**Unit II**

**SQL Tutorial**

**SQL stands for Structured Query Language (SQL).** It is the most commonly used language that enables us to create, delete and operate relational databases. Relational DBMS is a type of DBMS that stores data in the form of tables.

**Features of SQL**:

* It is a non procedural language.
* It is an English-like language.
* It can process a single record as well as sets of records at a time.
* It is different from a third generation language (C& COBOL). All SQL statements define what is to be done rather than how it is to be done.
* SQL is a data sub-language consisting of three built-in languages: Data definition language (DDL), Data manipulation language (DML), Data query language (DQL) and Data control language (DCL).
* SQL has facilities for defining database views, security, integrity constraints, transaction controls, etc

**Advantages of SQL:**

* A standard query language that provides an interface to RDBMS.
* Allows users to access data in relational database management systems.
* Allows users to describe the data.
* Allows users to define the data in database and manipulate that data.
* Allows embedding within other languages using SQL modules, libraries & pre-compilers.
* Allows users to create and drop databases and tables.
* Allows users to create view, stored procedure, functions in a database.
* Allows users to set permissions on tables, procedures and views

**History:**

* **1970 --** Dr. E. F. "Ted" of IBM is known as the father of relational databases. He described a relational model for databases.
* **1974 --** Structured Query Language appeared.
* **1978 --** IBM worked to develop Codd's ideas and released a product named System/R.
* **1986 --** IBM developed the first prototype of relational database and standardized by ANSI. The first relational database was released by Relational Software and its later becoming Oracle.

**SQL RDBMS Concepts**

**What is RDBMS?**

RDBMS stands for **R**elational **D**atabase **M**anagement **S**ystem. RDBMS is the basis for SQL and for all modern database systems like MS SQL Server, IBM DB2, Oracle, MySQL, and Microsoft Access. A Relational database management system (RDBMS) is a database management system (DBMS) that is based on the relational model as introduced by E. F. Codd.

**What is table?**

The data in RDBMS is stored in database objects called **tables**. The table is a collection of related data entries and it consists of columns and rows. Remember, a table is the most common and simplest form of data storage in a relational database. Following is the example of a

**CUSTOMERS table**:

+----+----------+-----+-----------+----------+

| **ID | NAME | AGE | ADDRESS | SALARY |**

+----+----------+-----+-----------+----------+

| 1 | Ramesh | 32 | Ahmedabad | 2000.00 |

| 2 | Khilan | 25 | Delhi | 1500.00 |

| 3 | kaushik | 23 | Kota | 2000.00 |

| 4 | Chaitali | 25 | Mumbai | 6500.00 |

| 5 | Hardik | 27 | Bhopal | 8500.00 |

| 6 | Komal | 22 | MP | 4500.00 |

| 7 | Muffy | 24 | Indore | 10000.00 |

+----+----------+-----+-----------+----------+

**What is field?**

Every table is broken up into smaller entities called fields. The fields in the CUSTOMERS table consist of ID, NAME, AGE, ADDRESS and SALARY.

A field is a column in a table that is designed to maintain specific information about every record in the table.

**What is record or row?**

A record, also called a row of data, is each individual entry that exists in a table. For example, there are 7 records in the above CUSTOMERS table. Following is a single row of data or record in the CUSTOMERS table:

+----+----------+-----+-----------+----------+

| 1 | Ramesh | 32 | Ahmedabad | 2000.00 |

+----+----------+-----+-----------+----------+

A record is a horizontal entity in a table.

**What is column?**

A column is a vertical entity in a table that contains all information associated with a specific field in a table. For example, a column in the CUSTOMERS table is ADDRESS, which represents location description and would consist of the following:

+-----------+

| ADDRESS |

+-----------+

| Ahmedabad |

| Delhi |

| Kota |

| Mumbai |

| Bhopal |

| MP |

| Indore |

+----+------+

**What is NULL value?**

A NULL value in a table is a value in a field that appears to be blank, which means a field with a NULL value is a field with no value. It is very important to understand that a NULL value is different than a zero value or a field that contains spaces. A field with a NULL value is one that has been left blank during record creation.

**Data Integrity**

The following categories of the data integrity exist with each RDBMS:

* **Entity Integrity:** There are no duplicate rows in a table.
* **Domain Integrity:** Enforces valid entries for a given column by restricting the type, the format, or the range of values.
* **Referential Integrity:** Rows cannot be deleted which are used by other records.
* **User-Defined Integrity:** Enforces some specific business rules that do not fall into entity, domain, or referential integrity.

**SQL RDBMS Databases**

There are many popular RDBMS available to work with. This tutorial gives a brief overview of few most popular RDBMS. This would help us to compare their basic features.

**MySQL**

MySQL is an open source SQL database, which is developed by Swedish company MySQL AB. MySQL is pronounced "my ess-que-ell," in contrast with SQL, pronounced "sequel."

MySQL is supporting many different platforms including Microsoft Windows, the major Linux distributions, UNIX, and Mac OS X. MySQL has free and paid versions, depending on its usage (non-commercial/commercial) and features. MySQL comes with a very fast, multi-threaded, multi-user, and robust SQL database server.

**Features:**

High Performance, High Availability, Scalability and Flexibility Run anything, Robust Transactional Support, Web and Data Warehouse Strengths, Strong Data Protection, Comprehensive Application Development, Management Ease, Open Source Freedom and 24 x 7 Support, Lowest Total Cost of Ownership.

**MS SQL Server**

MS SQL Server is a Relational Database Management System developed by Microsoft Inc. Its primary query languages are:

* T-SQL.
* ANSI SQL.

**Features:**

High Performance, High Availability, Database mirroring, Database snapshots, Service Broker, DDL triggers, Ranking functions, Row version-based isolation levels, XML integration, TRY...CATCH, Database Mail.

***ORACLE***

It is a very large and multi-user database management system. Oracle is a relational database management system developed by 'Oracle Corporation'. Oracle works to efficiently manage its resource, a database of information, among the multiple clients requesting and sending data in the network.

It is an excellent database server choice for client/server computing. Oracle supports all major operating systems for both clients and servers, including MSDOS, NetWare, UnixWare, OS/2 and most UNIX flavors.

**History:** Oracle began in 1977 and celebrating its 32 wonderful years in the industry (from 1977 to 2009).

**Features:**

Concurrency, Read Consistency, Locking Mechanisms ,Portability ,Self-managing database

SQL\*Plus ,Scheduler ,Resource Manager ,Data Warehousing ,Materialized views ,Bitmap indexes ,Table compression ,Parallel Execution ,Analytic SQL ,Data mining, Partitioning,

**MS-ACCESS:** This is one of the most popular Microsoft products. Microsoft Access is an entry-level database management software. MS Access database is not only an expensive but also powerful database for small-scale projects. MS Access uses the Jet database engine, which utilizes a specific SQL language dialect (sometimes referred to as Jet SQL). MS Access comes with the professional edition of MS Office package. MS Access has easy-to-use intuitive graphical interface.

**Features:**

* Users can create tables, queries, forms and reports and connect them together with macros.
* The import and export of data to many formats including Excel, Outlook, ASCII, dBase, Paradox, FoxPro, SQL Server, Oracle, ODBC, etc.
* There is also the Jet Database format (MDB or ACCDB in Access 2007), which can contain the application and data in one file. This makes it very convenient to distribute the entire application to another user, who can run it in disconnected environments.
* Microsoft Access offers parameterized queries. These queries and Access tables can be referenced from other programs like VB6 and .NET through DAO or ADO.
* The desktop editions of Microsoft SQL Server can be used with Access as an alternative to the Jet Database Engine.
* Microsoft Access is a file server-based database. Unlike client-server relational database management systems (RDBMS), Microsoft Access does not implement database triggers, stored procedures, or transaction logging.

In order to communicate with the database, SQL supports the following categories of languages:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 1. **DDL-Data Definition Language:**   **Command** | | **Description** | | |
| CREATE | | Creates a new table, a view of a table, or other object in database | | |
| ALTER | | Modifies an existing database object, such as a table. | | |
| DROP | | Deletes an entire table, a view of a table or other object in the  database. | | |
| 1. **DML-Data Manipulation Language:**   **Command** | | **Description** | | |
| INSERT | Creates a record | | |
| UPDATE | Modifies records | | |
| DELETE | Deletes records | | |
|  |  | | |
| 1. **DCL -Data Control Language:**   **Command** | | **Description** |
| GRANT | | Gives a privilege to user |
| REVOKE | | Takes back privileges granted from user |
| 1. **DQL -Data Query Language:**   **Command** | | **Description** |
| SELECT | | Retrieves certain records from one or more tables |

**SQL Data Types**

SQL data type is an attribute that specifies type of data of any object. Each column, variable and expression has related data type in SQL. We would use these data types while creating our tables. We would choose a particular data type for a table column based on our requirement. Some data types used in SQL Server are:

1. **Character data type**
2. char (size)- used to store fixed length alphanumerical value, where size is the total no.

of bytes.

b) varchar(size) & varchar2(size) - used to store variable length alphanumerical value,

where size is the total no. of bytes.

1. **Numeric data type**
2. int or interger or smallint- used to store integer numbers of various size.
3. numeric(p,s) or decimal(p,s)- used to store formatted numbers, where **p** is the

***precision***, the total number of decimal digits and **s** is the ***scale***, the no. of digits after

the decimal point.

e.g numeric(5,2) contains

Valid no. 125.50, 35.99 etc

1. **Date & Time data type- used to store date in YYYY-MM-DD format.**

a) date

b) time

c) datetime

d) timestamp

e) smalltime

valid date ‘1995-10-15’ , written within single quotes(‘ ‘)

invalid date ‘1999-13-02’

**SQL Operators**

**What is an Operator in SQL?**

An operator is a reserved word or a character used primarily in an SQL statement's WHERE clause to perform operation(s), such as comparisons and arithmetic operations. Operators are used to specify conditions in an SQL statement and to serve as conjunctions for multiple conditions in a statement.

* Arithmetic operators
* Comparison operators
* Logical operators
* Operators used to negate conditions

**SQL Arithmetic Operators:** Assume variable a holds 10 and variable b holds 20, then:

|  |  |  |
| --- | --- | --- |
| **Operator** | **Description** | **Example** |
| + | Addition - Adds values on either side of the operator | a + b will give 30 |
| - | Subtraction - Subtracts right hand operand from left hand operand | a - b will give -10 |
| \* | Multiplication - Multiplies values on either side of the operator | a \* b will give 200 |
| / | Division - Divides left hand operand by right hand operand | b / a will give 2 |
| % | Modulus - Divides left hand operand by right hand operand and returns remainder | b % a will give 0 |

***SQL Logical Operators:***

|  |  |
| --- | --- |
| Here is a list of all the logical operators available in SQL. ALL operators is used to compare a value to all values in another value set. | |
| **Operator** | **Description** |
| AND | The AND operator allows the existence of multiple conditions in an SQL statement's WHERE clause. |
| ANY | The ANY operator is used to compare a value to any applicable value in the list according to the condition. |
| BETWEEN | The BETWEEN operator is used to search for values that are within a set of values, given the minimum value and the maximum value. |
| EXISTS | The EXISTS operator is used to search for the presence of a row in a specified table that meets certain criteria. |
| IN | The IN operator is used to compare a value to a list of literal values that have been specified. |
| LIKE | The LIKE operator is used to compare a value to similar values using wildcard operators. |
| NOT | The NOT operator reverses the meaning of the logical operator with which it is used. Eg: NOT EXISTS, NOT BETWEEN, NOT IN, etc. **This is a negate operator.** |
| OR | The OR operator is used to combine multiple conditions in an SQL statement's WHERE clause. |
| IS NULL | The NULL operator is used to compare a value with a NULL value. |
| UNIQUE | The UNIQUE operator searches every row of a specified table for uniqueness (no duplicates). |

**1. DDL-Data Definition Language**

The Data definition language (DDL) defines a set of commands used in the creation and

modification of schema objects such as tables, indexes, views etc but not the data. These

commands provide the ability to create, alter and drop these objects.

**i) CREATE TABLE Command**

Creating a basic table involves naming the table and defining its columns and each column's data type. The SQL **CREATE TABLE** statement is used to create a new table.

Syntax: Basic syntax of CREATE TABLE statement is as follows:

**CREATE TABLE table\_name (**

**column1 datatype [constraints],**

**column2 datatype [constraints],**

**.....**

**columnN datatype [constraints],**

**additional\_table\_constraints**

**);**

CREATE TABLE is the keyword telling the database system what we want to do. In this case, we want to create a new table. The unique name or identifier for the table follows the CREATE TABLE statement.

Then in brackets comes the list defining each column in the table, what sort of data type it is and if any constraint on this column.

**Guidelines for creation of table:**

* Table name should start with an alphabet.
* In table name, blank spaces and single quotes are not allowed.
* Reserve words of that DBMS cannot be used as table name.
* Proper data types and size should be specified.
* Unique column name should be specified

**Column Constraints:**

PRIMARY KEY, UNIQUE, NOT NULL, CHECK, DEFAULT, REFERENCES.

The syntax becomes clearer with an example below. Example:

Following is an example, which creates a CUSTOMERS table with ID as primary key and NOT NULL are the constraints showing that these fields cannot be NULL while creating records in this table:

**SQL> CREATE TABLE CUSTOMERS ( ID INT PRIMARY KEY, NAME VARCHAR (20) NOT NULL, AGE INT NOT NULL, ADDRESS CHAR (25) , SALARY DECIMAL (18, 2));**

We can verify if our table has been created successfully by looking at the message displayed by the SQL server, otherwise we can use **DESC / sp\_help** for sql server 2005command as follows:

**SQL> DESC CUSTOMERS;**

**Or**

**SQL> sp\_help CUSTOMERS;**

+---------+---------------+------+-----+---------+-------+

| Field | Type | Null | Key | Default | Extra |

+---------+---------------+------+-----+---------+-------+

| ID | int(11) | NO | PRI | | |

| NAME | varchar(20) | NO | | | |

| AGE | int(11) | NO | | | |

| ADDRESS | char(25) | YES | | | |

| SALARY | decimal(18,2) | YES | | | |

***Create Table Using another Table***

A copy of an existing table can be created using a combination of the CREATE TABLE statement and the SELECT statement. The new table has the same column definitions. All columns or specific columns can be selected. When we create a new table using existing table, new table would be populated using existing values in the old table.

