17

Table Creation and Data Manipulation Commands

In This Chapter

17.1 INTRODUCTION

Uptil now, you have learnt quite a lot about working in MySQL. So far, you have learnt about writing simple SQL statements and using MySQL functions in queries. This chapter is going to introduce many new and useful commands of SQL. In this chapter we shall be discussing some more commands that let you create tables, add data in tables, modify/change data in tables, delete tuples from tables etc. Let us begin with table creation command– CREATE TABLE.

17.2 DATABASES IN MYSQL

Before you start creating tables, you need a database, which is the container of tables. The database acts as a central point of administration for the tables in the database. The actual data is stored in the tables, which provide a structured organization for the data and maintain the integrity of that data. To create tables, either you can create a new database or use an existing database. MySQL provides an empty database by the name **test**, which you can use to store your tables in. But it is recommended that you create new database before you create a group of some tables pertaining to a specific application. In the coming lines, you are going to learn how you can create databases in MySQL.

The first step in setting up a MySQL database is to create the actual database object, which serves as a container for the tables in that database.

17.2.1 Creating Databases

Creating databases is an easier task relatively. In simplest form the *Create Database* command takes the following syntax.

SYNTAX `CREATE DATABASE [IF NOT EXISTS] <database name> ;`

The IF NOT EXISTS clause, if used, will first test whether a database by the mentioned name already exists or not. If it does, then create database command is simply ignored, otherwise a database with the mentioned name is created.

Following are some example database creation commands :

```
CREATE DATABASE myDB ;
```
Creates database having name as MyDB

```
CREATE DATABASE IF NOT EXISTS myDB ;
```
Creates a database having name as MyDB, if there is no database by the name MyDB already existing

17.2.2 Opening Databases

Creating database is not enough. Before you create tables in it, you need to open the database. To open a database, you simply need to write the statement as per following syntax.

SYNTAX `USE <databasename> ;`

For example, to open **myDB** database that was created just before, you need to write :

```
USE myDB ;
```

The only thing you need to ensure before opening a database is that it must already exist *i.e.*, it must be already created. To check the names of existing databases, you may write following command :

```
SHOW DATABASES ;
```

17.2.3 Removing Databases

Sometimes, you need to remove a database when you don't need it anymore. But before making this decision, do make sure that you don't need data stored in different tables of the database. This is because, when you drop a database, all its tables also get removed along with the database.

To remove a database, you need to issue a command with following syntax.

SYNTAX `DROP DATABASE<database name> ;`

That is, to drop a database namely **myDB**, you 'll be writing :

```
DROP DATABASE myDB ;
```

Now that you know about how to create and use databases, you can move on to creating tables and adding data to them.

17.3 CREATING TABLES

Tables are defined with the CREATE TABLE command. When a table is created, its columns are named, data types and sizes are supplied for each column. Each table must have at least one column.

The syntax of CREATE TABLE command is :

SYNTAX
```
CREATE TABLE <table-name>
    (<column name> <data type> [ (<size>)],
    <columnname> <data type> [ ( <size> ) ... ]) ;
```

To create an *employee* table whose schema is as follows :

```
employee (ecode, ename, sex, grade, gross)
```

the *SQL* command will be

```
CREATE TABLE employee
(   ecode    integer,
    ename    char(20),
    sex      char(1),
    grade    char(2),
    gross    decimal ) ;
```

NOTE

Before issuing a CREATE TABLE command, make assure that its parent database has been opened using USE <database name> command.

When you create a table, you can place constraints on the values that can be entered into its fields. If this is specified, *SQL* will reject any values (entered / changed through INSERT/UPDATE command) that violate the criteria you define. Let us learn about constraints in details.

17.3.1 Data Integrity Through Constraints

A *constraint*, in general, refers to a condition or a check that is applied to a column (field) or set of columns in a table. The *constraints* applied to maintain data integrity are also known as ***integrity constraints***.

Once an integrity constraint is enabled, all data in the table must confirm to the rule that it specifies. Any subsequent SQL statement, that tries to enter or modify data in the table, is successfully carried out if and only if the data being entered or modified satisfies the integrity constraints.

CONSTRAINT

A **Constraint** is a condition or check applicable on a field or set of fields.

The two basic types of constraints are ***Column constraints*** and ***Table constraints***. The difference between the two is that *column constraints* apply only to individual columns, whereas *table constraints* apply to groups of one or more columns. The following is the syntax for the CREATE TABLE command, expanded to include constraints :

SYNTAX
```
CREATE TABLE <table name>
(   <column name><data type> [ ( <size> ) ] <column constraint>,
    <column name><data type> [ ( <size> ) ] <column constraint> ...
    <table constraint> (<column name>, [, <column name> ... ] ) ...) ;
```

The fields given in parenthesis after the table constraint(s) are the fields to which they apply. The column constraints apply to the columns whose definitions they follow.

For example, if you write the keywords NOT NULL immediately after the data type (and size) of a column, this means the column can never have empty values (*i.e.*, NULL[1] values). Otherwise *SQL* will assume that NULLs are permitted.

1. NULL is a special keyword in *SQL* that depicts an empty value. A column having NULL is not empty but stores an empty value. Two NULLs cannot be added ; subtracted or compared.

Consider the following *SQL* command.

```
CREATE TABLE employee
    (   ecode    integer      NOT NULL,
        ename    char (20)    NOT NULL,
        sex      char (1)     NOT NULL,
        grade    char (2),
        gross    decimal  ) ;
```

The above command creates a table called *employee* in which *ecode* column (integer type) can never be empty as its definition is followed by keywords NOT NULL. Similarly, the columns *ename* (*char (20)*) and *sex* (*char (1)*) can never have NULL values. Any attempt to put NULL values in these columns will be rejected.

