

Introduction to SQL:

Overview of the SQL Query Language, SQL Data Definition, Basic Structure of SQL Queries, Additional Basic Operations, Set Operations, Null Values, Aggregate Functions, Nested Sub queries, Modification of the Database.

Intermediate and Advanced SQL:

Join Expressions, Views , Integrity Constraints, SQL Data Types, Authorization. Functions and Procedures, Triggers.

History of SQL

"A Relational Model of Data for Large Shared Data Banks" was a paper which was published by the great computer scientist "E.F. Codd" in 1970.

The IBM researchers Raymond Boyce and Donald Chamberlin originally developed the SEQUEL (Structured English Query Language) after learning from the paper given by E.F. Codd. They both developed the SQL at the San Jose Research laboratory of IBM Corporation in 1970.

At the end of the 1970s, relational software Inc. developed their own first SQL using the concepts of E.F. Codd, Raymond Boyce, and Donald Chamberlin. This SQL was totally based on RDBMS. Relational Software Inc., which is now known as Oracle Corporation, introduced the Oracle V2 in June 1979, which is the first implementation of SQL language.

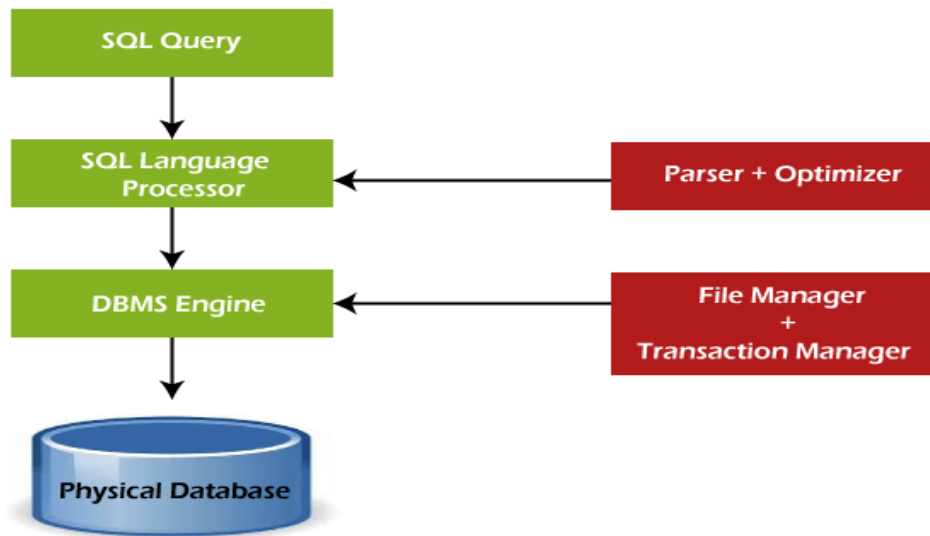
Process of SQL

When we are executing the command of SQL on any Relational database management system, then the system automatically finds the best routine to carry out our request, and the SQL engine determines how to interpret that particular command.

Structured Query Language contains the following four components in its process:

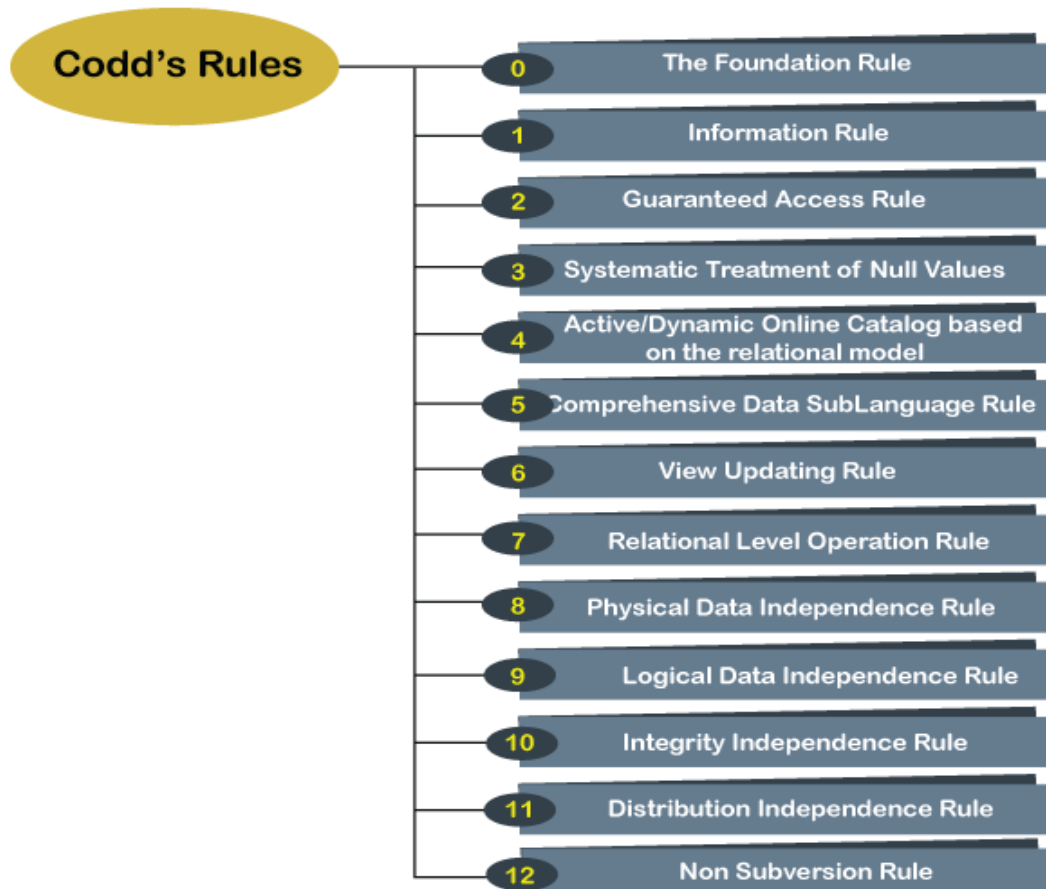
- Query Dispatcher
- Optimization Engines
- Classic Query Engine
- SQL Query Engine, etc.

A classic query engine allows data professionals and users to maintain non-SQL queries. The architecture of SQL is shown in the following diagram:



12 Codd's Rules

The rules were developed by Dr. Edgar F. Codd (E.F. Codd) in 1974, who has vast research knowledge on the Relational Model of database Systems. Codd presents his 12 rules for a database to test the concept of DBMS against his relational model, and if a database follows the rule, it is called a true relational database (RDBMS).



Rule 0: The Foundation Rule

The database must be in relational form. So that the system can handle the database through its relational capabilities.

Rule 1: Information Rule

A database contains various information, and this information must be stored in each cell of a table in the form of rows and columns.

Rule 2: Guaranteed Access Rule

Every single or precise data (atomic value) may be accessed logically from a relational database using the combination of primary key value, table name, and column name.

Rule 3: Systematic Treatment of Null Values

This rule defines the systematic treatment of Null values in database records. The null value has various meanings in the database, like missing the data, no value in a cell, inappropriate information, unknown data and the primary key should not be null.

Rule 4: Active/Dynamic Online Catalog based on the relational model

It represents the entire logical structure of the descriptive database that must be stored online and is known as a database dictionary. It authorizes users to access the database and implement a similar query language to access the database.

Rule 5: Comprehensive Data SubLanguage Rule

The relational database supports various languages, and if we want to access the database, the language must be the explicit, linear or well-defined syntax, character strings and supports the comprehensive: data definition, view definition, data manipulation, integrity constraints, and limit transaction management operations. If the database allows access to the data without any language, it is considered a violation of the database.

Rule 6: View Updating Rule

All views table can be theoretically updated and must be practically updated by the database systems.

Rule 7: Relational Level Operation (High-Level Insert, Update and delete) Rule

A database system should follow high-level relational operations such as insert, update, and delete in each level or a single row. It also supports union, intersection and minus operation in the database system.

Rule 8: Physical Data Independence Rule

All stored data in a database or an application must be physically independent to access the database. Each data should not depend on other data or an application. If data is updated or the physical structure of the database is changed, it will not show any effect on external applications that are accessing the data from the database.

Rule 9: Logical Data Independence Rule

It is similar to physical data independence. It means, if any changes occurred to the logical level (table structures), it should not affect the user's view (application). For example, suppose a table either split into two tables, or two table joins to create a single table, these changes should not be impacted on the user view application.

Rule 10: Integrity Independence Rule

A database must maintain integrity independence when inserting data into table's cells using the SQL query language. All entered values should not be changed or rely on any external factor or application to maintain integrity. It is also helpful in making the database-independent for each front-end application.

Rule 11: Distribution Independence Rule

The distribution independence rule represents a database that must work properly, even if it is stored in different locations and used by different end-users. Suppose a user accesses the database through an application; in that case, they should not be aware that another user uses particular data, and the data they always get is only located on one site. The end users can access the database, and these access data should be independent for every user to perform the SQL queries.

Rule 12: Non Subversion Rule

The non-submersion rule defines RDBMS as a SQL language to store and manipulate the data in the database. If a system has a low-level or separate language other than SQL to access the database system, it should not subvert or bypass integrity to transform data.

SQL Data Types

Data types are used to represent the nature of the data that can be stored in the database table.

Data types mainly classified into three categories for every database.

- String Data types
- Numeric Data types
- Date and time Data types

MySQL String Data Types

CHAR(Size)	It is used to specify a fixed length string that can contain numbers, letters, and special characters. Its size can be 0 to 255 characters. Default is 1.
VARCHAR(Size)	It is used to specify a variable length string that can contain numbers, letters, and special characters. Its size can be from 0 to 65535 characters.
BLOB(size)	It is used for BLOBs (Binary Large Objects). It can hold up to 65,535 bytes.

○ MySQL Numeric Data Types

INT(size)	It is used for the integer value. The size parameter specifies the max display width that is 255.
INTEGER(size)	It is equal to INT(size).
FLOAT(size, d)	It is used to specify a floating point number. Its size parameter specifies the total number of digits. The number of digits after the decimal point is specified by d parameter.
FLOAT(p)	It is used to specify a floating point number. MySQL used p parameter to determine whether to use FLOAT or DOUBLE.
BOOL	It is used to specify Boolean values true and false. Zero is considered as false, and nonzero values are considered as true.

○ MySQL Date and Time Data Types

DATE	It is used to specify date format YYYY-MM-DD. Its supported range is from '1000-01-01' to '9999-12-31'.
DATETIME(fsp)	It is used to specify date and time combination. Its format is YYYY-MM-DD hh:mm:ss. Its supported range is from '1000-01-01 00:00:00' to 9999-12-31 23:59:59'.
TIMESTAMP(fsp)	It is used to specify the timestamp. Its value is stored as the number of seconds since the Unix epoch('1970-01-01 00:00:00' UTC). Its format is YYYY-MM-DD hh:mm:ss. Its supported range is from '1970-01-01 00:00:01' UTC to '2038-01-09 03:14:07' UTC.
TIME(fsp)	It is used to specify the time format. Its format is hh:mm:ss. Its supported range is from '-838:59:59' to '838:59:59'
YEAR	It is used to specify a year in four-digit format. Values allowed in four digit format from 1901 to 2155, and 0000.

SQL Rules

The rules for writing SQL queries are given below:

- A ';' is used to end SQL statements.
- Statements may be split across lines, but keywords may not.
- Identifiers, operator names, and literals are separated by one or more spaces or other delimiters.
- A comma (,) separates parameters without a clause.
- A space separates a clause.
- Reserved words cannot be used as identifiers unless enclosed with double quotes.
- Identifiers can contain up to 30 characters.
- Identifiers must start with an alphabetic character.
- Characters and date literals must be enclosed within single quotes.
- Numeric literals can be represented by simple values.
- Comments may be enclosed between /* and */ symbols and maybe multi-line.

What are SQL commands?

Developers use structured query language (SQL) commands, which are specific keywords or SQL statements, to work with data stored in relational databases. The following are categories for SQL commands.

1. Data Definition Language

SQL commands used to create the database structure are known as data definition language (DDL). Based on the needs of the business, database engineers create and modify database objects using DDL. The CREATE command, for instance, is used by the database engineer to create database objects like tables, views, and indexes.

Command	Description
CREATE	Creates a new table, a view of a table, or other object in the database.
ALTER	Modifies an existing database object, such as a table
DROP	Deletes an entire table, a view of a table, or other objects in the database

2. Data Manipulation Language

A relational database can be updated with new data using data manipulation language (DML) statements. The INSERT command, for instance, is used by an application to add a new record to the database.

Command	Description
SELECT	Retrieves certain records from one or more tables.
INSERT	Creates a record.
UPDATE	Modifies records.
DELETE	Deletes records.

3. Data Control language

Data control language (DCL) is a programming language used by database administrators to control or grant other users access to databases. For instance, they can allow specific applications to manipulate one or more tables by using the GRANT command.

Command	Description
GRANT	Gives a privilege to the user.
REVOKE	Takes back privileges granted by the user.

4. Transaction Control Language

To automatically update databases, the relational engine uses transaction control language (TCL). For instance, the database can reverse a mistaken transaction using the ROLLBACK command.

Command	Description
COMMIT	Permanently saves the changes made in the current transaction
ROLLBACK	Cancels all changes made in the current transaction, returning the command to its previous state. It's executed when a failure occurs during the transaction.

Basic Structure of SQL Queries?

SQL syntax is a unique set of rules and guidelines to be followed while writing SQL statements.

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 (;).

Case Sensitivity

The most important point to be noted here is that SQL is case insensitive, which means **SELECT** and **Select** have same meaning in SQL statements. Whereas, MySQL makes difference in table names. So, if you are working with MySQL, then you need to give table names as they exist in the database.

SQL Table

Let us consider a table with the name CUSTOMERS shown below, and use it as a reference to demonstrate all the SQL Statements on the same.

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	Hyderabad	4500.00
7	Muffy	24	Indore	10000.00

SQL Statements

All the SQL statements require a **semicolon (;)** at the end of each statement. Semicolon is the standard way to separate different SQL statements which allows to include multiple SQL statements in a single line.

SQL CREATE DATABASE Statement

To store data within a database, you first need to create it. This is necessary to individualize the data belonging to an organization.

