SQL

DDL, DCL, DML, Joins, Subqueries and Functions

MY SQL

DDL, DCL, DML, Joins, Subqueries and Functions

- Insert, Create, Delete and Update Statements
- Operators
- Joins & Union
- Aggregation Fun / Date Fun / String Fun / Numeric Fun
- Sub Query / CTE – Common Table Expression
- Stored Procedure
- Functions
- Triggers / Cursor
- Index

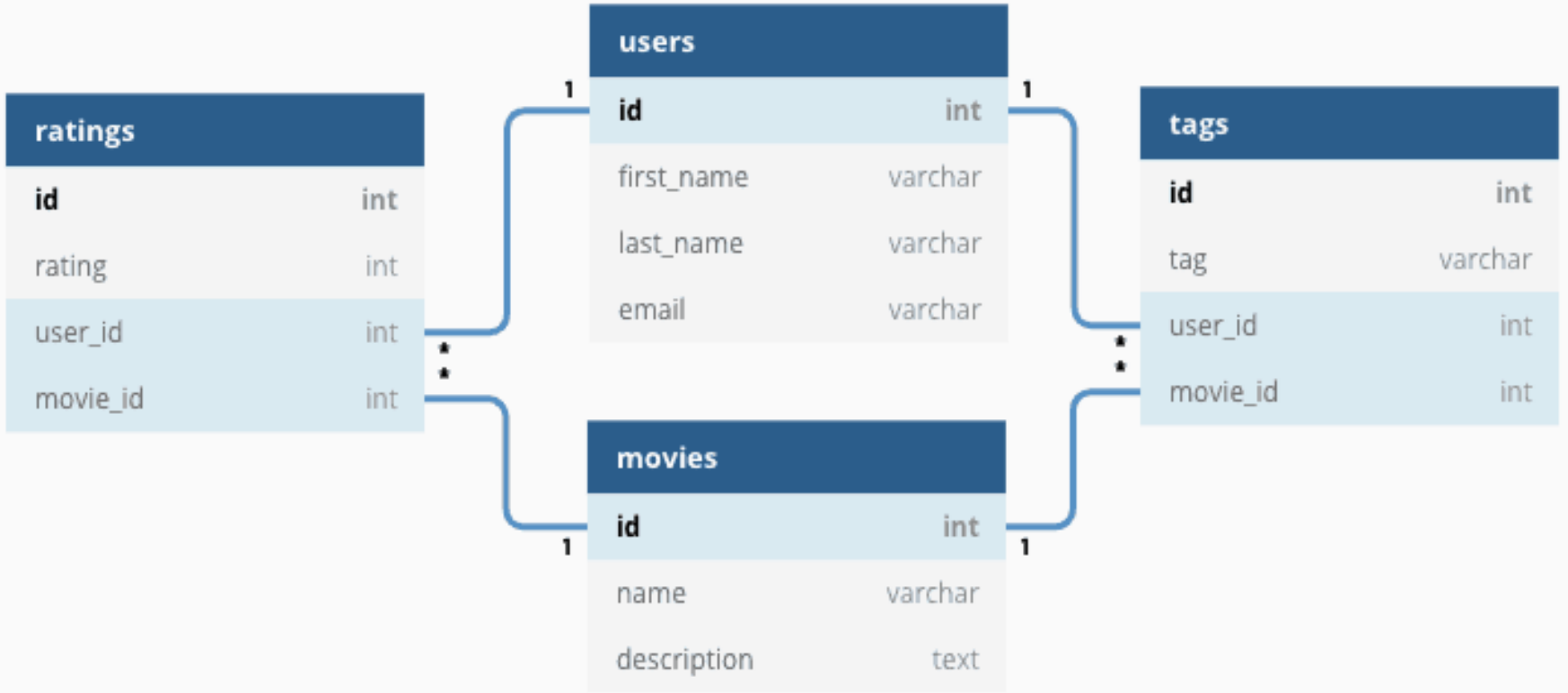
WHAT IS SQL?

- SQL – Structured Querying Language is a scripting language to access databases
- It includes database creation, deletion, fetching rows, modifying rows, etc.
- It helps in manipulating and retrieving data stored in a relational database like MySQL, MS Access, Oracle, Sybase, Informix, Postgres and SQL Server as these serves use SQL as their standard database language.

WHAT IS A RELATIONAL DATABASE?

- A relational database is a type of database that stores and provides access to data points that are related to one another.
- Relational databases are based on the relational model, an intuitive, straightforward way of representing data in tables.
- In a relational database, each row in the table is a record with a unique ID called the key.
- The columns of the table hold attributes of the data, and each record usually has a value for each attribute, making it easy to establish the relationships among data points.

WHAT IS A RELATIONAL DATABASE?



DIFFERENCE BETWEEN MySQL, PostgreSQL and SQL SERVER

PostgreSQL, MySQL, and Microsoft SQL Server (MS SQL) are three popular relational database management systems (RDBMS), each with its own strengths and capabilities. Let's compare these three databases in terms of various factors

Licensing and Cost:

PostgreSQL: PostgreSQL is open-source and released under the PostgreSQL License, which is similar to the MIT License. It is free to use, and there are no licensing costs.

MySQL: MySQL is also open-source and available under the GNU General Public License (GPL), but it also offers commercial licenses for additional features and support.

MS SQL: Microsoft SQL Server is a commercial database and requires licensing fees, which can vary based on the edition and features.

Features and Capabilities:

PostgreSQL: PostgreSQL is known for its advanced features and extensibility. It supports complex queries, data types, indexing options, and offers support for JSON, XML, and geospatial data. It has a reputation for data integrity and offers support for advanced indexing techniques.

MySQL: MySQL is known for its simplicity and ease of use. It provides a good balance between performance and features, with a focus on web applications and small to medium-sized databases. It offers various storage engines, including InnoDB and MyISAM.

MS SQL: Microsoft SQL Server offers a wide range of features including comprehensive business intelligence, data analysis, and reporting tools. It is often chosen for enterprise-level applications that require advanced analytics and reporting capabilities.

Performance:

PostgreSQL: PostgreSQL is known for its extensibility and support for complex queries. It can handle large amounts of data and is suitable for complex transactions.

MySQL: MySQL is known for its fast read operations and is often used for web applications. It performs well for simple to moderately complex queries and is well-suited for read-heavy workloads.

MS SQL: SQL Server is optimized for high-performance and scalability, making it suitable for large-scale databases and complex workloads.

Community and Support:

PostgreSQL: PostgreSQL has a strong open-source community and offers extensive documentation, forums, and resources for support.

MySQL: MySQL also has a strong community and offers a wide range of resources for support, including documentation and forums.

MS SQL: SQL Server benefits from Microsoft's extensive support and resources, including official documentation, forums, and paid support plans.

Platform Compatibility:

PostgreSQL: PostgreSQL is cross-platform and supports various operating systems including Windows, Linux, and macOS.

MySQL: MySQL is also cross-platform and supports various operating systems including Windows, Linux, and macOS.

MS SQL: While primarily associated with Windows, there is a version of SQL Server (SQL Server for Linux) that allows it to run on Linux systems as well.

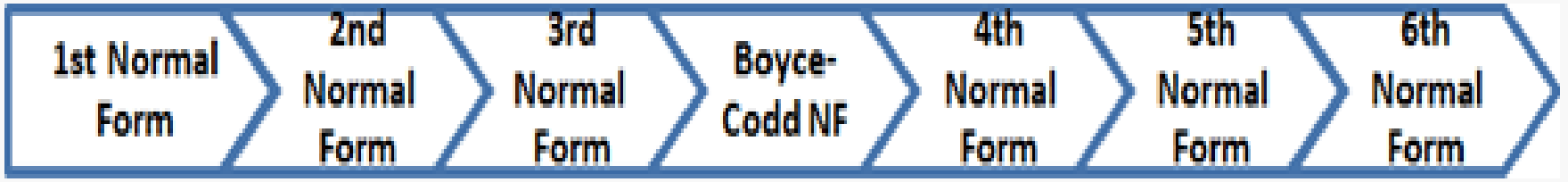
Normal Forms

What is Normalization?

- Normalization is the process of organizing the data in the database.
- Normalization is used to minimize the redundancy from a relation or set of relations. It is also used to eliminate the undesirable characteristics like Insertion, Update and Deletion Anomalies.
- Normalization divides the larger table into the smaller table and links them using relationship.
- The normal form is used to reduce redundancy from the database table.

Normal Forms

List of normal forms in database:



- 1NF (First Normal Form)
- 2NF (Second Normal Form)
- 3NF (Third Normal Form)
- BCNF (Boyce-Codd Normal Form)
- 4NF (Fourth Normal Form)
- 5NF (Fifth Normal Form)
- 6NF (Sixth Normal Form)

Normal Forms

Normalization examples:

- No Normalisation

FULL NAMES	PHYSICAL ADDRESS	MOVIES RENTED	SALUTATION
Janet Jones	First Street Plot No 4	Pirates of the Caribbean, Clash of the Titans	Ms.
Robert Phil	3rd Street 34	Forgetting Sarah Marshal, Daddy's Little Girls	Mr.
Robert Phil	5th Avenue	Clash of the Titans	Mr.

Normal Forms

Normalization examples:

- Rules for1NF (First Normal Form):

1. Each cell in the table should contain only one single value.
2. Every record in the table should be unique (no duplicate records allowed)

FULL NAMES	PHYSICAL ADDRESS	MOVIES RENTED	SALUTATION
Janet Jones	First Street Plot No 4	Pirates of the Caribbean	Ms.
Janet Jones	First Street Plot No 4	Clash of the Titans	Ms.
Robert Phil	3rd Street 34	Forgetting Sarah Marshal	Mr.
Robert Phil	3rd Street 34	Daddy's Little Girls	Mr.
Robert Phil	5th Avenue	Clash of the Titans	Mr.

Normal Forms

Normalization examples:

- Rules for 1NF (First Normal Form):

1. Each cell in the table should contain only one single value.
2. Every record in the table should be unique (no duplicate records allowed)

FULL NAMES	PHYSICAL ADDRESS	MOVIES RENTED	SALUTATION
Janet Jones	First Street Plot No 4	Pirates of the Caribbean	Ms.
Janet Jones	First Street Plot No 4	Clash of the Titans	Ms.
Robert Phil	3rd Street 34	Forgetting Sarah Marshal	Mr.
Robert Phil	3rd Street 34	Daddy's Little Girls	Mr.
Robert Phil	5th Avenue	Clash of the Titans	Mr.

Normal Forms

Basic understanding of primary key, composite key and foreign key:

- A primary key is a single column in a database which contains unique values in each row to represent each records respectively.
- A composite key is a primary key where multiple columns may be used to identify a record uniquely. Every composite key is a primary key but every primary key may or may not be composite key.
- Foreign key is the column values in database which references the primary key of other table. This way we can connect our tables to create relationships between them.

Normal Forms

Normalization examples:

- Rules for 2NF (Second Normal Form):

1. It should be 1NF.
2. Single column primary key.

MEMBERSHIP ID	FULL NAMES	PHYSICAL ADDRESS	SALUTATION
1	Janet Jones	First Street Plot No 4	Ms.
2	Robert Phil	3rd Street 34	Mr.
3	Robert Phil	5th Avenue	Mr.

MEMBERSHIP ID	MOVIES RENTED
1	Pirates of the Caribbean
1	Clash of the Titans
2	Forgetting Sarah Marshal
2	Daddy's Little Girls
3	Clash of the Titans

Normal Forms

Normalization examples:

- Rules for 3NF (Third Normal Form):

1. It should be 2NF.
2. Has no transitive functional dependencies

What are transitive functional dependencies?

A transitive functional dependency is when changing a non-key column, might cause any of the other non-key columns to change. (key column is column with primary, foreign or composite key).