17.3.1A Different Constraints

These constraints ensure database integrity, thus are sometimes called *database integrity constraints*. A few of them are :

- *Unique* constraint
- *Primary key* constraint
- *Default* constraint
- *Check* constraint
- *Foreign key* constraint

MySQL also supports some other constraints such as **ENUM** and **SET** constraints that are used to define columns that can contain only a given set of values. We are not going to talk about these constraints here as their coverage is beyond the scope of this book.

1. Unique Constraint

This constraint ensures that no two rows have the same value in the specified column(s). For example, UNIQUE constraint applied on *ecode* of *employee* table ensures that no rows have the same *ecode* value, as shown below.

```
CREATE TABLE employee
(   ecode    integer      NOT NULL UNIQUE,
    ename    char (20)    NOT NULL,
    sex      char (1)     NOT NULL,
    grade    char (2),
    gross    decimal ) ;
```

NOTE

Please note that a column that is specified as a primary key must also be unique. At the same time, a column that's unique may or may not be a primary key. In addition, multiple UNIQUE constraints can be defined on a table.

After the application of this constraint, all subsequent SQL statements are executed ensuring that the condition posed by this constraint is met. In other words, the values being entered or modified must ensure that the column *ecode's* value is not already existing in the table. Figure 17.1 illustrates this very concept. This constraint when applied to columns ensures that there cannot exist more than one NULL value in the column.

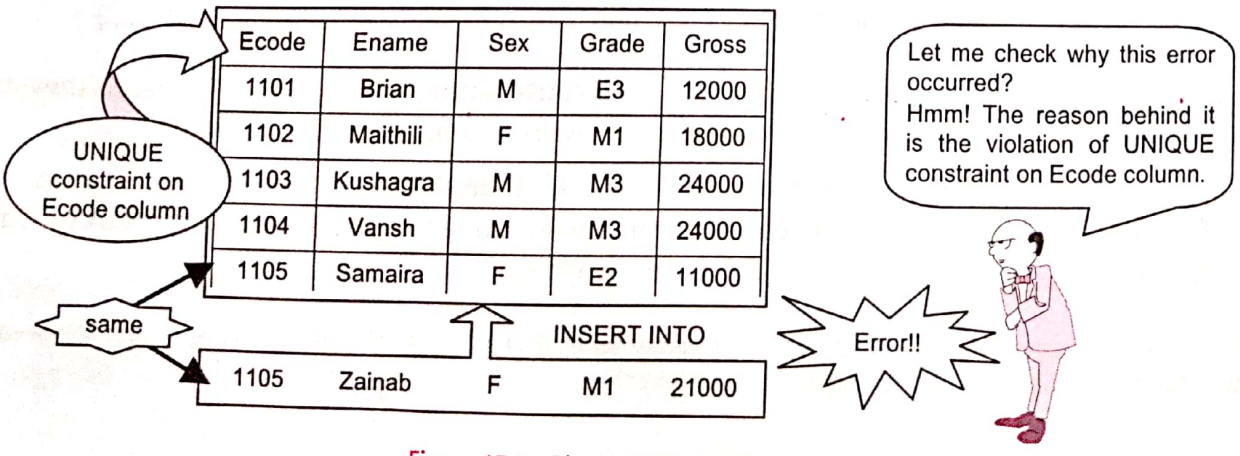

Fig. 17.1 The UNIQUE Constraint.

2. Primary Key Constraint

This constraint declares a column as the primary key of the table. This constraint is similar to unique constraint except that only one column (or one group of columns) can be applied in this constraint. The primary keys cannot allow NULL values, thus, this constraint must be applied to columns declared as NOT NULL. Consider the following *SQL* statement :

```
CREATE TABLE employee
(    ecode     integer      NOT NULL PRIMARY KEY,
     ename     char (20)    NOT NULL,
     sex       char (1)     NOT NULL,
     grade     char (2),
     gross     decimal ) ;
```

NOTE

There are differences between UNIQUE and PRIMARY KEY constraints. Though both ensure unique values for each row in a column, but UNIQUE allows NULL values whereas PRIMARY KEY does not. Also, there can exist multiple columns with UNIQUE constraints in a table, but there can exist only one column or one combination with PRIMARY KEY constraint.

3. Default Constraint

A default value can be specified for a column using the DEFAULT clause. When a user does not enter a value for the column (having default value), automatically the defined default value is inserted in the field. Consider the following *SQL* statement.

```
CREATE TABLE employee
(    ecode     integer      NOT NULL PRIMARY KEY,
     ename     char (20)    NOT NULL,
     sex       char (1)     NOT NULL,
     grade     char (2)     DEFAULT 'E1',
     gross     decimal) ;
```

According to above command, if no value is provided for *grade*, the default value of 'E1', will be entered. The datatype of the default value has to be compatible with the datatype of the column to which it is assigned. Insertion of NULL (as default value) is possible only if the column definition permits. (NOT NULL columns cannot have NULL as default). *A column can have only one default value.*

4. Check Constraint

This constraint limits values that can be inserted into a column of a table. For instance, consider the following *SQL* statement :

```
CREATE TABLE employee
(    ecode     integer      NOT NULL PRIMARY KEY,
     ename     char (20)    NOT NULL,
     sex       char (1)     NOT NULL,
     grade     char (2)     DEFAULT 'E1',
     gross     decimal      CHECK (gross > 2000) ) ;
```

This statement ensures that the value inserted for *gross* must be greater than 2000.