You can create a database using the following syntax

```
CREATE DATABASE database_name;
```

Let us try to create a sample database **sampleDB** in SQL using the **CREATE DATABASE** statement

```
CREATE DATABASE sampleDB
```

SQL USE Statement

Once the database is created, it needs to be used in order to start storing the data accordingly. Following is the syntax to change the current location to required database

```
USE database_name;
```

We can set the previously created sampleDB as the default database by using the **USE** statement in SQL

```
USE sampleDB;
```

SQL DROP DATABASE Statement

If a database is no longer necessary, you can also delete it. To delete/drop a database, use the following syntax

```
DROP DATABASE database_name;
```

You can also drop the sampleDB database by using the **DROP DATABASE** statement in SQL

```
DROP DATABASE sampleDB;
```

SQL CREATE TABLE Statement

In an SQL driven database, the data is stored in a structured manner, i.e. in the form of tables. To create a table, following syntax is used –

```
CREATE TABLE table_name(  
    column1 datatype,  
    column2 datatype,  
    column3 datatype,  
    ....  
    columnN datatype,  
    PRIMARY KEY( one or more columns )  
);
```

The following code block is an example, which creates a CUSTOMERS table given above, with an ID as a primary key and NOT NULL are the constraints showing that these fields cannot be NULL while creating records in this 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)  
);
```

SQL DESC Statement

Every table in a database has a structure of its own. To display the structure of database tables, we use the DESC statements. Following is the syntax –

```
DESC table_name;
```

The DESC Statement, however, only works in few RDBMS systems; hence, let us see an example by using DESC statement in the MySQL server –

```
DESC CUSTOMERS;
```

SQL INSERT INTO Statement

The SQL INSERT INTO Statement is used to insert data into database tables. Following is the syntax

```
INSERT INTO table_name( column1, column2....columnN)  
VALUES ( value1, value2....valueN);
```

The following example statements would create seven records in the empty CUSTOMERS table.

```
INSERT INTO CUSTOMERS VALUES  
(1, 'Ramesh', 32, 'Ahmedabad', 2000.00 ),  
(2, 'Khilan', 25, 'Delhi', 1500),  
(3, 'kaushik', 23, 'Kota', 2000),  
(4, 'Chaitali', 25, 'Mumbai', 6500),  
(5, 'Hardik', 27, 'Bhopal', 8500),  
(6, 'Komal', 22, 'Hyderabad', 4500),  
(7, 'Muffy', 24, 'Indore', 10000);
```

SQL SELECT Statement

In order to retrieve the result-sets of the stored data from a database table, we use the SELECT statement. Following is the syntax

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

To retrieve the data from CUSTOMERS table, we use the SELECT statement as shown below.

```
SELECT * FROM CUSTOMERS;
```

SQL UPDATE Statement

When the stored data in a database table is outdated and needs to be updated without having to delete the table, we use the UPDATE statement. Following is the syntax –

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

To see an example, 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;
```

SQL DELETE Statement

Without deleting the entire table from the database, you can also delete a certain part of the data by applying conditions. This is done using the DELETE FROM statement. Following is the syntax –

```
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;
```

SQL DROP TABLE Statement

To delete a table entirely from a database when it is no longer needed, following syntax is used

```
DROP TABLE table_name;
```

This query will drop the CUSTOMERS table from the database.

```
DROP TABLE CUSTOMERS;
```

SQL TRUNCATE TABLE Statement

The TRUNCATE TABLE statement is implemented in SQL to delete the data of the table but not the table itself. When this SQL statement is used, the table stays in the database like an empty table. Following is the syntax

```
TRUNCATE TABLE table_name;
```

Following query delete all the records of the CUSTOMERS table

```
TRUNCATE TABLE CUSTOMERS;
```

SQL ALTER TABLE Statement

The ALTER TABLE statement is used to alter the structure of a table. For instance, you can add, drop, and modify the data of a column using this statement. Following is the syntax

```
ALTER TABLE table_name  
{ADD|DROP|MODIFY} column_name {data_type};
```

Following is the example to ADD a New Column to the CUSTOMERS table using ALTER TABLE command

```
ALTER TABLE CUSTOMERS ADD GENDER char(1);
```

SQL ALTER TABLE Statement (Rename)

The ALTER TABLE statement is also used to change the name of a table as well. Use the syntax below

```
ALTER TABLE table_name RENAME TO new_table_name;
```

Following is the example to RENAME the CUSTOMERS table using ALTER TABLE command

```
ALTER TABLE CUSTOMERS RENAME TO NEW_CUSTOMERS;
```

SQL DISTINCT Clause

The DISTINCT clause in a database is used to identify the non-duplicate data from a column. Using the SELECT DISTINCT statement, you can retrieve distinct values from a column. Following is the syntax

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

As an example, let us use the **DISTINCT** keyword with a SELECT query. The repetitive salary 20000 will only be retrieved once and the other record is ignored.

```
SELECT DISTINCT SALARY FROM EMPLOYEE ORDER BY SALARY;
```

SQL WHERE Clause

The WHERE clause is used to filter rows from a table by applying a condition. Following is the syntax to retrieve filtered rows from a table

```
SELECT column1, column2....columnN  
FROM table_name  
WHERE CONDITION;
```

The following query is an example to fetch all the records from CUSTOMERS table where the salary is greater than 2000, using the SELECT statement

```
SELECT ID, NAME, SALARY  
FROM EMPLOYEE  
WHERE SALARY > 20000;
```

SQL AND/OR Operators

The AND/OR Operators are used to apply multiple conditions in the WHERE clause. Following is the syntax

```
SELECT column1, column2....columnN  
FROM table_name  
WHERE CONDITION-1 {AND|OR} CONDITION-2;
```

The following query is an example to fetch all the records from CUSTOMERS table where the salary is greater than 2000 AND age is less than 25, using the SELECT statement –

```
SELECT ID, NAME, SALARY FROM EMPLOYEE WHERE SALARY > 2000  
AND age < 25;
```

SQL IN Clause

The IN Operator is used to check whether the data is present in the column or not, using the WHERE clause. Following is the syntax

```
SELECT column1, column2....columnN  
FROM table_name  
WHERE column_name IN (val-1, val-2,...val-N);
```

For an example, we want to display records with NAME equal to 'Khilan', 'Hardik' and 'Muffy' (string values) using IN operator as follows

```
SELECT * FROM EMPLOYEE  
WHERE NAME IN ('Khilan', 'Hardik', 'Muffy');
```

SQL BETWEEN Clause

The BETWEEN Operator is used to retrieve the values from a table that fall in a certain range, using the WHERE clause. Following is the syntax

```
SELECT column1, column2....columnN  
FROM table_name  
WHERE column_name BETWEEN val-1 AND val-2;
```

Let us try to the BETWEEN operator to retrieve CUSTOMERS records whose AGE is between 20 and 25.

```
SELECT * FROM EMPLOYEE WHERE AGE BETWEEN 30 AND 45;
```

SQL LIKE Clause

The LIKE Operator is used to retrieve the values from a table that match a certain pattern, using the WHERE clause. Following is the syntax

```
SELECT column1, column2....columnN  
FROM table_name  
WHERE column_name LIKE { PATTERN };
```

As an example, let us try to display all the records from the CUSTOMERS table, where the SALARY starts with 2000.

```
SELECT * FROM EMPLOYEE WHERE SALARY LIKE '2000%';
```

SQL ORDER BY Clause

The ORDER BY Clause is used to arrange the column values in a given/specified order. Following is the syntax

```
SELECT column1, column2....columnN  
FROM table_name  
WHERE CONDITION  
ORDER BY column_name {ASC|DESC};
```

In the following example we are trying to sort the result in an ascending order by the alphabetical order of customer names

```
SELECT * FROM EMPLOYEE ORDER BY NAME ASC;
```

SQL GROUP BY Clause

The GROUP BY Clause is used to group the values of a column together. Following is the syntax

```
SELECT SUM(column_name)  
FROM table_name  
WHERE CONDITION  
GROUP BY column_name;
```


We are trying to group the customers by their age and calculate the total salary for each age group using the following query

```
SELECT ID,NAME, AGE, SUM(SALARY)
AS TOTAL_SALARY FROM EMPLOYEE
GROUP BY AGE;
```

SQL HAVING Clause

The HAVING clause is also used to filter a group of rows by applying a condition. Following is the syntax

```
SELECT SUM(column_name)
FROM table_name
WHERE CONDITION
GROUP BY column_name
HAVING (condition);
```

In the following example, we are trying to retrieve all records from the EMPLOYEE table where the sum of their salary is greater than 50000

```
SELECT ID,NAME, AGE, SUM(SALARY) AS
TOTAL_SALARY FROM CUSTOMERS GROUP BY
AGE HAVING TOTAL_SALARY >=50000 ;
```

SQL Set Operation

The SQL Set operation is used to combine the two or more SQL SELECT statements.

Types of Set Operation

1. Union
2. UnionAll
3. Intersect
4. Minus



1. Union

- The SQL Union operation is used to combine the result of two or more SQL SELECT queries.
- In the union operation, all the number of datatype and columns must be same in both the tables on which UNION operation is being applied.
- The union operation eliminates the duplicate rows from its resultset.

Syntax

```
SELECT column_name FROM table1  
UNION  
SELECT column_name FROM table2;
```

Example:

The First table

ID	NAME
1	Jack
2	Harry
3	Jackson

The Second table

ID	NAME
3	Jackson
4	Stephan
5	David

Union SQL query will be:

```
SELECT * FROM First  
UNION  
SELECT * FROM Second;
```

The resultset table will look like:

ID	NAME
1	Jack
2	Harry
3	Jackson
4	Stephan
5	David

2. Union All

Union All operation is equal to the Union operation. It returns the set without removing duplication and sorting the data.

Syntax:

```
SELECT column_name FROM table1  
UNION ALL  
SELECT column_name FROM table2;
```

Example: Using the above First and Second table.

Union All query will be like:

```
SELECT * FROM First  
UNION ALL  
SELECT * FROM Second;
```

The resultset table will look like:

ID	NAME
1	Jack
2	Harry
3	Jackson
3	Jackson
4	Stephan
5	David

3. Intersect

- It is used to combine two SELECT statements. The Intersect operation returns the common rows from both the SELECT statements.
- In the Intersect operation, the number of datatype and columns must be the same.
- It has no duplicates and it arranges the data in ascending order by default.

Syntax

```
SELECT column_name FROM table1  
INTERSECT  
SELECT column_name FROM table2;
```

Example:

Using the above First and Second table.

Intersect query will be:

```
SELECT * FROM First  
INTERSECT  
SELECT * FROM Second;
```

The resultset table will look like:

ID	NAME
3	Jackson

4. Minus

- It combines the result of two SELECT statements. Minus operator is used to display the rows which are present in the first query but absent in the second query.
- It has no duplicates and data arranged in ascending order by default.

Syntax:

```
SELECT column_name FROM table1  
MINUS  
SELECT column_name FROM table2;
```

Example

Using the above First and Second table.

Minus query will be:

```
SELECT * FROM First  
MINUS  
SELECT * FROM Second;
```

The resultset table will look like:

ID	NAME
1	Jack
2	Harry

NULL and NOT NULL in SQL

NULL and NOT NULL restrictions are critical in database design. These constraints control the presence or absence of values in a database table's columns, which affects data integrity and query results.

Null Constraint

A NULL value in SQL denotes the lack of data in a column. It means that the data needs to be included, unknown, or undefined. Columns that accept NULL values are deemed nullable. Most columns in SQL databases allow NULL values by default unless the NOT NULL constraint is used to specify otherwise.

Example of Null Constraint:

Consider this scenario: you have a table named Students that stores student information. Let's build this table with some columns, including one that accepts NULL values:

```
CREATE TABLE Students (  
  StudentID INT PRIMARY KEY,  
  FirstName VARCHAR(50) NOT NULL,  
  LastName VARCHAR(50) NOT NULL,  
  Age INT,  
  Address VARCHAR(100)  
);
```

In this example:

- StudentID, FirstName, and LastName columns are specified as NOT NULL, ensuring that every student record must have values for these columns.
- Age and Address columns allow NULL values, indicating that age and address information may not be available for all students.

Inserting data with null values:

Let's add some example data to the Student's table.

```
INSERT INTO Students (StudentID, FirstName, LastName, Age, Address)  
VALUES  
(1, 'John', 'Doe', 20, '123 Main St'),  
(2, 'Jane', 'Smith', NULL, NULL),  
(3, 'Michael', 'Johnson', 22, NULL);
```

In this data, insert

- The first record has values for every column.
- The second entry has NULL values for Age and Address.
- The third entry has a NULL value in the Address field.

Querying the Student's table

SELECT * FROM Students;

This query will return the following output:

StudentID	FirstName	LastName	Age	Address
1	John	Doe	20	123 Main St
2	Jane	Smith	NULL	NULL
3	Michael	Johnson	22	NULL

Explanation for the output:

- Each row indicates a student's record.
- The StudentID, FirstName, LastName, Age, and Address columns are shown.
- The Age column accepts NULL values hence, the second and third records contain NULL Age values.

Not Null Constraint

In contrast, the NOT NULL constraint enforces the requirement that a column cannot contain NULL values. Every row in a table must include a value for that particular field. Columns that are marked as NOT NULL are considered non-nullable.

Example of NOT NULL Constraint:

Let's modify the Students table to make the Age column NOT NULL:

```
ALTER TABLE Students  
MODIFY COLUMN Age INT NOT NULL;
```

The Age field will no longer accept NULL entries, guaranteeing that each student record has an age indicated.

Attempting to make this change will result in an error since the second record has a NULL value in the Age field. However, let us assume we omit this step and continue to query the Students table:

Querying the Students table after setting the field to NOT NULL:

```
SELECT * FROM Students;
```

The preceding error prevents the query from executing correctly. Nonetheless, after the problem is fixed by either updating the second item with a proper age or deleting the record, the query will return the following results:

StudentID	FirstName	LastName	Age	Address
1	John	Doe	20	123 Main St
3	Michael	Johnson	22	NULL

Explanation for the output:

- The second record, which had a NULL value in the Age column, has been changed or deleted to meet the NOT NULL requirement.
- Only records with valid ages are saved in the table.

Practical Implications

The use of NULL and NOT NULL constraints has major ramifications for database architecture, data integrity, and query results.

- **Data Integrity:** NOT NULL restrictions guarantee that critical data is always present, lowering the possibility of data inconsistency and mistakes. NULLs, on the other hand, provide greater flexibility when dealing with missing or optional information, but they must be handled carefully to avoid unexpected query results.
- **Query Results:** SQL queries that include columns with NULL values require particular treatment. If NULLs are not correctly handled, operations like comparisons and arithmetic might provide unexpected results. SQL has methods like IS NULL and IS NOT NULL that deal particularly with NULL data in queries.
- **Indexing and Performance:** NULL values might influence query and index performance. Some database systems evaluate NULL values differently than non-NULL values, which may influence index utilization and query optimization tactics.

