CHAPTER 9

MORE ON DATABASES AND SQL

Learning Objectives

After studying this lesson the students will be able to:

- Define the terms:

  (i) Group (Aggregate) functions, Constraints

  (ii) Cartesian Product, Join, Referential Integrity, Foreign Key.

- Write queries using aggregate functions and GROUP BY clause.
- Access data from multiple tables
- Create tables with PRIMARY KEY and NOT NULL constraints
- Add a constraint to a table, remove a constraint from a table, modify a column of a table using ALTER TABLE command.
- Delete a table using DROP TABLE.

In the previous class, you have learnt some database concepts and SQL commands. You have also learnt how to create databases and tables within databases and how to access data from a table using various clauses of SELECT command. In this chapter you shall learn some more clauses and functions in SQL to access data from a table and how to access data from multiple tables of a database.

Puzzle [8]

It was Iftar party in Lucknow that Mr. David met Mr. Naqvi. They became friends and exchanged their phone numbers. After a few days, Mr. David rang up and invited Mr. Naqvi for New Year party at his house and gave him his house number as follows:

"I live in a long street. Numbered on the side of my house are the houses one, two, three and so on. All the numbers on one side of my house add up to exactly the same as all the

numbers on the other side of my house. I know there are more than thirty houses on that side of the street, but not so many as 50."

With this information, Mr. Naqvi was able to find Mr. David's house number. Can you also find?

Such situations are faced by the developers of RDBMS software where they have to think of retrieval of data from multiple tables satisfying some specified conditions.

Let us now move ahead with SQL and more database concepts.

Ms. Shabana Akhtar is in-charge of computer department in a Shoe factory. She has created a database 'Shoes' with the following tables:

SHOES

(To store the information about various types of shoes made in the factory)

```
+--------+--------------+------+-----+---------+-------+
| Field  | Type         | Null | Key | Default | Extra |
+--------+--------------+------+-----+---------+-------+
| code   | char(4)      | NO   | PRI | NULL    |       |
| name   | varchar(20)  | YES  |     | NULL    |       |
| type   | varchar(10)  | YES  |     | NULL    |       |
| size   | int(2)       | YES  |     | NULL    |       |
| cost   | decimal(6,2) | YES  |     | NULL    |       |
| margin | decimal(4,2) | YES  |     | NULL    |       |
| Qty    | int(4)       | YES  |     | NULL    |       |
+--------+--------------+------+-----+---------+-------+
```

CUSTOMERS

(To store the data of customers)

```
+-----------+-------------+------+-----+---------+-------+
| Field     | Type        | Null | Key | Default | Extra |
+-----------+-------------+------+-----+---------+-------+
| cust_Code | char(4)     | NO   | PRI | NULL    |       |
| name      | varchar(30) | YES  |     | NULL    |       |
| address   | varchar(50) | YES  |     | NULL    |       |
| phone     | varchar(30) | YES  |     | NULL    |       |
| category  | char(1)     | YES  |     | NULL    |       |
+-----------+-------------+------+-----+---------+-------+
```

ORDERS

(To store the data of orders placed by customers)

```
+-------------+----------+------+-----+---------+-------+
| Field       | Type     | Null | Key | Default | Extra |
+-------------+----------+------+-----+---------+-------+
| order_no    | int(5)   | NO   | PRI | NULL    |       |
| cust_code   | char(4)  | YES  |     | NULL    |       |
| Shoe_Code   | char(4)  | YES  |     | NULL    |       |
| order_qty   | int(4)   | YES  |     | NULL    |       |
| order_date  | date     | YES  |     | NULL    |       |
| target_date | date     | YES  |     | NULL    |       |
+-------------+----------+------+-----+---------+-------+
```

Sample data stored in these tables is given below:

SHOES

```
+------+----------------+--------+------+--------+--------+------+
| Code | Name           | type   | size | cost   | margin | Qty  |
+------+----------------+--------+------+--------+--------+------+
| 1001 | School Canvas  | School |    6 | 132.50 |   2.00 | 1200 |
| 1002 | School Canvas  | School |    7 | 135.50 |   2.00 |  800 |
| 1003 | School Canvas  | School |    8 | 140.75 |   2.00 |  600 |
| 1011 | School Leather | School |    6 | 232.50 |   2.00 | 2200 |
| 1012 | School Leather | School |    7 | 270.00 |   2.00 | 1280 |
| 1013 | School Leather | School |    8 | 320.75 |   NULL | 1100 |
| 1101 | Galaxy         | Office |    7 | 640.00 |   3.00 |  200 |
| 1102 | Galaxy         | Office |    8 | 712.00 |   3.00 |  500 |
| 1103 | Galaxy         | Office |    9 | 720.00 |   3.00 |  400 |
| 1201 | Tracker        | Sports |    6 | 700.00 |   NULL |  280 |
| 1202 | Tracker        | Sports |    7 | 745.25 |   3.50 | NULL |
| 1203 | Tracker        | Sports |    8 | 800.50 |   3.50 |  600 |
| 1204 | Tracker        | Sports |    9 | 843.00 |   NULL |  860 |
+------+----------------+--------+------+--------+--------+------+
```

CUSTOMERS

```
+-----------+---------------+---------------------+------------------+----------+
|Cust_Code  |name           |address              |Phone             |Category  |
+-----------+---------------+---------------------+------------------+----------+
|C001       |Novelty Shoes  |Raja Nagar, Bhopal   |4543556, 97878989 |A         |
|C002       |Aaram Footwear |31, Mangal Bazar, Agra|NULL             |B         |
|C003       |Foot Comfort   |New Market, Saharanpur|51917142, 76877888|B        |
|C004       |Pooja Shoes    |Janak Puri, New Delhi|61345432, 98178989|A         |
|C005       |Dev Shoes      |Mohan Nagar, Ghaziabad|NULL             |C         |
+-----------+---------------+---------------------+------------------+----------+
```

ORDERS

```
+----------+-----------+-----------+-----------+------------+-------------+
| order_no | cust_code | Shoe_Code | order_qty | order_date | target_date |
+----------+-----------+-----------+-----------+------------+-------------+
|        1 | C001      | 1001      |       200 | 2008-12-10 | 2008-12-15  |
|        2 | C001      | 1002      |       200 | 2008-12-10 | 2008-12-15  |
|        3 | C003      | 1011      |       150 | 2009-01-08 | 2009-01-13  |
|        4 | C002      | 1012      |       250 | 2009-01-08 | 2009-01-13  |
|        5 | C001      | 1011      |       400 | 2009-01-10 | 2009-01-15  |
|        6 | C002      | 1101      |       300 | 2009-01-10 | 2009-01-15  |
|        7 | C004      | 1201      |       200 | 2009-01-10 | 2009-01-15  |
|        8 | C005      | 1102      |       350 | 2009-01-10 | 2009-01-15  |
|        9 | C001      | 1103      |       225 | 2009-01-13 | 2009-01-18  |
|       10 | C002      | 1203      |       200 | 2009-01-14 | 2009-01-19  |
+----------+-----------+-----------+-----------+------------+-------------+
```

Let us now see how this database helps Ms. Akhtar in generating various reports quickly.

Aggregate Functions

In class XI we studied about single row functions available in SQL. A single row function works on a single value. SQL also provides us multiple row functions. A multiple row function works on multiple values. These functions are called aggregate functions or group functions. These functions are:

S.No.	Function	Purpose
1	MAX()	Returns the MAXIMUM of the values under the specified column/expression.
2	MIN()	Returns the MINIMUM of the values under the specified column/expression.

3	AVG()	Returns the AVERAGE of the values under the specified column/expression.
4	SUM()	Returns the SUM of the values under the specified column/expression.
5	COUNT()	Returns the COUNT of the number of values under the specified column/expression.