**Syntax:** The basic syntax for creating a table from another table is as follows:

**CREATE TABLE NEW\_TABLE\_NAME AS**

**SELECT [ column1, column2...columnN ]**

**FROM EXISTING\_TABLE\_NAME [ WHERE ]**

Here, column1, column2...are the fields of existing table and same would be used to create fields of new table. Following is an example, which would create a table SALARY using CUSTOMERS table and having fields customer ID and customer SALARY:

**SQL> CREATE TABLE SALARY AS SELECT ID, SALARY FROM CUSTOMERS;**

This would create new table SALARY, which would have the following records:

+----+----------+

| ID | SALARY |

+----+----------+

| 1 | 2000.00 |

| 2 | 1500.00 |

| 3 | 2000.00 |

| 4 | 6500.00 |

| 5 | 8500.00 |

| 6 | 4500.00 |

| 7 | 10000.00 |

+----+----------+

ii) **ALTER TABLE Command:**

The SQL **ALTER TABLE** command is used for modification of existing structure of the table in the following situation:

* When a new column is to be added to the table structure.
* When the existing column definition has to be changed, i.e., changing the width of the data type or the data type itself.
* When integrity constraints have to be included or dropped.
* When a constraint has to be enabled or disabled.

**Syntax:**

The basic syntax of **ALTER TABLE** to add a new column in an existing table is as follows:

**ALTER TABLE table\_name ADD column name data\_type;**

The basic syntax of ALTER TABLE to **DROP COLUMN** in an existing table is as follows:

**ALTER TABLE table\_name DROP COLUMN column\_name;**

The basic syntax of ALTER TABLE to change the **DATA TYPE** of a column in a table is as follows:

**ALTER TABLE table\_name MODIFY COLUMN column\_name data\_type;**

The basic syntax of ALTER TABLE to add a constraint to a column in a table is as follows:

**ALTER TABLE table name ADD CONSTRAINT constraint\_name constraint\_type (column1, column2,..);**

The basic syntax of ALTER TABLE to add a **NOT NULL** constraint to a column in a table is as follows:

**ALTER TABLE table\_name MODIFY column\_name datatype NOT NULL;**

The basic syntax of ALTER TABLE to **ADD UNIQUE CONSTRAINT** to a table is as follows:

**ALTER TABLE table\_name**

**ADD CONSTRAINT MyUniqueConstraint UNIQUE(column1, column2...);**

The basic syntax of ALTER TABLE to **ADD CHECK CONSTRAINT** to a table is as follows:

**ALTER TABLE table\_name**

**ADD CONSTRAINT MyUniqueConstraint CHECK (CONDITION);**

The basic syntax of ALTER TABLE to **ADD PRIMARY KEY** constraint to a table is as follows:

**ALTER TABLE table\_name**

**ADD CONSTRAINT MyPrimaryKey PRIMARY KEY (column1, column2...);**

The basic syntax of ALTER TABLE to **DROP CONSTRAINT** from a table is as follows:

**ALTER TABLE table\_name DROP CONSTRAINT Constraint\_name;**

If we're using **MySQL,** the code is as follows:

**ALTER TABLE table\_name**

**DROP INDEX MyUniqueConstraint;**

The basic syntax of ALTER TABLE to **DROP PRIMARY KEY** constraint from a table is as follows:

**ALTER TABLE table\_name**

**DROP CONSTRAINT MyPrimaryKey;**

If we're using **MySQL,** the code is as follows:

**ALTER TABLE table\_name DROP PRIMARY KEY;**

Example: Consider the CUSTOMERS table having the following records:

+----+----------+-----+-----------+----------+

| ID | NAME | AGE | ADDRESS | SALARY |

+----+----------+-----+-----------+----------+

| 1 | Ramesh | 32 | Ahmedabad | 2000.00 |

| 2 | Khilan | 25 | Delhi | 1500.00 |

| 3 | kaushik | 23 | Kota | 2000.00 |

| 4 | Chaitali | 25 | Mumbai | 6500.00 |

| 5 | Hardik | 27 | Bhopal | 8500.00 |

| 6 | Komal | 22 | MP | 4500.00 |

| 7 | Muffy | 24 | Indore | 10000.00 |

+----+----------+-----+-----------+----------+

Following is the example to ADD a new column in an existing table:

ALTER TABLE CUSTOMERS ADD SEX char(1);

Now, CUSTOMERS table is changed and following would be output from SELECT statement:

+----+---------+-----+-----------+----------+------+

| ID | NAME | AGE | ADDRESS | SALARY | SEX |

+----+---------+-----+-----------+----------+------+

| 1 | Ramesh | 32 | Ahmedabad | 2000.00 | NULL |

| 2 | Ramesh | 25 | Delhi | 1500.00 | NULL |

| 3 | kaushik | 23 | Kota | 2000.00 | NULL |

| 4 | kaushik | 25 | Mumbai | 6500.00 | NULL |

| 5 | Hardik | 27 | Bhopal | 8500.00 | NULL |

| 6 | Komal | 22 | MP | 4500.00 | NULL |

| 7 | Muffy | 24 | Indore | 10000.00 | NULL |

+----+---------+-----+-----------+----------+------+

Following is the example to DROP sex column from existing table:

ALTER TABLE CUSTOMERS DROP SEX;

Now, CUSTOMERS table is changed and following would be output from SELECT statement:

+----+---------+-----+-----------+----------+

| ID | NAME | AGE | ADDRESS | SALARY |

+----+---------+-----+-----------+----------+

| 1 | Ramesh | 32 | Ahmedabad | 2000.00 |

| 2 | Ramesh | 25 | Delhi | 1500.00 |

| 3 | kaushik | 23 | Kota | 2000.00 |

| 4 | kaushik | 25 | Mumbai | 6500.00 |

| 5 | Hardik | 27 | Bhopal | 8500.00 |

| 6 | Komal | 22 | MP | 4500.00 |

| 7 | Muffy | 24 | Indore | 10000.00 |

iii) **SQL DROP or DELETE Table**

When an existing object is not required for further use, it is always better to eliminate it from the database. The SQL **DROP TABLE** statement is used to remove a table definition and all data, indexes, triggers, constraints, and permission specifications for that table.

**NOTE:** We have to be careful while using this command because once a table is deleted then all the information available in the table would also be lost forever.

**Syntax:**

Basic syntax of DROP TABLE statement is as follows:

**DROP TABLE table\_name;**

Example:

Let us first verify CUSTOMERS table and then we would delete it from the database:

SQL> DESC CUSTOMERS;

+---------+---------------+------+-----+---------+-------+

| Field | Type | Null | Key | Default | Extra |

+---------+---------------+------+-----+---------+-------+

| ID | int(11) | NO | PRI | | |

| NAME | varchar(20) | NO | | | |

| AGE | int(11) | NO | | | |

| ADDRESS | char(25) | YES | | NULL | |

| SALARY | decimal(18,2) | YES | | NULL | |

+---------+---------------+------+-----+---------+-------+

5 rows in set (0.00 sec)

This means CUSTOMERS table is available in the database, so let us drop it as follows:

**SQL> DROP TABLE CUSTOMERS;**

Query OK, 0 rows affected (0.01 sec)

Now, if we would try DESC command, then we would get error as follows:

**SQL> DESC CUSTOMERS;**

ERROR 1146 (42S02): Table 'TEST.CUSTOMERS' doesn't exist

Here, TEST is database name which we are using for our examples

**2. DML-Data Manipulation Language**

It is a set of SQL Commands that allows changing data within the database.

**i) SQL INSERT Query**

The SQL **INSERT INTO** Statement is used to add new rows of data to a table in the database.

**Syntax:** There are two basic syntaxes of INSERT INTO statement as follows:

**INSERT INTO TABLE\_NAME (column1, column2, column3,...columnN) VALUES (value1, value2, value3,...valueN);**

Here, column1, column2,...columnN are the names of the columns in the table into which we want to insert data.

We may not need to specify the column(s) name in the SQL query if we are adding values for all the columns of the table. But make sure the order of the values is in the same order as the columns in the table. The **SQL INSERT INTO** syntax would be as follows:

**INSERT INTO TABLE\_NAME VALUES (value1,value2,value3,...valueN);**

Example:

Following statements would create six records in CUSTOMERS table:

INSERT INTO CUSTOMERS (ID,NAME,AGE,ADDRESS,SALARY)

VALUES (1, 'Ramesh', 32, 'Ahmedabad', 2000.00 );

INSERT INTO CUSTOMERS (ID,NAME,AGE,ADDRESS,SALARY)

VALUES (2, 'Khilan', 25, 'Delhi', 1500.00 );

INSERT INTO CUSTOMERS (ID,NAME,AGE,ADDRESS,SALARY)

VALUES (3, 'kaushik', 23, 'Kota', 2000.00 );

INSERT INTO CUSTOMERS (ID,NAME,AGE,ADDRESS,SALARY)

VALUES (4, 'Chaitali', 25, 'Mumbai', 6500.00 );

INSERT INTO CUSTOMERS (ID,NAME,AGE,ADDRESS,SALARY)

VALUES (5, 'Hardik', 27, 'Bhopal', 8500.00 );

INSERT INTO CUSTOMERS (ID,NAME,AGE,ADDRESS,SALARY)

VALUES (6, 'Komal', 22, 'MP', 4500.00 );

We can create a record in CUSTOMERS table using **second syntax** as follows:

**INSERT INTO CUSTOMERS VALUES (7, 'Muffy', 24, 'Indore', 10000.00 );**

All the above statements would produce the following records in CUSTOMERS table:

+----+----------+-----+-----------+----------+

| ID | NAME | AGE | ADDRESS | SALARY |

+----+----------+-----+-----------+----------+

| 1 | Ramesh | 32 | Ahmedabad | 2000.00 |

| 2 | Khilan | 25 | Delhi | 1500.00 |

| 3 | kaushik | 23 | Kota | 2000.00 |

| 4 | Chaitali | 25 | Mumbai | 6500.00 |

| 5 | Hardik | 27 | Bhopal | 8500.00 |

| 6 | Komal | 22 | MP | 4500.00 |

| 7 | Muffy | 24 | Indore | 10000.00 |

+----+----------+-----+-----------+----------+

**ii) SQL UPDATE Query:** The SQL **UPDATE** Query is used to modify the existing records in a table. We can use WHERE clause with UPDATE query to update selected rows, otherwise all the rows would be affected.

**Syntax:** The basic syntax of UPDATE query with WHERE clause is as follows:

**UPDATE table\_name SET column1=value1, column2=value2...., columnN = valueN WHERE condition;**

We can combine N number of conditions using AND or OR operators.

Example: Following is an example, which would update ADDRESS for a customer whose ID is 6:

**SQL> UPDATE CUSTOMERS SET ADDRESS = 'Pune' WHERE ID = 6;**

Now, CUSTOMERS table would have the following records:

+----+----------+-----+-----------+----------+

| ID | NAME | AGE | ADDRESS | SALARY |

+----+----------+-----+-----------+----------+

| 1 | Ramesh | 32 | Ahmedabad | 2000.00 |

| 2 | Khilan | 25 | Delhi | 1500.00 |

| 3 | kaushik | 23 | Kota | 2000.00 |

| 4 | Chaitali | 25 | Mumbai | 6500.00 |

| 5 | Hardik | 27 | Bhopal | 8500.00 |

| 6 | Komal | 22 | Pune | 4500.00 |

| 7 | Muffy | 24 | Indore | 10000.00 |

+----+----------+-----+-----------+----------+

If we want to modify all ADDRESS and SALARY column values in CUSTOMERS table, we do not need to use WHERE clause and UPDATE query would be as follows:

**SQL> UPDATE CUSTOMERS SET ADDRESS = 'Pune', SALARY = 1000.00;**

Now, CUSTOMERS table would have the following records:

+----+----------+-----+---------+---------+

| ID | NAME | AGE | ADDRESS | SALARY |

+----+----------+-----+---------+---------+

| 1 | Ramesh | 32 | Pune | 1000.00 |

| 2 | Khilan | 25 | Pune | 1000.00 |

| 3 | kaushik | 23 | Pune | 1000.00 |

| 4 | Chaitali | 25 | Pune | 1000.00 |

| 5 | Hardik | 27 | Pune | 1000.00 |

| 6 | Komal | 22 | Pune | 1000.00 |

| 7 | Muffy | 24 | Pune | 1000.00 |

1. **SQL DELETE Query:** The SQL **DELETE** Query is used to delete the existing records from a table. We can use WHERE clause with DELETE query to delete selected rows, otherwise all the records would be deleted.

**Syntax:** The basic syntax of DELETE query with WHERE clause is as follows:

**DELETE FROM table\_name WHERE condition;**

We can combine N number of conditions using AND or OR operators.

Example:

Consider the CUSTOMERS table having the following records:

+----+----------+-----+-----------+----------+

| ID | NAME | AGE | ADDRESS | SALARY |

+----+----------+-----+-----------+----------+

| 1 | Ramesh | 32 | Ahmedabad | 2000.00 |

| 2 | Khilan | 25 | Delhi | 1500.00 |

| 3 | kaushik | 23 | Kota | 2000.00 |

| 4 | Chaitali | 25 | Mumbai | 6500.00 |

| 5 | Hardik | 27 | Bhopal | 8500.00 |

| 6 | Komal | 22 | MP | 4500.00 |

| 7 | Muffy | 24 | Indore | 10000.00 |

+----+----------+-----+-----------+----------+

Following is an example, which would DELETE a customer, whose ID is 6:

**SQL> DELETE FROM CUSTOMERS WHERE ID = 6;**

Now, CUSTOMERS table would have the following records:

+----+----------+-----+-----------+----------+

| ID | NAME | AGE | ADDRESS | SALARY |

+----+----------+-----+-----------+----------+

| 1 | Ramesh | 32 | Ahmedabad | 2000.00 |

| 2 | Khilan | 25 | Delhi | 1500.00 |

| 3 | kaushik | 23 | Kota | 2000.00 |

| 4 | Chaitali | 25 | Mumbai | 6500.00 |

| 5 | Hardik | 27 | Bhopal | 8500.00 |

| 7 | Muffy | 24 | Indore | 10000.00 |

+----+----------+-----+-----------+----------+

If we want to DELETE all the records from CUSTOMERS table, we do not need to use WHERE clause and DELETE query would be as follows:

**SQL> DELETE FROM CUSTOMERS;**

Now, CUSTOMERS table would not have any record.

1. **DQL: Data Query Language**

It is used for querying database i.e selecting/retrieving records from the database.

1. **SQL SELECT Query**

SQL **SELECT** Statement is used to fetch the data from a database table which returns data in the form of result table. These result tables are called result-sets.

Syntax: The basic syntax of SELECT statement is as follows:

**SELECT column1, column2, columnN FROM table\_name;**

Here, column1, column2...are the fields of a table whose values we want to fetch. If we want to fetch all the fields available in the field, then we can use the following syntax:

**SELECT \* FROM table\_name;**

Example: Consider the CUSTOMERS table.

Following is an example, which would fetch ID, Name and Salary fields of the customers available in CUSTOMERS table:

SQL> SELECT ID, NAME, SALARY FROM CUSTOMERS;

This would produce the following result:

+----+----------+----------+

| ID | NAME | SALARY |

+----+----------+----------+

| 1 | Ramesh | 2000.00 |

| 2 | Khilan | 1500.00 |

| 3 | kaushik | 2000.00 |

| 4 | Chaitali | 6500.00 |

| 5 | Hardik | 8500.00 |

| 6 | Komal | 4500.00 |

| 7 | Muffy | 10000.00 |

+----+----------+----------+

If we want to fetch all the fields of CUSTOMERS table, then use the following query:

SQL> SELECT \* FROM CUSTOMERS;

This would produce the following result:

+----+----------+-----+-----------+----------+

| ID | NAME | AGE | ADDRESS | SALARY |

+----+----------+-----+-----------+----------+

| 1 | Ramesh | 32 | Ahmedabad | 2000.00 |

| 2 | Khilan | 25 | Delhi | 1500.00 |

| 3 | kaushik | 23 | Kota | 2000.00 |

| 4 | Chaitali | 25 | Mumbai | 6500.00 |

| 5 | Hardik | 27 | Bhopal | 8500.00 |

| 6 | Komal | 22 | MP | 4500.00 |

| 7 | Muffy | 24 | Indore | 10000.00 |

1. **SELECT with WHERE Clause**

The SQL **WHERE** clause is used to specify a condition while fetching the data from single table or joining with multiple tables. If the given condition is satisfied, then only it returns specific value from the table.

Syntax: The basic syntax of SELECT statement with WHERE clause is as follows:

**SELECT column1, column2, columnN FROM table\_name WHERE condition;**

**Variations of SELECT Command:**

We can specify a condition using comparison (relational) or logical operators like >, <, =, LIKE, NOT etc. Below examples would make this concept clear.

Example:

Consider the CUSTOMERS table.

Following is an example, which would fetch ID, Name and Salary fields from the CUSTOMERS table where salary is greater than 2000:

SQL> SELECT ID, NAME, SALARY FROM CUSTOMERS WHERE SALARY > 2000;

This would produce the following result:

+----+----------+----------+

| ID | NAME | SALARY |

+----+----------+----------+

| 4 | Chaitali | 6500.00 |

| 5 | Hardik | 8500.00 |

| 6 | Komal | 4500.00 |

| 7 | Muffy | 10000.00 |

+----+----------+----------+

Following is an example, which would fetch ID, Name and Salary fields from the CUSTOMERS table for a customer with name **Hardik**. Here, it is important to note that all the strings should be given inside single quotes ('') where as numeric values should be given without any quote as in above example:

**SQL> SELECT ID, NAME, SALARY FROM CUSTOMERS WHERE NAME = 'Hardik';**

This would produce the following result:

+----+----------+----------+

| ID | NAME | SALARY |

+----+----------+----------+

| 5 | Hardik | 8500.00 |

There are six main clauses in a select statement:

* **FROM** which tables are required for the query
* **WHERE** the condition to filter the required rows
* **GROUP BY** how to aggregate the rows
* **HAVING** condition to filter aggregated groups
* **SELECT** the required result columns, i.e. the **project**
* **ORDER BY** the ordering imposed on the result set

**SQL GROUP BY Clause:**

SELECT SUM(column\_name) FROM table\_name

WHERE CONDITION GROUP BY column\_name;

**SQL HAVING Clause:**

SELECT SUM(column\_name) FROM table\_name

WHERE CONDITION GROUP BY column\_name

HAVING (arithematic function condition);

**SQL ORDER BY Clause:**

* It is used in the last portion of select statement
* By using this rows can be sorted
* By default it takes ascending order
* DESC: is used for sorting in descending order
* Sorting by column which is not in select list is possible
* Sorting by column Alias

SELECT column1, column2....columnN FROM table\_name

WHERE CONDITION ORDER BY column\_name {ASC|DESC};

**4. Data Control Language (DCL)**

It consists of the commands that control the user access to the database objects. It is mainly related to security issues i. e determining who has access to the database objects and what operations they can perform on them. It includes command to grant or revoke privileges to access the within the database.

**i) SQL GRANT Command:**

It is used by the DBA to grant privileges to the users.

**Syntax:**

**GRANT privilege(s) ON table\_name/view\_name TO <user\_id>, <group\_id>, <public>;**

**Example:**

**GRANT CREATE ON ITEMS TO user1;**

**GRANT CREATE, UPDATE, DROP, INSERT, SELECT ON ITEMS TO user1;**

**ii) SQL REVOKE Command:**

This command is used by the DBA to revoke privileges from the users. It is the opposite to the GRANT Command.

**Syntax:**

**REVOKE privilege(s) ON table\_name/view\_name FROM <user\_id>, <group\_id>, <public>;**

**Example:**

**REVOKE CREATE ON ITEMS FROM user1;**

**REVOKE CREATE, UPDATE, DROP, INSERT, SELECT ON ITEMS FROM user1;**

Here the DBA has revoked the privileges that were previously granted to user\_id named user1.

**SQL Integrity Constraints**

Constraints are the rules enforced on data columns on table. These are used to limit the type of data that can go into a table. This ensures the accuracy and reliability of the data in the database.

Constraints could be column level or table level. Column level constraints are applied only to one column, whereas table level constraints are applied to the whole table.

Following are commonly used constraints available in SQL:

1. **PRIMARY Key**: Uniquely identified each rows/records in a database table
2. **NOT NULL** Constraint: Ensures that a column cannot have NULL value.
3. **DEFAULT** Constraint: Provides a default value for a column when none is specified.
4. **UNIQUE** Constraint: Ensures that all values in a column are different.
5. **FOREIGN Key**: Uniquely identified a rows/records in any another database table.
6. **CHECK Constraint**: The CHECK constraint ensures that all values in a column satisfy certain conditions.
7. **PRIMARY Key:**

A primary key is a field in a table which uniquely identifies each row/record in a database table. Primary keys must contain unique values. A primary key column cannot have NULL values. A table can have only one primary key, which may consist of single or multiple fields. When multiple fields are used as a primary key, they are called a **composite key**. If a table has a primary key defined on any field(s), then we can not have two records having the same value of that field(s).

Create Primary Key: Here is the syntax to define ID attribute as a primary key in a CUSTOMERS table.

**CREATE TABLE CUSTOMERS ( ID INT NOT NULL, NAME VARCHAR (20) NOT NULL,**

**AGE INT NOT NULL, ADDRESS CHAR (25), SALARY DECIMAL (18, 2), PRIMARY KEY (ID));**

To create a PRIMARY KEY constraint on the "ID" column when CUSTOMERS table already exists, use the following SQL syntax:

**ALTER TABLE CUSTOMER ADD PRIMARY KEY (ID);**

**NOTE:** If we use the ALTER TABLE statement to add a primary key, the primary key column(s) must already have been declared to not contain NULL values (when the table was first created).

For defining a PRIMARY KEY constraint on multiple columns, use the following SQL syntax:

**CREATE TABLE CUSTOMERS (**

**ID INT NOT NULL,**

**NAME VARCHAR (20) NOT NULL,**

**AGE INT NOT NULL UNIQUE,**

**ADDRESS CHAR (25),**

**SALARY DECIMAL (18, 2),**

**PRIMARY KEY (ID, NAME)**

**);**

To create a PRIMARY KEY constraint on the "ID" and "NAMES" columns when CUSTOMERS table already exists, use the following SQL syntax:

**ALTER TABLE CUSTOMERS ADD CONSTRAINT PK\_CUSTID PRIMARY KEY (ID, NAME);**

Delete Primary Key: We can clear the primary key constraints from the table, Use Syntax:

**ALTER TABLE CUSTOMERS DROP PRIMARY KEY;**

1. **NOT NULL Constraint:**

By default, a column can hold NULL values. If we do not want a column to have a NULL value, then we need to define such constraint on this column specifying that NULL is now not allowed for that column. A NULL is not the same as no data, rather, it represents unknown data.

Example:

For example, the following SQL creates a new table called CUSTOMERS and adds five columns, three of which, ID and NAME and AGE, specify not to accept NULLs:

**CREATE TABLE CUSTOMERS (**

**ID INT NOT NULL,**

**NAME VARCHAR (20) NOT NULL,**

**AGE INT NOT NULL, ADDRESS CHAR (25) ,**

**SALARY DECIMAL (18, 2), PRIMARY KEY (ID)**

**);**

If CUSTOMERS table has already been created, then to add a NOT NULL constraint to SALARY column in Oracle and MySQL, we would write a statement similar to the following:

**ALTER TABLE CUSTOMERS MODIFY SALARY DECIMAL (18, 2) NOT NULL;**

1. **DEFAULT Constraint:**

The DEFAULT constraint provides a default value to a column when the INSERT INTO statement does not provide a specific value. Example:

For example, the following SQL creates a new table called CUSTOMERS and adds five columns. Here, SALARY column is set to 5000.00 by default, so in case INSERT INTO statement does not provide a value for this column. Then by default this column would be set to 5000.00.

**CREATE TABLE CUSTOMERS (ID INT PRIMARY KEY, NAME VARCHAR (20) NOT NULL, AGE INT NOT NULL, ADDRESS CHAR (25) , SALARY DECIMAL (18, 2) DEFAULT 5000.00 );**

If CUSTOMERS table has already been created, then to add a DEFAULT constraint to SALARY column, we would write a statement similar to the following:

**ALTER TABLE CUSTOMERS MODIFY SALARY DECIMAL (18, 2) DEFAULT 5000.00;**

**Drop Default Constraint:**

To drop a DEFAULT constraint, use the following SQL:

**ALTER TABLE CUSTOMERS ALTER COLUMN SALARY DROP DEFAULT;**

1. **UNIQUE Constraint:**

The UNIQUE Constraint prevents two records from having identical values in a particular column. In the CUSTOMERS table, for example, we might want to prevent two or more people from having identical age.

Example: the following SQL creates a new table called CUSTOMERS and adds five columns. Here, AGE column is set to UNIQUE, so that we cannot have two records with same age:

**CREATE TABLE CUSTOMERS (ID INT PRIMARY KEY, NAME VARCHAR (20) NOT NULL, AGE INT NOT NULL UNIQUE, ADDRESS CHAR (25), SALARY DECIMAL (18, 2));**

If CUSTOMERS table has already been created, then to add a UNIQUE constraint to AGE column, we would write a statement similar to the following:

**ALTER TABLE CUSTOMERS MODIFY AGE INT NOT NULL UNIQUE;**

We can also use following syntax, which supports naming the constraint in multiple columns as well:

**ALTER TABLE CUSTOMERS ADD CONSTRAINT myUniqueConstraint UNIQUE(AGE, SALARY);**

DROP a UNIQUE Constraint:

To drop a UNIQUE constraint, use the following SQL:

**ALTER TABLE CUSTOMERS DROP CONSTRAINT myUniqueConstraint;**

If we are using MySQL, then we can use the following syntax:

**ALTER TABLE CUSTOMERS DROP INDEX myUniqueConstraint;**

1. **FOREIGN Key:**

A foreign key is a key used to link two tables together. This is sometimes called a referencing key. Foreign Key is a column or a combination of columns whose values match a Primary Key in a different table.

**The relationship between 2 tables matches the Primary Key in one of the tables with a Foreign Key in the second table.** If a table has a primary key defined on any field(s), then we cannot have two records having the same value of that field(s).

Example: Consider the structure of the two tables as follows:

***CUSTOMERS***table:

**CREATE TABLE CUSTOMERS (**

**ID INT PRIMARY KEY,**

**NAME VARCHAR (20) NOT NULL,**

**AGE INT NOT NULL UNIQUE,**

**ADDRESS CHAR (25),**

**SALARY DECIMAL (18, 2));**

***ORDERS*** table:

**CREATE TABLE ORDERS (**

**ID INT PRIMARY KEY,**

**DATE DATETIME,**

**CUSTOMER\_ID INT references CUSTOMERS (ID),**

**AMOUNT double);**

If ORDERS table has already been created, and the foreign key has not yet been set, use the syntax for specifying a foreign key by altering a table.

**ALTER TABLE ORDERS ADD FOREIGN KEY (Customer\_ID) REFERENCES CUSTOMERS (ID);**

**DROP a FOREIGN KEY Constraint:**

To drop a FOREIGN KEY constraint, use the following SQL:

**ALTER TABLE ORDERS DROP FOREIGN KEY;**

1. **CHECK Constraint:**

The CHECK Constraint enables a condition to check the value being entered into a record. If the condition evaluates to false, the record violates the constraint and isn’t entered into the table. Example: the following SQL creates a new table called CUSTOMERS and adds five columns. Here, we add a CHECK with AGE column, so that we can not have any CUSTOMER below 18 years:

**CREATE TABLE CUSTOMERS (**

**ID INT PRIMARY KEY,**

**NAME VARCHAR (20) NOT NULL,**

**AGE INT NOT NULL CHECK (AGE >= 18),**

**ADDRESS CHAR (25),**

**SALARY DECIMAL (18, 2));**

If CUSTOMERS table has already been created, then to add a CHECK constraint to AGE column, we would write a statement similar to the following:

**ALTER TABLE CUSTOMERS MODIFY AGE INT NOT NULL CHECK (AGE >= 18);**

We can also use following syntax, which supports naming the constraint in multiple columns as well:

**ALTER TABLE CUSTOMERS ADD CONSTRAINT myCheckConstraint CHECK(AGE >= 18);**

**DROP a CHECK Constraint:**

To drop a CHECK constraint, use the following SQL. This syntax does not work with MySQL:

**ALTER TABLE CUSTOMERS DROP CONSTRAINT myCheckConstraint;**

**SQL Syntax(Quick Reference)**

SQL is followed by unique set of rules and guidelines called Syntax. This tutorial gives us a quick start with SQL by listing all the basic SQL Syntax: All the SQL statements start with any of the keywords like SELECT, INSERT, UPDATE, DELETE, ALTER, DROP, CREATE, USE, SHOW and all the statements end with a semicolon (;).

Important point to be noted is that SQL is **case insensitive,** which means SELECT and select have same meaning in SQL statements, but MySQL makes difference in table names. So if we are working with MySQL, then we need to give table names as they exist in the database.

**SQL SELECT Statement:**

SELECT column1, column2....columnN FROM table\_name;

**SQL DISTINCT Clause:**

SELECT DISTINCT column1, column2....columnN FROM table\_name;

**SQL WHERE Clause:**

SELECT column1, column2....columnN FROM table\_name

WHERE CONDITION;

**SQL IN Clause:**

SELECT column1, column2....columnN FROM table\_name

WHERE column\_name IN (val-1, val-2,...val-N);

**SQL AND/OR Clause:**

SELECT column1, column2....columnN FROM table\_name

WHERE CONDITION-1 {AND|OR} CONDITION-2;

**SQL BETWEEN Clause:**

SELECT column1, column2....columnN FROM table\_name

WHERE column\_name BETWEEN val-1 AND val-2;

**SQL LIKE Clause:**

SELECT column1, column2....columnN FROM table\_name

WHERE column\_name LIKE { PATTERN };

**SQL ORDER BY Clause:**

SELECT column1, column2....columnN FROM table\_name

WHERE CONDITION ORDER BY column\_name {ASC|DESC};

**SQL GROUP BY Clause:**

SELECT SUM(column\_name) FROM table\_name

WHERE CONDITION GROUP BY column\_name;

**SQL COUNT Clause:**

SELECT COUNT(column\_name) FROM table\_name

WHERE CONDITION;

**SQL HAVING Clause:**

SELECT SUM(column\_name) FROM table\_name

WHERE CONDITION

GROUP BY column\_name

HAVING (arithematic function condition);

**SQL CREATE TABLE Statement:**

CREATE TABLE table\_name (

column1 datatype,

column2 datatype,

.....

columnN datatype,

PRIMARY KEY( one or more columns ));

**SQL DROP TABLE Statement:**

DROP TABLE table\_name;

**SQL CREATE INDEX Statement:**

CREATE UNIQUE INDEX index\_name

ON table\_name ( column1, column2,...columnN);

**SQL DROP INDEX Statement:**

ALTER TABLE table\_name

DROP INDEX index\_name;

**SQL DESC Statement:**

DESC table\_name;

**SQL TRUNCATE TABLE Statement:**

TRUNCATE TABLE table\_name;

**SQL ALTER TABLE Statement:**

ALTER TABLE table\_name {ADD|DROP|MODIFY} column\_name {data\_ype};

**SQL ALTER TABLE Statement (Rename):**

ALTER TABLE table\_name RENAME TO new\_table\_name;

**SQL INSERT INTO Statement:**

INSERT INTO table\_name( column1, column2....columnN) VALUES ( value1, alue2....valueN);

**SQL UPDATE Statement:**

UPDATE table\_name

SET column1 = value1, column2 = value2....columnN=valueN

[ WHERE CONDITION ];

**SQL DELETE Statement:**

DELETE FROM table\_name

WHERE {CONDITION};

**SQL CREATE DATABASE Statement:**

**CREATE DATABASE database\_name;**

**SQL DROP DATABASE Statement:**

DROP DATABASE database\_name;

**SQL USE Statement:**

USE DATABASE database\_name;

**SQL COMMIT Statement:**

COMMIT;

**SQL ROLLBACK Statement:**

ROLLBACK;

Here are simple examples showing usage of SQL Arithmetic Operators:

**SQL> select 10+ 20;**

+--------+

| 10+ 20 |

+--------+

| 30 |

+--------+

1 row in set (0.00 sec)

**SQL> select 10 \* 20;**

+---------+

| 10 \* 20 |

+---------+

| 200 |

+---------+

1 row in set (0.00 sec)

**SQL> select 10 / 5;**

+--------+

| 10 / 5 |

+--------+

| 2.0000 |

+--------+

1 row in set (0.03 sec)

**SQL> select 12 % 5;**

+---------+

| 12 % 5 |

+---------+

| 2 |

+---------+

1 row in set (0.00 sec)

Consider the CUSTOMERS table having the following records:

**SQL> SELECT \* FROM CUSTOMERS;**

+----+----------+-----+-----------+----------+

| ID | NAME | AGE | ADDRESS | SALARY |

+----+----------+-----+-----------+----------+

| 1 | Ramesh | 32 | Ahmedabad | 2000.00 |

| 2 | Khilan | 25 | Delhi | 1500.00 |

| 3 | kaushik | 23 | Kota | 2000.00 |

| 4 | Chaitali | 25 | Mumbai | 6500.00 |

| 5 | Hardik | 27 | Bhopal | 8500.00 |

| 6 | Komal | 22 | MP | 4500.00 |

| 7 | Muffy | 24 | Indore | 10000.00 |

+----+----------+-----+-----------+----------+

7 rows in set (0.00 sec)

Here are simple examples showing usage of SQL Comparison Operators:

**SQL> SELECT \* FROM CUSTOMERS WHERE SALARY > 5000;**

+----+----------+-----+---------+----------+

| ID | NAME | AGE | ADDRESS | SALARY |

+----+----------+-----+---------+----------+

| 4 | Chaitali | 25 | Mumbai | 6500.00 |

| 5 | Hardik | 27 | Bhopal | 8500.00 |

| 7 | Muffy | 24 | Indore | 10000.00 |

+----+----------+-----+---------+----------+

3 rows in set (0.00 sec)

**SQL> SELECT \* FROM CUSTOMERS WHERE SALARY = 2000;**

+----+---------+-----+-----------+---------+

| ID | NAME | AGE | ADDRESS | SALARY |

+----+---------+-----+-----------+---------+

| 1 | Ramesh | 32 | Ahmedabad | 2000.00 |

| 3 | kaushik | 23 | Kota | 2000.00 |

+----+---------+-----+-----------+---------+

2 rows in set (0.00 sec)

**SQL> SELECT \* FROM CUSTOMERS WHERE SALARY != 2000;**

+----+----------+-----+---------+----------+

| ID | NAME | AGE | ADDRESS | SALARY |

+----+----------+-----+---------+----------+

| 2 | Khilan | 25 | Delhi | 1500.00 |

| 4 | Chaitali | 25 | Mumbai | 6500.00 |

| 5 | Hardik | 27 | Bhopal | 8500.00 |

| 6 | Komal | 22 | MP | 4500.00 |

| 7 | Muffy | 24 | Indore | 10000.00 |

+----+----------+-----+---------+----------+

5 rows in set (0.00 sec)

**SQL> SELECT \* FROM CUSTOMERS WHERE SALARY <> 2000;**

+----+----------+-----+---------+----------+|

ID | NAME | AGE | ADDRESS | SALARY |

+----+----------+-----+---------+----------+

| 2 | Khilan | 25 | Delhi | 1500.00 |

| 4 | Chaitali | 25 | Mumbai | 6500.00 |

| 5 | Hardik | 27 | Bhopal | 8500.00 |

| 6 | Komal | 22 | MP | 4500.00 |

| 7 | Muffy | 24 | Indore | 10000.00 |

+----+----------+-----+---------+----------+

5 rows in set (0.00 sec)

**SQL> SELECT \* FROM CUSTOMERS WHERE SALARY >= 6500;**

+----+----------+-----+---------+----------+

| ID | NAME | AGE | ADDRESS | SALARY |

+----+----------+-----+---------+----------+

| 4 | Chaitali | 25 | Mumbai | 6500.00 |

| 5 | Hardik | 27 | Bhopal | 8500.00 |

| 7 | Muffy | 24 | Indore | 10000.00 |

+----+----------+-----+---------+----------+

3 rows in set (0.00 sec)

Here are simple examples showing usage of **SQL Comparison and Logical Operators:**

SQL> SELECT \* FROM CUSTOMERS WHERE AGE >= 25 AND SALARY >= 6500;

+----+----------+-----+---------+---------+

| ID | NAME | AGE | ADDRESS | SALARY |

| 4 | Chaitali | 25 | Mumbai | 6500.00 |

| 5 | Hardik | 27 | Bhopal | 8500.00 |

+----+----------+-----+---------+---------+

2 rows in set (0.00 sec)

SQL> SELECT \* FROM CUSTOMERS WHERE AGE >= 25 OR SALARY >= 6500;

+----+----------+-----+-----------+----------+

| ID | NAME | AGE | ADDRESS | SALARY |

| 1 | Ramesh | 32 | Ahmedabad | 2000.00 |

| 2 | Khilan | 25 | Delhi | 1500.00 |

| 4 | Chaitali | 25 | Mumbai | 6500.00 |

| 5 | Hardik | 27 | Bhopal | 8500.00 |

| 7 | Muffy | 24 | Indore | 10000.00 |

+----+----------+-----+-----------+----------+

5 rows in set (0.00 sec)

SQL> SELECT \* FROM CUSTOMERS WHERE AGE IS NOT NULL;

+----+----------+-----+-----------+----------+

| ID | NAME | AGE | ADDRESS | SALARY |

| 1 | Ramesh | 32 | Ahmedabad | 2000.00 |

| 2 | Khilan | 25 | Delhi | 1500.00 |

| 3 | kaushik | 23 | Kota | 2000.00 |

| 4 | Chaitali | 25 | Mumbai | 6500.00 |

| 5 | Hardik | 27 | Bhopal | 8500.00 |

| 6 | Komal | 22 | MP | 4500.00 |

| 7 | Muffy | 24 | Indore | 10000.00 |

+----+----------+-----+-----------+----------+

7 rows in set (0.00 sec)

SQL> SELECT \* FROM CUSTOMERS WHERE NAME LIKE 'Ko%';

+----+-------+-----+---------+---------+

| ID | NAME | AGE | ADDRESS | SALARY |

| 6 | Komal | 22 | MP | 4500.00 |

+----+-------+-----+---------+---------+

1 row in set (0.00 sec)

SQL> SELECT\* FROM CUSTOMERS WHERE AGE IN ( 25, 27 );

+----+----------+-----+---------+---------+

| ID | NAME | AGE | ADDRESS | SALARY |

| 2 | Khilan | 25 | Delhi | 1500.00 |

| 4 | Chaitali | 25 | Mumbai | 6500.00 |

| 5 | Hardik | 27 | Bhopal | 8500.00 |

+----+----------+-----+---------+---------+

3 rows in set (0.00 sec)

SQL> SELECT \* FROM CUSTOMERS WHERE AGE BETWEEN 25 AND 27;

+----+----------+-----+---------+---------+

| ID | NAME | AGE | ADDRESS | SALARY |

| 2 | Khilan | 25 | Delhi | 1500.00 |

| 4 | Chaitali | 25 | Mumbai | 6500.00 |

| 5 | Hardik | 27 | Bhopal | 8500.00 |

+----+----------+-----+---------+---------+

3 rows in set (0.00 sec)

SQL> SELECT AGE FROM CUSTOMERS

WHERE EXISTS (SELECT AGE FROM CUSTOMERS WHERE SALARY > 6500);

**| AGE** |

+-----+

| 32 |

| 25 |

| 23 |

| 25 |

| 27 |

| 22 |

| 24 |

+-----+

7 rows in set (0.02 sec)

SQL> SELECT \* FROM CUSTOMERS

WHERE AGE > ALL (SELECT AGE FROM CUSTOMERS WHERE SALARY > 6500);

+----+--------+-----+-----------+---------+

| ID | NAME | AGE | ADDRESS | SALARY |

| 1 | Ramesh | 32 | Ahmedabad | 2000.00 |

+----+--------+-----+-----------+---------+

1 row in set (0.02 sec)

SQL> SELECT \* FROM CUSTOMERS WHERE AGE > ANY (SELECT AGE FROM CUSTOMERS WHERE SALARY > 6500);

+----+----------+-----+-----------+---------+

| ID | NAME | AGE | ADDRESS | SALARY |

| 1 | Ramesh | 32 | Ahmedabad | 2000.00 |

| 2 | Khilan | 25 | Delhi | 1500.00 |

| 4 | Chaitali | 25 | Mumbai | 6500.00 |

| 5 | Hardik | 27 | Bhopal | 8500.00 |

+----+----------+-----+-----------+---------+

4 rows in set (0.00 sec)

**SQL Expressions**

An expression is a combination of one or more values, operators, and SQL functions that evaluate to a value. SQL EXPRESSIONs are like formulas and they are written in query language. We can also use them to query the database for specific set of data.