MEMBERSHIP ID	FULL NAMES	PHYSICAL ADDRESS	SALUTATION
1	Janet Jones	First Street Plot No 4	Ms.
2	Robert Phil	3rd Street 34	Mr.
3	Robert Phil	5th Avenue	Mr.

Change in Name

May Change Salutation

Normal Forms

Normalization examples:

- Rules for 3NF (Third Normal Form):

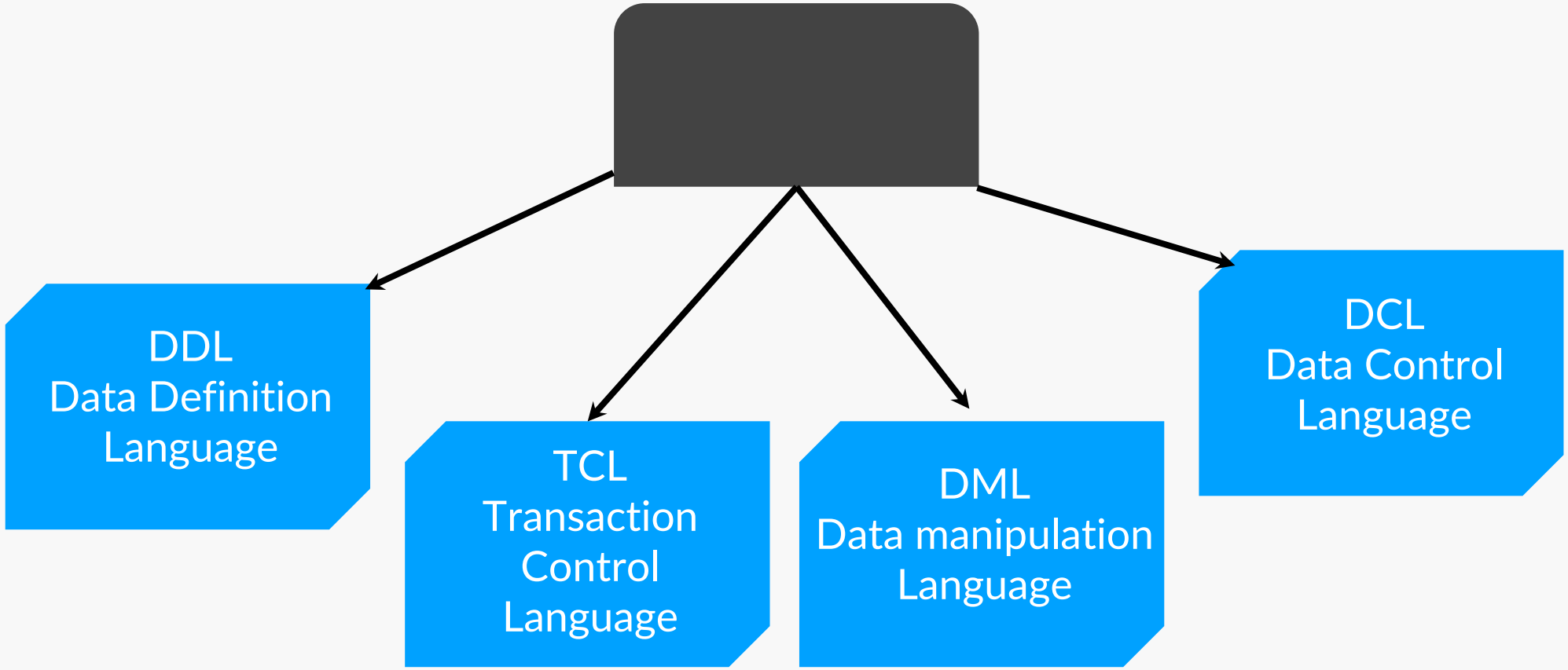
1. It should be 2NF.
2. Has no transitive functional dependencies

Membership ID	Full Names	Physical Address	Salutation ID
1	Janet Jones	First Street Plot No 4	2
2	Robert Phil	3rd Street 34	1
3	Robert Phil	5th Avenue	1

Membership ID	Movies Rented
1	Pirates of the Caribbean
1	Clash of the Titans
2	Forgetting Sarah Marshal
2	Daddy's Little Girls
3	Clash of the Titans

Salutation ID	Salutation
1	Mr.
2	Ms.
3	Mrs.
4	Dr.

SQL CLASSIFICATION



DDL - DATA DEFINITION LANGUAGE

- DDL or Data Definition Language consists of the SQL commands that can be used to define the database schema.

- DDL allows to add / modify / delete the logical structures which contain the data or which allow users to access / maintain the data

  - CREATE - to create a database and its objects
  - ALTER - alters the structure of the existing databases
  - DROP - delete objects from the databases
  - TRUNCATE - remove all records from a table, including memory allocated for the records are removed

DML- DATA MANIPULATION LANGUAGE

- DML commands enable us to work with data that goes into the database such as INSERT, SELECT, UPDATE and DELETE records

  - SELECT - Retrieves data from a Table/Database
  - INSERT - Insert data into a Table/Database
  - UPDATE - Updates existing data within a Table/Database
  - DELETE - Deletes all records from a database Table/Database

DCL - DATA CONTROL LANGUAGE

- DCL commands allow us to give permission on the particular database or table such as GRANT & REVOKE

  - GRANT – Allows user to access privileges to the database/table
  - REVOKE - withdraw users access privileges given by using the GRANT command

TCL - TRANSACTION CONTROL LANGUAGE

- TCL Commands lets to commit and revoke changes in the server. Also, it allows us to save points wherever it is appropriate.

  - COMMIT– Saves all the changes done in the last transaction permanently.
  - ROLLBACK - Cancels all the modifications done in the last transaction.
  - SAVEPOINTS - Rollback operation will be performed on a part of transaction. The save points commands sets a name transaction save-point by the name of identifier. At the same time the present transaction has a save point with the similar name, then the old save point is deleted and a new one is assigned.

WORKING ON DATABASE

- Create Database
- Drop Database
- Manage Database

- CREATE DATABASE statement is used to create a new SQL database.
- The database name must be unique within the server instance.
- If you try to create a database with a name that already exists, MySQL throws an error.

Syntax

```
CREATE DATABASE DatabaseName
```

WORKING ON DATABASE

Create Database

Drop Database

Manage Database

- DROP DATABASE statement is used to delete an existing database in the instance.
- The DROP DATABASE statement deletes all tables in the database permanently.

Syntax

```
DROP DATABASE database_name
```

WORKING ON DATABASE

Create Database

Drop Database

Manage Database

- When you have multiple databases in your SQL Schema, then before starting your operation, you should select a database where all the operations would be performed.
- The SQL USE statement is used to select any existing database.

Syntax

```
USE DatabaseName
```

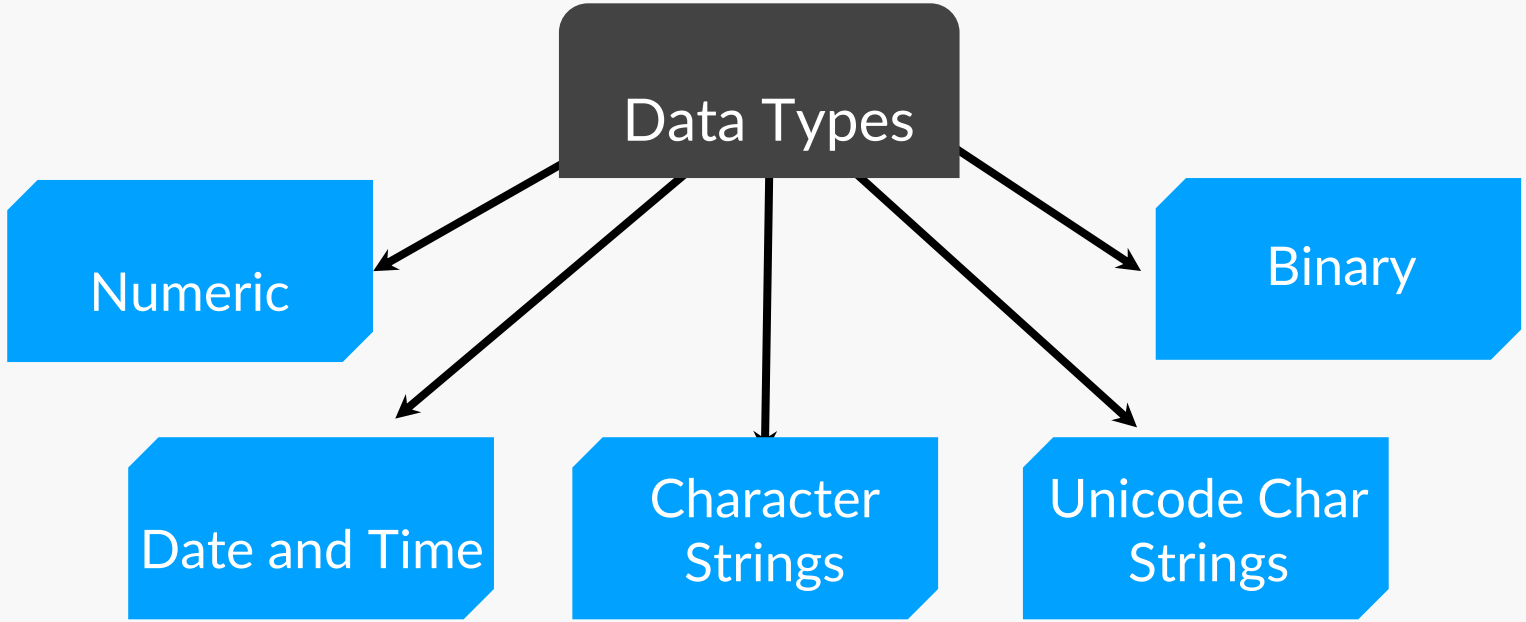
- Once a database is created, you can check it in the list of databases by SQL command SHOW DATABASE

Syntax

```
SHOW DATABASES
```

DATA TYPES

- Data Types are attributes that specifies the type of data of any object.
- Each column, variable and expression has a related data type in SQL.
- We can use these data types while creating our tables.



DATA TYPES

Numeric Data Types

Approximate Numeric Data Types

Date and Time Data Types

Character Strings Data Types

Unicode Character Strings Data Types

Binary Data Types

Datatype	Starts with	Ends with
bit	0	1
tinyint	0	255
smallint	-32,768	32,767
int	-2,147,483,648	2,147,483,647
bigint	-9,223,372,036,854,770,000	9,223,372,036,854,770,000
decimal	1E+38	10^38 -1
numeric	1E+38	10^38 -1

DATA TYPES

- Numeric Data Types
- **Approximate Numeric Data Types**
- Date and Time Data Types
- Character Strings Data Types
- Unicode Character Strings Data Types
- Binary Data Types

Datatype	Starts with	Ends with
float	-1.79E + 308	1.79E + 308
real	-3.40E + 38	3.40E + 38

DATA TYPES

- Numeric Data Types
- Approximate Numeric Data Types
- **Date and Time Data Types**
- Character Strings Data Types
- Unicode Character Strings Data Types
- Binary Data Types

Datatype	Description
DATE	Stores date in the format YYYY-MM-DD
TIME	Stores time in the format HH:MI:SS
DATETIME	Stores date and time information in the format YYYY-MM-DD HH:MI:SS
TIMESTAMP	Stores number of seconds passed since the Unix epoch ('1970-01-01 00:00:00' UTC)
YEAR	Stores year in 2 digit or 4 digit format. Range 1901 to 2155 in 4-digit format. Range 70 to 69, representing 1970 to 2069.

