```
mysql> SELECT COUNT(*) FROM Employee;
Output: 5                (NUL

mysql> SELECT COUNT(Ecode) FROM Employee;
Output: 5                (No NULL value exists in the column Ecode.)

mysql> SELECT COUNT(Salary) FROM Employee;
Output: 3                (NULL values are ignored while counting the total number of
                         records on the basis of Salary.)
```

7.20 GROUP BY

The GROUP BY clause can be used in a SELECT statement to collect data across multiple records and group the results by one or more columns. It groups the rows on the basis of the values present in one of the columns and then the aggregate functions are applied on any column of these groups to obtain the result of the query.

This clause can be explained with reference to the table Student; the rows can be divided into four groups on the basis of the column Stream. One group of rows belongs to "Science" stream, another belongs to "Commerce" stream, the third group belongs to "Humanities" stream and the fourth belongs to "Vocational" stream. Thus, by using GROUP BY clause, the rows can be divided on the basis of the stream column.

Syntax for the GROUP BY clause is:

SELECT <column1, column2, ...column_n>, <aggregate_function (expression)>
FROM <tables>
WHERE <conditions>
GROUP BY <column1>, <column2>, ... <column_n>;

Here, *column_names* must include the columns on the basis of which grouping is to be done.

aggregate_function can be a function such as sum(), count(), max(), min(), avg(), etc.

For example, to display the name, stream, marks and count the total number of students who have secured more than 90 marks according to their stream.

```
mysql> SELECT Name, Stream, COUNT(*) AS "Number of students"
       FROM Student
       WHERE Marks>90
       GROUP BY Stream;
```

Resultant table: Student

Name	Stream	Number of students	Marks
Raj Kumar	Science	1	93
Deep Singh	Commerce	2	98

2 rows in a set (0.02 sec)

7.21 HAVING CLAUSE

The HAVING clause is used in combination with the GROUP BY clause. It can be used in a SELECT statement to filter the records by specifying a condition which a GROUP BY returns.

The purpose of using HAVING clause with GROUP BY is to allow aggregate functions to be used along with the specified condition. [This is because the aggregate functions are not allowed to be used with WHERE clause as it is evaluated on a single row whereas the aggregate functions are evaluated on a group of rows.]

Thus, if any aggregate function is to be used after the FROM clause in a SELECT command, then instead of using the WHERE clause, HAVING clause should be used.

The Syntax for HAVING clause is:

SELECT <column1>, <column2>, ...<column_n>, <aggregate_function (expression)>

FROM <tables>

WHERE <condition/predicates>

GROUP BY [<column1, column2, ... column_n]

HAVING [<condition1 ... condition_n>];

For example,

```
mysql> SELECT Stream, SUM(Marks) AS "Total Marks"
       FROM Student
       GROUP BY Stream
       HAVING MAX(Marks) <85;
```

Stream	Total Marks
Science	151
Commerce	0
Humanities	158
Vocational	152

In the given output, the sum(Marks) for Commerce Stream is 0 since none of the values for field Marks in the Commerce Stream is less than 85.

CTM: SELECT statement can contain only those attributes which are already present in the GROUP **BY** clause.

7.22 AGGREGATE FUNCTIONS AND CONDITIONS ON GROUPS (HAVING CLAUSE)

You may use any condition on group, if required. HAVING <condition> clause is used to apply a condition on a group.

```
mysql> SELECT Job, SUM(Pay) FROM EMPLOYEE GROUP BY Job HAVING
       SUM(Pay)>=8000;

mysql> SELECT Job, SUM(Pay) FROM EMPLOYEE GROUP BY Job HAVING
       AVG(Pay)>=7000;

mysql> SELECT Job, SUM(Pay) FROM EMPLOYEE GROUP BY Job HAVING COUNT(*)>=5;

mysql> SELECT Job, MIN(Pay), MAX(Pay), AVG(Pay) FROM EMPLOYEE
       GROUP BY Job HAVING SUM(Pay)>=8000;

mysql> SELECT Job, SUM(Pay) FROM EMPLOYEE WHERE City='Dehradun'
       GROUP BY Job HAVING COUNT(*)>=5;
```

WHERE vs HAVING

WHERE clause works in respect to the whole table but **HAVING** clause works on Group only. If WHERE and HAVING both are used, then WHERE will be executed first. Where is used to put a condition on individual row of a table whereas HAVING is used to put a condition on an individual group formed by GROUP BY clause in a SELECT statement.

Aggregate Functions and Group (GROUP BY Clause): Other Combinations

Consider the following table Employee with NULL values against the Salary field for some employees:

Employee

Ecode	Ename	Salary	Job	City
E1	Ritu Jain	NULL	Manager	Delhi
E2	Vikas Verma	4500	Executive	Jaipur
E3	Rajat Chaudhary	6000	Clerk	Kanpur
E4	Leena Arora	NULL	Manager	Bengaluru
E5	Shikha Sharma	8000	Accountant	Kanpur

None of the aggregate functions takes NULL into consideration. NULL values are simply ignored by all the aggregate functions as clearly shown in the examples given below.

An aggregate function may be applied on a column with DISTINCT or * (ALL) symbol. If nothing is given, ALL scope is assumed.

- **Using sum (<Column>)**

  This function returns the sum of values in the given column or expression.

```
mysql> SELECT SUM(Salary) FROM Employee;
mysql> SELECT SUM(DISTINCT Salary) FROM Employee;
mysql> SELECT SUM(Salary) FROM Employee WHERE City='Kanpur';
mysql> SELECT SUM(Salary) FROM Employee GROUP BY City HAVING
       City='Kanpur';
mysql> SELECT Job, SUM(Salary) FROM Employee GROUP BY Job;
```

- **Using min (<Column>)**

  This function returns the Minimum value in the given column.

```
mysql> SELECT  MIN(Salary)  FROM  Employee;
mysql> SELECT MIN(Salary) FROM Employee GROUP BY City HAVING City='Kanpur';
mysql> SELECT Job, MIN(Salary) FROM Employee GROUP BY Job;
```

- **Using max (<Column>)**

  This function returns the Maximum value in the given column.

```
mysql> SELECT  MAX(Salary)  FROM  Employee;
mysql> SELECT  MAX(Salary)  FROM  Employee  WHERE  City='Kanpur';
mysql> SELECT MAX(Salary) FROM Employee GROUP BY City HAVING  City='Kanpur';
```

- **Using avg (<Column>)**

  This function returns the Average value in the given column.

```
mysql> SELECT AVG(Salary) FROM Employee;
mysql> SELECT AVG(Salary) FROM Employee GROUP BY City; HAVING
       City='Kanpur';
```

➢ **Using count (<*|Column>)**

This function returns the number of rows in the given column.

```
mysql> SELECT COUNT (*) FROM Employee;
mysql> SELECT COUNT(Salary) FROM Employee GROUP BY City;
mysql> SELECT COUNT(*), SUM(Salary) FROM Employee GROUP BY Job HAVING
       Job= "Manager";
```

7.23 SQL JOINS

An SQL JOIN clause is used to combine rows from two or more tables, based on a common field between them. While querying for a join, more than one table is considered in FROM clause. The process/function of combining data from multiple tables is called a JOIN.

SQL can extract data from two or even more than two related tables by performing either a physical or virtual join on the tables using WHERE clause.

This is unlike cartesian product which make all possible combinations of tuples. While using the JOIN clause of SQL, we specify conditions on the related attributes of two tables within the FROM clause. Usually, such an attribute is the primary key in one table and foreign key in another table.

The types of SQL joins are as follows:

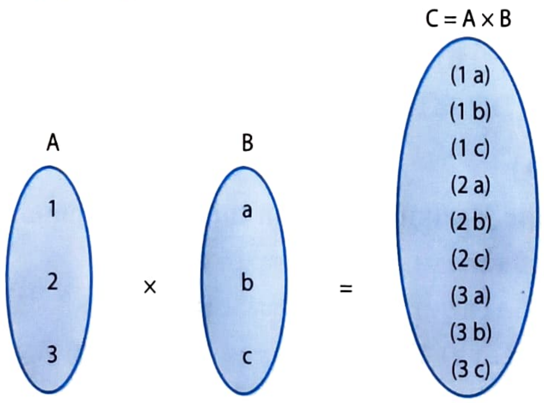
1. Cartesian Product (Cross Product)
2. Equi Join
3. Natural Join

Cartesian Product (Cross Product)

The Cartesian product is also termed as cross product or cross-join. The Cartesian product is a binary operation and is denoted by (×). The degree of the new relation formed is the sum of the degrees of two relations on which Cartesian product is performed. The number of tuples in the new relation is equal to the product of the number of tuples of the two tables on which Cartesian product is performed.

