**DML Commands**

DML is an abbreviation for Data Manipulation Language.

Data Manipulation Language or DML represents a collection of programming languages explicitly used to make changes in the database, such as:

1. CRUD operations to create, read, update, and delete data
2. Using the INSERT, SELECT, UPDATE and Delete commands.

DML commands are often part of a more extensive database language, for instance, SQL (Structure Query Language). These DML commands may have a specific syntax to manage data in that language.

DML Commands provide a way to read, update, delete, or merge data precisely. In the beginning, DML commands were part of computer programs only, but with the popularity of SQL, they have now become a part of database management. Data Manipulation Languages (DML) have two primary classifications: Procedural and Non-procedural programming (declarative programming).

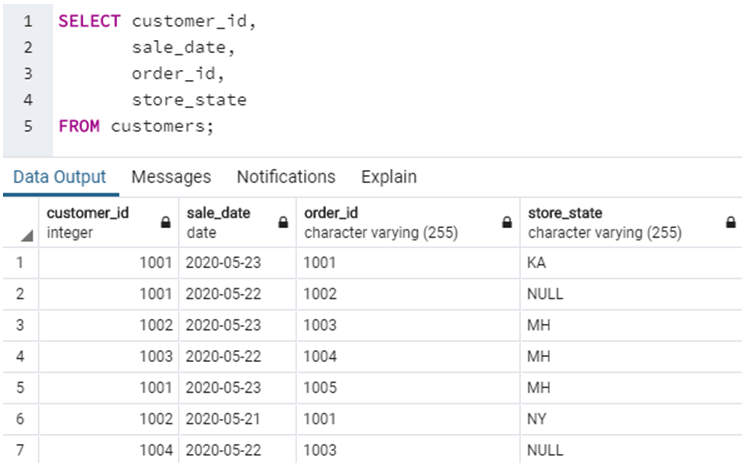
**1. SELECT**

SELECT command or statement in SQL is used to fetch data records from the database table and present it in the form of a result set. It is usually considered as a DQL command but it can also be considered as DML.

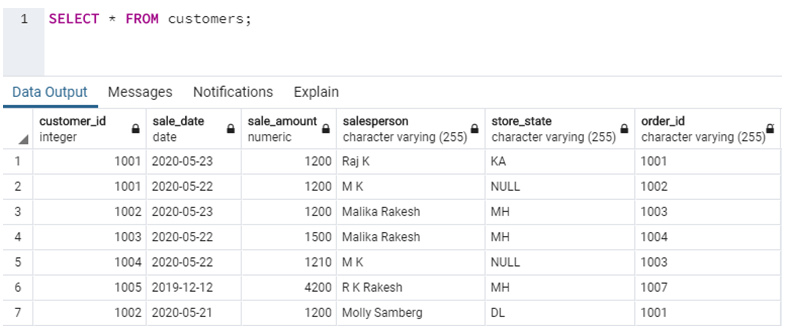
The basic syntax for writing a SELECT query in SQL is as follows :

SELECT column\_name1, column\_name2, …  
FROM table\_name  
WHERE condition\_ expression;

The parameters used in the above syntax are as follows :

* **column\_name1, column\_name2, … :** Specify the column\_names which have to be fetched or selected for the final result set.
* **table\_name:** Specify the name of the database table from which these results have to be fetched.
* **condition\_expression:** Specify the condition expression for filtering records for the final result set.

In this example, we have fetched fields such as customer\_id, sale\_date, order\_id and store\_state from customers table. Next, suppose if we want to fetch all the records from the customers table. This can be achieved by a simple query as shown below.



**2. INSERT**

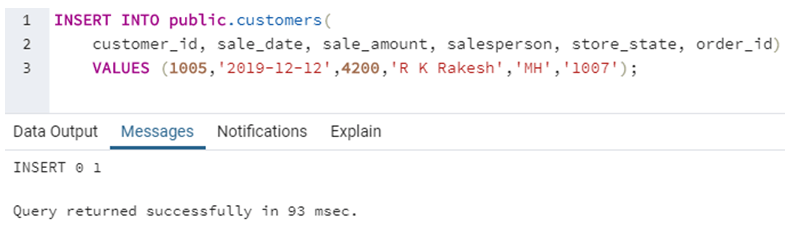
INSERT commands in SQL are used to insert data records or rows in a database table. In an INSERT statement, we specify both the column\_names for which the entry has to be made along with the data value that has to be inserted.