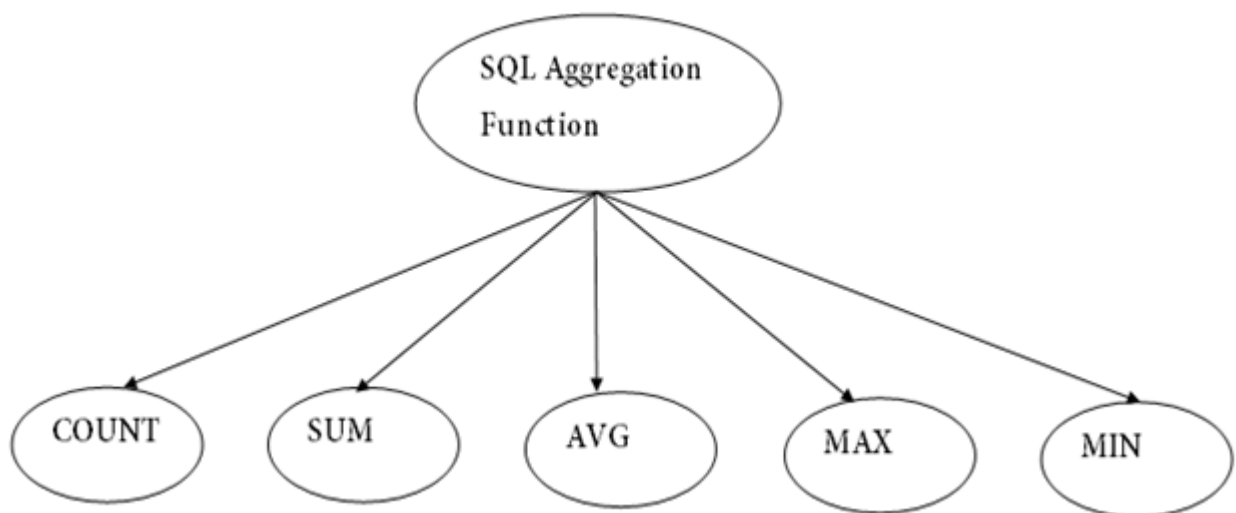
Best Practices

- Use NOT NULL restrictions on necessary data fields to maintain data integrity and consistency.
- Allowing NULL values should be done with caution, especially in columns that are used for crucial actions or restrictions.
- Include NULL handling principles and recommendations in the database schema documentation to maintain uniformity across applications and development teams.
- Use the right SQL methods and operators to handle NULL values in queries to avoid unexpected behaviour.

SQL Aggregate Functions

- SQL aggregation function is used to perform the calculations on multiple rows of a single column of a table. It returns a single value.
- It is also used to summarize the data.

Types of SQL Aggregate Functions



1. COUNT Function

- COUNT function is used to Count the number of rows in a database table. It can work on both numeric and non-numeric data types.

- COUNT function uses the COUNT(*) that returns the count of all the rows in a specified table. COUNT(*) considers duplicate and Null.

Syntax

```
COUNT(*)
or
COUNT( [ALL|DISTINCT] expression )
```

Sample table:**PRODUCT_MAST**

PRODUCT	COMPANY	QTY	RATE	COST
Item1	Com1	2	10	20
Item2	Com2	3	25	75
Item3	Com1	2	30	60
Item4	Com3	5	10	50
Item5	Com2	2	20	40
Item6	Cpm1	3	25	75
Item7	Com1	5	30	150
Item8	Com1	3	10	30
Item9	Com2	2	25	50
Item10	Com3	4	30	120

Example: COUNT()

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

Output:

```
10
```

Example: COUNT with WHERE

```
SELECT COUNT(*)
FROM PRODUCT_MAST;
WHERE RATE >= 20;
```

Output:

7

Example: COUNT() with DISTINCT

```
SELECT COUNT(DISTINCT COMPANY)
FROM PRODUCT_MAST;
```

Output:

3

Example: COUNT() with GROUP BY

```
SELECT COMPANY, COUNT(*)
FROM PRODUCT_MAST
GROUP BY COMPANY;
```

Output:

```
Com1 5
Com2 3
Com3 2
```

Example: COUNT() with HAVING

```
SELECT COMPANY, COUNT(*)
FROM PRODUCT_MAST
GROUP BY COMPANY
HAVING COUNT(*)>2;
```

Output:

```
Com1 5
Com2 3
```

2. SUM Function

Sum function is used to calculate the sum of all selected columns. It works on numeric fields only.

Syntax

```
SUM()  
or  
SUM( [ALL|DISTINCT] expression )
```

Example: SUM()

```
SELECT SUM(COST)  
FROM PRODUCT_MAST;
```

Output:

670

Example: SUM() with WHERE

```
SELECT SUM(COST)  
FROM PRODUCT_MAST  
WHERE QTY>3;
```

Output:

320

Example: SUM() with GROUP BY

```
SELECT SUM(COST)  
FROM PRODUCT_MAST  
WHERE QTY>3  
GROUP BY COMPANY;
```

Output:

Com1 150
Com2 170

Example: SUM() with HAVING

```
SELECT COMPANY, SUM(COST)
FROM PRODUCT_MAST
GROUP BY COMPANY
HAVING SUM(COST)>=170;
```

Output:

```
Com1 335
Com3 170
```

3. AVG function

The AVG function is used to calculate the average value of the numeric type. AVG function returns the average of all non-Null values.

Syntax

```
AVG()
or
AVG( [ALL|DISTINCT] expression )
```

Example:

```
SELECT AVG(COST)
FROM PRODUCT_MAST;
```

Output:

```
67.00
```

4. MAX Function

MAX function is used to find the maximum value of a certain column. This function determines the largest value of all selected values of a column.

Syntax

```
MAX()
or
MAX( [ALL|DISTINCT] expression )
```

Example:

```
SELECT MAX(RATE)
FROM PRODUCT_MAST;
```

30

5. MIN Function

MIN function is used to find the minimum value of a certain column. This function determines the smallest value of all selected values of a column.

Syntax

```
MIN()
or
MIN( [ALL|DISTINCT] expression )
```

Example:

```
SELECT MIN(RATE)
FROM PRODUCT_MAST;
```

Output:

10

SQL Nested Sub Query

A Subquery is a query within another SQL query and embedded within the WHERE clause.

Important Rule:

- A subquery can be placed in a number of SQL clauses like WHERE clause, FROM clause, HAVING clause.
- You can use Subquery with SELECT, UPDATE, INSERT, DELETE statements along with the operators like =, <, >, >=, <=, IN, BETWEEN, etc.

- A subquery is a query within another query. The outer query is known as the main query, and the inner query is known as a subquery.
- Subqueries are on the right side of the comparison operator.
- A subquery is enclosed in parentheses.
- In the Subquery, ORDER BY command cannot be used. But GROUP BY command can be used to perform the same function as ORDER BY command.

1. Subqueries with the Select Statement

SQL subqueries are most frequently used with the Select statement.

Syntax

```
SELECT column_name  
FROM table_name  
WHERE column_name expression operator  
( SELECT column_name from table_name WHERE ... );
```

Example

Consider the EMPLOYEE tables have the following records:

ID	NAME	AGE	ADDRESS	SALARY
1	John	20	US	2000.00
2	Stephan	26	Dubai	1500.00
3	David	27	Bangkok	2000.00
4	Alina	29	UK	6500.00
5	Kathrin	34	Bangalore	8500.00
6	Harry	42	China	4500.00
7	Jackson	25	Mizoram	10000.00

The subquery with a SELECT statement will be:

```
SELECT *  
FROM EMPLOYEE  
WHERE ID IN (SELECT ID  
FROM EMPLOYEE  
WHERE SALARY > 4500);
```

This would produce the following result:

ID	NAME	AGE	ADDRESS	SALARY
4	Alina	29	UK	6500.00
5	Kathrin	34	Bangalore	8500.00
7	Jackson	25	Mizoram	10000.00

2. Subqueries with the INSERT Statement

- SQL subquery can also be used with the Insert statement. In the insert statement, data returned from the subquery is used to insert into another table.
- In the subquery, the selected data can be modified with any of the character, date functions.

Syntax:

```
INSERT INTO table_name (column1, column2, column3....)
SELECT *
FROM table_name
WHERE VALUE OPERATOR
```

Example

Consider a table EMPLOYEE_BKP with similar as EMPLOYEE.

Now use the following syntax to copy the complete EMPLOYEE table into the EMPLOYEE_BKP table.

```
INSERT INTO EMPLOYEE_BKP
SELECT * FROM EMPLOYEE
WHERE ID IN (SELECT ID
FROM EMPLOYEE);
```

3. Subqueries with the UPDATE Statement

The subquery of SQL can be used in conjunction with the Update statement. When a subquery is used with the Update statement, then either single or multiple columns in a table can be updated.

Syntax

```
UPDATE table
SET column_name = new_value
WHERE VALUE OPERATOR
(SELECT COLUMN_NAME
FROM TABLE_NAME
WHERE condition);
```

Example

Let's assume we have an EMPLOYEE_BKP table available which is backup of EMPLOYEE table. The given example updates the SALARY by .25 times in the EMPLOYEE table for all employee whose AGE is greater than or equal to 29.

```
UPDATE EMPLOYEE
SET SALARY = SALARY * 0.25
WHERE AGE IN (SELECT AGE FROM EMPLOYEE_BKP
WHERE AGE >= 29);
```

This would impact three rows, and finally, the EMPLOYEE table would have the following records.

ID	NAME	AGE	ADDRESS	SALARY
1	John	20	US	2000.00
2	Stephan	26	Dubai	1500.00
3	David	27	Bangkok	2000.00
4	Alina	29	UK	1625.00
5	Kathrin	34	Bangalore	2125.00
6	Harry	42	China	1125.00
7	Jackson	25	Mizoram	10000.00

4. Subqueries with the DELETE Statement

The subquery of SQL can be used in conjunction with the Delete statement just like any other statements mentioned above.

Syntax

```
DELETE FROM TABLE_NAME  
WHERE VALUE OPERATOR  
(SELECT COLUMN_NAME  
FROM TABLE_NAME  
WHERE condition);
```

Example

Let's assume we have an EMPLOYEE_BKP table available which is backup of EMPLOYEE table. The given example deletes the records from the EMPLOYEE table for all EMPLOYEE whose AGE is greater than or equal to 29.

```
DELETE FROM EMPLOYEE  
WHERE AGE IN (SELECT AGE FROM EMPLOYEE_BKP  
WHERE AGE >= 29 );
```

This would impact three rows, and finally, the EMPLOYEE table would have the following records.

ID	NAME	AGE	ADDRESS	SALARY
1	John	20	US	2000.00
2	Stephan	26	Dubai	1500.00
3	David	27	Bangkok	2000.00
7	Jackson	25	Mizoram	10000.00

SQL JOINS

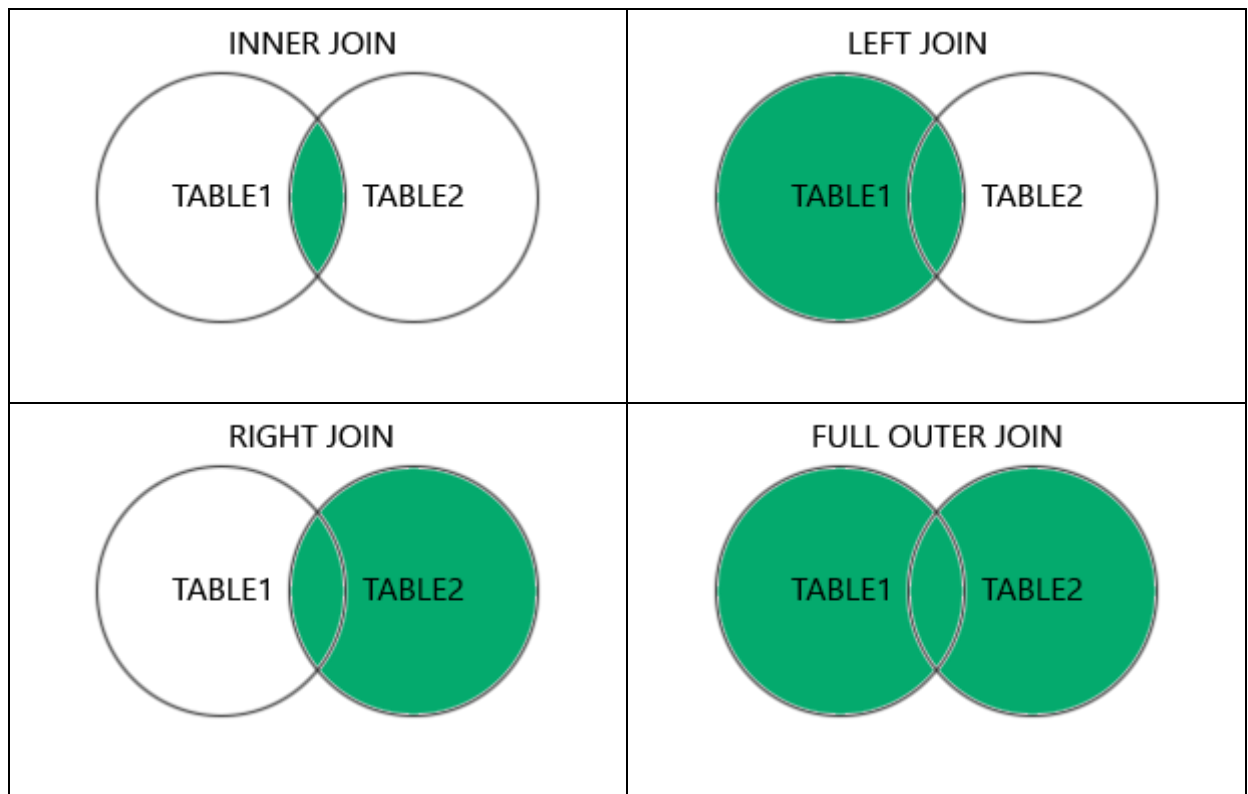
As the name shows, JOIN means *to combine something*. In case of SQL, JOIN means **"to combine two or more tables"**.

The SQL JOIN clause takes records from two or more tables in a database and combines it together.

Different Types of SQL JOINS

Here are the different types of the JOINS in SQL:

- **(INNER) JOIN**: Returns records that have matching values in both tables
- **LEFT (OUTER) JOIN**: Returns all records from the left table, and the matched records from the right table
- **RIGHT (OUTER) JOIN**: Returns all records from the right table, and the matched records from the left table
- **FULL (OUTER) JOIN**: Returns all records when there is a match in either left or right table



In the process of joining, rows of both tables are combined in a single table.

Why SQL JOIN is used?

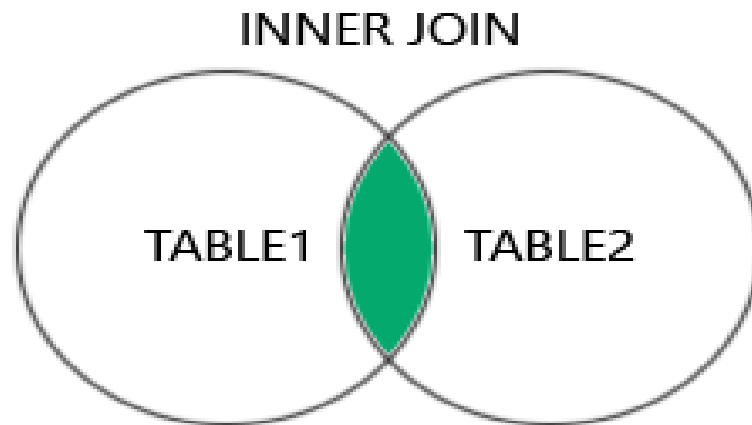
If you want to access more than one table through a select statement.