MAX() :

MAX() function is used to find the highest value of any column or any expression based on a column. MAX() takes one argument which can be any column name or a valid expression involving a column name. e.g.,

Purpose	Statement	Output

To find the highest cost of any type of shoe in the factory.	`SELECT MAX(cost)`
	`FROM shoes;`
		`+-----------+`
		`| MAX(cost) |`
		`+-----------+`
		`|    843.00 |`
		`+-----------+`

To find the highest cost of any shoe of type 'School'.	`SELECT MAX(cost)`
	`FROM shoes`
	`WHERE type =`
	`'School';`
		`+-----------+`
		`| MAX(cost) |`
		`+-----------+`
		`|    320.75 |`
		`+-----------+`

To find the highest selling price of any type of shoe.	`SELECT`
	`MAX(cost+cost*margin/`
	`100)`
	`FROM shoes;`
		`+--------------------------+`
		`| MAX(cost+cost*margin/100) |`
		`+--------------------------+`
		`|               828.517500000 |`
		`+--------------------------+`

To find the highest selling price of any type of shoe rounded to 2 decimal places.	`SELECT`
	`ROUND(MAX(cost+cost*mar`
	`gin/100),2)`
	`AS "Max. SP" FROM`
	`shoes;`
		`+---------+`
		`| Max. SP |`
		`+---------+`
		`|  733.36 |`
		`+---------+`

To find the highest selling price of any type of shoe rounded to 2 decimal places.	`SELECT ROUND(MAX(cost+cost*margin/100),2) AS "Max. SP" FROM shoes;`	`+----------+`
		`| Max. SP |`
		`+----------+`
		`|  733.36 |`
		`+----------+`

MIN() :

MIN() function is used to find the lowest value of any column or an expression based on a column. MIN() takes one argument which can be any column name or a valid expression involving a column name. e.g.,

Purpose	Statement	Output

To find the lowest cost of any type of shoe in the factory.	`SELECT MIN(cost) FROM shoes;`	`+-----------+`
		`| MIN(cost) |`
		`+-----------+`
		`|    843.00 |`
		`+-----------+`

To find the lowest cost of any shoe of type 'School'.	`SELECT MIN(cost) FROM shoes WHERE type = 'School';`	`+-----------+`
		`| MIN(cost) |`
		`+-----------+`
		`|    320.75 |`
		`+-----------+`

To find the lowest selling price of any type of shoe rounded to 2 decimal places.	`SELECT ROUND(MIN(cost+cost*margin/100),2) AS "Min. SP" FROM shoes;`	`+----------+`
		`| Min. SP |`
		`+----------+`
		`|  135.15 |`
		`+----------+`

AVG() :

AVG() function is used to find the average value of any column or an expression based on a column. AVG() takes one argument which can be any column name or a valid expression involving a column name. Here we have a limitation: the argument of AVG() function can be of numeric (int/decimal) type only. Averages of String and Date type data are not defined. E.g.,

Purpose	Statement	Output

To find the average margin from shoes table.	`SELECT AVG(margin)`
	`   FROM shoes;`
		`+-------------+`
		`| AVG(margin) |`
		`+-------------+`
		`|    2.600000 |`
		`+-------------+`

To find the average cost from the shoes table.	`SELECT AVG(cost) FROM shoes;`	`+------------+`
		`| AVG(cost)  |`
		`+------------+`
		`| 491.750000 |`
		`+------------+`

To find the average quantity in stock for the shoes of type Sports.	`SELECT AVG(qty)`
	`   FROM shoes`
	`   WHERE type =`
	`   'Sports';`
		`+----------+`
		`| AVG(qty) |`
		`+----------+`
		`| 580.0000 |`
		`+----------+`

SUM() :

SUM() function is used to find the total value of any column or an expression based on a column. SUM() also takes one argument which can be any column name or a valid expression involving a column name. Like AVG(), the argument of SUM() function can be of numeric (int/decimal) type only. Sums of String and Date type data are not defined. e.g.,

Purpose	Statement	Output

To find the total quantity present in the stock	`SELECT SUM(Qty) FROM Shoes;`	`+-----------+`
		`| SUM(Qty)  |`
		`+-----------+`
		`|     10020 |`
		`+-----------+`

To find the total order quantity	`SELECT SUM(order_qty)`
	`FROM orders;`
		`+-----------------+`
		`| SUM(order_qty)  |`
		`+-----------------+`
		`|            2475 |`
		`+-----------------+`

To find the the total value (Quanitity x Cost) of Shoes of type 'Office' present in the inventory	`SELECT SUM(cost*qty)`
	`FROM shoes`
	`WHERE type =`
	`'Office';`
		`+----------------+`
		`| SUM(cost*qty)  |`
		`+----------------+`
		`|      772000.00 |`
		`+----------------+`

COUNT() :

COUNT() function is used to count the number of values in a column. COUNT() takes one argument which can be any column name, an expression based on a column, or an asterisk (*). When the argument is a column name or an expression based on a column, COUNT() returns the number of non-NULL values in that column. If the argument is a *, then COUNT() counts the total number of rows satisfying the condition, if any, in the table. e.g.,

Purpose	Statement	Output
To count the total number of records in the table Shoes.	`SELECT COUNT(*) FROM shoes;`	`+----------+` `| COUNT(*) |` `+----------+` `|       13 |` `+----------+`
To count the different types of shoes that the factory produces	`SELECT COUNT(distinct type) FROM shoes;`	`+----------------------+` `| COUNT(distinct type) |` `+----------------------+` `|                    3 |` `+----------------------+`
To count the records for which the margin is greater than 2.00	`SELECT COUNT(margin) FROM shoes WHERE margin > 2;`	`+---------------+` `| COUNT(margin) |` `+---------------+` `|             5 |` `+---------------+`
To count the number of customers in 'A' category	`SELECT COUNT(*) FROM customers WHERE category ='A';`	`+----------+` `| COUNT(*) |` `+----------+` `|        2 |` `+----------+`
To count the number of orders of quantity more than 300	`SELECT COUNT(*) FROM orders WHERE order_qty > 300;`	`+----------+` `| COUNT(*) |` `+----------+` `|        2 |` `+----------+`

Aggregate functions and NULL values:

None of the aggregate functions takes NULL into consideration. NULL is simply ignored by all the aggregate functions. For example, the statement:

```
SELECT COUNT(*) FROM shoes;
```

produces the following output:

```
+----------+
| COUNT(*) |
+----------+
|       13 |
+----------+
```

Indicating that there are 13 records in the Shoes table. Whereas the query:

```
SELECT COUNT(margin) FROM shoes;
```

produces the output:

```
+---------------+
| COUNT(margin) |
+---------------+
|            10 |
+---------------+
```

This output indicates that there are 10 values in the margin column of Shoes table. This means there are 3 (13-10) NULLs in the margin column.

This feature of aggregate functions ensures that NULLs don't play any role in actual calculations. For example, the following statement:

```
SELECT AVG(margin) FROM shoes;
```

produces the output:

```
+-------------+
| AVG(margin) |
+-------------+
|    2.600000 |
+-------------+
```

The average margin has been calculated by adding all the 10 non NULL values from the margin column and dividing the sum by 10 and not by 13.

Know more!

There are some more aggregate functions available in MySQL. Try to find out what are those. Also try to use them.

GROUP BY

In practical applications many times there arises a need to get reports based on some groups of data. These groups are based on some column values. For example,

- *The management of the shoe factory may want to know what is the total quantity of shoes of various types. i.e., what is the total quantity of shoes of type School, Office, and Sports each.*
- *The management may also want to know what is the maximum, minimum, and average margin of each type of shoes.*
- *It may also be required to find the total number of customers in each category.*

There are many such requirements.

SQL provides GROUP BY clause to handle all such requirements.

For the above three situations, the statements with GROUP BY clause are given below:

In the first situation we want MySQL to divide all the records of shoes table into different groups based on their type (GROUP BY type) and for each group it should display the type and the corresponding total quantity (SELECT type, SUM(qty)). So the complete statement to do this is:

```
SELECT type, SUM(qty) FROM shoes
     GROUP BY type;
```
G1

and the corresponding output is:

```
+--------+----------+
| type   | SUM(qty) |
+--------+----------+
| Office |     1100 |
| School |     7180 |
| Sports |     1740 |
+--------+----------+
```

Similarly, for the second situation the statement is:

```
SELECT type, MIN(margin), MAX(margin), AVG(margin)
   FROM shoes GROUP BY type;                          G2
```

and the corresponding output is:

```
+--------+-------------+-------------+-------------+
| type   | MIN(margin) | MAX(margin) | AVG(margin) |
+--------+-------------+-------------+-------------+
| Office |        3.00 |        3.00 |    3.000000 |
| School |        2.00 |        2.00 |    2.000000 |
| Sports |        3.50 |        3.50 |    3.500000 |
+--------+-------------+-------------+-------------+
```

In the third situation we want MySQL to divide all the records of Customers table into different groups based on the their Category (GROUP BY Category) and for each group it should display the Category and the corresponding number of records (SELECT Category, COUNT(*)). So the complete statement to do this is:

```
SELECT category, COUNT(*) FROM customers GROUP BY category;   G3
```

```
+----------+----------+
| category | COUNT(*) |
+----------+----------+
| A        |        2 |
| B        |        2 |
| C        |        1 |
+----------+----------+
```

Let us have some more examples.