DATA TYPES

- Numeric Data Types
- Approximate Numeric Data Types
- Date and Time Data Types
- **Character Strings Data Types**
- Unicode Character Strings Data Types
- Binary Data Types

Datatype	Description
CHAR	Fixed length with maximum length of 8,000 characters
VARCHAR	Variable length storage with maximum length of 8,000 characters
VARCHAR(max)	Variable length storage with provided max characters, not supported in MySQL
TEXT	Variable length storage with maximum size of 2GB data

DATA TYPES

- Numeric Data Types
- Approximate Numeric Data Types
- Date and Time Data Types
- Character Strings Data Types
- **Unicode Character Strings Data Types**
- Binary Data Types

Datatype	Description
NCHAR	Fixed length with maximum length of 4,000 characters
NVARCHAR	Variable length storage with maximum length of 4,000 characters
NVARCHAR(max)	Variable length storage with provided max characters
NTEXT	Variable length storage with maximum size of 1GB data

DATA TYPES

- Numeric Data Types
- Approximate Numeric Data Types
- Date and Time Data Types
- Character Strings Data Types
- Unicode Character Strings Data Types
- **Binary Data Types**

Datatype	Description
BINARY	Fixed length with maximum length of 8,000 bytes
VARBINARY	Variable length storage with maximum length of 8,000 bytes
VARBINARY(max)	Variable length storage with provided max bytes
IMAGE	Variable length storage with maximum size of 2GB binary data

TABLE CONSTRAINTS

- Constraints are the rules enforced on the data columns of a table.
- These 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 database.

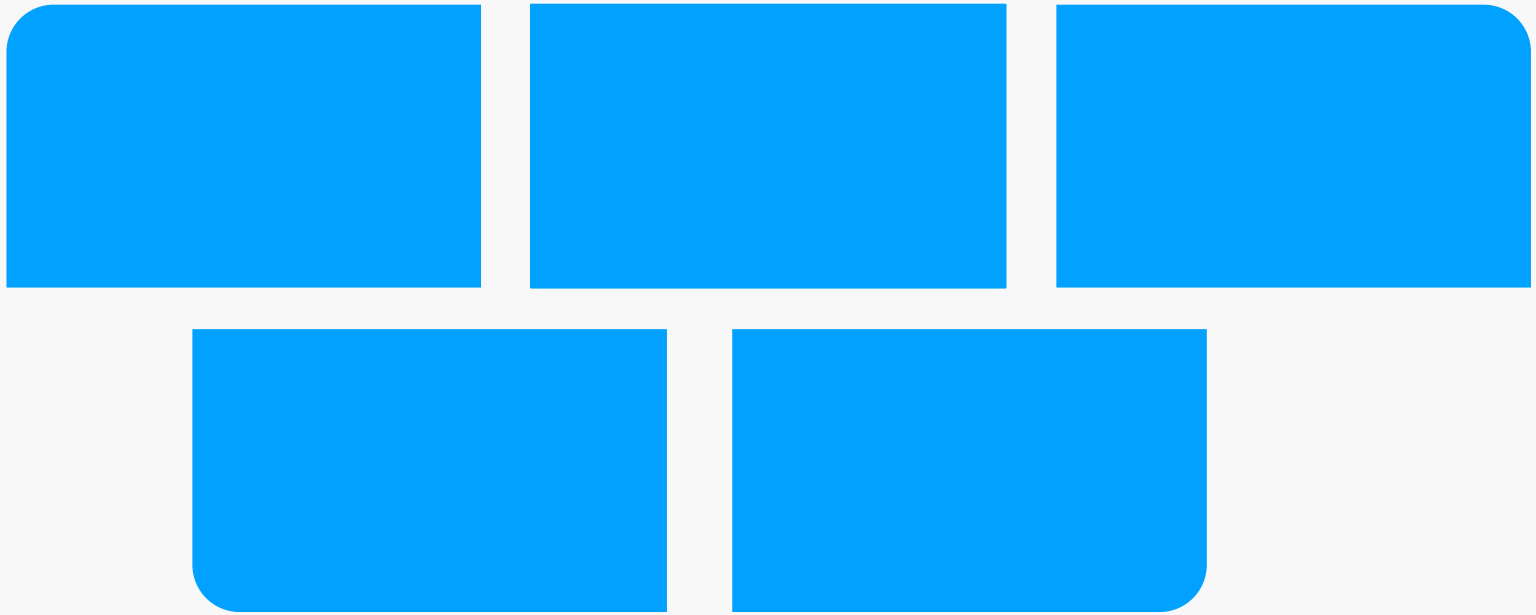


TABLE CONSTRAINTS

- Not Null
- Unique
- Primary Key
- Foreign Key
- Check

- By default, a column can hold NULL values. If we do not want a column to have any EMPTY value, then we need to define such a constraint on this column specifying that EMPTY SPACE is now not allowed for that column.

Example

```
CREATE TABLE CUSTOMERS(
ID INT NOT NULL,
NAME VARCHAR (20) NOT  NULL,
AGE INT NOT NULL,
ADDRESS CHAR (25),
SALARY DECIMAL (18, 2))
```

TABLE CONSTRAINTS

Not Null

Unique

Primary Key

Foreign Key

Check

- Ensures that all values in a column are different.
- The UNIQUE Constraint prevents two records from having identical values in a column.

Example

```
CREATE TABLE CUSTOMERS(
ID INT NOT NULL,
NAME VARCHAR (20) NOT NULL,
AGE INT NOT NULL UNIQUE,
ADDRESS CHAR (25),
SALARY DECIMAL(18,2) DEFAULT 5000.00)
```

TABLE CONSTRAINTS

- Not Null
- Unique
- **Primary Key**
- Foreign Key
- Check

- A primary key is a field in a table which uniquely identifies each row/record in a database table.
- Primary keys must contain unique values.
- A primary key column cannot have NULL values.
- A table can have only one primary key.

Example

```
CREATE TABLE CUSTOMERS(
ID INT NOT NULL,
NAME VARCHAR (20) NOT NULL,
AGE INT NOT NULL UNIQUE,
ADDRESS CHAR (25),
SALARY DECIMAL(18,2) DEFAULT 5000.00)
PRIMARY KEY (ID))
```

TABLE CONSTRAINTS

- Not Null
- Unique
- Primary Key
- **Foreign Key**
- Check

- A foreign key is a key used to link two tables together. This is sometimes also called as a referencing key.
- A Foreign Key is a column whose values match a Primary Key in a different table.

Example

```
CREATE TABLE CUSTOMERS(
ID INT NOT NULL,
NAME VARCHAR (20) NOT NULL,
AGE INT NOT NULL UNIQUE,
ADDRESS CHAR (25),
SALARY DECIMAL(18,2) DEFAULT 5000.00)
FOREIGN KEY (ID))
```

TABLE CONSTRAINTS

- Not Null
- Unique
- Primary Key
- Foreign Key
- **Check**

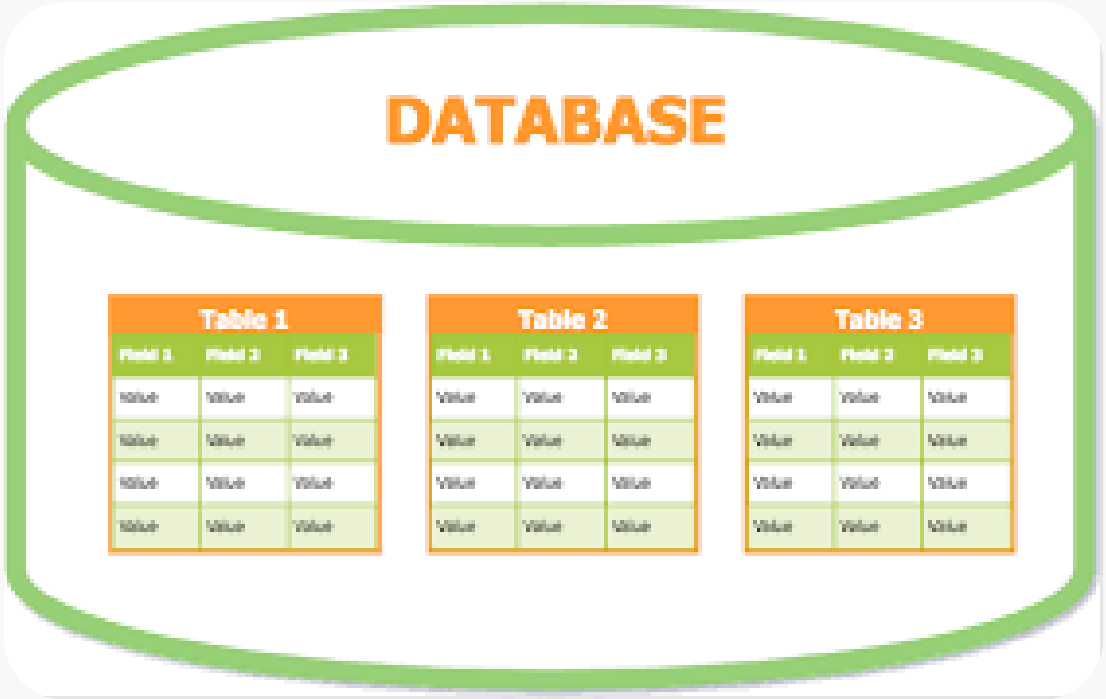
- The CHECK Constraint enables a condition to check the value being entered into a record.
- If the condition evaluates to false, the record violates the constraint and will not be entered into the table.

Example

```
CREATE TABLE New_table (ID INT NOT NULL,
NAME VARCHAR (20)  NOT NULL,
AGE INT NOT NULL CHECK (AGE >= 18),
ADDRESS CHAR(25),
SALARY DECIMAL (18, 2), PRIMARY KEY (ID) )
```

WORKING WITH TABLES

- Inside databases, working with tables becomes essential.
- Creating tables, Accessing values from them, modifying and deleting them are part of this module.



WORKING WITH TABLES

- Create Table
- INSERT Query
- UPDATE Query
- ALTER Table
- DELETE Query
- DROP Table
- Delete Table
- Truncate Table

- Creating a basic table involves naming the table and defining its columns and each column's data type.
- While creating, it is necessary to specify the data type, constraints to the table.

Syntax

CREATE TABLE <table_name>(column1 datatype <const>, column2 datatype,...)

Example

```
CREATE TABLE CUSTOMERS(
ID INT ,
NAME VARCHAR (20),
AGE INT,
ADDRESS CHAR (25),
SALARY DECIMAL (18, 2))
```

WORKING WITH TABLES

- Create Table
- **INSERT Query**
- UPDATE Query
- ALTER Table
- DELETE Query
- DROP Table
- Delete Table
- Truncate Table

- INSERT INTO Statement is used to add new rows of data to a table in the database.
- Multiple rows of data can be added at an instance.

Syntax

```
INSERT INTO TABLE_NAME (column1, column2, column3,...columnN)
VALUES  (value1, value2 , value3,...valueN)
Or
INSERT INTO TABLE_NAME VALUES (value1,value2,value3,...valueN)
```

Example

```
INSERT INTO CUSTOMERS (SALARY,ID,NAME,AGE,ADDRESS)
VALUES (2000.00 , 1, 'Ramesh ', 32, 'Ahmedabad' )
Or
INSERT INTO CUSTOMERS VALUES (7, 'Muffy', 24, 'Indore', 10000.00)
```

WORKING WITH TABLES

- Create Table
- INSERT Query
- **UPDATE Query**
- ALTER Table
- DELETE Query
- DROP Table
- Delete Table
- Truncate Table

- UPDATE Query is used to modify the existing records in a table. This can be used with the WHERE clause to update the selected rows, otherwise all the rows would be affected.

Syntax

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

Example

```
UPDATE CUSTOMERS
SET ADDRESS = 'Pune'
WHERE ID = 6
```

WORKING WITH TABLES

- Create Table
- INSERT Query
- UPDATE Query
- **ALTER Table**
- DELETE Query
- DROP Table
- Delete Table
- Truncate Table

- Alter Table statement helps in modifying tables, the values in it and manipulate the same.