If you want to combine two or more table then SQL JOIN statement is used .it combines rows of that tables in one table and one can retrieve the information by a SELECT statement.

The joining of two or more tables is based on common field between them.

1. SQL INNER JOIN also known as simple join is the most common type of join.

How to use SQL join or SQL Inner Join?

**Syntax**

```
SELECT column_name(s)
FROM table1
INNER JOIN table2
ON table1.column_name = table2.column_name;
```

Let an example to deploy SQL JOIN process:

1.Staff table

ID	Staff_NAME	Staff_AGE	STAFF_ADDRESS	Monthley_Package
1	ARYAN	22	MUMBAI	18000
2	SUSHIL	32	DELHI	20000
3	MONTY	25	MOHALI	22000
4	AMIT	20	ALLAHABAD	12000

2.Payment table

Pavment ID	DATE	Staff ID	AMOUNT
101	30/12/2009	1	3000.00
102	22/02/2010	3	2500.00
103	23/02/2010	4	3500.00

So if you follow this JOIN statement to join these two tables ?

```
SELECT Staff_ID, Staff_NAME, Staff_AGE, AMOUNT  
FROM STAFF s INNER JOIN PAYMENT p ON s.ID =p.STAFF_ID;
```

This will produce the result like this:

STAFF_ID	NAME	Staff_AGE	AMOUNT
1	ARYAN	22	3000
3	MONTY	25	2500
4	AMIT	25	3500

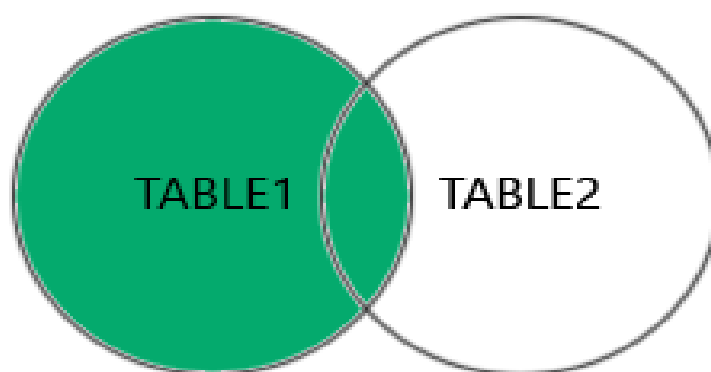
2. LEFT JOIN

The **LEFT JOIN** keyword returns all records from the left table (table1), and the matching records from the right table (table2). The result is 0 records from the right side, if there is no match.

LEFT JOIN Syntax

```
SELECT column_name(s)  
FROM table1  
LEFT JOIN table2  
ON table1.column_name = table2.column_name;
```

LEFT JOIN



Consider we have the following tables with the given data:

Table 1: employee

EmployeeID	Employee Name	Employee Salary
1	Arun Tiwari	50000
2	Sachin Rathi	64000
3	Harshal Pathak	48000
4	Arjun Kuwar	46000
5	Sarthak Gada	62000
6	Saurabh Sheik	53000
7	Shubham Singh	29000
8	Shivam Dixit	54000
9	Vicky Gujral	39000
10	Vijay Bose	28000

Table 2: department

DenartmentID	Denartment Name	Emmployee ID
1	Production	1
2	Sales	3
3	Marketing	4
4	Accounts	5
5	Development	7
6	HR	9
7	Sales	10

Example :

Write a query to perform left outer join considering employee table as the left table and department table as the right table.

Query:

```
SELECT e.EmployeeID, e.Employee_Name, e.Employee_Salary, d.DepartmentID, d.
Department_Name FROM employee e LEFT OUTER JOIN department d ON e.Empl
oyeeID = d.Employee_ID;
```

You will get the following output:

Employee ID	Employee_Name	Employee_Salary	DepartmentID	Department_Name
1	Arun Tiwari	50000	1	Production
2	Sachin Rathi	64000	NULL	NULL
3	Harshal Pathak	48000	2	Sales
4	Arjun Kuwar	46000	3	Marketing
5	Sarthak Gada	62000	4	Accounts
6	Saurabh Sheik	53000	NULL	NULL
7	Shubham	29000	5	Development
8	Shivam Dixit	54000	NULL	NULL
9	Vicky Gujral	39000	6	HR
10	Vijay Bose	28000	7	Sales

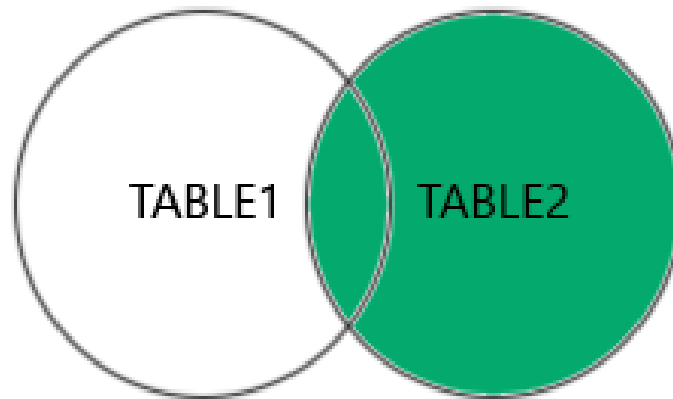
3. Right Join:

The **RIGHT JOIN** keyword returns all records from the right table (table2), and the matching records from the left table (table1). The result is 0 records from the left side, if there is no match.

RIGHT JOIN Syntax

```
SELECT column_name(s)
FROM table1
RIGHT JOIN table2
ON table1.column_name = table2.column_name;
```

RIGHT JOIN

**Example 1:**

Write a query to perform right join considering employee table as the left table and department table as the right table.

Query:

```
SELECT e.EmployeeID, e.Employee_Name, e.Employee_Salary, d.DepartmentID, d.
Department_Name FROM employee e RIGHT JOIN department d ON e.EmployeeID
= d.Employee_ID;
```

You will get the following output:

EmployeeID	Employee_Name	Employee_Salary	DepartmentID	Department_Name
1	Arun Tiwari	50000	1	Production
3	Harshal Pathak	48000	2	Sales
4	Arjun Kuwar	46000	3	Marketing
5	Sarthak Gada	62000	4	Accounts
7	Shubham Singh	29000	5	Development
9	Vicky Gujral	39000	6	HR
10	Vijay Bose	28000	7	Sales

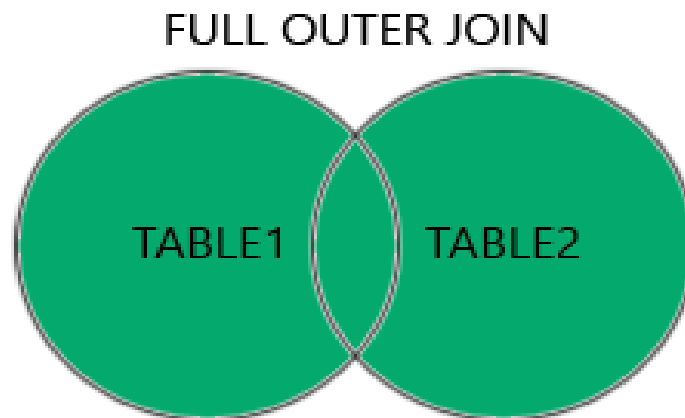
4. Full Join:

The **FULL JOIN** returns all records when there is a match in left (table1) or right (table2) table records.

Tip: **FULL OUTER JOIN** and **FULL JOIN** are the same.

FULL OUTER JOIN Syntax

```
SELECT column_name(s)
FROM table1
FULL OUTER JOIN table2
ON table1.column_name = table2.column_name
WHERE condition;
```



Example 1:

Write a query to perform full outer join considering the employee table as the left table and department table as the right table.

Query:

```
SELECT e.EmployeeID, e.Employee_Name, e.Employee_Salary, d.DepartmentID, d.
Department_Name
FROM Employee e,
FULL OUTER JOIN Department d ON e.EmployeeID = d.Employee_ID ;
```

You will get the following output:

EmployeeID	Employee_Name	Employee_Salary	DepartmentID	Department_Name
1	Arun Tiwari	50000	1	Production
3	Harshal Pathak	48000	2	Sales
4	Arjun Kuwar	46000	3	Marketing
5	Sarthak Gada	62000	4	Accounts
7	Shubham Singh	29000	5	Developme
9	Vicky Gujral	39000	6	HR
10	Vijay Bose	28000	7	Sales
2	Sachin Rathi	64000	NULL	NULL
6	Saurabh Sheik	53000	NULL	NULL
8	Shivam Dixit	54000	NULL	NULL

SQL Views

SQL provides the concept of VIEW, which hides the complexity of the data and restricts unnecessary access to the database. It permits the users to access only a particular column rather than the whole data of the table.

The **View** in the Structured Query Language is considered as the virtual table, which depends on the result-set of the predefined SQL statement.

Like the SQL tables, Views also store data in rows and columns, but the rows do not have any physical existence in the database.

Any database administrator and user can easily create the View by selecting the columns from one or more database tables. They can also delete and update the views according to their needs.

A view can store either all the records of the table or a particular record from the table using the WHERE clause.

Create a SQL View

You can easily create a View in Structured Query Language by using the CREATE VIEW statement. You can create the View from a single table or multiple tables.

Syntax to Create View from Single Table

```
CREATE VIEW View_Name AS  
SELECT Column_Name1, Column_Name2, ....., Column_NameN  
FROM Table_Name  
WHERE condition;
```

In the syntax, View_Name is the name of View you want to create in SQL. The SELECT command specifies the rows and columns of the table, and the WHERE clause is optional, which is used to select the particular record from the table.

Syntax to Create View from Multiple Tables

You can create a View from multiple tables by including the tables in the SELECT statement.

```
CREATE VIEW View_Name AS  
SELECT Table_Name1.Column_Name1, Table_Name1.Column_Name2, Table_Name  
e2.Column_Name2, ....., Table_NameN.Column_NameN  
FROM Table_Name1, Table_Name2, ....., Table_NameN  
WHERE condition;
```

Example to Create a View from Single table

Let's consider the **Student_Details** table, which consists of Stu_ID, Stu_Name, Stu_Subject, and Stu_Marks columns. The data of the Student_Details is shown in the following table:

Student ID	Stu Name	Stu Subject	Stu Marks
1001	Akhil	Math	85
1002	Balram	Science	78
1003	Bheem	Math	87
1004	Chetan	English	95
1005	Diksha	Hindi	99
1006	Raman	Computer	90
1007	Sheetal	Science	68

Suppose, you want to create a view with Stu_ID, Stu_Subject, and Stu_Marks of those students whose marks are greater than 85. For this issue, you have to type the following query:

```
CREATE VIEW Student_View AS
SELECT Student_ID, Stu_Subject, Stu_Marks
FROM Student_Details
WHERE Stu_Marks > 85;
Select * FROM Student_View;
```

Output:

Student ID	Stu Subject	Stu Marks
1001	Math	85
1003	Math	87
1004	English	95
1005	Hindi	99
1006	Computer	90

Update an SQL View

We can also modify existing data and insert the new record into the view in the Structured Query Language. A view in SQL can only be modified if the view follows the following conditions:

1. You can update that view which depends on only one table. SQL will not allow updating the view which is created more than one table.
2. The fields of view should not contain NULL values.
3. The view does not contain any subquery and DISTINCT keyword in its definition.
4. The views cannot be updatable if the SELECT statement used to create a View contains JOIN or HAVING or GROUP BY clause.
5. If any field of view contains any SQL aggregate function, you cannot modify the view.

Syntax to Update a View

```
CREATE OR REPLACE VIEW View_Name AS  
SELECT Column_Name1, Column_Name2, ....., Column_NameN  
FROM Table_Name  
WHERE condition;
```

Example to Update a View

If we want to update the above Student_View and add the Stu_Name attribute from the Student table in the view, you have to type the following Replace query in SQL:

```
CREATE OR REPLACE VIEW Student_View AS  
SELECT Student_ID, Stu_Name, Stu_Subject, Stu_Marks  
FROM Student_Details  
WHERE Stu_Subject = 'Math';
```

The above statement updates the existing Student_View and updates the data based on the SELECT statement.

Now, you can see the virtual table by typing the following query:

```
SELECT * FROM Student_View;
```

Output:

Student ID	Stu Name	Stu Subject	Stu Marks
1001	Akhil	Math	85
1003	Bheem	Math	87

Drop a View

We can also delete the existing view from the database if it is no longer needed. The following SQL DROP statement is used to delete the view:

```
DROP VIEW View_Name;
```

Example to Drop a View

Suppose, you want to delete the above Student_View, then you have to type the following query in the Structured Query Language:

```
DROP VIEW Student_View;
```

Constraints in SQL

Constraints in SQL means we are applying certain conditions or restrictions on the database. This further means that before inserting data into the database, we are checking for some conditions. If the condition we have applied to the database holds true for the data which is to be inserted, then only the data will be inserted into the database tables.

Constraints in SQL can be categorized into two types:

1. Column Level Constraint:

Column Level Constraint is used to apply a constraint on a single column.

2. Table Level Constraint:

Table Level Constraint is used to apply a constraint on multiple columns.

Constraints available in SQL are:

1. NOT NULL
2. UNIQUE
3. PRIMARY KEY
4. FOREIGN KEY
5. CHECK
6. DEFAULT
7. CREATE INDEX
8. AUTO_INCREMENT

1. NOT NULL

- NULL means empty, i.e., the value is not available.
- Whenever a table's column is declared as NOT NULL, then the value for that column cannot be empty for any of the table's records.
- There must exist a value in the column to which the NOT NULL constraint is applied.

NOTE: *NULL does not mean zero. NULL means empty column, not even zero.*

Syntax to apply the NOT NULL constraint during table creation:

```
CREATE TABLE TableName  
(ColumnName1 datatype NOT NULL,  
  ColumnName2 datatype,...,  
  ColumnNameN datatype);
```

Example:

Create a student table and apply a NOT NULL constraint on one of the table's column while creating a table.

```
CREATE TABLE stud  
(StudentID INT NOT NULL,  
  Student_FirstName VARCHAR(20),  
  Student_LastName VARCHAR(20),  
  Student_PhoneNumber VARCHAR(20),  
  Student_Email_ID VARCHAR(40));
```

mysql> **DESC** student;

```

Command Prompt - sqlplus;

SQL> DESC stud;
Name                               Null?    Type
-----
STUDENTID                         NOT NULL NUMBER(38)
STUDENT_FIRSTNAME                  VARCHAR2(20)
STUDENT_LASTNAME                   VARCHAR2(20)
STUDENT_PHONENUMBER                VARCHAR2(20)
STUDENT_EMAIL_ID                   VARCHAR2(40)

SQL>

```

Syntax to apply the NOT NULL constraint on an existing table's column:

ALTER TABLE TableName
 MODIFY New_ColumnName Datatype **NOT NULL**;

Example:

Consider we have an existing table student, without any constraints applied to it. Later, we decided to apply a NOT NULL constraint to one of the table's column. Then we will execute the following query:

mysql> **ALTER TABLE** stud MODIFY Student_Email_ID **varchar(40) NOT NULL**;

```

Command Prompt - sqlplus;

SQL> ALTER TABLE stud MODIFY Student_Email_ID varchar(40) NOT NULL;
Table altered.

SQL> DESC stud;
Name                               Null?    Type
-----
STUDENTID                         NOT NULL NUMBER(38)
STUDENT_FIRSTNAME                  VARCHAR2(20)
STUDENT_LASTNAME                   VARCHAR2(20)
STUDENT_PHONENUMBER                VARCHAR2(20)
STUDENT_EMAIL_ID                   NOT NULL VARCHAR2(40)

SQL>

```

2. UNIQUE

- Duplicate values are not allowed in the columns to which the UNIQUE constraint is applied.
- The column with the unique constraint will always contain a unique value.
- This constraint can be applied to one or more than one column of a table, which means more than one unique constraint can exist on a single table.