Consider the following statement:

```
SELECT cust_code, SUM(order_qty)
      FROM orders GROUP BY cust_code;
```

This statement produces the following output. Try to explain this this output.

```
+-----------+----------------+
| cust_code | SUM(order_qty) |
+-----------+----------------+
| C001      |           1025 |
| C002      |            750 |
| C003      |            150 |
| C004      |            200 |
| C005      |            350 |
+-----------+----------------+
```

Do the same for the following statement also:

```
    SELECT shoe_code, SUM(order_qty)
        FROM orders GROUP BY shoe_code;
```

```
+-----------+----------------+
| shoe_code | SUM(order_qty) |
+-----------+----------------+
| 1001      |            200 |
| 1002      |            200 |
| 1011      |            550 |
| 1012      |            250 |
| 1101      |            300 |
| 1102      |            350 |
| 1103      |            225 |
| 1201      |            200 |
| 1203      |            200 |
+-----------+----------------+
```

If you carefully observe these examples, you will find that GROUP BY is always used in conjunction with some aggregate function(s). A SELECT command with GROUP BY clause has a column name and one or more aggregate functions which are applied on that column and grouping is also done on this column only.

HAVING :

Sometimes we do not want to see the whole output produced by a statement with GROUP BY clause. We want to see the output only for those groups which satisfy some condition. It means we want to put some condition on individual groups (and not on individual records). A condition on groups is applied by HAVING clause. As an example reconsider the statement G1 discussed above. The statement produced three records in the output - one for each group. Suppose, we are interested in viewing only those groups' output for which the total quantity is more than 1500 (SUM(Qty) > 1500). As this condition is applicable to groups and not to individual rows, we use HAVING clause as shown below:

```
SELECT type, SUM(qty) FROM shoes

   GROUP BY type HAVING SUM(qty) > 1500;
```

```
+--------+----------+
| type   | SUM(qty) |
+--------+----------+
| School |     7180 |
| Sports |     1740 |
+--------+----------+
```

Now suppose for G2 we want the report only for those types for which the average margin is more than 2. For this, following is the statement and the corresponding output:

```
SELECT type, SUM(qty) FROM shoes

   GROUP BY type HAVING AVG(margin) >2;
```

```
+--------+----------+
| type   | SUM(qty) |
+--------+----------+
| Office |     1100 |
| Sports |     1740 |
+--------+----------+
```

In these statements if we try to put the condition using WHERE instead of HAVING, we shall get an error. Another way of remembering this is that whenever a condition involves an aggregate function, then we use HAVING clause in conjunction with GROUP BY clause.

Situations may also arise when we want to put the conditions on individual records as well as on groups. In such situations we use both WHERE (for individual records) and HAVING (for groups) clauses. This can be explained with the help of the following examples:

- The management of the shoe factory may want to know what is the total quantity of shoes, of sizes other than 6, of various types. i.e., what is the total quantity of shoes (of sizes other than 6) of type School, Office, and Sports each.

Moreover, the report is required only for those groups for which the total quantity is more than 1500.

- The management may also want to know what is the maximum, minimum, and average margin of each type of shoes. But in this reports shoes of sizes 6 and 7 only should be included. Report is required only for those groups for which the minimum margin is more than 2.

The statements and their outputs corresponding to above requirements are given below:

```
SELECT type, SUM(qty) FROM shoes

    WHERE size <> 6◄────────────Checks individual row

    GROUP BY type HAVING sum (qty) > 1500;◄───Checks individual group

                  +--------+----------+
                  | type   | SUM(qty) |
                  +--------+----------+
                  | School |     3780 |
                  +--------+----------+
```

```
SELECT type, MIN(margin), MAX(margin), AVG(margin) FROM shoes

    WHERE size in (6,7)

    GROUP BY type having MIN(margin) > 2;

    +--------+-------------+-------------+-------------+
    | type   | MIN(margin) | MAX(margin) | AVG(margin) |
    +--------+-------------+-------------+-------------+
    | Office |        3.00 |        3.00 |    3.000000 |
    | Sports |        3.50 |        3.50 |    3.500000 |
    +--------+-------------+-------------+-------------+
```

Displaying Data from Multiple Tables

In each situation that we have faced so far, the data was extracted from a single table. There was no need to refer to more than one tables in the same statement. But many times, in real applications of databases, it is required to produce reports which need data from more than one tables. To understand this consider the following situations:

- *The management of the shoe factory wants a report of orders which lists three columns: Order_No, corresponding customer name, and phone number. - (MT-1)*

  *In this case order number will be taken from **Orders** table and corresponding customer name from **Customers** table.*

- *The management wants a four-column report containing order_no, order_qty, name of the corresponding shoe and its cost. - (MT-2)*

  *In this case order number and order quantity will be taken from **Orders** table and corresponding shoe name and cost from **Shoes** table.*

- *The management wants the names of customers who have placed any order of quantity more than 300. - (MT-3)*

  *In this case Order quantity will be checked in **Orders** table and for each record with quantity more than 300, corresponding **Customer** name will be taken from Customers table.*

- *The management wants a report in which with each Order_No management needs name of the corresponding customer and also the total cost (Order quantity x Cost of the shoe) of the order are shown. - (MT-4)*

  *In this case order number will be taken from **Orders** table and corresponding customer name from **Customers** table. For the cost of each order the quantity will be taken from Orders table and the Cost from **Shoes** table.*

In all these cases, the data is to be retrieved from multiple tables. SQL allows us to write statements which retrieve data from multiple tables.

To understand how this is done, consider the following tables of a database.

Product

```
+------+-------------+
| Code | Name        |
+------+-------------+
| P001 | Toothpaste  |
| P002 | Shampoo     |
| P003 | Conditioner |
+------+-------------+
```

Supplier

```
+-----------+--------------+-------------+
| Sup_Code  | Name         | Address     |
+-----------+--------------+-------------+
| S001      | DC & Company | Uttam Nagar |
| S002      | SURY Traders | Model Town  |
+-----------+--------------+-------------+
```

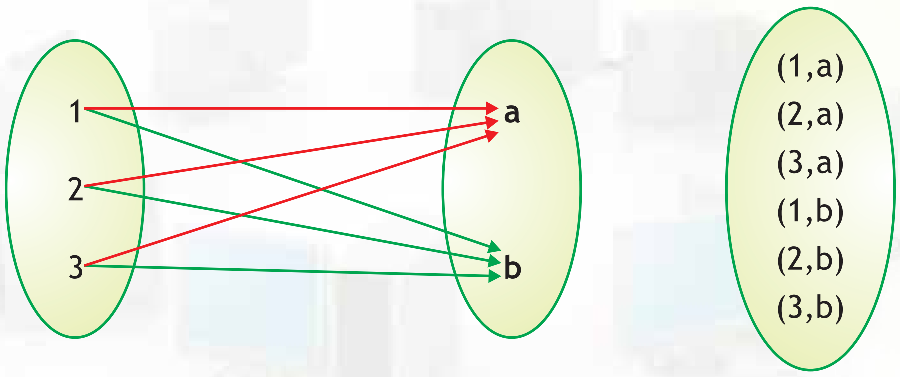
Order_table

```
+----------+--------+----------+
| Order_No | P_Code | Sup_Code |
+----------+--------+----------+
|        1 | P001   | S002     |
|        2 | P002   | S002     |
+----------+--------+----------+
```

These tables are taken just to explain the current concept.

Cartesian Product or Cross Join of tables :

Cartesian product (also called Cross Join) of two tables is a table obtained by pairing up each row of one table with each row of the other table. This way if two tables contain 3 rows and 2 rows respectively, then their Cartesian product will contain 6 (=3x2) rows. This can be illustrated as follows:



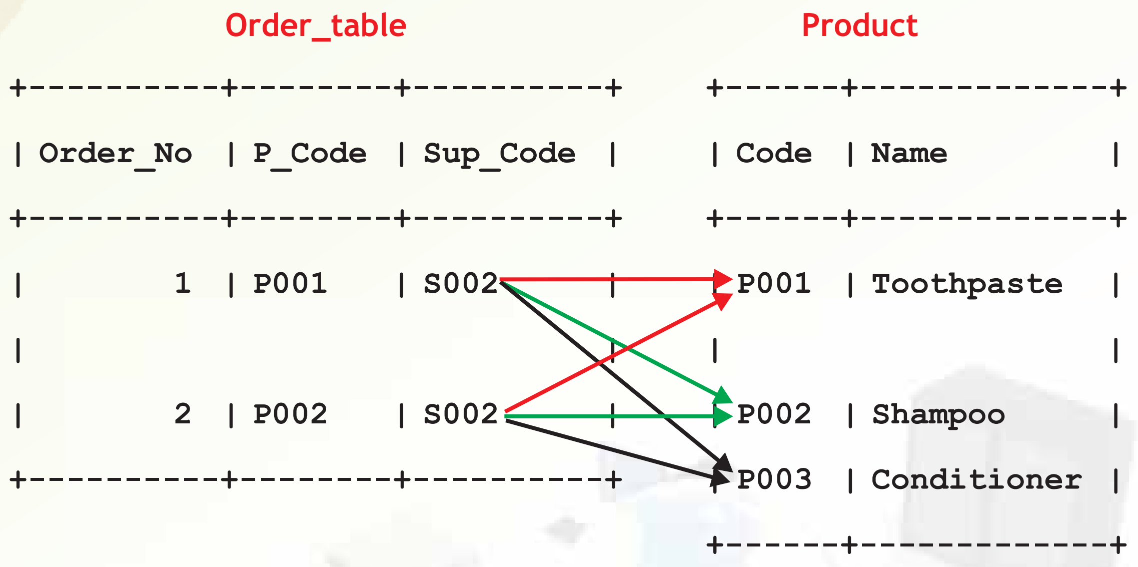
Cartesian product of two tables

Notice that the arrows indicate the 'ordered pairing'.

