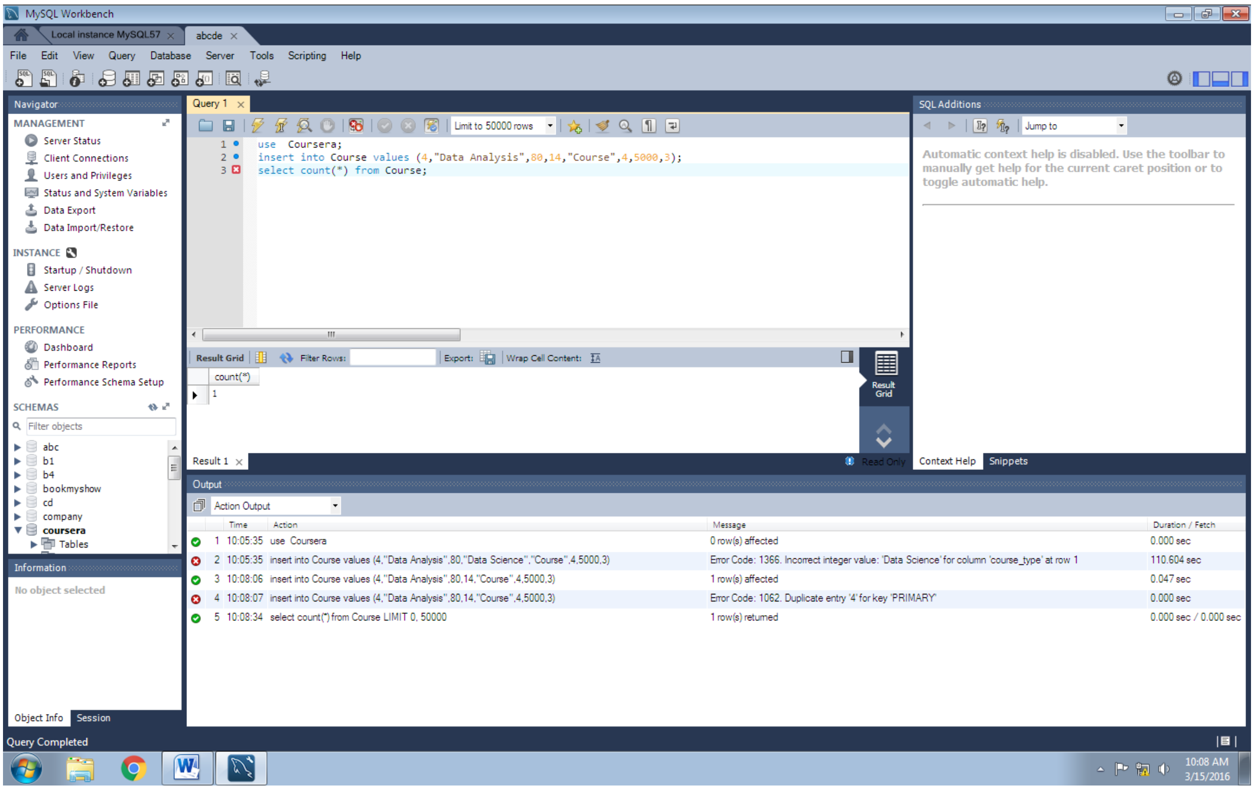


insert into Course values (4,"Data Analysis",80,"Data Science","Course",4,5000,3);