- Using the UNIQUE constraint, you can also modify the already created tables.

Syntax to apply the UNIQUE constraint on a single column:

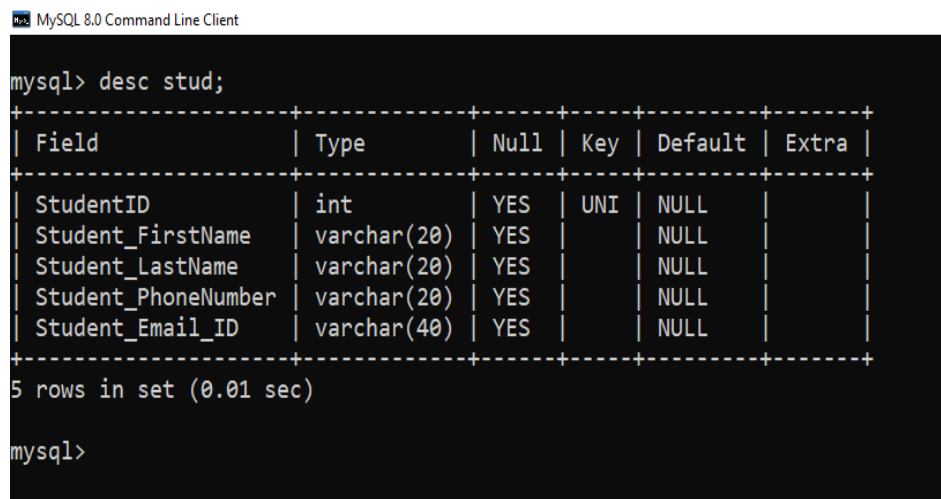
```
CREATE TABLE TableName
(ColumnName1 datatype UNIQUE,
ColumnName2 datatype, ...,
ColumnNameN datatype);
```

Example:

Create a student table and apply a UNIQUE constraint on one of the table's column while creating a table.

```
mysql> CREATE TABLE stud(
StudentID INT UNIQUE,
Student_FirstName VARCHAR(20),
Student_LastName VARCHAR(20),
Student_PhoneNumber VARCHAR(20),
Student_Email_ID VARCHAR(40));

mysql> DESC student;
```



```
MySQL 8.0 Command Line Client

mysql> desc stud;
+-----+-----+-----+-----+-----+-----+
| Field          | Type          | Null | Key | Default | Extra |
+-----+-----+-----+-----+-----+-----+
| StudentID      | int           | YES  | UNI | NULL    |       |
| Student_FirstName | varchar(20)   | YES  |     | NULL    |       |
| Student_LastName | varchar(20)   | YES  |     | NULL    |       |
| Student_PhoneNumber | varchar(20)   | YES  |     | NULL    |       |
| Student_Email_ID | varchar(40)   | YES  |     | NULL    |       |
+-----+-----+-----+-----+-----+-----+
5 rows in set (0.01 sec)

mysql>
```

Syntax to apply the UNIQUE constraint on an existing table's column:

```
ALTER TABLE TableName ADD UNIQUE (ColumnName);
```

Example:

Consider we have an existing table student, without any constraints applied to it. Later, we decided to apply a UNIQUE constraint to one of the table's column. Then we will execute the following query:

```
mysql> ALTER TABLE stud ADD UNIQUE (Student_Email_ID);
```

```
MySQL 8.0 Command Line Client

mysql> desc stud;
+-----+-----+-----+-----+-----+-----+
| Field          | Type          | Null | Key | Default | Extra |
+-----+-----+-----+-----+-----+-----+
| StudentID      | int           | YES  | UNI | NULL    |       |
| Student_FirstName | varchar(20)   | YES  |     | NULL    |       |
| Student_LastName  | varchar(20)   | YES  |     | NULL    |       |
| Student_PhoneNumber | varchar(20)   | YES  |     | NULL    |       |
| Student_Email_ID | varchar(40)   | YES  | UNI | NULL    |       |
+-----+-----+-----+-----+-----+-----+
5 rows in set (0.00 sec)

mysql> _
```

3. PRIMARY KEY

- PRIMARY KEY Constraint is a combination of NOT NULL and Unique constraints.
- NOT NULL constraint and a UNIQUE constraint together forms a PRIMARY constraint.
- The column to which we have applied the primary constraint will always contain a unique value and will not allow null values.

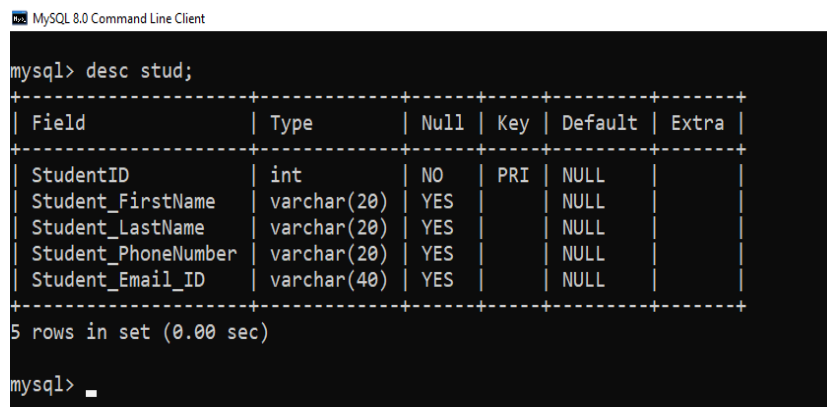
Syntax of primary key constraint during table creation:

```
CREATE TABLE TableName (
ColumnName1 datatype PRIMARY KEY,
ColumnName2 datatype, ...,
ColumnNameN datatype);
```

Example:

Create a student table and apply the PRIMARY KEY constraint while creating a table.

```
mysql> CREATE TABLE stud(
StudentID INT PRIMARY KEY,
Student_FirstName VARCHAR(20),
Student_LastName VARCHAR(20),
Student_PhoneNumber VARCHAR(20),
Student_Email_ID VARCHAR(40));
```



```
mysql> desc stud;
```

Field	Type	Null	Key	Default	Extra
StudentID	int	NO	PRI	NULL	
Student_FirstName	varchar(20)	YES		NULL	
Student_LastName	varchar(20)	YES		NULL	
Student_PhoneNumber	varchar(20)	YES		NULL	
Student_Email_ID	varchar(40)	YES		NULL	

5 rows in set (0.00 sec)

```
mysql> _
```

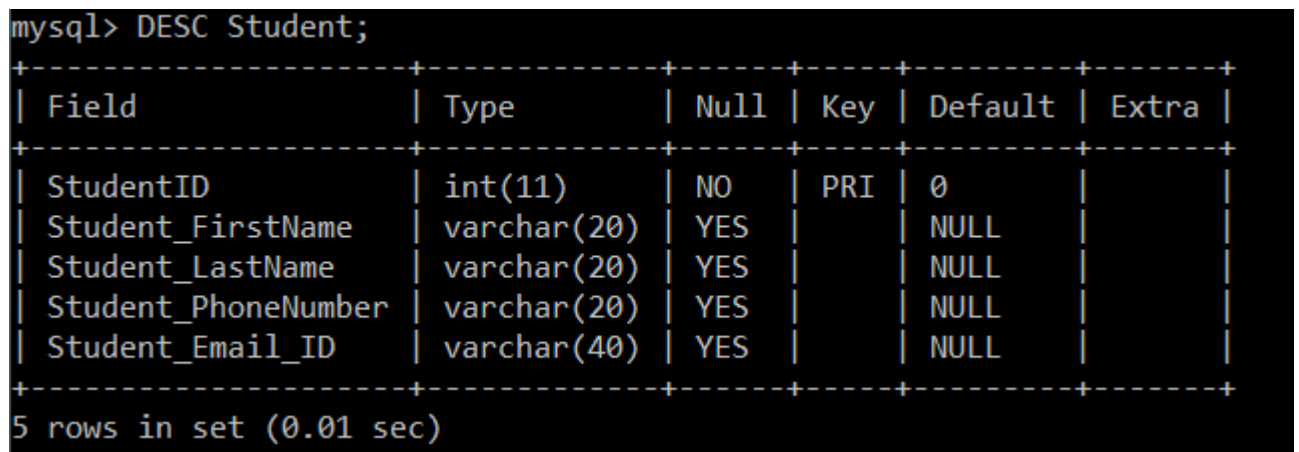
Syntax to apply the primary key constraint on an existing table's column:

```
ALTER TABLE TableName ADD PRIMARY KEY (ColumnName);
```

Example:

Consider we have an existing table student, without any constraints applied to it. Later, we decided to apply the PRIMARY KEY constraint to the table's column. Then we will execute the following query:

```
mysql> ALTER TABLE student ADD PRIMARY KEY (StudentID);
```



```
mysql> DESC Student;
```

Field	Type	Null	Key	Default	Extra
StudentID	int(11)	NO	PRI	0	
Student_FirstName	varchar(20)	YES		NULL	
Student_LastName	varchar(20)	YES		NULL	
Student_PhoneNumber	varchar(20)	YES		NULL	
Student_Email_ID	varchar(40)	YES		NULL	

5 rows in set (0.01 sec)

4. FOREIGN KEY

- A foreign key is used for referential integrity.
- When we have two tables, and one table takes reference from another table, i.e., the same column is present in both the tables and that column acts as a primary key in one table. That particular column will act as a foreign key in another table.

Syntax to apply a foreign key constraint during table creation:

```
CREATE TABLE tablename(
ColumnName1 Datatype(SIZE) PRIMARY KEY,
ColumnNameN Datatype(SIZE),
FOREIGN KEY( ColumnName )
REFERENCES PARENT_TABLE_NAME(Primary_Key_ColumnName));
```

Example:

Create an employee table and apply the FOREIGN KEY constraint while creating a table.

To create a foreign key on any table, first, we need to create a primary key on a table.

```
mysql> CREATE TABLE employee (Emp_ID INT NOT NULL PRIMARY KEY,
Emp_Name VARCHAR (40), Emp_Salary VARCHAR (40));
```

```
mysql> DESC employee;
```

Field	Type	Null	Key	Default	Extra
Emp_ID	int(11)	NO	PRI	NULL	
Emp_Name	varchar(40)	YES		NULL	
Emp_Salary	varchar(40)	YES		NULL	

3 rows in set (0.01 sec)

Now, we will write a query to apply a foreign key on the department table referring to the primary key of the employee table, i.e., Emp_ID.

```
mysql> CREATE TABLE department(
Dept_ID INT NOT NULL PRIMARY KEY,
Dept_Name VARCHAR(40),
Emp_ID INT NOT NULL,
FOREIGN KEY(Emp_ID) REFERENCES employee(Emp_ID));
```

```
mysql> DESC department;
```

Field	Type	Null	Key	Default	Extra
Dept_ID	int(11)	NO	PRI	NULL	
Dept_Name	varchar(40)	YES		NULL	
Emp_ID	int(11)	NO	MUL	NULL	

3 rows in set (0.01 sec)

Syntax to apply the foreign key constraint on an existing table's column:

ALTER TABLE Parent_TableName **ADD FOREIGN KEY** (ColumnName) **REFERENCES** Child_TableName (ColumnName);

Example:

Consider we have an existing table employee and department. Later, we decided to apply a FOREIGN KEY constraint to the department table's column. Then we will execute the following query:

```
mysql> ALTER TABLE department ADD FOREIGN KEY (Emp_ID) REFERENCES employee (Emp_ID);
```

```
mysql> DESC department;
```

Field	Type	Null	Key	Default	Extra
Dept_ID	int(11)	NO	PRI	NULL	auto_increment
Dept_Name	varchar(40)	YES		NULL	
Emp_ID	int(11)	NO	MUL	NULL	

3 rows in set (0.10 sec)

5. CHECK

- Whenever a check constraint is applied to the table's column, and the user wants to insert the value in it, then the value will first be checked for certain conditions before inserting the value into that column.
- **For example:** if we have an age column in a table, then the user will insert any value of his choice. The user will also enter even a negative value or any other

invalid value. But, if the user has applied check constraint on the age column with the condition age greater than 18. Then in such cases, even if a user tries to insert an invalid value such as zero or any other value less than 18, then the age column will not accept that value and will not allow the user to insert it due to the application of check constraint on the age column.

Syntax to apply check constraint on a single column:

```
CREATE TABLE TableName (
ColumnName1 datatype CHECK (ColumnName1 Condition),
ColumnName2 datatype, ...,
ColumnNameN datatype);
```

Example:

Create a student table and apply CHECK constraint to check for the age less than or equal to 15 while creating a table.

```
mysql> CREATE TABLE stud
(StudentID INT,
Student_FirstName VARCHAR(20),
Student_LastName VARCHAR(20),
Student_PhoneNumber VARCHAR(20),
Student_Email_ID VARCHAR(40),
Age INT CHECK( Age <= 15));
```

MySQL 8.0 Command Line Client

```
mysql> desc stud;
+-----+-----+-----+-----+-----+-----+
| Field          | Type      | Null | Key | Default | Extra |
+-----+-----+-----+-----+-----+-----+
| StudentID      | int       | YES  |     | NULL    |       |
| Student_FirstName | varchar(20) | YES  |     | NULL    |       |
| Student_LastName | varchar(20) | YES  |     | NULL    |       |
| Student_PhoneNumber | varchar(20) | YES  |     | NULL    |       |
| Student_Email_ID | varchar(40) | YES  |     | NULL    |       |
| Age            | int       | YES  |     | NULL    |       |
+-----+-----+-----+-----+-----+-----+
6 rows in set (0.00 sec)
```

```
INSERT INTO stud values(1,'Kishore','K','949949949','kk@g.c',10);
```

```
INSERT INTO stud values(1,'Kishore','K','949949949','kk@g.c',18);
```

MySQL 8.0 Command Line Client

```
mysql> INSERT INTO stud values(1,'Kishore','K','949949949','kk@g.c',10);  
Query OK, 1 row affected (0.01 sec)
```

```
mysql> INSERT INTO stud values(1,'Kishore','K','949949949','kk@g.c',18);  
ERROR 3819 (HY000): Check constraint 'stud_chk_1' is violated.  
mysql>
```

Syntax to apply check constraint on an existing table's column:

ALTER TABLE TableName **ADD CHECK** (ColumnName Condition);

Example:

Consider we have an existing table student. Later, we decided to apply the CHECK constraint on the student table's column. Then we will execute the following query:

```
mysql> ALTER TABLE student ADD CHECK ( Age >=25 );
```

MySQL 8.0 Command Line Client

```
mysql> ALTER TABLE stud ADD CHECK ( Age >=25 );  
Query OK, 0 rows affected (0.06 sec)  
Records: 0 Duplicates: 0 Warnings: 0  
  
mysql>
```

6. DEFAULT

Whenever a default constraint is applied to the table's column, and the user has not specified the value to be inserted in it, then the default value which was specified while applying the default constraint will be inserted into that particular column.