The number of columns in the Cartesian product is the sum of the number of columns in both the tables. In SQL, Cartesian product of two rows is obtained by giving the names of both tables in FROM clause. An example of Cartesian product is shown below:

```
SELECT * FROM order_table, product;
```

To give the output of this query, MySQL will pair the rows of the mentioned tables as follows:



```
+----------+--------+----------+     +------+-------------+
| Order_No | P_Code | Sup_Code |     | Code | Name        |
+----------+--------+----------+     +------+-------------+
|        1 | P001   | S002     |      P001 | Toothpaste  |
|                              |     |                    |
|        2 | P002   | S002     |      P002 | Shampoo     |
+----------+--------+----------+      P003 | Conditioner |
                                     +------+-------------+
```

And the following output will be produced:

```
+----------+--------+----------+------+-------------+
| Order_No | P_Code | Sup_Code | Code | Name        |    -(CP-1)
+----------+--------+----------+------+-------------+
|        1 | P001   | S002     | P001 | Toothpaste  |
|        2 | P002   | S002     | P001 | Toothpaste  |
|        1 | P001   | S002     | P002 | Shampoo     |
|        2 | P002   | S002     | P002 | Shampoo     |
|        1 | P001   | S002     | P003 | Conditioner |
|        2 | P002   | S002     | P003 | Conditioner |
+----------+--------+----------+------+-------------+
```

Here we observe that the Cartesian product contains all the columns from both tables. Each row of the first table (Order_table) is paired with each row of the second table (Product).

If we change the sequence of table names in the FROM clause, the result will remain the same but the sequence of rows and columns will change. This can be observed in the following statement and the corresponding output.

```
SELECT * FROM product, order_table;
```

```
+------+-------------+----------+--------+----------+
| Code | Name        | Order_No | P_Code | Sup_Code |     -(CP-2)
+------+-------------+----------+--------+----------+
| P001 | Toothpaste  |        1 | P001   | S002     |
| P001 | Toothpaste  |        2 | P002   | S002     |
| P002 | Shampoo     |        1 | P001   | S002     |
| P002 | Shampoo     |        2 | P002   | S002     |
| P003 | Conditioner |        1 | P001   | S002     |
| P003 | Conditioner |        2 | P002   | S002     |
+------+-------------+----------+--------+----------+
```

We can have Cartesian product of more than two tables also. Following is the Cartesian Product of three tables:

```
SELECT * FROM order_table, supplier, product;
```

-(CP-3)

```
+----------+--------+----------+----------+-------------+------------+------+-------------+
| Order_No | P_Code | Sup_Code | Sup_Code | Name        | Address    | Code |Name         |
+----------+--------+----------+----------+-------------+------------+------+-------------+
|        1 | P001   | S002     | S001     | DC & Company| Uttam Nagar| P001 |Toothpaste   |
|        2 | P002   | S002     | S001     | DC & Company| Uttam Nagar| P001 |Toothpaste   |
|        1 | P001   | S002     | S002     | SURY Traders| Model Town | P001 |Toothpaste   |
|        2 | P002   | S002     | S002     | SURY Traders| Model Town | P001 |Toothpaste   |
|        1 | P001   | S002     | S001     | DC & Company| Uttam Nagar| P002 |Shampoo      |
|        2 | P002   | S002     | S001     | DC & Company| Uttam Nagar| P002 |Shampoo      |
|        1 | P001   | S002     | S002     | SURY Traders| Model Town | P002 |Shampoo      |
|        2 | P002   | S002     | S002     | SURY Traders| Model Town | P002 |Shampoo      |
|        1 | P001   | S002     | S001     | DC & Company| Uttam Nagar| P003 |Conditioner  |
|        2 | P002   | S002     | S001     | DC & Company| Uttam Nagar| P003 |Conditioner  |
|        1 | P001   | S002     | S002     | SURY Traders| Model Town | P003 |Conditioner  |
|        2 | P002   | S002     | S002     | SURY Traders| Model Town | P003 |Conditioner  |
+----------+--------+----------+----------+-------------+------------+------+-------------+
```

The complete Cartesian product of two or more tables is, generally, not used directly. But, some times it is required. Suppose the company with the above database wants to send information of each of its products to each of its suppliers. For follow-up, the management wants a complete list in which each Supplier's detail is paired with each Product's detail. For this, the computer department can produce a list which is the Cartesian product of Product and Supplier tables, as follows:

```
SELECT *, '          ' AS Remarks FROM Product, Supplier;
```

to get the following report:

```
+------+-------------+----------+--------------+-------------+----------+
| Code | Name        | Sup_Code | Name         | Address     | Remarks  |
+------+-------------+----------+--------------+-------------+----------+
| P001 | Toothpaste  | S001     | DC & Company | Uttam Nagar |          |
| P001 | Toothpaste  | S002     | SURY Traders | Model Town  |          |
| P002 | Shampoo     | S001     | DC & Company | Uttam Nagar |          |
| P002 | Shampoo     | S002     | SURY Traders | Model Town  |          |
| P003 | Conditioner | S001     | DC & Company | Uttam Nagar |          |
| P003 | Conditioner | S002     | SURY Traders | Model Town  |          |
+------+-------------+----------+--------------+-------------+----------+
```

Equi- Join of tables :

The complete Cartesian product of two or more tables is, generally, not used directly. Sometimes the complete Cartesian product of two tables may give some confusing information also. For example, the first Cartesian product (CP-1) indicates that each order (Order Numbers 1 and 2) is placed for each Product (Code 'P001', 'P002', 'P003'). But this is incorrect!

Similar is the case with CP-2 and CP-3 also.

But we can extract meaningful information from the Cartesian product by placing some conditions in the statement. For example, to find out the product details corresponding to each Order details, we can enter the following statement:

```
SELECT * FROM order_table, product WHERE p_code = code;
```

```
+----------+--------+----------+------+------------+
| Order_No | P_Code | Sup_Code | Code | Name       |
+----------+--------+----------+------+------------+
|        1 | P001   | S002     | P001 | Toothpaste |
|        2 | P002   | S002     | P002 | Shampoo    |
+----------+--------+----------+------+------------+
```

Two table names are specified in the FROM clause of this statement, therefore MySQL creates a Cartesian product of the tables. From this Cartesian product MySQL selects only those records for which P_Code (Product code specified in the Order_table table) matches Code (Product code in the Product table). These selected records are then displayed.

It always happens that whenever we have to get the data from more than one tables, there is some common column based on which the meaningful data is extracted from the tables. We specify table names in the FROM clause of SELECT command. We also give the condition specifying the matching of common column. (When we say common column, it does not mean that the column names have to be the same. It means that the columns should represent the same data with the same data types.) Corresponding to this statement, internally the Cartesian product of the tables is made. Then based on the specified condition the meaningful data is extracted from this Cartesian product and displayed.

Let us take another example of producing a report which displays the supplier name and address corresponding to each order.

```
SELECT Order_No, Order_table.Sup_Code, Name, Address
    FROM order_table, supplier
    WHERE order_table.sup_code = supplier.sup_code;
```

```
+----------+----------+-------------+-----------+
| Order_No | Sup_Code | Name        | Address   |
+----------+----------+-------------+-----------+
|        1 | S002     | SURY Traders| Model Town |
|        2 | S002     | SURY Traders| Model Town |
+----------+----------+-------------+-----------+
```

In this statement the tables referred are Order_table and Supplier. In these tables sup_code is the common column. This column exists with same name in both the tables. Therefore whenever we mention it, we have to specify the table from which we want to extract this column. This is known as qualifying the column name. If we don't qualify the common column name, the statement would result into an error due to the ambiguous the column names.

Following is another example of equi-join. This time with three tables.

```
Select Order_no, Product.name as Product, Supplier.Name as Supplier
       From order_table, Product, Supplier
              WHERE order_table.Sup_Code = Supplier.Sup_Code
              and P_Code = Code;
```

The output produced by this statement is:

```
+----------+-----------+-------------+
| Order_no | Product   | Supplier    |
+----------+-----------+-------------+
|        1 | Toothpaste | SURY Traders |
|        2 | Shampoo   | SURY Traders |
+----------+-----------+-------------+
```

Let us now get back to our original Shoe database and see how Ms. Akhtar uses the concept of joins to extract data from multiple tables.

