Unit 1 Overview of Basic SQL statements

Introduction

SQL is Structured Query Language, which is a computer language for storing, manipulating and retrieving data stored in a relational database.

SQL is the standard language for Relational Database System. All the Relational Database Management Systems (RDMS) like MySQL, MS Access, Oracle, Sybase, Informix, Postgres and SQL Server use SQL as their standard database language.

SQL Process

When you are executing an SQL command for any RDBMS, the system determines the best way to carry out your request and SQL engine figures out how to interpret the task.

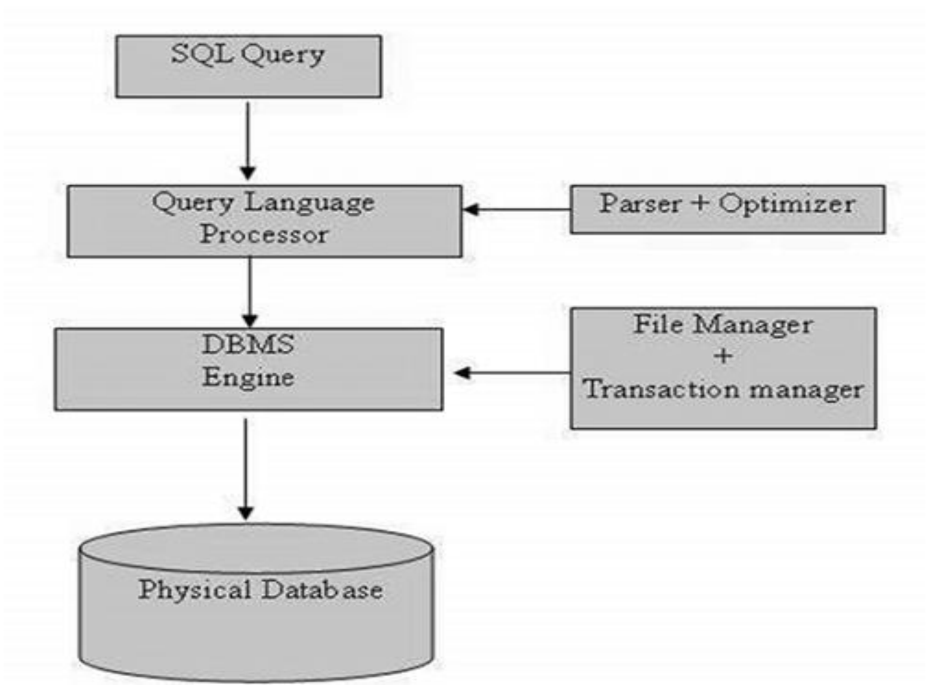
There are various components included in this process.

These components are −

- Query Dispatcher
- Optimization Engines
- Classic Query Engine
- SQL Query Engine, etc.

A classic query engine handles all the non-SQL queries, but a SQL query engine won't handle logical files.

Following is a simple diagram showing the SQL Architecture −



SQL Commands

DDL (Data Definition Language):

Data Definition Language is used to define the database structure or schema. DDL is also used to specify additional properties of the data. The storage structure and access methods used by the database system by a set of statements in a special type of DDL called a data storage and definition language. These statements define the implementation details of the database schema, which are usually hidden from the users. The data values stored in the database must satisfy certain consistency constraints.

Some Commands:

CREATE : to create objects in database

ALTER : alters the structure of database

DROP : delete objects from database

RENAME : rename an objects

Following SQL DDL-statement defines the department table :

```
create table department
(dept_name  char(20),
  building   char(15),
  budget     numeric(12,2));
```

Execution of the above DDL statement creates the department table with three columns – dept_name, building, and budget; each of which has a specific datatype associated with it.

DML (Data Manipulation Language):

DML statements are used for managing data with in schema objects.
DML are of two types –

1. **Procedural DMLs** : require a user to specify what data are needed and how to get those data.
2. **Declerative DMLs** (also referred as **Non-procedural DMLs**) : require a user to specify what data are needed without specifying how to get those data.
   Declarative DMLs are usually easier to learn and use than procedural DMLs. However, since a user does not have to specify how to get the data, the database system has to figure out an efficient means of accessing data.

Some Commands :

SELECT: retrieve data from the database

INSERT: insert data into a table

UPDATE: update existing data within a table

DELETE: deletes all records from a table, space for the records remain

Example of SQL query that finds the names of all instructors in the History department :

```
select instructor.name
 from instructor
 where instructor.dept_name = 'History';
```

The query specifies that those rows from the table instructor where the dept_name is History must be retrieved and the name attributes of these rows must be displayed.

Basic SQL Statements

1.1 Select statement with all clauses

1.1.1 SQL SELECT

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

Syntax

The basic syntax of the SELECT statement is as follows −

```
SELECT column1, column2, columnN FROM table_name;
```

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

```
SELECT * FROM table_name;
```

The following code is an example, which would fetch the ID, Name and Salary fields of the customers available in CUSTOMERS table.

```
SELECT ID, NAME, SALARY FROM CUSTOMERS;
```

If you want to fetch all the fields of the CUSTOMERS table, then you should use the following query.

```
SELECT * FROM CUSTOMERS;
```

1.1.2 SQL SELECT DISTINCT Statement

The SELECT DISTINCT statement is used to return only distinct (different) values. Inside a table, a column often contains many duplicate values; and sometimes you only want to list the different (distinct) values.