Syntax: Consider the basic syntax of the SELECT statement as follows:

**SELECT column1, column2, columnN**

**FROM table\_name WHERE [CONDITION|EXPRESSION];**

There are different types of SQL expressions, which are mentioned below:

***SQL -Boolean Expressions:*** SQL Boolean Expressions fetch the data on the basis of matching single value. Following is the syntax:

**SELECT column1, column2, columnN FROM table\_name**

**WHERE SINGLE VALUE MATCHTING EXPRESSION;**

Consider the CUSTOMERS table having the following records:

SQL> SELECT \* FROM CUSTOMERS;

+----+----------+-----+-----------+----------+

| ID | NAME | AGE | ADDRESS | SALARY |

+----+----------+-----+-----------+----------+

| 1 | Ramesh | 32 | Ahmedabad | 2000.00 |

| 2 | Khilan | 25 | Delhi | 1500.00 |

| 3 | kaushik | 23 | Kota | 2000.00 |

| 4 | Chaitali | 25 | Mumbai | 6500.00 |

| 5 | Hardik | 27 | Bhopal | 8500.00 |

| 6 | Komal | 22 | MP | 4500.00 |

| 7 | Muffy | 24 | Indore | 10000.00 |

+----+----------+-----+-----------+----------+ 7 rows in set (0.00 sec)

Here is simple example showing usage of SQL Boolean Expressions:

SQL> SELECT \* FROM CUSTOMERS WHERE SALARY = 10000;

+----+-------+-----+---------+----------+

| ID | NAME | AGE | ADDRESS | SALARY |

+----+-------+-----+---------+----------+

| 7 | Muffy | 24 | Indore | 10000.00 |

+----+-------+-----+---------+----------+

1 row in set (0.00 sec)

***SQL -Numeric Expression:*** This expression is used to perform any mathematical operation in any query. Following is the syntax:

**SELECT numerical\_expression as OPERATION\_NAME**

**[FROM table\_name WHERE CONDITION] ;**

Here numerical\_expression is used for mathematical expression or any formula. Following is a simple examples showing usage of SQL Numeric Expressions:

SQL> SELECT (15 + 6) AS ADDITION

+----------+

| ADDITION |

+----------+

| 21 |

+----------+

1 row in set (0.00 sec)

There are several **built-in functions like avg(), sum(), count(), etc.,** to perform what is known as aggregate data calculations against a table or a specific table column.

SQL> SELECT COUNT(\*) AS "RECORDS" FROM CUSTOMERS;

+---------+

| RECORDS |

+---------+

| 7 |

+---------+

1 row in set (0.00 sec)

***SQL -Date Expressions:*** Date Expressions return current system date and time values:

SQL> SELECT CURRENT\_TIMESTAMP;

+---------------------+

| Current\_Timestamp |

+---------------------+

| 2009-11-12 06:40:23 |

+---------------------+

1 row in set (0.00 sec)

Another date expression is as follows:

SQL> SELECT GETDATE();;

+-------------------------+

| GETDATE |

+-------------------------+

| 2009-10-22 12:07:18.140 |

+-------------------------+

1 row in set (0.00 sec)

**SQL AND and OR Operators**

The SQL **AND** and **OR** operators are used to combine multiple conditions to narrow data in an SQL statement. These two operators are called conjunctive operators. These operators provide a means to make multiple comparisons with different operators in the same SQL statement.

The AND Operator: The **AND** operator allows the existence of multiple conditions in an SQL statement's WHERE clause.

**Syntax:** The basic syntax of AND operator with WHERE clause is as follows:

**SELECT column1, column2, columnN FROM table\_name**

**WHERE [condition1] AND [condition2]...AND [conditionN];**

We can combine N number of conditions using AND operator. For an action to be taken by the SQL statement, whether it be a transaction or query, all conditions separated by the AND must be TRUE.

Example:

Consider the CUSTOMERS table having the following records:

+----+----------+-----+-----------+----------+

| ID | NAME | AGE | ADDRESS | SALARY |

+----+----------+-----+-----------+----------+

| 1 | Ramesh | 32 | Ahmedabad | 2000.00 |

| 2 | Khilan | 25 | Delhi | 1500.00 |

| 3 | kaushik | 23 | Kota | 2000.00 |

| 4 | Chaitali | 25 | Mumbai | 6500.00 |

| 5 | Hardik | 27 | Bhopal | 8500.00 |

| 6 | Komal | 22 | MP | 4500.00 |

| 7 | Muffy | 24 | Indore | 10000.00 |

+----+----------+-----+-----------+----------+

Following is an example, which would fetch ID, Name and Salary fields from the CUSTOMERS table where salary is greater than 2000 AND age is less tan 25 years:

**SQL> SELECT ID, NAME, SALARY FROM CUSTOMERS WHERE SALARY > 2000 AND age < 25;**

This would produce the following result:

+----+-------+----------+

| ID | NAME | SALARY |

+----+-------+----------+

| 6 | Komal | 4500.00 |

| 7 | Muffy | 10000.00 |

+----+-------+----------+

**The OR Operator:**

The OR operator is used to combine multiple conditions in an SQL statement's WHERE clause.

**Syntax:** The basic syntax of OR operator with WHERE clause is as follows:

**SELECT column1, column2, columnN FROM table\_name**

**WHERE [condition1] OR [condition2]...OR [conditionN]**

We can combine N number of conditions using OR operator. For an action to be taken by the SQL statement, whether it be a transaction or query, only any ONE of the conditions separated by the OR must be TRUE.

Example: Consider the CUSTOMERS table having the following records:

+----+----------+-----+-----------+----------+

| ID | NAME | AGE | ADDRESS | SALARY |

+----+----------+-----+-----------+----------+

| 1 | Ramesh | 32 | Ahmedabad | 2000.00 |

| 2 | Khilan | 25 | Delhi | 1500.00 |

| 3 | kaushik | 23 | Kota | 2000.00 |

| 4 | Chaitali | 25 | Mumbai | 6500.00 |

| 5 | Hardik | 27 | Bhopal | 8500.00 |

| 6 | Komal | 22 | MP | 4500.00 |

| 7 | Muffy | 24 | Indore | 10000.00 |

+----+----------+-----+-----------+----------+

Following is an example, which would fetch ID, Name and Salary fields from the CUSTOMERS table where salary is greater than 2000 OR age is less tan 25 years:

**SQL> SELECT ID, NAME, SALARY FROM CUSTOMERS WHERE SALARY > 2000 OR age < 25;**

This would produce the following result:

+----+----------+----------+

| ID | NAME | SALARY |

+----+----------+----------+

| 3 | kaushik | 2000.00 |

| 4 | Chaitali | 6500.00 |

| 5 | Hardik | 8500.00 |

| 6 | Komal | 4500.00 |

| 7 | Muffy | 10000.00 |

+----+----------+----------+

**SQL LIKE Clause:** The SQL **LIKE** clause is used to compare a value to similar values using wildcard operators. There are two wildcards used in conjunction with the LIKE operator:

* The percent sign (%)
* The underscore (\_)

The percent sign represents zero, one, or multiple characters. The underscore represents a single number or character. The symbols can be used in combinations.

**Syntax:**

The basic syntax of % and \_ is as follows:

**SELECT FROM table\_name WHERE column LIKE 'XXXX%'**

**or**

**SELECT FROM table\_name WHERE column LIKE '%XXXX%'**

**or**

**SELECT FROM table\_name WHERE column LIKE 'XXXX\_'**

**or**

**SELECT FROM table\_name WHERE column LIKE '\_XXXX'**

**or**

**SELECT FROM table\_name WHERE column LIKE '\_XXXX\_'**

We can combine N number of conditions using AND or OR operators. Here, XXXX could be any numeric or string value.

Example:

Here are number of examples showing WHERE part having different LIKE clause with '%' and '\_' operators:

|  |  |
| --- | --- |
| **Statement** | **Description** |
| WHERE SALARY LIKE '200%' | Finds any values that start with 200 |
| WHERE SALARY LIKE '%200%' | Finds any values that have 200 in any position |
| WHERE SALARY LIKE '\_00%' | Finds any values that have 00 in the second and third positions |
| WHERE SALARY LIKE '2\_%\_%' | Finds any values that start with 2 and are at least 3 characters in length |
| WHERE SALARY LIKE '%2' | Finds any values that end with 2 |
| WHERE SALARY LIKE '\_2%3' | Finds any values that have a 2 in the second position and end with a 3 |
| WHERE SALARY LIKE '2\_\_\_3' | Finds any values in a five-digit number that start with 2 and end with 3 |

**SQL ORDER BY Clause:** The SQL **ORDER BY** clause is used to sort the data in ascending or descending order, based on one or more columns. Some database sorts query results in ascending order by default.

**Syntax:** The basic syntax of ORDER BY clause is as follows:

**SELECT column-list FROM table\_name**

**[WHERE condition] [ORDER BY column1, column2, .. columnN] [ASC | DESC];**

We can use more than one column in the ORDER BY clause. Make sure whatever column we are using to sort, that column should be in column-list.

Example:

Consider the CUSTOMERS table having the following records:

+----+----------+-----+-----------+----------+

| ID | NAME | AGE | ADDRESS | SALARY |

+----+----------+-----+-----------+----------+

| 1 | Ramesh | 32 | Ahmedabad | 2000.00 |

| 2 | Khilan | 25 | Delhi | 1500.00 |

| 3 | kaushik | 23 | Kota | 2000.00 |

| 4 | Chaitali | 25 | Mumbai | 6500.00 |

| 5 | Hardik | 27 | Bhopal | 8500.00 |

| 6 | Komal | 22 | MP | 4500.00 |

| 7 | Muffy | 24 | Indore | 10000.00 |

+----+----------+-----+-----------+----------+

Following is an example, which would sort the result in ascending order by NAME and SALARY:

**SQL> SELECT \* FROM CUSTOMERS ORDER BY NAME, SALARY;**

This would produce the following result:

| ID | NAME | AGE | ADDRESS | SALARY |

+----+----------+-----+-----------+----------+

| 4 | Chaitali | 25 | Mumbai | 6500.00 |

| 5 | Hardik | 27 | Bhopal | 8500.00 |

| 3 | kaushik | 23 | Kota | 2000.00 |

| 2 | Khilan | 25 | Delhi | 1500.00 |

| 6 | Komal | 22 | MP | 4500.00 |

| 7 | Muffy | 24 | Indore | 10000.00 |

| 1 | Ramesh | 32 | Ahmedabad | 2000.00 |

+----+----------+-----+-----------+----------+

Following is an example, which would sort the result in descending order by NAME:

**SQL> SELECT \* FROM CUSTOMERS ORDER BY NAME DESC;**

This would produce the following result:

+----+----------+-----+-----------+----------+

| ID | NAME | AGE | ADDRESS | SALARY |

+----+----------+-----+-----------+----------+

| 1 | Ramesh | 32 | Ahmedabad | 2000.00 |

| 7 | Muffy | 24 | Indore | 10000.00 |

| 6 | Komal | 22 | MP | 4500.00 |

| 2 | Khilan | 25 | Delhi | 1500.00 |

| 3 | kaushik | 23 | Kota | 2000.00 |

| 5 | Hardik | 27 | Bhopal | 8500.00 |

| 4 | Chaitali | 25 | Mumbai | 6500.00 |

**SQL Group By:** The SQL **GROUP BY** clause is used in collaboration with the SELECT statement to arrange identical data into groups. The GROUP BY clause follows the WHERE clause in a SELECT statement and precedes the ORDER BY clause.

**Syntax**: The basic syntax of GROUP BY clause is given below. The GROUP BY clause must follow the conditions in the WHERE clause and must precede the ORDER BY clause if one is used.

**SELECT column1, column2,… FROM table\_name WHERE conditions GROUP BY column1, column2 ORDER BY column1, column2;**

Example:

Consider the CUSTOMERS table having the following records:

+----+----------+-----+-----------+----------+

| ID | NAME | AGE | ADDRESS | SALARY |

+----+----------+-----+-----------+----------+

| 1 | Ramesh | 32 | Ahmedabad | 2000.00 |

| 2 | Khilan | 25 | Delhi | 1500.00 |

| 3 | kaushik | 23 | Kota | 2000.00 |

| 4 | Chaitali | 25 | Mumbai | 6500.00 |

| 5 | Hardik | 27 | Bhopal | 8500.00 |

| 6 | Komal | 22 | MP | 4500.00 |

| 7 | Muffy | 24 | Indore | 10000.00 |

+----+----------+-----+-----------+----------+

If we want to know the total amount of salary on each customer, then GROUP BY query would be as follows:

**SQL> SELECT NAME, SUM(SALARY) FROM CUSTOMERS GROUP BY NAME;**

This would produce the following result:

+----------+-------------+

| NAME | SUM(SALARY) |

+----------+-------------+

| Chaitali | 6500.00 |

| Hardik | 8500.00 |

| kaushik | 2000.00 |

| Khilan | 1500.00 |

| Komal | 4500.00 |

| Muffy | 10000.00 |

| Ramesh | 2000.00 |

+----------+-------------+

Now, let us have following table where CUSTOMERS table has the following records with duplicate names:

+----+----------+-----+-----------+----------+

| ID | NAME | AGE | ADDRESS | SALARY |

+----+----------+-----+-----------+----------+

| 1 | Ramesh | 32 | Ahmedabad | 2000.00 |

| 2 | Ramesh | 25 | Delhi | 1500.00 |

| 3 | kaushik | 23 | Kota | 2000.00 |

| 4 | kaushik | 25 | Mumbai | 6500.00 |

| 5 | Hardik | 27 | Bhopal | 8500.00 |

| 6 | Komal | 22 | MP | 4500.00 |

| 7 | Muffy | 24 | Indore | 10000.00 |

+----+----------+-----+-----------+----------+

Now again, if we want to know the total amount of salary on each customer, then GROUP BY query would be as follows:

**SQL> SELECT NAME, SUM(SALARY) FROM CUSTOMERS GROUP BY NAME;**

This would produce the following result:

+---------+-------------+

| NAME | SUM(SALARY) |

+---------+-------------+

| Hardik | 8500.00 |

| kaushik | 8500.00 |

| Komal | 4500.00 |

| Muffy | 10000.00 |

| Ramesh | 3500.00 |

+---------+-------------+

**SQL Distinct Keyword**

The SQL **DISTINCT** keyword is used in conjunction with SELECT statement to eliminate all the duplicate records and fetching only unique records.