The basic syntax for writing INSERT statements in SQL is as follows :

INSERT INTO table\_name (column\_name\_1, column\_name\_2, column\_name\_3, ...)

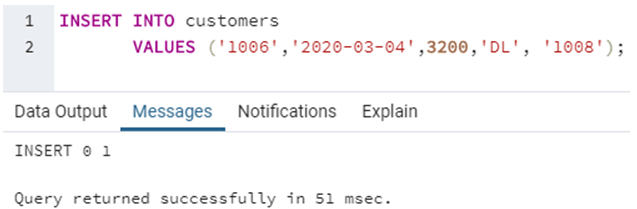
VALUES (value1, value2, value3, ...)

By VALUES, we mean the value of the corresponding columns.



Here we have tried to insert a new row in the Customers table using the INSERT command. The query accepts two sets of arguments, namely field names or column names and their corresponding values.

Suppose if we have to insert values into all the fields of the database table, then we need not specify the column names, unlike the previous query. Follow the following query for further illustration.



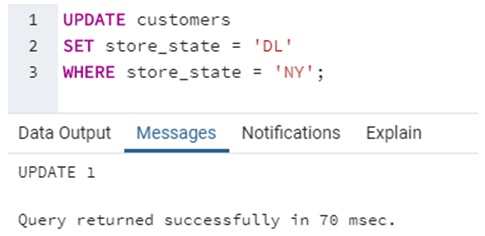
In this example, we have successfully inserted all the values without having to specify the fieldnames.

**3. UPDATE**

UPDATE command or statement is used to modify the value of an existing column in a database table.

The syntax for writing an UPDATE statement is as follows :

UPDATE table\_name  
SET column\_name\_1 = value1, column\_name\_2 = value2, ...  
WHERE condition;



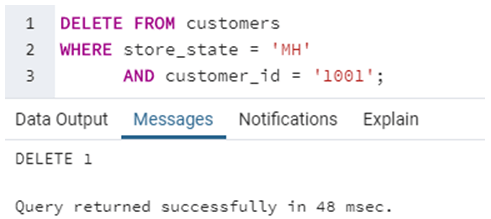
In this example, we have modified the value of store\_state for a record where store\_state was ‘NY’ and set it to a new value ‘DL’.

#### 4. DELETE

DELETE statement in SQL is used to remove one or more rows from the database table. It does not delete the data records permanently. We can always perform a rollback operation to undo a DELETE command. With DELETE statements we can use the WHERE clause for filtering specific rows.

The syntax for writing an DELETE statement is as follows :

DELETE FROM table\_name WHERE condition;



In this example, we have removed a row from the customer’s table where store\_state was ‘MH’ and customer\_id was ‘1001’.

## **DCL**

DCL is the abstract of Data Control Language. Data Control Language includes commands such as GRANT, and is concerned with rights, permissions, and other controls of the database system. DCL is used to grant/revoke permissions on databases and their contents. DCL is simple, but MySQL permissions are a bit complex. DCL is about security. DCL is used to control the database transaction. DCL statements allow you to control who has access to a specific object in your database.

1. GRANT

2. REVOKE

### **GRANT**

It provides the user's access privileges to the database. The MySQL database offers both the administrator and user a great extent of the control options. The administration side of the process includes the possibility for the administrators to control certain user privileges over the MySQL server by restricting their access to an entire database or usage limiting permissions for a specific table. It creates an entry in the security system that allows a user in the current database to work with data in the current database or execute specific statements.

Syntax :

Statement permissions:

GRANT { ALL | statement [ ,...n ] }  
TO security\_account [ ,...n ]

Normally, a database administrator first uses CREATE USER to create an account, then GRANT to define its privileges and characteristics.

For example:

CREATE USER vatsa@'localhost' IDENTIFIED BY 'mypass';

GRANT ALL ON MY\_TABLE TO vatsa@'localhost';

GRANT SELECT ON Users TO vatsa@'localhost';

**REVOKE**

The REVOKE statement enables system administrators and to revoke (back permission) the privileges from MySQL accounts.