For the situation MT-1, she writes the query:

```
SELECT order_no , name, phone
    FROM orders, customers
    WHERE orders.cust_code = customers.cust_code;
```

and get the following required output:

```
+----------+---------------+-------------------+
| order_no | name          | phone             |
+----------+---------------+-------------------+
|        1 | Novelty Shoes | 4543556, 97878989  |
|        2 | Novelty Shoes | 4543556, 97878989  |
|        5 | Novelty Shoes | 4543556, 97878989  |
|        9 | Novelty Shoes | 4543556, 97878989  |
|        4 | Aaram Footwear | NULL              |
|        6 | Aaram Footwear | NULL              |
|       10 | Aaram Footwear | NULL              |
|        3 | Foot Comfort  | 51917142, 76877888 |
|        7 | Pooja Shoes   | 61345432, 98178989 |
|        8 | Dev Shoes     | NULL              |
+----------+---------------+-------------------+
```

Following are the queries and corresponding outputs for the situations MT-2, MT-3, and MT-4 respectively:

```
SELECT order_no , Order_Qty, name, cost
    FROM orders, shoes WHERE Shoe_Code = code;
```

```
+----------+-----------+---------------+--------+
| order_no | Order_Qty | name          | cost   |
+----------+-----------+---------------+--------+
|        1 |       200 | School Canvas | 132.50 |
|        2 |       200 | School Canvas | 135.50 |
|        3 |       150 | School Leather| 232.50 |
|        4 |       250 | School Leather| 270.00 |
|        5 |       400 | School Leather| 232.50 |
|        6 |       300 | Galaxy        | 640.00 |
|        7 |       200 | Tracker       | 700.00 |
|        8 |       350 | Galaxy        | 712.00 |
|        9 |       225 | Galaxy        | 720.00 |
|       10 |       200 | Tracker       | 800.50 |
+----------+-----------+---------------+--------+
```

```
SELECT name, address FROM orders, customers
WHERE orders.cust_code = customers.cust_code
and order_qty > 300;
```

```
+--------------+-----------------------+
| name         | address               |
+--------------+-----------------------+
| Novelty Shoes | Raja Nagar, Bhopal   |
| Dev Shoes    | Mohan Nagar, Ghaziabad |
+--------------+-----------------------+
```

```
 SELECT order_no, Order_Qty, customers.name,
    cost*order_qty as 'Order Cost'
    FROM orders, shoes, Customers
    WHERE Shoe_Code = code
          and Orders.Cust_Code = Customers.Cust_Code
    order by order_no;
```

```
+-----------+-----------+----------------+------------+
| order_no  | Order_Qty | name           | Order Cost |
+-----------+-----------+----------------+------------+
|         1 |       200 | Novelty Shoes  |   26500.00 |
|         2 |       200 | Novelty Shoes  |   27100.00 |
|         3 |       150 | Foot Comfort   |   34875.00 |
|         4 |       250 | Aaram Footwear |   67500.00 |
|         5 |       400 | Novelty Shoes  |   93000.00 |
|         6 |       300 | Aaram Footwear |  192000.00 |
|         7 |       200 | Pooja Shoes    |  140000.00 |
|         8 |       350 | Dev Shoes      |  249200.00 |
|         9 |       225 | Novelty Shoes  |  162000.00 |
|        10 |       200 | Aaram Footwear |  160100.00 |
+-----------+-----------+----------------+------------+
```

Here is another statement extracting data from multiple tables. Try to find out what will be its output and then try this statement on computer and check whether you thought of the correct output.

```
SELECT order_no , Order_Qty, name, cost
    FROM orders, shoes
    WHERE Shoe_Code = code and order_qty > 200;
```

Foreign Key :

As we have just seen, in a join the data is retrieved from the Cartesian product of two tables by giving a condition of equality of two corresponding columns - one from each table. Generally, this column is the Primary Key of one table. In the other table this column is the Foreign key. Such a join which is obtained by putting a condition of equality on cross join is called an 'equi-join'. As an example, once again consider the Product, Supplier, and Order tables referenced earlier. For quick reference these tables are shown once again:

Product

```
+------+-------------+
| Code | Name        |
+------+-------------+
| P001 | Toothpaste  |
| P002 | Shampoo     |
| P003 | Conditioner |
+------+-------------+
```

Supplier

```
+----------+-------------+-------------+
| Sup_Code | Name        | Address     |
+----------+-------------+-------------+
| S001     | DC & Company | Uttam Nagar |
| S002     | SURY Traders | Model Town  |
+----------+-------------+-------------+
```

Order_table

```
+----------+--------+----------+
| Order_No | P_Code | Sup_Code |
+----------+--------+----------+
|        1 | P001   | S002     |
|        2 | P002   | S002     |
+----------+--------+----------+
```

In these tables there is a common column between Product and Order_table tables (Code and P_Code respectively) which is used to get the Equi-Join of these two tables. Code is the Primary Key of Product table and in Order_table table it is not so (we can place more than one orders for the same product). In the order_table, P_Code is a Foreign Key. Similarly, Sup_Code is the primary key in Supplier table whereas it is a Foreign Key is Order_table table. A foreign key in a table is used to ensure referential integrity and to get Equi-Join of two tables.

Referential Integrity: Suppose while entering data in Order_table we enter a P_Code that does not exist in the Product table. It means we have placed an order for an item that does not exist! We should and can always avoid such human errors. Such errors are avoided by explicitly making P_Code a foreign key of Order_table table which always references the Product table to make sure that a non-existing product code is not entered in the Order_table table. Similarly, we can also make Sup_Code a Foreign key in Order_table table which always references Customer table to check validity of Cust_code. This can be done, but how to do it is beyond the scope of this book.

This property of a relational database which ensures that no entry in a foreign key column of a table can be made unless it matches a primary key value in the corresponding related table is called Referential Integrity.

Union

Union is an operation of combining the output of two SELECT statements. Union of two SELECT statements can be performed only if their outputs contain same number of columns and data types of corresponding columns are also the same. The syntax of UNION in its simplest form is:

```
SELECT <select_list>
    FROM <tablename>
    [WHERE <condition> ]
 UNION [ALL]
SELECT <select_list>
    FROM <tablename>
    [WHERE <condition> ];
```

Union does not display any duplicate rows unless ALL is specified with it.

Example:

Suppose a company deals in two different categories of items. Each category contains a number of items and for each category there are different customers. In the database there are two customer tables: Customer_Cat_1 and Customer_Cat_2. If it is required to produce a combined list of all the customers, then it can be done as follows:

```
SELECT Cust_Code from Customer_Cat_1
UNION
SELECT Cust_Code from Customer_Cat_2;
```

If a customer exists with same customer code in both the tables, its code will be displayed only once - because Union does display duplicate rows. If we explicitly want the duplicate rows, then we can enter the statement:

```
SELECT Cust_Code from Customer_Cat_1
UNION ALL
SELECT Cust_Code from Customer_Cat_2;
```

Constraints

Many times it is not possible to keep a manual check on the data that is going into the tables using INSERT or UPDATE commands. The data entered may be invalid. MySQL provides some rules, called Constraints, which help us, to some extent, ensure validity of the data. These constraints are:

S.No.	Constraint	Purpose
1.	PRIMARY KEY	Sets a column or a group of columns as the Primary Key of a table. Therefore, NULLs and Duplicate values in this column are not accepted.
2.	NOT NULL	Makes sure that NULLs are not accepted in the specified column.
3.	FOREIGN KEY	Data will be accepted in this column, if same data value exists in a column in another related table. This other related table name and column name are specified while creating the foreign key constraint.
4.	UNIQUE	Makes sure that duplicate values in the specified column are not accepted.
5.	ENUM	Defines a set of values as the column domain. So any value in this column will be from the specified values only.
6.	SET	Defines a set of values as the column domain. Any value in this column will be a seubset of the specied set only.

We shall discuss only the PRIMARY KEY and NOT NULL constraints in this book. Other constraints are beyond the scope of this book.

PRIMARY KEY:

Recall that primary key of a table is a column or a group of columns that uniquely identifies a row of the table. Therefore no two rows of a table can have the same primary key value. Now suppose that the table Shoes is created with the following statement:

```
CREATE TABLE Shoes
     (Code CHAR(4), Name VARCHAR(20), type VARCHAR(10),
     size INT(2), cost DECIMAL(6,2), margin DECIMAL(4,2),
     Qty INT(4));
```

We know that in this table Code is the Primary key. But, MySQL does not know that! Therefore it is possible to enter duplicate values in this column or to enter NULLs in this column. Both these situations are unacceptable.