When a check constraint involves more than one column from the same table, it is specified after all the columns have been defined. For instance,

```
CREATE TABLE items
(    icode     char (5) NOT NULL PRIMARY KEY,
     descp     char (20)    NOT NULL,
     rol       integer,
     qoh       integer,
     CHECK     (ROL < QOH ) ) ;
```

This statement compares two columns *ROL* and *QOH*, thus, these two columns must be defined before this CHECK constraint. Check constraint can consist of :

- A list of constant expressions specified using IN. For example,

```
descp char (20) CHECK
(descp IN ('NUT', 'BOLT', 'SCREW', 'WRENCH', 'NAIL') )
```

- Range of constant expressions specified using BETWEEN. The upper and lower boundary values are included in the range. For example :

```
price decimal CHECK (price BETWEEN 253.00 and 770.00)
```

- A pattern specified using LIKE. For example, if we give a date in *Char* format, then we may check :

```
ordate char (10) NOT NULL CHECK (ordate LIKE '- -/- -/- - - -')
```

- Multiple conditions using OR, AND etc. For example,

```
CHECK    (    (discount = 0.15 AND city = 'HISSAR') OR
              (discount = 0.13 AND city = 'JAIPUR') OR
              (discount = 0.17 AND city = 'MOHALI')    )
```

For example, in the following CREATE TABLE statement,

```
CREATE TABLE Customer
(SID integer CHECK (SID > 0),
Last_Name varchar (30),
First_Name varchar(30) );
```

Column *SID* has a constraint — *its value must only include integers greater than 0*. So, attempting to execute the following statement,

```
INSERT INTO Customer VALUES ('-3','Gonzalves','Linda');
```

should result in an error because the values for SID must be greater than 0. But you will find that your query is executed without checking the condition specified through CHECK constraint. *The reason being* MySQL ignores CHECK constraint internally.

NOTE

Please note that MySQL does not enforce the CHECK constraint although MySQL syntax allows the definition of CHECK constraint. As per MySQL documentation, 'Check constraint gets parsed but IGNORED'.

5. Foreign Key Constraint

In an RDBMS, tables reference one another through common fields and to ensure validity of references, referential integrity is enforced. Referential integrity is a system of rules that a DBMS uses to ensure that relationships between records in related tables are valid, and that users don't accidentally delete or change related data. Referential integrity is ensured through FOREIGN KEY constraint. This is implemented as explained in the following paragraph.

Whenever two tables are related by a common column (or set of columns), then the related column(s) in the **parent table** (or **primary table**) should be either declared a PRIMARY KEY or UNIQUE key and the related column(s) in the **child table** (or **related table**) should have FOREIGN KEY constraint. For instance, if we have following two tables :

```
Items (Itemno, Description, Price, QOH)
Orders (Orderno, Orderdate, Itemno, Qty)
```

Underlined columns indicate primary key

Both the tables are related through common column **Itemno**. The column **Itemno** is ***primary key*** in *parent table* **Items** and it should be declared ***foreign key*** in *child table* **Orders** to enforce referential integrity *i.e.*, both the tables should be created as follows :

```
CREATE TABLE Items
(    Itemno char(5) NOT NULL PRIMARY KEY,
       ⋮
);
CREATE TABLE Orders                                    child table.
(    Orderno Number (6, 0) NOT NULL PRIMARY KEY,
     ⋮
     Itemno char(5) REFERENCES Items (Itemno),         primary key of parent table.
     ⋮
);
          foreign key
                                  parent table's name
```

Notice that the related column ***Itemno*** in child table **Orders** has been declared *foreign key* through REFERENCES clause. To enforce a foreign key constraint at column level, the syntax to be followed is :

SYNTAX

```
columnname datatype (size) REFERENCES tablename
[(columnname)] [ON DELETE CASCADE] [ON UPDATE CASCADE][2]
```

Here the first *columnname* is the name of related column in *child table* and the *columnname* appearing after REFERENCES clause is the name of related column in the *parent table*. The *tablename* after REFERENCES clause is the name of *parent table* or *primary table* to which the column in current table is related.

- If you skip the related *columnname* in parent table, then, by default MySQL will reference the primary key of *parent table*.
- If you write ON DELETE CASCADE also, while defining foreign key constraint then in case a row or a tuple is deleted in *parent table*, all its related tuples or rows in the *child table* will automatically be deleted.
- If you write ON UPDATE CASCADE also, while defining foreign key constraint then in case a row or a tuple is updated in *parent table*, all its related tuples or rows in the *child table* will automatically be updated.

The foreign key constraint can also be applied through FOREIGN KEY table constraint, as shown below :

```
CREATE TABLE Orders ( OrderID SMALLINT NOT NULL PRIMARY KEY,
                      ModelID SMALLINT NOT NULL,
                      ModelDescrip VARCHAR(40),
                      SupplierID SMALLINT NOT NULL,
                      FOREIGN KEY (ModelID) REFERENCES Models (ModelID)
                      ON DELETE CASCADE ON UPDATE CASCADE,
                      FOREIGN KEY (SupplierID) REFERENCES Suppliers (SupplierID)
                      ON DELETE CASCADE ON UPDATE CASCADE      );
```

ModelID and SupplierID are being defined as foreign keys

2. You can also use **RESTRICT | SET NULL | NO ACTION** in place of **CASCADE**. **RESTRICT** will reject the operation ; NO ACTION will not allow deletion or updation of a primary key as long as there is a foreign key dependent on it ; SET NULL will set NULL in the foreign key column.