SELECT DISTINCT Syntax

SELECT DISTINCT *column1*, *column2, ...*
FROM *table_name*;

SELECT Example Without DISTINCT

The following SQL statement selects ALL (including the duplicates) values from the "Country" column in the "Customers" table:

Example

```
SELECT Country FROM Customers;
```

1.1.3 The SQL WHERE Clause

The WHERE clause is used to filter records. The WHERE clause is used to extract only those records that fulfill a specified condition.

Syntax

The basic syntax of the SELECT statement with the WHERE clause is as shown below.

```
SELECT column1, column2, columnN
FROM table_name
WHERE [condition]
```

You can specify a condition using the comparison or logical operators like >, <, =, **LIKE, NOT**, etc. The following examples would make this concept clear.

The following code is an example which would fetch the ID, Name and Salary fields from the CUSTOMERS table, where the salary is greater than 2000 −

```
SELECT ID, NAME, SALARY

FROM CUSTOMERS

WHERE SALARY > 2000;
```

The following query is an example, which would fetch the ID, Name and Salary fields from the CUSTOMERS table for a customer with the name **Hardik**.

Here, it is important to note that all the strings should be given inside single quotes (''). Whereas, numeric values should be given without any quote as in the above example.

```
SELECT ID, NAME, SALARY

FROM CUSTOMERS

WHERE NAME = 'Hardik';
```

1.1.4 The SQL AND, OR and NOT Operators

The WHERE clause can be combined with AND, OR, and NOT operators. The AND and OR operators are used to filter records based on more than one condition:

- The AND operator displays a record if all the conditions separated by AND are TRUE.
- The OR operator displays a record if any of the conditions separated by OR is TRUE.

The NOT operator displays a record if the condition(s) is NOT TRUE.

AND Syntax

SELECT *column1*, *column2, ...*
FROM *table_name*
WHERE *condition1* AND *condition2* AND *condition3 ...*;

OR Syntax

SELECT *column1*, *column2, ...*
FROM *table_name*
WHERE *condition1* OR *condition2* OR *condition3 ...*;

NOT Syntax

SELECT *column1*, *column2, ...*
FROM *table_name*
WHERE NOT *condition*;

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

```
SELECT ID, NAME, SALARY

FROM CUSTOMERS

WHERE SALARY > 2000 AND age < 25;
```

The following code block has a query, which would fetch the ID, Name and Salary fields from the CUSTOMERS table, where the salary is greater than 2000 OR the age is less than 25 years.

```
 SELECT ID, NAME, SALARY

FROM CUSTOMERS

WHERE SALARY > 2000 OR age < 25;
```

NOT Example

The following SQL statement selects all fields from "Customers" where country is NOT "Germany":

Example

```
SELECT * FROM Customers
WHERE NOT Country='Germany';
```

1.1.5 SQL ORDER BY

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 the query results in an ascending order by default.

Syntax

The basic syntax of the ORDER BY clause is as follows −

```
SELECT column-list
FROM table_name
[WHERE condition]
[ORDER BY column1, column2, .. columnN] [ASC | DESC];
```

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

The following code block has an example, which would sort the result in an ascending order by the NAME and the SALARY −

```
SELECT * FROM CUSTOMERS

  ORDER BY NAME, SALARY;
```

The following code block has an example, which would sort the result in the descending order by NAME.

```
SELECT * FROM CUSTOMERS

  ORDER BY NAME DESC;
```

1.1.6 SQL GROUP BY

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

Syntax

The basic syntax of a GROUP BY clause is shown in the following code block. 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
```

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

```
SELECT NAME, SUM(SALARY) FROM CUSTOMERS

  GROUP BY NAME;
```

1.1.7 SQL LIKE

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. These 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_'
```

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

Example

The following table has a few examples showing the WHERE part having different LIKE clause with '%' and '_' operators −

Sr.No.	**Statement & Description**
1	**WHERE SALARY LIKE '200%'** Finds any values that start with 200.
2	**WHERE SALARY LIKE '%200%'** Finds any values that have 200 in any position.
3	**WHERE SALARY LIKE '_00%'** Finds any values that have 00 in the second and third positions.
4	**WHERE SALARY LIKE '2_%_%'** Finds any values that start with 2 and are at least 3 characters in length.
5	**WHERE SALARY LIKE '%2'** Finds any values that end with 2.
6	**WHERE SALARY LIKE '_2%3'** Finds any values that have a 2 in the second position and end with a 3.

7	**WHERE SALARY LIKE '2___3'** Finds any values in a five-digit number that start with 2 and end with 3.

Following is an example, which would display all the records from the CUSTOMERS table, where the SALARY starts with 200.

```
SELECT * FROM CUSTOMERS

WHERE SALARY LIKE '200%';
```

1.1.8 SQL TOP

The SQL **TOP** clause is used to fetch a TOP N number or X percent records from a table.

