SQL

Structured Query Language

Niranjan Reddy Palli

1. WHAT IS SQL ? WHY SQL ?
    a. What is Data ?
    b. What is a Database ?
    c. What is a DBMS ?
    d. What is SQL ? Why SQL ?
2. SQL ARCHITECTURE
3. BASIC COMMANDS OF SQL
    a. DQL
    b. DDL
    c. DML
    d. DCL
    e. TCL
4. DATATYPES
    a. Numeric Datatypes
    b. Character Datatypes
    c. Date-Time Datatypes
    d. Binary Datatypes
5. CONSTRAINTS
    a. Not Null
    b. Unique
    c. Default
    d. Check
    e. PRIMARY KEY
    f. FOREIGN KEY
6. OPERATORS
7. CONCEPTS OF SQL
    - Select
    - Distinct
    - Alias
    - Comments
    - Where
    - Is Null / Is Not Null
    - AND / OR
    - IN / Not In
    - Between
    - Order By
    - Limit
    - Like / Not Like
    - Wildcards
    - Aggregate Functions
    - Group By
    - Having
    - Query Processing Order
    - Joins
    - Union / Union All
    - Sub Query
    - Temporary Tables
    - Case Statements
    - Cloning Tables
    - Window Functions
    - Views
    - Index
    - Stored Procedures
    - CTE (With Clause)
    - Functions
    - String Functions
    - Numeric Functions
    - Date and Time Functions
    - Advanced Functions

What is SQL ? Why SQL ?

To answer this, we need to know what **DATA** really is ?

What is **DATA** ?

DATA means information. Data can be information or facts related to anything in consideration.

For eg : Your name, age, height, weight etc are some facts or information about ‘YOU’.

Data is stored in a DATABASE.

Types of DATA

1. Qualitative Data :

   Represent some characteristics or attributes. They depict descriptions that may be observed but cannot be computed or calculated. Sometimes referred as dimensions.

   Eg : Your name, hometown, email-id, address etc.

2. Quantitative Data

   Represented numerically and can be measured. Calculations can be performed. Also called as facts.

   Eg : Marks, salary, age etc.

What is a **DATABASE** ?

A DATABASE is a systematic collection of data for easy access, management and manipulation.

- Supports electronic storage and manipulation of data.
- Make data management easy.

Eg : Facebook stores all the data related to you, your friends, activities, messages, adds and lot more under your account id in their database. Later they use this data for different analysis purposes.

Types of DATABASES

1. Centralized Database
2. Distributed Database
3. Relational Database
4. NoSQL Database
5. Cloud Database
6. Hierarchial Database etc.

Databases are managed by DBMS.

What is a DBMS ?

DBMS is a DataBase Management System. It is a collection of programs that enables its users to access databases, manipulate, report and represent data. DBMS also helps to control access to database.
Eg : MySQL, SSMS

We are going to use **Relational Databases**, which is based on relational data model and stores data in the form of **ROWS** and **COLUMNS**, which together form a **TABLE**. Eg : MySQL, MS SQL Server, Oracle

RDBMS - **Relational Data Base Management System** is the DBMS used to access the data from **Relational Databases**.

To access the DATA stored in a DATABASE, we need to give inputs or commands to the DBMS.

SQL is the language that is used to give inputs or commands to the **RDBMS**, in order to access the **DATA** stored in a **Relational DATABASE**.

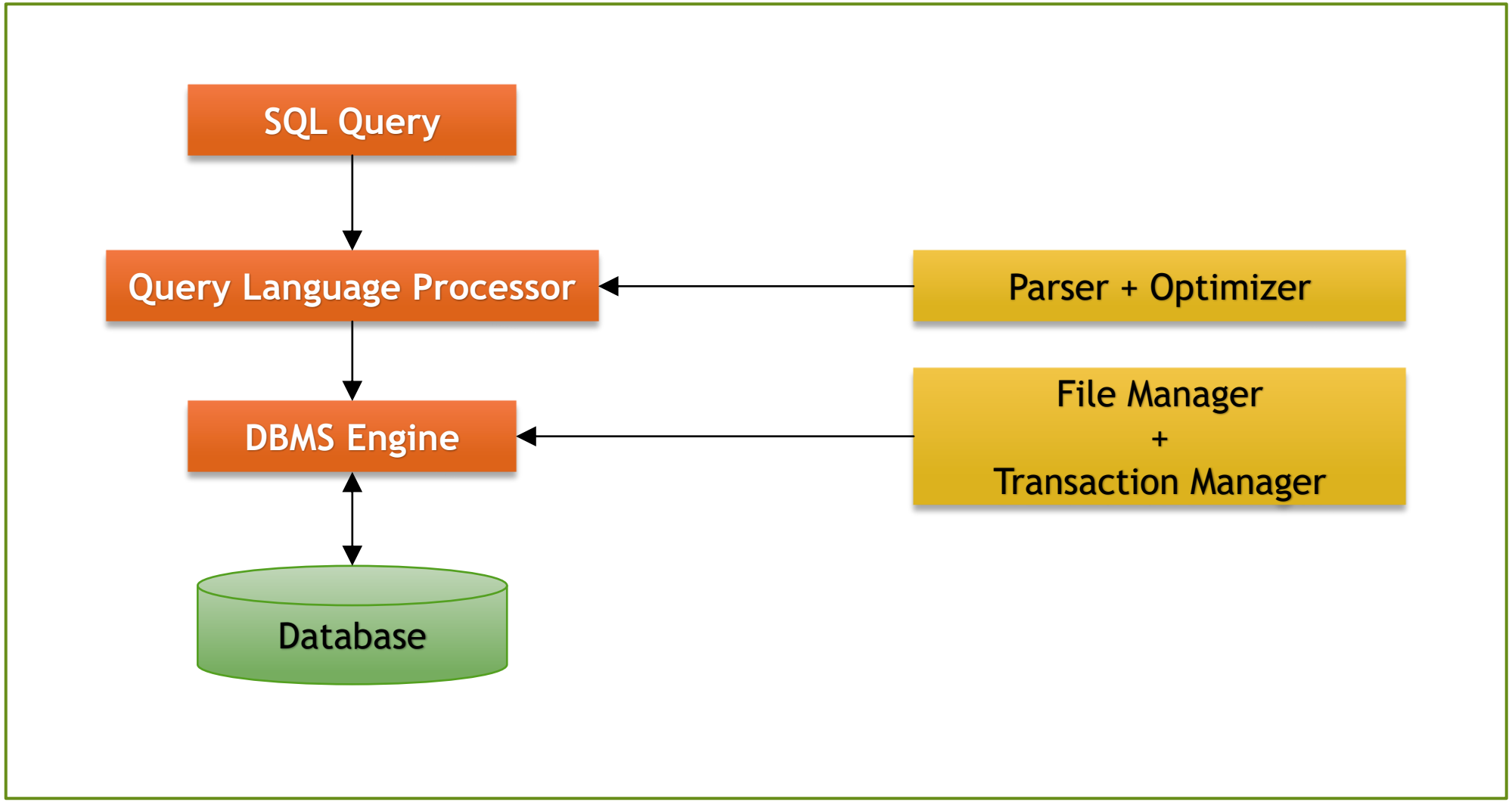
What is SQL ?

- SQL is a Structured Query Language, which is a computer language used for storing, manipulating and retrieving data stored in relational database.
- SQL is an ANSI (American National Standards Institute) standard, but there are many different versions of the SQL language.
- SQL is the standard language for dealing with relational databases.
- SQL is used to create, insert, update, delete, search/query the records from a table in a database.

Why SQL ?

- Allows users to access data in relational database management systems.
- Allows users to define the data in database and manipulate that data.
- Allows users to create and drop databases and tables.
- Allows users to create views, stored procedures, functions in a database.
- Allows users to set permissions on tables, procedures and views.
- Allows to embed within other languages using SQL modules, libraries and pre-compilers.

SQL Architecture



Basic SQL commands

DQL	DDL	DML	DCL	TCL
• Select	▪ Create ▪ Alter ▪ Drop ▪ Truncate	• Insert • Update • Delete	• Grant • Revoke	• Commit • Rollback • Savepoint

DQL - Data Query Language

1. Select

```
Select column1, column2, column3
from table_name ;

Select * from table_name ;
```

DDL - Data Definition Language

1. Create

```
Create table table_name
 (
      Column1   Datatype,
      Column2  Datatype,
      COlumn3  Datatype,
      .
      .
      Column100 Datatype,
      Primary Key (One or more Columns)
 ) ;

Create database database_name ;
```

2. Alter

```
Alter table table_name
ADD|MODIFY Column_name  Datatype ;

Alter table table_name
DROP Column_name ;

Alter table table_name
RENAME to new_table_name ;
```

3. Drop

```
Drop table table_name;
Drop database database_name ;
```

4. Truncate

```
Truncate table table_name ;
```

DML - Data Manipulation Language

1. Insert

```
Insert into table_name (column1, column2)
values (value1, value2) ;

Insert into table_name values
(value1, value2),
(value1, value2),
.
.
(value100, value100) ;
```

2. Update

```
Update table_name
set column1 = value1, column2 = value2,..
where condition ;
```

3. Delete

```
Delete from table_name
Where condition ;

Delete from table_name ;
```

DCL - Data Control Language

1. Grant

```
Grant privilege_name on object_name
To [user_name | public | role_name] ;

Grant select, insert on Students
To Ram ;
```

2. Revoke

```
Revoke privilege_name on object_name
from [user_name | public | role_name] ;

Revoke select, insert on Students
from Ram ;
```

TCL - Transaction Control Language

1. Commit

The COMMIT command saves all the transactions to the database since the last COMMIT or ROLLBACK command.

```
DELETE FROM Student WHERE AGE = 20 ;
COMMIT ;
```

(Saves the changes made to the table Student)

2. Rollback

The process of reversing changes is called rollback. This command can only be used to undo transactions since the last COMMIT or ROLLBACK command was issued.

```
DELETE FROM Student WHERE AGE = 20 ;
ROLLBACK ;
```

3. Savepoint

Creates points within the groups of transactions in which it has to ROLLBACK.

A SAVEPOINT is a point in a transaction to which you can roll the transaction back to a certain point without rolling back the entire transaction.

```
SAVEPOINT SAVEPOINT_NAME ;
```

This command is used only in the creation of SAVEPOINT among all the transactions.
In general ROLLBACK is used to undo a group of transactions.

```
ROLLBACK TO SAVEPOINT_NAME;
```

```
SAVEPOINT SP1 ;
//Savepoint created.
DELETE FROM Student WHERE AGE = 20 ;
//deleted
SAVEPOINT SP2 ;
//Savepoint created.
```

```
ROLLBACK TO SP1 ;
//Rollback completed.
```

SQL Data Types

Data type is an attribute that specifies the type of data that a column can store.

Numeric Datatypes

Data Type	Description
Bit	Boolean type with values 0 (false), 1 (true), or NULL. Default is 1.
Tinyint	Tiny whole numbers from 0 to 255.
Smallint	Small whole numbers from -32,768 to 32,768.
Int	Whole numbers from -2,147,483,648 to 2,147,483,648.
Bigint	Large whole numbers from -9,223,372,036, 854,775,808 to 9,223,372,036, 854,775,808.
Decimal	Exact number with a fixed precision (no. of digits) and scale (no. of decimal places) from -10^38 +1 to 10^38 +1. Decimal(10,2) takes upto 10 numbers before point and rounds to 2 decimal points after the decimal point.
Numeric	Same as decimal. Numeric(5, 2)
Float	An approximate number with floating point data. Do not store the exact values specified. They store **an extremely close** approximation of the value
Real	Identical to Float(24). 24 represents precesion (7 digits) and storage size (4 bytes).
Smallmoney	Monetary value from -214,748.3648 to 214,748.3647.
Money	Monetary value from -922,337,203,685,477.5808 to 922,337,203,685,477.5807.

Non Unicode Character Strings Data Types

- Stores data at 1 byte per character

Data Type	Description
Char	Fixed length non-Unicode characters. Max length of 8,000 characters.
Varchar	Variable-length non-Unicode data. Max length of 8,000 characters.
Varchar(max)	Variable-length non-Unicode data. Can hold character data of size 2GB.
Text	Variable-length non-Unicode data. Can hold 2,147,483,647 bytes (2GB) of data or characters.

Unicode Character Strings Data Types

- Stores data at 2 bytes per character. N stands for national.

Data Type	Description
nchar	Fixed length Unicode data. Maximum length of 4,000 characters.
nvarchar	Variable length Unicode data. Maximum length of 4,000 characters.
nvarchar(max)	Variable-length Unicode data. Can hold character data of size 2GB.
ntext	Variable-length Unicode data. Can hold 2,147,483,647 bytes (2GB) of data or characters.

The key difference between **varchar** and **nvarchar** is the way they are stored, varchar is stored as regular 8-bit data (**1 byte per character**) and nvarchar stores data at **2 bytes per character**. Due to this reason, nvarchar can hold up to 4000 characters and it takes double the space as SQL varchar.

If you have requirements to store **UNICODE or multilingual data**, **nvarchar** is the best choice.

Date and Time Datatypes

Data Type	Description
Date	Date in *YYYY-MM-DD* format. Supported range is from 1000-01-01 to 9999-12-31.
Time	Time in *hh:mm:ss* format. Supported range is from 00:00:00 to 23:59:59
Datetime	Date and time in *YYYY-MM-DD hh:mm:ss* format. Supported range is from 1753-01-01 00:00:00 to 9999-12-31 23:59:59.
Smalldatetime	Date and time in *YYYY-MM-DD hh:mm:ss* format. Supported range is from 1900-01-01 00:00:00 to 2079-06-06 23:59:59.

Binary Datatypes

- Stores data in binary format.

Data Type	Description
Binary	Fixed length binary string. Max size of 8000 bytes.
Varbinary	Variable length binary storage. Max size of 8000 bytes.
Varbinary(max)	Variable length binary storage. Can hold character data of size 2GB. Max indicates maximum characters.
Image	Variable length binary image storage. Can hold character data of size 2GB.

➢ **Image, text and ntext data types will be removed in a future version of SQL Server.**

SQL Constraints

- Constraints are rules that help to maintain data and referential integrity.
- Data integrity means that the data in the database is valid.
- Referential integrity means that the relationships between tables are valid.
- Contraints prevent incorrect data from entering into the database and ensures the accuracy and reliability of the data.

Types of Constraints

Not Null

- NOT NULL constraint specifies that NULL values are not accepted in a column.
- A NULL is not the same as no data, rather, it represents unknown data.

To create a new table with not null constraint

```
Create table Students (
ID int not null,
Name varchar(100) not null,
Age int
) ;
```

For already existing columns

```
Alter table students
Modify age int not null ;
```

Unique

- A UNIQUE constraint specifies that the values in a column must be unique.

To create a new table with not null constraint

```
Create table Students (
ID  int  not null  unique,
Name  varchar(100)  not null,
Email  varchar(250)  unique,
Phone_No  varchar(50)
) ;
```

For already existing columns

```
Alter table students
Modify  Phone_no  varchar(50)  unique ;

Alter table students
Add  unique(Pone_no) ;
```

To create unique constraint on 2 or more columns

```
Alter table students
Add  constraint  unq_sudents  unique (Name, phone_no)
;
```

The combination of these columns “name” and “phone_no” is unique -- not the individual columns.

To drop an existing unique constraint

```
Alter table students
Drop  constraint  unq_students ;
```

Default

- A DEFAULT constraint specifies a default column value if no value is specified.

Creating a new table with default constraint

```
Create table Students (
ID  int  not null  unique,
Name  varchar(100)  not null,
Age int default 15
) ;
```

For already existing columns

```
Alter table students
Modify age int default 15;
```

```
Alter table students
Add constraint df_students default 15 for age ;
```
(Does not work in MySQL)

To drop an existing default constraint

```
Alter table students
Alter column age
Drop default ;
```

```
Alter table students
Drop  constraint  df_students ;
```
(Does not work in MySQL)

Check

- CHECK specifies that the values must satisfy certain conditions.

Creating a new table with check constraint

```
Create table Students (
ID  int  not null  unique,
Name  varchar(100)  not null,
Age int check (age > 10)
) ;
```

For already existing columns

```
Alter table students
Modify age int check (age > 10);

Alter table students
Add constraint ck_age check (age > 10) ;
```

To drop an existing check constraint

```
Alter table students
Alter column age
Drop check ;

Alter table students
Drop  check  ck_age ;

Alter table students
Drop constraint ck_age;
```

Primary Key

- A PRIMARY KEY is a field that uniquely identifies each row/record in a table.
- The column values must be unique and cannot be NULL.
- A table can only have one primary key which may consist of single or multiple fields.
- When multiple fields are used as a primary key, they are called a COMPOSITE KEY.

Creating a new table with a Primary Key

```
Create table Students (
ID  int  Primary Key,
Name  varchar(100)  not null,
Age int default 15
) ;
```

```
Create table students (
ID int,
Name  varchar(100)  not null,
Age int default 15,
Primary key(ID, Name)
) ;
```

For already existing columns

```
Alter table students
Add primary key(ID) ;
```

```
Alter table students
Add constraint pk_students primary key (ID, Name);
```

To drop an existing Primary Key

```
Alter table students
Drop  Primary Key;
```

Foreign Key

- A FOREIGN KEY is a key used to link two tables together. This is sometimes called a REFERENCING KEY.
- A Foreign Key is a column or a combination of columns whose values match with a Primary Key in a different table.
- A table can have any number of foreign keys.

Creating a new table with a Foreign Key

```
Create table School_details (
S_ID  int  Primary Key,
School_Name  varchar(100)  not null,
Address varchar(200)
) ;
```

```
Create table Students (
ID  int  Primary Key,
Name  varchar(100)  not null,
Age int default 15,
School_ID int,
Foreign key (school_id) references School_details(S_ID)
) ;
```

Creating a new table with a Foreign Key

```
Create table Students (
ID  int  Primary Key,
Name  varchar(100)  not null,
Age int default 15,
School_ID int,
Constraint fk_st Foreign key (school_id)
references School_details(S_ID)
) ;
```

```
Create table Students (
ID  int  Primary Key,
Name  varchar(100)  not null,
School_ID int Foreign key references School_details(S_ID),
Age int default 15
) ;
```

(Does not work in MySQL)

For already existing columns

```
Alter table students
Add Foreign Key(School_ID) references school_details(S_ID) ;
```

If we don’t give a name to the keys, then a default name like ‘students_ibfk_1’ will be created automatically which can be found in the schema.

```
Alter table students
Add constraint fk_students
foreign key (School_ID) references school_details(S_ID) ;
```

To drop an existing Foreign Key

```
Alter table students
Drop  Foreign key students_ibfk_1;
```

```
Alter table students
Drop constraint students_ibfk_1;
```

Operators

Arithmatic Operators

- **+** Addition (a+b)
- **-** Subtraction (a-b)
- ***** Multiplication (a*b)
- **/** Division (a/b)
- **%** Modulus (a % b)

Comparison Operators

- **=** Equals
- **!=** Not Equals
- **<>** Not Equals
- **>** Greater Than
- **<** Less Than
- **>=** Greater than or Equal to
- **<=** Less than or Equal to

Logical Operators

Operator	Description
ALL	The ALL operator is used to compare a value to all values in another value set
AND	The AND operator allows the existence of multiple conditions in an SQL statement's WHERE clause.
ANY	The AND operator allows the existence of multiple conditions in an SQL statement's WHERE clause.
Between	Used to get the values that are between two given values.
Exists	The AND operator allows the existence of multiple conditions in an SQL statement's WHERE clause.
IN	The IN operator is used to compare a value to a list of literal values that have been specified.
Like	The LIKE operator is used to compare a value to similar values using wildcard operators.
Not	The NOT operator reverses the meaning of the logical operator with which it is used. Eg: NOT EXISTS, NOT BETWEEN, NOT IN, etc. This is a negate operator.
OR	The OR operator is used to combine multiple conditions in an SQL statement's WHERE clause.
Is Null	The NULL operator is used to compare a value with a NULL value.
Unique	The UNIQUE operator searches every row of a specified table for uniqueness (no duplicates).

Concepts of SQL

- Select
- Distinct
- Alias
- Comments
- Where
- Is Null / Is Not Null
- AND / OR
- IN / Not In
- Between
- Order By
- Limit
- Like / Not Like
- Wildcards
- Aggregate Functions
- Group By
- Having
- Query Processing Order
- Joins
- Union / Union All
- Sub Query
- Temporary Tables
- Case Statements
- Cloning Tables
- Window Functions
- Views
- Index
- Stored Procedures
- CTE (With Clause)
- Functions
- String Fuctions
- Numeric Functions
- Date and Time Functions
- Advanced Functions

Select
Select * from students ;
Select ID, name from students ;

Distinct
Select distinct name from students ;
Select distinct * from students ;

Alias
Select Id, name as student_name
From students ;

Select Id, name
From students as s
Left join Marks as m
On s.id = m.id ;

Comments
Used to comment out part of text in the code.
-- unwanted text
‘’’ unwanted text ’’’
/* unwanted text */

Where
Used to filter the data by applying filter conditions.
Select * from students
Where id = 10 ;
Select * from students
Where name = ‘Ram’ ;

Is null
Select * from students
Where age is null ;

Is not null
Select * From students
Where name is not null ;

In
Select * from students
Where id in (1,2,3) ;

Between
Select * from students
Where age between 10 and 20 ;

Order By

Used to sort the output data.

Select * from students
Order by id ;
Select * from students
Order by id **desc** ;
Select * from students
Order by name **asc**, age **desc** ;

Limit

Used to get limited no of records.

Select * from students
Limit 10;

Like / Not Like

Like operator is used in where clause to search for a specified pattern in a column.

Select * From students
Where name **like** 'PATTERN' ;

Select * From students
Where name not **like** 'PATTERN' ;

Wildcards

Used in like clause to search for a specified pattern in a column.

Select * from students
Where name **Like 'A%'** ;

Select * from students`
Where name **not like 'A%'** ;

Wildcard	Description	Example
%	Represents zero or more characters	bl% finds bl, black, blue, and blob
_	Represents any single character	h_t finds hot, hat, and hit
[]	Represents any single character within the brackets	h[oa]t finds hot and hat, but not hit
^	Represents any single character not in the brackets	h[^oa]t finds hit, but not hot and hat
-	Represents any single character within the specified range	c[a-b]t finds cat and cbt

MySQL supports only '%' and ' _' wildcards.

Aggregate Functions

Used to perform summarized calculations or aggregations like sum, average, count, minimum, maximum etc,.
Generally used with Group By.

Sum()

Select **sum**(marks) as Total
From students ;

Avg()

Select **avg**(marks) as average_marks
From students ;

Count()

Select **count**(ID) as 'No of students'
From students ;

Min()

Select **min**(marks) as Least_marks
From students ;

Max()

Select **max**(marks) as Highest_marks
From students ;

Group By

The **Group By** statement groups rows that have the same values, into summary rows.
In general, it gives us a **summarized table**.
Group By is often used with **aggregate functions** to group the result set by one or more columns.

Select school_name from school_details
Group by student_name ;

Select name, **sum**(marks)
From marks
Group by name ;

Select class, section, **count**(ID) as No_of_students
From students
Group by class, section ;

Select name, **sum**(marks) as Total, **avg**(marks) as average,
Min(marks) as least_marks, **Max**(marks) as Highest_marks
From Marks
Group by name ;

Having

Used to filter the summarized data which is created after grouping.
Having is used as a filter with-in the group by statement.
We cannot use ‘WHERE’ filter inside group by statemet.
Hence having is used.

```
Select name, sum(marks) as total
from students
Group by name
Having sum(marks) > 70 ;
```

```
Select name, sum(marks) as total
From students
Where class = ’10th’
Group by name
Having sum(marks) > 70 ;
```

NOTE : HAVING comes only after Group By Statement
And it is used only when Group By is used.

Query Processing Order

The Query Processing Order is the order in which SQL Server executes a given query.

From > Where > Group By > Having > Select > Order By > Limit

From table
Where conditions
Group by columns
Having conditions
Select columns
Order by columns
Limit ;

A Normal SQL Query

```
Select name, sum(marks) as total
From students
Where class = '10th'
Group by name
Having sum(marks) > 70
Order by total desc
Limit 3 ;
```

How the Query actually Executes

```
From students
Where class = '10th'
Group by name
Having sum(marks) > 70
Select name, sum(marks) as total
Order by total desc
Limit 3 ;
```

Write a query for the following

1. Find the customers details who are from USA.
2. Find the customers whose postal code is missing.
3. Find the customers whose postal code and state are missing.
4. Find customers who don’t have any credit limit.
5. Find customers who are from one of the countries USA, France, Norway.
6. Find all the customers whose customernumber is from 100 to 150.
7. Find all the customer details who has the highest credit limit.
8. Find the customers details whose name start with A.
9. Find count of customers whose name end with ‘.’ (dot)
10. Find highest, lowest, average and sum of creditlimit.

JOINS

The term “**JOIN**” in English language refers to combine, attach or to join two or more things together.

A **JOIN clause** is used to combine data from two or more tables, based on a common column between them.

You wanted to learn SQL, and I was ready to teach SQL. You are **students** and I am a **teacher**. We both (**Students table** & **teachers table**) are joined together on a **common interest column** which is SQL.
Hence a join is made between students and teachers based on a common column SQL.

Syntax : **Students JOIN Teachers**

So where will the common interest column “**SQL**” comes into play. We have to give the common interest column as a **condition** to join the two tables. This will be the **JOIN CONDITION**.

Syntax : **Students JOIN Teachers ON Interest_column = Interest_column**

From the above syntax, SQL does not know Which interest_column is coming from which table. So for that we give the table address (i.e, table_name) before the column_name.

Syntax : **Students JOIN Teachers ON Students.Interest_column = Teachers.Interest_column**

Instead of giving the entire table_name before each column, we can assign an alias to the table names for our usage comfort.

Syntax : **Students as S JOIN Teachers as T ON S.Interest_column = T.Interest_column**

So the above line of code will give us a combined table, which is created after joining Students and Teachers tables.

Now, the combined table will act as a new table which has the combined data from Students and Teachers after joining. So to query the new data we use the normal SQL concepts only.

Syntax :

```
Select * from
Students as S JOIN Teachers as T ON S.Interest_column = T.Interest_column ;
```

This will give us all the data after joining Students and Teachers. This is how JOINS work.
We can perform all the operations on the joined data just as we do on a regular table.

Students

S_Name	S_Age	S_Interest
Abhi	15	SQL
Cherry	16	Maths
Chintu	15	Science
Vinod	14	English
Madhu	17	Telugu

Teachers

T_Name	T_Age	T_Interest
Niranjan	30	SQL
Venkat	34	Maths
Ravi	29	Science
Ram	35	English
Krishna	32	Telugu

JOINS

What JOIN does is, it just attaches the two tables together based on the common column.

S_Name	S_Age	S_Interest	T_Name	T_Age	T_Interest
Abhi	15	SQL	Niranjan	30	SQL
Cherry	16	Maths	Venkat	34	Maths
Chintu	15	Science	Ravi	29	Science
Vinod	14	English	Ram	35	English
Madhu	17	Telugu	Krishna	32	Telugu

Joined table

There are different types of joins based on what data we get as output. They are

- **INNER JOIN** (or) **JOIN**
- **LEFT JOIN** (or) **LEFT OUTER JOIN**
- **RIGHT JOIN** (or) **RIGHT OUTER JOIN**
- **FULL JOIN** (or) **FULL OUTER JOIN**
- **CROSS JOIN**
- **SELF JOIN**

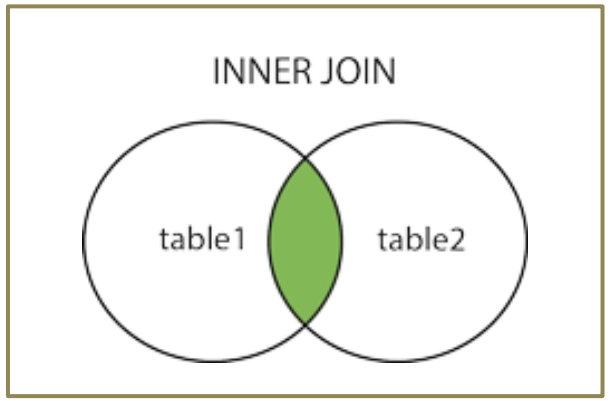
INNER JOIN (or) JOIN

INNER JOIN returns only the matching records from both the tables i.e., the rows which satisfy the joining condition.

We can also represent **INNER JOIN** as just **JOIN**.

Syntax :

```
Select column_names
from table1
INNER JOIN table2
ON table1.column_name = table2.column_name ;
```



When the value for the common column in the first table matches with a value in common column from the second table then, the row corresponding to that matched values from both the tables gets joined.

LEFT JOIN (or) LEFT OUTER JOIN

While using JOINs in SQL, the first table is considered as **LEFT** table and second table is considered as **RIGHT** table.
First table is also referred to as base table.

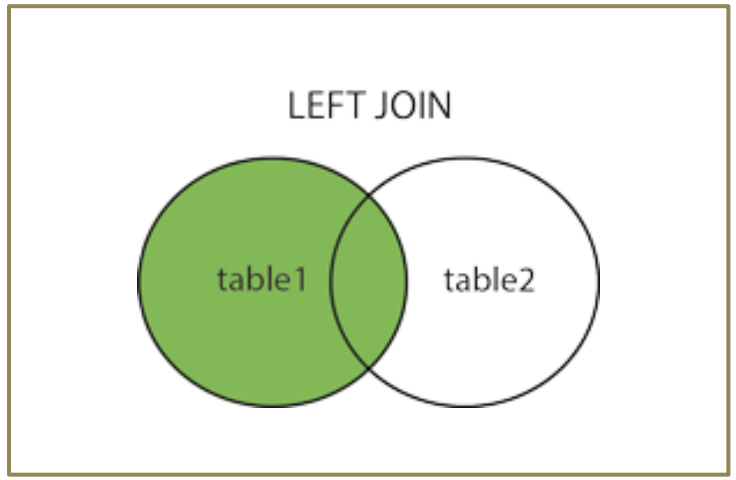
LEFT JOIN returns all the records (or rows) from the left table and only the matching records (records that satisfy the joining condition) from the right table.

- A LEFT JOIN performs a join starting with the left table. All the records from the left table are included first.
- Then, any matching records from the right table will be included.
- Rows without a match in the right table will have NULL column values in the output.
- **LEFT JOIN** is also called as **LEFT OUTER JOIN**.

Syntax :

```
Select column_names
from table1
LEFT JOIN table2
ON table1.column_name = table2.column_name ;

Select column_names
from table1
LEFT OUTER JOIN table2
ON table1.column_name = table2.column_name ;
```



RIGHT JOIN (or) RIGHT OUTER JOIN

Right Join is similar to Left Join but Left Join takes left table as base table and Right Join takes right table as base table.

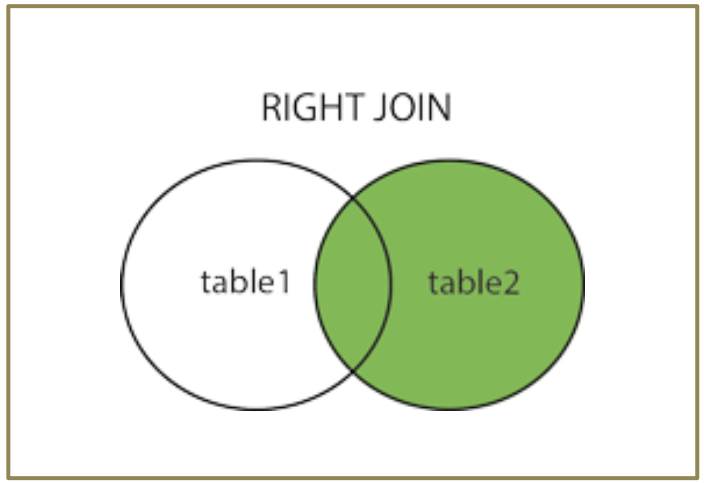
RIGHT JOIN returns all the records (or rows) from the right table and only the matching records (records that satisfy the joining condition) from the left table.

- A Right JOIN performs a join starting with the right table. All the records from the right table are included first.
- Then, any matching records from the left table will be included.
- Rows without a match in the left table will have NULL column values in the output.
- **RIGHT JOIN** is also called as **RIGHT OUTER JOIN**.

Syntax :

```
Select column_names
from table1
RIGHT JOIN table2
ON table1.column_name = table2.column_name ;

Select column_names
from table1
RIGHT OUTER JOIN table2
ON table1.column_name = table2.column_name ;
```



FULL JOIN (or) FULL OUTER JOIN

Full Join combines the functionality of both **LEFT JOIN** and **RIGHT JOIN**.

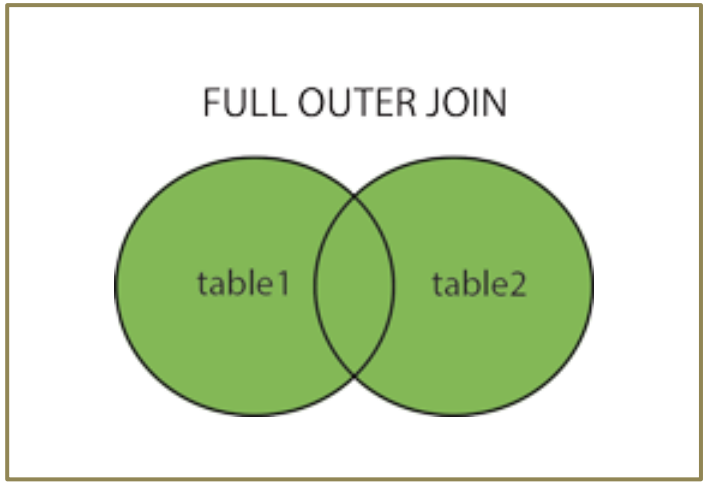
FULL JOIN returns all the records from both left and right tables. If the records from both tables match, then the matching records are joined. If the records from any one of the tables do not match with the records from the other table, then these unmatched records are joined with null values.

- A Full Join performs a join starting from the first table in the query.
- First, all the matching records from the two tables that satisfy the join condition are included.
- Then, all the unmatched records from the first table that do not satisfy the join condition are joined with null values and are included.
- Next, all the unmatched records from the second table that do not satisfy the join condition are joined with nul values and are included.
- Hence FULL JOIN returns all the records from both the left and right tables.

FULL JOIN can potentially return very large result sets.
FULL JOIN and **FULL OUTER JOIN** are same.

Syntax :

```
Select column_names
from table1
FULL JOIN table2
ON table1.column_name = table2.column_name ;
```



❑ **Full Join is not supported in MySQL.**

CROSS JOIN

CROSS JOIN is used to generate a paired combination of each row of the first table with each row of the second table.
This join type is also known as **Cartesian Join**.

Cross Join returns the cartesian product of the records from the two tables.

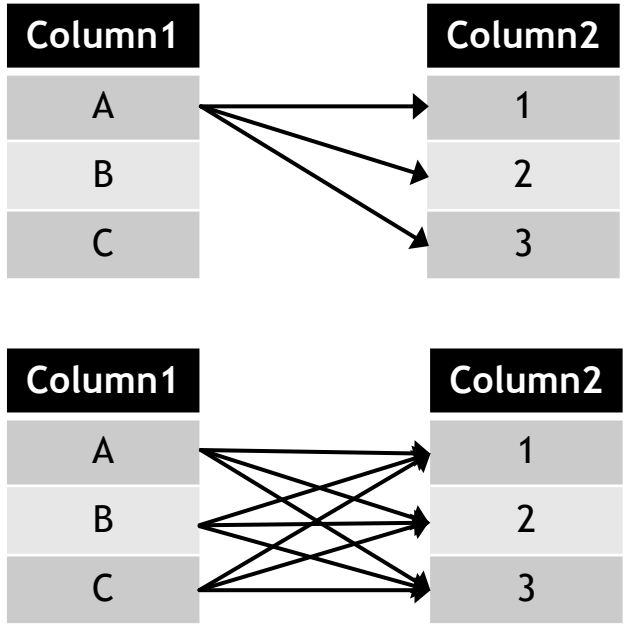
- CROSS JOIN performs a join starting from the first table in the query.
- The first record from the first table is joined with 1st , 2nd, 3rd and so on with all the records from the second table.
- Similarly, the second record from the first table is joined with all the records from the second table.
- This continues for all the records in the first table.
- Hence, all the records from the first table are joined with all the records from the second table.

When **CROSS JOIN** is used for tables that have a high number of rows, it might affect the performance negatively.
If a cross join is performed on two tables having 100 rows each, then row count of the result set after cross joining will be 100*100 = 10000

Syntax :

```
SELECT ColumnNames
FROM Table1
CROSS JOIN Table2 ;
```

```
Select Columns_names
From table1, table2 ;
```



Column1	Column2
A	1
A	2
A	3
B	1
B	2
B	3
C	1
C	2
C	3

After CROSS JOIN

SELF JOIN

A **SELF JOIN** is a regular join, but the table is joined with itself. A single table is joined with itself.
The self join is often used to query hierarchical data or to compare a row with other rows within the same table.

To perform a self join, you must use table aliases to not repeat the same table name twice in a single query. Note that referencing a table twice or more in a query without using table aliases will cause an error.

This can be useful when modeling hierarchies.

Used in situations when you want to find out who is the manager for each employee from the employees. The manager is also in the same list of employees.

Syntax :

```
Select Columns
From table1 as t1
Self Join table1 as t2
ON Join_condition;

Select columns
From table1 as t1,  table1 as t2
Where Condition ;
```

If we need to find out who is the principle for the respective teacher

```
Select t1.t_name as teacher, t2.t_name as Principle
from teachers as t1
self join teachers as t2
on t2.t_ID = t1.Principle_ID ;

Select t1.t_name as teacher, t2.t_name as Principle
from teachers as t1, teachers as t2
where t2.t_ID = t1.Principle_ID ;
```

INNER JOIN, LEFT JOIN, RIGHT JOIN, FULL JOIN can also be used as SELF JOIN. Instead of two tables we use the same table to join with itself.

UNION / UNION ALL

UNION / UNION ALL clause/operator is used to combine the results of two or more SELECT statements.

- Every SELECT statement within UNION must have the same number of columns.
- The columns must also have the same data types.
- The columns in every SELECT statement must also be in the same order.

The only difference between UNION and UNION ALL is that, UNION ALL returns all the records from the two select statements whereas UNION returns only the unique records from the two select statements.

Syntax :

UNION

```
Select Columns from Table1
UNION
Select columns from Table2 ;
```

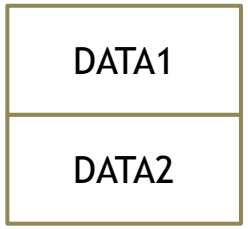
UNION ALL

```
Select Columns from Table1
UNION ALL
Select columns from Table2 ;
```

UNION and JOINS

UNION adds data horizontally i.e., below the data while JOINS adds data vertically i.e., side by side.

UNION



JOINS

SUB QUERY

A **Subquery** is a query within a query.

A Subquery is used to return data that will be used in the main query.
Subqueries can be used with the **SELECT, INSERT, UPDATE, and DELETE** statements along with the operators like **=, , >=, <=, IN, NOT IN, EXISTS** etc.

- You can place the Subquery in a number of SQL clauses: **WHERE** clause, **HAVING** clause, **FROM** clause.
- Subqueries can be used with SELECT, UPDATE, INSERT, DELETE statements along with expression operator. It could be equality operators like IN, NOT IN EXISTS, NOT EXISTS etc or comparison operator such as =, >, =, <=.
- A subquery is a query within another query. The outer query is called as **Main Query** and inner query is called as **Subquery**.
- The subquery generally executes first when the subquery doesn't have any co-relation with the main query, when there is a co-relation the parser takes the decision on the fly on which query to execute on precedence and uses the output of the subquery accordingly.
- Subquery must be enclosed in parentheses ().
- Subqueries are on the right side of the comparison operator.
- Use single-row operators with single-row Subqueries. Use multiple-row operators with multiple-row Subqueries.

❑ **ORDER BY** command cannot be used inside a Subquery.

❑ A subquery cannot contain **LIKE** clause and **BETWEEN** in some cases.

❑ A subquery in an **UPDATE** statement cannot retrieve data from the same table in which data is to be updated.

Syntax :

```
Select columns From
(select columns from table_name) as tbl1 ;
```

```
Select a.* from
(select id, name from students) a ;
```

```
Select columns From table_name
Where column_name Operator
(
select column_name from table1
) ;
```

```
Select * from students
Where id IN
(
Select id from students
where age > 10
) ;
```

```
Select columns From table_name
Where condition
Group by column_name
Having column_name operator
(
Select column_name from table1
) ;
```

```
Select department, count(emp_id) from
departments
Group by department
Having count(emp_id) >
(
select count(emp_id) from departments
Where department = 'Physics'
) ;
```

TEMPORARY TABLES

As its name indicates, **Temporary Tables** are used to store data temporarily.

- Temp tables are almost similar to normal tables in a database.
- We can perform all the operations on temp tables that we do on a normal table like **SELECT, INSERT, UPDATE, DELETE** etc,.
- The only difference is that temporary tables are produced in TempDB and are automatically deleted when the last connection to the query window that created the temp table is terminated.
- We can use Temporary Tables to store and process intermediate results.
- If temp tables are created in a Stored Procedure, they are destroyed in the same Stored Procedure itself.

Syntax :

```
Create TEMPORARY TABLE table_name
(
Column_name1  Datatype,
Column_name2  Datatype
) ;
```

After creating a Temporary Table, we can use it just like a normal table to insert data, copy data from other table and perform all the operations as a normal table.

These Temporary Tables, once created, will stay as long as the session is alive.
Temporary Tables are destroyed or deleted automatically when you terminate or close the current session.

If you want to delete a Temporary Table even before terminating the session, you can use the **DROP** command.

Syntax : **Drop table temp_table_name ;**

MS SQL Server has 2 types of Temporary tables

1. Local Temporary Tables
    - These types of Temp tables are only accessible to the session that produced them.
    - Local Temp Tables are destroyed automatically when the current procedure or session that has created the temp tables end.
    - They are denoted by the prefix # .

    Syntax :

```
Create Table #table_name
(
Column_name1  Datatype,
Column_name2  Datatype
) ;
```

2. Global Temporary Tables
    - These are accessible to all sessions and users simultaneously.
    - They are automatically deleted when the last session that used the temporary table has ended.
    - They are denoted by the prefix ## .

    Syntax :

```
Create Table ##table_name
(
Column_name1  Datatype,
Column_name2  Datatype
) ;
```

CASE STATEMENTS

CASE Statements are used to **tag** or **flag** the records based on a set of given conditions. It works as an **IF-ELSE** loop.
Can be used in statements like SELECT, UPDATE, SET.

Syntax :

```
CASE
    When Condition1 then Result1
    When Condition2 then Result2
    …
    When ConditionN then ResultN
    ELSE Result
END ;
```

- The CASE Statement goes through all the conditions in the order mentioned.
- When a condition is met then, returns the result corresponding to that condition and stops the loop.
- If no condition is met then, it returns the result given in the ELSE clause.
- If there is no ELSE clause and no conditions are satisfied then CASE returns a NULL for that record.
- It adds a new column with the flag created using the conditions.
- We can give an alias to the flag column that will be created when case statement is used.

Syntax :

```
Select *,
        CASE
                When Condition1 then Result1
                When Condition2 then Result2
                …
                When ConditionN then ResultN
                ELSE Result
        END as flag_column
From  Table_name
Where  Condition
Group by column_name
Having  Condition
Order BY column_name
Limit  N
;
```

Example :

```
Select * from
(
Select *,
CASE
        When creditlimit < 100000 then ‘Small Range Customers’
        When creditlimit between 100000 to 200000 then ‘Mid
        Range Customers’
        Else ‘Premium Customers’
END as Type_of_Customer
From Customers
) as A
Where Type_of_Customer = “Premium Customers”
;
```

CLONING TABLES

You may encounter a situation where you need to create a copy of an entire table with all the data in it. In these situations we can use cloning table statements or copying tables.

Syntax :

```
Create table new_table_name as
Select * from old_table_name ;

Create table new_table_name LIKE
Old_table_name

Insert into New_table_name
Select * from Old_table_name ;

Select * Into new_table_name
From old_table _name ;
```

```
Create table new_table_name as
Select Column1, column2
from old_table_name
Where condition ;

Select Column1, column2
Into new_table_name
From old_table _name
Where condition ;

Insert into New_table_name (column1, column2)
Select Column1, column2
from Old_table_name
Where condition ;
```

We can copy only the required columns by specifying the required column names in the copy statement.

We can also copy data based on a specific condition also.

WINDOW FUNCTIONS

Window Functions are used to perform calculations and aggregations on partitions or divisions of data in a table and assign that corresponding value to every record in the table.

- It Performs aggregations similar to Group By clause.
- Group By clause groups the data and returns a single record but window functions groups the data and returns all the records with an added column of aggregation performed.
- When using Group By, we need to specify all the columns except the aggregating column in the Group By clause whereas in window functions we don’t need to specify them.
- Window Functions add an extra column to every record with the corresponding result value.
- They are used in situations where we need the aggregate values by grouping divisions of data along with all the data in the table.

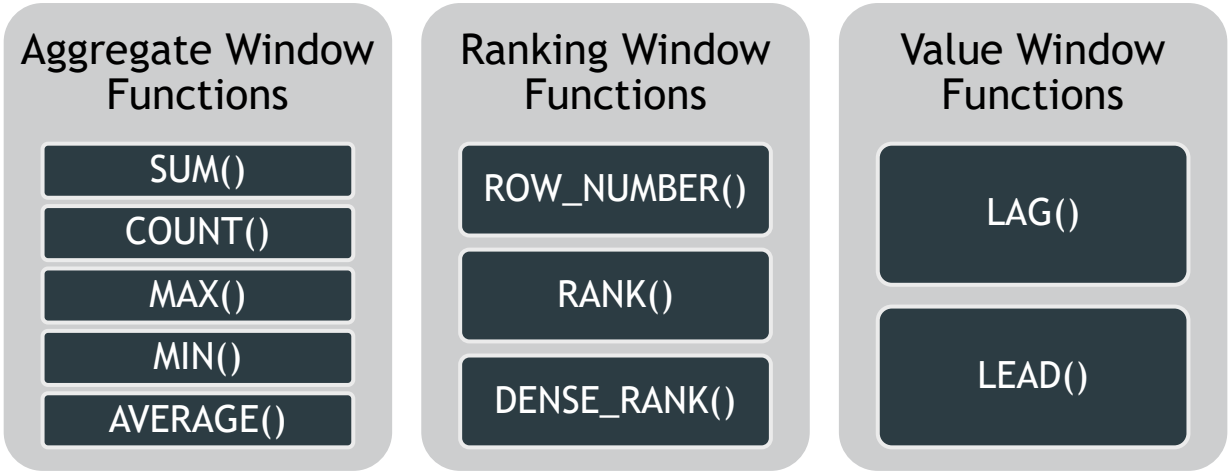
Syntax :

```
WINDOW_FUNCTION(Column_name) OVER (PARTITION BY Column_name ORDER BY Column_name)
```

Example :

```
Select *,
Window_function(column1) over (Partition By Column2 Order By Column2) as column_name
From table_name ;
```

There are 3 types of Window Functions. They are



1. AGGREGATE WINDOW FUNCTIONS

Aggregate Window Functions are used to perform aggregations like sum, count, average, max and min.

Syntax :

```
SUM(Column1) OVER (PARTITION BY Column_name ORDER BY Column_name)
```

For all the other aggregate functions replace sum with the required function.

2. RANKING WINDOW FUNCTIONS

1. ROW_NUMBER()

ROW_NUMBER() Window Function is used to assign a number from 1 to all the rows returned in the result-set in the order specified in the ORDER BY clause.

Syntax :

```
Row_Number() over (Partition By Column1 Order By Column2)
```

2. RANK()

RANK() Window Function is used to assign a rank to all the records in the result-set in the order specified in the ORDER BY clause.

The difference b/w Rank() and Row_Number() is that, Row_Number assigns a number to all the records consecutively even if there is a tie b/w values ie., if there are multiple same values.

Rank() assigns the same rank to the same values and skips that many no of ranks, the number of records that have matching value and assigns the next rank to the next different value.

Syntax :

```
Rank() Over (Partition By Column1 Order By Column2)
```

3. DENSE_RANK()

Dense_Rank() is identical to Rank().

If there are matching values, Dense_Rank() assigns the same rank to the matching records and will assign the consecutive rank to the next records without skipping the ranks for matching values.

Syntax :

```
Dense_Rank() Over (Partition By Column1 Order By Column2)
```

The only difference b/w **Row_Number(), Rank()** and **Dense_Rank()** is how they handle records with matching values.

3. VALUE WINDOW FUNCTIONS

1. LAG()

LAG() Window Function is used to assign the lag value or previous record value to the current row beside it in a new column.

It is used in situations where we need to compare previous record values with the current record values.

Syntax :

```
Lag(Column_name, N, Default_value) Over (Partition By Column1 Order By Column2)
```

- Column_name is the column for which we need the lag value.
- N is the number of preceding rows to the current row. If not given any value, it takes the default vaue 1.
- Dafault_Value is the value that will be assigned if no preceeding value is found for the current record.

2. LEAD()

LEAD() Window Function is used to assign the leading value or the next record value to the current row beside it in a new column.

It is used in situations where we need to compare the current record value with the leading records values.

Syntax :

```
Lead(Column_name, N, Default_value) Over (Partition By Column1 Order By column2)
```

- Column_name is the column for which we need the lead value.
- N is the number of succeeding rows to the current row. If not given any value, it takes the default vaue 1.
- Dafault_Value is the value that will be assigned if no succeeding value is found for the current record.

VIEWS

VIEWS are virtual tables, which depends on the result-set of a predefined SQL statement.

- They don't exist in the database for real. So they don't require any storage in the database.
- Views can be created by selecting columns from one or more tables.
- Views can be used for data security. We can let the users access the data through a view, without granting the users permissions to directly access the data.

Syntax to create a view

```
Create VIEW view_name as
Select * from table_name
Where condition ;
```

We an also update a view which is already created. But there are some limitations to it.

- We can only update a view which is created from one table. We cannot update a view which is created from **more than one tables.**
- The records should not contain **NULL** values.
- We cannot update a view if the view is created by using **JOINs, GROUP BY, HAVING** clause.
- We cannot update the view if the view is created by using **DISTINCT** keyword.
- We cannot update a view which is created by using a **SUBQUERY**.

Syntax to update a view

```
Create (or) Replace VIEW view_name as
Select * from table_name
Where condition ;
```

We can delete or drop a VIEW once we don't need it any further.

Syntax :

```
DROP VIEW view_name ;
```

INDEX

Indexes are used to retrieve data from the database more quickly than normal. We can create indexes on columns which we query more often. **Querying columns with indexes takes less time.**

- An Index is a schema object.
- The users cannot see the indexes, they are just used to speed up searches/queries.
- An index helps to speed up select queries, where clauses and join statements.
- It slows down data input, with the update and the insert statements.
- We can create or drop indexes without effecting the data in the table.
- A table with indexes takes more time to update than updating a table without indexes because the indexes also need an update.
- So, only create indexes on columns that will be frequently searched against.

Syntax :

Create INDEX index_name on table_name(column1, column2) ;

Creates an index on the columns column1 and column2.

UNIQUE INDEX

Unique Index does not allow duplicate records.

Syntax :

Create UNIQUE INDEX index_name on table_name (column1, column2) ;

We can alter an Index which is already created in a table.

Syntax :

Alter table table_name
ADD INDEX index_name (columns) ;

We can drop an index once we don't need it further.

Syntax :

Alter table table_name
DROP INDEX index_name ;

DROP INDEX index_name ;

Works in MS SQL. Does not work in MySQL

STORED PROCEDURES

A Stored Procedure is a prepared and pre-written SQL code saved in the database under a name, that can be used again and again when necessary.

So if we have an SQL query that we write over and over multiple times, we can store or save the code in a stored procedure and call the procedure to execute the code.

- Stored Procedures increase the performance.
- They reduce traffic b/w application and database server.
- Stored Procedures are reusable.

Syntax :

```
Delimiter @@
Create PROCEDURE proc_name()
BEGIN
    SQL_statement ;
END @@
Delimiter ;
```

To Execute the Stored Procedure

Syntax :

```
Call proc_name ;
```

MS SQL SERVER

```
Create Procedure proc_name
As
SQL_Statement
Go ;
```

```
Exec proc_name ;
```

We can also pass parameters to Stored Procedures and use the parameter value to execute the Procedure dynamically.

Syntax :

```
Delimiter @@
Create PROCEDURE proc_name(Var_country varchar(50))
BEGIN
    select * from customers
    where country = Var_country ;
END @@
Delimiter ;
```

Now we can execute the procedure by passing the country variable.

Syntax :

```
Call proc_name("USA") ;
```

This will give us all the data where country = "USA"

To delete or drop a procedure

Syntax :

```
Drop Procedure proc_name ;
```

MS SQL SERVER

```
Create procedure proc_name @country varchar(50)
AS
Select * from Customers
Where country = @ country
GO ;
```

To execute the procedure

```
Exec proc_name @country = "France" ;
```

To delete or drop a procedure

Syntax :

```
Drop Procedure proc_name ;
```

CTE – COMMON TABLE EXPRESSIONS

Common Table Expressions are a temporary named result set that you can reference within a Select, Insert, Update or Delete statement

- CTEs work as virtual tables.
- They are created during the execution of a query, used by the query, and eliminated after query execution.
- We can define a CTE by adding a WITH clause.
- A WITH caluse can include one or more CTEs separated by commas.
- CTEs are similar to subqueries.

Syntax :

```
WITH cte_name1 as
( SQL_Query_1 ),
Cte_name2 as
( SQL_Query_2 )
Select * from cte_name1, cte_name2
Where condition ;
```

Example :

```
With cte1 as
( select country, avg(creditlimit) as avg_credit
From customers group by country )
Select country, Max(avg_credit) as max_avg_credit
From customers group by country ;
```

SQL code that is with the WITH clause acts as the inner query in a sub query and the last query acts as the main query.

FUNCTIONS

SQL Functions are a set of SQL statements stored or saved under a name, that perform a specific task. They are similar to stored procedures.

- Functions help reusability.
- Functions take input parameters, perform actions and returns results.
- We cant use a function to insert, update or delete records.

There are a number of different SQL functions available in different SQL versions.

All these functions are generally divided into 4 types.

1. **String Functions**
2. **Numeric Functions**
3. **Date and Time Functions**
4. **Advanced Functions**

STRING FUNCTIONS

STRING FUNCTIONS are the functions that are used with **STRING** or **TEXT** datatype columns like char, varchar, nvarchar etc,.

LOWER / LCASE

Converts a given string to Lower Case.

Syntax :

```
LOWER( Text )
LCASE( Text )
```

UPPER / UCASE

Converts a given string to Upper Case.

Syntax :

```
UPPER( Text )
UCASE( Text )
```

LENGTH / CHAR_LENGTH / CHARACTER_LENGTH

Returns length of the given string.

Syntax :

```
LENGTH( Text )
CHAR_LENGTH( Text )
CHARACTER_LENGTH( Text )
```

CONCAT

ADDS two or more strings together.

Syntax :

```
CONCAT(String1, String2,….)
```

CONCAT_WS

Adds two or more strings together with a specified **Separator** b/w the strings.

Syntax :

```
CONCAT_WS(Separator, String1, String2, …)
```

LEFT

Extracts specified number of characters from a given string starting from the left side.

Syntax :

```
LEFT(String, No_of_characters)
```

RIGHT

Extracts specified number of characters from a given string starting from the right side.

Syntax :

```
RIGHT(String, No_of_characters)
```

TRIM

Removes any spaces present before or after the specified string.

Syntax :

```
TRIM( String )
```

LTRIM

Removes the leading spaces from a specified string.

Syntax :

LTRIM(String)

RTRIM

Removes the trailing spaces present after a specified string.

Syntax :

RTRIM(String)

FORMAT

Formats a given number to a format like '#,###,###.##' rounded to a specified number of decimal places.

Syntax :

FORMAT(Number, Decimal_places)

REPEAT

Repeats a given string, in the specified number of times.

Syntax :

REPEAT(String, number)

REVERSE

Reverses a given string.

Syntax :

REVERSE(String)

LOCATE

Returns the position of first occurrence of a substring in a string.

Syntax :

LOCATE(Substring, String, Start)

Start is the starting position for the search. Default is 1.

Returns 0, if the substring is not found.

POSITION

Identical to **LOCATE** function.

Syntax :

POSITION(Substring IN String)

INSTR

Identical to **LOCATE** function.

Syntax :

INSTR(String, Substring)

REPLACE

Replaces all the occurrence of a substring in a string with a new substring.

Syntax :

REPLACE (String, Old_Substring, New_Substring)

SUBSTRING

Extracts a substring from a string starting from the specified position and extracts specified number of characters.

Syntax :

SUBSTRING(String, Start, Length)

Start is the position from where we need to extract.

Length is the number of characters we need to extract starting from the start position.

SUBSTR

Identical to **SUBSTRING** function.

Syntax :

SUBSTR (String, Start, Length)

MID

Identical to **SUBSTRING** function.

Syntax :

MID (String, Start, Length)

NUMERIC FUNCTIONS

Numeric Functions are the functions that are used with numerical data type columns like int, float, decimal etc,.

SUM

Returns sum of all the given column values.

Syntax :

SUM(Expression)

COUNT

Returns count of all the given column values.

Syntax :

COUNT(Expression)

AVG

Returns average of all the given column values.

Syntax :

AVG(Expression)

MAX

Returns maximum value of all the given column values.

Syntax :

MAX(Expression)

MIN

Returns minimum value of all the given column values.

Syntax :

MIN(Expression)

Round

Rounds a given number to specified number of decimal places.

Syntax :

ROUND(Number, No_of_decimals)

CEIL

Returns smallest integer value that is bigger than or equal to given number. CEIL() generally rounds up a given number to 0 decimals.

Syntax :

CEIL(Number)

CEILING

Identical to CEIL function.

Syntax :

CEILING(Number)

FLOOR

Returns largest integer value that is smaller than or equal to given number. FLOOR() generally rounds down a given number to 0 decimals.

Syntax :

FLOOR(Number)

TRUNCATE

Truncates a given number to a specified number of decimal places.

Syntax :

TRUNCATE(Number, No_of_decimals)

ABS

Returns the Absolute (Positive) value of a given number.

Syntax :

ABS(Number)

MOD

Returns the remainder of a number divided by another number.

Syntax :

MOD(x, y)

X is divided by y and the remainder is returned.

RAND

Returns a random decimal number b/w 0 and 1.

Syntax :

RAND()

SQRT

Returns the square root of a given number.

Syntax :

SQRT(Number)

POWER

Returns the value of a number raised to the power of another number.

Syntax :

POWER(x, y)

Returns x^y

POW

Returns the value of a number raised to the power of another number.

Syntax :

POW(x, y)

Returns x^y

Identical to POWER function.

GREATEST

Returns the greatest value from a list of given numbers.

Syntax :

GREATEST(a, b, c, ….)

LEAST

Returns the smallest value from a list of given numbers.

Syntax :

LEAST(a, b, c, ….)

DATE and TIME FUNCTIONS

CURRENT_DATE / CURDATE

Returns today's date in 'YYYY-MM-DD' format.

Syntax :

```
CURRENT_DATE()
CURDATE()
```

CURRENT_TIME / CURTIME

Returns current time in 'HH-MM-SS' format.

Syntax :

```
CURRENT_TIME()
CURTIME()
```

CURRENT_TIMESTAMP / NOW / SYSDATE

Returns today's date and time in 'YYYY-MM-DD HH-MM-SS' format.

Syntax :

```
CURRENT_TIMESTAMP ()
NOW()
SYSDATE()
```

We can also use **GETDATE()** function to get the current date and time. But it is not supported in MySQL.

DATE

Extracts the DATE part from a given date-time expression.

Syntax :

```
DATE( Expression )
```

TIME

Extracts the TIME part from a given date-time expression.

Syntax :

```
TIME( Expression )
```

YEAR

Extracts the YEAR from a given date or date-time expression.

Syntax :

```
YEAR( Expression )
```

MONTH

Extracts the MONTH from a given date or date-time expression.

Syntax :

```
MONTH( Expression )
```

DAY / DAYOFMONTH

Extracts the DAY of the month from a given date or date-time expression.

Syntax :

```
DAY( Expression )
DAYOFMONTH( Expression )
```

HOUR

Extracts the HOUR part from a given time or date-time expression.

Syntax :

```
HOUR( Expression )
```

MINUTE

Extracts the MINUTE part from a given time or date-time expression.

Syntax :

MINUTE(Expression)

SECOND

Extracts the SECOND part from a given time or date-time expression.

Syntax :

SECOND(Expression)

WEEK

Returns WEEK number(0-53) for a given date or date-time expression.

Syntax :

WEEK(Expression)

WEEKOFYEAR

Returns WEEK number number(0-53) for a given date or date-time expression. This function considers Monday as the first day of week.

Syntax :

WEEKDAY(Expression)

WEEKDAY

Returns WEEKDAY number(0-6 starting Monday) ie. day number in a week for a given date or date-time expression.

Syntax :

WEEKDAY(Expression)

DAYOFWEEK

Returns day number(1-7 starting Sunday) in a week for a given date or date-time expression.

Syntax :

DAYOFWEEK(Expression)

DAYOFYEAR

Returns day number(1-366) in a year for a given date or date-time expression.

Syntax :

DAYOFYEAR(Expression)

QUARTER

Returns the QUARTER (1-4) of the year for a given date or date-time expression.

Syntax :

QUARTER(Expression)

MONTHNAME

Returns the name of the month for a given date or date-time expression.

Syntax :

MONTHNAME(Expression)

DAYNAME

Returns the name of the day for a given date or date-time expression.

Syntax :

DAYNAME(Expression)

LAST_DAY

Returns the date of last day of the month for a given date or date-time expression.

Syntax :

LAST_DAY(Expression)

EXTRACT

Extracts the required part from a given date or date-time expression..

Syntax :

EXTRACT(Required_Part FROM Expression)

Expression is date or date-time.

Required_Part can be Year, Quarter, Month, Week, Day, Hour,

Minute, Second, Microsecond, Year_month etc,.

STR_TO_DATE

Returns a date based on a given string and specified format.

Syntax :

STR_TO_DATE(String, Format)

The format should be in the format of the given string for the function to convert the string to date.

DATE_FORMAT

Formats a given date or date-time expression to a specified format.

Syntax :

DATE_FORMAT(Expression, Format)

Format can be created by using combination of the following

%M Month name in full (January to December)
%b Abbreviated month name (Jan to Dec)
%m Month name as a numeric value (00 to 12)
%D Day of the month as a numeric value, followed by suffix(1st,2nd)
%d Day of the month as a numeric value (01 to 31)
%H Hour (00 to 23)
%h Hour (00 to 12)
%i Minutes (00 to 59)
%S Seconds (00 to 59)
%W Weekday name in full (Sunday to Saturday)
%Y Year as a numeric, 4-digit value
%y Year as a numeric, 2-digit value

TIME_FORMAT

Formats a given time or date-time expression to a specified format.

Syntax :

TIME_FORMAT(Expression, Format)

Format can be created in the same way as mentioned in Date_Format

DATE_ADD / ADDDATE

Adds a specified time or date interval to a date or date-time expression and returns the date

Syntax :

DATE_ADD(Date, INTERVAL Value AddUnit)

ADDDATE(Date, INTERVAL Value AddUnit)

ADDDATE(Date, days_to_add)

AddUnit is the Day, month, year, hour, minute, second, week etc.

Value is the number to add to the date

Example : **DATE_ADD('2022-12-01', INTERVAL 10 DAY)**

Output : 2022-12-11

DATE_SUB / SUBDATE

Subtracts a specified time or date interval from a date or date-time expression and returns the date

Syntax :

DATE_SUB(Date, INTERVAL Value SubUnit)

SUBDATE(Date, INTERVAL Value SubUnit)

SUBDATE(Date, days_to_subtract)

Similar to Date_Add / AddDate, but Date_Sub subtracts and

Date_Add adds interval to given date.

ADDTIME

Adds time interval to time or date-time expression and returns time / date-time.

Syntax :

ADDTIME(Expression, AddTime)

AddTime can be specified in seconds or in time format also.

ADDTIME("2022-12-12 10:30:25", 300) adds 300 seconds i.e, 3

,minutes to the timestamp.

ADDTIME("2022-10-11 05:20:10", "2:30:20") adds 2 hours 30

minutes 20 seconds to the given date-time.

SUBTIME

Subtracts specified time interval from time or date-time expression and returns time / date-time.

Syntax :

SUBTIME(Expression, AddTime)

Similar to ADDTIME function.

DATEDIFF

Returns number of days between two dates or date-time expressions.

Syntax :

DATEDIFF(Date1, Date2)

TIMEDIFF

Returns the difference between two times or date-time expressions,
in the format of HH:MM:SS.

Syntax :

TIMEDIFF(Time1, Time2)

FROM_DAYS

Returns a date from a given numeric date value.

Syntax :

FROM_DAYS(Number)

Number is the number of days from the start date 0000-00-00 .

TO_DAYS

Returns number of days in between the given date or date-time expression and the date 0000-00-00.

Syntax :

TO_DAYS(Date)

ADVANCED FUNCTIONS

CAST
Converts a value of any datatype to the specified datatype.

Syntax :

CAST(Value AS Datatype)

CAST is used to modify the datatype of a column while querying the data.

CONVERT
Converts a value into the specified datatype.

Syntax :

CONVERT(Value, Datatype)

CONVERT is also used to modify the datatype of a column while querying the data.

COALESCE
Returns the first non-null value from a given set.

Syntax :

COALESCE(Value1, Value2, Value3, …. ValueN)

ISNULL
Returns 1 if a given expression IS NULL and 0 if it is NOT NULL

Syntax :

ISNULL(Expression)

IFNULL
Returns a specified value if the expression IS NULL.
If the expression IS NOT NULL then returns the expression.

Syntax :

IFNULL(Expression, Alternate_Value)

NULLIF
Compares two given expressions and returns NULL if they are equal.
If they are not equal, then first expression is returned.

Syntax :

NULLIF(Expression1, Expression2)

IF
Returns a specified value if a given condition is TRUE, or another specified value if the given condition is FALSE.

Syntax :

IF(Condition, Value_if_TRUE, Value_if_FALSE)

VERSION
Returns the current version of MySQL database.

Syntax :

VERSION()

USER / SESSION_USER / SYSTEM_USER
Returns the current user name and host name for MySQL connection.

Syntax :

USER()
SESSION_USER()
SYSTEM_USER()

1. Find customers who have placed at least 1 order.
2. Find customers who have not ordered anything.
3. Find the no. of orders from each country.
4. Find the details of top 5 customers who have placed more no of orders.
5. Find out which employee is responsible for the most no of orders.
6. Find out which customer has placed the most valuable order.
7. Find the top 5 most valuable orders and the customer details who placed the order.

1. Rank the performance of employees based on the number of orders.
2. Calculate the orders distribution by month and by year.
3. Find the no. of days taken to ship each order. And find the average shipping days
4. Find the customer details who have placed an order and then cancelled it.
5. Calculate Orders distribution by product category.