Role of FOREIGN KEY/REFERENCES constraint

- It results into the rejection of INSERT (statement used to insert new data) or UPDATE (statement used to modify existing data) if a corresponding value does not currently exist in the *primary table*. (*See* Fig. 17.2)
- If ON DELETE CASCADE option has been set then upon deleting a tuple in the *parent table*, all its related records in the *child table* get deleted. In the absence of this clause, it rejects the deletion operation if corresponding records in the *child table* exist.
- The foreign key column in the *child table* must reference a PRIMARY KEY or UNIQUE column in the *parent table*.
- Both the related columns (in parent table and in child table) should have the same data type.
- May reference to the same table in which it is being defined *i.e.*, it may reference the same table name as in the CREATE TABLE command.

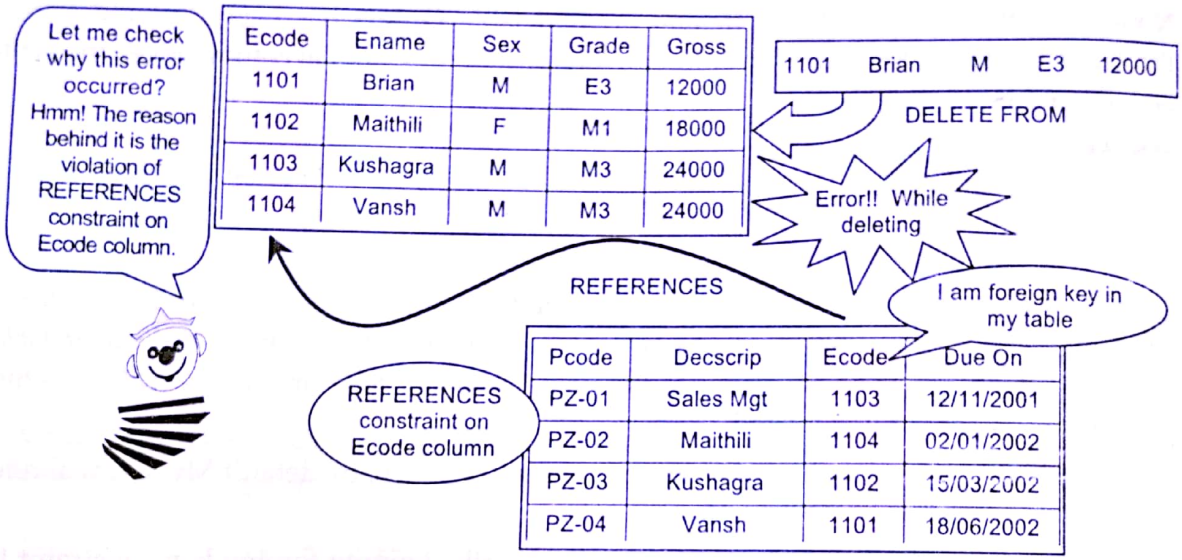

Figure 17.2 FOREIGN KEY/REFERENCE Constraint.

Important Foreign Key and Storage Engine

MySQL is capable of creating databases in many different storage engines that offer different features. The default storage engine is ISAM but ironically it does not support foreign keys. So, if you find that your foreign key constraint are being ignored or not implemented, you need to change the storage engine of your database tables to InnoDB storage engine.

To do so you simply need to write following command for your connecting tables :

```
ALTER TABLE <tablename> ENGINE = innodb ;
```

e.g.,

```
ALTER TABLE table1 ENGINE = innodb ;
```

Once done, you can create and implement foreign keys for your tables.

17.3.1B Applying Table Constraints

When a constraint is to be applied on a group of columns of the table, it is called ***table constraint***. The table constraints appear in the end of table definition. For instance, if you want combination of *icode* and *descp* of table *items* to be unique, you may write it as follows :

```
CREATE TABLE items
(   icode    char (5)      NOT NULL,
    descp    char (20)     NOT NULL,
    ROL      integer,
    QOH      integer,
    CHECK    (ROL < QOH),
    UNIQUE   (icode, descp) ) ;
```

these are a table constraints

The above statement ensures that the combination of *icode* and *desc* in each row must be unique.

Similarly, if you want to define primary key that contains more than one column, you can use PRIMARY KEY table constraint. For instance, if you want to declare a primary key for the table *members* as the combination of columns *firstname* and *lastname*, it can be done as follows :

```
CREATE TABLE members
(   firstname        char (15)  NOT NULL,
    lastname         char (15)  NOT NULL,
    city             char (20),
    PRIMARY KEY (firstname, lastname) ) ;
```

this is a table constraint

To define foreign key constraint for a group of columns, FOREIGN KEY table constraints should be used. For example, if a group of columns say ***orderno*** and ***items***, is to be declared foreign key, it should be done as follows :

```
CREATE TABLE Orders
(   Orderno Number(6, 0) NOT NULL,
        ⋮
    Itemno char(5),
        ⋮
    FOREIGN KEY (Orderno, Itemno) REFERENCES OrdMaster (OrderNO, ItemNO)
        ⋮
) ;
```

17.3.1C Assigning Names to Constraints

By default, MySQL assigns a unique name to each constraint defined by you. MySQL names constraints as :

```
SYS_Cn
```

where *n* is an integer that makes the constraint name unique. For instance, SYS_C003217, SYS_C001592 etc. are constraint names generated by MySQL. However, you, yourself, can name a constraint created by you. This can be done as per following syntax :

SYNTAX `CONSTRAINT <name-of-constraint> <definition-of-constraint>`

For example, consider the following two CREATE TABLE statements :

```
CREATE TABLE Items
    (   Itemno char(5) CONSTRAINT p_Itemkey PRIMARY KEY,
            ⋮
    ) ;
CREATE TABLE Orders
    (   Orderno  ,
            ⋮    ,
        CONSTRAINT Group_fkey_Orders FOREIGN KEY (Orderno, Itemno)
        REFERENCES OrdMaster (Order#, Item#)   ) ;
```