**Syntax:**

REVOKE

priv\_type [(column\_list)]  
[, priv\_type [(column\_list)]] ...  
ON [object\_type] priv\_level  
FROM user [, user] ...  
REVOKE ALL PRIVILEGES, GRANT OPTION  
FROM user [, user] ...

For example:

REVOKE INSERT ON \*.\* FROM 'vatsa'@'localhost';

## **SQL Constraints**

SQL Constraints are used to specify the rules for the data in a table. These are used to limit which type of data must be stored in the database, and aims to increase the accuracy and reliability of the data stored in the database.

So, constraints make sure that there is no violation in terms of a transaction of the data, yet there is any violation found; the action gets terminated.

There are two types of constraints which can be applied:

1. **Column-level constraints** – These constraints are applied to a single column
2. **Table-level constraints** – These constraints are the application to the complete table

Moving forward in this article, let us understand the different types of constraints. Also, I am going to consider the following table to help you understand better.

## **Different SQL Constraints available:**

### NOT NULL Constraint

The NOT NULL constraint makes sure that a column cannot have a NULL value. You can use the NOT NULL constraint either while [creating the table](https://www.edureka.co/blog/create-table-in-sql/) database or while modifying it.

#### Example

#### NOT NULL Constraint on CREATE TABLE

Write a query to create the above Students table, where the StudentID and StudentName cannot be NULL.

**CREATE** **TABLE** Students(

StudentID **int** NOT NULL,

StudentName **varchar**(255) NOT NULL,

Age **int**, City **varchar**(255) );

### **NOT NULL Constraint on ALTER TABLE**

Write a query to alter the above Students table, where a new column of DOB must be added, and it should not have any NULL values.

|  |
| --- |
| **ALTER** **TABLE** Students **ADD** **COLUMN** DOB year NOT NULL; |

Moving on in this article on SQL Constraints, let us understand how to use the UNIQUE constraint.

### **UNIQUE Constraint**

The UNIQUE constraint is used to make sure that all the values in a column are unique. You can use the UNIQUE constraint either on multiple columns or on a single column with. Apart from this, you can go forward and use the UNIQUE constraint to modify the existing tables.

**Note:**

1. While creating tables, a PRIMARY KEY constraint automatically has a UNIQUE constraint, to guarantee the uniqueness of a column.
2. A table can have many UNIQUE constraints but can have a single primary key constraint.

#### Example:

#### UNIQUE Constraint on CREATE TABLE

Write a query to create a table Students, with columns StudentID, StudentName, Age and City. Here, the StudentID must be unique for each and every record.

|  |  |
| --- | --- |
| 1  2  3  4 | CREATE TABLE Students (  StudentID int NOT NULL UNIQUE,  StudentName varchar(255)  NOT NULL, Age int, City varchar(255) ); |

#### Name a UNIQUE constraint on multiple columns

To name a unique constraint and to define it for multiple columns you can refer to the following example:

Write a query to create a table Students, with columns StudentID, StudentName, Age and City. Here, the StudentID, and StudentName must be unique for each and every record.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 1  2  3  4  5  6 | CREATE TABLE Students (   |  |  | | --- | --- | |  |  | |  |  |   StudentID int NOT NULL,  StudentName varchar(255) NOT NULL,  Age int,  City varchar(255) CONSTRAINT Stu\_Example  UNIQUE (StudentID,StudentName) ); |

Here, Stu\_Example is the name given to the unique constraint applied on the StudentID and StudentName.

#### UNIQUE Constraint on ALTER TABLE

Write a query to alter the Students table, where a UNIQUE constraint must be added to the StudentID column.

|  |  |
| --- | --- |
| 1 | ALTER TABLE Students ADD UNIQUE (StudentID); |

Similarly, if you want to use the UNIQUE constraint on multiple columns and also name it, you can write a query as follows:

|  |  |
| --- | --- |
|  | ALTER TABLE Students ADD CONSTRAINT Stu\_Example UNIQUE (StudentID,StudentName); |

#### Drop a UNIQUE constraint

To drop the constraint specified on a column, you can use the naming convention which you might have mentioned while adding the constraint.