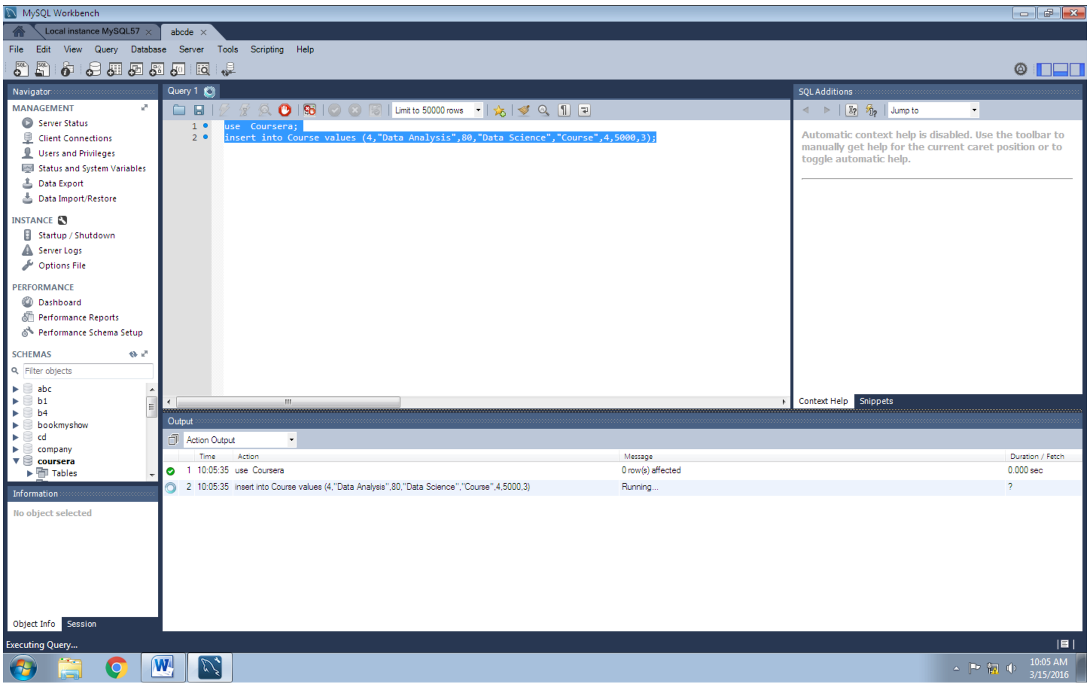


By user 2

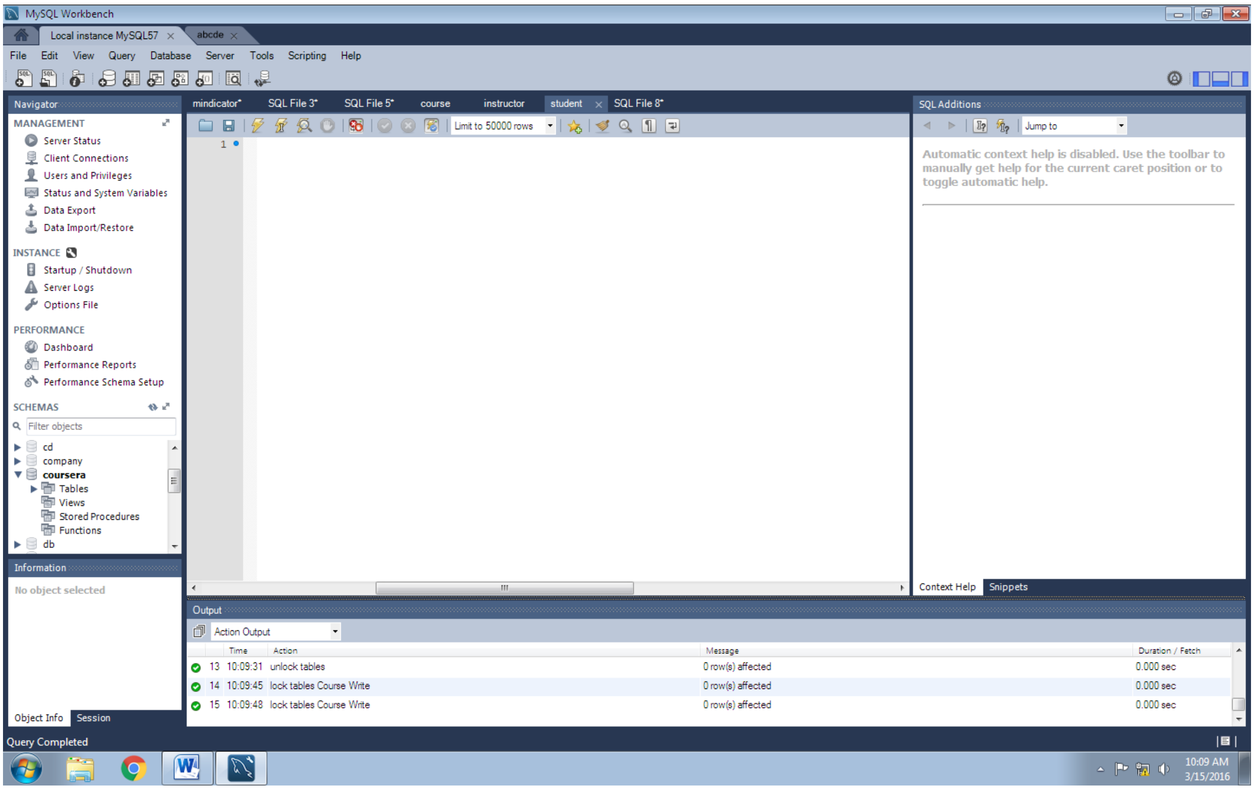
User 2 trying to read