EXAMPLE 17.1 *Create table* ***Employee*** *with the following structure :*

Name of Column	ID	First_Name	Last_Name	User_ID	Salary
Type	Number(4)	Varchar(30)	Varchar(30)	Varchar(10)	Number(9,2)

Ensure the following specification in created table :

- *ID should be declared as Primary Key*
- *User_ID should be unique*
- *Salary must be greater than 5000.*
- *First_Name & Last_Name must not remain blank.*

Solution.

```
CREATE TABLE Employee
(   ID Number(4) NOT NULL PRIMARY KEY,
    First_Name Varchar(30) NOT NULL,
    Last_Name Varchar(30) NOT NULL,
    User_ID Varchar(10) UNIQUE,
    Salary Number (9,2) CHECK (Salary > 5000)    ) ;
```

MySQL responds as : `Query OK, 0 rows affected`

EXAMPLE 17.2 *Create another table* ***Job*** *with following structure :*

Name of column	*Type*
Job_id	Number(4)
Job_des	Varchar(30)
Alloc_on	Date
Due_on	Date
Emp_id	Number(4)

Ensure the following specifications in the table :

- *Job_id* is the primary key
- *Job_des, Alloc_on, Due_on* cannot be left blank.
- *Emp_id* is foreign key here that is related to *ID* column of earlier created table *Employee*.

Solution.

```
CREATE TABLE Job
(   Job_id Number(4) NOT NULL PRIMARY KEY,
    Job_Des Varchar(30) NOT NULL,
    Alloc_on Date NOT NULL,
    Due_on Date NOT NULL,
    Emp_id Number(4) REFERENCES Employee (ID) ) ;
```

MySQL responds as : `Query OK, 0 rows affected`

NOTE

If you want to view the CREATE TABLE statement of an existing table, you may write : **SHOW CREATE TABLE <tablename> ;**.

17.3.1D Viewing a Table Structure

Once you have created a table, you may want to view its structure. Anytime, if you want to view an already created/existing table's structure, you may use DESC[RIBE] command of MySQL * Plus.

The syntax of this command is as follows :

SYNTEAX `DESC[RIBE] <tablename> ;`

For example, if you write

```
DESC Pet ;
```

or

```
DESCRIBE Pet ;
```

MySQL will show you the complete structure of table named PET as shown below :

```
+---------+-------------+--------+---------+---------+---------+
| Field   | Type        | Null   | Key     | Default | Extra   |
+---------+-------------+--------+---------+---------+---------+
| name    | varchar(20) | YES    |         | NULL    |         |
| owner   | varchar(20) | YES    |         | NULL    |         |
| species | varchar(20) | YES    |         | NULL    |         |
| sex     | char(1)     | YES    |         | NULL    |         |
| birth   | date        | YES    |         | NULL    |         |
| death   | date        | YES    |         | NULL    |         |
+---------+-------------+--------+---------+---------+---------+
6 rows in set (0.08 sec)
```

Once you have created tables in your database, you can check the tables' names in the database. To check what all tables are there in database, simply write :

```
SHOW TABLES ;
```

it will show you the names of all the tables in the database.

17.3.1E Creating Table from Existing Table

You can define a table and put data into it without going through the usual data definition process. This can be done by using SELECT statement with CREATE TABLE.

The new table stores the result produced by the SELECT statement. The name of the new table must be unique. Following query illustrates this :

```
CREATE TABLE orditem AS
(    SELECT icode, descp
     FROM items
     WHERE QOH < ROL
);
```

NOTE

Alternatively, you may also write : **SHOW COLUMNS FROM <tablename> ;** to view column details of a table.

This will create a new table called *orditem* that stores two columns : *itemcode* and *description* for the items that have their *QOH* less than *ROL* in the relation ***items***.

The newly created table **Orditem** will look as it is shown below :

Table : *Orditem*

Icode	Descp
I06	Cream Roll

The table *Orditem* has derived its contents from table **items**.

If you do not specify the WHERE clause, *icode* & *descp* from all rows of the *items* relation will be copied into *Orditem table*.

The CREATE TABLE <table> AS SELECT... is useful for creating test tables, new tables as copies of existing tables, and for making smaller tables out of large tables.

PriS CREATING TABLES, INSERTING DATA IN TABLES — Progress in SQL 17.1 & 17.2

Please check the practical component-book – Progress in Computer Science with Python and fill it there in PriS 17.1, 17.2 under Chapter 17 after practically doing it on the computer.

>>>❖<<<

17.4 CHANGING DATA WITH DML COMMANDS

As you know that DML (Data Manipulation Language) division of SQL, is dedicated to manipulating data in one way or another. This section is going to talk about various DML commands that are used for changing data in tables. Of these commands, INSERT INTO command is used for entering data into tables.

17.4.1 Inserting Data into Table

Values are placed in and removed from attributes of a relation with three DML commands : INSERT, DELETE and UPDATE. These are all referred to in *SQL* as *update* commands in a generic sense. In our text, lowercase "update" will indicate these commands generically and the uppercase for the keyword UPDATE.