Syntax to apply default constraint during table creation:

```
CREATE TABLE TableName (  
ColumnName1 datatype DEFAULT Value,  
ColumnName2 datatype, ...,  
ColumnNameN datatype);
```

Example:

Create a student table and apply the default constraint while creating a table.

```
mysql> CREATE TABLE stud(
StudentID INT,
Student_FirstName VARCHAR(20),
Student_LastName VARCHAR(20),
Student_PhoneNumber VARCHAR(20),
Student_Email_ID VARCHAR(40) DEFAULT "vjit@gmail.com");
```

```
MySQL 8.0 Command Line Client

mysql> desc stud;
+-----+-----+-----+-----+-----+-----+
| Field          | Type          | Null | Key | Default          | Extra |
+-----+-----+-----+-----+-----+-----+
| StudentID      | int           | YES  |     | NULL             |       |
| Student_FirstName | varchar(20)   | YES  |     | NULL             |       |
| Student_LastName | varchar(20)   | YES  |     | NULL             |       |
| Student_PhoneNumber | varchar(20)   | YES  |     | NULL             |       |
| Student_Email_ID | varchar(40)   | YES  |     | vjit@gmail.com    |       |
+-----+-----+-----+-----+-----+-----+
5 rows in set (0.00 sec)
```

Syntax to apply default constraint on an existing table's column:

```
ALTER TABLE TableName ALTER ColumnName SET DEFAULT Value;
```

Example:

Consider we have an existing table student. Later, we decided to apply the DEFAULT constraint on the student table's column. Then we will execute the following query:

```
mysql> ALTER TABLE student ALTER Student_Email_ID SET DEFAULT "vjit@gmail.com";
```

```
MySQL 8.0 Command Line Client

mysql> desc stud;
+-----+-----+-----+-----+-----+-----+
| Field          | Type          | Null | Key | Default          | Extra |
+-----+-----+-----+-----+-----+-----+
| StudentID      | int           | YES  |     | NULL             |       |
| Student_FirstName | varchar(20)   | YES  |     | NULL             |       |
| Student_LastName | varchar(20)   | YES  |     | NULL             |       |
| Student_PhoneNumber | varchar(20)   | YES  |     | NULL             |       |
| Student_Email_ID | varchar(40)   | YES  |     | vjit@gmail.com    |       |
+-----+-----+-----+-----+-----+-----+
5 rows in set (0.00 sec)
```


7. CREATE INDEX

CREATE INDEX constraint is used to create an index on the table. Indexes are not visible to the user, but they help the user to speed up the searching speed or retrieval of data from the database.

Syntax to create an index on single column:

CREATE INDEX IndexName **ON** TableName (ColumnName 1);

Example:

Create an index on the student table and apply the default constraint while creating a table.

mysql> **CREATE INDEX** idx_StudentID **ON** student (StudentID);

```

MySQL 8.0 Command Line Client

mysql> show indexes from stud;
+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| Table | Non_unique | Key_name | Seq_in_index | Column_name | Collation | Cardinality | Sub_part | Packed | Null | Index_type | Comment | Index_comment | Visible | Exp |
+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| stud  | 0          | PRIMARY | 1            | StudentID   | A         | 2           | NULL    | NULL   | NULL | BTREE     |         |               | YES    | NUL |
+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
1 row in set (0.00 sec)

mysql> CREATE INDEX idx_StudentID ON stud (StudentID);
Query OK, 0 rows affected (0.06 sec)
Records: 0 Duplicates: 0 Warnings: 0

mysql> show indexes from stud;
+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| Table | Non_unique | Key_name | Seq_in_index | Column_name | Collation | Cardinality | Sub_part | Packed | Null | Index_type | Comment | Index_comment | Visible |
+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| stud  | 0          | PRIMARY | 1            | StudentID   | A         | 2           | NULL    | NULL   | NULL | BTREE     |         |               | YES    |
| NULL |            |         |              |              |           |             |         |         |      |           |         |               |        |
| stud  | 1          | idx_StudentID | 1          | StudentID   | A         | 2           | NULL    | NULL   | NULL | BTREE     |         |               | YES    |
| NULL |            |         |              |              |           |             |         |         |      |           |         |               |        |
+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
2 rows in set (0.01 sec)

mysql>

```

Syntax to create an index on an existing table:

ALTER TABLE TableName **ADD INDEX** (ColumnName);

Consider we have an existing table student. Later, we decided to apply the DEFAULT constraint on the student table's column. Then we will execute the following query:

mysql> **ALTER TABLE** stud **ADD INDEX** (StudentID);

MySQL 8.0 Command Line Client

```
mysql> CREATE INDEX idx_StudentID ON stud (StudentID);
Query OK, 0 rows affected (0.03 sec)
Records: 0 Duplicates: 0 Warnings: 0
```

```
mysql> desc stud;
```

Field	Type	Null	Key	Default	Extra
StudentID	int	YES	MUL	NULL	
Student_FirstName	varchar(20)	YES		NULL	
Student_LastName	varchar(20)	YES		NULL	
Student_PhoneNumber	varchar(20)	YES		NULL	
Student_Email_ID	varchar(40)	YES		vjit@gmail.com	

```
5 rows in set (0.00 sec)
```

```
mysql> _
```

8. AUTO_INCREMENT

MySQL uses the **AUTO_INCREMENT** keyword to perform an auto-increment feature.

By default, the starting value for **AUTO_INCREMENT** is 1, and it will increment by 1 for each new record.

```
CREATE TABLE stud(
StudentID INT AUTO_INCREMENT PRIMARY KEY,
Student_FirstName VARCHAR(20),
Student_LastName VARCHAR(20),
Student_PhoneNumber VARCHAR(20),
Student_Email_ID VARCHAR(40) DEFAULT "vjit@gmail.com");
```

MySQL 8.0 Command Line Client

```
mysql> desc stud;
```

Field	Type	Null	Key	Default	Extra
StudentID	int	NO	PRI	NULL	auto_increment
Student_FirstName	varchar(20)	YES		NULL	
Student_LastName	varchar(20)	YES		NULL	
Student_PhoneNumber	varchar(20)	YES		NULL	
Student_Email_ID	varchar(40)	YES		vjit@gmail.com	

```
5 rows in set (0.00 sec)
```

```
mysql>
```

```
INSERT INTO stud (
Student_FirstName, Student_LastName , Student_PhoneNumber , Student_Email_ID )
values ( 'Kishore','K','949949949','kk@g.c' );
INSERT INTO stud (
Student_FirstName, Student_LastName , Student_PhoneNumber , Student_Email_ID )
values('VJIT','Hyd','949949949','vjit@g.c');
```

```
mysql> select * from stud;
+-----+-----+-----+-----+-----+
| StudentID | Student_FirstName | Student_LastName | Student_PhoneNumber | Student_Email_ID |
+-----+-----+-----+-----+-----+
| 1 | Kishore | K | 949949949 | kk@g.c |
| 2 | VJIT | Hyd | 949949949 | vjit@g.c |
+-----+-----+-----+-----+-----+
2 rows in set (0.00 sec)

mysql> .
```

- **AUTO_INCREMENT=100** -- it starts from 100 and incremented by 1
- **SET @@AUTO_INCREMENT_INCREMENT=2;** -- incremented by 2
- **SET @@SESSION.AUTO_INCREMENT_INCREMENT=2;** -- incremented by 2 in the session
- **SET @@GLOBAL.AUTO_INCREMENT_INCREMENT=2;** ---- incremented by 2 globally

Authorization in SQL

We may assign a user several forms of authorizations on parts of the database. Authorizations on data include:

- Authorization to read data.
- Authorization to insert new data.
- Authorization to update data.
- Authorization to delete data.

Each of these types of authorizations is called a **privilege**. We may authorize the user all, none, or a combination of these types of privileges on specified parts of a database, such as a relation or a view.

GRANT and REVOKE

- In **MySQL**, creating a user account using the **CREATE USER** statement is just the first step in user management.

```
CREATE USER account_name@localhost IDENTIFIED BY 'password';
CREATE USER vjit@localhost IDENTIFIED BY 'vjit@123';
```

- While this command establishes a new user, it does not assign any privileges.
- To control what actions a user can perform, the GRANT statement is essential for assigning specific privileges to user accounts.

Syntax:

```
GRANT privileges_names ON object TO user;
```

Explanation:

- **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.

Types of Privileges

Privileges: The privileges that can be granted to the users are listed below along with the description:

Privilege	Description
SELECT	<i>select statement on tables</i>
INSERT	<i>insert statement on the table</i>
DELETE	<i>delete statement on the table</i>
INDEX	<i>Create an index on an existing table</i>
CREATE	<i>Create table statements</i>
ALTER	<i>Ability to perform ALTER TABLE to change the table definition</i>
DROP	<i>Drop table statements</i>
ALL	<i>Grant all permissions except GRANT OPTION</i>
UPDATE	<i>Update statements on the table</i>
GRANT	<i>Allows to grant the privilege that</i>

Granting Privileges to Users

1. To grant Select Privilege to a table named “**users**” where User Name is **Amit**, the following GRANT statement should be executed.
2. The general syntax of specifying a username is: ‘**user_name**’@’**address**’.
3. If the user ‘**Amit**’ is on the local host then we have to mention it as ‘**Amit**’@’**localhost**’. Or suppose if the ‘**Amit**’ username is on **192.168.1.100** IP address then we have to mention it as ‘**Amit**’@’**192.168.1.100**’.
4. ‘**user_name**’@’**address**’ – When you’re granting or revoking permissions in MySQL, you use the ‘username’ or ‘hostname’ format to tell which users are allowed or denied. This is important for keeping security and access control in place, so here’s why we use it:

1. Granting SELECT Privilege to a User

To grant the SELECT privilege on a table named “**users**” to a user named “**Amit**”, use the following command:

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

2. Granting Multiple Privileges

To grant multiple privileges, such as **SELECT**, **INSERT**, **DELETE**, and **UPDATE** to the user “**Amit**”,

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

3. Granting All Privileges

To grant all privileges on the “**users**” table to “**Amit**”, use:

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

4. Granting Privileges to All Users

To grant a specific privilege (e.g., **SELECT**) to all users on the “**users**” table, execute:

```
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**”.

5. Granting Privileges on Functions/Procedures

While using functions and procedures, the Grant statement can be used to grant users the ability to execute the functions and procedures in MySQL.

Granting Execute Privilege: Execute privilege gives the ability to execute a function or procedure.

Syntax:

```
GRANT EXECUTE ON [ PROCEDURE | FUNCTION ] object TO user;
```

```
GRANT EXECUTE ON FUNCTION function_name TO 'Amit'@'localhost';
```

Example for granting **EXECUTE** privilege on a function named “**CalculateSalary**”:

```
GRANT EXECUTE ON FUNCTION CalculateSalary TO 'Amit'@'localhost';
```

6. 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';
```

Output:

```
GRANTS FOR Amit@localhost
```

```
GRANT USAGE ON *.* TO `SUPER`@`localhost`
```

Revoking Privileges from Users

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.

Various Ways of Revoking Privileges From a User

1. Revoking SELECT Privilege

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 FROM 'Amit'@'localhost';
```

2. Revoking Multiple Privileges

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 FROM  
'Amit'@'localhost';
```

3. Revoking All Privileges

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 FROM 'Amit'@'localhost';
```

4. Revoking Privileges on Functions/Procedures

While using functions and procedures, the revoke statement can be used to revoke the privileges from users which have been EXECUTE privileges in the past.

Syntax:

```
REVOKE EXECUTE ON [ PROCEDURE | FUNCTION ] object FROM User;
```

5. Revoking EXECUTE Privileges on a Function in MySQL

If there is a function called “CalculateSalary” and you want to revoke EXECUTE access to the user named Amit, then the following revoke statement should be executed.

```
REVOKE EXECUTE ON FUNCTION Calculatesalary FROM 'Amit'@'localhost';
```

6. Revoking EXECUTE Privilege to a Users on a Procedure in MySQL

If there is a procedure called “DBMSProcedure” and you want to revoke EXECUTE access to the user named Amit, then the following revoke statement should be executed.


```
REVOKE EXECUTE ON PROCEDURE DBMSProcedure FROM  
'Amit'@'localhost';
```

PL/SQL Function

The PL/SQL Function is very similar to PL/SQL Procedure. The main difference between procedure and a function is, a function must always return a value, and on the other hand a procedure may or may not return a value. Except this, all the other things of PL/SQL procedure are true for PL/SQL function too.

Syntax to create a function:

```
CREATE [OR REPLACE] FUNCTION function_name [parameters]  
[(parameter_name [IN | OUT | IN OUT] type [, ...])]  
RETURN return_datatype  
{IS | AS}  
BEGIN  
    < function_body >  
END [function_name];
```

Here:

- **Function_name:** specifies the name of the function.
- **[OR REPLACE]** option allows modifying an existing function.
- The **optional parameter list** contains name, mode and types of the parameters.
- **IN** represents that value will be passed from outside and
- **OUT** represents that this parameter will be used to return a value outside of the function.
- An **IN OUT** parameter passes an initial value to a subprogram and returns an updated value to the caller.

The function must contain a return statement.

- **RETURN** clause specifies that data type you are going to return from the function.
- **Function_body** contains the executable part.

- The AS keyword is used instead of the IS keyword for creating a standalone function.

PL/SQL Function Example

Let's see a simple example to **create a function**.

```
create or replace function adder(n1 in number, n2 in number)
return number
is
n3 number(8);
begin
n3 :=n1+n2;
return n3;
end;
/
```

Now write another program to **call the function**.

```
DECLARE
    n3 number(2);
BEGIN
    n3 := adder(11,22);
    dbms_output.put_line('Addition is: ' || n3);
END;
/
```

Output:

```
Addition is: 33
Statement processed.
0.05 seconds
```

Write a PL/SQL Function to find reverse of a given number.