For example, if we have to write a query to drop the UNIQUE constraint we created above, you can write the query as follows:

|  |  |
| --- | --- |
| 1 | **ALTER** **TABLE** Students **DROP** **CONSTRAINT** Stu\_Example; |

Next in this article on SQL Constraints, let us understand how to use the CHECK constraint.

### CHECK Constraint

The CHECK constraint makes sure that all values in a column satisfy a specific condition.

#### Example:

#### CHECK Constraint on CREATE TABLE

Write a query to create a table Students, with columns StudentID, StudentName, Age and City. Here, the city must be MUmbai.

|  |  |
| --- | --- |
| 1  2  3  4  5 | CREATE TABLE Students (  StudentID int NOT NULL UNIQUE,  StudentName varchar(255) NOT NULL,  Age int,  City varchar(255)CHECK (City==’Mumbai’) ); |

#### CHECK constraint on multiple columns

To use the check constraint on multiple columns you can write a query as below:

Write a query to create a table Students, with columns StudentID, StudentName, Age and City. Here, the City must be Mumbai, and the age of students must be > 19.

|  |  |
| --- | --- |
| 1  2  3  4  5 | CREATE TABLE Students (  StudentID int NOT NULL,  StudentName varchar(255) NOT NULL,  Age int,  City varchar(255)CHECK (City==’Mumbai’ AND Age>19)); |

Similarly, you can use the CHECK constraint with the ALTER TABLE comm

#### CHECK Constraint on ALTER TABLE

Write a query to alter the Students table, where a CHECK constraint must be added to the City column. Here, the city must be Mumbai.

|  |  |
| --- | --- |
| 1 | **ALTER** **TABLE** Students **ADD** **CHECK** (City=='Mumbai'); |

Similarly, if you want to use the CHECK constraint by giving it a name you can write a query as follows:

|  |  |
| --- | --- |
| 1 | **ALTER** **TABLE** Students **ADD** **CONSTRAINT** StuCheckExample **CHECK**(City=='Mumbai'); |

#### Drop a CHECK constraint

To drop the constraint specified on a column, you can use the naming convention which you might have mentioned while adding the constraint.

For example, if we have to write a query to drop the CHECK constraint we created above, you can write the query as follows:

|  |  |
| --- | --- |
| 1 | **ALTER** **TABLE** Students **DROP** **CONSTRAINT** StuCheckExample; |

### **DEFAULT Constraint**

The DEFAULT constraint is used to mention a set of default values for a column when no value is specified. Similar to that of the other constraints, we can use this constraint on the CREATE and ALTER table command.

#### Example

Write a query to create a table Students, with columns StudentID, StudentName, Age and City. Also when there is no value inserted in the City column, automatically Delhi must be included.

|  |  |
| --- | --- |
| 1  2  3  4  5 | CREATE TABLE Students (  StudentID int NOT NULL,  StudentName varchar(255) NOT NULL,  Age int,  City varchar(255)DEFAULT ‘Delhi’); |

#### DEFAULT Constraint on ALTER TABLE

To use the DEFAULT constraint with the [ALTER TABLE command](https://www.edureka.co/blog/alter-table/), you can write a query as follows:

|  |  |
| --- | --- |
|  | ALTER TABLE Students ADD CONSTRAINT StuDefauExample DEFAULT 'Mumbai' FOR  City; |

#### ****Drop a DEFAULT constraint****

To drop the DEFAULT constraint you can use the ALTER TABLE command as follows:

|  |  |
| --- | --- |
| 1 | **ALTER TABLE Students ALTER COLUMN City DROP DEFAULT;** |

## **Primary Key Constraint**

A Primary key uniquly identifies each row in a table. It cannot accept null and duplicate data. One or more of the columns of a table can contain a Primary key.  
  
**Column Level**  
  
**Syntax**

Create Table Table\_Name

(

   Column\_Name Datatype Constraint Constraint\_Name Primary Key,

) 

**Example**

Create Table Employee

(

   IId int constraint Const\_primary\_IId primary key,

   Name nvarchar(50)

) 

**Table Level  
  
Syntax**

Alter Table Table\_Name

Add constraint Constraint\_Name Primary Key(Column\_Name) 

**Example**

Alter Table Employee

Add constraint Constraint\_Name Primary Key(IId)

## **Foreign Key Constraint**

A Foreign Key is a field in a database table that is a Primary key in another table. A Foreign key creates a relation between two tables. The first table contains a primary key and the second table contains a foreign key.  
  