1. To add a Column:

Syntax:

```
ALTER TABLE table_name
ADD new_column_name column_definition [FIRST | AFTER column_name]
```

2. To Rename a Column:

Syntax:

```
ALTER TABLE table_name
CHANGE COLUMN original_name new_name column_definition [FIRST | AFTER column_name]
```

WORKING WITH TABLES

- Create Table
- INSERT Query
- UPDATE Query
- ALTER Table
- **DELETE Query**
- DROP Table
- Delete Table
- Truncate Table

- DELETE Query is used to delete the existing records from a table.
- You can use the WHERE clause with a DELETE query to delete the selected rows, otherwise all the records would be deleted.

Syntax

DELETE FROM table_name

Or

DELETE FROM table_name WHERE [condition]

Example

DELETE FROM CUSTOMERS

Or

DELETE FROM CUSTOMERS WHERE ID = 6

WORKING WITH TABLES

- Create Table
- INSERT Query
- UPDATE Query
- ALTER Table
- DELETE Query
- **DROP Table**
- Delete Table
- Truncate Table

- DROP TABLE statement is used to remove a table definition and all the data, indexes, triggers, constraints and permission specifications for that table.

Syntax

DROP TABLE table_name

Example

DROP TABLE CUSTOMERS

WORKING WITH TABLES

- Create Table
- INSERT Query
- UPDATE Query
- ALTER Table
- DELETE Query
- DROP Table
- **Delete Table**
- Truncate Table

- DELETE TABLE statement is used to remove a table's data but all the definations, indexes, triggers, constraints and permission specifications for that table exists.

Syntax

DELETE TABLE table_name

Example

DELETE TABLE CUSTOMERS

WORKING WITH TABLES

- Create Table
- INSERT Query
- UPDATE Query
- ALTER Table
- DELETE Query
- DROP Table
- Delete Table
- **Truncate Table**

- TRUNCATE TABLE command is used to delete complete data from an existing table.
- You can also use DROP TABLE command to delete complete table but it would remove complete table structure form the database and you would need to re-create this table once again if you wish you store some data.

Syntax

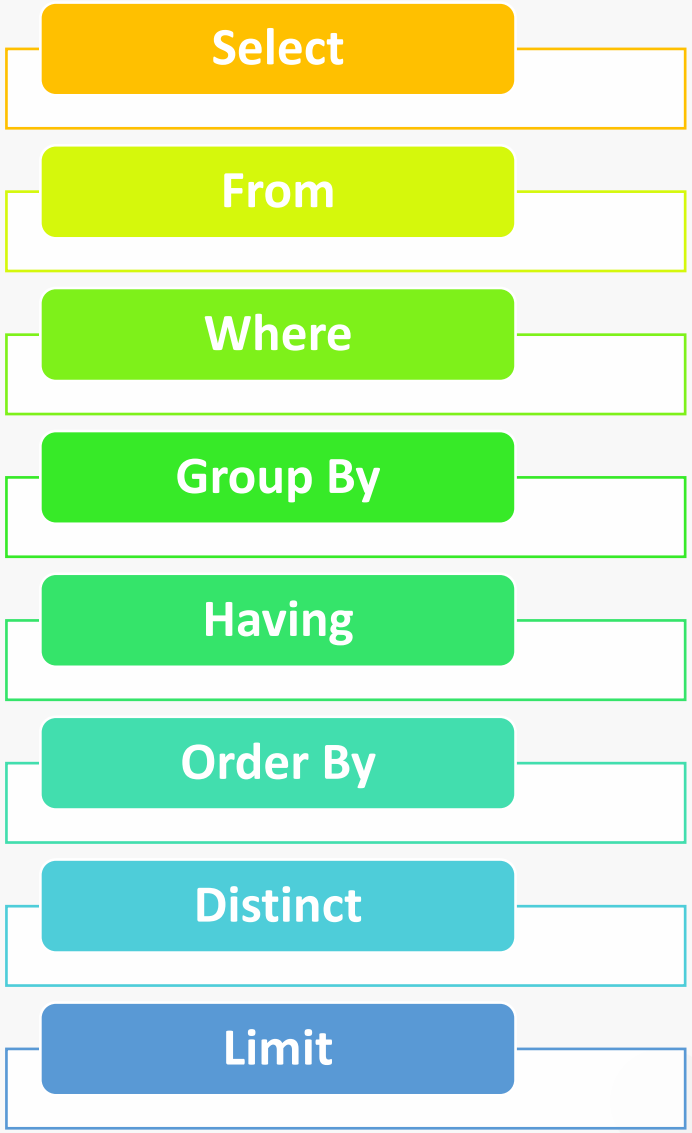
TRUNCATE TABLE table_name

Example

TRUNCATE TABLE CUSTOMERS

SQL STATEMENTS

- SQL statements are used to perform tasks such as update data on a database, or retrieve data from a database.
- Most of the data accessing/retrieval you need to perform on a database are done with these statements.
- These statements are not case sensitive. For ex., select is the same as SELECT



ORDER OF OPERATIONS

- SQL Select statements follow an order in which the queries will be executed as follows.

1. FROM, including
   JOINs
2. WHERE
3. GROUP BY
4. HAVING
5. WINDOW functions
6. SELECT
7. DISTINCT
8. UNION
9. ORDER BY
10. LIMIT and OFFSET

SQL STATEMENTS

- Select
- From
- Where
- Group By
- Having
- Order By
- Distinct
- Limit

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

- This statement also works as a print statement enabling us to customize the output. It will allow arithmetic, logical and aggregational operations to perform on the selected columns.

Syntax

```
SELECT * FROM table_name
Or
SELECT column1, column2, columnN FROM table_name
```

Example

```
SELECT * FROM CUSTOMERS
Or
SELECT ID, NAME, SALARY FROM CUSTOMERS
```

SQL STATEMENTS

- Select
- From
- Where
- Group By
- Having
- Order By
- Distinct
- Limit

- FROM clause is used to access the tables in the respective databases.
- It is the first statement that will get executed amongst all the other statements.

Syntax

```
SELECT column1, column2
FROM table_name
```

Example

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

SQL STATEMENTS

- Select
- From
- **Where**
- Group By
- Having
- Order By
- Distinct
- Limit

- WHERE clause is used to specify a condition while fetching the data from a single table or by joining with multiple tables.
- If the given condition is satisfied, then only it returns a specific value from the table.

Syntax

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

Example

```
SELECT ID, NAME, SALARY FROM CUSTOMERS WHERE SALARY > 2000
Or
SELECT ID, NAME, SALARY FROM CUSTOMERS WHERE NAME ='Hardik'
```

SQL STATEMENTS

- Select
- From
- Where
- **Group By**
- Having
- Order By
- Distinct
- Limit

- GROUP BY clause is used in collaboration with the SELECT statement to arrange identical data into groups.
- Helps in categorizing elements in the column value.

Syntax

```
SELECT column1, column2
FROM table_name
GROUP BY column1
```

Example

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

SQL STATEMENTS

- Select
- From
- Where
- Group By
- **Having**
- Order By
- Distinct
- Limit

- HAVING clause is used to filter the data from the already grouped data.
- To use Having Clause, we have to use Group By Clause. It is because this is applied after the Group by clause.

Syntax

```
SELECT [Column1],...[ColumnN] Aggregate_Function(Column_Name)
FROM [Source]
WHERE [Conditions] -- Optional
GROUP BY [Column1],...[ColumnN]
HAVING  [Conditions] -- Condition is on Aggregate Function(Column_Name)
```

SQL STATEMENTS

- Select
- From
- Where
- Group By
- Having
- **Order By**
- Distinct
- Limit

- ORDER BY clause is used to sort the data in ascending or descending order, based on one or more columns.
- Some databases sort the query results in an ascending order by default. MySQL does the same.

Syntax

SELECT * FROM table_name [ORDER BY Column1][ASC | DESC]

Example

SELECT * FROM CUSTOMERS ORDER BY NAME

Or

SELECT * FROM CUSTOMERS ORDER BY NAME DESC

SQL STATEMENTS

- Select
- From
- Where
- Group By
- Having
- Order By
- **Distinct**
- Limit

- Distinct Statement helps in retrieving unique records from the column value.

Syntax

SELECT DISTINCT column1 FROM table_name

Example

SELECT DISTINCT SALARY FROM CUSTOMERS

SQL STATEMENTS

- Select
- From
- Where
- Group By
- Having
- Order By
- Distinct
- **Limit**

- LIMIT clause is used to restrict the output to any given number so as to fetch limited number of records.

Syntax

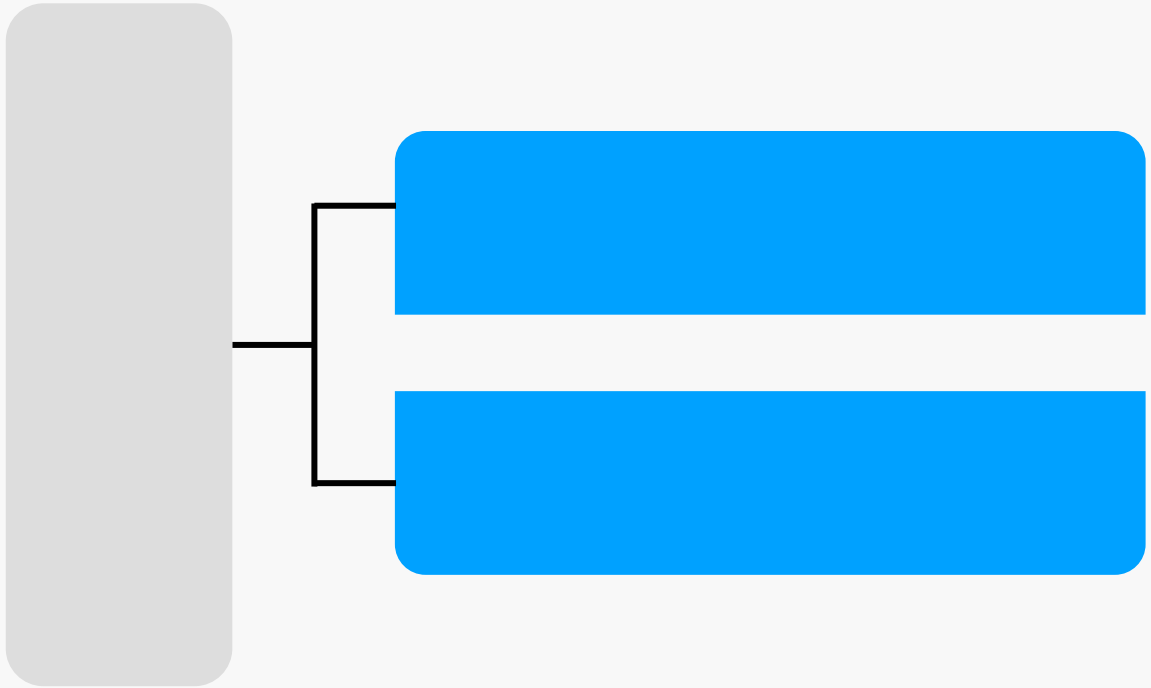
SELECT * FROM table_name LIMIT

Example

SELECT * FROM CUSTOMERS LIMIT 3

SQL OPERATORS

Operators are used to specify conditions in an SQL statement and to serve as conjunctions for multiple conditions in a statement.



ARITHMETIC OPERATORS

Operator	Meaning
+	Addition
-	Subtraction
*	Multiplication
/	Division
%	Returns the integer remainder of a division. For example, 21 % 6 = 3 because the remainder of 21 divided by 6 is 3.

LOGICAL OPERATORS

- **AND**
- OR
- NOT
- BETWEEN
- LIKE
- NOT LIKE
- IFNULL
- IN
- IS
- NOT IN
- ISNULL
- IS NOT NULL

- AND operator is generally used in the WHERE Clause to apply multiple filters on the records returned by SELECT Statement.
- This operator returns values only if all the conditions are satisfied.