There may be a situation when we have multiple duplicate records in a table. While fetching such records, it makes more sense to fetch only unique records instead of fetching duplicate records.

Syntax:

The basic syntax of DISTINCT keyword to eliminate duplicate records is as follows:

**SELECT DISTINCT column1, column2,....., columnN FROM table\_name WHERE condition;**

Example:

Consider the CUSTOMERS table having the following records:

+----+----------+-----+-----------+----------+

| ID | NAME | AGE | ADDRESS | SALARY |

+----+----------+-----+-----------+----------+

| 1 | Ramesh | 32 | Ahmedabad | 2000.00 |

| 2 | Khilan | 25 | Delhi | 1500.00 |

| 3 | kaushik | 23 | Kota | 2000.00 |

| 4 | Chaitali | 25 | Mumbai | 6500.00 |

| 5 | Hardik | 27 | Bhopal | 8500.00 |

| 6 | Komal | 22 | MP | 4500.00 |

| 7 | Muffy | 24 | Indore | 10000.00 |

+----+----------+-----+-----------+----------+

First, let us see how the following SELECT query returns duplicate salary records:

**SQL> SELECT SALARY FROM CUSTOMERS ORDER BY SALARY;**

This would produce the following result where salary 2000 is coming twice which is a duplicate record from the original table.

+----------+

| SALARY |

+----------+

| 1500.00 |

| 2000.00 |

| 2000.00 |

| 4500.00 |

| 6500.00 |

| 8500.00 |

| 10000.00 |

+----------+

Now, let us use DISTINCT keyword with the above SELECT query and see the result:

**SQL> SELECT DISTINCT SALARY FROM CUSTOMERS ORDER BY SALARY;**

This would produce the following result where we do not have any duplicate entry:

+----------+

| SALARY |

+----------+

| 1500.00 |

| 2000.00 |

| 4500.00 |

| 6500.00 |

| 8500.00 |

| 10000.00 |

+----------+

**SQL SORTING Results**

The SQL **ORDER BY** clause is used to sort the data in ascending or descending order, based on one or more columns. Some databases sort query results in ascending order by default.

**Syntax:**

The basic syntax of ORDER BY clause which would be used to sort result in ascending or descending order is as follows:

**SELECT column-list FROM table\_name [WHERE condition]**

**[ORDER BY column1, column2, .. columnN] [ASC | DESC];**

We can use more than one column in the ORDER BY clause. Make sure whatever column we are using to sort, that column should be in column-list.

Example:

Consider the CUSTOMERS table having the following records:

+----+----------+-----+-----------+----------+

| ID | NAME | AGE | ADDRESS | SALARY |

+----+----------+-----+-----------+----------+

| 1 | Ramesh | 32 | Ahmedabad | 2000.00 |

| 2 | Khilan | 25 | Delhi | 1500.00 |

| 3 | kaushik | 23 | Kota | 2000.00 |

| 4 | Chaitali | 25 | Mumbai | 6500.00 |

| 5 | Hardik | 27 | Bhopal | 8500.00 |

| 6 | Komal | 22 | MP | 4500.00 |

| 7 | Muffy | 24 | Indore | 10000.00 |

+----+----------+-----+-----------+----------+

Following is an example, which would sort the result in ascending order by NAME and SALARY:

**SQL> SELECT \* FROM CUSTOMERS ORDER BY NAME, SALARY;**

This would produce the following result:

+----+----------+-----+-----------+----------+

| ID | NAME | AGE | ADDRESS | SALARY |

+----+----------+-----+-----------+----------+

| 4 | Chaitali | 25 | Mumbai | 6500.00 |

| 5 | Hardik | 27 | Bhopal | 8500.00 |

| 3 | kaushik | 23 | Kota | 2000.00 |

| 2 | Khilan | 25 | Delhi | 1500.00 |

| 6 | Komal | 22 | MP | 4500.00 |

| 7 | Muffy | 24 | Indore | 10000.00 |

| 1 | Ramesh | 32 | Ahmedabad | 2000.00 |

+----+----------+-----+-----------+----------+

Following is an example, which would sort the result in descending order by NAME:

**SQL> SELECT \* FROM CUSTOMERS ORDER BY NAME DESC;**

This would produce the following result:

+----+----------+-----+-----------+----------+

| ID | NAME | AGE | ADDRESS | SALARY |

+----+----------+-----+-----------+----------+

| 1 | Ramesh | 32 | Ahmedabad | 2000.00 |

| 7 | Muffy | 24 | Indore | 10000.00 |

| 6 | Komal | 22 | MP | 4500.00 |

| 2 | Khilan | 25 | Delhi | 1500.00 |

| 3 | kaushik | 23 | Kota | 2000.00 |

| 5 | Hardik | 27 | Bhopal | 8500.00 |

| 4 | Chaitali | 25 | Mumbai | 6500.00 |

+----+----------+-----+-----------+----------+

**SQL Joins**

The SQL **Joins** clause is used to combine records from two or more tables in a database. A JOIN is a means for combining fields from two tables by using values common to each.

Consider the following two tables,

**(a) CUSTOMERS table is as follows:**

**+----+----------+-----+-----------+----------+**

**| ID | NAME | AGE | ADDRESS | SALARY |**

**+----+----------+-----+-----------+----------+**

| 1 | Ramesh | 32 | Ahmedabad | 2000.00 |

| 2 | Khilan | 25 | Delhi | 1500.00 |

| 3 | kaushik | 23 | Kota | 2000.00 |

| 4 | Chaitali | 25 | Mumbai | 6500.00 |

| 5 | Hardik | 27 | Bhopal | 8500.00 |

| 6 | Komal | 22 | MP | 4500.00 |

| 7 | Muffy | 24 | Indore | 10000.00 |

**+----+----------+-----+-----------+----------+**

**(b) Another table is ORDERS as follows:**

**+-----+---------------------+-------------+--------+**

**|OID | DATE | CUSTOMER\_ID | AMOUNT |**

**+-----+---------------------+-------------+--------+**

**|** 102 | 2009-10-08 00:00:00 | 3 | 3000 |

| 100 | 2009-10-08 00:00:00 | 3 | 1500 |

| 101 | 2009-11-20 00:00:00 | 2 | 1560 |

| 103 | 2008-05-20 00:00:00 | 4 | 2060 |

Now, let us join these two tables in our SELECT statement as follows:

SQL> SELECT ID, NAME, AGE, AMOUNT

FROM CUSTOMERS, ORDERS

WHERE CUSTOMERS.ID = ORDERS.CUSTOMER\_ID;

This would produce the following result:

+----+----------+-----+--------+

| ID | NAME | AGE | AMOUNT |

+----+----------+-----+--------+

| 3 | kaushik | 23 | 3000 |

| 3 | kaushik | 23 | 1500 |

| 2 | Khilan | 25 | 1560 |

| 4 | Chaitali | 25 | 2060 |

+----+----------+-----+--------+

Here, it is noticeable that the join is performed in the WHERE clause. Several operators can be used to join tables, such as =, <, >, <>, <=, >=, !=, BETWEEN, LIKE, and NOT; they can all be used to join tables. However, the most common operator is the equal symbol.

***SQL Join Types:***

There are different types of joins available in SQL:

* **INNER JOIN**: returns rows when there is a match in both tables.
* **LEFT JOIN**: returns all rows from the left table, even if there are no matches in the right table.
* **RIGHT JOIN**: returns all rows from the right table, even if there are no matches in the left table.
* **FULL JOIN**: returns rows when there is a match in one of the tables.
* **SELF JOIN**: is used to join a table to itself as if the table were two tables, temporarily renaming at least one table in the SQL statement.
* **CARTESIAN JOIN**: returns the Cartesian product of the sets of records from the two or more joined tables.

***INNER JOIN***

The most frequently used and important of the joins is the **INNER JOIN**. They are also referred to as an EQUIJOIN. The INNER JOIN creates a new result table by combining column values of two tables (table1 and table2) based upon the join-predicate. The query compares each row of table1 with each row of table2 to find all pairs of rows which satisfy the join-predicate. When the join-predicate is satisfied, column values for each matched pair of rows of A and B are combined into a result row.

**Syntax:**

The basic syntax of **INNER JOIN** is as follows:

**SELECT table1.column1, table2.column2... FROM table1**

**INNER JOIN table2**

**ON table1.common\_filed = table2.common\_field;**

Now, let us join these two tables using **INNER JOIN** as follows:

SQL> SELECT ID, NAME, AMOUNT, DATE FROM CUSTOMERS INNER JOIN ORDERS

ON CUSTOMERS.ID = ORDERS.CUSTOMER\_ID;

This would produce the following result:

+----+----------+--------+---------------------+

| ID | NAME | AMOUNT | DATE |

+----+----------+--------+---------------------+

| 3 | kaushik | 3000 | 2009-10-08 00:00:00 |

| 3 | kaushik | 1500 | 2009-10-08 00:00:00 |

| 2 | Khilan | 1560 | 2009-11-20 00:00:00 |

| 4 | Chaitali | 2060 | 2008-05-20 00:00:00 |

+----+----------+--------+---------------------+

***LEFT JOIN***

The SQL **LEFT JOIN** returns all rows from the left table, even if there are no matches in the right table. This means that if the ON clause matches 0 (zero) records in right table, the join will still return a row in the result, but with NULL in each column from right table.

This means that a left join returns all the values from the left table, plus matched values from the right table or NULL in case of no matching join predicate.

Syntax: The basic syntax of **LEFT JOIN** is as follows:

**SELECT table1.column1, table2.column2...**

**FROM table1**

**LEFT JOIN table2**

**ON table1.common\_filed = table2.common\_field;**

Here given condition could be any given expression based on our requirement.

Example:

Now, let us join these two tables using **LEFT JOIN** as follows:

**SQL> SELECT ID, NAME, AMOUNT, DATE FROM CUSTOMERS LEFT JOIN ORDERS ON CUSTOMERS.ID = ORDERS.CUSTOMER\_ID;**

This would produce the following result:

+----+----------+--------+---------------------+

| ID | NAME | AMOUNT | DATE |

+----+----------+--------+---------------------+

| 1 | Ramesh | NULL | NULL |

| 2 | Khilan | 1560 | 2009-11-20 00:00:00 |

| 3 | kaushik | 3000 | 2009-10-08 00:00:00 |

| 3 | kaushik | 1500 | 2009-10-08 00:00:00 |

| 4 | Chaitali | 2060 | 2008-05-20 00:00:00 |

| 5 | Hardik | NULL | NULL |

| 6 | Komal | NULL | NULL |

| 7 | Muffy | NULL | NULL |

+----+----------+--------+---------------------+

***RIGHT JOIN***

The SQL **RIGHT JOIN** returns all rows from the right table, even if there are no matches in the left table. This means that if the ON clause matches 0 (zero) records in left table, the join will still return a row in the result, but with NULL in each column from left table.

This means that a right join returns all the values from the right table, plus matched values from the left table or NULL in case of no matching join predicate.

Syntax: The basic syntax of **RIGHT JOIN** is as follows:

**SELECT table1.column1, table2.column2...**

**FROM table1**

**RIGHT JOIN table2**

**ON table1.common\_filed = table2.common\_field;**

Now, let us join these two tables using **RIGHT JOIN** as follows:

**SQL> SELECT ID, NAME, AMOUNT, DATE FROM CUSTOMERS RIGHT JOIN ORDERS ON CUSTOMERS.ID = ORDERS.CUSTOMER\_ID;**

This would produce the following result:

+------+----------+--------+---------------------+

| ID | NAME | AMOUNT | DATE |

+------+----------+--------+---------------------+

| 3 | kaushik | 3000 | 2009-10-08 00:00:00 |

| 3 | kaushik | 1500 | 2009-10-08 00:00:00 |

| 2 | Khilan | 1560 | 2009-11-20 00:00:00 |

| 4 | Chaitali | 2060 | 2008-05-20 00:00:00 |

+------+----------+--------+---------------------+

***FULL JOIN***

The SQL **FULL JOIN** combines the results of both left and right outer joins. The joined table will contain all records from both tables, and fill in NULLs for missing matches on either side.

Syntax: The basic syntax of **FULL JOIN** is as follows:

**SELECT table1.column1, table2.column2...**

**FROM table1**

**FULL JOIN table2**

**ON table1.common\_filed = table2.common\_field;**

Here given condition could be any given expression based on our requirement.

Now, let us join these two tables using **FULL JOIN** as follows:

**SQL> SELECT ID, NAME, AMOUNT, DATE FROM CUSTOMERS FULL JOIN ORDERS ON CUSTOMERS.ID = ORDERS.CUSTOMER\_ID;**

This would produce the following result:

+------+----------+--------+---------------------+

| ID | NAME | AMOUNT | DATE |

+------+----------+--------+---------------------+

| 1 | Ramesh | NULL | NULL |

| 2 | Khilan | 1560 | 2009-11-20 00:00:00 |

| 3 | kaushik | 3000 | 2009-10-08 00:00:00 |

| 3 | kaushik | 1500 | 2009-10-08 00:00:00 |

| 4 | Chaitali | 2060 | 2008-05-20 00:00:00 |

| 5 | Hardik | NULL | NULL |

| 6 | Komal | NULL | NULL |

| 7 | Muffy | NULL | NULL |

| 3 | kaushik | 3000 | 2009-10-08 00:00:00 |

| 3 | kaushik | 1500 | 2009-10-08 00:00:00 |

| 2 | Khilan | 1560 | 2009-11-20 00:00:00 |

| 4 | Chaitali | 2060 | 2008-05-20 00:00:00 |

+------+----------+--------+---------------------+

If our Database does not support FULL JOIN like MySQL does not support FULL JOIN, then we can use **UNION ALL** clause to combine two JOINS as follows:

SQL> SELECT ID, NAME, AMOUNT, DATE

FROM CUSTOMERS

LEFT JOIN ORDERS ON CUSTOMERS.ID = ORDERS.CUSTOMER\_ID

UNION ALL

SELECT ID, NAME, AMOUNT, DATE

FROM CUSTOMERS

RIGHT JOIN ORDERS

ON CUSTOMERS.ID = ORDERS.CUSTOMER\_ID

***SELF JOIN***

The SQL **SELF JOIN** is used to join a table to itself as if the table were two tables, temporarily renaming at least one table in the SQL statement.

Syntax:

The basic syntax of **SELF JOIN** is as follows:

**SELECT a.column\_name, b.column\_name...**

**FROM table1 a, table1 b**

**WHERE a.common\_filed = b.common\_field;**

Here, WHERE clause could be any given expression based on our requirement.

Now, let us join this table using **SELF JOIN** as follows:

**SQL> SELECT a.ID, b.NAME, a.SALARY FROM CUSTOMERS a, CUSTOMERS b**

**WHERE a.SALARY < b.SALARY;**

This would produce the following result:

+----+----------+---------+

| ID | NAME | SALARY |

+----+----------+---------+

| 2 | Ramesh | 1500.00 |

| 2 | kaushik | 1500.00 |

| 1 | Chaitali | 2000.00 |

| 2 | Chaitali | 1500.00 |

| 3 | Chaitali | 2000.00 |

| 6 | Chaitali | 4500.00 |

| 1 | Hardik | 2000.00 |

| 2 | Hardik | 1500.00 |

| 3 | Hardik | 2000.00 |

| 4 | Hardik | 6500.00 |

| 6 | Hardik | 4500.00 |

| 1 | Komal | 2000.00 |

| 2 | Komal | 1500.00 |

| 3 | Komal | 2000.00 |

| 1 | Muffy | 2000.00 |

| 2 | Muffy | 1500.00 |

| 3 | Muffy | 2000.00 |

| 4 | Muffy | 6500.00 |

| 5 | Muffy | 8500.00 |

| 6 | Muffy | 4500.00 |

+----+----------+---------+

***CARTESIAN JOIN***

The **CARTESIAN JOIN** or **CROSS JOIN** returns the cartesian product of the sets of records from the two or more joined tables. Thus, it equates to an inner join where the join-condition always evaluates to True or where the join-condition is absent from the statement.

Syntax:

The basic syntax of **INNER JOIN** is as follows:

**SELECT table1.column1, table2.column2...**

**FROM table1, table2 [, table3 ]**

Now, let us join these two tables using **INNER JOIN** as follows:

**SQL> SELECT ID, NAME, AMOUNT, DATE FROM CUSTOMERS, ORDERS;**

This would produce the following result:

+----+----------+--------+---------------------+

| ID | NAME | AMOUNT | DATE |

+----+----------+--------+---------------------+

| 1 | Ramesh | 3000 | 2009-10-08 00:00:00 |

| 1 | Ramesh | 1500 | 2009-10-08 00:00:00 |

| 1 | Ramesh | 1560 | 2009-11-20 00:00:00 |

| 1 | Ramesh | 2060 | 2008-05-20 00:00:00 |

| 2 | Khilan | 3000 | 2009-10-08 00:00:00 |

| 2 | Khilan | 1500 | 2009-10-08 00:00:00 |

| 2 | Khilan | 1560 | 2009-11-20 00:00:00 |

| 2 | Khilan | 2060 | 2008-05-20 00:00:00 |

| 3 | kaushik | 3000 | 2009-10-08 00:00:00 |

| 3 | kaushik | 1500 | 2009-10-08 00:00:00 |

| 3 | kaushik | 1560 | 2009-11-20 00:00:00 |

| 3 | kaushik | 2060 | 2008-05-20 00:00:00 |

| 4 | Chaitali | 3000 | 2009-10-08 00:00:00 |

| 4 | Chaitali | 1500 | 2009-10-08 00:00:00 |

| 4 | Chaitali | 1560 | 2009-11-20 00:00:00 |

| 4 | Chaitali | 2060 | 2008-05-20 00:00:00 |

| 5 | Hardik | 3000 | 2009-10-08 00:00:00 |

| 5 | Hardik | 1500 | 2009-10-08 00:00:00 |

| 5 | Hardik | 1560 | 2009-11-20 00:00:00 |

| 5 | Hardik | 2060 | 2008-05-20 00:00:00 |

| 6 | Komal | 3000 | 2009-10-08 00:00:00 |

| 6 | Komal | 1500 | 2009-10-08 00:00:00 |

| 6 | Komal | 1560 | 2009-11-20 00:00:00 |

| 6 | Komal | 2060 | 2008-05-20 00:00:00 |

| 7 | Muffy | 3000 | 2009-10-08 00:00:00 |

| 7 | Muffy | 1500 | 2009-10-08 00:00:00 |

| 7 | Muffy | 1560 | 2009-11-20 00:00:00 |

| 7 | Muffy | 2060 | 2008-05-20 00:00:00 |

+----+----------+--------+---------------------+

**SQL Unions Clause**

The SQL **UNION** clause/operator is used to combine the results of two or more SELECT statements without returning any duplicate rows. To use UNION, each SELECT must have the same number of columns selected, the same number of column expressions, the same data type, and have them in the same order, but they do not have to be the same length.

Syntax:

The basic syntax of **UNION** is as follows:

**SELECT column1 [, column2 ] FROM table1 [, table2 ]**

**[WHERE condition]**

**UNION**

**SELECT column1 [, column2 ] FROM table1 [, table2 ]**

**[WHERE condition]**

Here given condition could be any given expression based on our requirement.

Now, let us join these two tables in our SELECT statement as follows:

**SQL> SELECT ID, NAME, AMOUNT, DATE FROM CUSTOMERS**

**LEFT JOIN ORDERS ON CUSTOMERS.ID = ORDERS.CUSTOMER\_ID**

**UNION**

**SELECT ID, NAME, AMOUNT, DATE FROM CUSTOMERS**

**RIGHT JOIN ORDERS**

**ON CUSTOMERS.ID = ORDERS.CUSTOMER\_ID;**

This would produce the following result:

+------+----------+--------+---------------------+

| ID | NAME | AMOUNT | DATE |

+------+----------+--------+---------------------+

| 1 | Ramesh | NULL | NULL |

| 2 | Khilan | 1560 | 2009-11-20 00:00:00 |

| 3 | kaushik | 3000 | 2009-10-08 00:00:00 |

| 3 | kaushik | 1500 | 2009-10-08 00:00:00 |

| 4 | Chaitali | 2060 | 2008-05-20 00:00:00 |

| 5 | Hardik | NULL | NULL |

| 6 | Komal | NULL | NULL |

| 7 | Muffy | NULL | NULL |

+------+----------+--------+---------------------+

**The UNION ALL Clause:**

The UNION ALL operator is used to combine the results of two SELECT statements including duplicate rows. The same rules that apply to UNION apply to the UNION ALL operator.

Syntax:

The basic syntax of **UNION ALL** is as follows:

**SELECT column1 [, column2 ] FROM table1 [, table2 ]**

**[WHERE condition]**

**UNION ALL**

**SELECT column1 [, column2 ] FROM table1 [, table2 ]**

**[WHERE condition]**

Here given condition could be any given expression based on our requirement.

Example:

Now, let us join these two tables in our SELECT statement as follows:

**SQL> SELECT ID, NAME, AMOUNT, DATE FROM CUSTOMERS**

**LEFT JOIN ORDERS ON CUSTOMERS.ID = ORDERS.CUSTOMER\_ID**

**UNION ALL**

**SELECT ID, NAME, AMOUNT, DATE FROM CUSTOMERS**

**RIGHT JOIN ORDERS ON CUSTOMERS.ID = ORDERS.CUSTOMER\_ID;**

This would produce the following result:

+------+----------+--------+---------------------+

| ID | NAME | AMOUNT | DATE |

+------+----------+--------+---------------------+

| 1 | Ramesh | NULL | NULL |

| 2 | Khilan | 1560 | 2009-11-20 00:00:00 |

| 3 | kaushik | 3000 | 2009-10-08 00:00:00 |

| 3 | kaushik | 1500 | 2009-10-08 00:00:00 |

| 4 | Chaitali | 2060 | 2008-05-20 00:00:00 |

| 5 | Hardik | NULL | NULL |

| 6 | Komal | NULL | NULL |

| 7 | Muffy | NULL | NULL |

| 3 | kaushik | 3000 | 2009-10-08 00:00:00 |

| 3 | kaushik | 1500 | 2009-10-08 00:00:00 |

| 2 | Khilan | 1560 | 2009-11-20 00:00:00 |

| 4 | Chaitali | 2060 | 2008-05-20 00:00:00 |

+------+----------+--------+---------------------+

**There are two other clauses (i.e., operators), which are very similar to UNION clause:**

* **SQL INTERSECT Clause: is used to combine two SELECT statements, but returns rows only from the first SELECT statement that are identical to a row in the second SELECT statement.**
* **SQL EXCEPT Clause : combines two SELECT statements and returns rows from the first SELECT statement that are not returned by the second SELECT statement.**

**INTERSECT Clause**

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

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

Syntax:

The basic syntax of **INTERSECT** is as follows:

SELECT column1 [, column2 ] FROM table1 [, table2 ] [WHERE condition]

INTERSECT

SELECT column1 [, column2 ] FROM table1 [, table2 ] [WHERE condition]

Here given condition could be any given expression based on our requirement.

Example:

Consider the following two tables, (a) CUSTOMERS table is as follows:

+----+----------+-----+-----------+----------+

| ID | NAME | AGE | ADDRESS | SALARY |

+----+----------+-----+-----------+----------+

| 1 | Ramesh | 32 | Ahmedabad | 2000.00 |

| 2 | Khilan | 25 | Delhi | 1500.00 |

| 3 | kaushik | 23 | Kota | 2000.00 |

| 4 | Chaitali | 25 | Mumbai | 6500.00 |

| 5 | Hardik | 27 | Bhopal | 8500.00 |

| 6 | Komal | 22 | MP | 4500.00 |

| 7 | Muffy | 24 | Indore | 10000.00 |

+----+----------+-----+-----------+----------+

(b) Another table is ORDERS as follows:

+-----+---------------------+-------------+--------+

|OID | DATE | CUSTOMER\_ID | AMOUNT |

+-----+---------------------+-------------+--------+

| 102 | 2009-10-08 00:00:00 | 3 | 3000 |

| 100 | 2009-10-08 00:00:00 | 3 | 1500 |

| 101 | 2009-11-20 00:00:00 | 2 | 1560 |

| 103 | 2008-05-20 00:00:00 | 4 | 2060 |

+-----+---------------------+-------------+--------+

Now, let us join these two tables in our SELECT statement as follows:

SQL> SELECT ID, NAME, AMOUNT, DATE

FROM CUSTOMERS

LEFT JOIN ORDERS

ON CUSTOMERS.ID = ORDERS.CUSTOMER\_ID

INTERSECT

SELECT ID, NAME, AMOUNT, DATE

FROM CUSTOMERS

RIGHT JOIN ORDERS

ON CUSTOMERS.ID = ORDERS.CUSTOMER\_ID;

This would produce the following result:

+------+---------+--------+---------------------+

| ID | NAME | AMOUNT | DATE |

+------+---------+--------+---------------------+

| 3 | kaushik | 3000 | 2009-10-08 00:00:00 |

| 3 | kaushik | 1500 | 2009-10-08 00:00:00 |

| 2 | Ramesh | 1560 | 2009-11-20 00:00:00 |

| 4 | kaushik | 2060 | 2008-05-20 00:00:00 |

+------+---------+--------+---------------------+

EXCEPT Clause

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

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

Syntax:

The basic syntax of **EXCEPT** is as follows:

SELECT column1 [, column2 ] FROM table1 [, table2 ] [WHERE condition]

EXCEPT

SELECT column1 [, column2 ] FROM table1 [, table2 ] [WHERE condition]

Here given condition could be any given expression based on our requirement.

Example:

Consider the following two tables, (a) CUSTOMERS table is as follows:

+----+----------+-----+-----------+----------+

| ID | NAME | AGE | ADDRESS | SALARY |

+----+----------+-----+-----------+----------+

| 1 | Ramesh | 32 | Ahmedabad | 2000.00 |

| 2 | Khilan | 25 | Delhi | 1500.00 |

| 3 | kaushik | 23 | Kota | 2000.00 |

| 4 | Chaitali | 25 | Mumbai | 6500.00 |

| 5 | Hardik | 27 | Bhopal | 8500.00 |

| 6 | Komal | 22 | MP | 4500.00 |

| 7 | Muffy | 24 | Indore | 10000.00 |

+----+----------+-----+-----------+----------+

(b) Another table is ORDERS as follows:

+-----+---------------------+-------------+--------+

|OID | DATE | CUSTOMER\_ID | AMOUNT |

+-----+---------------------+-------------+--------+

| 102 | 2009-10-08 00:00:00 | 3 | 3000 |

| 100 | 2009-10-08 00:00:00 | 3 | 1500 |

| 101 | 2009-11-20 00:00:00 | 2 | 1560 |

| 103 | 2008-05-20 00:00:00 | 4 | 2060 |

+-----+---------------------+-------------+--------+

Now, let us join these two tables in our SELECT statement as follows:

SQL> SELECT ID, NAME, AMOUNT, DATE FROM CUSTOMERS

LEFT JOIN ORDERS ON CUSTOMERS.ID = ORDERS.CUSTOMER\_ID

EXCEPT

SELECT ID, NAME, AMOUNT, DATE FROM CUSTOMERS

RIGHT JOIN ORDERS ON CUSTOMERS.ID = ORDERS.CUSTOMER\_ID;

This would produce the following result:

+----+---------+--------+---------------------+

| ID | NAME | AMOUNT | DATE |

+----+---------+--------+---------------------+

| 1 | Ramesh | NULL | NULL |

| 5 | Hardik | NULL | NULL |

| 6 | Komal | NULL | NULL |

| 7 | Muffy | NULL | NULL |

+----+---------+--------+---------------------+

**SQL NULL Values**

The SQL **NULL** is the term used to represent a missing value. A NULL value in a table is a value in a field that appears to be blank.

A field with a NULL value is a field with no value. It is very important to understand that a NULL value is different than a zero value or a field that contains spaces.

Syntax:

The basic syntax of **NULL** while creating a table:

SQL> CREATE TABLE CUSTOMERS(

ID INT NOT NULL,

NAME VARCHAR (20) NOT NULL,

AGE INT NOT NULL,

ADDRESS CHAR (25) ,

SALARY DECIMAL (18, 2),

PRIMARY KEY (ID)

);

Here, **NOT NULL** signifies that column should always accept an explicit value of the given data type. There are two columns where we did not use NOT NULL, which means these columns could be NULL.

A field with a NULL value is one that has been left blank during record creation.

Example:

The NULL value can cause problems when selecting data, however, because when comparing an unknown value to any other value, the result is always unknown and not included in the final results.

We must use the **IS NULL** or **IS NOT NULL** operators in order to check for a NULL value.

Consider the following table, CUSTOMERS having the following records:

+----+----------+-----+-----------+----------+

| ID | NAME | AGE | ADDRESS | SALARY |

+----+----------+-----+-----------+----------+

| 1 | Ramesh | 32 | Ahmedabad | 2000.00 |

| 2 | Khilan | 25 | Delhi | 1500.00 |

| 3 | kaushik | 23 | Kota | 2000.00 |

| 4 | Chaitali | 25 | Mumbai | 6500.00 |

| 5 | Hardik | 27 | Bhopal | 8500.00 |

| 6 | Komal | 22 | MP | |

| 7 | Muffy | 24 | Indore | |

+----+----------+-----+-----------+----------+

Now, following is the usage of **IS NOT NULL** operator:

SQL> SELECT ID, NAME, AGE, ADDRESS, SALARY

FROM CUSTOMERS

WHERE SALARY IS NOT NULL;

This would produce the following result:

+----+----------+-----+-----------+----------+

| ID | NAME | AGE | ADDRESS | SALARY |

1 | Ramesh | 32 | Ahmedabad | 2000.00 |

| 2 | Khilan | 25 | Delhi | 1500.00 |

| 3 | kaushik | 23 | Kota | 2000.00 |

| 4 | Chaitali | 25 | Mumbai | 6500.00 |

| 5 | Hardik | 27 | Bhopal | 8500.00 |

+----+----------+-----+-----------+----------+

Now, following is the usage of **IS NULL** operator:

SQL> SELECT ID, NAME, AGE, ADDRESS, SALARY

FROM CUSTOMERS

WHERE SALARY IS NULL;

This would produce the following result:

+----+----------+-----+-----------+----------+

| ID | NAME | AGE | ADDRESS | SALARY |

+----+----------+-----+-----------+----------+

| 6 | Komal | 22 | MP | |

| 7 | Muffy | 24 | Indore | |

+----+----------+-----+-----------+----------+

**SQL Alias Syntax**

We can rename a table or a column temporarily by giving another name known as alias. The use of table aliases means to rename a table in a particular SQL statement. The renaming is a temporary change and the actual table name does not change in the database. The column aliases are used to rename a table's columns for the purpose of a particular SQL query.

**Syntax:** The basic syntax of **table** alias is as follows:

**SELECT column1, column2....**

**FROM table\_name AS alias\_name**

**WHERE [condition];**

The basic syntax of **column** alias is as follows:

**SELECT column\_name AS alias\_name**

**FROM table\_name**

**WHERE [condition];**

**Example:**

Consider the following two tables, (a) CUSTOMERS table is as follows:

+----+----------+-----+-----------+----------+

| ID | NAME | AGE | ADDRESS | SALARY |

+----+----------+-----+-----------+----------+

| 1 | Ramesh | 32 | Ahmedabad | 2000.00 |

| 2 | Khilan | 25 | Delhi | 1500.00 |

| 3 | kaushik | 23 | Kota | 2000.00 |

| 4 | Chaitali | 25 | Mumbai | 6500.00 |

| 5 | Hardik | 27 | Bhopal | 8500.00 |

| 6 | Komal | 22 | MP | 4500.00 |

| 7 | Muffy | 24 | Indore | 10000.00 |

+----+----------+-----+-----------+----------+

(b) Another table is ORDERS as follows:

+-----+---------------------+-------------+--------+

|OID | DATE | CUSTOMER\_ID | AMOUNT |

+-----+---------------------+-------------+--------+

| 102 | 2009-10-08 00:00:00 | 3 | 3000 |

| 100 | 2009-10-08 00:00:00 | 3 | 1500 |

| 101 | 2009-11-20 00:00:00 | 2 | 1560 |

| 103 | 2008-05-20 00:00:00 | 4 | 2060 |

+-----+---------------------+-------------+--------+

Now, following is the usage of **table alias**:

**SQL> SELECT C.ID, C.NAME, C.AGE, O.AMOUNT FROM CUSTOMERS AS C, ORDERS AS O WHERE C.ID = O.CUSTOMER\_ID;**

This would produce the following result:

+----+----------+-----+--------+

| ID | NAME | AGE | AMOUNT |

+----+----------+-----+--------+

| 3 | kaushik | 23 | 3000 |

| 3 | kaushik | 23 | 1500 |

| 2 | Khilan | 25 | 1560 |

| 4 | Chaitali | 25 | 2060 |

+----+----------+-----+--------+

Following is the usage of **column alias**:

**SQL> SELECT ID AS CUSTOMER\_ID, NAME AS CUSTOMER\_NAME FROM CUSTOMERS WHERE SALARY IS NOT NULL;**

This would produce the following result:

+-------------+---------------+

| CUSTOMER\_ID | CUSTOMER\_NAME |

+-------------+---------------+

| 1 | Ramesh |

| 2 | Khilan |

| 3 | kaushik |

| 4 | Chaitali |

| 5 | Hardik |

| 6 | Komal |

| 7 | Muffy |

+-------------+---------------+

**SQL -Using Views**

A view is nothing more than a SQL statement that is stored in the database with an associated name. A view is actually a composition of a table in the form of a predefined SQL query.

A view can contain all rows of a table or select rows from a table. A view can be created from one or many tables which depends on the written SQL query to create a view.

Views, which are kind of virtual tables, allow users to do the following:

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

***Creating Views:***

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

The basic CREATE VIEW syntax is as follows:

**CREATE VIEW view\_name AS**

**SELECT column1, column2.....**

**FROM table\_name WHERE condition;**

We can include multiple tables in our SELECT statement in very similar way as we use them in normal SQL SELECT query.

**Example:** Consider the CUSTOMERS table having the following records:

+----+----------+-----+-----------+----------+

| ID | NAME | AGE | ADDRESS | SALARY |

+----+----------+-----+-----------+----------+

| 1 | Ramesh | 32 | Ahmedabad | 2000.00 |

| 2 | Khilan | 25 | Delhi | 1500.00 |

| 3 | kaushik | 23 | Kota | 2000.00 |

| 4 | Chaitali | 25 | Mumbai | 6500.00 |

| 5 | Hardik | 27 | Bhopal | 8500.00 |

| 6 | Komal | 22 | MP | 4500.00 |

| 7 | Muffy | 24 | Indore | 10000.00 |

+----+----------+-----+-----------+----------+

Now, following is the example to create a view from CUSTOMERS table. This view would be used to have customer name and age from CUSTOMERS table:

**SQL > CREATE VIEW CUSTOMERS\_VIEW AS SELECT name, age FROM CUSTOMERS;**

Now, we can query CUSTOMERS\_VIEW in similar way as we query an actual table. Following is the example:

SQL > SELECT \* FROM CUSTOMERS\_VIEW;

This would produce the following result:

+----------+-----+

| name | age |

+----------+-----+

| Ramesh | 32 |

| Khilan | 25 |

| kaushik | 23 |

| Chaitali | 25 |

| Hardik | 27 |

| Komal | 22 |

| Muffy | 24 |

+----------+-----+

**The WITH CHECK OPTION:**

The WITH CHECK OPTION is a CREATE VIEW statement option. The purpose of the WITH CHECK OPTION is to ensure that all UPDATE and INSERTs satisfy the condition(s) in the view definition. If they do not satisfy the condition(s), the UPDATE or INSERT returns an error.

The following is an example of creating same view CUSTOMERS\_VIEW with the WITH CHECK OPTION:

**CREATE VIEW CUSTOMERS\_VIEW AS**

**SELECT name, age**

**FROM CUSTOMERS**

**WHERE age IS NOT NULL**

**WITH CHECK OPTION;**

The WITH CHECK OPTION in this case should deny the entry of any NULL values in the view's AGE column, because the view is defined by data that does not have a NULL value in the AGE column.

**Updating a View:** A view can be updated under certain conditions:

* The SELECT clause may not contain the keyword DISTINCT.
* The SELECT clause may not contain summary functions.
* The SELECT clause may not contain set functions.
* The SELECT clause may not contain set operators.
* The SELECT clause may not contain an ORDER BY clause.
* The FROM clause may not contain multiple tables.
* The WHERE clause may not contain subqueries.
* The query may not contain GROUP BY or HAVING.
* Calculated columns may not be updated.
* All NOT NULL columns from the base table must be included in the view in order for the INSERT query to function.

So if a view satisfies all the above mentioned rules then we can update a view. Following is an example to update the age of Ramesh:

**SQL > UPDATE CUSTOMERS\_VIEW SET AGE = 35 WHERE name='Ramesh';**

This would ultimately update the base table CUSTOMERS and same would reflect in the view itself. Now, try to query base table, and SELECT statement would produce the following result:

+----+----------+-----+-----------+----------+

| ID | NAME | AGE | ADDRESS | SALARY |

+----+----------+-----+-----------+----------+

| 1 | Ramesh | 35 | Ahmedabad | 2000.00 |

| 2 | Khilan | 25 | Delhi | 1500.00 |

| 3 | kaushik | 23 | Kota | 2000.00 |

| 4 | Chaitali | 25 | Mumbai | 6500.00 |

| 5 | Hardik | 27 | Bhopal | 8500.00 |

| 6 | Komal | 22 | MP | 4500.00 |

| 7 | Muffy | 24 | Indore | 10000.00 |

+----+----------+-----+-----------+----------+

**Inserting Rows into a View:** Rows of data can be inserted into a view. The same rules that apply to the UPDATE command also apply to the INSERT command. Here, we cannot insert rows in CUSTOMERS\_VIEW because we have not included all the NOT NULL columns in this view, otherwise we can insert rows in a view in similar way as we insert them in a table.

**Deleting Rows into a View:** Rows of data can be deleted from a view. The same rules that apply to the UPDATE and INSERT commands apply to the DELETE command. Following is an example to delete a record having AGE= 22.

**SQL > DELETE FROM CUSTOMERS\_VIEW WHERE age = 22;**

This would ultimately delete a row from the base table CUSTOMERS and same would reflect in the view itself. Now, try to query base table, and SELECT statement would produce the following result:

+----+----------+-----+-----------+----------+

| ID | NAME | AGE | ADDRESS | SALARY |

+----+----------+-----+-----------+----------+

| 1 | Ramesh | 35 | Ahmedabad | 2000.00 |

| 2 | Khilan | 25 | Delhi | 1500.00 |

| 3 | kaushik | 23 | Kota | 2000.00 |

| 4 | Chaitali | 25 | Mumbai | 6500.00 |

| 5 | Hardik | 27 | Bhopal | 8500.00 |

| 7 | Muffy | 24 | Indore | 10000.00 |

+----+----------+-----+-----------+----------+

**Dropping Views:** Obviously, where we have a view, we need a way to drop the view if it is no longer needed. The syntax is very simple as given below:

**DROP VIEW view\_name;**

Following is an example to drop CUSTOMERS\_VIEW from CUSTOMERS table:

**SQL > DROP VIEW CUSTOMERS\_VIEW;**

**SQL HAVING CLAUSE:** The HAVING clause enables we to specify conditions that filter which group results appear in the final results. The WHERE clause places conditions on the selected columns, whereas the HAVING clause places conditions on groups created by the GROUP BY clause.

**Syntax:** The following is the position of the HAVING clause in a query:

**SELECT**

**FROM**

**WHERE**

**GROUP BY**

**HAVING**

**ORDER BY**

The HAVING clause must follow the GROUP BY clause in a query and must also precede the ORDER BY clause if used. The following is the syntax of the SELECT statement, including the HAVING clause:

**SELECT column1, column2**

**FROM table1, table2**

**WHERE [ conditions ]**

**GROUP BY column1, column2**

**HAVING [ conditions ]**

**ORDER BY column1, column2**

**Example:** Consider the CUSTOMERS table having the following records:

+----+----------+-----+-----------+----------+

| ID | NAME | AGE | ADDRESS | SALARY |

+----+----------+-----+-----------+----------+

| 1 | Ramesh | 32 | Ahmedabad | 2000.00 |

| 2 | Khilan | 25 | Delhi | 1500.00 |

| 3 | kaushik | 23 | Kota | 2000.00 |

| 4 | Chaitali | 25 | Mumbai | 6500.00 |

| 5 | Hardik | 27 | Bhopal | 8500.00 |

| 6 | Komal | 22 | MP | 4500.00 |

| 7 | Muffy | 24 | Indore | 10000.00 |

+----+----------+-----+-----------+----------+

Following is the example, which would display record for which similar age count would be more than or equal to 2:

**SQL > SELECT \* FROM CUSTOMERS GROUP BY age HAVING COUNT(age) >= 2;**

This would produce the following result:

+----+--------+-----+---------+---------+

| ID | NAME | AGE | ADDRESS | SALARY |

+----+--------+-----+---------+---------+

| 2 | Khilan | 25 | Delhi | 1500.00 |