To make sure that such data is not accepted by MySQL, we can set Code as the primary key of Shoes table. It can be done by using the PRIMARY KEY clause at the time of table creation as follows:

```
CREATE TABLE Shoes
     (Code CHAR(4) PRIMARY KEY, Name VARCHAR(20),
     type VARCHAR(10), size INT(2), cost DECIMAL(6,2),
     margin DECIMAL(4,2), Qty INT(4));
```

or as follows:

```
CREATE TABLE Shoes
     (Code CHAR(4), Name VARCHAR(20), type VARCHAR(10),
     size INT(2), cost DECIMAL(6,2), margin DECIMAL(4,2),
     Qty INT(4), PRIMARY KEY (Code));
```

To create a table Bills with the combination of columns Order_No and Cust_Code as the primary key, we enter the statement:

```
CREATE TABLE bills
     (Order_Num INT(4) PRIMARY KEY,
     cust_code VARCHAR(4) PRIMARY KEY,
     bill_Date DATE, Bill_Amt DECIMAL(8,2));
```

Contrary to our expectation, we get an error (Multiple primary key defined) with this statement. The reason is that MySQL interprets this statement as if we are trying to create two primary keys of the table - Order_Num, and Cust_code. But a table can have at most one primary key. To set this combination of columns a primary key we have to enter the statement as follows:

```
CREATE TABLE bills
    (Order_Num INT(4), cust_code VARCHAR(4),
    bill_Date date, Bill_Amt DECIMAL(8,2),
    PRIMARY KEY(Order_Num, cust_code));
```

Let us now check the table structure with the command: DESC bills;

The table structure is as shown below:

```
+-----------+-------------+------+-----+---------+-------+
| Field     | Type        | Null | Key | Default | Extra |
+-----------+-------------+------+-----+---------+-------+
| Order_Num | INT(4)      | NO   | PRI | 0       |       |
| cust_code | VARCHAR(4)  | NO   | PRI |         |       |
| bill_Date | date        | YES  |     | NULL    |       |
| Bill_Amt  | DECIMAL(8,2)| YES  |     | NULL    |       |
+-----------+-------------+------+-----+---------+-------+
```

These columns constitute the primary key of the table

NULLs cannot be accepted in these columns.

NOT NULL:

Many times there are some columns of a table in which NULL values should not be accepted. We always want some known valid data values in these columns. For example, we cannot have an order for which the customer code is not known. It means whenever we enter a row in the orders table, corresponding customer code cannot be NULL. Similarly while entering records in the Shoes table, we have to mention the Shoe size, it cannot be set NULL. There may be any number of such situations.

While creating a table we can specify in which columns NULLs should not be accepted as follows:

```
CREATE TABLE Shoes
     (Code CHAR(4) PRIMARY KEY, Name VARCHAR(20),
     type VARCHAR(10), size INT(2) NOT NULL,
     cost DECIMAL(6,2), margin DECIMAL(4,2), Qty INT(4));
CREATE TABLE bills
     (Order_Num INT(4), cust_code VARCHAR(4),
     bill_Date DATE, Bill_Amt DECIMAL(8,2) NOT NULL,
     PRIMARY KEY (Order_Num, cust_code));
```

Now if we try to enter a NULL in the specified column, MySQL will reject the entry and give an error.

Viewing Constraints, Viewing the Columns Associated with Constraints :

After creating a table, we can view its structure using DESC command. The table structure also includes the constraints, if any. Therefore, when we use DESC command, we are shown the table structure as well as constraints, if any. A constraint is shown beside the column name on which it is applicable. E.g., the statement:

```
DESC Shoes;
```

displays the table structure as follows:

```
+--------+--------------+------+-----+---------+-------+
| Field  | Type         | Null | Key | Default | Extra |
+--------+--------------+------+-----+---------+-------+
| Code   | CHAR(4)      | NO   | PRI | NULL    |       |
| Name   | VARCHAR(20)  | YES  |     |         |       |
| type   | VARCHAR(10)  | YES  |     | NULL    |       |
| size   | INT(2)       | NO   |     | 0       |       |
| cost   | DECIMAL(6,2) | YES  |     | NULL    |       |
| margin | DECIMAL(4,2) | YES  |     | NULL    |       |
| Qty    | INT(4)       | YES  |     | NULL    |       |
+--------+--------------+------+-----+---------+-------+
```

ALTER TABLE

In class XI, we have studied that a new column can be added to a table using ALTER TABLE command. Now we shall study how ALTER TABLE can be used:

- to add a constraint
- to remove a constraint
- to remove a column from a table
- to modify a table column

Add, Modify, and Remove constraints :

If we create a table without specifying any primary key, we can still specify its primary key by ALTER TABLE command. Suppose we have created the Shoes table without specifying any Primary key, then later we can enter the statement as follows:

```
ALTER TABLE Shoe ADD PRIMARY KEY(code);
```

This will set Code as the primary key of the table. But if the Code column already contains some duplicate values, then this statement will give an error.

In MySQL, it is also possible to change the primary key column(s) of a table. Suppose, in the Shoes table, istread of Code, we want to set the combination of 'Name' and 'Size' as the primary key. For this first we have to DROP the already existing primary key (i.e., Code) and then add the new primary key (i.e., Name and Size). The corresponding statements are as follows:

```
ALTER TABLE Shoes DROP PRIMARY KEY;
```

After this statement, there is no primary key of Shoe table. Now we can add the new primary key as follows:

```
ALTER TABLE Shoe ADD PRIMARY KEY (Name, Size);
```

Now if we see the table structure by DESC Shoes; statement, it will be shown as follows:

```
+--------+-------------+------+-----+---------+-------+
| Field  | Type        | Null | Key | Default | Extra |
+--------+-------------+------+-----+---------+-------+
| Code   | CHAR(4)     | NO   |     | NULL    |       |
| Name   | VARCHAR(20) | NO   | PRI |         |       |
| type   | VARCHAR(10) | YES  |     | NULL    |       |
| size   | INT(2)      | NO   | PRI | 0       |       |
| cost   | DECIMAL(6,2)| YES  |     | NULL    |       |
| margin | DECIMAL(4,2)| YES  |     | NULL    |       |
| Qty    | INT(4)      | YES  |     | NULL    |       |
+--------+-------------+------+-----+---------+-------+
```

In MySQL, it is not possible to add or drop NOT NULL constraint explicitly after the table creation. But it can be done using MODIFY clause of ALTER TABLE command. As an example, suppose we don't want to accept NULL values in bill_date column of bills table, we can issue the statement:

```
ALTER TABLE bills MODIFY bill_date DATE NOT NULL;
```

Later on if we wish to change this status again, we can do so by entering the command:

```
ALTER TABLE bills MODIFY bill_date DATE NULL;
```

Remove and Modify columns :

ALTER TABLE can be used to remove a column from a table. This is done using DROP clause in ALTER TABLE command. The syntax is as follws:

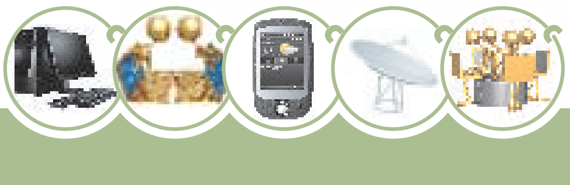
```
ALTER TABLE <tablename> DROP <columnname>
    [, DROP <columnname> [, DROP <columnname> [, . . . ]]];
```

Following are some self explanatory examples of SQL statemenets to remove columns from tables:

```
ALTER TABLE Shoes DROP Qty;

ALTER TABLE Orders DROP Cust_Code;

ALTER TABLE Student DROP Class, DROP RNo, DROP Section;
```



Although any column of a table can be removed, MySQL puts the restriction that a primary key column can be removed only if the remaining, primary key columns, if any, do not contain any duplicate entry. This can be understood more clearly with the help of following example:

The Name and Size columns of the Shoe table constitute its primary key. Now if we drop the Name column from the table, Size will be the remaining Primary Key column of the table. Therefore, duplicate entries in the Size column should not be allowed. To ensure this, before removing Name column from the table, MySQL checks that there are no duplicate entries present in the Size column of the table. If there are any, then the statement trying to remove Name column from the table will result in an error and the Name column will not be removed. If there are no duplicate enteries in the Size column, then Name column will be removed. Similar will be the case with the Name column, if we try to remove Size column. But there won't be any problem if we try to remove both the primary key columns simultaneously with one ALTER TABLE statement as follows:

```
    ALTER TABLE Shoes DROP name, DROP size;
```

ALTER TABLE can also be used to change the data type of a table column. For this the syntax is as follows:

```
    ALTER TABLE <tablename> MODIFY <col_name> <new datatype>
        [,MODIFY <col_name> <new datatype>
            [,MODIFY <col_name> <new data type>  [, . . . ]]];
```

e.g., the statement:

```
    ALTER TABLE shoes modify code CHAR(5), modify type VARCHAR(20);
```

changes the data type of column Code to CHAR(5) and that of type to VARCHAR(20).