Syntax:

```
AND()
```

Example:

```
SELECT First_Name
FROM customers
WHERE Profession LIKE '%Developer'
AND Yearly_Income > 75000
```

LOGICAL OPERATORS

- AND
- **OR**
- NOT
- BETWEEN
- LIKE
- NOT LIKE
- IFNULL
- IN
- IS
- NOT IN
- ISNULL
- IS NOT NULL

- OR operator is generally used in the WHERE Clause to apply multiple filters on the records returned by SELECT Statement.
- This operator returns values even if any one of the condition is satisfied.

Syntax:

OR()

Example:

```
SELECT First_Name
FROM customers
WHERE Profession LIKE '%Developer'
OR Yearly_Income > 75000
```

LOGICAL OPERATORS

- AND
- OR
- **NOT**
- BETWEEN
- LIKE
- NOT LIKE
- IFNULL
- IN
- IS
- NOT IN
- ISNULL
- IS NOT NULL

- NOT operator is generally used in the WHERE clause along with AND & OR operator to retrieve the opposite of the specified condition.

Syntax:

```
NOT()
```

Example:

```
SELECT First_Name
FROM customers
WHERE NOT(Profession LIKE '%Developer'
OR Yearly_Income > 75000)
```

LOGICAL OPERATORS

- AND
- OR
- NOT
- **BETWEEN**
- LIKE
- NOT LIKE
- IFNULL
- IN
- IS
- NOT IN
- ISNULL
- IS NOT NULL

- BETWEEN operator retrieves values or strings between a range as specified by the user.
- It can also extract range of dates from the given input.

Syntax:

BETWEEN Value1 AND Value2

Example:

```
SELECT First_Name
FROM customers
WHERE Income BETWEEN 50000 AND 80000
```

LOGICAL OPERATORS

- AND
- OR
- NOT
- BETWEEN
- **LIKE**
- NOT LIKE
- IFNULL
- IN
- IS
- NOT IN
- ISNULL
- IS NOT NULL

- LIKE operator is used to perform a wildcard search on tables.
- This operator uses Wildcards to extract the records matching the specified pattern.

Syntax:

LIKE Wildcard_Expression

Example:

```
SELECT First_Name
FROM customers
WHERE First_Name LIKE 'R%'
```

LOGICAL OPERATORS

- AND
- OR
- NOT
- BETWEEN
- LIKE
- **NOT LIKE**
- IFNULL
- IN
- IS
- NOT IN
- ISNULL
- IS NOT NULL

- NOT LIKE operator is used as an antonym to like operator to perform wildcard searches on tables.
- This operator uses Wildcards to extract the records matching the specified pattern.

Syntax:

NOT LIKE Wildcard_Expression

Example:

```
SELECT First_Name
FROM customers
WHERE First_Name NOT LIKE 'R%'
```

LOGICAL OPERATORS

- AND
- OR
- NOT
- BETWEEN
- LIKE
- NOT LIKE
- **IFNULL**
- IN
- IS
- NOT IN
- ISNULL
- IS NOT NULL

- IFNULL operator is used to replace the NULLs with custom values or custom text.
- Can replace only one static value in the place of nulls.

Syntax:

IFNULL(expression1, expression2)

Example:

```
SELECT CustomerKey, FirstName,
IFNULL(MiddleName, 'No Middle Name') AS Name1
FROM customers
```

LOGICAL OPERATORS

- AND
- OR
- NOT
- BETWEEN
- LIKE
- NOT LIKE
- IFNULL
- **IN**
- IS
- NOT IN
- ISNULL
- IS NOT NULL

- IN Operator checks the given expression against the Values inside the IN operator.
- If there is at least one match, then SELECT Statement returns the records

Syntax:

IN (Value1,...., ValueN)

Example:

```
SELECT First_Name
FROM customers
WHERE Income IN (80000, 90000,70000)
```

LOGICAL OPERATORS

- AND
- OR
- NOT
- BETWEEN
- LIKE
- NOT LIKE
- IFNULL
- IN
- **IS**
- NOT IN
- ISNULL
- IS NOT NULL

- IS Operator is useful to test the given expression or value against a Boolean value - True, False and Unknown.

Syntax:

IS ()

Example:

SELECT First_Name IS True, Last_Name is False
FROM customers

LOGICAL OPERATORS

- AND
- OR
- NOT
- BETWEEN
- LIKE
- NOT LIKE
- IFNULL
- IN
- IS
- **NOT IN**
- ISNULL
- IS NOT NULL

- NOT IN Operator checks the given expression against the Values inside the NOT IN operator.
- An antonym for IN operator.

Syntax:

NOT IN (Value1,...., ValueN)

Example:

```
SELECT First_Name
FROM customers
WHERE Income NOT IN (80000, 90000)
```

LOGICAL OPERATORS

- AND
- OR
- NOT
- BETWEEN
- LIKE
- NOT LIKE
- IFNULL
- IN
- IS
- NOT IN
- **ISNULL**
- IS NOT NULL

- IS NULL Operator is used to test whether the user given expression or column value is NULL or not.
- The same can be used under both Select and Where statements.

Syntax:

```
IS NULL
```

Example:

```
SELECT First_Name
FROM customers
WHERE MiddleName IS NULL
```

LOGICAL OPERATORS

- AND
- OR
- NOT
- BETWEEN
- LIKE
- NOT LIKE
- IFNULL
- IN
- IS
- NOT IN
- ISNULL
- **IS NOT NULL**

- IS NOT NULL Operator is used to test whether the user given expression or column value is NULL or not.
- The same can be used under both Select and Where statements.

Syntax:

```
IS NOT NULL
```

Example:

```
SELECT First_Name
FROM customers
WHERE MiddleName IS NOT NULL
```

AGGREGATION FUNCTIONS

- An aggregate function performs a calculation on a set of values, and returns a single value. Except for COUNT, aggregate functions ignore null values.

- Aggregate functions are often used with the GROUP BY clause of the SELECT statement.

- All aggregate functions are deterministic. In other words, aggregate functions return the same value each time that they are called, when called with a specific set of input values.

AGGREGATION FUNCTIONS

- **Sum**
- Average
- Minimum
- Maximum
- Count
- STD Population
- STD Sample
- Var Population
- Var Sample

- SUM function returns the total sum of a numeric column.

Syntax

SELECT SUM(column_name) FROM table_name

Example

SELECT SUM(salary) FROM CUSTOMERS

AGGREGATION FUNCTIONS

- Sum
- **Average**
- Minimum
- Maximum
- Count
- STD Population
- STD Sample
- Var Population
- Var Sample

- AVG function returns the average value of a numeric column.

Syntax

SELECT AVG(column_name) FROM table_name

Example

SELECT AVG(salary) FROM CUSTOMERS

AGGREGATION FUNCTIONS

- Sum
- Average
- **Minimum**
- Maximum
- Count
- STD Population
- STD Sample
- Var Population
- Var Sample

- MIN function returns the smallest value of the selected column.

Syntax

SELECT MIN(column_name) FROM table_name

Example

SELECT MIN(salary) FROM CUSTOMERS

AGGREGATION FUNCTIONS

- Sum
- Average
- Minimum
- **Maximum**
- Count
- STD Population
- STD Sample
- Var Population
- Var Sample

- MAX function returns the largest value from the selected column.

Syntax

SELECT MAX(column_name) FROM table_name

Example

SELECT MAX(salary) FROM CUSTOMERS

AGGREGATION FUNCTIONS

- Sum
- Average
- Minimum
- Maximum
- **Count**
- STD Population
- STD Sample
- Var Population
- Var Sample

- COUNT function returns the number of rows that matches a specified criteria.

Syntax

SELECT COUNT(column_name) FROM table_name

Or

SELECT COUNT(*) FROM table_name

Example

SELECT COUNT(salary) FROM CUSTOMERS

or

SELECT COUNT(*) FROM CUSTOMERS

AGGREGATION FUNCTIONS

- Sum
- Average
- Minimum
- Maximum
- Count
- **STD Population**
- STD Sample
- Var Population
- Var Sample

- The Standard deviation Population function is used to retrieve the variance in the entire dataset rather than calculating it from sample.
- The denominator here will consider the entire count of values unlike Standard deviation Sample.

Syntax

STDDEV_POP(Column_Name)

Example

SELECT STDDEV_POP(Income)

AGGREGATION FUNCTIONS

- Sum
- Average
- Minimum
- Maximum
- Count
- STD Population
- **STD Sample**
- Var Population
- Var Sample

- The Standard deviation Sample function is used to retrieve the variance in the randomly sampled data.
- Every time the sampling changes, the range gets deflected as well.

Syntax

STDDEV_SAMP(Column_Name)

Example

SELECT STDDEV_SAMP(Income)

AGGREGATION FUNCTIONS

- Sum
- Average
- Minimum
- Maximum
- Count
- STD Population
- STD Sample
- **Var Population**
- Var Sample

- The Var Population function is used to retrieve the variance in the entire dataset rather than calculating it from sample.
- The denominator here will consider the entire count of values unlike Var Sample.

Syntax

VAR_POP(Column_Name)

Example

SELECT VAR_POP(Income)

AGGREGATION FUNCTIONS

- Sum
- Average
- Minimum
- Maximum
- Count
- STD Population
- STD Sample
- Var Population
- **Var Sample**

- The Var Sample function is used to retrieve the variance in the randomly sampled data.
- Every time the sampling changes, the variance gets deflected as well.

Syntax

VAR_SAMP(Column_Name)

Example

SELECT VAR_SAMP(Income)

DATE FUNCTIONS

- In SQL, dates are complicated, since while working with database, the format of the date in table must be matched with the input date in order to insert.

- SQL server's date functions provide you set of functions that you can use to manipulate dates.

- The function are used for a wide variety of operation such as adding weeks to a date, calculating the difference between two dates, or to decompose a date into its fundamental parts.

DATE FUNCTIONS

- ADDDATE
- ADDTIME
- CURRENT_DATE
- CURRENT_TIME
- CURRENT_TIMESTAMP
- DATE
- DATE_ADD
- DATEDIFF
- DAY
- DAYNAME
- HOUR
- MAKEDATE

- ADDDATE function helps in adding date parts to a date or an expression.
- It can take only two date datatypes as arguments for performing this operation.

Syntax:

ADDDATE(Date, INTERVAL expression Unit)

Example:

SELECT ADDDATE('2019-02-28 23:59:59', INTERVAL 31 DAY)
SELECT ADDDATE('2019-02-28 23:59:59', INTERVAL 16 WEEK)
SELECT ADDDATE('2019-02-28 23:59:59', INTERVAL 18 MONTH)

DATE FUNCTIONS

- ADDDATE
- ADDTIME
- CURRENT_DATE
- CURRENT_TIME
- CURRENT_TIMES TAMP
- DATE
- DATE_ADD
- DATEDIFF
- DAY
- DAYNAME
- HOUR
- MAKEDATE

- ADDTIME function helps in adding time parts to a date or an expression.
- It can take only two time datatypes as arguments for performing this operation.