**Column Level  
  
Syntax**

Create Table Table\_Name

(

   Column\_Name Datatype Constraint Constraint\_Name References Reference\_Table\_Name(Reference\_Column\_Name)

)

**Example**

Create Table Employee\_

(

   IId int constraint Cons\_Reference References My\_Constraint(IId),

   Age int,

   Salary int

) 

**Table Level  
  
Syntax**

ALTER TABLE Table\_Name

ADD CONSTRAINT Constraint\_Name FOREIGN KEY(Column\_Name)

REFERENCES Reference\_Table (Column\_Name) 

**Example**

ALTER TABLE Employee\_ 

ADD CONSTRAINT Cons\_Emp\_Foreign FOREIGN KEY(IId)

REFERENCES My\_Constraint(IId)

## **The SQL ORDER BY Keyword**

The ORDER BY keyword is used to sort the result-set in ascending or descending order.

The ORDER BY keyword sorts the records in ascending order by default. To sort the records in descending order, use the DESC keyword.

### **ORDER BY Syntax**

SELECT column1, column2, ...  
FROM table\_name  
ORDER BY column1, column2, ... ASC|DESC;

In this example, we have a table called *customers* with the following data:

| customer\_id | last\_name | first\_name | favorite\_website |
| --- | --- | --- | --- |
| 4000 | Jackson | Joe | techonthenet.com |
| 5000 | Smith | Jane | digminecraft.com |
| 6000 | Ferguson | Samantha | bigactivities.com |
| 7000 | Reynolds | Allen | checkyourmath.com |
| 8000 | Anderson | Paige | NULL |
| 9000 | Johnson | Derek | techonthenet.com |

**SQL Statement**

SELECT \*

FROM customers

ORDER BY last\_name;

| customer\_id | last\_name | first\_name | favorite\_website |
| --- | --- | --- | --- |
| 8000 | Anderson | Paige | NULL |
| 6000 | Ferguson | Samantha | bigactivities.com |
| 4000 | Jackson | Joe | techonthenet.com |
| 9000 | Johnson | Derek | techonthenet.com |
| 7000 | Reynolds | Allen | checkyourmath.com |
| 5000 | Smith | Jane | digminecraft.com |

## **ORDER BY DESC Example**

In this example, we have a table called suppliers with the following data:

| supplier\_id | supplier\_name | city | state |
| --- | --- | --- | --- |
| 100 | Microsoft | Redmond | Washington |
| 200 | Google | Mountain View | California |
| 300 | Oracle | Redwood City | California |
| 400 | Kimberly-Clark | Irving | Texas |
| 500 | Tyson Foods | Springdale | Arkansas |
| 600 | SC Johnson | Racine | Wisconsin |
| 700 | Dole Food Company | Westlake Village | California |
| 800 | Flowers Foods | Thomasville | Georgia |
| 900 | Electronic Arts | Redwood City | California |

**Query:**

SELECT \*

FROM suppliers

WHERE supplier\_id > 400

ORDER BY supplier\_id DESC;

| supplier\_id | supplier\_name | city | state |
| --- | --- | --- | --- |
| 900 | Electronic Arts | Redwood City | California |
| 800 | Flowers Foods | Thomasville | Georgia |
| 700 | Dole Food Company | Westlake Village | California |
| 600 | SC Johnson | Racine | Wisconsin |
| 500 | Tyson Foods | Springdale | Arkansas |

# **Distinct Keyword**

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

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

**Syntax**

The basic syntax of DISTINCT keyword to eliminate the 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 |

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

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 |

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

# **Row Number**

Row number is the most common **ranking function** used in SQL Server. The ROW\_NUMBER() function **generates a sequential number for each row within a partition** in the resultant output. In each partition, the first-row number begins with 1. We must always use the **ORDER BY** clause to ensure that the numbers are assigned in the proper sequence. The values returned by this function are of **BIG INT** data type. This function comes with SQL Server 2005 and later MS SQL versions.