```
DECLARE
    num int;
    c int;
    n int;
```

```
rev int:=0;
rem int;
FUNCTION reverse_n( num IN int)
RETURN int
IS
z int;
BEGIN
    n := num;
    WHILE (n > 0)
    LOOP
        rem := mod(n, 10);
        rev := (rev * 10) + rem;
        n := trunc(n / 10);
    END LOOP;
    z := rev;
    RETURN z;
END ;
BEGIN
    num := &int;
    c := reverse_n(num);
    dbms_output.put_line('The reverse of the given number is ' || c);
END;
/
```

Command Prompt - sqlplus

```
SQL> @F:\VJIT\DBMS\LAB\PLSQL\rev.sql;
Enter value for int: 6543
old 24:      num := &int;
new 24:      num := 6543;
The reverse of the given number is  3456

PL/SQL procedure successfully completed.

SQL>
```

PL/SQL function example using table

```
CREATE TABLE Employees (  
    emp_id NUMBER,  
    emp_name VARCHAR2(50),  
    emp_salary NUMBER,  
    emp_dept VARCHAR2(50)  
);  
INSERT INTO Employees VALUES (1, 'John Doe', 50000, 'HR');  
INSERT INTO Employees VALUES (2, 'Jane Smith', 60000, 'IT');  
INSERT INTO Employees VALUES (3, 'Bob Johnson', 45000, 'Finance');  
INSERT INTO Employees VALUES (4, 'Dark Knight', 100000, 'Vigilance');  
INSERT INTO Employees VALUES (5, 'R2D2', 234, 'Mechanical Maintenance');  
CREATE OR REPLACE FUNCTION total_salary  
RETURN INTEGER  
AS  
    total INTEGER:=0;  
BEGIN  
    SELECT SUM(emp_salary)  
    INTO total  
    FROM Employees;  
  
    RETURN total;  
END;  
/  
  
DECLARE  
totalsal INTEGER;  
BEGIN  
totalsal:=total_salary();  
    DBMS_OUTPUT.PUT_LINE('The total expenditure on employees is ' || totalsal);  
END;  
/
```

```
Command Prompt - sqlplus;  
SQL> @F:\VJIT\DBMS\LAB\PLSQL\funtable.sql;  
  
Function created.  
  
The total expenditure on employees is 255234  
  
PL/SQL procedure successfully completed.  
  
SQL>
```

PL/SQ L Recursive Function

You already know that a program or a subprogram can call another subprogram. When a subprogram calls itself, it is called recursive call and the process is known as recursion.

Example to calculate the factorial of a number

Let's take an example to calculate the factorial of a number. This example calculates the factorial of a given number by calling itself recursively.

DECLARE

```
num number;  
factorial number;
```

FUNCTION fact(x number)

RETURN number

IS

```
f number;
```

BEGIN

```
IF x=0 THEN
```

```
    f := 1;
```

ELSE

```
    f := x * fact(x-1);
```

END IF;

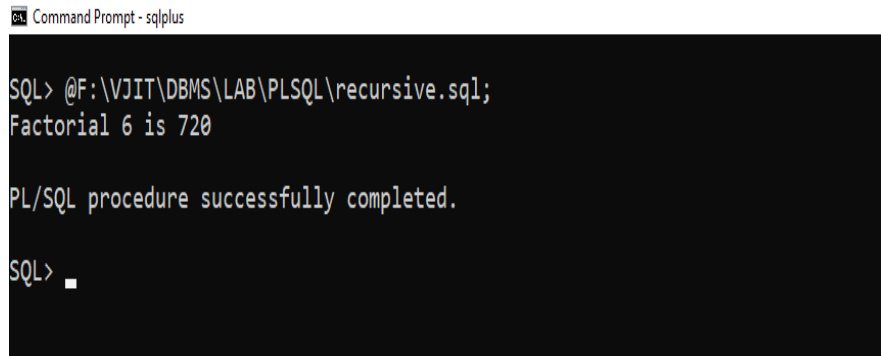
RETURN f;

END;

BEGIN

```
num:= 6;
```

```
factorial := fact(num);  
dbms_output.put_line(' Factorial ' || num || ' is ' || factorial);  
END;  
/
```



```
SQL> @F:\VJIT\DBMS\LAB\PLSQL\recursive.sql;  
Factorial 6 is 720  
  
PL/SQL procedure successfully completed.  
  
SQL> ■
```

PL/SQL Drop Function

Syntax for removing your created function:

If you want to remove your created function from the database, you should use the following syntax.

DROP FUNCTION function_name;

PL/SQL Procedure

The PL/SQL stored procedure or simply a procedure is a PL/SQL block which performs one or more specific tasks. It is just like procedures in other programming languages.

The procedure contains a header and a body.

- **Header:** The header contains the name of the procedure and the parameters or variables passed to the procedure.
- **Body:** The body contains a declaration section, execution section and exception section similar to a general PL/SQL block.

How to pass parameters in procedure:

When you want to create a procedure or function, you have to define parameters. There are three ways to pass parameters in procedure:

1. **IN parameters:** The IN parameter can be referenced by the procedure or function. The value of the parameter cannot be overwritten by the procedure or the function.
2. **OUT parameters:** The OUT parameter cannot be referenced by the procedure or function, but the value of the parameter can be overwritten by the procedure or function.
3. **INOUT parameters:** The INOUT parameter can be referenced by the procedure or function and the value of the parameter can be overwritten by the procedure or function.

A procedure may or may not return any value.

PL/SQL Create Procedure

Syntax for creating procedure:

```
CREATE [OR REPLACE] PROCEDURE procedure_name  
    [parameters] [(parameter_name [IN | OUT | IN OUT] type [, ...])]  
  
IS  
    [declaration_section]  
BEGIN  
    executable_section  
[EXCEPTION  
    exception_section]  
END [procedure_name];
```

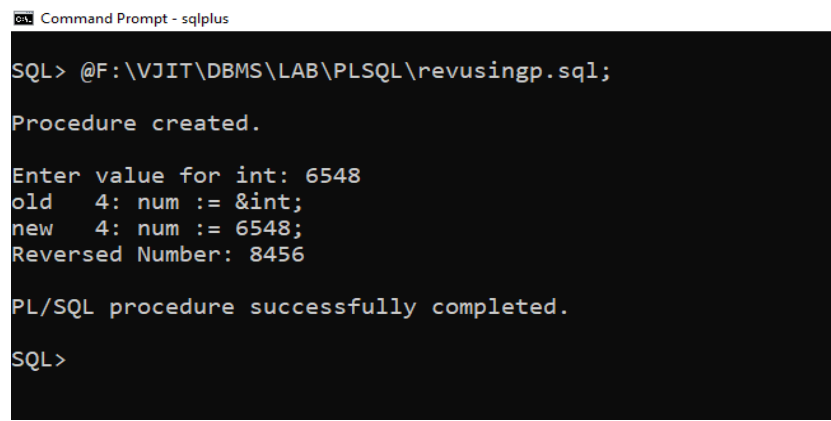
Create a procedure to find reverse of a given number.

```
CREATE OR REPLACE PROCEDURE reverse_number (num int) IS  
    rnum    int := num;  
    rev int := 0;  
    rem    int;  
BEGIN  
    -- Loop until the number becomes 0  
    WHILE rnum > 0 LOOP  
        rem := rnum MOD 10;      -- Get the last digit  
        rev := (rev * 10) + rem; -- Append it to the reverse  
    END
```

```

    rnum := TRUNC(rnum / 10);    -- Remove the last digit
END LOOP;
-- Output the reversed number
DBMS_OUTPUT.PUT_LINE('Reversed Number: ' || rev);
END;
/
-- To test the procedure
DECLARE
    num int;
BEGIN
    num := &int;
    reverse_number(num);
END;
/

```



```

SQL> @F:\VJIT\DBMS\LAB\PLSQL\revusingp.sql;

Procedure created.

Enter value for int: 6548
old 4: num := &int;
new 4: num := 6548;
Reversed Number: 8456

PL/SQL procedure successfully completed.

SQL>

```

Parameter Modes in PL/SQL Subprograms

The following table lists out the parameter modes in PL/SQL subprograms –

S.No	Parameter Mode & Description
1	IN An IN parameter lets you pass a value to the subprogram. It is a read-only parameter. Inside the subprogram, an IN parameter acts like a constant. It cannot be assigned a value. You can pass a constant, literal, initialized variable, or expression as an IN parameter. You can also initialize it to a default value; however, in that case, it is omitted from the subprogram call. It is the default mode of parameter passing. Parameters are passed by reference.

2	OUT An OUT parameter returns a value to the calling program. Inside the subprogram, an OUT parameter acts like a variable. You can change its value and reference the value after assigning it. The actual parameter must be variable and it is passed by value.
3	IN OUT An IN OUT parameter passes an initial value to a subprogram and returns an updated value to the caller. It can be assigned a value and the value can be read. The actual parameter corresponding to an IN OUT formal parameter must be a variable, not a constant or an expression. Formal parameter must be assigned a value. Actual parameter is passed by value.

IN , OUT Parameter Passing to Procedure

DECLARE

a number;

b number;

c number;

PROCEDURE findMin(x IN number, y IN number, z OUT number) IS

BEGIN

IF x < y THEN

z:= x;

ELSE

z:= y;

END IF;

END;

BEGIN

a:= &a;

b:= &b;

findMin(a, b, c);

dbms_output.put_line(' Minimum of (a,b) : ' || c);

END;

/


```
Command Prompt - sqlplus

SQL> @F:\VJIT\DBMS\LAB\PLSQL\insertp.sql;
Enter value for a: 77
old 14: a:= &a;
new 14: a:= 77;
Enter value for b: 55
old 15: b:= &b;
new 15: b:= 55;
Minimum of (a,b) : 55

PL/SQL procedure successfully completed.

SQL> _
```

IN OUT Parameter Passing to Procedure

DECLARE

a number;

PROCEDURE squareNum(x IN OUT number) IS

BEGIN

x := x * x;

END;

BEGIN

a:= &a;

squareNum(a);

dbms_output.put_line(' Square of (a): ' || a);

END;

/

```
Command Prompt - sqlplus

SQL> @F:\VJIT\DBMS\LAB\PLSQL\inout.sql;
Enter value for a: 5
old 8: a:= &a;
new 8: a:= 5;
Square of (a): 25

PL/SQL procedure successfully completed.

SQL> _
```

Create procedure using Database Table

In this example, we are going to insert record in user table. So you need to create user table first.

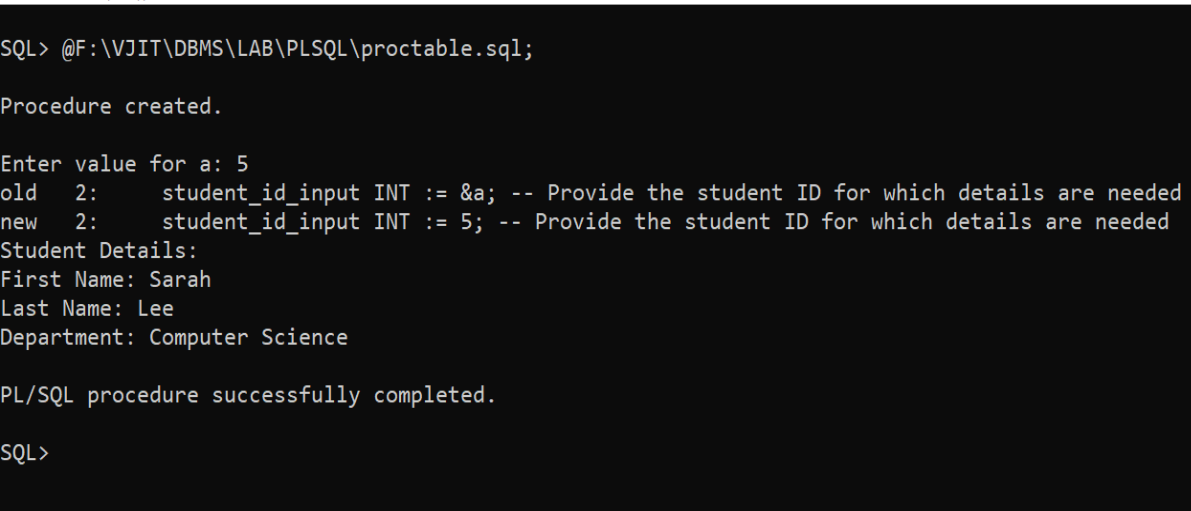
Table creation & Insert data:

```
CREATE TABLE students (  
    student_id INT,  
    first_name VARCHAR(50),  
    last_name VARCHAR(50),  
    department VARCHAR(50),  
    age INT  
);  
INSERT INTO VALUES (1, 'John', 'Doe', 'Computer Science', 20);  
INSERT INTO VALUES (2, 'Jane', 'Smith', 'Mathematics', 22),  
INSERT INTO VALUES (3, 'Michael', 'Johnson', 'Biology', 21),  
INSERT INTO VALUES (4, 'Emily', 'Brown', 'Physics', 23),  
INSERT INTO VALUES (5, 'Sarah', 'Lee', 'Computer Science', 22);
```

PL/SQL program to call procedure

```
CREATE OR REPLACE PROCEDURE get_student_details (  
    student_id_in IN INT,  
    first_name_out OUT VARCHAR2,  
    last_name_out OUT VARCHAR2,  
    department_out OUT VARCHAR2  
)  
AS  
BEGIN  
    SELECT first_name, last_name, department  
    INTO first_name_out, last_name_out, department_out  
    FROM students  
    WHERE student_id = student_id_in;  
END;  
/  
DECLARE  
    student_id_input INT := &a; -- Provide the student ID for which details are needed  
    first_name_output VARCHAR2(50);
```

```
last_name_output VARCHAR2(50);
department_output VARCHAR2(50);
BEGIN
    get_student_details(student_id_input, first_name_output, last_name_output,
department_output);
    DBMS_OUTPUT.PUT_LINE('Student Details:');
    DBMS_OUTPUT.PUT_LINE('First Name: ' || first_name_output);
    DBMS_OUTPUT.PUT_LINE('Last Name: ' || last_name_output);
    DBMS_OUTPUT.PUT_LINE('Department: ' || department_output);
END;
/
```



```
Command Prompt - sqlplus
SQL> @F:\VJIT\DBMS\LAB\PLSQL\proctable.sql;

Procedure created.

Enter value for a: 5
old 2:      student_id_input INT := &a; -- Provide the student ID for which details are needed
new 2:      student_id_input INT := 5; -- Provide the student ID for which details are needed
Student Details:
First Name: Sarah
Last Name: Lee
Department: Computer Science

PL/SQL procedure successfully completed.