Syntax:

ADDTIME(DateTime1 or Time_Expression1, Time_Expression2)

Example:

SELECT ADDTIME('10:11:22', '12:10:12')

DATE FUNCTIONS

- ADDDATE
- ADDTIME
- **CURRENT_DATE**
- CURRENT_TIME
- CURRENT_TIMES TAMP
- DATE
- DATE_ADD
- DATEDIFF
- DAY
- DAYNAME
- HOUR
- MAKEDATE

- CURRENT_DATE function retrieves the current date without time values in it.
- This is a dynamic function and returns the values in either YYYY-MM-DD or YYYYMMDD string format.
- It takes no argument

Syntax:

CURRENT_DATE()

DATE FUNCTIONS

- ADDDATE
- ADDTIME
- CURRENT_DATE
- **CURRENT_TIME**
- CURRENT_TIMES
- TAMP
- DATE
- DATE_ADD
- DATEDIFF
- DAY
- DAYNAME
- HOUR
- MAKEDATE

- CURRENT_TIME function retrieves the current time without date values in it.
- This is a dynamic function and returns the values in either HH:MM:SS or HHMMSS string format.
- It takes no argument

Syntax:

CURRENT_TIME()

DATE FUNCTIONS

- ADDDATE
- ADDTIME
- CURRENT_DATE
- CURRENT_TIME
- CURRENT_TIMESTAMP
- DATE
- DATE_ADD
- DATEDIFF
- DAY
- DAYNAME
- HOUR
- MAKEDATE

- CURRENT_TIMESTAMP function helps in creating a timestamp for the current timeperiod.
- This is a dynamic function and returns the values in YYYY-MM-DD HH:MM:SS format.
- It takes no argument

Syntax:

CURRENT_TIMESTAMP()

DATE FUNCTIONS

- ADDDATE
- ADDTIME
- CURRENT_DATE
- CURRENT_TIME
- CURRENT_TIMES TAMP
- **DATE**
- DATE_ADD
- DATEDIFF
- DAY
- DAYNAME
- HOUR
- MAKEDATE

- DATE function extracts the date part from a given date or Date Time expression.

Syntax:

DATE(date or DateTime expression)

Example:

SELECT DATE('2017-10-25 01:05:22')

DATE FUNCTIONS

- ADDDATE
- ADDTIME
- CURRENT_DATE
- CURRENT_TIME
- CURRENT_TIMES TAMP
- DATE
- **DATE_ADD**
- DATEDIFF
- DAY
- DAYNAME
- HOUR
- MAKEDATE

- DATE_ADD function adds the user-specified intervals to a given date and returns the date or Date Time.

Syntax:

DATE_ADD(Date, INTERVAL expression Unit)

Example:

SELECT DATE_ADD('2018-12-31 23:30:15', INTERVAL 30 SECOND)

DATE FUNCTIONS

- ADDDATE
- ADDTIME
- CURRENT_DATE
- CURRENT_TIME
- CURRENT_TIMES
- TAMP
- DATE
- DATE_ADD
- **DATEDIFF**
- DAY
- DAYNAME
- HOUR
- MAKEDATE

- DATEDIFF function which is useful to find the difference between two dates and returns the number of days.
- It takes two arguments and both must be date datatype.

Syntax:

DATEDIFF(Expression1, Expression2)

Example:

SELECT DATEDIFF('2019-02-28', '2019-01-01')

DATE FUNCTIONS

- ADDDATE
- ADDTIME
- CURRENT_DATE
- CURRENT_TIME
- CURRENT_TIMES
- TAMP
- DATE
- DATE_ADD
- DATEDIFF
- **DAY**
- DAYNAME
- HOUR
- MAKEDATE

- DAY function returns the day part from any date or an expression.
- It always returns the values from 1 to 31.

Syntax:

DAY(date or expression)

Example:

SELECT DAY('2016-11-25')

DATE FUNCTIONS

- ADDDATE
- ADDTIME
- CURRENT_DATE
- CURRENT_TIME
- CURRENT_TIMES TAMP
- DATE
- DATE_ADD
- DATEDIFF
- DAY
- **DAYNAME**
- HOUR
- MAKEDATE

- DAYNAME function simply returns the full name of the day for a given date or an expression.

Syntax:

DAYNAME(date or expression)

Example:

SELECT DAYNAME('2019-03-05')

DATE FUNCTIONS

- ADDDATE
- ADDTIME
- CURRENT_DATE
- CURRENT_TIME
- CURRENT_TIMES TAMP
- DATE
- DATE_ADD
- DATEDIFF
- DAY
- DAYNAME
- **HOUR**
- MAKEDATE

- Hour function returns hour from the passed column value or an expression.
- It returns value from 0 to 23.

Syntax:

HOUR(Time or DateTime expression)

Example:

SELECT HOUR('23:12:33')

DATE FUNCTIONS

- ADDDATE
- ADDTIME
- CURRENT_DATE
- CURRENT_TIME
- CURRENT_TIMES
- TAMP
- DATE
- DATE_ADD
- DATEDIFF
- DAY
- DAYNAME
- HOUR
- **MAKEDATE**

- MAKEDATE function helps in creating date from the different time arguments given by the user.
- It takes year and day arguments.

Syntax:

MAKEDATE(year, day-of-year)

Example:

SELECT MAKEDATE(2017, 25)

DATE FUNCTIONS

- **MAKETIME**
- MINUTE
- MONTH
- MONTHNAME
- YEAR
- NOW
- SECOND
- QUARTER
- STR_TO_DATE
- TIME
- TIMEDIFF
- WEEK

- MAKETIME function helps in creating time from the different time arguments given by the user.
- It takes minutes, hours and seconds.

Syntax:

MAKETIME(hour, minutes, seconds)

Example:

SELECT MAKETIME(11, 10, 25)

DATE FUNCTIONS

- MAKE TIME
- **MINUTE**
- MONTH
- MONTHNAME
- YEAR
- NOW
- SECOND
- QUARTER
- STR_TO_DATE
- TIME
- TIMEDIFF
- WEEK

- MINUTE function returns the minute time part from any date or an expression.
- It always returns the values from 0 to 59.

Syntax:

MINUTE(Time or DateTime expression)

Example:

SELECT MINUTE('2016-12-22 12:23:45')

DATE FUNCTIONS

- MAKE TIME
- MINUTE
- **MONTH**
- MONTHNAME
- YEAR
- NOW
- SECOND
- QUARTER
- STR_TO_DATE
- TIME
- TIMEDIFF
- WEEK

- MONTH function returns the month date part from any date or an expression.
- It always returns the values from 1 to 12 for all the 12 months respectively.

Syntax:

MONTH(date or expression)

Example:

SELECT MONTH('2016-12-22')

DATE FUNCTIONS

- MAKE TIME
- MINUTE
- MONTH
- MONTHNAME
- YEAR
- NOW
- SECOND
- QUARTER
- STR_TO_DATE
- TIME
- TIMEDIFF
- WEEK

- MONTHNAME function simply returns the full name of the Month for a given date or an expression.

Syntax:

MONTHNAME(date or expression)

Example:

SELECT MONTHNAME('2016-12-22')

DATE FUNCTIONS

- MAKE TIME
- MINUTE
- MONTH
- MONTHNAME
- **YEAR**
- NOW
- SECOND
- QUARTER
- STR_TO_DATE
- TIME
- TIMEDIFF
- WEEK

- YEAR function returns the Year from the given date or expression.
- This function returns value range from 1000 to 9999.

Syntax:

YEAR(date or expression)

Example:

SELECT YEAR('2016-11-25')

DATE FUNCTIONS

- MAKE TIME
- MINUTE
- MONTH
- MONTHNAME
- YEAR
- **NOW**
- SECOND
- QUARTER
- STR_TO_DATE
- TIME
- TIMEDIFF
- WEEK

- NOW function always returns the current Date and Time.
- By default, it returns the date in YYYY-MM-DD HH:MM:SS or YYYYMMDDHHMMSS.uuu format.
- Since it does not take any argument, it can be simply called as below.

Syntax:

```
NOW()
```

DATE FUNCTIONS

- MAKE TIME
- MINUTE
- MONTH
- MONTHNAME
- YEAR
- NOW
- **SECOND**
- QUARTER
- STR_TO_DATE
- TIME
- TIMEDIFF
- WEEK

- SECOND function is used to return the Seconds value from a given time.
- This function returns integer value range from 0 to 59.

Syntax:

SECOND(Time or expression)

Example:

SELECT SECOND('2019-02-05 10:20:12')

DATE FUNCTIONS

- MAKE TIME
- MINUTE
- MONTH
- MONTHNAME
- YEAR
- NOW
- SECOND
- **QUARTER**
- STR_TO_DATE
- TIME
- TIMEDIFF
- WEEK

- QUARTER function returns the quarter of the year number from a given date.
- This function always returns integer range from 1 to 4.

Syntax:

QUARTER(date or expression)

Example:

SELECT QUARTER('2016-07-15')

DATE FUNCTIONS

- MAKE TIME
- MINUTE
- MONTH
- MONTHNAME
- YEAR
- NOW
- SECOND
- QUARTER
- **STR_TO_DATE**
- TIME
- TIMEDIFF
- WEEK

- STR_TO_DATE function takes a string as an input and converts the same into Date or DateTime or Time values based on the format given by the user.

- Both date and time parts can be printed as output.

Syntax:

STR_TO_DATE(string_expression, format)

Example:

SELECT STR_TO_DATE('12-31-2019', '%m %d %Y')

DATE FUNCTIONS

- MAKE TIME
- MINUTE
- MONTH
- MONTHNAME
- YEAR
- NOW
- SECOND
- QUARTER
- STR_TO_DATE
- **TIME**
- TIMEDIFF
- WEEK

- TIME function helps in extracting the Time part from the given time or DateTime.
- It handles all three arguments of time.

Syntax:

TIME(Time or DateTime or expression)

Example:

SELECT TIME('2016-05-19 11:14:34')

DATE FUNCTIONS

- MAKE TIME
- MINUTE
- MONTH
- MONTHNAME
- YEAR
- NOW
- SECOND
- QUARTER
- STR_TO_DATE
- TIME
- **TIMEDIFF**
- WEEK

- TIMEDIFF function is used to find the difference between two time expressions and it eventually returns the time value as output.

Syntax:

TIMEDIFF(Expression1, Expression2)

Example:

SELECT TIMEDIFF('10:12:22', '02:05:11')

DATE FUNCTIONS

- MAKE TIME
- MINUTE
- MONTH
- MONTHNAME
- YEAR
- NOW
- SECOND
- QUARTER
- STR_TO_DATE
- TIME
- TIMEDIFF
- **WEEK**

- WEEK function returns the Week number of the given date. This function accepts two arguments.

- The first one is date and the second argument decides whether the week should start with Sunday or Monday. Finally, it returns value in the range of 0 to 53 or 1 to 53.

Syntax:

Week(date, Mode)

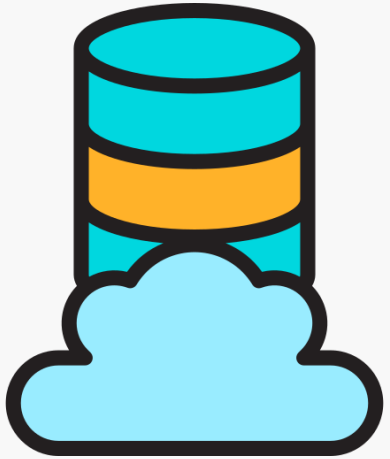
Example:

```
SELECT WEEK('2019-02-21', 0)
SELECT WEEK('2019-02-21', 1)
```

NUMERIC FUNCTIONS

- Numeric functions are primarily used for data manipulation and/or mathematical calculations.

- All numeric functions return values of data type SQL_FLOAT except for ABS, ROUND, TRUNCATE, SIGN, FLOOR, and CEILING, which return values of the same data type as the input parameters.



NUMERIC FUNCTIONS

- **CEILING**
- FLOOR
- EXP
- LOG
- LN
- MOD
- PI
- POWER
- RAND
- SIGN
- SQRT
- ROUND

- CEILING function rounds up your value irrespective of the decimals.
- Even if the decimal is .01, it rounds up the value to the next highest integer.

Syntax:

SELECT CEILING (Numeric_Expression)

Example:

SELECT CEILING('712.17')

NUMERIC FUNCTIONS

- CEILING
- **FLOOR**
- EXP
- LOG
- LN
- MOD
- PI
- POWER
- RAND
- SIGN
- SQRT
- ROUND

- FLOOR function rounds down your value irrespective of the decimals.
- Even if the decimal is .99, it rounds up the value to the next highest integer.

Syntax:

SELECT FLOOR (Numeric_Expression)

Example:

SELECT FLOOR('712.89')

NUMERIC FUNCTIONS

- CEILING
- FLOOR
- **EXP**
- LOG
- LN
- MOD
- PI
- POWER
- RAND
- SIGN
- SQRT
- ROUND

- EXP function is used to return E raised to the power of given value.
- By default, E is the base of natural logarithm.