When we give a statement to chage the data type of a column, MySQL executes that statement correctly only if the change in data type does not lead to any data loss. E.g., if we try to change the data type of order_date column of orders table from date to int, we'll get an error. This is because the data already stored in this column cannot be converted into int type. Similarly, if a column of VARCHAR(10) type conatins some data value which is 10 characters long, then the data type of this column cannot be converted to VARCHAR(n), where n is an integer less than 10.

DROP TABLE

Sometimes there is a requirement to remove a table from the database. In such cases we don't want merely to delete the data from the table, but we want to delete the table itself. DROP TABLE command is used for this purpose. The syntax of DROP TABLE command is as follows:

```
DROP TABLE <tablename>;
```

e.g. to remove the table Orders from the database we enter the statement:

```
DROP TABLE Orders;
```

And after this statement orders table is no longer available in the database. It has been removed.

Summary

- **Aggregate or Group functions:** MySQL provides Aggregate or Group functions which work on a number of values of a column/expression and return a single value as the result. Some of the most frequently used. Aggregate functions in MySQL are : MIN(), MAX(), AVG(), SUM(), COUNT().
- **Data Types in aggregate functions:** MIN(), MAX(), and COUNT() work on any type of values - Numeric, Date, or String. AVG(), and SUM() work on only Numeric values (INT and DECIMAL).
- **NULLs in aggregate functions:** Aggregate functions totally ignore NULL values present in a column.
- **GROUP BY:** GROUP BY clause is used in a SELECT statement in conjunction with aggregate functions to group the result based on distinct values in a column.
- **HAVING:** HAVING clause is used in conjuction with GROUP BY clause in a SELECT statement to put condition on groups.
- **WHERE Vs HAVING:** WHERE is used to put a condition on individual row of a table whereas HAVING is used to put condition on individual group formed by GROUP BY clause in a SELECT statement.

- **Cartesian Product (or Cross Join):** Cartesian product of two tables is a table obtained by pairing each row of one table with each row of the other. A cartesian product of two tables contains all the columns of both the tables.
- **Equi-Join:** An equi join of two tables is obtained by putting an equality condition on the Cartesian product of two tables. This equality condition is put on the common column of the tables. This common column is, generally, primary key of one table and foreign key of the other.
- **Foreign Key:** It is a column of a table which is the primary key of another table in the same database. It is used to enforce referential integrity of the data.
- **Referential Integrity:** The property of a relational database which ensures that no entry in a foreign key column of a table can be made unless it matches a primary key value in the corresponding column of the related table.
- **Union:** Union is an operation of combining the output of two SELECT statements.
- **Constraints:** These are the rules which are applied on the columns of tables to ensure data integrity and consistency.
- **ALTER TABLE:** ALTER TABLE command can be used to Add, Remove, and Modify columns of a table. It can also be used to Add and Remove constraints.
- **DROP TABLE:** DROP TABLE command is used to delete tables.

EXERCISES

MULTIPLE CHOICE QUESTIONS

1. Which of the following will give the same answer irrespective of the NULL values in the specified column:

   a. MIN() b. MAX()

   c. SUM() d. None of the above

2. An aggregate function:

   a. Takes a column name as its arguments

   b. May take an expression as its argument

c. Both (a) and (b)

d. None of (a) and (b)

3. HAVING is used in conjunction with

a.	WHERE	b.	GROUP BY clause
c.	Aggregate functions	d.	None of the above

4. In the FROM clause of a SELECT statement

a. Multiple Column Names are specified.

b. Multiple table names are specified.

c. Multiple Column Names may be specified.

d. Multiple table names may be specified.

5. JOIN in RDBMS refers to

a.	Combination of multiple columns	b.	Combination of multiple rows
c.	Combination of multiple tables	d.	Combination of multiple databases

6. Equi-join is formed by equating

a.	Foreign key with Primary key	b.	Each row with all other rows
c.	Primary key with Primary key	d.	Two tables

7. Referential integrity

a. Must be maintained

b. Cannot be maintained

c. Is automatically maintained by databases

d. Should not be maintained

8. A Primary key column

a.	Can have NULL values	b.	Can have duplicate values
c.	Both (a) and (b)	d.	Neither (a) nor (b)

9. Primary Key of a table can be
   a. Defined at the time of table creation only.
   b. Defined after table creation only.
   c. Can be changed after table creation
   d. Cannot be changed after table creation
10. Two SELECT commands in a UNION
    a. Should select same number of columns.
    b. Should have different number of columns
    c. Both (a) and (b)
    d. Neither (a) nor (b)

ANSWER THE FOLLOWING QUESTIONS

1. Why are aggregate functions called so? Name some aggregate functions.
2. Why is it not allowed to give String and Date type arguments for SUM() and AVG() functions? Can we give these type of arguments for other functions?
3. How are NULL values treated by aggregate functions?
4. There is a column C1 in a table T1. The following two statements:

   ```
   SELECT COUNT(*) FROM T1;    and SELECT COUNT(C1) from T1;
   ```

   are giving different outputs. What may be the possible reason?
5. What is the purpose of GROUP BY clause?
6. What is the difference between HAVING and WHERE clauses? Explain with the help of an example.
7. What is the Cartesian product of two table? Is it same as an Equi-join?
8. There are two table T1 and T2 in a database. Cardinality and degree of T1 are 3 and 8 respectively. Cardinality and degree of T2 are 4 and 5 respectively. What will be the degree and Cardinality of their Cartesian product?

9. What is a Foreign key? What is its importance?

10. What are constraints? Are constraints useful or are they hinderance to effective management of databases?

11. In a database there is a table Cabinet. The data entry operator is not able to put NULL in a column of Cabinet? What may be the possible reason(s)?

12. In a database there is a table Cabinet. The data entry operator is not able to put duplicate values in a column of Cabinet? What may be the possible reason(s)?

13. Do Primary Key column(s) of a table accept NULL values?

14. There is a table T1 with combination of columns C1, C2, and C3 as its primary key? Is it possible to enter:

    a. NULL values in any of these columns?

    b. Duplicate values in any of these columns?

15. At the time of creation of table X, the data base administrator specified Y as the Primary key. Later on he realized that instead of Y, the combination of column P and Q should have been the primary key of the table. Based on this scenario, answer the following questions:

    a. Is it possible to keep Y as well as the combination of P and Q as the primary key?

    b. What statement(s) should be entered to change the primary key as per the requirement.

16. Does MySQL allow to change the primary key in all cases? If there is some special case, please mention.

17. What are the differences between DELETE and DROP commands of SQL?

LAB EXERCISES

1. In a database create the following tables with suitable constraints :

STUDENTS

```
+-------+---------------+-------+-----+------+-----------------------+-----------------+
| AdmNo | Name          | Class | Sec | RNo  | Address               | Phone           |
+-------+---------------+-------+-----+------+-----------------------+-----------------+
|  1271 | Utkarsh Madaan|    12 | C   |    1 | C-32, Punjabi Bagh    | 4356154         |
|  1324 | Naresh Sharma |    10 | A   |    1 | 31, Mohan Nagar       | 435654          |
|  1325 | Md. Yusuf     |    10 | A   |    2 | 12/21, Chand Nagar    | 145654          |
|  1328 | Sumedha       |    10 | B   |   23 | 59, Moti Nagar        | 4135654         |
|  1364 | Subya Akhtar  |    11 | B   |   13 | 12, Janak Puri        | NULL            |
|  1434 | Varuna        |    12 | B   |   21 | 69, Rohini            | NULL            |
|  1461 | David DSouza  |    11 | B   |    1 | D-34, Model Town      | 243554, 98787665 |
|  2324 | Satinder Singh|    12 | C   |    1 | 1/2, Gulmohar Park    | 143654          |
|  2328 | Peter Jones   |    10 | A   |   18 | 21/32B, Vishal Enclave | 24356154       |
|  2371 | Mohini Mehta  |    11 | C   |   12 | 37, Raja Garden       | 435654, 6765787 |
+-------+---------------+-------+-----+------+-----------------------+-----------------+
```

SPORTS

```
+-------+-------------+-------------+-------+
| AdmNo | Game        | CoachName   | Grade |
+-------+-------------+-------------+-------+
|  1324 | Cricket     | Narendra    | A     |
|  1364 | Volleball   | M.P. Singh  | A     |
|  1271 | Volleball   | M.P. Singh  | B     |
|  1434 | Basket Ball | I. Malhotra | B     |
|  1461 | Cricket     | Narendra    | B     |
|  2328 | Basket Ball | I. Malhotra | A     |
|  2371 | Basket Ball | I. Malhotra | A     |
|  1271 | Basket Ball | I. Malhotra | A     |
|  1434 | Cricket     | Narendra    | A     |
|  2328 | Cricket     | Narendra    | B     |
|  1364 | Basket Ball | I. Malhotra | B     |
+-------+-------------+-------------+-------+
```

a) Based on these tables write SQL statements for the following queries:

i. Display the lowest and the highest classes from the table STUDENTS.

ii. Display the number of students in each class from the table STUDENTS.

iii. Display the number of students in class 10.

iv. Display details of the students of Cricket team.

v. Display the Admission number, name, class, section, and roll number of the students whose grade in Sports table is 'A'.

vi. Display the name and phone number of the students of class 12 who are play some game.

vii. Display the Number of students with each coach.

viii. Display the names and phone numbers of the students whose grade is 'A' and whose coach is Narendra.

b) Identify the Foreign Keys (if any) of these tables. Justify your choices.

c) Predict the the output of each of the following SQL statements, and then verify the output by actually entering these statements:

i. SELECT class, sec, count(*) FROM students GROUP BY class, sec;

ii. SELECT Game, COUNT(*) FROM Sports GROUP BY Game;

iii. SELECT game, name, address FROM students, Sports

WHERE students.admno = sports.admno AND grade = 'A';

iv. SELECT Game FROM students, Sports

WHERE students.admno = sports.admno AND Students.AdmNo = 1434;

2. In a database create the following tables with suitable constraints :

ITEMS

```
+--------+--------------+--------------+------+
| I_Code | Name         | Category     | Rate |
+--------+--------------+--------------+------+
| 1001   | Masala Dosa  | South Indian |   60 |
| 1002   | Vada Sambhar | South Indian |   40 |
| 1003   | Idli Sambhar | South Indian |   40 |
| 2001   | Chow Mein    | Chinese      |   80 |
| 2002   | Dimsum       | Chinese      |   60 |
| 2003   | Soup         | Chinese      |   50 |
| 3001   | Pizza        | Italian      |  240 |
| 3002   | Pasta        | Italian      |  125 |
+--------+--------------+--------------+------+
```

BILLS

```
+--------+-----------+--------+-----+
| BillNo | Date      | I_Code | qty |
+--------+-----------+--------+-----+
|      1 | 2010-04-01 | 1002   |   2 |
|      1 | 2010-04-01 | 3001   |   1 |
|      2 | 2010-04-01 | 1001   |   3 |
|      2 | 2010-04-01 | 1002   |   1 |
|      2 | 2010-04-01 | 2003   |   2 |
|      3 | 2010-04-02 | 2002   |   1 |
|      4 | 2010-04-02 | 2002   |   4 |
|      4 | 2010-04-02 | 2003   |   2 |
|      5 | 2010-04-03 | 2003   |   2 |
|      5 | 2010-04-03 | 3001   |   1 |
|      5 | 2010-04-03 | 3002   |   3 |
+--------+-----------+--------+-----+
```

a) Based on these tables write SQL statements for the following queries:

   i. Display the average rate of a South Indian item.

   ii. Display the number of items in each category.

   iii. Display the total quantity sold for each item.

   iv. Display total quanity of each item sold but don't display this data for the items whose total quantity sold is less than 3.

   v. Display the details of bill records along with Name of each corresponding item.

   vi. Display the details of the bill records for which the item is 'Dosa'.

   vii. Display the bill records for each Italian item sold.

   viii. Display the total value of items sold for each bill.

b) Identify the Foreign Keys (if any) of these tables. Justify your answer.

c) Answer with justification (Think independently. More than one answers may be correct. It all depends on your logical thinking):

   i. Is it easy to remember the Category of item with a given item code? Do you find any kind of pattern in the items code? What could be the item code of another South Indian item?

ii. What can be the possible uses of Bills table? Can it be used for some analysis purpose?

iii. Do you find any columns in these tables which can be NULL? Is there any column which must not be NULL?

3. In a database create the following tables with suitable constraints :

VEHICLE

```
+---------+-------------+------+-----+---------+-------+
| Field   | Type        | Null | Key | Default | Extra |
+---------+-------------+------+-----+---------+-------+
| RegNo   | char(10)    | NO   | PRI |         |       |
| RegDate | date        | YES  |     | NULL    |       |
| Owner   | varchar(30) | YES  |     | NULL    |       |
| Address | varchar(50) | YES  |     | NULL    |       |
+---------+-------------+------+-----+---------+-------+
```

CHALLAN

```
+------------+----------+------+-----+---------+-------+
| Field      | Type     | Null | Key | Default | Extra |
+------------+----------+------+-----+---------+-------+
| Challan_No | int(11)  | NO   | PRI | 0       |       |
| Ch_Date    | date     | YES  |     | NULL    |       |
| RegNo      | char(10) | YES  |     | NULL    |       |
| Offence    | int(3)   | YES  |     | NULL    |       |
+------------+----------+------+-----+---------+-------+
```

OFFENCE

```
+--------------+-------------+------+-----+---------+-------+
| Field        | Type        | Null | Key | Default | Extra |
+--------------+-------------+------+-----+---------+-------+
| Offence_Code | int(3)      | NO   | PRI | 0       |       |
| Off_desc     | varchar(30) | YES  |     | NULL    |       |
| Challan_Amt  | int(4)      | YES  |     | NULL    |       |
+--------------+-------------+------+-----+---------+-------+
```

a) Based on these tables write SQL statements for the following queries:

i. Display the dates of first registration and last registration from the table Vehicle.

ii. Display the number of challans issued on each date.

iii. Display the total number of challans issued for each offence.

iv. Display the total number of vehicles for which the 3rd and 4th characters of RegNo are '6C'.

v. Display the total value of challans issued for which the Off_Desc is 'Driving without License'.

vi. Display details of the challans issued on '2010-04-03' along with Off_Desc for each challan.

vii. Display the RegNo of all vehicles which have been challaned more than once.

viii. Display details of each challan alongwith vehicle details, Off_desc, and Challan_Amt.

b) Identify the Foreign Keys (if any) of these tables. Justify your choices.

c) Should any of these tables have some more column(s)? Think, discuss in peer groups, and discuss with your teacher.

4. In a database create the following tables with suitable constraints:

Table: Employee

No	Name	Salary	Zone	Age	Grade	Dept
1	Mukul	30000	West	28	A	10
2	Kritika	35000	Centre	30	A	10
3	Naveen	32000	West	40		20
4	Uday	38000	North	38	C	30
5	Nupur	32000	East	26		20
6	Moksh	37000	South	28	B	10
7	Shelly	36000	North	26	A	30

Table: Department

Dept	DName	MinSal	MaxSal	HOD
10	Sales	25000	32000	1
20	Finance	30000	50000	5
30	Admin	25000	40000	7

a) Based on these tables write SQL statements for the following queries:

i. Display the details of all the employees who work in Sales department.

ii. Display the Salary, Zone, and Grade of all the employees whose HOD is Nupur.

iii. Display the Name and Department Name of all the employees.

iv. Display the names of all the employees whose salary is not within the specified range for the corresponding department.

v. Display the name of the department and the name of the corresponding HOD for all the departments.

b) Identify the Foreign Keys (if any) of these tables. Justify your choices.

TEAM BASED TIME BOUND EXERCISE:

(Team size recommended: 3 students each team)

1. A chemist shop sells medicines manufactured by various pharmaceutical companies. When some medicine is sold, the corresponding stock decreases and when some medicines are bought (by the chemist shop) from their suppliers, the corresponding stock increases. Now the shop wants to keep computerized track of its inventory. The shop owner should be able to find

- The current stock of any medicine.
- The total sale amount of any specific time period (a specific day, or month, or any period between two specific dates)
- The details of all the medicines from a specific supplier.

- The details of all the medicines from a specific manufacturer.
- Total value of the medicines in the stock.

There may be a number of other reports which the shop owner may like to have.

The job of each team is to design a database for this purpose. Each team has to specify:

- The structure (with constraints) of each of the tables designed (with justification).
- How the tables are related to each other (foreign keys).
- How the design will fulfill all the mentioned requirements.
- At least 10 reports that can be generated with the database designed.

2. To expand its business, XYZ Mall plans to go online. Anyone who shops at the Mall will be given a membership number and Password which can be used for online shopping. With this membership number and password, customers can place their orders online. The mall will maintain the customers' data and orders' data. A person is put on duty to keep constantly checking the Orders data. Whenever an order is received, its processing has to start at the earliest possible.

   The Orders' data will be analysed periodically (monthly, quarterly, annually - whatever is suitable) to further improve business and customer satisfaction.

   The job of each team is to design a database for this purpose. Each team has to specify:

   - The structure (with constraints) of each of the tables designed (with justification).
   - How the tables are related to each other (foreign keys).
   - How the design will fulfill all the mentioned requirements.
   - At least 10 reports that can be generated with the database designed.