SQL>
```

PL/SQL Drop Procedure

Syntax for drop procedure

DROP PROCEDURE procedure_name;

Example of drop procedure

DROP PROCEDURE pro1;

Triggers in PL/SQL

Trigger is a predefined program that automatically gets executed in response to specific events occurring in a database. These events could be associated with tables, views, schemas, or the entire database. A trigger is stored in the database and gets invoked repeatedly when specified conditions are met. They are programmed to respond to different events, including database manipulation (DML) statements, database definition (DDL) statements, and various database operations like LOGON, LOGOFF, STARTUP, or SHUTDOWN.

Benefits of Triggers

There are many benefits of triggers which include:

1. They automate the creation of derived column values, reducing manual efforts.
2. They ensure referential integrity, maintaining consistency between the tables.
3. Triggers also enable event logging, providing valuable insights into usage patterns.
4. Triggers support synchronous replication of tables, ensuring real-time data consistency across different parts of database.
5. They also contribute to enforcing security measures and safeguarding sensitive information by preventing the execution of invalid transactions.

Creating Triggers

To create a trigger, we can use the below syntax:

```
CREATE [OR REPLACE] TRIGGER name_of_trigger
{ BEFORE | AFTER | INSTEAD OF }
{ INSERT [OR] | UPDATE [OR] | DELETE }
[OF col_name]
ON table_name
[REFERENCING OLD AS o NEW AS n]
[FOR EACH ROW]
WHEN (condition)
BEGIN
-- Declaration statements
-- These are the optional statements where you can declare the variables.
```

- Executable statements
- These are the actions/operations the trigger performs when it is fired automatically.

EXCEPTION

- Exception handling statements
- These handle errors that may occur during the execution of triggers.

END;

CREATE [OR REPLACE] TRIGGER name_of_trigger:

- **CREATE:** This starts the creation of a trigger.
- **OR REPLACE:** This is an optional clause that allows updating an existing trigger with a similar name.
- **TRIGGER trigger_name:** This mentions the name of the trigger being created or updated.

2. {BEFORE | AFTER | INSTEAD OF}:

- **BEFORE:** This means that the trigger gets executed before triggering event
- **AFTER:** This means that the trigger gets executed after triggering event.
- **INSTEAD OF:** This is used for triggers on views, allowing us to replace default action with the trigger's defined custom action.

3. {INSERT [OR] | UPDATE [OR] | DELETE}:

This clause mentions the type of DML operation that the trigger will respond to. These are the available options:

- **INSERT:** The trigger executes in response to an INSERT statement.
- **UPDATE:** The trigger executes in response to an UPDATE statement.
- **DELETE:** The trigger executes in response to a DELETE statement.

4. [OF col_name]:

This clause is an optional clause mentioning the name of column affected by the trigger.

5. [ON table_name]:

This clause is an optional clause that associates trigger with a specific table.

6. [REFERENCING OLD AS o NEW AS n]:

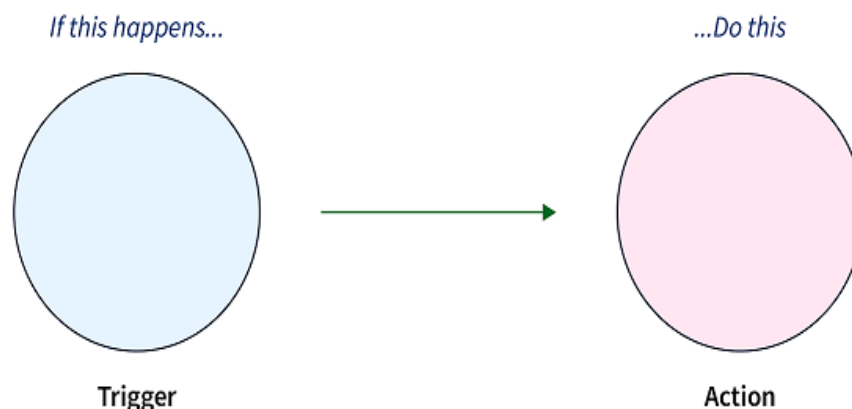
This clause is an optional clause allowing the references to old and new values in the trigger body.

7. [FOR EACH ROW]:

This is an optional clause indicating a row-level trigger. This gets executed once for each affected row and without this clause, the trigger is considered a statement-level trigger, executing once independent of number of affected rows.

8. WHEN (condition):

This is an optional clause specifying a condition that, triggers the action defined in the trigger body when true.

**Types of PL/SQL Triggers**

Trigger timing and operations forms different combinations such as

1. BEFORE INSERT
2. BEFORE DELETE
3. BEFORE UPDATE
4. AFTER INSERT
5. AFTER DELETE
6. AFTER UPDATE

are known as conditional triggers.

Conditional Trigger: Before

Trigger is activated before the operation on the table or view is performed.

Example of PL/SQL Triggers

```
CREATE TABLE Stud (  
    Id INT,  
    Name VARCHAR2(20),  
    Score INT  
);  
CREATE TABLE Result (  
    Id INT,  
    Name VARCHAR2(20),  
    Score INT  
);
```

```
INSERT INTO Stud (Id, Name, Score) VALUES (1, 'Sam', 800);  
INSERT INTO Stud (Id, Name, Score) VALUES (2, 'Ram', 699);  
INSERT INTO Stud (Id, Name, Score) VALUES (3, 'Tom', 250);  
INSERT INTO Stud (Id, Name, Score) VALUES (4, 'Om', 350);  
INSERT INTO Stud (Id, Name, Score) VALUES (5, 'Jay', 750);
```

Command Prompt - sqlplus

```
SQL> select * from Stud;
```

ID	NAME	SCORE
1	Sam	800
2	Ram	699
3	Tom	250
4	Om	350
5	Jay	750

```
SQL> select * from Result;
```

```
no rows selected
```

```
SQL>
```

BEFORE INSERT Trigger

```
-- BEFORE INSERT trigger  
CREATE OR REPLACE TRIGGER BEFORE_INSERT  
BEFORE INSERT ON Stud  
FOR EACH ROW  
BEGIN  
    INSERT INTO Result (Id, Name, Score)  
    VALUES (:NEW.Id, :NEW.Name, :NEW.Score);
```

```
END;
```

```
/
```

```
INSERT INTO Stud (Id, Name, Score) VALUES (6, 'Arjun', 500);
```

Command Prompt - sqlplus

```
SQL> INSERT INTO Stud (Id, Name, Score) VALUES (6, 'Arjun', 500);
```

```
1 row created.
```

```
SQL> select * from Stud;
```

ID	NAME	SCORE
1	Sam	800
2	Ram	699
3	Tom	250
4	Om	350
5	Jay	750
6	Arjun	500

```
6 rows selected.
```

```
SQL> select * from Result;
```

ID	NAME	SCORE
6	Arjun	500

```
SQL>
```

BEFORE DELETE Trigger

```
-- BEFORE DELETE trigger
CREATE OR REPLACE TRIGGER BEFORE_DELETE
BEFORE DELETE ON Stud
FOR EACH ROW
BEGIN
    INSERT INTO Result (Id, Name, Score)
    VALUES (:OLD.Id, :OLD.Name, :OLD.Score);
END;
/
```

```
DELETE FROM Stud WHERE Id = 3;
```



```

Command Prompt - sqlplus
SQL> @F:\VJIT\DBMS\LAB\PLSQL\beforedelete.sql;
Trigger created.
SQL> DELETE FROM Stud WHERE Id = 3;
1 row deleted.
SQL> select * from Stud;

   ID NAME      SCORE
-----
    1 Sam        800
    2 Ram        699
    4 Om          350
    5 Jay        750
    6 Arjun       500

SQL> select * from Result;

   ID NAME      SCORE
-----
    6 Arjun       500
    3 Tom         250

SQL>

```

BEFORE UPDATE Trigger

```

-- BEFORE UPDATE trigger
CREATE OR REPLACE TRIGGER BEFORE_UPDATE
BEFORE UPDATE ON Stud
FOR EACH ROW
BEGIN
    INSERT INTO Result (Id, Name, Score)
    VALUES (:OLD.Id, :OLD.Name, :OLD.Score);
END;
/

```

```
UPDATE Stud SET Score = 900 WHERE Id = 5;
```

```

Command Prompt - sqlplus
SQL> @F:\VJIT\DBMS\LAB\PLSQL\beforeupdate.sql;
Trigger created.
SQL> UPDATE Stud SET Score = 900 WHERE Id = 5;
1 row updated.
SQL> select * from Stud;

   ID NAME      SCORE
-----
    1 Sam        800
    2 Ram        699
    4 Om          350
    5 Jay        900
    6 Arjun       500

SQL> select * from Result;

   ID NAME      SCORE
-----
    6 Arjun       500
    3 Tom         250
    5 Jay         750

SQL>

```

Conditional Trigger: After

Trigger is activated After the operation on the table or view is performed.

```
CREATE TABLE Stud (  
    Id INT,  
    Name VARCHAR2(20),  
    Score INT  
);  
CREATE TABLE Result (  
    Id INT,  
    Name VARCHAR2(20),  
    Score INT  
);
```

```
INSERT INTO Stud (Id, Name, Score) VALUES (1, 'Sam', 800);  
INSERT INTO Stud (Id, Name, Score) VALUES (2, 'Ram', 699);  
INSERT INTO Stud (Id, Name, Score) VALUES (3, 'Tom', 250);  
INSERT INTO Stud (Id, Name, Score) VALUES (4, 'Om', 350);  
INSERT INTO Stud (Id, Name, Score) VALUES (5, 'Jay', 750);
```

Command Prompt - sqlplus

```
SQL> select * from Stud;
```

ID	NAME	SCORE
1	Sam	800
2	Ram	699
3	Tom	250
4	Om	350
5	Jay	750

```
SQL> select * from Result;
```

```
no rows selected
```

```
SQL>
```

AFTER DELETE Trigger

```
-- AFTER DELETE trigger  
CREATE OR REPLACE TRIGGER AFTER_DELETE  
AFTER DELETE ON Stud  
FOR EACH ROW  
BEGIN  
    INSERT INTO Result (Id, Name, Score)  
    VALUES (:OLD.Id, :OLD.Name, :OLD.Score);  
END;  
/
```

```
DELETE FROM Stud WHERE Id = 4;
```

```
Command Prompt - sqlplus

SQL> @F:\VJIT\DBMS\LAB\PLSQL\afterdelete.sql;
Trigger created.

SQL> DELETE FROM Stud WHERE Id = 4;
1 row deleted.

SQL> select * from Stud;

   ID NAME      SCORE
-----
    1 Sam        800
    2 Ram        699
    3 Tom        250
    5 Jay        750

SQL> select * from Result;

   ID NAME      SCORE
-----
    4 Om        350

SQL> _
```

AFTER UPDATE Trigger

```
-- AFTER UPDATE trigger
CREATE OR REPLACE TRIGGER AFTER_UPDATE
AFTER UPDATE ON Stud
FOR EACH ROW
BEGIN
    INSERT INTO Result (Id, Name, Score)
    VALUES (:NEW.Id, :NEW.Name, :NEW.Score);
END;
/
```

```
UPDATE Stud SET Score = 1050 WHERE Id = 5;
```

```
Command Prompt - sqlplus
SQL> @F:\VJIT\DBMS\LAB\PLSQL\afterupdate.sql;

Trigger created.

SQL> UPDATE Stud SET Score = 1050 WHERE Id = 5;

1 row updated.

SQL> select * from Stud;

   ID NAME          SCORE
-----
   1 Sam            800
   2 Ram            699
   3 Tom            250
   5 Jay           1050

SQL> select * from Result;

   ID NAME          SCORE
-----
   4 Om             350
   5 Jay           1050

SQL>
```

AFTER INSERT Trigger

```
-- AFTER INSERT trigger
CREATE OR REPLACE TRIGGER BEFORE_INSERT
AFTER INSERT ON Stud
FOR EACH ROW
BEGIN
    INSERT INTO Result (Id, Name, Score)
    VALUES (:NEW.Id, :NEW.Name, :NEW.Score);
END;
/
```

```
INSERT INTO Stud (Id, Name, Score) VALUES (6, 'Arjun', 500);
```

Command Prompt - sqlplus

```
SQL> @F:\VJIT\DBMS\LAB\PLSQL\afterinsert.sql;
```

```
Trigger created.
```

```
SQL> INSERT INTO Stud (Id, Name, Score) VALUES (6, 'Arjun', 500);
```

```
1 row created.
```

```
SQL> select * from Stud;
```

ID	NAME	SCORE
1	Sam	800
2	Ram	699
3	Tom	250
5	Jay	1050
6	Arjun	500

```
SQL> select * from Result;
```

ID	NAME	SCORE
4	Om	350
5	Jay	1050
6	Arjun	500

```
SQL>
```

Another Example of PL/SQL Triggers

Let's understand PL/SQL Triggers with the help of an example. We will create the "employees" table having three columns:

```
CREATE TABLE employees (  
    id NUMBER PRIMARY KEY,  
    name VARCHAR2(100),  
    salary NUMBER  
);
```

Now, let's create a trigger that computes and shows the salary difference whenever an update operation is performed on the "employees" table:

```
CREATE OR REPLACE TRIGGER employees_table_trigger
BEFORE INSERT OR UPDATE ON employees
FOR EACH ROW
BEGIN
  DECLARE
    difference_in_salary NUMBER;
  BEGIN
    difference_in_salary := :NEW.salary - :OLD.salary;

    dbms_output.put_line('Old salary of the employee: ' || :OLD.salary);
    dbms_output.put_line('New salary of the employee: ' || :NEW.salary);
    dbms_output.put_line('Calculated salary difference: ' || difference_in_salary);
  END;
END;
/
```

Triggering a Trigger

Let's insert a few of the entries in the "employees" table:

```
INSERT INTO employees (id,name,salary)
VALUES (1, 'Kriti', 7500.00 );

INSERT INTO employees (id,name,salary)
VALUES (2, 'Manaya', 2000.00 );

INSERT INTO employees (id,name,salary)
VALUES (3, 'Sonya', 10000.00 );
```

When a record is inserted in the "employee" table, the above-created trigger will be fired and it will display the following output –

Command Prompt - sqlplus;

```
SQL> @F:\VJIT\DBMS\LAB\PLSQL\triggers.sql;
```

Trigger created.

```
SQL> INSERT INTO employees (id,name,salary)
2 VALUES (1, 'Kriti', 7500.00 );
```

Old salary of the employee:

New salary of the employee: 7500

Calculated salary difference:

1 row created.

```
SQL> INSERT INTO employees (id,name,salary)
2 VALUES (2, 'Manaya', 2000.00 );
```

Old salary of the employee:

New salary of the employee: 2000

Calculated salary difference:

1 row created.

```
SQL> INSERT INTO employees (id,name,salary)
2 VALUES (3, 'Sonya', 10000.00 );
```

Old salary of the employee:

New salary of the employee: 10000

Calculated salary difference:

Windows taskbar with search bar and icons.

Command Prompt - sqlplus;

```
SQL> UPDATE employees
```

```
2 SET salary = salary + 5000
```

```
3 WHERE id = 2;
```

Old salary of the employee: 2000

New salary of the employee: 7000

Calculated salary difference: 5000

1 row updated.

```
SQL> █
```