Syntax:

SELECT EXP()

Example:

SELECT EXP(100)

NUMERIC FUNCTIONS

- CEILING
- FLOOR
- EXP
- **LOG**
- LN
- MOD
- PI
- POWER
- RAND
- SIGN
- SQRT
- ROUND

- LOG function helps in returning the natural logarithmic value for a given number.
- If we specify the base value, then it returns the log value with base.

Syntax:

SELECT LOG()
SELET LOG(Base, X)

Example:

SELECT LOG(10)
SELECT LOG(2,10)

NUMERIC FUNCTIONS

- CEILING
- FLOOR
- EXP
- LOG
- **LN**
- MOD
- PI
- POWER
- RAND
- SIGN
- SQRT
- ROUND

- LN function is useful in returning the natural logarithmic value of a given number with base E.
- LN(0) will always be null.

Syntax:

```
SELECT LN()
```

Example:

```
SELECT LN(10)
```

NUMERIC FUNCTIONS

- CEILING
- FLOOR
- EXP
- LOG
- LN
- **MOD**
- PI
- POWER
- RAND
- SIGN
- SQRT
- ROUND

- MOD function is helpful in finding the Modulus from a division.
- This function finds the remainder of different values as a result of division.

Syntax:

SELECT MOD(X, Y)

Example:

SELECT MOD(26, 5)

NUMERIC FUNCTIONS

- CEILING
- FLOOR
- EXP
- LOG
- LN
- MOD
- **PI**
- POWER
- RAND
- SIGN
- SQRT
- ROUND

- PI function is always used to return the Pi value which is 3.141593.
- Please note that it will not take any arguments.

Syntax:

PI()

NUMERIC FUNCTIONS

- CEILING
- FLOOR
- EXP
- LOG
- LN
- MOD
- PI
- **POWER**
- RAND
- SIGN
- SQRT
- ROUND

- POWER function calculates the power for a specified value in the expression.
- POW can be used as an alternative for this function.

Syntax:

SELECT POWER (Float_Expression, Value)

Example:

SELECT POWER(5, 4)

NUMERIC FUNCTIONS

- CEILING
- FLOOR
- EXP
- LOG
- LN
- MOD
- PI
- POWER
- **RAND**
- SIGN
- SQRT
- ROUND

- RAND function is used to return random numbers between 0 to 1.
- This function cannot return values greater than 1 or lesser than 0.
- It takes no argument.

Syntax:
RAND()

NUMERIC FUNCTIONS

- CEILING
- FLOOR
- EXP
- LOG
- LN
- MOD
- PI
- POWER
- RAND
- **SIGN**
- SQRT
- ROUND

- SIGN function returns only three outputs. If the number in the column is positive, it returns (1). If the same is in negative, it returns (-1).
- In other cases, it returns (0).

Syntax:

SIGN (Numeric_Expression)

Example:

SELECT SIGN(-240.75)
SELECT SIGN(240.75)
SELECT SIGN(0)

NUMERIC FUNCTIONS

- CEILING
- FLOOR
- EXP
- LOG
- LN
- MOD
- PI
- POWER
- RAND
- SIGN
- **SQRT**
- ROUND

- SQRT returns the square root for any number.
- It always works with Integers. This can return the square root for a negative integer as well.

Syntax:

SQRT(Numeric_Expression)

Example:

SELECT SQRT(265)

NUMERIC FUNCTIONS

- CEILING
- FLOOR
- EXP
- LOG
- LN
- MOD
- PI
- POWER
- RAND
- SIGN
- SQRT
- **ROUND**

- ROUND function rounds up your value to a given decimal basis the input.
- It can round the value without decimals as well.

Syntax:

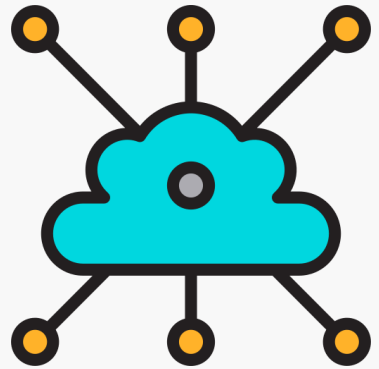
ROUND (Numeric_Expression, length)

Example:

SELECT ROUND(1252.12757, 2)

STRING FUNCTIONS

- The built in String functions make it possible for you to find and alter text values, such as VARCHAR and CHAR datatypes in the server.
- A string function is a function that takes a string value as an input regardless of the data type of the returned value
- These scalar functions perform an operation on a string input value and return a string or numeric value.



STRING FUNCTIONS

- **CONCAT**
- FIND_IN_SET
- INSTR
- LCASE
- LEFT
- LENGTH
- LOWER
- LTRIM
- REPEAT

- Concat function helps in grouping two or more strings together either using an arithmetic expression or the function itself.
- To use a delimiter between strings, make sure that it is specified inside the function.

Syntax:

SELECT CONCAT (String 1, String 2,.., String N)

Example:

SELECT CONCAT('Learn', ' MySQL Server')

STRING FUNCTIONS

- CONCAT
- **FIND_IN_SET**
- INSTR
- LCASE
- LEFT
- LENGTH
- LOWER
- LTRIM
- REPEAT

- FIND_IN_SET helps in identifying the position of a substring within a string list (N number of string separated by comma) and returns those positions.
- It takes a string and a string_in_list argument

Syntax:

FIND_IN_SET(Str, String_List)

Example:

SELECT FIND_IN_SET('c', 'a,b,c,d,e,f')

STRING FUNCTIONS

- CONCAT
- FIND_IN_SET
- **INSTR**
- LCASE
- LEFT
- LENGTH
- LOWER
- LTRIM
- REPEAT

- INSTR returns the first occurrence of a specified substring from a given string.
- It will always retrieve you the index position of the first character of the substring that we are looking for.

Syntax:

INSRT (String, Sub_String)

Example:

SELECT INSTR('We are working at ABC company', 'ABC')

STRING FUNCTIONS

- CONCAT
- FIND_IN_SET
- INSTR
- **LCASE**
- LEFT
- LENGTH
- LOWER
- LTRIM
- REPEAT

- LCASE function returns the lowercase values for a supplied string.
- This function is a synonym for LOWER function

Syntax:
LCASE(Expression)

Example:
LCASE('SQLSEVER')

STRING FUNCTIONS

- CONCAT
- FIND_IN_SET
- INSTR
- LCASE
- **LEFT**
- LENGTH
- LOWER
- LTRIM
- REPEAT

- LEFT function retrieves ‘N’ number of characters from the left side/beginning of the values.
- The arguments are the column in which you have to subset the data and the length of the characters you need.

Syntax:

LEFT (String_Expression, value)

Example:

SELECT LEFT('Learn MySQL Server', 11)

STRING FUNCTIONS

- CONCAT
- FIND_IN_SET
- INSTR
- LCASE
- LEFT
- **LENGTH**
- LOWER
- LTRIM
- REPEAT

- LENGTH function counts the number of characters that are present in the supplied column value.
- The output will always be an integer.

Syntax:

LENGTH (String_Expression)

Example:

SELECT LENGTH ('Learn MySQL Server')

STRING FUNCTIONS

- CONCAT
- FIND_IN_SET
- INSTR
- LCASE
- LEFT
- LENGTH
- **LOWER**
- LTRIM
- REPEAT

- LOWER function converts the entire string into lowercase.

Syntax:
LOWER (Expression)

Example:
Select Lower('SQLSERVER')

STRING FUNCTIONS

- CONCAT
- FIND_IN_SET
- INSTR
- LCASE
- LEFT
- LENGTH
- LOWER
- **LTRIM**
- REPEAT

- LTRIM removes extra spaces from the left side of the string.
- However, if there are any extra spaces at the end or the middle of the values, it will be unaffected.

Syntax:

LTRIM(String_Expression)

Example:

SELECT LTRIM(' Hi')

STRING FUNCTIONS

- CONCAT
- FIND_IN_SET
- INSTR
- LCASE
- LEFT
- LENGTH
- LOWER
- LTRIM
- **REPEAT**

- REPEAT function, as the name suggests, repeats a character or string ‘N’ number of times.
- The column and the number of times repetition needs to happen are asked to be supplied.

Syntax:

REPEAT (Expression, int_Expression)

Example:

SELECT REPEAT('MySQL ', 4)

STRING FUNCTIONS

- REPLACE
- REVERSE
- RIGHT
- RTRIM
- SPACE
- SUBSTR
- SUBSTR_INDEX
- TRIM
- UPPER

- REPLACE helps you update the values in a column/cell with just three arguments.
- The column, the old string and the replacement for it are supposed to be supplied as arguments.

Syntax:

REPLACE (Expression, Change_String, Replace_String)

Example:

SELECT REPLACE('123456', 34, 75)

STRING FUNCTIONS

- REPLACE
- **REVERSE**
- RIGHT
- RTRIM
- SPACE
- SUBSTR
- SUBSTR_INDEX
- TRIM
- UPPER

- REVERSE function simply reverses the order of characters of strings or numbers to give you a palindrome.
- This will retain the cases as it is. i.e., Lower case will be lower and vice versa.

Syntax:

REVERSE (String_Expression)

Example:

SELECT REVERSE('Learn MySQL Server')

STRING FUNCTIONS

- REPLACE
- REVERSE
- **RIGHT**
- RTRIM
- SPACE
- SUBSTR
- SUBSTR_INDEX
- TRIM
- UPPER

- RIGHT function retrieves ‘N’ number of characters from the right side/end of the values.
- The arguments are the column in which you have to subset the data and the length of the characters you need.

Syntax:

RIGHT (String_Expression, value)

Example:

SELECT RIGHT('Learn MySQL Server', 12)

STRING FUNCTIONS

- REPLACE
- REVERSE
- RIGHT
- **RTRIM**
- SPACE
- SUBSTR
- SUBSTR_INDEX
- TRIM
- UPPER

- RTRIM removes extra spaces from the right side of the string.
- However, if there are any extra spaces at the beginning or the middle of the values, it will be unaffected.

Syntax:

RTRIM(String_Expression)

Example:

```
SELECT RTRIM('MySQL          ')
```

STRING FUNCTIONS

- REPLACE
- REVERSE
- RIGHT
- RTRIM
- **SPACE**
- SUBSTR
- SUBSTR_INDEX
- TRIM
- UPPER

- SPACE function creates ‘N’ number of spaces between values.
- The only argument it will take is the number of spaces to create, which should always be an Integer.

Syntax:

SPACE(Integer)

Example:

SELECT CONCAT(‘First Name', SPACE(5), ‘Last Name')

STRING FUNCTIONS

- REPLACE
- REVERSE
- RIGHT
- RTRIM
- SPACE
- **SUBSTR**
- SUBSTR_INDEX
- TRIM
- UPPER

- SUBSTR helps is retrieving a particular length of characters from a string.
- The index position to subset the data can also be specified.

Syntax:

SELECT SUBSTRING (String, Position, Length)

Example:

SELECT SUBSTRING('Learn MySQL', 3, 7)

STRING FUNCTIONS

- REPLACE
- REVERSE
- RIGHT
- RTRIM
- SPACE
- SUBSTR
- **SUBSTR_INDEX**
- TRIM
- UPPER

- SUBSTR_INDEX helps to subset a substring from the occurrence of delimiter count.
- Any type of delimiter can be accessed using this function.

Syntax:

SUBSTRING_INDEX(String_Expression, Delimiter, Count)

Example:

SELECT SUBSTRING_INDEX('www.google.co.in', '.', 2)

STRING FUNCTIONS

- REPLACE
- REVERSE
- RIGHT
- RTRIM
- SPACE
- SUBSTR
- SUBSTR_INDEX
- **TRIM**
- UPPER

- TRIM function simply chops off the extra spaces that are present in your cells, be it numbers, be it strings.

