15.2 Aggregate (Group) Functions

Till now you have learnt to work with functions that operate on individual rows in a table e.g., if you use **Round( )** function then it will round off values from each row of the table.

MySQL also supports and provides **group functions** or **aggregate functions**. As you can make out that the group functions or aggregate functions work upon groups of rows, rather than on single rows. That is why, these functions are sometimes also called **multiple row functions**.

Many group functions accept the following options :

DISTINCT This option causes a group function to consider only distinct values of the argument expression.

ALL This option causes a group function to consider all values including all duplicates.

The usage of these options will become clear with the coverage of examples in this section.

All the examples that we'll be using here, shall be based upon following table **empl**.

Table 15.1 Database table empl

EMPNO	ENAME	JOB	MGR	HIREDATE	SAL	COMM	DEPTNO
8369	SMITH	CLERK	8902	1990-12-18	800.00	NULL	20
8499	ANYA	SALESMAN	8698	1991-02-20	1600.00	300.00	30
8521	SETH	SALESMAN	8698	1991-02-22	1250.00	500.00	30
8566	MAHADEVAN	MANAGER	8839	1991-04-02	2985.00	NULL	20
8654	MOMIN	SALESMAN	8698	1991-09-28	1250.00	1400.00	30
8698	BINA	MANAGER	8839	1991-05-01	2850.00	NULL	30
8839	AMIR	PRESIDENT	NULL	1991-11-18	5000.00	NULL	10
8844	KULDEEP	SALESMAN	8698	1991-09-08	1500.00	0.00	30
8882	SHIAVNSH	MANAGER	8839	1991-06-09	2450.00	NULL	10
8886	ANOOP	CLERK	8888	1993-01-12	1100.00	NULL	20
8888	SCOTT	ANALYST	8566	1992-12-09	3000.00	NULL	20
8900	JATIN	CLERK	8698	1991-12-03	950.00	NULL	30
8902	FAKIR	ANALYST	8566	1991-12-03	3000.00	NULL	20
8934	MITA	CLERK	8882	1992-01-23	1300.00	NULL	10

1. AVG

This function computes the average of given data.

Syntax

```
AVG([DISTINCT | ALL] n)
```

❖ Returns average value of parameter(s) *n*.

Argument type : Numeric *Return value* : Numeric

Example 15.1. *Calculate average salary of all employees listed in table empl.*

Solution.

```
mysql> SELECT AVG(sal) "Average"
       FROM empl ;
```

```
+-------------+
| Average     |
+-------------+
| 2073.928571 |
+-------------+
1 row in set (0.01 sec)
```

2. COUNT

This function counts the number of rows in a given column or expression.

Syntax

```
COUNT({ * [DISTINCT | ALL] expr})
```

- Returns the number of rows in the query.
- If you specify argument *expr*, this function returns rows where *expr* is not null. You can count either all rows, or only distinct values of *expr*.
- If you specify the asterisk (*), this function returns all rows, including duplicates and nulls.

Argument type : Numeric *Retrun value* : Numeric

Example 15.2. *Count number of records in table empl.*

Solution.

```
mysql> SELECT COUNT(*) "Total"
       FROM empl ;
```

```
+----------+
| Total    |
+----------+
|   14     |
+----------+
1 row in set (0.00 sec)
```

Example 15.3. *Count number of jobs in table empl.*

Solution.

```
mysql> SELECT COUNT(job) "Job Count"
       FROM empl ;
```

```
+-------------+
| Job Count   |
+-------------+
|     14      |
+-------------+
1 row in set (0.01 sec)
```

Example 15.4. *How many distinct jobs are listed in table empl ?*

Solution.

```
mysql> SELECT COUNT(DISTINCT job) "Distinct Jobs"
       FROM empl ;
```

```
+----------------+
| Distinct Jobs  |
+----------------+
|       15       |
+----------------+
1 row in set (0.04 sec)
```

3. MAX

This function returns the maximum value from a given column or expression.

Syntax

```
MAX([DISTINCT| ALL] expr)
```

- Returns maximum value of argument *expr*.

Argument type : Numeric *Return value* : Numeric

Example 15.5. *Display maximum salary from table empl.*

Solution.

```
mysql> SELECT MAX(sal) "Maximum Salary"
       FROM empl ;
```

```
+---------------+
| Maximum Salary |
+---------------+
|    5000.00    |
+---------------+
1 row in set (0.01 sec)
```

4. MIN

This function returns the minimum value from a given column or expression.

Syntax

```
MIN([DISTINCT | ALL] expr)
```

❖ Returns minimum value of *expr*.

Argument type : Numeric *Return value* : Numeric

Example 15.6. Display the Joining date of seniormost employee.

Solution.

```
mysql> SELECT MIN(hiredate) "Minimum Hire Date"
       FROM empl ;
```

```
+------------------+
| Minimum Hire Date |
+------------------+
|      1990-12-18  |
+------------------+
1 row in set (0.06 sec)
```

5. SUM

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

Syntax

```
SUM([DISTINCT | ALL] n)
```

❖ Returns sum of values of *n*.

Argument type : Numeric *Return value* : Numeric

Example 15.7. Display total salary of all employees listed in table empl.

Solution.

```
mysql> SELECT SUM(sal) "Total Salary"
       FROM empl ;
```

```
+---------------+
| Total Salary  |
+---------------+
|    29035.00   |
+---------------+
1 row in set (0.01 sec)
```

Some more examples of group functions are being given below :

These functions are called *aggregate functions* because they operate on aggregates of tuples. The result of an aggregate function is a single value.

Examples :

1. To calculate the total gross for employees of grade 'E2', the command is :

```
SELECT SUM(gross)  FROM employee
WHERE grade = 'E2' ;
```

2. To display the average gross of employees with grades 'E1' or 'E2', the command used is :

```
SELECT AVG(gross)  FROM employee
WHERE (grade = 'E1' OR grade = 'E2' ) ;
```

3. To count the number of employees in *employee* table, the *SQL* command is :

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

4. To count the number of cities, the different members belong to, you use the following command :

```
SELECT COUNT(DISTINCT city) FROM members ;
```

Here the DISTINCT keyword ensures that multiple entries of the same *city* are ignored. The * is the only argument that includes NULLs when it is used only with COUNT, functions other than COUNT disregard NULLs in any case.

If you want to count the entries including repeats, the keyword ALL is used. The following command will COUNT the number of non NULL *city* fields in the *members* table :

```
SELECT COUNT(ALL city)
FROM members ;
```

In general, GROUP functions

- Return a single value for a set of rows
- Can be applied to any numeric values, and some Text types and DATE values.

15.3 Types of SQL Functions

Now that you have learnt different types of functions, let us talk about their broad categories. SQL supports many and many functions. All these functions can be generally categorized into following *two* types :

- Single Row (or Scalar) functions.
- Multiple Row (or Group or Aggregate) functions.

(*i*) ***Single Row functions*** work with a single row at a time. A single row function returns a result for every row of a queried table.

Examples of *Single row functions* are the functions that you learnt in Class XI *Text/Character functions* such as year( ), day( ), etc.

(*ii*) ***Multiple Row or Group functions*** work with data of multiple rows at a time and return aggregated value.

Examples of multiple row functions are the group functions that you have learnt in previous section *i.e.*, sum( ), count( ), max( ), min( ), Avg( ) etc.

The difference between these two types of functions is in the number of rows they act upon. A **single row function** works with the data of a single row at a time and returns a single result for each row queried upon ; a **multiple row function** works with a group of rows and returns a single result for that group.

15.4 Grouping Result – GROUP BY

The GROUP BY clause combines all those records that have identical values in a particular field or a group of fields. This grouping results into one summary record per group if group-functions are used with it. In other words, the GROUP BY clause is used in SELECT statements to divide the table into groups. Grouping can be done by a column name, or with aggregate functions in which case the aggregate produces a value for each group.

For example, to calculate the *number of employees in each grade*, you use the command

```
SELECT job, COUNT(*)
FROM empl
GROUP BY job ;
```

GROUP BY applies the aggregate functions independently to a series of groups that are defined by having a field value in common.

```
+-----------+----------+
|   job     | COUNT(*) |
+-----------+----------+
| ANALYST   |    2     |
| CLERK     |    4     |
| MANAGER   |    3     |
| PRESIDENT |    1     |
| SALESMAN  |    4     |
+-----------+----------+
```

Now consider the following query, which is also grouping records based on *deptno*.

```
mysql> SELECT deptno, COUNT(*), SUM(sal)
       FROM empl
       GROUP BY deptno ;
```

```
+---------+----------+------------+
| deptno  | COUNT(*) | SUM(sal)   |
+---------+----------+------------+
|  10     |        3 |  8750.00   |
|  20     |        5 | 10885.00   |
|  30     |        6 |  9400.00   |
+---------+----------+------------+
```

As you can make out that the above query is displaying count of records and sum of salaries in each group and the groups are formed on the basis of *deptno*. Thus, from the above output, you can make out that in department number 10, there are 3 employees (records) and total of all salaries is 8750.00 ; in department number 20, there are 5 employees and total of salaries is 10885.00 ; and so on.

15.4.1 Nested Groups – Grouping on Multiple Columns

With GROUP BY clause, you can create groups within groups. Such type of grouping is called *Nested grouping*. This can be done by specifying in GROUP BY expression, where the *first field* determines the *highest group level*, the *second field* determines the *second group level*, and so on. The *last field* determines the *lowest level of grouping*.

In order to fully understand this concept, consider Table **15.1 empl** on page 446.

See there are multiple records having same value for field **Deptno**, we can group records on the basis of field *Deptno*. For instance, if you want to count the number of employees in each group, you need to issue a query statement as given below :

```
SELECT COUNT(empno) FROM empl
GROUP BY Deptno ;
```

And the result produced by this query is

```
+---------------+
|count(empno)   |
+---------------+
|      3        |
|      5        |
|      6        |
+---------------+
```

But can you make out, these are employee-counts for which departments ? To get this information, you may modify the SELECT list as :

```
SELECT Deptno, COUNT(empno)
FROM empl
GROUP BY Deptno ;
```

Now the result will be :

```
+---------+--------------+
| deptno  | count(empno) |
+---------+--------------+
|  10     |      3       |
|  20     |      5       |
|  30     |      6       |
+---------+--------------+
```

See, now it is more clear. But one thing that you should keep in mind is that while grouping, you should include only those values in the *select list* that either have the same value for a group or contain a

group (aggregate) function *i.e.*, a **group-expression**. Like in the above query, the first expression **Deptno** field has one (same) value for a group and the other expression **COUNT(empno)** contains a group function.

MySQL as such would not create any error even if you include a **non-group expression** in the select-list. A **non-group field** (or expression) is the field that has different values in the rows belonging to the group.

In this case, it will return **the value from first record** of the group for that non-group field *e.g.*, if you issue command like :

```
SELECT deptno, count(empno), mgr
FROM empl
GROUP BY deptno ;
```

This is non-group field as it has multiple values for a group.

The output returned will be :

```
+---------+--------------+------+
| deptno  | count(empno) | mgr  |
+---------+--------------+------+
|   10    |      3       | NULL |
|   20    |      5       | 8902 |
|   30    |      6       | 8698 |
+---------+--------------+------+
```

See, first record of group with deptno 10 has value NULL in mgr field ; first record of group with deptno 20 has 8902 in mgr field ; and first record of group with deptno 30 has value 8698 in mgr field. Hence this output.

NOTE

In the select list of a group, only those fields or expressions state be included that either return single value for a group or are constants. Otherwise you may not get authentic results.

To create a group within a group *i.e.*, nested group, you need to specify multiple fields in the GROUP BY expression. If you have a look at the records of *Empl* table, you can make out that there exists a *group of jobs* within the *department group* as there are same values for **Job** field, in one department group's records.

To group records *job wise* within *Deptno* wise, you need to issue a query statement like :

```
SELECT Deptno, Job, COUNT(empno)
FROM empl
GROUP BY Deptno, Job ;
```

And the result produced is :

```
+--------+-----------+---------------+
| deptno |    job    | count(empno)  |
+--------+-----------+---------------+
|   10   | CLERK     |       1       |
|   10   | MANAGER   |       1       |
|   10   | PRESIDENT |       1       |
|   20   | ANALYST   |       2       |
|   20   | CLERK     |       2       |
|   20   | MANAGER   |       1       |
|   30   | CLERK     |       1       |
|   30   | MANAGER   |       1       |
|   30   | SALESMAN  |       4       |
+--------+-----------+---------------+
```

15.4.2 Placing Conditions on Groups – HAVING Clause

The HAVING clause places conditions on groups in contrast to WHERE clause that places conditions on individual rows. While WHERE conditions cannot include aggregate functions, HAVING conditions can do so.

Now consider the following query, which is also grouping records based on *deptno*.

```
mysql> SELECT deptno, COUNT(*), SUM(sal)
       FROM empl
       GROUP BY deptno ;
```

deptno	COUNT(*)	SUM(sal)
10	3	8750.00
20	5	10885.00
30	6	9400.00

As you can make out that the above query is displaying count of records and sum of salaries in each group and the groups are formed on the basis of *deptno*. Thus, from the above output, you can make out that in department number 10, there are 3 employees (records) and total of all salaries is 8750.00 ; in department number 20, there are 5 employees and total of salaries is 10885.00 ; and so on.

15.4.1 Nested Groups – Grouping on Multiple Columns

With GROUP BY clause, you can create groups within groups. Such type of grouping is called *Nested grouping*. This can be done by specifying in GROUP BY expression, where the *first field* determines the *highest group level*, the *second field* determines the *second group level*, and so on. The *last field* determines the *lowest level of grouping*.

In order to fully understand this concept, consider Table **15.1 empl** on page 446.

See there are multiple records having same value for field **Deptno**, we can group records on the basis of field *Deptno*. For instance, if you want to count the number of employees in each group, you need to issue a query statement as given below :

```
SELECT COUNT(empno) FROM empl
GROUP BY Deptno ;
```

And the result produced by this query is

count(empno)
3
5
6

But can you make out, these are employee-counts for which departments ? To get this information, you may modify the SELECT list as :

```
SELECT Deptno, COUNT(empno)
FROM empl
GROUP BY Deptno ;
```

Now the result will be :

deptno	count(empno)
10	3
20	5
30	6

See, now it is more clear. But one thing that you should keep in mind is that while grouping, you should include only those values in the *select list* that either have the same value for a group or contain a

group (aggregate) function *i.e.*, a **group-expression.** Like in the above query, the first expression **Deptno** field has one (same) value for a group and the other expression **COUNT(empno)** contains a group function.

MySQL as such would not create any error even if you include a **non-group expression** in the select-list. A **non-group field** (or expression) is the field that has different values in the rows belonging to the group.

In this case, it will return **the value from first record** of the group for that non-group field *e.g.*, if you issue command like :

```
SELECT deptno, count(empno), mgr
FROM empl
GROUP BY deptno ;
```

This is non-group field as it has multiple values for a group.

The output returned will be :

```
+--------+--------------+------+
| deptno | count(empno) | mgr  |
+--------+--------------+------+
|   10   |      3       | NULL |
|   20   |      5       | 8902 |
|   30   |      6       | 8698 |
+--------+--------------+------+
```

See, first record of group with deptno 10 has value NULL in mgr field ; first record of group with deptno 20 has 8902 in mgr field ; and first record of group with deptno 30 has value 8698 in mgr field. Hence this output.

NOTE

In the select list of a group, only those fields or expressions state be included that either return single value for a group or are constants. Otherwise you may not get authentic results.

To create a group within a group *i.e.*, nested group, you need to specify multiple fields in the GROUP BY expression. If you have a look at the records of *Empl* table, you can make out that there exists a *group of jobs* within the *department group* as there are same values for **Job** field, in one department group's records.

To group records *job wise* within *Deptno* wise, you need to issue a query statement like :

```
SELECT Deptno, Job, COUNT(empno)
FROM empl
GROUP BY Deptno, Job ;
```

And the result produced is :

```
+--------+-----------+---------------+
| deptno |    job    | count(empno)  |
+--------+-----------+---------------+
|   10   | CLERK     |       1       |
|   10   | MANAGER   |       1       |
|   10   | PRESIDENT |       1       |
|   20   | ANALYST   |       2       |
|   20   | CLERK     |       2       |
|   20   | MANAGER   |       1       |
|   30   | CLERK     |       1       |
|   30   | MANAGER   |       1       |
|   30   | SALESMAN  |       4       |
+--------+-----------+---------------+
```

15.4.2 Placing Conditions on Groups – HAVING Clause

The HAVING clause places conditions on groups in contrast to WHERE clause that places conditions on individual rows. While WHERE conditions cannot include aggregate functions, HAVING conditions can do so.

For example, to calculate the average gross and total gross for employees belonging to 'E4' grade, the command would be :

```
SELECT AVG(gross), SUM(gross)
FROM employee
GROUP BY grade
HAVING grade = 'E4' ;
```

This condition would be applicable on group and not on individual rows

To display the jobs where the number of employees is less than 3, you use the command :

```
SELECT JOB, COUNT(*)
FROM empl
GROUP BY job
HAVING count(*) < 3 ;
```

This will produce the following output :

```
+-----------+-----------+
|    JOB    | COUNT(*)  |
+-----------+-----------+
| ANALYST   |     2     |
| PRESIDENT |     1     |
+-----------+-----------+
2 rows in set (0.11 sec)
```

The HAVING clause can contain either a simple boolean expression (*i.e.*, an expression or condition that results into *true* or *false*) or use aggregate function in the having condition.

You can include more than one condition in HAVING clause, of course, by using logical operators.

Consider this :

```
SELECT Deptno, AVG(Comm), AVG(Sal)
FROM  empl
GROUP BY Deptno
HAVING AVG(Comm) > 750 AND
AVG(Sal) > 2000 ;
```

Also, you can use an aggregate function in the HAVING clause even if it is not in the SELECT list. Consider the following query to understand this :

```
SELECT Deptno, AVG(Sal)
FROM empl
GROUP BY Deptno
HAVING COUNT(*) <= 3 ;
```

You can also use IN or BETWEEN operators with HAVING clause. Following two queries illustrate this :

```
SELECT Deptno, Job, AVG(Sal)
FROM empl
GROUP BY Deptno, Job
HAVING JOB IN ('CLERK', 'SALESMAN') ;
SELECT Deptno, Job, SUM(sal)
FROM empl
GROUP BY Deptno, Job
HAVING SUM(sal) BETWEEN 3000 AND 7000 ;
```

NOTE

The WHERE clause is used to restrict records in a query (i.e., WHERE condition is applied on individual row before grouping) whereas HAVING clause restricts the records after they have been grouped (i.e., HAVING condition is applied on groups

15.4.3 Non-Group Expressions with GROUP BY

As mentioned before, if you include a non-group expression in the select-list of a query with GROUP BY, MySQL will not produce any error. Rather it will pick value of the specified non-group field from the

first row of the group. But we do not recommend this practice because it will produce ambiguous results. For instance, consider the following query and its output.

```
mysql>  SELECT ename, sum(sal)
        FROM empl
        GROUP BY deptno ;
```

```
+--------+----------+
| ename  | sum(sal) |
+--------+----------+
| AMIR   |  8750.00 |
| SMITH  | 10885.00 |
| ANYA   |  9400.00 |
+--------+----------+
```

See, isn't it conveying that AMIR's salary-sum is 8750.00, SMITH'S 10885.00 and ANYA'S 9400.00 ?

Thus, we recommend not to use non-group expressions in GROUP BY query unless otherwise necessary.

Let Us Revise

- *The GROUP BY clause combines all those records that have identical value in a particular field or a group of fields.*
- *GROUP BY clause is used to divide the result in groups.*
- *A group within another group is called **Nested Group**.*
- *Nested grouping can be done by providing multiple fields in the GROUP BY expression.*
- *All fields containing a NULL value are considered to have a value and are grouped to have a value and are grouped with the fields containing non-NULL values.*
- *The SELECT list of a group can include expressions returning single value per group or constants.*
- *The HAVING clause is used to specify filtering condition for groups.*
- *The difference between WHERE and HAVING clause is that WHERE conditions are applicable on individual rows whereas HAVING conditions are applicable on groups as formed by GROUP BY clause.*

15.5 Joins

A **join** is a query that combines rows from two or more tables. In a *join-query*, more than one table are listed in FROM clause. The function of combining data from multiple tables is called *joining*.

Consider the following query :

```
SELECT *
FROM EMP1, DEPT ;
```

This query will give you *the Cartesian product i.e.*, all possible concatenations are formed of all rows of both the tables EMP1 and DEPT. That is, when no particular rows (using WHERE clause) and columns (through SELECT list) are selected. Such an operation is also known as **Unrestricted Join**. It returns **n1** × **n2** rows where **n1** is number of rows in first table and **n2** is number of rows in second table.

NOTE

*In **unrestricted join** or **Cartesian product** of two tables, all possible concatenations are formed of all rows of both the tables.*

But joining is not this at all. It is much more useful in many situations where we need to derive data from more than one table.

Looking at the EMPL table we can see that we have a column called DEPTNO. Suppose we also have a table called DEPT which carries attributes such as department number, department name, and location of the department. Here is the data stored in that table :

```
SQL > SELECT * FROM DEPT ;
```

DEF

A **join** is a query that combines rows from two or more tables.

Table 15.2 *Dept*

DEPTNO	DNAME	LOC
10	ACCOUNTING	NEW DELHI
20	RESEARCH	CHENNAI
30	SALES	KOLKATA
40	OPERATIONS	MUMBAI
50	MARKETING	BANGLORE
60	PRODUCTION	AHMEDABAD

Now suppose we need to know the office location for the employee named 'ANOOP'.

In reviewing the DEPT table above we note that department 20 is in 'CHENNAI' but the table does not include the *employee's name field*.

By reviewing the EMPL table (Table 15.1) we can see that 'ANOOP' works in department 20 but the location of that department is not contained within this table.

Now the question is : *How do we get this information without doing two separate SELECT statements* ? And the answer is *Join-Query*. Let us see how we can do this.

Since both the EMPL table and the DEPT table carry the column DEPTNO, we can join the tables on this common column and relate rows from the DEPT table with rows in the EMPL table. In doing this, we are creating a virtual table (a table created in main memory), which contains all the attributes for rows from the DEPT table and the EMPL table where the DEPTNO is equal. This virtual table contains the following attributes : DEPTNO, DNAME, LOC, EMPNO, ENAME, JOB, MGR, HIREDATE, SAL, COMM.

In MySQL we are able to join a multitude of tables together in this fashion. Although it may not be practical it is possible. Be aware that the more tables we join, the more complicated the query becomes.

In order to find the location of the employee named ANOOP, we would use the following query.

```
mysql>  SELECT ENAME, LOC
        FROM EMPL, DEPT
        WHERE ENAME = 'ANOOP'
        AND EMPL.DEPTNO = DEPT.DEPTNO ;        -- see this is join condition
```

```
+--------+----------+
| ename  | loc      |
+--------+----------+
| ANOOP  | CHENNAI  |
+--------+----------+
1 row in set (0.13 sec)
```

The virtually joined table that we created, consists of one row that meets the specifications of the WHERE clause (ENAME = 'ANOOP'). Since we selected only ENAME and LOC fields, only those columns were returned. However, we can retrieve as many attributes from each table as we need.

Thus we can say that a *join* is used when an SQL query requires data from more than one table on database. *Joins* can compare two or more tables (or views) by specifying a column from each, comparing the values in those columns row by row and concatenating rows that have matching values. The FROM clause is used to specify which tables and views have to be joined. The columns being joined must have same or comparable datatypes.

Let us consider one more example. If you want to know the name, job, deptno, deptname and department location of all the employees, what would you do ? Yes, you are right : you have to use joins and you need to join the two tables **empl** and **dept** on the basis of their common field **deptno**. And the query will be :

Example 15.8. *Write query to join two tables empl and dept on the basis of field deptno.*

```
mysql> SELECT * FROM empl, dept
    ->WHERE empl.deptno = dept.deptno ;
```

And the output produced by above join query will be :

See two identical columns

empno	ename	job	mgr	hiredate	sal	comm	deptno	deptno	dname	loc
8369	SMITH	CLERK	8902	1990-12-18	800.00	NULL	20	20	RESEARCH	CHENNAI
8499	ANYA	SALESMAN	8698	1991-02-20	1600.00	300.00	30	30	SALES	KOLKATA
8521	SETH	SALESMAN	8698	1991-02-22	1250.00	500.00	30	30	SALES	KOLKATA
8566	MAHADEVAN	MANAGER	8839	1991-04-02	2985.00	NULL	20	20	RESEARCH	CHENNAI
8654	MOMIN	SALESMAN	8698	1991-09-28	1250.00	1400.00	30	30	SALES	KOLKATA
8698	BINA	MANAGER	8839	1991-05-01	2850.00	NULL	30	30	SALES	KOLKATA
8839	AMIR	PRESIDENT	NULL	1991-11-18	5000.00	NULL	10	10	ACCOUNTING	NEW DELHI
8844	KULDEEP	SALESMAN	8698	1991-09-08	1500.00	0.00	30	30	SALES	KOLKATA
8882	SHIAVNSH	MANAGER	8839	1991-06-09	2450.00	NULL	10	10	ACCOUNTING	NEW DELHI
8886	ANOOP	CLERK	8888	1993-01-12	1100.00	NULL	20	20	RESEARCH	CHENNAI
8888	SCOTT	ANALYST	8566	1992-12-09	3000.00	NULL	20	20	RESEARCH	CHENNAI
8900	JATIN	CLERK	8698	1991-12-03	950.00	NULL	30	30	SALES	KOLKATA
8902	FAKIR	ANALYST	8566	1991-12-03	3000.00	NULL	20	20	RESEARCH	CHENNAI
8934	MITA	CLERK	8882	1992-01-23	1300.00	NULL	10	10	ACCOUNTING	NEW DELHI

14 rows in set (0.01 sec)

Hey, did you notice, there are two fields by the name deptno – one deptno from table **empl** and one from table **dept**. This happened because we specified * (asterisk) in the *select-list* of our join query. If you specifically mention the field names in the *select-list* of join query, then only the specified fields will appear. That is, if you change above query to following query then duplicate **deptno** column will get removed :

```
SELECT empl.*, dname, loc
FROM empl, dept
WHERE empl.deptno = dept.deptno ;
```

This means that *empl.** *i.e.*, all the fields of table *empl* along with *dname*, *loc* fields (from table dept) should appear in the output.

Qualified Names

Did you notice that in all the WHERE conditions of Join-queries given so far, the field names are given as :

```
<tablename>.<fieldname>
```

This type of field names are called **qualified field names**. Qualified field names are very useful in identifying a field if the two joining tables have fields with same name. For example, if we say **deptno** field from joining tables **empl** and **dept**, you'll definitely ask – '**deptno** field of which table ?' To avoid such an ambiguity, the qualified field names are used.

Therefore, in the above given queries, the columns that have similar names in both the tables have been given *qualified names i.e.*, *table-name.column-name*, to avoid ambiguity. If the tables do not have the columns with the same name, using table names as qualifiers in the SELECT statement is optional.

NOTE

Qualifying a column name uniquely identifies it from similar columns in other tables.

The FROM clause lists all the tables involved and WHERE clause specifies the join. Without a WHERE clause, each row of the first table will be joined with every row of the second table which results in a *Cartesian Product*. Multiple conditions can be incorporated in the WHERE clause using the logical operators AND, OR and NOT. And the relational operators =, >, <, >=, <=, < >are used as join operators.

Now consider the following example.

Example 15.9. Display details like department number, department name, employee number, employee name, job and salary. And order the rows by employee number with department number.

Solution.

```
mysql> SELECT EMPL.DEPTNO, DNAME, EMPNO, ENAME, JOB, SAL
    FROM EMPL, DEPT
    WHERE EMPL.DEPTNO = DEPT.DEPTNO
    ORDER BY EMPL.DEPTNO, EMPNO ;                    -- Join on DEPTNO
```

The output produced by this query will be as shown below :

```
+------+----------+------+---------+---------+---------+
|deptno |dname     |empno |  ename  |   job   |sal      |
+------+----------+------+---------+---------+---------+
| 10   | ACCOUNTING | 8839 | AMIR    | PRESIDENT | 5000.00 |
| 10   | ACCOUNTING | 8882 | SHIAVNSH | MANAGER | 2450.00 |
| 10   | ACCOUNTING | 8934 | MITA    | CLERK   | 1300.00 |
| 20   | RESEARCH   | 8369 | SMITH   | CLERK   |  800.00 |
| 20   | RESEARCH   | 8566 | MAHADEVAN | MANAGER | 2985.00 |
| 20   | RESEARCH   | 8886 | ANOOP   | CLERK   | 1100.00 |
| 20   | RESEARCH   | 8888 | SCOTT   | ANALYST | 3000.00 |
| 20   | RESEARCH   | 8902 | FAKIR   | ANALYST | 3000.00 |
| 30   | SALES      | 8499 | ANYA    | SALESMAN | 1600.00 |
| 30   | SALES      | 8521 | SETH    | SALESMAN | 1250.00 |
| 30   | SALES      | 8654 | MOMIN   | SALESMAN | 1250.00 |
| 30   | SALES      | 8698 | BINA    | MANAGER | 2850.00 |
| 30   | SALES      | 8844 | KULDEEP | SALESMAN | 1500.00 |
| 30   | SALES      | 8900 | JATIN   | CLERK   |  950.00 |
+------+----------+------+---------+---------+---------+
14 rows in set (0.01 sec)
```

15.5.1 Using Table Aliases

Consider once again the query given in example 15.9. Don't you find something unusual about this query ? Since we wanted to list DEPTNO as part of the output and this attribute resides within each table, we had to tell MySQL which table (it doesn't matter which one) we wanted to use to extract DEPTNO. When you have an attribute, which resides in more than one table if you do not specify the table in the SELECT statement you will get an ambiguity error. For example, had we given the following query for example 15.9 :

```
mysql> SELECT deptno, dname, empno, ename, job, sal
    -> FROM empl, dept
    -> WHERE empl.deptno = dept.deptno
    -> ORDER BY empl.deptno, empno ;
```

ambiguous column (from which table ? – not specified)

MySQL would have shown us the error as given below :

```
Column 'deptno' in field list is ambiguous
```

DEF

A Table Alias is a temporary **label** given along with table name in FROM clause.

That means every time you have to refer to a column, which is in more than one table participating in the join query, you have to *qualify the column name by* giving its table name along with it. Doesn't this method require more typing since for every such column, table name is to be typed ? But don't worry, there is a shorter way of doing so – *the table aliases*. A table alias can be used anywhere in the SELECT statement.

To cut down on the amount of typing required in your queries you can use aliases for table names in the SELECT and WHERE clauses. For example, if you wanted to use the abbre- viation 'E' for the EMPL table in your query all you need to is tell MySQL that EMPL will be referenced by E in the FROM clause. The following queries would result in the same output :

```
mysql> SELECT E.DEPTNO, DNAME, EMPNO, ENAME, JOB, SAL
       FROM EMPL E, DEPT
       WHERE E.DEPTNO = DEPT.DEPTNO                    -- table alias E used for table EMPL.
       ORDER BY E.DEPTNO, EMPNO ;                      -- join on field DEPTNO
```

Table alias

```
mysql> SELECT E.DEPTNO, DNAME, EMPNO, ENAME, JOB, SAL
       FROM EMPL E, DEPT D
                                                       -- table alias E for table EMPL,
       WHERE E.DEPTNO = D.DEPTNO                       -- table alias D for table DEPT
       ORDER BY E.DEPTNO, EMPNO ;                      -- join on field DEPTNO
```

15.5.2 Additional Search Conditions in Joins

Once you have joined tables, you can filter information from joined table by incorporating additional search conditions. For example, refer to *example 15.9* once again. If you want to extract such information only for SALES department, you can do so by giving a query similar to the one given in example 15.10.

Example 15.10. *Refer to example 15.9. Do this only for SALES department.*

Solution.

```
mysql> SELECT E.DEPTNO, DNAME, EMPNO, ENAME, JOB, SAL
       FROM EMPL E, DEPT D  WHERE E.DEPTNO = D.DEPTNO
       AND DNAME = 'SALES'  ORDER BY E.DEPTNO, EMPNO ;
```

```
+-------+-------+------+-------+---------+--------+
|deptno | dname | empno| ename |   job   |  sal   |
+-------+-------+------+-------+---------+--------+
|  30   | SALES | 8499 | ANYA  | SALESMAN|1600.00 |
|  30   | SALES | 8521 | SETH  | SALESMAN|1250.00 |
|  30   | SALES | 8654 | MOMIN | SALESMAN|1250.00 |
|  30   | SALES | 8698 | BINA  | MANAGER |2850.00 |
|  30   | SALES | 8844 | KULDEEP| SALESMAN|1500.00 |
|  30   | SALES | 8900 | JATIN | CLERK   | 950.00 |
+-------+-------+------+-------+---------+--------+
```

Example 15.11. *Display details like department number, department name, employee number, employee name, job, and salary. And order the rows by employee number with department number. These details should be only for employees earning atleast Rs. 1500 and of SALES department.*

Solution.

```
mysql> SELECT E.DEPTNO, DNAME, EMPNO, ENAME, JOB, SAL
       FROM EMPL E, DEPT D  WHERE E.DEPTNO = D.DEPTNO
       AND DNAME = 'SALES'  AND SAL >= 1500
       ORDER BY E.DEPTNO, EMPNO ;
```

```
+-------+-------+------+-------+---------+--------+
|deptno | dname | empno| ename |   job   |  sal   |
+-------+-------+------+-------+---------+--------+
|  30   | SALES | 8499 | ANYA  | SALESMAN|1600.00 |
|  30   | SALES | 8698 | BINA  | MANAGER |2850.00 |
|  30   | SALES | 8844 | KULDEEP| SALESMAN|1500.00 |
+-------+-------+------+-------+---------+--------+
```

15.5.3 Joining More Than Two Tables

Sometimes you need to extract information from more than two tables. In this case, you need to join more than two tables. This can be achieved by specifying all the required table names in FROM clause and by providing all join conditions using AND operator. To understand this, consider the following example.

Example 15.12. *Display the employee details in the following format :*

```
Ename Department Job Sal Grade.
```

Here the Department is department and Grade is the salary grade. The salary grades as per salary ranges are given in table SalaryGrade that contains following information.

Table SalaryGrade

GRADE	LOSAL	HISAL
.....	.....	.....
1	700	1200
2	1201	1400
3	1401	2000
4	2001	3000
5	3001	9999

That is, the salaries falling between range specified with losal and hisal have the corresponding grade e.g., salaries falling between 700 to 1200 have grade 1 ; between 1201 to 1400, have grade 2 ; and so on.

Frame an SQL query to attain the desired result.

Solution.

```
mysql> SELECT ENAME, DNAME, JOB, SAL, GRADE
       FROM EMPL E, DEPT D, SalaryGrade S
       WHERE E.DEPTNO = D.DEPTNO                    -- first join condition
       AND SAL BETWEEN LOSAL AND HISAL              -- second join condition
       ORDER BY E.DEPTNO, EMPNO ;
```

And the output that you receive is as desired :

```
+----------+----------+----------+---------+-------+
| ename    | dname    | job      | sal     | grade |
+----------+----------+----------+---------+-------+
| AMIR     | ACCOUNTING | PRESIDENT | 5000.00 |   5   |
| SHIAVNSH | ACCOUNTING | MANAGER   | 2450.00 |   4   |
| MITA     | ACCOUNTING | CLERK     | 1300.00 |   2   |
| SMITH    | RESEARCH   | CLERK     |  800.00 |   1   |
| MAHADEVAN | RESEARCH  | MANAGER   | 2985.00 |   4   |
| ANOOP    | RESEARCH   | CLERK     | 1100.00 |   1   |
| SCOTT    | RESEARCH   | ANALYST   | 3000.00 |   4   |
| FAKIR    | RESEARCH   | ANALYST   | 3000.00 |   4   |
| ANYA     | SALES      | SALESMAN  | 1600.00 |   3   |
| SETH     | SALES      | SALESMAN  | 1250.00 |   2   |
| MOMIN    | SALES      | SALESMAN  | 1250.00 |   2   |
| BINA     | SALES      | MANAGER   | 2850.00 |   4   |
| KULDEEP  | SALES      | SALESMAN  | 1500.00 |   3   |
| JATIN    | SALES      | CLERK     |  950.00 |   1   |
+----------+----------+----------+---------+-------+
```

Notice that the second join condition is not equality based. We can have join conditions based on equality and other operators also. Depending upon the way these joins are created, they are called differently. Coming sections discuss such types of joins.

15.5.4 Equi Join

In an *equijoin*, the values in the columns being joined are *compared for equality*. All the columns in the tables-being-joined are included in the results. Consider the following example.

Recall the output of example 15.8 on page 455 that has double **deptno** fields. This was the result of an **equi-join**.

In equi-join, all the columns from joining table appear in the output even if they are identical.

DEF
The Join, in which columns are compared for equality, is called **Equi-Join.**

15.5.5 Non-Equi-Joins

The comparison operator used in the join condition in Example 15.9 was the equals sign, "=". Such join queries are called *equi-joins*. A *non-equi-join* is a query that specifies some relationship other than equality between the columns.

If you refer to *example 15.12* carefully, you'll find that the second join condition is the example of a non-equi-join. If we had to join only EMPL table and SalaryGrade table, it would have been a non equijoin. This has been illustrated in example 15.13.

Example 15.13. *Display employee details for Analysts in the following format.*

Ename Job Sal Grade

Solution.

```
mysql>  SELECT ENAME, JOB, SAL, GRADE
        FROM EMPL E, SalaryGrade S
        WHERE SAL BETWEEN LOSAL AND HISAL      -- non-equijoin condition
        AND JOB = 'ANALYST' ;
```

```
+-------+--------+---------+------+
| ename |  job   |   sal   | grade|
+-------+--------+---------+------+
| SCOTT | ANALYST | 3000.00 |   4  |
| FAKIR | ANALYST | 3000.00 |   4  |
+-------+--------+---------+------+
```

15.5.6 Natural Join

By definition, the results of an equijoin contain two identical columns. One of the two identical columns can be eliminated by restating the query. This result is called a ***Natural Join*** (Equi join minus one of the two identical columns).

For example, if you restate the query of example 15.8 as :

```
SELECT   empl.*, dname, loc
FROM     empl, Dept
WHERE    empl.deptno = dept.dept no ;
```

See this time no duplicate column

DEF
The Join in which only one of the identical columns (coming from joined tables) exists, is called **Natural Join.**

empno	ename	job	mgr	hiredate	sal	comm	deptno	dname	loc
8369	SMITH	CLERK	8902	1990-12-18	800.00	NULL	20	RESEARCH	CHENNAI
8499	ANYA	SALESMAN	8698	1991-02-20	1600.00	300.00	30	SALES	KOLKATA
8521	SETH	SALESMAN	8698	1991-02-22	1250.00	500.00	30	SALES	KOLKATA
8566	MAHADEVAN	MANAGER	8839	1991-04-02	2985.00	NULL	20	RESEARCH	CHENNAI
8654	MOMIN	SALESMAN	8698	1991-09-28	1250.00	1400.00	30	SALES	KOLKATA
8698	BINA	MANAGER	8839	1991-05-01	2850.00	NULL	30	SALES	KOLKATA
8839	AMIR	PRESIDENT	NULL	1991-11-18	5000.00	NULL	10	ACCOUNTING	NEW DELHI
8844	KULDEEP	SALESMAN	8698	1991-09-08	1500.00	0.00	30	SALES	KOLKATA
8882	SHIAVNSH	MANAGER	8839	1991-06-09	2450.00	NULL	10	ACCOUNTING	NEW DELHI
8886	ANOOP	CLERK	8888	1993-01-12	1100.00	NULL	20	RESEARCH	CHENNAI
8888	SCOTT	ANALYST	8566	1992-12-09	3000.00	NULL	20	RESEARCH	CHENNAI
8900	JATIN	CLERK	8698	1991-12-03	950.00	NULL	30	SALES	KOLKATA
8902	FAKIR	ANALYST	8566	1991-12-03	3000.00	NULL	20	RESEARCH	CHENNAI
8934	MITA	CLERK	8882	1992-01-23	1300.00	NULL	10	ACCOUNTING	NEW DELHI

14 rows in set (0.01 sec)

15.5.7 Joining Tables Using JOIN Clauses of SQL SELECT

Till now you have learnt how to create joined-tables using traditional SQL style. MySQL also supports special clauses – **JOIN (and NATURAL JOIN) clause** – as part of SQL SELECT statement. The syntax for using JOIN clasue of SQL SELECT :

```
SELECT *
FROM <table1>
[CROSS][NATURAL] JOIN <table2>
[ON (<Join-Condition>) | USING (<joinfields>)] ;
```

It works as follows :

- To create **Cartesian product** of two tables (say **empl** and **dept**) write query as :

```
SELECT *
FROM empl JOIN Dept ;
```

If no join-condition is specified with a JOIN clause, cartesian-product is output.

Cross Join

You can also produce cartesian product of two tables using CROSS JOIN clause as follows :

```
SELET *
FROM empl CROSS JOIN dept ;
```

The above query will produce cartesian product of tables *empl* and *dept*.

The cartesian product of two tables is also known as **CROSS JOIN**. The **cross join** (or **cartesian product**) is a very basic type of join that simply matches each row from one table to every row from another table.

DEF

The **cross join** (or cartesian product) is a very basic type of join that simply matches each row from one table to every row from another table.

Cross join, by default, does not apply any filtering condition on the joined data. However, if you want to filter, you can use WHERE clause of SELECT query.

- To create equijoin on some common field (say **deptno**) of two tables (**empl** and **dept** here), write query as :

```
SELECT *
FROM empl e
JOIN dept d
ON (e.deptno = d.deptno) ;
```

Join condition is specified with ON clause.

You may write above query without table-aliases *i.e.*, as

```
SELECT *
FROM empl JOIN dept
ON (empl.deptno = dept.deptno) ;
```

Both the above queries will produce output similar to that of Example 15.8 on page 455.

Natural Join

As you know that equi-join has two identical columns, to remove which you specify the desired columns in the **select-list**. However, MySQL provides a NATURAL JOIN clause for you that does the same for you that is, one of the duplicate columns get removed from the output.

NOTE

Please note that with NATURAL JOIN clause, you need not specify a JOIN-Condition. It will automatically create natural-join through appropriate common field.

All you need to do is to write :

```
SELECT *
FROM <table1> ;
NATURAL JOIN <table2> ;
```

That is, above query will now become :

```
SELECT *
FROM empl NATURAL JOIN dept ;
```

with NATURAL JOIN clause, you need not specify Join Condition

ering Output

If you want to further filter the records from a join-output, you can write the condition with a WHERE clause. For instance, following query will first create a natural join from tables **empl** and **dept** and then extract only those records where **sal** is more than 2000 ;

```
mysql>  SELECT * FROM empl e
        NATURAL JOIN dept d
        WHERE sal > 2000 ;
```

*Two joining tables whose **Natural Join** is being produced.*

deptno	empno	ename	job	mgr	hiredate	sal	comm	dname	loc
20	8566	MAHADEVAN	MANAGER	8839	1991-04-02	2985.00	NULL	RESEARCH	CHENNAI
30	8698	BINA	MANAGER	8839	1991-05-01	2850.00	NULL	SALES	KOLKATA
10	8839	AMIR	PRESIDENT	NULL	1991-11-18	5000.00	NULL	ACCOUNTING	NEW DELHI
10	8882	SHIAVNSH	MANAGER	8839	1991-06-09	2450.00	NULL	ACCOUNTING	NEW DELHI
20	8888	SCOTT	ANALYST	8566	1992-12-09	3000.00	NULL	RESEARCH	CHENNAI
20	8902	FAKIR	ANALYST	8566	1991-12-03	3000.00	NULL	RESEARCH	CHENNAI

Similarly, you can use WHERE clause with JOIN to filter records from an equi-join. For example, following query will first create equi-join from tables *empl* and *dept* and then filter those records where *sal* is < 2000.

```
mysql>  SELECT * FROM empl JOIN dept  ON (empl.deptno = dept.deptno)
   ->   WHERE sal < 2000 ;
```

empno	ename	job	mgr	hiredate	sal	comm	deptno	dname	loc
8369	SMITH	CLERK	8902	1990-12-18	800.00	NULL	20	RESEARCH	CHENNAI
8499	ANYA	SALESMAN	8698	1991-02-20	1600.00	300.00	30	SALES	KOLKATA
8521	SETH	SALESMAN	8698	1991-02-22	1250.00	500.00	30	SALES	KOLKATA
8654	MOMIN	SALESMAN	8698	1991-09-28	1250.00	1400.00	30	SALES	KOLKATA
8844	KULDEEP	SALESMAN	8698	1991-09-08	1500.00	0.00	30	SALES	KOLKATA
8886	ANOOP	CLERK	8888	1993-01-12	1100.00	NULL	20	RESEARCH	CHENNAI
8900	JATIN	CLERK	8698	1991-12-03	950.00	NULL	30	SALES	KOLKATA
8934	MITA	CLERK	8882	1992-01-23	1300.00	NULL	10	ACCOUNTING	NEW DELHI

```
8 rows in set (0.01 sec)
```

You can specify join fields through **USING** clause also in place of **ON** clause, but with **USING** clause you just have to specify the name of join-field whereas with **ON** clause, join-condition is specified.

That is above query can also be reframed as :

```
SELECT *
FROM empl
JOIN dept
USING (deptno) ;
```

USING subclause with JOIN clause produces a ***Natural join*** whereas ON subclause with JOIN clause produces ***equi-join***.

NOTE

Difference between ON and USING sub-clauses of JOIN clause of SELECT is that ON clause requires a complete join-condition whereas USING clause requires just the name of a join field. USING subclause produces natural join whereas ON clause produces equi-join.

15.5.7A Left, Right Joins

When you join tables based on some condition, you may find that only some, not all rows from either table match with rows of other table. When you display an equi-join or natural-join, it shows only the matched rows. What if you want to know which all rows from a table did not match with other. In such a case, MySQL left or right JOIN can be very helpful and it refers to the order in which the tables are put together and the results are displayed.

Left Join

You can use LEFT JOIN clause of SELECT to produce left join *i.e.*, as :

```
SELECT <select-list>
FROM <table1> LEFT JOIN <table2>
ON <joining-condition> ;
```

When using LEFT JOIN all rows from the first table will be returned whether there are matches in the second table or not. For unmatched rows of first table, NULL is shown in columns of second table. For example, if table **members** contains members' main infor- mation and table **options** contains members' optional information, any records in **options** table would be tied to a particular *id* in the **members** table.

```
SELECT name, lastname, optonalinfo
FROM members LEFT JOIN options ON members.id = options.id ;
```

The result of this query would return *name* and *lastname* values from the **members** table and all available values from the **options** table. NULL is returned for non-existing values in **options** table. Consider the tables *empl* and *dept*. Table dept has now the following records (notice new records)

```
+--------+------------+-----------+
| deptno |   dname    |    loc    |
+--------+------------+-----------+
|   10   | ACCOUNTING | NEW DELHI |
|   20   | RESEARCH   | CHENNAI   |
|   30   | SALES      | KOLKATA   |
|   40   | OPERATIONS | MUMBAI    |
|   50   | MARKETING  | BANGLORE  |
|   60   | PRODUCTION | AHMEDABAD |
|   70   | EXPORT     | MUMBAI    |
|   80   | IMPORT     | LUCKNOW   |
+--------+------------+-----------+
```

Aashi

Now if we write a query as shown below :

```
SELECT dept.*, ename, mgr FROM dept LEFT JOIN empl
ON dept.deptno = empl.deptno ;
```

This is the first table, hence all the rows from dept table will be returned and unmatched rows will be filled with NULL for the right table's columns.

It will produce output that follows :

```
+--------+------------+-----------+--------+------+
| deptno | dname      | loc       | ename  | mgr  |
+--------+------------+-----------+--------+------+
|     10 | ACCOUNTING | NEW DELHI | AMIR   | NULL |
|     10 | ACCOUNTING | NEW DELHI | SHIAVNSH | 8839 |
|     10 | ACCOUNTING | NEW DELHI | MITA   | 8882 |
|     10 | RESEARCH   | CHENNAI   | SMITH  | 8902 |
|     20 | RESEARCH   | CHENNAI   | MAHADEVAN | 8839 |
|     20 | RESEARCH   | CHENNAI   | ANOOP  | 8888 |
|     20 | RESEARCH   | CHENNAI   | SCOTT  | 8566 |
|     20 | RESEARCH   | CHENNAI   | FAKIR  | 8566 |
|     20 | RESEARCH   | KOLKATA   | ANYA   | 8698 |
|     20 | SALES      | KOLKATA   | SETH   | 8698 |
|     30 | SALES      | KOLKATA   | MOMIN  | 8698 |
|     30 | SALES      | KOLKATA   | BINA   | 8839 |
|     30 | SALES      | KOLKATA   | KULDEEP | 8698 |
|     30 | SALES      | KOLKATA   | JATIN  | 8698 |
|     30 | SALES      | MUMBAI    | NULL   | NULL |
|     30 | OPERATIONS | BANGLORE  | NULL   | NULL |
|     40 | MARKETING  | AHMEDABAD | NULL   | NULL |
|     50 | PRODUCTION | MUMBAI    | NULL   | NULL |
|     60 | EXPORT     | LUCKNOW   | NULL   | NULL |
|     70 | IMPORT     |           |        |      |
|     80 |            |           |        |      |
+--------+------------+-----------+--------+------+
19 rows in set (0.00 sec)
```

See, how MySQL works on it.

MySQL starts with the left table (*dept* here). For each row from the table *dept* MySQL scans the table pl, finds the deptno of the employee and returns employee name(ename) and manager's code(mgr). en the employee name(ename) and manager's code(mgr) are joined with the matching row from the ble *dept*. For unmatched rows, it returns NULL for ename and mgr fields.

ght Join

RIGHT JOIN works just like the LEFT JOIN but with table order reversed. All rows from the second ble are going to be returned whether or not there are matches in the first table. Right join is produced in e same manner as that of left join, just you need to replace the keyword LEFT with RIGHT.

You can use RIGHT JOIN clause of SELECT to produce right join *i.e.,* as per follwoing syntax :

```
SELECT <select-list>
FROM <table1> RIGHT JOIN <table2>
ON <joining-condition> ;
```

That is to produce right join of **members** and **options** table, your query will be :

```
SELECT name, lastname, optonalinfo
FROM members RIGHT JOIN options ON members.id = options.id ;
```

15.6 Using DATA from Multiple Tables : Union & Intersection

Other than joins, other ways of using data from multiple tables are *cartesian product, union* and *intersection*. All these methods have been explained in **Appendix D**, *page 562*.

HOTS (Higher Order Thinking Skills)

SOLVED PROBLEMS

1. *Examine the description of the EMPLOYEES table :*

```
EMP_ID NUMBER(4) NOT NULL
LAST_NAME VARCHAR(30) NOT NULL
FIRST_NAME VARCHAR(30)
DEPT_ID NUMBER(2)
JOB_CAT VARCHAR(30)
SALARY NUMBER(8,2)
```

Which statement shows the maximum salary paid in each job category of each department?

A.
```
SELECT dept_id, job_cat, MAX(salary)
FROM employees
WHERE salary > MAX(salary);
```

B.
```
SELECT dept_id, job_cat, MAX(salary)
FROM employees
GROUP BY dept_id, job_cat;
```

C.
```
SELECT dept_id, job_cat, MAX(salary)
FROM employees;
```

D.
```
SELECT dept_id, job_cat, MAX(salary)
FROM employees
GROUP BY dept_id;
```

E.
```
SELECT dept_id,
GROUP BY dept_id, job_cat, salary;
```

Solution. B

The answer *B* provides correct syntax and semantics to show the maximum salary paid in each job category of each department.

Answer **A** is *incorrect,* because this query will not return any row because condition *SALARY > MAX(SALARY)* is FALSE.

Answer **C** is *incorrect*, because this query will return error because you cannot show maximum salary with DEPT_ID and JOB_CAT without grouping by these columns.

Answer **D** is *incorrect*, because the GROUP BY clause is missing JOB_ID column.

Answer **E** is *incorrect*, because you don't need to group results of query by SALARY in the GROUP BY column.

2. *Examine the structure of the EMPL and DEPT tables:*

EMPL

Column name	Data type	Remarks
EMPLOYEE_ID	NUMBER	NOT NULL, Primary Key
EMP_NAME	VARCHAR(30)	
JOB_ID	VARCHAR(20)	
SALARY	NUMBER	
MGR_ID	NUMBER	References EMPLOYEE_ID COLUMN
DEPARTMENT ID	NUMBER	Foreign key to DEPARTMENT ID column of the DEPT table

DEPT

Column name	Data type	Remarks
DEPARTMENT_ID	NUMBER	NOT NULL, Primary Key
DEPARTMENT_NAME	VARCHAR(30)	
MGR_ID	NUMBER	References MGR_ID column of the EMPL table

Evaluate this SQL statement:

```
SELECT employee_id, e.department_id, department_name, salary
FROM EMPL e, DEPT d
WHERE e.department_id = d.department_id;
```

Which SQL statement is equivalent to the above SQL statement?

A.
```
SELECT employee_id, department_id, department_name, salary
FROM EMPL
WHERE department_id MATCHES department_id of DEPT;
```

B.
```
SELECT employee_id, department_id, department_name,salary
FROM EMPL
NATURAL JOIN DEPT;
```

C.
```
SELECT employee_id, d.department_id, department_name,salary
FROM EMPL e
JOIN DEPT d
ON e.department_id = d.department_id;
```

D.
```
SELECT employee_id, department_id, department_name,Salary
FROM EMPL
JOIN DEPT
USING (e.department_id, d.department_id);
```

Solution. C. The query C shows correct JOIN ON clause syntax and provides equivalent to the above SQL statement.

3. *The EMPL table contains these columns:*

LAST_NAME	VARCHAR(25)
SALARY	NUMBER(6,2)
DEPARTMENT_ID	NUMBER(6)

You need to display the employees who have not been assigned to any department. You write the SELECT statement:

```
SELECT LAST_NAME, SALARY, DEPARTMENT_ID
FROM EMPL, DEPT
WHERE DEPARTMENT_ID = NULL;
```

But the query is not producing the result. Identify the problem and suggest the solution.

Solution. The operator in the WHERE clause is the problem.

The operator in the WHERE clause should be changed to display the desired results. The correct operator for *NULL comparison is IS* operator, so the correct query would be :

```
SELECT LAST_NAME, SALARY, DEPARTMENT_ID
FROM EMPL, DEPT
WHERE DEPARTMENT_ID IS NULL;
```

4. *Examine the description of the MARKS table :*

```
STD_ID NUMBER(4)
STUDENT_NAME VARCHAR(30)
SUBJ1 NUMBER(3)
SUBJ2 NUMBER(3)
```

SUBJ1 and SUBJ2 indicate the marks obtained by a student in two subjects. Examine this SELECT statement based on the MARKS table :

```
SELECT subj1 + subj2 total_marks, std_id
FROM marks
WHERE subj1 > AVG(subj1) AND subj2 > AVG(subj2)
ORDER BY total_marks;
```

What is the result of the SELECT statement?

A. *The statement executes successfully and returns the student ID and sum of all marks for each student who obtained more than the average mark in each subject.*

B. *The statement returns an error at the SELECT clause.*

C. *The statement returns an error at the WHERE clause.*

D. *The statement returns an error at the ORDER BY clause.*

Solution C. The statement returns an error at the WHERE clause.

The given SQL statement returns an error at the WHERE clause because group function AVG() cannot be used in the WHERE clause. Group functions can be used in SELECT clause and GROUP BY clause. They allow us to perform data operations on several values in a column of data as though the column were one collective group of data.

5. *Examine the structure of the EMPLOYEES, DEPARTMENTS, and LOCATIONS tables.*

EMPLOYEES

EMPLOYEE_ID	NUMBER	NOT NULL, Primary Key
EMP_NAME	VARCHAR(30)	
JOB_ID	VARCHAR(20)	
SALARY	NUMBER	
MGR_ID	NUMBER	References EMPLOYEE_ID column
DEPARTMENT_ID	NUMBER	Foreign key to DEPARTMENT_ID column of the DEPARTMENTS table

DEPARTMENTS

DEPARTMENT_ID	NUMBER	NOT NULL, Primary Key
DEPARTMENT_NAME	VARCHAR(30)	
MGR_ID	NUMBER	References MGR_ID column of the EMPLOYEES table
LOCATION_ID	NUMBER	Foreign key to LOCATION_ID column of the LOCATIONS table LOCATIONS
LOCATION_ID	NUMBER	NOT NULL, Primary Key,
CITY	VARCHAR(30)	

Which two SQL statements produce the name, department name, and the city of all the employees who earn more than 10000? (Choose two)

A.
```
SELECT emp_name, department_name, city
FROM employees e
JOIN departments d
USING (department_id)
JOIN locations l
USING (location_id)
WHERE salary > 10000;
```

B.
```
SELECT emp_name, department_name, city
FROM employees e, departments d, locations l
JOIN ON (e.department_id = d.department id)
AND (d.location_id = l.location_id)
AND salary > 10000;
```

C.
```
SELECT emp_name, department_name, city
FROM employees e, departments d, locations l
WHERE salary > 10000;
```

D.
```
SELECT emp_name, department_name, city
FROM employees e, departments d, locations l
WHERE e.department_id = d.department_id
AND d.location_id = l.location_id
AND salary > 10000;
```

E
```
SELECT emp_name, department_name, city
FROM employees e
NATURAL JOIN    departments, locations
WHERE salary > 10000;
```

Solution. B, D

16.2 Integrity Constraints

One major responsibility of database management system is that it must maintain **data integrity**. In other words, data in the database must be consistent and correct. A DBMS ensures this through **integrity constraints** or simply **constraints**.

In MySQL, you can create and define integrity constraints through CREATE TABLE and ALTER TABLE DDL commands, which you are going to learn a little later in this chapter.

DEF

Integrity constraints (or **constraints**) are the rules that a database must comply at all times. Integrity constraints determine what all changes are permissible to a database.

How MySQL maintains data integrity ?

MySQL maintains data integrity through the constraints that are defined. After each change in the database, MySQL first tests whether the database with changes complies with all relevant integrity constraints. If it does, MySQL confirms the changes and database is said to be valid (*i.e.*, maintains data integrity). If the database with the changes violates any of the constraints, then MySQL rejects the changes and produces error-message. Database remains in the state prior to these changes *i.e.*, in valid state.

NOTE

*Valid database means **consistent** and **correct** data. Data is **consistent** if individual data items do not contradict one another ; data is **correct** if it satisfies all relevant constraints.*

16.3 Create Table with Constraints

In Class XI, you have learnt to create tables using CREATE TABLE command. You learnt a bit about constraints also, but this time you'll learn to use CREATE TABLE command with constraints in a *little-detailed*[1] manner. Let us quickly recall how to use CREATE TABLE command :

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> ... ] ) ...) ;
```

Consider the following CREATE TABLE example :

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

As you can see that first three columns have NOT NULL written in their definition. This means you are imposing **NOT NULL constraint** on these columns, which means these columns cannot store NULL values ever.

Also, a NOT NULL constraint is always defined as *column-level* only.

NOTE

NOT NULL constraint on a column ensures that the column does not store NULL value ever.

16.3.1 Defining Other Constraints

As you know that *constraints* let you restrict behaviours on columns. You can create these *constraints* when you define a table with a CREATE TABLE command. You have learnt to use NOT NULL constraint in previous lines.

1. See, an oxymoron :-)

In this section, we shall be talking about following *constraints* with CREATE TABLE command :

- *Primary Key* Constraint
- *Foreign Key* Constraint

You can also add, modify, or drop these constraints with an ALTER TABLE statement after you've created the table.

Recall that when a constraint is applicable to just one column, it is known as **column constraint** whereas a constraint that affects multiple rows or columns of a table is known as **table constraint**.

Two Styles of Defining Constraints

With CREATE TABLE command, you can define *constraints* in two different ways :

(*i*) in the line of column definition (**inline definition**) – column level constraint,

(*ii*) in a separate line after all the column definitions (**out-of-line definition**) – table level constraint.

A constraint defined on the same line as its column is called an **inline constraint**, while a constraint on its own line in a CREATE TABLE statement is an **out-of-line constraint**. *Out-of-line constraints* must reference (or point to) the column that they constrain.

16.3.1A Primary Key Constraint

A *primary key* refers *to* a column or group of columns of a table of which the values are always unique and it is designated to identify each row in the table uniquely. You must know that NULL values are not permitted in columns that form part of a primary key.

Primary keys can be defined in *two* ways :

(*i*) as column constraint (*ii*) as table constraint

Defining Primary Key as Column Constraint

This way of defining primary key is useful if the primary key consists of just one column *i.e.*, one single field has been designated as the primary key of the table. In this case, the term PRIMARY KEY is simply added to the column definition as shown below :

```
CREATE TABLE <table-name>
(    column definition PRIMARY KEY,
     other column definitions here );
```

A primary key does not allow NULL values, hence a column designed as a primary key must have NOT NULL constraint applied on it.

Consider the following example

Example 16.1. *Create a table namely MEMBERS, including the primary key, which is the field* **Memberno.**

Solution.

```
CREATE TABLE MEMBERS (
Memberno INTEGER NOT NULL PRIMARY KEY,
::
Club CHAR(4) ) ;
```

inline constraint as it is defined in the same line of column definition.

See, in the above example, the primary key is defined after the null specification. The null specification may also be specified behind the primary key.

Defining Primary Key as Table Constraint

This way of defining primary key is useful if the primary key consists of more than one column i.e., a composite-key (combination of more than one field) has been designated as the primary key of the table. However, you can also adopt this style for defining a single-column primary-key as well.

In this case, the term **PRIMARY KEY** is added to the table definition in a separate row as shown below:

```
CREATE TABLE <table-name>
(    column1 definition,
     other column definitions here,
     PRIMARY KEY(<primary key fields > )
);
```

Example 16.2. *Create a table namely MEMBERS, including the primary key, which is the field Memberno. Define the primary key as a table constraint i.e., as out-of-line constraint.*

Solution.
```
CREATE TABLE MEMBERS
(
    Memberno INTEGER NOT NULL,
    ::
    Club CHAR(4),
    PRIMARY KEY(Memberno)
);
```

NOTE

A composite primary key can be defined as only a table integrity constraint whereas a single-field primary key can be defined in both ways : as column constraint and as table constraint.

Example 16.3. *Create a StuStreams table to record stream chosen by the student along with other student-details; the StreamCode and Rollno columns together form a composite primary key.*

Solution.
```
CREATE TABLE STUSTREAMS
(
    STREAMCODE INTEGER NOT NULL,
    ROLLNO INTEGER NOT NULL,
    NAME VARCHAR(50) NOT NULL,
    ::
    PRIMARY KEY (STREAMCODE, ROLLNO)
);
```

NOTE

Please note that if a column that is part of a primary key has not been defined as NOT NULL, MySQL internally defines the column as NOT NULL.

16.3.1B 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 :

```
Players (PlayerNo, Name, dob, sex, address, city, phone, teamno)
Teams (Teamno, Details, Strength)
```

underlined columns indicate primary keys

Now, all team members stored in *Players* table must exist in the table *Teams*.

This type of relationship between two tables and their attributes (fields) is called a *referential integrity constraint*. Referential integrity constraints are a special type of integrity constraints that can be implemented as a *foreign key* with the CREATE TABLE statements.

IMPORTANT

Important for Foreign Key Implementation in MySQL

In MySQL, foreign keys can be used only for tables that are created with the storage engine *InnoDB*; the others storages engines donot support foreign keys. This is one of the reasons to prefer *InnoDB*. Because of this, we assume in this chapter that *InnoDB* is the default storage engine. You can determine for your tables what storage engine they have by writing the following statement on *MySQL* prompt.

```
SHOW CREATE TABLE <table-name> ;
```

It will show you the storage engine for your table in the last line of the output.

For example, if you write :

```
mysql> SHOW CREATE TABLE dept ;
```

```
+-------+-----------------------------------------------------+
| Table |                  Create Table                       |
+-------+-----------------------------------------------------+
| dept  | CREATE TABLE `dept` (                               |
|       |   `deptno` decimal(2,0) NOT NULL DEFAULT '0',       |
|       |   `dname`  varchar(14) DEFAULT NULL,                |
|       |   `loc`    varchar(13) DEFAULT NULL,                |
|       |   PRIMARY KEY (`deptno`)                            |
|       | ) ENGINE = InnoDB DEFAULT CHARSET = latin1          |
+-------+-----------------------------------------------------+
1 row in set (0.09 sec)
```

The storage engine for table dept.

If you find that your table has a storage engine which is not *InnoDB* (default is *MyISAM*), you can change the storage engine for your table by writing the following statement :

```
ALTER TABLE <table-name> ENGINE = InnoDB ;
```

For instance, following command will change the storage engine of table *SalaryGrade* to *InnoDB*.

```
ALTER TABLE SalaryGrade ENGINE = InnoDB ;
```

Or at the time of CREATE TABLE, you can specify storage engine as follows :

```
CREATE TABLE <table name>
(
    :: Column definitions here
) ENGINE = InnoDB ;
```

e.g.,

```
CREATE TABLE Stu (id INT NOT NULL,
Name VARCHAR(30) NOT NULL,
    PRIMARY KEY (id)
) ENGINE = INNODB ;
```

Implementing Foreign Key Constraint

Before you implement a foreign key constraint, carefully go through the definition of foreign key that states that a *Foreign key* is a non-key column of a table (*child table*) that draws its values from *primary key* (or unique key) of another table (called *parent table*).

A foreign key constraint is hence implemented in a child table by adding the following details out-of-line in the end of all column-definitions :

```
CREATE TABLE <child_table-name>
(    column definitions here
     ::
[CONSTRAINT [symbol] FOREIGN KEY
     REFERENCES Parent_tbl_name (Primary_key,...)
     [ON DELETE RESTRICT | CASCADE | SET NULL | NO ACTION]
     [ON UPDATE RESTRICT | CASCADE | SET NULL | NO ACTION]
     ::
) ;
```

For instance, consider the following example (assuming the default storage engine is *InnoDB*) :

```
CREATE TABLE parentTable  (  id INT NOT NULL PRIMARY KEY
                          );
CREATE TABLE childTable
(    id INT,
     parent_id INT,
     FOREIGN KEY (parent_id) REFERENCES parentTable(id)
);
```

You can also define a foreign key through *inline definition*, but *out-of-line definition* is conventionally preferred.

DEF

The table in which a foreign key is defined is called a **referencing table** or **child table**. A table to which a foreign key points is called a **referenced table or parent table**.

Example 16.4. Create the PLAYERS table so that team-numbers must appear TEAMS in the table. We assume that the TEAMS table has already been created with the TEAMNO column as the primary key.

Solution.

```
CREATE TABLE PLAYERS
(    PLAYERNO INTEGER NOT NULL,
     NAME CHAR(15) NOT NULL,
     DOB DATE,
     SEX CHAR(1) NOT NULL,
     ADDRESS VARCHAR(100) NOT NULL,
     CITY VARCHAR(30) NOT NULL,
     PHONE CHAR(15),
     TEAMNO CHAR(4) UNIQUE,
     PRIMARY KEY (PLAYERNO),
     FOREIGN KEY (TEAMNO) REFERENCES TEAMS (TEAMNO) ON DELETE SET NULL
) ;
```

Understanding Foreign Key Specification

In the above example, you added the following foreign key specification to PLAYERS table's CREATE TABLE statement.

```
FOREIGN KEY (TEAMNO) REFERENCES TEAMS (TEAMNO) ON DELETE SET NULL
```

Now carefully notice that each foreign key specification consists of *three* parts :

(*i*) Specifying *column* as the foreign key
example : FOREIGN KEY (TEAMNO) in above specification

(*ii*) Specifying parent-table and its reference-key *i.e.*, to which the foreign key refers
example : REFERENCES TEAMS (TEAMNO) in above specification

(*iii*) Specifying *referencing action* that decides what action to take place in case of DELETE or UPDATE operations
example : ON DELETE SET NULL in above specification

Let us understand foreign key referencing action in details.

The *referencing action* can be one the following :

```
RESTRICT | CASCADE | SET NULL | NO ACTION
```

As you know that you can specify the *Referencing Action* through following clauses :

```
[ON DELETE referencing_action ]
[ON UPDATE referencing_action]
```

NOTE

A foreign key cannot refer to a random group of columns ; it must be a combination of columns for which the combined value is guaranteed unique i.e., only the primary key or alternate keys (those defined with UNIQUE constraint) can be referred to.

Referencing action with ON DELETE clause determines what to do in case of a DELETE occurs in the parent table.

Referencing action with ON UPDATE clause determines what to do in case of a UPDATE occurs in the parent table.

- **CASCADE.** This action states that if a DELETE or UPDATE operation affects a row from the parent table, then automatically **delete** or **update** the matching rows in the child table *i.e.*, cascade the action to child table.
- **SET NULL.** This action states that if a DELETE or UPDATE operation affects a row from the parent table, then set the foreign key column or columns in the child table to NULL.

  If you specify a SET NULL action, *make sure that you have not declared the columns in the child table as NOT NULL.*
- **NO ACTION.** In standard SQL, *NO ACTION means no action* in the sense that an attempt to delete or update a primary key value is not allowed to proceed if there is a related foreign key value in the referenced table. In this case, *InnoDB* rejects the DELETE or UPDATE operation for the parent table.
- **RESTRICT.** This action rejects the DELETE or UPDATE operation for the parent table. Specifying RESTRICT (or NO ACTION) is the same as omitting the ON DELETE or ON UPDATE clause.

The following rules apply when a foreign key is specified :

- The *referenced table* must already have been created by a CREATE TABLE statement or must be the table that is currently being created. In the latter case, the *referencing table* is the same as the *referenced table.*
- A primary key *must* be defined for the *referenced table.*
- A null value is permitted in a foreign key, although a primary key can never contain null values. This means that the contents of a foreign key are correct if each non-null value occurs in a specific primary key.
- The number of columns in the foreign key must be the same as the number of columns in the primary key (or unique key) of the *referenced table.*
- The data types of the columns in the foreign key must match those of the corresponding columns in the primary key of the *referenced table.*

Naming the Constraint

While defining the foreign key, you can use keyword CONSTRAINT followed by constraint_name, if you want to provide a name to your foreign key constraint, *e.g.*,

```
CONSTRAINT fkey1 FOREIGN KEY (deptno) REFERENCES dept (deptno)
```

Above foreign key specification names the foreign key constraint as *fkey1* along with defining the constraint.

Example 16.5. *Create two tables*

```
Customer(customer_id, name )
Customer_sales (transaction_id, amount, customer_id)
```

Underlined columns indicate primary keys and coloured column indicates foreign key.

Make sure that no action should take place in case of a DELETE or UPDATE in the parent table. Name the foreign key constraint as fk_cust.

Solution.

```
CREATE TABLE customer
(    customer_id    INT NOT NULL,
     name           VARCHAR(30),
     PRIMARY KEY (customer_id)
) TYPE = INNODB ;

CREATE TABLE customer_sales
(    transaction_id     INT NOT NULL,
     amount             INT,
     customer_id            INT NOT NULL,
     PRIMARY KEY (transaction_id),
     CONSTRAINT fk_cust FOREIGN KEY (customer_id) REFERENCES customer (customer_id)
     ON DELETE NO ACTION
     ON UPDATE NO ACTION
) TYPE = INNODB ;
```

Self-referencing Tables

The referenced and referencing table associated with a foreign key may be the same. Such a table is called a *self-referencing table*, and this is known as *self-referential integrity*. Consider following example :

```
CREATE TABLE empl (
    empno DECIMAL(4,0) NOT NULL,
    ename VARCHAR(10),
    job VARCHAR(9),
    mgr DECIMAL(4,0),
    hiredate DATE,
    sal DECIMAL(7,2),
    comm DECIMAL(7,2),
    deptno DECIMAL(2,0),
    PRIMARY KEY (empno),
    CONSTRAINT fkey1 FOREIGN KEY (deptno) REFERENCES dept (deptno)
    CONSTRAINT fkey2 FOREIGN KEY (mgr) REFERENCES empl (empno)
) ;
```

In the above example, last line of table-creation defines a foreign key constraint (namely **fkey2**) for column *mgr* that refers to *empno* column of its own table *i.e.*, **empl** is *self-referencing* table here.

16.4 Alter Table with Constraints

Sometimes we need to make changes in the definitions of existing table. For such a thing, you can use ALTER TABLE statement of MySQL. You can use ALTER TABLE statement for many things, such as :

- adding columns to a table
- deleting columns of a table
- enabling/disabling constraints
- modifying column-definitions of a table
- adding constraints to table

and many more.

Let us learn to use ALTER TABLE command for these purposes.

16.4.1 Adding Columns to a Table

To add a column to a table, you can use ALTER TABLE command as per following syntax :

```
ALTER TABLE <tablename>
ADD [COLUMN] <column name> <datatype> [NOT NULL]
        [<integrity constraint definition>] ;
```

For example,

To add a new column called TYPE to the TEAMS table (to indicate whether it is a women's or a men's team), you can write :

```
ALTER TABLE TEAMS
ADD TYPE CHAR(1) ;
```

A new column by the name TYPE will be added to the table, where each row will contain NULL value for the new column.

However if you specify NOT NULL constraint while adding a new column , MySQL adds the new column with the default value of that datatype *e.g.*, for INT type it will add 0, for CHAR types , it will add a space, and so on.

Example 16.6. *Given a table namely Testt with following data in it.*

Col1	Col2
1	A
2	G

Now following commands are given for the table. Predict the table contents after each of the following statements :

(*i*) `ALTER TABLE testt ADD col3 INT ;`

(*ii*) `ALTER TABLE testt ADD col4 INT NOT NULL ;`

(*iii*) `ALTER TABLE testt ADD col5 CHAR(3) NOT NULL ;`

(*iv*) `ALTER TABLE testt ADD col6 VARCHAR(3) ;`

Solution. (*i*)

```
+-------+-------+-------+
| col1  | col2  | col3  |
+-------+-------+-------+
|  1    |  A    | NULL  |
|  2    |  G    | NULL  |
+-------+-------+-------+
```

(*ii*)

col1	col2	col3	col4
1	A	NULL	0
2	G	NULL	0

(*iii*)

col1	col2	col3	col4	col5
1	A	NULL	0	
2	G	NULL	0	

There are spaces in col5 for each row.

(*iv*)

col1	col2	col3	col4	col5	col6
1	A	NULL	0		NULL
2	G	NULL	0		NULL

Example 16.7. *Write statement to add two new columns (category and league) to the TEAMS table. Category must not allow null values.*

Solution.

```
ALTER TABLE TEAMS
ADD (CATEGORY VARCHAR(20) NOT NULL,
LEAGUE INTEGER) ;
```

NOTE

If in ALTER TABLE an added column has NOT NULL constraint then for the new column, MySQL fills in an actual value: the value 0 for numeric columns, the empty string for alphanumeric columns, the date 0000-00-00 for date data types, and the time 00:00:00 for time data types.

16.4.2 Modifying Columns

You can also use ALTER TABLE statement to change many properties of columns. To do so, you can use ALTER TABLE in the following manner :

```
ALTER TABLE <table name>
CHANGE [ COLUMN ] <old column name> <new column name> <column definition>
```

Or

```
ALTER TABLE <table name>
MODIFY [ COLUMN ] <column name> <column definition>
```

Column definition is the traditional way of defining column *i.e.,*

```
<column name> <data type> [ NOT NULL ] [ < integrity constraint> ]
```

Both MODIFY and CHANGE allow you to make changes in column definition, but there is a slight difference between the *two* :

- With CHANGE clause, you always need to specify the name of the column twice like :

  CHANGE <column-name> <column-definition with column name once again>

  It will become clear to you when you go through the following examples :
- with MODIFY you cannot change the name of a column whereas with CHANGE, you can do so.

In this section, we shall not be covering all the changes that can be done through CHANGE or MODIFY clauses of ALTER TABLE, but we shall be talking about only the basic changes as guided by the syllabus.

Now consider the following examples :

To change the data type of the PLAYERNO column in the PLAYERS table from INTEGER to SMALLINT, you can write :

```
ALTER TABLE PLAYERS
CHANGE PLAYERNO PLAYERNO SMALLINT ;
```

Column being changed — *New column definition* — *see with CHANGE clause, column name comes twice*

If you want to write the same command with MODIFY clause, you will write :

```
ALTER TABLE PLAYERS
MODIFY PLAYERNO SMALLINT ;
```

Column name does not come twice — *New column definition*

Example 16.8. *In table PLAYERS, increase the length of the CITY column from 30 to 40.*

Solution.
```
ALTER TABLE PLAYERS
CHANGE CITY CITY VARCHAR(40) NOT NULL ;
```

Example 16.9. *Rewrite example 16.8 with MODIFY clause.*

Solution.
```
ALTER TABLE PLAYERS
MODIFY CITY VARCHAR(40) NOT NULL ;
```

16.4.3 Deleting Columns

To delete a column from the table, the ALTER TABLE command takes the following form :

```
ALTER TABLE <table name>
DROP [ COLUMN ] <old column name> ;
```

For example, to delete column TYPE from the TEAMS table, you will write:

```
ALTER TABLE TEAMS
DROP TYPE ;
```

Or
```
ALTER TABLE TEAMS
DROP COLUMN TYPE ;
```

16.4.4 Adding/Removing Constraints to Table

You can also use ALTER TABLE statement to add constraints to your existing table by using it in following manner :

```
ALTER TABLE <table-name>
ADD <constraint -definition> ;
```

The syntax for adding integrity constraints with an ALTER TABLE statement is identical to the syntax for table integrity constraints in the CREATE TABLE statement. That is, to add a PRIMARY KEY constraint, you need to write ALTER TABLE as per following syntax :

Syntax

```
ALTER TABLE <table-name>
ADD [CONSTRAINT [ <constraint name> ] ]
PRIMARY KEY ( <column name> )]
```

Similarly, to add a FOREIGN KEY constraint, you need to write ALTER TABLE as per following syntax :

Syntax

```
ALTER TABLE <table-name>
ADD [CONSTRAINT [ <constraint name> ] ]
FOREIGN KEY ( <column list>) REFERENCES <parent-table> (<column-name>)
[ON DELETE RESTRICT | CASCADE | SET NULL | NO ACTION ]
[ON UPDATE RESTRICT | CASCADE | SET NULL | NO ACTION ]
```

For example, to add a primary key constraint to a table *Tbl1 (id, col1, col2, col3)*, you will be writing something like :

```
ALTER TABLE tbl1
ADD PRIMARY KEY (id);
```

Naming a Constraint

You can name a constraint through CONSTRAINT <constraint-name> clause, *e.g.,* to give a name (say *pk1*) to the primary key constraint in the previous statement, write it as :

```
ALTER TABLE tbl1
ADD CONSTRAINT pk1 PRIMARY KEY (id) ;
```

Example 16.10. *Create the two tables Tt1 and Tt2, such that column (say A) of table Tt1 is the foreign key referring to column A of table Tt2 (constraint name fk1). Tables are not created as yet.*

Solution.

```
CREATE TABLE Tt1
(
    A INTEGER NOT NULL PRIMARY KEY,
    B INTEGER NOT NULL
) ;

CREATE TABLE Tt2
(
    A INTEGER NOT NULL PRIMARY KEY,
    B INTEGER NOT NULL,
) ;

ALTER TABLE Tt1
ADD CONSTRAINT fk1 FOREIGN KEY (A) REFERENCES Tt2 (A) ON DELETE CASCADE ;
```

Removing Constraints

To remove **primary key constraint** from a table, you need to write :

```
ALTER TABLE <table-name>
DROP PRIMARY KEY ;
```

To remove **foreign key constraint** from a table, you need to write :

```
ALTER TABLE <table-name>
DROP FOREIGN KEY <constraint-name> ;
```

Example 16.11. *Delete the primary key from the PLAYERS table.*

Solution.

```
ALTER TABLE PLAYERS DROP PRIMARY KEY ;
```

Example 16.12. *Delete the foreign key constraint namely fk1 from the Tt1 table.*

Solution.

```
ALTER TABLE Tt1 DROP FOREIGN KEY fk1 ;
```

16.5 Viewing Constraints and their Columns

To view all the information about how the table was created including its constraints, you need to write following statement :

```
SHOW CREATE TABLE <table name> ;
```

It will give you detailed information about the table.

For example, look at the following query and the output produced by it :

```
mysql> SHOW CREATE TABLE empl;
+------------+------------------------------------------------------------------+
| Table      |    Create Table                                                  |
+------------+------------------------------------------------------------------+
| empl       |    CREATE TABLE 'empl' (                                         |
|            |       'empno' decimal(4,0) NOT NULL,                             |
|            |       'ename' varchar(10) DEFAULT NULL,                          |
|            |       'job' varchar(9) DEFAULT NULL,                             |
|            |       'mgr' decimal(4,0) DEFAULT NULL,                           |
|            |       'hiredate' date DEFAULT NULL,                              |
|            |       'sal' decimal(7,2) DEFAULT NULL,                           |
|            |       'comm' decimal(7,2) DEFAULT NULL,                          |
|            |       'deptno' decimal(2,0) DEFAULT NULL,                        |
|            |       PRIMARY KEY ('empno'),                                     |
|            |       KEY 'fkey1' ('deptno'),                                    |
|            |       CONSTRAINT 'fkey1' FOREIGN KEY ('deptno') REFERENCES 'dept' ('deptno') |
|            |    ) ENGINE=InnoDB DEFAULT CHARSET = latin1                      |
+------------+------------------------------------------------------------------+
```

See, we have coloured the rows showing the constraint details in the above output. Notice that in table *empl*, primary key constraint has been defined without any *constraint-name* whereas the foreign key constraint has been defined with a name ***fkey1***.

16.6 Enabling/Disabling Constraints

In MySQL, you cannot disable a PRIMARY KEY constraint but you can drop it if you want through ALTER TABLE command. However, you can disable/enable FOREIGN KEY constraints. To do so, you need to use system variable FOREIGN_KEY_CHECKS as follows :

To disable foreign keys

```
SET FOREIGN_KEY_CHECKS = 0 ;
```

To enable foreign keys

```
SET FOREIGN_KEY_CHECKS = 1 ;
```

After disabling foreign keys, you can successfully add into a table a row that does not meet foreign key criteria. However, we do not recommend that foreign key constraints should be disabled.

16.7 Dropping Tables & databases

delete → deleted data from table.

To delete or drop tables from database, you can use DROP TABLE command. With DROP TABLE statement, MySQL also removes the descriptions of the table from all relevant catalog tables, along with all integrity constraints, indexes, and privileges that are "linked" to that table. In fact, MySQL removes each database object that has no right to exist after the table has been deleted.

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.