For example,

If A={1,2,3} and B={a,b,c}, find A × B.



Cartesian Product for A × B

Columns after Cartesian Product = 1+1 = 2

Rows after Cartesian Product = 3×3 = 9

Table: student

Rollno	Name
1	Rohan
2	Jaya
3	Teena

Table: games

gameno	gname
10	Football
11	Lawn tennis

Cartesian product for student × games:

```
mysql> SELECT Name, gname FROM Student, games;
```

student X games

Name	gname
Rohan	Football
Jaya	Football
Teena	Football
Rohan	Lawn Tennis
Jaya	Lawn Tennis
Teena	Lawn Tennis

Therefore, a Cartesian product is formed when no join conditions exist or are invalid. When we perform Cartesian product between two tables, all the rows in the first table are joined to all the rows in the second table. Using Cartesian product operation results in a large number of rows as the output, so it is seldom used.

Equi Join

An Equi join is a simple SQL join condition that uses the equal to sign (=) as a comparison operator for defining a relationship between two tables on the basis of a common field, *i.e.*, primary key and foreign key.

Syntax for Equi Join:

SELECT <column1>, <column2>,....

FROM <table1>, <table2>

WHERE <table1. Primary key column> = <table2.foreign key column>;

For example,

Table: Student

Rollno	Name
1	Rohan
2	Jaya
3	Teena
4	Diksha

Table: Fees

Rollno	Fee
4	4500
2	5500
3	5000

Resultant Table

Rollno	Name	Fee
2	Jaya	5500
3	Teena	5000
4	Diksha	4500

```
mysql> SELECT A.Rollno, A.Name, B.Fee FROM
       Student A, Fees B
       WHERE A.Rollno = B.Rollno;
```

In the given SELECT statement, A and B are the alias names (alternate names) for the tables student and fees respectively. So, we can always use alternate name in place of primary name for the two tables to be joined.

(d) To display the CODE, NAME, VTYPE from both the tables with distance travelled (KM) is less than 90 km.

Ans.
```
SELECT A.CODE, NAME,
VTYPE FROM TRAVEL A,
VEHICLE B
WHERE A.CODE=B.CODE AND KM<90;
```

JOIN Operation using Table alias names

(e) To display the NAME and amount to be paid for vehicle code as 105. Amount to be paid is calculated as the product of KM and PERKM.

Ans.
```
SELECT NAME,
KM*PERKM FROM TRAVEL A,
VEHICLE B
WHERE A.CODE=B.CODE AND A.CODE='105';
```

Natural Join

The JOIN in which only one of the identical columns exists is called Natural Join. It is similar to Equi Join except that duplicate columns are eliminated in Natural Join that would otherwise appear in Equi Join.

In Natural Join, we can also specify the names of columns to fetch in place of (*), which is responsible for the appearance of common column twice in the output. In other words, the SQL Natural Join is a type of Equi Join and is structured in such a way that columns with the same name of associated tables will appear only once. The following two conditions are must for Natural Join:

- The associated tables have one or more pairs of identically named columns.
- The columns must be of same data type.

In NATURAL JOIN, the join condition is not required; it automatically joins based on the common column value.

Natural Join is an alternative method to Equi Join.

Syntax is:

```
SELECT * FROM <Table1> NATURAL JOIN <Table2>;
```

For example,

```
mysql>SELECT * FROM Student NATURAL JOIN Fees;
```

Resultant Table

Rollno	Name	Fee
2	Jaya	5500
3	Teena	5000
4	Diksha	4500

3 rows in a set (0.00 sec)