Syntax:

TRIM('String_Expression')

Example:

SELECT TRIM(' Hello World ')

STRING FUNCTIONS

- REPLACE
- REVERSE
- RIGHT
- RTRIM
- SPACE
- SUBSTR
- SUBSTR_INDEX
- TRIM
- **UPPER**

- UPPER function converts the entire string to upper case

Syntax:

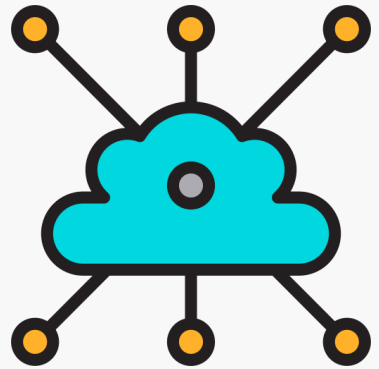
UPPER (Expression)

Example:

SELECT UPPER('sqlserver')

WINDOW FUNCTIONS

- Non-Aggregate Functions.
- For each row it forms a query and performs a calculation using rows related to that row.
- Window functions are performed on the result set after all JOIN, WHERE, GROUP BY, and HAVING clauses and before the ORDER BY, LIMIT and SELECT DISTINCT.



WINDOW FUNCTIONS

- **LEAD**
- LAG
- ROW_NUMBER
- RANK
- DENSE_RANK

- Returns the value of expr from the row that leads (follows) the current row by N rows within its partition. If there is no such row, the return value is default.
- For example, if N is 3, the return value is default for the last two rows. If N or default are missing, the defaults are 1 and NULL, respectively.
- N must be a literal nonnegative integer. If N is 0, expr is evaluated for the current row.

Syntax:

LEAD(expr [, N[, default]]) [null_treatment] over_clause

Example:

```
SELECT t, val,  LEAD(val) OVER w AS 'lEAD',  FROM series
WINDOW w AS (ORDER BY t);
```

WINDOW FUNCTIONS

- LEAD
- LAG
- ROW_NUMBER
- RANK
- DENSE_RANK

- Returns the value of expr from the row that lags (precedes) the current row by N rows within its partition.
- If there is no such row, the return value is default. For example, if N is 3, the return value is default for the first two rows.
- If N or default are missing, the defaults are 1 and NULL, respectively.

Syntax:

LAG(expr [, N[, default]]) [null_treatment] over_clause

Example:

```
SELECT t, val,  LAG(val) OVER w AS 'lag',  FROM series
WINDOW w AS (ORDER BY t);
```

WINDOW FUNCTIONS

- LEAD
- LAG
- **ROW_NUMBER**
- RANK
- DENSE_RANK

- Returns the number of the current row within its partition. Rows numbers range from 1 to the number of partition rows.
- ORDER BY affects the order in which rows are numbered. Without ORDER BY, row numbering is nondeterministic.

Syntax:

ROW_NUMBER() over_clause

Example:

```
SELECT val, ROW_NUMBER() OVER w AS 'row_number',
FROM numbers WINDOW w AS (ORDER BY val);
```

WINDOW FUNCTIONS

- LEAD
- LAG
- ROW_NUMBER
- **RANK**
- DENSE_RANK

- Returns the rank of the current row within its partition, with gaps. Peers are considered ties and receive the same rank.
- This function does not assign consecutive ranks to peer groups if groups of size greater than one exist; the result is non contiguous rank numbers.
- This function should be used with ORDER BY to sort partition rows into the desired order. Without ORDER BY, all rows are peers.

Syntax:

RANK() over_clause

Example:

```
SELECT val, RANK() OVER w AS 'Rank',
FROM numbers WINDOW w AS (ORDER BY val);
```

WINDOW FUNCTIONS

- LEAD
- LAG
- ROW_NUMBER
- RANK
- DENSE_RANK

- Returns the rank of the current row within its partition, without gaps. Peers are considered ties and receive the same rank.
- This function assigns consecutive ranks to peer groups; the result is that groups of size greater than one do not produce non contiguous rank numbers.

Syntax:

DENSE_RANK() over_clause

Example:

```
SELECT val, DENSE_RANK() OVER w AS 'Dense Rank',
FROM numbers WINDOW w AS (ORDER BY val);
```

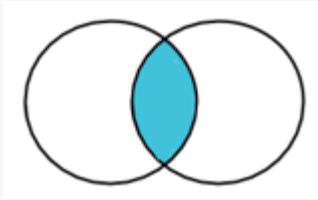
SQL JOINS

- The SQL **Joins** clause is used to combine records from two or more tables in a database.
- A JOIN is a means for combining fields from two tables by using values common to each.
- Filter and search in combination result and also tables data. Reduce duplicate records in combination result.

Types of Joins

JOINS

- Inner
- Left
- Right
- Cross



- **INNER JOIN** creates a new result table by combining column values of two tables (table1 and table2) based upon the common column that is present in both these tables.
- The query compares each row of table1 with each row of table2 to find all pairs of rows which satisfy the join-predicate.
- When the join-predicate is satisfied, column values for each matched pair of rows of A and B are combined into a result row.

Syntax

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

Example

```
SELECT ID, NAME, AMOUNT, DATE
FROM CUSTOMERS INNER JOIN ORDERS
ON CUSTOMERS.ID = ORDERS.CUSTOMER_ID
```

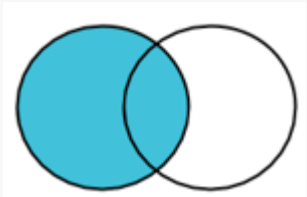
JOINS

Inner

Left

Right

Cross



- **LEFT JOIN** returns all rows from the left table, even if there are no matches in the right table.
- This means that if the ON clause matches 0 (zero) records in the right table; the join will still return a row in the result, but with NULL in each column from the right table.
- This means that a left join returns all the values from the left table, plus matched values from the right table or NULL in case of no matching join predicate.

Syntax

```
SELECT table1.column1, table2.column2...
FROM table1 LEFT JOIN table2
ON table1.common_field = table2.common_field
```

Example

```
SELECT ID, NAME, AMOUNT, DATE
FROM CUSTOMERS LEFT JOIN  ORDERS
ON CUSTOMERS.ID = ORDERS.CUSTOMER_ID
```

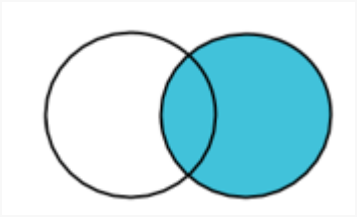
JOINS

Inner

Left

Right

Cross



- **RIGHT JOIN** returns all rows from the right table, even if there are no matches in the left table.
- This means that if the ON clause matches 0 (zero) records in the left table; the join will still return a row in the result, but with NULL in each column from the left table.
- This means that a right join returns all the values from the right table, plus matched values from the left table or NULL in case of no matching join predicate.

Syntax

```
SELECT table1.column1, table2.column2...
FROM table1 RIGHT JOIN table2
ON table1.common_field = table2.common_field
```

Example

```
SELECT ID, NAME, AMOUNT, DATE
FROM CUSTOMERS RIGHT JOIN  ORDERS
ON CUSTOMERS.ID = ORDERS.CUSTOMER_ID
```

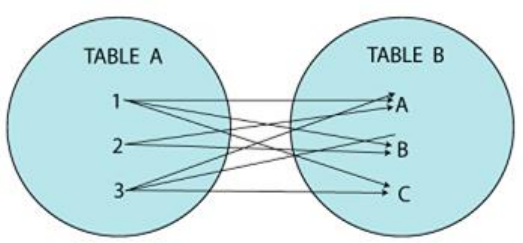
JOINS

Inner

Left

Right

Cross



- **CROSS JOIN** creates a cartesian product by fetching records from both the tables for every single line item.
- The query fetches record for each row of table1 with each row of table2 to yield a cartesian product even if it is not matching.
- The Cartesian product means Number of Rows present in Table 1 Multiplied by Number of Rows present in Table 2.

Syntax

```
SELECT table1.column1, table2.column2...
FROM table1 CROSS JOIN table2
```

Example

```
SELECT ID, NAME, AMOUNT, DATE
FROM CUSTOMERS CROSS JOIN ORDERS
```

SET OPERATORS

- The UNION clause/operator is used to combine the results of two or more SELECT statements without returning any duplicate rows.
- To use this UNION clause, each SELECT statement must have
  - ❖ The same number of columns selected
  - ❖ The same number of column expressions
  - ❖ The same data type and
  - ❖ Have them in the same order

Union

Union ALL

Syntax

```
SELECT column1, column2 FROM table1 [WHERE condition]
UNION
SELECT column1, column2 FROM table2 [WHERE condition]
```

SET OPERATORS

Union

Union ALL

- The UNION ALL operator is used to combine the results of two SELECT statements including duplicate rows.
- The same rules that apply to the UNION clause will apply to the UNION ALL operator.
- More than 2 tables of same structure can be aligned together with the help of set operators.

Syntax

```
SELECT column1, column2 FROM table1 [WHERE condition]
UNION ALL
SELECT column1, column2 FROM table2 [WHERE condition]
```

SUB - QUERIES

- A Subquery or an Inner query or a Nested query is a query within another SQL query which is embedded within the WHERE/FROM clause.
- Subqueries can be used with the SELECT, INSERT, UPDATE, and DELETE statements along with the operators like =, <, >, >=, <=, IN, BETWEEN, etc.

Syntax

SELECT [Column_Name] FROM table WHERE column_name OPERATOR

(SELECT [Column_Name] FROM table [WHERE])

Example

SELECT * FROM CUSTOMERS WHERE ID =

(SELECT ID FROM CUSTOMERS WHERE SALARY > 4500)

SUB - QUERIES

- **Where Clause**
- ANY & ALL
- FROM Statement
- Exists Method
- Correlated

- Using WHERE clause, multiple operators can be bought into considerations while writing queries on multiple levels.
- It can accommodate the membership operators like IN and NOT IN.

Example:

```
SELECT CustomerName, CheckNo, Sales FROM Payroll
WHERE Sales =
(SELECT MAX(Sales) FROM Payroll)
```

SUB - QUERIES

- Where Clause
- ANY & ALL
- FROM Statement
- Exists Method
- Correlated

- The ANY operator returns true if any of the subquery values satisfy the condition.

Example

```
SELECT * FROM Customers
WHERE age = ANY (SELECT
age
FROM  customers
where salary > 2500)
```

- The All operator returns true if all of the subquery values satisfy the condition.

Example

```
SELECT * FROM Customers
WHERE age = ALL (SELECT
age
FROM  customers
Where salary > 2500)
```

SUB - QUERIES

- Where Clause
- ANY & ALL
- **FROM Statement**
- Exists Method
- Correlated

- FROM clause can be used as a bracket to retrieve values from another table and subset to supply as input for the superset table.

Example:

```
SELECT MAX(Sales), MIN(Sales), FLOOR(AVG(Sales))
FROM
(SELECT OrdNo, SUM(OrdNo) AS Sales from table)
```

SUB - QUERIES

- Where Clause
- ANY & ALL
- FROM Statement
- **Exists Method**
- Correlated

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

Syntax

SELECT column_name FROM table_name WHERE EXISTS

(SELECT column_name FROM table_name WHERE condition)

Example

SELECT * FROM Customers WHERE age EXISTS

(SELECT age FROM customers where salary > 2500)

SUB - QUERIES

- Where Clause
- ANY & ALL
- FROM Statement
- Exists Method
- **Correlated**

- A Correlated sub-query is dependent on each and every query written within.
- It takes output from each and every query and supply the same as input to the above queries as it follows Bottom-up approach.

Example:

```
SELECT Products, MRP
FROM products P
WHERE Sales >
(SELECT AVG(Sales) FROM products WHERE
Products = P.Products)
```