17.4.1A INSERT INTO Command

The rows (tuples) are added to relations using INSERT command of *SQL*. In its simplest form, INSERT takes the following syntax :

SYNTAX

```
INSERT INTO <tablename> [ <column list> ]
VALUES ( <value>, <value> ... ) ;
```

For example, to enter a row into *employee* table (defined earlier), you could use the following statement :

```
INSERT INTO employee
VALUES (1001, 'Ravi', 'M', 'E4', 4670.00) ;
```

See the order of values matches the order of columns in the CREATE TABLE command of *employee*. The same can be done with an alternate command as shown below :

```
INSERT INTO employee (ecode, ename, sex, grade, gross)
VALUES (1001, 'Ravi', 'M', 'E4', 4670.00) ;
```

The INSERT statement adds a new row to *employee* giving a value for every column in the row. Note that the data values are in the same order as the column names in the table. Data can be added only to some columns in a row by specifying the columns and their data.

For instance, if you want to insert only *ecode, ename* and *sex* columns, you use the command :

```
INSERT INTO employee (ecode, ename, sex)
VALUES (2014, 'Manju', 'F') ;
```

The columns that are not listed in the INSERT command will have their default value, if it is defined for them, otherwise, NULL value.

If any other column (that does not have a **default value** and is defined NOT NULL) is skipped or omitted, an error message is generated and the row is not added.

NOTE

In an INSERT statement, only those columns can be omitted that have either default value defined or they allow NULL values.

EXAMPLE 17.3 *Add the following data in the above Table (Employee) as instructed.*

ID	*First_Name*	*Last_Name*	*User_ID*	*Salary*
1	Dim	Joseph	Jdim	5000
2	Jagannath	Mishra	jnmishra	4000
3	Siddharth	Mishra	smishra	8000
4	Shankar	Giri	sgiri	7000
5	Gautam	Buddha	bgautam	2000

(*i*) *Populate table with first record mentioning the column list in the insert clause.*

(*ii*) *Populate table with next two records without mentioning the column list in the insert clause.*

(*iii*) *Populate table with 4th record and enter only ID and First_Name.*

(*iv*) *Populate table with 5th recorded and enter ID, User_ID, and Last_Name only.*

Solution. (*i*)
```
INSERT INTO Employee (ID, First_name, Last_name, User_ID, Salary)
Values (1, 'Dim', 'Joseph', 'Jdim', 5000) ;
```

(*ii*)
```
INSERT INTO Employee
Values (2, 'Jagannath', 'Mishra', 'jnmishra', 4000) ;
INSERT INTO Employee
Values (3, 'Siddarth', 'Mishra', 'smishra', 8000) ;
```

(*iii*)
```
INSERT INTO Employee (ID, First_name)
Values (4, 'Shankar') ;
```

(*iv*)
```
INSERT INTO Employee (ID, User_ID, Last_Name)
Values (5, 'bgautam', 'Buddha') ;
```

As you know that following command is used to insert a new row with values for all columns of the table :

```
INSERT INTO <tablename>
VALUES (<value1>, <value2>.....) ;
```

And the following INSERT INTO command allows you to insert a new row with values of select columns :

```
INSERT INTO <tablename> (<columnname1>, <columnname2>.....)
VALUES (<value1>, <value2>.....) ;
```

Let us now learn how you can insert nulls, date-values and data from other table(s).

17.4.1B Inserting NULL values

To insert value NULL in a specific column, you can type NULL without quotes and NULL will be inserted in that column. Consider the following statement :

```
INSERT INTO EMPL (Empno, Ename, Job, Mgr, Hiredate, Sal, Comm, Deptno)
VALUES (8100, 'YASH', 'ANALYST', NULL, '10-MAY-03', 6000, NULL, 20) ;
```

See, for **Mgr** and **Comm** columns, NULL values have been inserted.

17.4.1C Inserting Dates

Dates are by default entered in 'YYYY-MM-DD' format *i.e.*, first four digits depicting *year*, followed by a hyphen, followed by 2 digits *month*, followed by a hyphen and a two digit day. All this is enclosed in single quotes. In the above two INSERT INTO statements given in previous section, the dates (HIREDATE) have been given in the same format.

17.4.1D Inserting Data from Another Table

INSERT command can also be used to take or derive values from one table and place them in another by using it with a query. To do this, simply replace the VALUES clause with an appropriate query as shown in the following example :

```
INSERT INTO branch1
SELECT * FROM branch2
WHERE gross > 7000.00 ;
```

This query will generate data to be inserted into table

It will extract all those rows from *branch2* that have *gross* more than 7000.00 and insert this produced result into the table *branch1*. To insert using a query, the following conditions must be true :

(*i*) Both the tables must be already created.

(*ii*) The columns of the tables being inserted into, must match the columns output by the subquery.

EXAMPLE 17.4 *Insert into empl table, the employee numbers, names, and salaries of all those employees of* **Temp** *table that have completed 2 years. The structure of* **Temp** *table is as follows :*

Enum	NOT NULL Number(4)
Ename	Varchar(10)
Sal	Number(7,2)
Months	Number(3)

Solution.

```
mysql> INSERT INTO Empl(Empno, Ename, Sal)
       SELECT Enum, Ename, Sal
       FROM Temp
       WHERE Months >= 24 ;
```

NOTE

Issue a command **COMMIT ;** to make the changes to a table permanent.

17.4.2 Modifying Data with UPDATE Command

Sometimes you need to change some or all of the values in an existing row. This can be done using the UPDATE command of *SQL*. The UPDATE command specifies the rows to be changed

using the WHERE clause, and the new data using the SET keyword. The new data can be a specified constant, an expression or data from other tables. For example, to change the reorder level *ROL* of all items to 250, you would write

```
UPDATE items
SET ROL = 250 ;
```

If you want to change *ROL* to 400 only for those items that have *ROL* as 300, you use the command

```
UPDATE items
SET ROL = 400
WHERE ROL = 300 ;
```

Updating Multiple Columns

To update multiple columns, multiple column assignments can be specified with SET clause, separated by commas. All of the said assignments will still be made to the table, a single row at a time. To update the *ROL* and *QOH* for items having icode less than 'I040', we shall write

```
UPDATE items
SET ROL = 400, QOH = 700
WHERE icode < 'I040' ;
```

Using Expressions in Update

Scalar expressions can also be used in the SET clause of the UPDATE command. Suppose, if you want to increase the gross pay of all the employees by Rs. 900, you could use the following expression :

```
UPDATE employee
SET gross = gross + 900 ;
```

To double the gross pay of employees of grade 'E3' and 'E4', you use the command :

```
UPDATE employee
SET gross = gross * 2
WHERE (grade = 'E3' OR grade = 'E4') ;
```

Updating to NULL Values

The NULL values can also be entered just as other values. For example, a new grade is to be introduced and all the employees with grade 'E4' have to be promoted to it. But for the time being this grade is not known, thus NULL values are to be inserted for grades 'E4'. This can be done as follows :

```
mysql> UPDATE employees
       SET grade = NULL
       WHERE grade = 'E4' ;
```

NOTE

Don't forget to issue **COMMIT ;** after making changes to your table.

17.4.3 Deleting Data with DELETE Command

While working with tables, we may reach at a situation where we no longer need some rows of data. In such a case, we would like to remove such rows. This can be done by using DELETE command.

The DELETE command removes rows from a table. This removes the entire rows, not individual field values, so no field argument is needed or accepted. The DELETE statement takes the following general form :

```
DELETE FROM <tablename>
[WHERE <predicate> ] ;
```

To remove all the contents of *items* table, you use the command :

```
DELETE FROM items ;
```

The table would now be empty and could be destroyed with a DROP TABLE command (covered later in the section 17.5.2).

Even some specific rows from a table can also be deleted. To determine which rows are deleted, you use a condition, just as you do for queries. For instance, to remove the tuples from *employee* table that have *gross* less than 2200.00, the following command is used :

```
DELETE FROM employee
WHERE gross < 2200.00 ;
```

17.5 MORE DDL COMMANDS

As you know, the DDL division of SQL is responsible for defining, redefining, modifying, dropping various database objects. Uptil now, you have learnt to use different database objects. This section is going to discuss some more DDL commands that prove very useful while redefining or dropping database object.

You have already worked with one DDL command – the CREATE TABLE command in the beginning of the chapter. That is, you already know how to create tables along with constraints' definition. This section is going to discuss DDL commands for altering tables, adding comments, enabling/disabling constraints, truncating tables, renaming objects, dropping tables, views and synonyms etc. Let us begin with ALTER TABLE command that is used to alter a table's definition.

17.5.1 ALTER TABLE Command

When we define a system, we specify what data we need to store, the size and data type of that data. What can we do when the requirements change ? We can alter the tables to accommodate the changed requirements.

The ALTER TABLE command is used to change definitions of existing tables. Usually, it can add columns to a table. Sometimes it can delete columns (depending on privileges) or change their sizes. In general, in MySQL SQL, ALTER TABLE command is used :

- to add a column
- to add an integrity constraint
- to redefine a column (datatype, size, default value).

17.5.1A Adding Columns

Typically, the syntax to add a column to a table is as follows :

SYNTAX

```
ALTER TABLE <table name> ADD <column name> <data type><size> [<constraint name>] ;
```

The new column will be added with NULL values for all rows currently in the table. It is generally possible to add several new columns, separated by commas, in a single command. It may be possible to drop or alter columns. Most often, altering columns will simply be a matter of increasing their size. For instance, to add a new column *tel_number* of type *integer* in table *Empl* you may give :

```
ALTER TABLE Empl
ADD (tel_number integer) ;
```

EXAMPLE 17.5 *In table EMPL (Chapter 16, Solved Problems), add a column named THRIFTPLAN of datatype NUMBER with a maximum of seven digits and two decimal places and a column named LOANCODE of datatype CHAR with a size of one and an integrity constraint that checks that load code should be one of these characters : 'E', 'C', 'H', 'P'.*

Solution.

```
mysql> ALTER TABLE empl
       ADD ( thriftplan NUMBER(7,2),
       loancode CHAR(1) CHECK (loancode IN('E', 'C', 'H', 'P'))) ;
```

While adding columns, you can define these types of integrity constraints (NOT NULL, UNIQUE, PRIMARY KEY, referential integrity (REFERENCES), DEFAULT and CHECK constraints) on existing columns using the ADD clause and in table constraint syntax.

EXAMPLE 17.6 *In table EMPL (Chapter 16, Solved Problems), add a column named SPECIAL PAY of datatype NUMBER with maximum of seven digits with 2 digits after decimal and with a DEFAULT value of 2400.*

Solution.

```
mysql> ALTER TABLE empl
       ADD(specialpay NUMBER(7,2) DEFAULT 2400) ;
```

If you use the ADD clause to add a new column to the table, then the initial value of each row for the new column is NULL. You can add a column with a NOT NULL constraint only to a table that contains no rows.

17.5.1B Modifying Column Definitions

You can use the MODIFY clause to change any of the following parts of a column definition :

- datatype
- size
- default value
- NOT NULL column constraint
- Order of column

The MODIFY clause need only specify the column name and the modified part of the definition, rather than the entire column definition.

NOTE

You can add a column with a NOT NULL constraint only to a table that contains no rows.

Datatypes and Sizes

You can change a CHAR column to VARCHAR and a VARCHAR to CHAR only if the column contains nulls in all rows or if you do not attempt to change the column size. You can change any column's datatype or decrease any column's size if all rows for the column contain nulls. However, you can always increase the size of a CHAR or VARCHAR column or the precision of a numeric column.