Note − All the databases do not support the TOP clause. For example MySQL supports the **LIMIT** clause to fetch limited number of records while Oracle uses the **ROWNUM** command to fetch a limited number of records.

Syntax

The basic syntax of the TOP clause with a SELECT statement would be as follows.

```
SELECT TOP number|percent column_name(s)
FROM table_name
WHERE [condition]
```

The following query is an example on the SQL server, which would fetch the top 3 records from the CUSTOMERS table.

```
SELECT TOP 3 * FROM CUSTOMERS;
```

If you are using MySQL server, then here is an equivalent example −

```
SELECT * FROM CUSTOMERS

LIMIT 3;
```

If you are using an Oracle server, then the following code block has an equivalent example.

```
SELECT * FROM CUSTOMERS

WHERE ROWNUM <= 3;
```

1.1.9 SQL functions

SQL has many built-in functions for performing processing on string or numeric data. Following is the list of all useful SQL built-in functions −

- SQL COUNT Function - The SQL COUNT aggregate function is used to count the number of rows in a database table.
- SQL MAX Function - The SQL MAX aggregate function allows us to select the highest (maximum) value for a certain column.
- SQL MIN Function - The SQL MIN aggregate function allows us to select the lowest (minimum) value for a certain column.
- SQL AVG Function - The SQL AVG aggregate function selects the average value for certain table column.
- SQL SUM Function - The SQL SUM aggregate function allows selecting the total for a numeric column.

MIN() Example

The following SQL statement finds the price of the cheapest product:

Example

```
SELECT MIN(Price) AS SmallestPrice
FROM Products;
```

MAX() Example

The following SQL statement finds the price of the most expensive product:

Example

```
SELECT MAX(Price) AS LargestPrice
FROM Products;
```

COUNT()

Example

The following SQL statement finds the number of products:

Example

```
SELECT COUNT(ProductID)
FROM Products;
```

AVG() Example

The following SQL statement finds the average price of all products:

Example

```
SELECT AVG(Price)
FROM Products;
```

SUM() Example

The following SQL statement finds the sum of the "Quantity" fields in the "OrderDetails" table:

Example

```
SELECT SUM(Quantity)
FROM OrderDetails;
```

1.1.10 SQL IN Operator

The IN operator allows you to specify multiple values in a WHERE clause. The IN operator is a shorthand for multiple OR conditions.

IN Syntax

SELECT *column_name(s)*
FROM *table_name*
WHERE *column_name* IN (*value1*, *value2*, ...);

or:

SELECT *column_name(s)*
FROM *table_name*
WHERE *column_name* IN (*SELECT STATEMENT*);

IN Operator Examples

The following SQL statement selects all customers that are located in "Germany", "France" and "UK":

Example

```
SELECT * FROM Customers
WHERE Country IN ('Germany', 'France', 'UK');
```

The following SQL statement selects all customers that are NOT located in "Germany", "France" or "UK":

Example

```
SELECT * FROM Customers
WHERE Country NOT IN ('Germany', 'France', 'UK');
```

The following SQL statement selects all customers that are from the same countries as the suppliers:

Example

```
SELECT * FROM Customers
WHERE Country IN (SELECT Country FROM Suppliers);
```

1.1.11 SQL BETWEEN Operators

The BETWEEN operator selects values within a given range. The values can be numbers, text, or dates.The BETWEEN operator is inclusive: begin and end values are included.

BETWEEN Syntax

SELECT *column_name(s)*
FROM *table_name*
WHERE *column_name* BETWEEN *value1* AND *value2;*

BETWEEN Example

The following SQL statement selects all products with a price BETWEEN 10 and 20:

Example

```
SELECT * FROM Products
WHERE Price BETWEEN 10 AND 20;
```

NOT BETWEEN Example

To display the products outside the range of the previous example, use NOT BETWEEN:

Example

```
SELECT * FROM Products
WHERE Price NOT BETWEEN 10 AND 20;
```

Example

```
SELECT * FROM Products
WHERE ProductName BETWEEN 'Carnarvon Tigers' AND 'Mozzarella di Giovanni'
ORDER BY ProductName;
```

1.1.12 SQL Aliases

SQL aliases are used to give a table, or a column in a table, a temporary name. Aliases are often used to make column names more readable. An alias only exists for the duration of the query.

Alias Column Syntax

SELECT *column_name* AS *alias_name*
FROM *table_name;*

Alias Table Syntax

```
SELECT column_name(s)
FROM table_name AS alias_name;
```

Alias for Columns Examples

The following SQL statement creates two aliases, one for the CustomerID column and one for the CustomerName column:

Example

```
SELECT CustomerID AS ID, CustomerName AS Customer
FROM Customers;
```

The following SQL statement creates two aliases, one for the CustomerName column and one for the ContactName column. **Note:** It requires double quotation marks or square brackets if the alias name contains spaces:

Example

```
SELECT CustomerName AS Customer, ContactName AS [Contact Person]
FROM Customers;
```

The following SQL statement creates an alias named "Address" that combine four columns (Address, PostalCode, City and Country):

Example

```
SELECT CustomerName, Address + ', ' + PostalCode + ' ' + City + ', ' + Country AS Address
FROM Customers;
```

Note: To get the SQL statement above to work in MySQL use the following:

```
SELECT CustomerName, CONCAT(Address,', ',PostalCode,', ',City,', ',Country) AS Address
FROM Customers;
```

Alias for Tables Example

The following SQL statement selects all the orders from the customer with CustomerID=4 (Around the Horn). We use the "Customers" and "Orders" tables, and give them the table aliases of "c" and "o" respectively (Here we use aliases to make the SQL shorter):

Example

```
SELECT o.OrderID, o.OrderDate, c.CustomerName
FROM Customers AS c, Orders AS o
WHERE c.CustomerName="Around the Horn" AND c.CustomerID=o.CustomerID;
```

The following SQL statement is the same as above, but without aliases:

Example

```
SELECT Orders.OrderID, Orders.OrderDate, Customers.CustomerName
FROM Customers, Orders
WHERE Customers.CustomerName="Around the
Horn" AND Customers.CustomerID=Orders.CustomerID;
```

1.1.13 SQL GROUP BY Statement

The GROUP BY statement group rows that have the same values into summary rows, like "find the number of customers in each country". The GROUP BY statement is often used with aggregate functions (COUNT, MAX, MIN, SUM, AVG) to group the result-set by one or more columns.

GROUP BY Syntax

```
SELECT column_name(s)
FROM table_name
WHERE condition
GROUP BY column_name(s)
ORDER BY column_name(s);
```

SQL GROUP BY Examples

The following SQL statement lists the number of customers in each country:

Example

```
SELECT COUNT(CustomerID), Country
FROM Customers
GROUP BY Country;
```

The following SQL statement lists the number of customers in each country, sorted high to low:

Example

```
SELECT COUNT(CustomerID), Country
FROM Customers GROUP BY Country
ORDER BY COUNT(CustomerID) DESC;
```

1.1.14 SQL HAVING Clause

The HAVING clause was added to SQL because the WHERE keyword could not be used with aggregate functions.

HAVING Syntax

```
SELECT column_name(s)
FROM table_name
WHERE condition
GROUP BY column_name(s) HAVING condition
ORDER BY column_name(s);
```

SQL HAVING Examples

The following SQL statement lists the number of customers in each country. Only include countries with more than 5 customers:

Example

```
SELECT COUNT(CustomerID), Country
FROM Customers
GROUP BY Country
HAVING COUNT(CustomerID) > 5;
```

The following SQL statement lists the number of customers in each country, sorted high to low (Only include countries with more than 5 customers):

Example

```
SELECT COUNT(CustomerID), Country
FROM Customers
GROUP BY Country
HAVING COUNT(CustomerID) > 5
ORDER BY COUNT(CustomerID) DESC;
```

1.1.15 SQL EXISTS Operator

The EXISTS operator is used to test for the existence of any record in a subquery. The EXISTS operator returns true if the subquery returns one or more records.

EXISTS Syntax

```
SELECT column_name(s)
FROM table_name
WHERE EXISTS
(SELECT column_name FROM table_name WHERE condition);
```

SQL EXISTS Examples

The following SQL statement returns TRUE and lists the suppliers with a product price less than 20:

Example

```
SELECT SupplierName
FROM Suppliers
WHERE EXISTS (SELECT ProductName FROM Products WHERE Products.SupplierID =
Suppliers.supplierID AND Price < 20);
```

1.1.16 SQL ANY and ALL Operators

The ANY and ALL operators are used with a WHERE or HAVING clause. The ANY operator returns true if any of the subquery values meet the condition. The ALL operator returns true if all of the subquery values meet the condition.

ANY Syntax

SELECT *column_name(s)*
FROM *table_name*
WHERE *column_name operator* ANY
(SELECT *column_name* FROM *table_name* WHERE *condition*);

ALL Syntax

SELECT *column_name(s)*
FROM *table_name*
WHERE *column_name operator* ALL
(SELECT *column_name* FROM *table_name* WHERE *condition*);

SQL ANY Examples

The ANY operator returns TRUE if any of the subquery values meet the condition.

The following SQL statement returns TRUE and lists the productnames if it finds ANY records in the OrderDetails table that quantity = 10:

Example

```
SELECT ProductName
FROM Products
WHERE ProductID = ANY (SELECT ProductID FROM OrderDetails WHERE Quantity = 10);
```

The following SQL statement returns TRUE and lists the productnames if it finds ANY records in the OrderDetails table that quantity > 99:

Example

```
SELECT ProductName
FROM Products
WHERE ProductID = ANY (SELECT ProductID FROM OrderDetails WHERE Quantity > 99);
```

SQL ALL Example

The ALL operator returns TRUE if all of the subquery values meet the condition.

The following SQL statement returns TRUE and lists the productnames if ALL the records in the OrderDetails table has quantity = 10:

Example

```
SELECT ProductName
FROM Products
WHERE ProductID = ALL (SELECT ProductID FROM OrderDetails WHERE Quantity = 10);
```

1.2 Insert, Update, Delete, Drop, adding and removing constraints like etc.

1.2.1 SQL INSERT INTO Statement

The INSERT INTO statement is used to insert new records in a table.

INSERT INTO Syntax

It is possible to write the INSERT INTO statement in two ways.The first way specifies both the column names and the values to be inserted:

```
INSERT INTO table_name (column1, column2, column3, ...)
VALUES (value1, value2, value3, ...);
```

If you are adding values for all the columns of the table, you do not need to specify the column names in the SQL query. However, make sure the order of the values is in the same order as the columns in the table. The INSERT INTO syntax would be as follows:

```
INSERT INTO table_name
VALUES (value1, value2, value3, ...);
```

INSERT INTO Example

The following SQL statement inserts a new record in the "Customers" table:

Example

```
INSERT INTO Customers (CustomerName, ContactName, Address, City, PostalCode, Country)
VALUES ('Cardinal', 'Tom B. Erichsen', 'Skagen 21', 'Stavanger', '4006', 'Norway');
```

Insert Data Only in Specified Columns

It is also possible to only insert data in specific columns.

The following SQL statement will insert a new record, but only insert data in the "CustomerName", "City", and "Country" columns (CustomerID will be updated automatically):

Example

```
INSERT INTO Customers (CustomerName, City, Country)
VALUES ('Cardinal', 'Stavanger', 'Norway');
```

1.2.2 SQL INSERT INTO SELECT Statement

The INSERT INTO SELECT statement copies data from one table and inserts it into another table.

- INSERT INTO SELECT requires that data types in source and target tables match
- The existing records in the target table are unaffected

INSERT INTO SELECT Syntax

Copy all columns from one table to another table:

```
INSERT INTO table2
SELECT * FROM table1
WHERE condition;
```

Copy only some columns from one table into another table:

```
INSERT INTO table2 (column1, column2, column3, ...)
SELECT column1, column2, column3, ...
FROM table1
WHERE condition;
```

SQL INSERT INTO SELECT Examples

The following SQL statement copies "Suppliers" into "Customers" (the columns that are not filled with data, will contain NULL):

Example

```
INSERT INTO Customers (CustomerName, City, Country)
SELECT SupplierName, City, Country FROM Suppliers;
```

The following SQL statement copies "Suppliers" into "Customers" (fill all columns):

Example

```
INSERT INTO Customers (CustomerName, ContactName, Address, City, PostalCode, Country)
SELECT SupplierName, ContactName, Address, City, PostalCode, Country FROM Suppliers;
```

The following SQL statement copies only the German suppliers into "Customers":

Example

```
INSERT INTO Customers (CustomerName, City, Country)
SELECT SupplierName, City, Country FROM Suppliers
WHERE Country='Germany';
```

1.2.3 SQL UPDATE

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

Syntax

The basic syntax of the UPDATE query with a WHERE clause is as follows −

```
UPDATE table_name
SET column1 = value1, column2 = value2...., columnN = valueN
WHERE [condition];
```

The following query will update the ADDRESS for a customer whose ID number is 6 in the table.

```
UPDATE CUSTOMERS

SET ADDRESS = 'Pune'

WHERE ID = 6;
```

If you want to modify all the ADDRESS and the SALARY column values in the CUSTOMERS table, you do not need to use the WHERE clause as the UPDATE query would be enough as shown in the following code block.

```
UPDATE CUSTOMERS

SET ADDRESS = 'Pune', SALARY = 1000.00;
```

1.2.4 SQL DELETE

The SQL DELETE Query is used to delete the existing records from a table.

You can use the WHERE clause with a DELETE query to delete the selected rows, otherwise all the records would be deleted.

Syntax

The basic syntax of the DELETE query with the WHERE clause is as follows −

```
DELETE FROM table_name
WHERE [condition];
```

The following code has a query, which will DELETE a customer, whose ID is 6.

```
DELETE FROM CUSTOMERS

WHERE ID = 6;
```

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

```
DELETE FROM CUSTOMERS;
```

Now, the CUSTOMERS table would not have any record.

1.2.5 SQL DROP DATABASE Statement

The DROP DATABASE statement is used to drop an existing SQL database.

Syntax

```
DROP DATABASE databasename;
```

DROP DATABASE Example

The following SQL statement drops the existing database "testDB":

Example

```
DROP DATABASE testDB;
```

1.2.5 SQL DROP TABLE Statement

The DROP TABLE statement is used to drop an existing table in a database.

Syntax

```
DROP TABLE table_name;
```

SQL DROP TABLE Example

The following SQL statement drops the existing table "Shippers":

Example

```
DROP TABLE Shippers;
```

1.2.6 SQL TRUNCATE TABLE

The TRUNCATE TABLE statement is used to delete the data inside a table, but not the table itself.

Syntax

```
TRUNCATE TABLE table_name;
```

Example

```
TRUNCATE TABLE Shippers;
```

1.2.7 SQL DROP COLUMN Statement

The DROP COLUMN statement is used to drop an existing column in a database.

Syntax

```
ALTER TABLE table_name DROP column_name;
```

SQL DROP COLUMN Example

The following SQL statement drops the existing column "Contact":

Example

```
ALTER TABLE tbl_emplyoee DROP Contact;
```

1.2.8 Adding and Removing Constraints

Constraints can be added to a new table or to an existing table. To add a unique or primary key, a referential constraint, or a check constraint, use the CREATE TABLE or the ALTER TABLE statement. To remove a constraint, use the ALTER TABLE statement.

For example, add a primary key to an existing table using the ALTER TABLE statement:

```
ALTER TABLE CORPDATA.DEPARTMENT
  ADD PRIMARY KEY (DEPTNO)
```

To make this key a unique key, replace the keyword PRIMARY with UNIQUE.
You can remove a constraint using the same ALTER TABLE statement:

```
ALTER TABLE CORPDATA.DEPARTMENT
  DROP PRIMARY KEY (DEPTNO)
```

1.3 Views, granting and removing privileges

1.3.1 Views

Views in SQL are kind of virtual tables. A view also has rows and columns as they are in a real table in the database. We can create a view by selecting fields from one or more tables present in the database. A View can either have all the rows of a table or specific rows based on certain condition.

Syntax:

```
CREATE VIEW view_name AS
SELECT column1, column2.....
FROM table_name
WHERE condition;
```

Examples:

- **Creating View from a single table:**
  - In this example we will create a View named DetailsView from the table StudentDetails.

```
• CREATE VIEW DetailsView AS
• SELECT NAME, ADDRESS
• FROM StudentDetails
• WHERE S_ID < 5;
```

  To see the data in the View, we can query the view in the same manner as we query a table.

```
SELECT * FROM DetailsView;
```

- **Creating View from multiple tables**: In this example we will create a View named MarksView from two tables StudentDetails and StudentMarks. To create a View from multiple tables we can simply include multiple tables in the SELECT statement.

```
CREATE VIEW MarksView AS
SELECT StudentDetails.NAME, StudentDetails.ADDRESS, StudentMarks.MARKS
FROM StudentDetails, StudentMarks
WHERE StudentDetails.NAME = StudentMarks.NAME;
```

To display data of View MarksView:

```
SELECT * FROM MarksView;
```

Drop a View using the DROP statement.

Syntax:

```
DROP VIEW view_name;
```

For example, if we want to delete the View **MarksView**, we can do this as:

```
DROP VIEW MarksView;
```

UPDATING VIEWS

There are certain conditions needed to be satisfied to update a view. If any one of these conditions is **not** met, then we will not be allowed to update the view.

1. The SELECT statement which is used to create the view should not include GROUP BY clause or ORDER BY clause.
2. The SELECT statement should not have the DISTINCT keyword.
3. The View should have all NOT NULL values.
4. The view should not be created using nested queries or complex queries.

5. The view should be created from a single table. If the view is created using multiple tables then we will not be allowed to update the view.

So, if a view satisfies all the above-mentioned rules then you can update that view. The following code block has an example to update the age of Ramesh.

```
UPDATE CUSTOMERS_VIEW

SET AGE = 35

WHERE name = 'Ramesh';
```

This would ultimately update the base table CUSTOMERS and the same would reflect in the view itself.

CREATE OR REPLACE VIEW

We can use the **CREATE OR REPLACE VIEW** statement to add or remove fields from a view.

Syntax:

```
CREATE OR REPLACE VIEW view_name AS

SELECT column1,coulmn2,..

FROM table_name

WHERE condition;
```

For example, if we want to update the view **MarksView** and add the field AGE to this View from **StudentMarks** Table, we can do this as:

```
CREATE OR REPLACE VIEW MarksView AS

SELECT StudentDetails.NAME, StudentDetails.ADDRESS, StudentMarks.MARKS, StudentMarks.AGE

FROM StudentDetails, StudentMarks

WHERE StudentDetails.NAME = StudentMarks.NAME;
```

Inserting a row in a view:

We can insert a row in a View in a same way as we do in a table. We can use the INSERT INTO statement of SQL to insert a row in a View.

Syntax:

```
INSERT view_name(column1, column2 , column3,..)

VALUES(value1, value2, value3..);
```

Example:

In the below example we will insert a new row in the View DetailsView which we have created above in the example of "creating views from a single table".

```
INSERT INTO DetailsView(NAME, ADDRESS)
VALUES("Suresh","Gurgaon");
```

Deleting a row from a View:

Deleting rows from a view is also as simple as deleting rows from a table. We can use the DELETE statement of SQL to delete rows from a view. Also deleting a row from a view first delete the row from the actual table and the change is then reflected in the view.

Syntax:

```
DELETE FROM view_name
WHERE condition;
```

Example:

In this example we will delete the last row from the view DetailsView which we just added in the above example of inserting rows.

```
DELETE FROM DetailsView
WHERE NAME="Suresh";
```

1.3.2 Granting and Removing Privileges

CREATE USER Statement

The **CREATE USER** statement in MySQL allows us to create new MySQL accounts or in other words, the **CREATE USER** statement is used to create a database account that allows the user to log into the MySQL database.

Syntax:
The syntax for the CREATE USER statement in MySQL is:

CREATE USER user_account IDENTIFIED BY password;

Parameters used:

1. **user_account:** It is the name that the user wants to give to the database account.The user_account should be in the format **'username'@'hostname'**
2. **password:**It is the password used to assign to the user_account.The password is specified in the IDENTIFIED BY clause.

Below are the different ways in which CREATE USER statement can be used:

1. **Creating a new user**: For creating a new user “gfguser1” that connects to the MySQL database server from the localhost with the password “abcd”, the CREATE USER statement should be used in the following way.

Syntax:

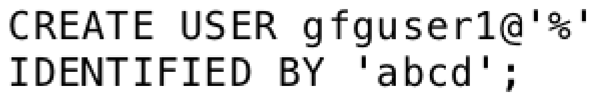
CREATE USER gfguser1@localhost IDENTIFIED BY 'abcd';

Note: The create user statement only creates a new user,it does not grant any permissions to the user.

2. **Creating more than one user**: For creating more than one new user that connects to the MySQL database server from the localhost, the CREATE USER statement should be used in the following way.
   Syntax:
3. CREATE USER
4. 'gfguser2'@'localhost' IDENTIFIED BY 'efgh',
5. 'gfguser3'@'localhost' IDENTIFIED BY 'uvxy';

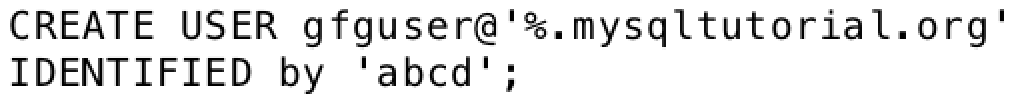
The above code creates two new users with username “gfguser2” and “gfguser3” with passwords “efgh” and “uvxy” respectively.

6. **Allowing a user account to connect from any host**: To allow a user account to connect from any host, the percentage (%) wildcard is used in the following way.
   Syntax:



```
CREATE USER gfguser1@'%'
IDENTIFIED BY 'abcd';
```

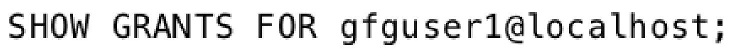
To allow the user account to connect to the database server from any subdomain of the “mysqltutorial.org” host, then the percentage wildcard % should be used as follows:
Syntax:



```
CREATE USER gfguser@'%.mysqltutorial.org'
IDENTIFIED by 'abcd';
```

7. **Viewing permissions of an User Account**: The “Show Grants” statement is used to view the permissions of a user account.The show grants statement is used in the following way:
   Synatx:
8. SHOW GRANTS FOR user-account;

Example:



```
SHOW GRANTS FOR gfguser1@localhost;
```

Grant / Revoke Privileges

Syntax:

GRANT privileges_names ON object TO user;

Parameters Used:

- **privileges_name**: These are the access rights or privileges granted to the user.
- **object:**It is the name of the database object to which permissions are being granted. In the case of granting privileges on a table, this would be the table name.
- **user:**It is the name of the user to whom the privileges would be granted.

Let us now learn about different ways of granting privileges to the users:

1. **Granting SELECT Privilege to a User in a Table**: To grant Select Privilege to a table named "users" where User Name is Amit, the following GRANT statement should be executed.

   ```
   GRANT SELECT ON Users TO'Amit'@'localhost;
   ```

2. **Granting more than one Privilege to a User in a Table**: To grant multiple Privileges to a user named "Amit" in a table "users", the following GRANT statement should be executed.

   ```
   GRANT SELECT, INSERT, DELETE, UPDATE ON Users TO 'Amit'@'localhost;
   ```

3. **Granting All the Privilege to a User in a Table**: To Grant all the privileges to a user named "Amit" in a table "users", the following Grant statement should be executed.

   ```
   GRANT ALL ON Users TO 'Amit'@'localhost;
   ```

4. **Granting a Privilege to all Users in a Table**: To Grant a specific privilege to all the users in a table "users", the following Grant statement should be executed.

   ```
   GRANT SELECT  ON Users TO '*'@'localhost;
   ```

In the above example the "*" symbol is used to grant select permission to all the users of the table "users".

Checking the Privileges Granted to a User: To see the privileges granted to a user in a table, the SHOW GRANTS statement is used. To check the privileges granted to a user named "Amit" and host as "localhost", the following SHOW GRANTS statement will be executed:

```
SHOW GRANTS FOR  'Amit'@localhost';
```

Revoking Privileges from a Table

The Revoke statement is used to revoke some or all of the privileges which have been granted to a user in the past.

Syntax:

```
REVOKE privileges ON object FROM user;
```

Parameters Used:

- **object:**It is the name of the database object from which permissions are being revoked. In the case of revoking privileges from a table, this would be the table name.
- **user:**It is the name of the user from whom the privileges are being revoked.

Different ways of revoking privileges from a user:

1. **Revoking SELECT Privilege to a User in a Table**: To revoke Select Privilege to a table named "users" where User Name is Amit, the following revoke statement should be executed
   .
   ```
   REVOKE SELECT ON  users TO 'Amit'@localhost';
   ```
2. **Revoking more than Privilege to a User in a Table**: To revoke multiple Privileges to a user named "Amit" in a table "users", the following revoke statement should be executed.
   ```
   REVOKE SELECT, INSERT, DELETE, UPDATE ON Users TO 'Amit'@'localhost;
   ```
3. **Revoking All the Privilege to a User in a Table**: To revoke all the privileges to a user named "Amit" in a table "users", the following revoke statement should be executed.
   ```
   REVOKE ALL ON Users TO 'Amit'@'localhost;
   ```
4. **Revoking a Privilege to all Users in a Table**: To Revoke a specific privilege to all the users in a table "users", the following revoke statement should be executed.
   ```
   REVOKE SELECT  ON Users TO '*'@'localhost;
   ```

1.4 Joins and its types

Join (Inner, Left, Right and Full Joins)

A SQL Join statement is used to combine data or rows from two or more tables based on a common field between them. Different types of Joins are:

- INNER JOIN
- LEFT JOIN
- RIGHT JOIN
- FULL JOIN

1. INNER JOIN: The INNER JOIN keyword selects all rows from both the tables as long as the condition satisfies. This keyword will create the result-set by combining all rows from both the tables where the condition satisfies i.e value of the common field will be same.
Syntax:

```
SELECT table1.column1,table1.column2,table2.column1,....
FROM table1
INNER JOIN table2
ON table1.matching_column = table2.matching_column;
```

This query will show the names and age of students enrolled in different courses.

```
•    SELECT StudentCourse.COURSE_ID, Student.NAME, Student.AGE FROM Student
•    INNER JOIN StudentCourse
•    ON Student.ROLL_NO = StudentCourse.ROLL_NO;
```

2. LEFT JOIN: This join returns all the rows of the table on the left side of the join and matching rows for the table on the right side of join. The rows for which there is no matching row on right side, the result-set will contain *null*. LEFT JOIN is also known as LEFT OUTER JOIN.

Syntax:

```
3.   SELECT table1.column1,table1.column2,table2.column1,....
4.   FROM table1
5.   LEFT JOIN table2
6.   ON table1.matching_column = table2.matching_column;
```

Example (LEFT JOIN):

```
SELECT Student.NAME,StudentCourse.COURSE_ID

FROM Student

LEFT JOIN StudentCourse

ON StudentCourse.ROLL_NO = Student.ROLL_NO;
```

3. RIGHT JOIN: RIGHT JOIN is similar to LEFT JOIN. This join returns all the rows of the table on the right side of the join and matching rows for the table on the left side of join. The rows for which there is no matching row on left side, the result-set will contain *null*. RIGHT JOIN is also known as RIGHT OUTER JOIN.

Syntax:

```
SELECT table1.column1,table1.column2,table2.column1,....

FROM table1

RIGHT JOIN table2

ON table1.matching_column = table2.matching_column;
```

Example (RIGHT JOIN):

```
SELECT Student.NAME,StudentCourse.COURSE_ID

FROM Student

RIGHT JOIN StudentCourse

ON StudentCourse.ROLL_NO = Student.ROLL_NO;
```

4. FULL JOIN: FULL JOIN creates the result-set by combining result of both LEFT JOIN and RIGHT JOIN. The result-set will contain all the rows from both the tables. The rows for which there is no matching, the result-set will contain *NULL* values.

Syntax:

```
SELECT table1.column1,table1.column2,table2.column1,....

FROM table1

FULL JOIN table2

ON table1.matching_column = table2.matching_column;
```

Example (FULL JOIN):

```
SELECT Student.NAME,StudentCourse.COURSE_ID

FROM Student

FULL JOIN StudentCourse

ON StudentCourse.ROLL_NO = Student.ROLL_NO;
```

5. CARTESIAN JOIN: The CARTESIAN JOIN is also known as CROSS JOIN. In a CARTESIAN JOIN there is a join for each row of one table to every row of another table. This usually happens when the matching column or WHERE condition is not specified.

- In the absence of a WHERE condition the CARTESIAN JOIN will behave like a CARTESIAN PRODUCT . i.e., the number of rows in the result-set is the product of the number of rows of the two tables.
- In the presence of WHERE condition this JOIN will function like a INNER JOIN.
- Generally speaking, Cross join is similar to an inner join where the join-condition will always evaluate to True

Syntax:

```
SELECT table1.column1 , table1.column2, table2.column1...

FROM table1

CROSS JOIN table2;
```

Example (CARTESIAN JOIN):

- In the below query we will select NAME and Age from Student table and COURSE_ID from StudentCourse table. In the output you can see that each row of the table Student is joined with every row of the table StudentCourse. The total rows in the result-set = 4 * 4 = 16.

```
SELECT Student.NAME, Student.AGE, StudentCourse.COURSE_ID

FROM Student
```

```
CROSS JOIN StudentCourse;
```

6. SELF JOIN: As the name signifies, in SELF JOIN a table is joined to itself. That is, each row of the table is joined with itself and all other rows depending on some conditions. In other words we can say that it is a join between two copies of the same table.

Syntax:

```
SELECT a.coulmn1 , b.column2

FROM table_name a, table_name b

WHERE some_condition;
```

Example(SELF JOIN):

```
SELECT a.ROLL_NO , b.NAME

FROM Student a, Student b

WHERE a.ROLL_NO < b.ROLL_NO;
```