User 2 trying to write into locked table

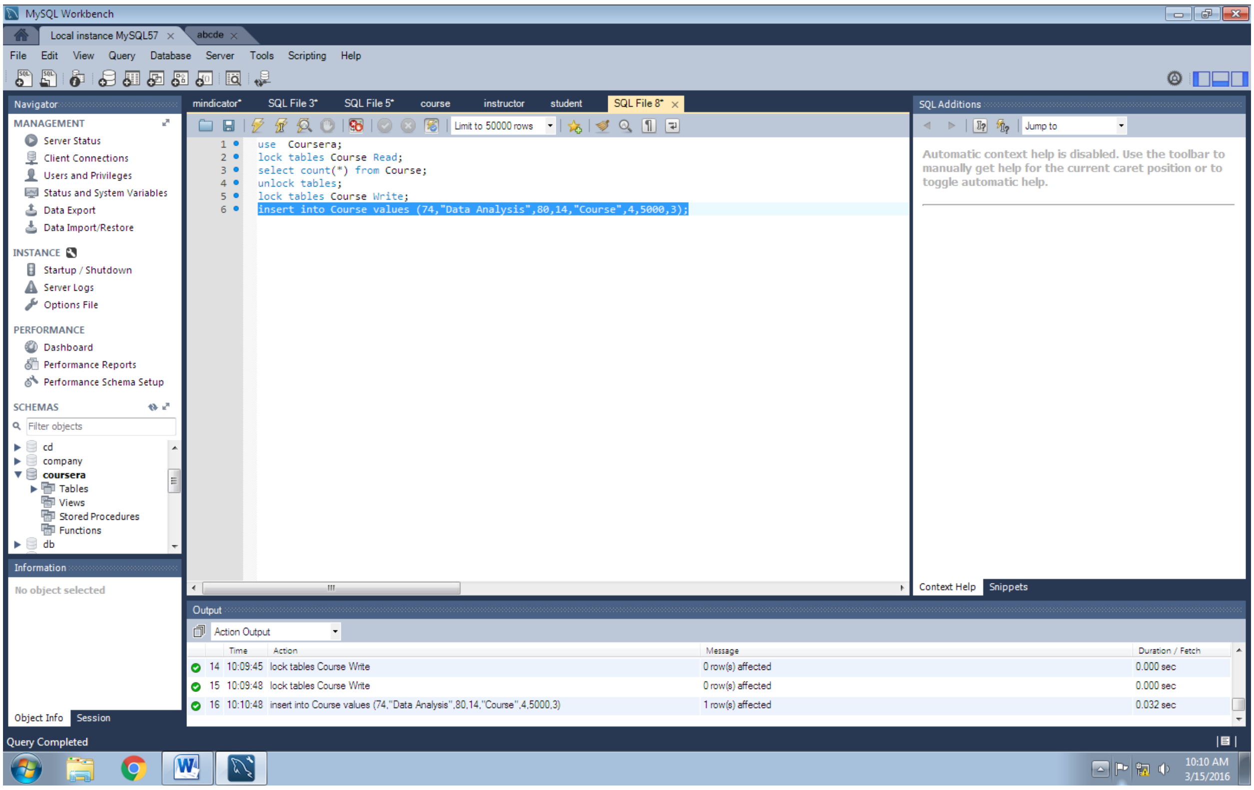


User 1 implementing write lock

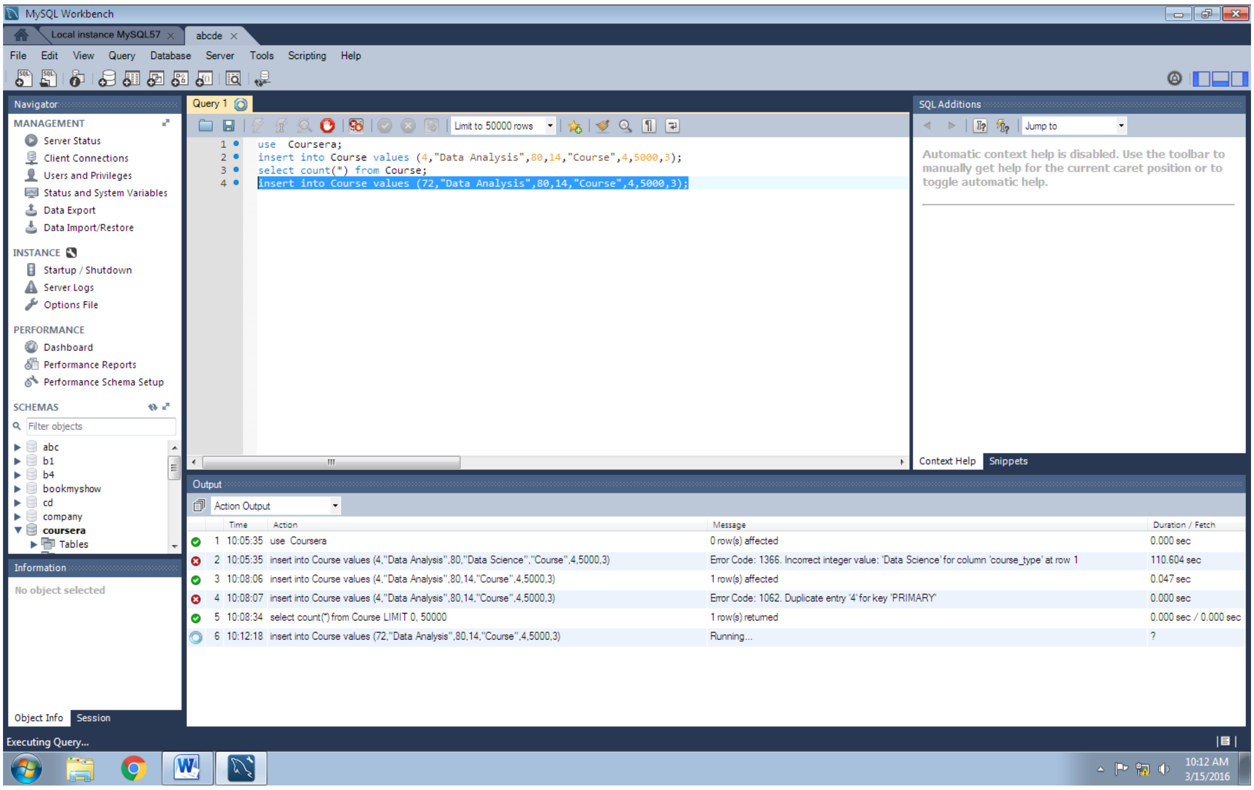


User 1 allowed to write

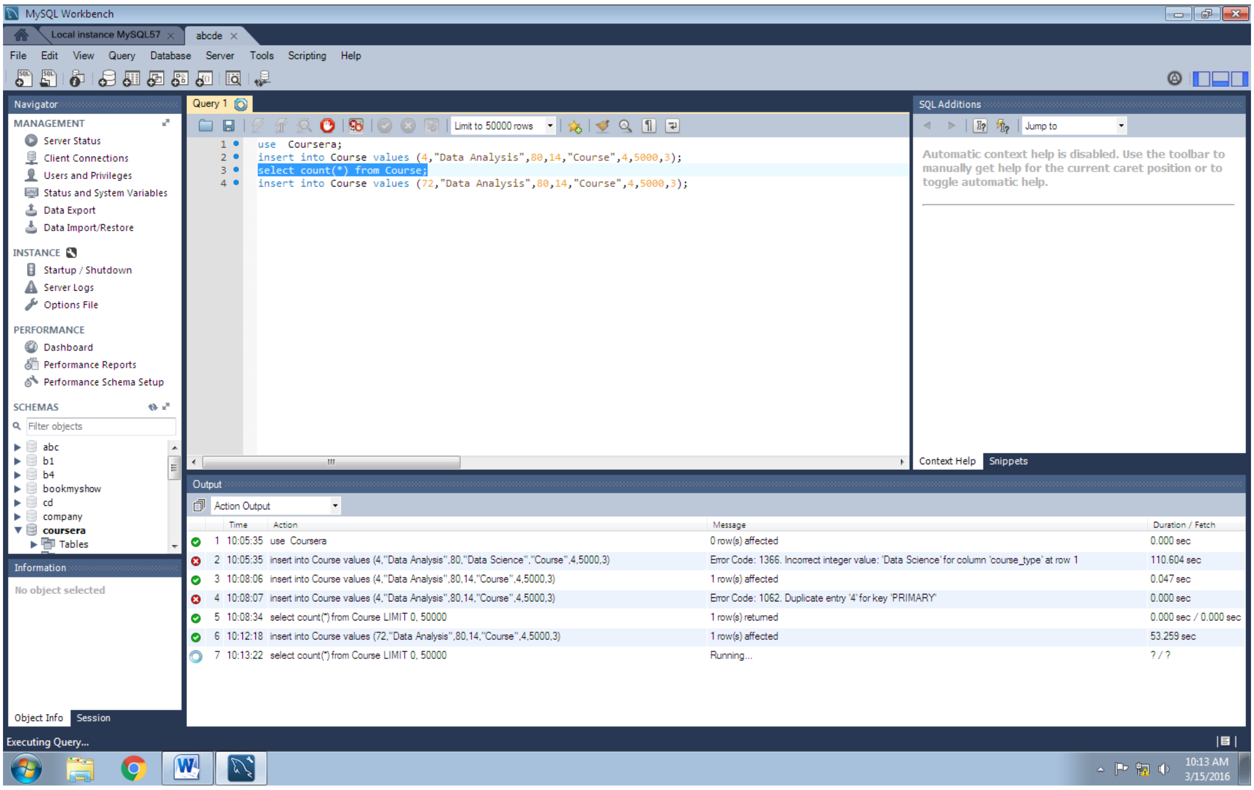
insert into Course values (74,"Data Analysis",80,14,"Course",4,5000,3);



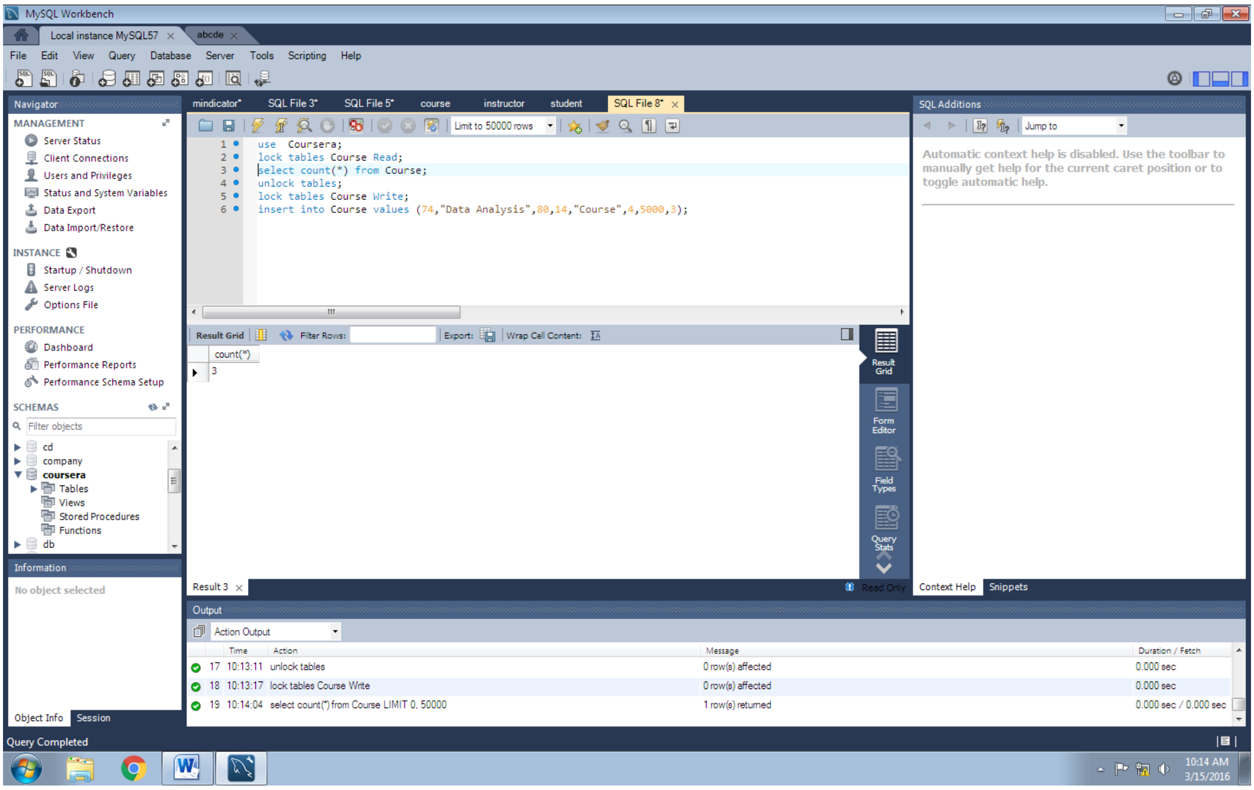
User 2 trying to write on write lock by user1



User 2 trying to read on write lock by user1



User 1 reading on its write lock



Conclusion: Locks of mysql is used to demonstrate concurrency.