NOTE

You can change any column's datatype or decrease any column's size if all rows for the column contain nulls.

Default Values

A change to a column's default value only affects rows subsequently inserted into the table. Such a change does not change default values previously inserted.

Integrity Constraints

The only type of integrity constraint that you can add to an existing column using the MODIFY clause with the column constraint syntax is a NOT NULL constraint. However, you can define other types of integrity constraints (UNIQUE, PRIMARY KEY, referential integrity, and CHECK constraints) on existing columns using the ADD clause and the table constraint syntax.

You can define a NOT NULL constraint on an existing column only if the column contains no nulls.

Order of Column

With MODIFY clause you can also reorder the columns within a table. For this you need to use *FIRST* or *AFTER <colname>* clause to specify the position of the column. To modify existing columns of table, ALTER TABLE command can be used according to following syntax :

```
ALTER TABLE <tablename>
MODIFY (columname newdatatype (newsize) ) [FIRST|AFTER column] ;
```

To modify column *Job* of table *Empl* (Chapter 16) to have new width of 30 characters, you may give :

```
ALTER TABLE Empl
MODIFY (Job char(30) ) ;
```

Similarly, if you want to reorder an existing column say **ProjID** to be the first column in the table **Assignment**, you may write :

```
ALTER TABLE Assignment
MODIFY ProjID INT FIRST ;
```

EXAMPLE 17.7 *In table EMPL, increase the size of the THRIFTPLAN column to nine digits :*

Solution.

```
mysql> ALTER TABLE empl
       MODIFY(thriftplan NUMBER(9,2)) ;
```

EXAMPLE 17.8 *In table ACCOUNTS, modify the BAL column so that it has a default value of 0.*

Solution.

```
mysql> ALTER TABLE accounts
       MODIFY(bal DEFAULT 0) ;
```

EXAMPLE 17.9 *Add a PRIMARY KEY data constraint on the column supplier_no belonging to the table supplier_master.*

Solution.

```
ALTER TABLE supplier_master
ADD PRIMARY KEY(supplier_no) ;
```

NOTE

Please note that MySQL will reject FOREIGN KEY constraint if its database engine is not INNODB. So your query involving a FOREIGN KEY constraint may be rejected also.

Changing a Column Name

Sometimes you may need to change the name of one of your columns. For this you can use CHANGE clause of ALTER TABLE command as per following syntax :

```
ALTER TABLE
CHANGE [COLUMN] old_col_name new_col_name column_definition ;
```

For instance, to change the existing column namely First_Name of table Student, to FirstName you may write:

See, complete column description is given along with the new name.

```
ALTER TABLE Customers
CHANGE First_Name FirstName VARCHAR(20);
```

17.5.1C Removing Table Components

If you plan to remove a component of a table, then you may use the DROP clause of ALTER TABLE. The keywords mostly used with DROP clause of ALTER TABLE command are :

PRIMARY KEY	Drops the table's primary key constraint.
COLUMN <columnname>	Removes mentioned column from the table
FOREIGN KEY <constraintname>	Removes the mentioned foreign key constraint from the table.

Now consider the following examples :

```
ALTER TABLE Empl
DROP PRIMARY KEY, DROP FOREIGN KEY fk_1, DROP COLUMN deptno ;
```

In the above given ALTER TABLE statement, the primary key, the FOREIGN KEY constraint namely fk_1, and the column namely **deptno** are all removed form the **Empl** table.

EXAMPLE 17.10 *Drop the primary key of the DEPT table.*

Solution.

```
mysql> ALTER TABLE dept
       DROP PRIMARY KEY CASCADE ;
```

The CASCADE option drops any foreign keys that reference the primary key.

EXAMPLE 17.11 *Drop the DNAME column of the DEPT table.*

Solution.

```
mysql> ALTER TABLE dept
       DROP COLUMN dname ;
```

17.5.2 The DROP TABLE Command

The DROP TABLE command of *SQL* lets you drop a table from the database. The syntax for using a DROP TABLE command is :

```
DROP TABLE [IF EXISTS] <tablename>
```

That is, to drop a table *items*, you need to write :

```
DROP TABLE items ;
```

Once this command is given, the table name is no longer recognized and no more commands can be given on that object.

The IF EXISTS clause of DROP TABLE first checks whether the given table exists in the database or not. If it does, then it drops the mentioned table from the database. For instance, consider the following query :

```
DROP TABLE IF EXISTS players ;
```

The above query will first check for existence of *players* table in current database. If it exists, then it (table *players*) will be dropped from the database.

Check Point 17.1

1. Which command is used for creating tables ?
2. What is a constraint ? Name some constraints that you can apply to enhance database integrity.
3. What is the role of UNIQUE constraint ? How is PRIMARY KEY constraint different from UNIQUE constraint ?
4. What is primary key ? What is PRIMARY KEY constraint ?
5. What is NOT NULL constraint ? What is DEFAULT constraint ?
6. When a column's value is skipped in an INSERT command, which value is inserted in the database ?
7. Can a column defined with NOT NULL constraint, be skipped in an INSERT command ?
8. How would you view the structure of table Dept ?
9. Table Empl has same structure as that of table EMPL Write a query statement to insert data from table NewEmpl into EMPL where salary and comm is more than Rs. 4000.
10. What is the error in following statement ?

```
UPDATE EMPL ;
```

1. Identify the error :

```
DELETE ALL FROM TABLE EMPL ;
```

2. Differentiate between DDL and DML.

rip DDL COMMANDS Progress In Python 17.3

This 'Progress in Python' session is aimed at laying string foundation of decision constructs.

>>>❖<<<