### **Syntax**

The following are the syntax that illustrates the ROW\_NUMBER() function:

ROW\_NUMBER() OVER (

  [PARTITION BY partition\_expression, ... ]

ORDER BY sort\_expression [ASC | DESC], ...

)

**ROW NUMBER() function's syntax:**

**OVER**

It's important to understand this clause that specifies the window or set of rows that the window function operates. The PARTITION BY and ORDER BY are the two possible clauses of the OVER clause. The OVER clause's ORDER BY expression is supported when the rows must come in a specific order for the function to execute.

**PARTITION BY**

* It is an optional clause that splits the result set into partitions (groups of rows). Then ROW NUMBER() function is applied to each partition and assigns each partition's rank number separately.
* If we omit the partition by clause, the ROW\_NUMBER function will treat the whole result as a single partition and provide ranking in the top to bottom order.

**ORDER BY**

This clause allows us to sort the rows of the result set within each partition. It is a required clause because the ROW\_NUMBER() function is order-dependent.

### **Example**

Let us understand how the ROW\_NUMBER function works in the SQL Server table with an example. First, we will create a table named "**Persons**" using the below statement:

**CREATE** **TABLE** Persons (

  person\_name **varchar**(45) NOT NULL,

  product **varchar**(45) **DEFAULT** NULL,

  country **varchar**(25) **DEFAULT** NULL,

  price **float**,

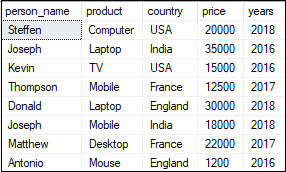
  years **int** NOT NULL

);

 we will add some records into this table using the below statement:

1. INSERT INTO Persons (person\_name, product, country, price, years)
2. VALUES ('Steffen', 'Computer', 'USA', 20000.00, 2018),
3. ('Joseph', 'Laptop', 'India', 35000.00, 2016),
4. ('Kevin', 'TV', 'USA', 15000.00, 2016),
5. ('Thompson', 'Mobile', 'France', 12500.00, 2017),
6. ('Donald', 'Laptop', 'England', 30000.00, 2018),
7. ('Joseph', 'Mobile', 'India', 18000.00, 2018),
8. ('Matthew', 'Desktop', 'France', 22000.00, 2017),
9. ('Antonio', 'Mouse', 'England', 1200.00, 2016);

Next, verify the data using the SELECT statement. We will get the below output:



**1. Simple ROW\_NUMBER() Example**

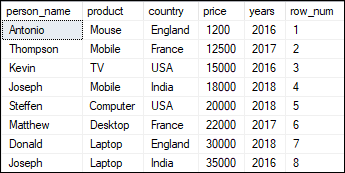
The following statement displays the person's detail and adds a sequential integer number to each row by using the ROW\_NUMBER():

**SELECT** \*, ROW\_NUMBER()

OVER (**ORDER** **BY** price) **AS** row\_num

**FROM** Persons;

Here, we have not specified the PARTITION BY clause so that the ROW\_NUMBER() function will treat the whole result set as a single partition. After execution of the statement, we will get the following output:



1. **ROW\_NUMBER() over Partitions Example**

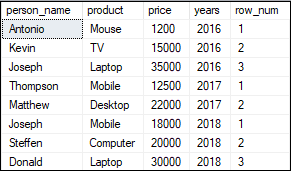
The ROW NUMBER() function is used in this example provides a sequential number to each record within a partition in a table. It always reinitialized the row number when the year changes:

SELECT person\_name, product, price, years,

ROW\_NUMBER() OVER ( PARTITION BY years ORDER BY price) AS row\_num

FROM Persons;

Here, we have used the PARTITION BY clause that divides the **'Persons'** table into partitions based on the **'years'** column. After execution, we will get the below output:



**ROW\_NUMBER() Example for Pagination**

We can also use the ROW\_NUMBER() function for pagination. **For example**, if we want to get all of a person's information in an application by pages, we'll first assign each row a sequential number using the ROW\_NUMBER() function. Second, sort the rows by the requested page.

The following statement explains it more clearly:

SELECT \* FROM ( SELECT ROW\_NUMBER()

OVER (ORDER BY price) AS row\_num, person\_name, product, price

FROM Persons) P

WHERE row\_num > 3 AND row\_num <= 6;

It will give the following output:

