What is SQL?

- SQL stands for Structured Query Language
- SQL lets you access and manipulate databases
- SQL keywords are NOT case sensitive

RDBMS

- RDBMS stands for Relational Database Management System.
- RDBMS is the basis for SQL, and for all modern database systems such as MS SQL Server, IBM DB2, Oracle, MySQL, and Microsoft Access.
- The data in RDBMS is stored in database objects called tables. A table is a collection of related data entries and it consists of columns and rows.

Some of The Most Important SQL Commands

- SELECT - extracts data from a database
- UPDATE - updates data in a database
- DELETE - deletes data from a database
- INSERT INTO - inserts new data into a database
- CREATE DATABASE - creates a new database
- ALTER DATABASE - modifies a database
- CREATE TABLE - creates a new table
- ALTER TABLE - modifies a table
- DROP TABLE - deletes a table
- CREATE INDEX - creates an index (search key)
- DROP INDEX - deletes an index

SELECT Syntax

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

- Here, column1, column2, ... are the field names of the table you want to select data from. If you want to select all the fields available in the table, use the following syntax:

```
SELECT * FROM table_name;
```

SELECT DISTINCT Syntax

- Distinct also count null values but only one if present.

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

```
SELECT DISTINCT * FROM table_name;
```

WHERE Syntax

- It is used to extract only those records that fulfill a specified condition.

```
SELECT column1, column2, ...
FROM table_name
WHERE condition;
```

AND Syntax

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

OR Syntax

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

NOT Syntax

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

ORDER BY Syntax

- The ORDER BY keyword sorts the records in ascending order by default. To sort the records in descending order, use the DESC keyword. To sort each column manually write ASC|DESC after each columns.

```
SELECT column1, column2, ...
FROM table_name
ORDER BY column1, column2, ... ASC|DESC;
```

INSERT INTO Syntax

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

- Specify both the column names and the values to be inserted:

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

- If you are adding values for all the columns of the table, you do not need to specify the column names in the SQL query. However, make sure the order of the values is in the same order as the columns in the table

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

What is a NULL Value?

A field with a NULL value is a field with no value.
If a field in a table is optional, it is possible to insert a new record or update a record without adding a value to this field. Then, the field will be saved with a NULL value.

- Use the IS NULL and IS NOT NULL operators to check the NULL values.

IS NULL Syntax

```
SELECT column_names
FROM table_name
WHERE column_name IS NULL;
```

IS NOT NULL Syntax

```
SELECT column_names
FROM table_name
WHERE column_name IS NOT NULL;
```

UPDATE Syntax

- The UPDATE statement is used to modify the existing records in a table.

```
UPDATE table_name
SET column1 = value1, column2 = value2, ...
WHERE condition;
```

DELETE Syntax

- The DELETE statement is used to delete existing records in a table. If no condition is given then all the records get deleted.

```
DELETE FROM table_name WHERE condition;
```

SELECT TOP Syntax

- It is used to specify the number of records to return.MySQL supports the LIMIT clause to select a limited number of records

```
SELECT column_name(s)
FROM table_name
WHERE condition
LIMIT number;
```

- The LIMIT clause accepts an optional second parameter.When two parameters are specified, the first parameter specifies the offset of the first row to return i.e. the starting point, whereas the second parameter specifies the maximum number of rows to return. The offset of the initial row is 0 (not 1).

```
SELECT column_name(s)
FROM table_name
WHERE condition
LIMIT number1, number2;
```

- Aggregate functions are functions that perform calculations on a set of values and return a single value as a result. These functions are commonly used to summarize or manipulate data in various ways, such as computing sums, averages, counts, maximum values, and minimum values within a group of rows.

MIN()|MAX() Syntax

```
SELECT MIN|MAX(column_name)
FROM table_name
WHERE condition;
```

COUNT() Syntax

- count(*) gives total no of rows, includes non-null, null as well as duplicates.
- count(column_name) gives total no of Non-NULL values includes duplicates.
- distinct column_name gives all the distinct values in the column even null if present.

```
SELECT COUNT(column_name)
FROM table_name
WHERE condition;
```

AVG() Syntax

- It ignores the null values thus it becomes equal to sum(column_name) / count(column_name);

```
SELECT AVG(column_name)
FROM table_name
WHERE condition;
```

SUM() Syntax

It ignores the null values or say consider it as zero.

```
SELECT SUM(column_name)
FROM table_name
WHERE condition;
```

LIKE Syntax

- search for a specified pattern in a column.

There are two wildcards often used in conjunction with the LIKE operator. A wildcard character is used to substitute one or more characters in a string.:

- The percent sign (%) represents zero, one, or multiple characters
- The underscore sign (_) represents one, single character

```
SELECT column1, column2, ...
FROM table_name
WHERE columnN LIKE pattern;
```

- 'a%' - Finds any values that start with "a"
- '%a' - Finds any values that end with "a"
- '%or%' - Finds any values that have "or" in any position
- '_r%' - Finds any values that have "r" in the second position
- 'a_%' - Finds any values that start with "a" and are at least 2 characters in length
- 'a__%' - Finds any values that start with "a" and are at least 3 characters in length
- 'a%o' - Finds any values that start with "a" and ends with "o"
- % Represents zero or more characters
- _ Represents a single character
- [] Represents any single character within the brackets
- ^ Represents any character not in the brackets
- The keyword - Represents any single character within the specified range

IN Syntax

- allows us to specify multiple values in a WHERE clause.
- a shorthand for multiple OR conditions.

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

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

- If want to exclude use NOT IN instead of IN.

BETWEEN Syntax

- Selects values within a given range. The values can be numbers, text, or dates.
- It is inclusive: begin and end values are included.
- Use NOT BETWEEN if want to select out of this range.

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

Alias

- used to give a table, or a column in a table, a temporary name.
- used to make column names more readable.
- only exists for the duration of that query.
- created with the AS keyword.

Alias Column Syntax

```
SELECT column_name AS alias_name
FROM table_name;
```

Alias Table Syntax

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

JOIN

It combine rows from two or more tables, based on a related column between them.

- (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
- CROSS JOIN is used to combine all possibilities of the two or more tables and returns the result that contains every row from all contributing tables. The CROSS JOIN is also known as CARTESIAN JOIN, which provides the Cartesian product of all associated tables.

UNION

The UNION operator is used to combine the result-set of two or more SELECT statements.

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

```
SELECT column_name(s) FROM table1
UNION
SELECT column_name(s) FROM table2;
```

UNION ALL

The UNION operator selects only distinct values by default. To allow duplicate values, use UNION ALL

```
SELECT column_name(s) FROM table1
UNION ALL
SELECT column_name(s) FROM table2;
```

GROUP BY

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

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

HAVING

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

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

EXISTS

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

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

The ANY operator:

returns a boolean value as a result
returns TRUE if ANY of the subquery values meet the condition
ANY means that the condition will be true if the operation is true for any of the values in the range.

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

The ALL operator:

returns a boolean value as a result
returns TRUE if ALL of the subquery values meet the condition

is used with SELECT, WHERE and HAVING statements
ALL means that the condition will be true only if the operation is true for all values in the range.

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

CASE

- The CASE expression goes through conditions and returns a value when the first condition is met (like an if-then-else statement). So, once a condition is true, it will stop reading and return the result. If no conditions are true, it returns the value in the ELSE clause.
- If there is no ELSE part and no conditions are true, it returns NULL.

```
CASE
    WHEN condition1 THEN result1
    WHEN condition2 THEN result2
    WHEN conditionN THEN resultN
    ELSE result
END;
```

Example :

```
SELECT OrderID, Quantity,
CASE
    WHEN Quantity > 30 THEN 'The quantity is greater than 30'
    WHEN Quantity = 30 THEN 'The quantity is 30'
    ELSE 'The quantity is under 30'
END AS QuantityText
FROM OrderDetails;
```

NULL:

NULL is a special marker that indicates the absence of a value or the lack of data in a particular column of a row. It's not the same as an empty string ('') or zero (0). Here are some key points to understand about NULL in MySQL:

Meaning: NULL represents missing or unknown data. It's used to indicate that a value is not available, not applicable, or has not been provided.
Handling: Arithmetic operations involving NULL usually result in a NULL value. For example, NULL + 5 or NULL * 10 both result in NULL. Comparisons involving NULL generally return NULL as well.

Aggregates: Most aggregate functions ignore NULL values when performing calculations. For instance, if you use the SUM aggregate function, it won't include NULL values in its calculation.
Logical Operators: Comparisons with NULL involve special handling. The result of a comparison involving NULL and a value (except for IS NULL and IS NOT NULL checks) is generally NULL. Use the IS NULL and IS NOT NULL conditions to explicitly check for NULL values.
Operations with NULL: To handle NULL values in expressions, you can use the IS NULL and IS NOT NULL conditions, as well as the COALESCE function or the NULLIF function to handle possible NULL values in expressions.

CREATE DATABASE

- Used to create new SQL database.

```
CREATE DATABASE databasename;
```

DROP DATABASE

- Drop / Delete an existing SQL database.

```
DROP DATABASE databasename;
```

BACKUP DATABASE

- create a full back up of an existing SQL database.

```
BACKUP DATABASE databasename
TO DISK = 'filepath';
```

BACKUP WITH DIFFERENTIAL

- A differential back up only backs up the parts of the database that have changed since the last full database backup.

```
BACKUP DATABASE databasename
TO DISK = 'filepath'
WITH DIFFERENTIAL;
```

CREATE TABLE

```
CREATE TABLE table_name (
    column1 datatype,
    column2 datatype,
    column3 datatype,
   ....
);
```

or

```
CREATE TABLE new_table_name AS
    SELECT column1, column2,...
    FROM existing_table_name
    WHERE ....;
```

DROP TABLE

- Delete the table along with schema.

```
DROP TABLE table_name;
```

TRUNCATE TABLE

- Delete the data inside a table, but not the table itself.
- It is different from delete function as in case of delete each row is deleted one by one while in case of truncate it creates a new table with schema only and then deletes the previous table completely.

```
TRUNCATE TABLE table_name;
```

ALTER TABLE - ADD Column

- Add column in the table.

```
ALTER TABLE table_name
ADD column_name datatype;
```

ALTER TABLE - DROP COLUMN

- Delete the column from the table.

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

ALTER TABLE - MODIFY COLUMN

- change the data type of a column in a table.

```
ALTER TABLE table_name
MODIFY COLUMN column_name datatype;
```

SQL Constraints:

SQL constraints are used to specify rules for the data in a table.
Constraints are used to limit the type of data that can go into a table. This ensures the accuracy and reliability of the data in the table. If there is any violation between the constraint and the data action, the action is aborted.
Constraints can be column level or table level. Column level constraints apply to a column, and table level constraints apply to the whole table.

The following constraints are commonly used in SQL:

- NOT NULL - Ensures that a column cannot have a NULL value
- UNIQUE - Ensures that all values in a column are different
- PRIMARY KEY - A combination of a NOT NULL and UNIQUE. Uniquely identifies each row in a table
- FOREIGN KEY - Prevents actions that would destroy links between tables
- CHECK - Ensures that the values in a column satisfies a specific condition
- DEFAULT - Sets a default value for a column if no value is specified
- CREATE INDEX - Used to create and retrieve data from the database very quickly

NOT NULL Constraint:

By default, a column can hold NULL values.
The NOT NULL constraint enforces a column to NOT accept NULL values.
This enforces a field to always contain a value, which means that you cannot insert a new record, or update a record without adding a value to this field.
Example:

```
CREATE TABLE Persons (
    ID int NOT NULL,
    LastName varchar(255) NOT NULL,
    FirstName varchar(255) NOT NULL,
    Age int
);
```

```
ALTER TABLE Persons
MODIFY COLUMN Age int NOT NULL;
```

UNIQUE Constraint

The UNIQUE constraint ensures that all values in a column are different.
Both the UNIQUE and PRIMARY KEY constraints provide a guarantee for uniqueness for a column or set of columns.
A PRIMARY KEY constraint automatically has a UNIQUE constraint.
However, you can have many UNIQUE constraints per table, but only one PRIMARY KEY constraint per table.

```
CREATE TABLE Persons (
    ID int NOT NULL UNIQUE,
    LastName varchar(255) NOT NULL,
    FirstName varchar(255),
    Age int
);
```

```
CREATE TABLE Persons (
    ID int NOT NULL,
    LastName varchar(255) NOT NULL,
    FirstName varchar(255),
    Age int,
    UNIQUE (ID)
);
```

```
CREATE TABLE Persons (
    ID int NOT NULL,
    LastName varchar(255) NOT NULL,
    FirstName varchar(255),
    Age int,
    CONSTRAINT UC_Person UNIQUE (ID,LastName)
);
```

```
ALTER TABLE Persons
ADD UNIQUE (ID);
```

```
ALTER TABLE Persons
ADD CONSTRAINT UC_Person UNIQUE (ID,LastName);
```

PRIMARY KEY CONSTRAINTS

PRIMARY KEY constraint uniquely identifies each record in a table.
Primary keys must contain UNIQUE values, and cannot contain NULL values.
A table can have only ONE primary key; and in the table, this primary key can consist of single or multiple columns (fields).

- Syntax is same as that of UNIQUE constraints.

FOREIGN KEY Constraint

The FOREIGN KEY constraint is used to prevent actions that would destroy links between tables.
A FOREIGN KEY is a field (or collection of fields) in one table, that refers to the PRIMARY KEY in another table.
The table with the foreign key is called the child table, and the table with the primary key is called the referenced or parent table.

```
CREATE TABLE Orders (
    OrderID int NOT NULL,
    OrderNumber int NOT NULL,
    PersonID int,
    PRIMARY KEY (OrderID),
    FOREIGN KEY (PersonID) REFERENCES Persons(PersonID)
);
```

```
CREATE TABLE Orders (
    OrderID int NOT NULL,
    OrderNumber int NOT NULL,
    PersonID int,
    PRIMARY KEY (OrderID),
    CONSTRAINT FK_PersonOrder FOREIGN KEY (PersonID)
    REFERENCES Persons(PersonID)
);
```

CHECK Constraint

The CHECK constraint is used to limit the value range that can be placed in a column.
If you define a CHECK constraint on a column it will allow only certain values for this column.
If you define a CHECK constraint on a table it can limit the values in certain columns based on values in other columns in the row.

```
CREATE TABLE Persons (
    ID int NOT NULL,
    LastName varchar(255) NOT NULL,
    FirstName varchar(255),
    Age int,
    CHECK (Age>=18)
);
```

```
CREATE TABLE Persons (
    ID int NOT NULL,
    LastName varchar(255) NOT NULL,
    FirstName varchar(255),
    Age int CHECK (Age>=18)
);
```

```
CREATE TABLE Persons (
    ID int NOT NULL,
    LastName varchar(255) NOT NULL,
    FirstName varchar(255),
    Age int,
    City varchar(255),
    CONSTRAINT CHK_Person CHECK (Age>=18 AND City='Sandnes')
);
```

DEFAULT Constraint

The DEFAULT constraint is used to set a default value for a column.
The default value will be added to all new records, if no other value is specified.

```
CREATE TABLE Persons (
    ID int NOT NULL,
    LastName varchar(255) NOT NULL,
    FirstName varchar(255),
    Age int,
    City varchar(255) DEFAULT 'Sandnes'
);
```

DATA TYPES

- CHAR(size) - A FIXED length string (can contain letters, numbers, and special characters). The size parameter specifies the column length in characters - can be from 0 to 255. Default is 1
- VARCHAR(size) - A VARIABLE length string (can contain letters, numbers, and special characters). The size parameter specifies the maximum string length in characters - can be from 0 to 65535

- BINARY(size)- Equal to CHAR(), but stores binary byte strings. The size parameter specifies the column length in bytes. Default is 1
- VARBINARY(size) - Equal to VARCHAR(), but stores binary byte strings. The size parameter specifies the maximum column length in bytes.
- BOOL(BOOLEAN) - Zero is considered as false, nonzero values are considered as true.
- INT(size) - A medium integer. Signed range is from -2147483648 to 2147483647. Unsigned range is from 0 to 4294967295. The size parameter specifies the maximum display width (which is 255). INTEGER(size) is Equal to INT(size)
- DATE - A date. Format: YYYY-MM-DD. The supported range is from '1000-01-01' to '9999-12-31'
- DATETIME(fsp) - A date and time combination. Format: YYYY-MM-DD hh:mm:ss. The supported range is from '1000-01-01 00:00:00' to '9999-12-31 23:59:59'. Adding DEFAULT and ON UPDATE in the column definition to get automatic initialization and updating to the current date and time
- TIMESTAMP(fsp) - A timestamp. TIMESTAMP values are stored as the number of seconds since the Unix epoch ('1970-01-01 00:00:00' UTC). Format: YYYY-MM-DD hh:mm:ss. The supported range is from '1970-01-01 00:00:01' UTC to '2038-01-09 03:14:07' UTC. Automatic initialization and updating to the current date and time can be specified using DEFAULT CURRENT_TIMESTAMP and ON UPDATE CURRENT_TIMESTAMP in the column definition
- TIME(fsp) - A time. Format: hh:mm:ss. The supported range is from '-838:59:59' to '838:59:59'
- YEAR - A year in four-digit format. Values allowed in four-digit format: 1901 to 2155, and 0000.

MYSQL FUNCTIONS :

Link : https://www.w3schools.com/sql/sql_ref_mysql.asp

char_length - gives the no of characters in the string. also includes the spaces.
SELECT CHAR_LENGTH("SQL Tutorial") AS LengthOfString; // ans = 12.
SELECT CHAR_LENGTH(replace("SQL Tutorial", ' ', '')) AS LengthOfString; // ans = 11.

The DATEDIFF() function returns the number of days between two date values.
datediff(date1, date2)

The IFNULL() function returns a specified value if the expression is NULL.
If the expression is NOT NULL, this function returns the expression.
IFNULL(expression, alt_value)

The IF() function returns a value if a condition is TRUE, or another value if a condition is FALSE.
IF(condition, value_if_true, value_if_false)

WINDOW FUNCTIONS :

https://www.scaler.com/topics/window-functions-in-mysql/

```
Window_Function()
    OVER(   [<PARTITION BY ClAUSE>]
            [<ORDER BY Clause>]
            [<ROW or RANGE Clause>]
        )
```

Partition Clause

This clause is used to divide or breaks the rows into partitions, and the partition boundary separates these partitions. The window function operates on each partition, and when it crosses the partition boundary, it will be initialized again.

```
PARTITION BY column_name1, column_name2
```

ORDER BY Clause

This clause is used to specify the order of the rows within a partition.

```
ORDER BY column_name1, column_name2
```

Frame Definition

Sets the frame size.

```
RANGE BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING
```

```
RANGE BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW
```

```
RANGE BETWEEN INTERVAL '1' DAY PRECEDING AND CURRENT ROW
```

```
RANGE BETWEEN 1 PRECEDING AND CURRENT ROW
```

ROW_NUMBER() Function

ROW_NUMBER is one of the most common ranking window functions in MySQL which provides a normal serial number to each row present within a partition. It is just a normal serial-wise numbering of rows in ascending order. Even the rows which have the same value when using the ORDER BY clause has been given a different row number.

```
SELECT
row_number() OVER([<PARTITION BY ClAUSE>]
                  [<ORDER BY Clause>]
                  [<ROW or RANGE Clause>])
                  as alias_name
FROM table_name;
```

RANK()

RANK() is a kind of ranking window function in MySQL which is used to assign ranks to the rows within an ordered partition. The rows which contain the same values will be assigned the same rank, with the next ranking or rankings skipped. For example, if we have 4 items with the same value starting with ranking 1 then all 4 items will have rank 1, and the ranking from the 5th item would be 5 with 2,3,4 skipped.

```
SELECT
RANK() OVER ([<PARTITION BY ClAUSE>]
             [<ORDER BY Clause>]
             [<ROW or RANGE Clause>])
             as alias_name
FROM table_name;
```

DENSE_RANK()

The DENSE_RANK() function is a type of ranking window function in MySQL which is used to assign rankings to the rows within an ordered partition. DENSE_RANK() is almost as same as the RANK() window function but in DENSE_RANK() no rankings are skipped even if the rows have the same value.

```
SELECT
DENSE_RANK() OVER ([<PARTITION BY ClAUSE>]
             [<ORDER BY Clause>]
             [<ROW or RANGE Clause>])
             as alias_name
FROM table_name;
```

LAG() and LEAD() Functions

LAG() and LEAD() both are types of analytical window functions in MySQL. As the name suggests, the LAG() function starts the value from the previous row (returns NULL if starting from the first row as no preceding row exists). It returns the value of the row before the current row of the partition. Whereas the LEAD() function returns the value of the row after the current row in a partition. It starts from the value of the next row (returns NULL if no row if no more rows are available).

```
SELECT
LAG() OVER([<PARTITION BY ClAUSE>]
           [<ORDER BY Clause>]
           [<ROW or RANGE Clause>])
FROM table_name;
```

```
SELECT
LEAD() OVER([<PARTITION BY ClAUSE>]
            [<ORDER BY Clause>]
            [<ROW or RANGE Clause>])
FROM table_name;
```

FIRST_VALUE() and LAST_VALUE() Functions

FIRST_VALUE() and LAST_VALUE() are one of the most commonly used analytics window functions in MySQL. The FIRST_VALUE() function shows the value from the first row of a window frame or within a partition. LAST_VALUE() will return the value of the last row of a window frame or within a partition.

```
SELECT
FIRST_VALUE() OVER([<PARTITION BY ClAUSE>]
                   [<ORDER BY Clause>]
                   [<ROW or RANGE Clause>])
FROM table_name;
```

```
SELECT
LAST_VALUE() OVER([<PARTITION BY ClAUSE>]
                  [<ORDER BY Clause>]
                  [<ROW or RANGE Clause>])
FROM table_name;
```

NTILE() Function

NTILE() function is a kind of analytical window function in MySQL which is used to divide rows into groups within an ordered partition. Each of the divided groups will be assigned a number which will be the same for that group of rows. Every group is represented by a number.

```
SELECT
NTILE(number_of_groups) OVER([<PARTITION BY ClAUSE>]
                             [<ORDER BY Clause>]
                             [<ROW or RANGE Clause>])
                             as alias_name
FROM table_name;
```

Nth_Value

Nth_Value Function is a window function that is used to retrieve the value of the Nth row from a window frame. If there is no Nth row then the function returns NULL.

```
NTH_VALUE(expression, N)
From First
Over (
    partition_clause
    order_clause
    frame_clause
)
```

CTE

It stands for common table expression.

Example :

```
WITH
  cte1 AS (SELECT a, b FROM table1),
  cte2 AS (SELECT c, d FROM table2)
SELECT b, d FROM cte1 JOIN cte2
WHERE cte1.a = cte2.c;
```

IMPORTANT PROBLEMS :

https://leetcode.com/problems/students-and-examinations/?envType=study-plan-v2&envId=top-sql-50

https://leetcode.com/problems/managers-with-at-least-5-direct-reports/?envType=study-plan-v2&envId=top-sql-50

https://leetcode.com/problems/confirmation-rate/?envType=study-plan-v2&envId=top-sql-50

https://leetcode.com/problems/percentage-of-users-attended-a-contest/?envType=study-plan-v2&envId=top-sql-50

https://leetcode.com/problems/queries-quality-and-percentage/?envType=study-plan-v2&envId=top-sql-50

https://leetcode.com/problems/game-play-analysis-iv/?envType=study-plan-v2&envId=top-sql-50

https://leetcode.com/problems/exchange-seats/description/?envType=study-plan-v2&envId=top-sql-50

https://leetcode.com/problems/last-person-to-fit-in-the-bus/description/?envType=study-plan-v2&envId=top-sql-50

https://leetcode.com/problems/consecutive-numbers/description/?envType=study-plan-v2&envId=top-sql-50

https://leetcode.com/problems/count-salary-categories/description/?envType=study-plan-v2&envId=top-sql-50

https://leetcode.com/problems/product-price-at-a-given-date/?envType=study-plan-v2&envId=top-sql-50

https://leetcode.com/problems/department-top-three-salaries/description/?envType=study-plan-v2&envId=top-sql-50

https://leetcode.com/problems/investments-in-2016/description/?envType=study-plan-v2&envId=top-sql-50

https://leetcode.com/problems/friend-requests-ii-who-has-the-most-friends/description/?envType=study-plan-v2&envId=top-sql-50

https://leetcode.com/problems/restaurant-growth/?envType=study-plan-v2&envId=top-sql-50

https://leetcode.com/problems/movie-rating/description/?envType=study-plan-v2&envId=top-sql-50

https://leetcode.com/problems/human-traffic-of-stadium/solutions/4049359/no-complex-window-functions-only-logic/

https://leetcode.com/problems/trips-and-users/description/

https://leetcode.com/problems/market-analysis-i/description/

https://leetcode.com/problems/capital-gainloss/description/

https://leetcode.com/problems/tree-node/description/