SQL TRAINING

➢ Table of Contents

31. Common Table Expression (CTE).
32. Recursive CTE.
33. SQL Server Temporary Tables.
34. SQL View.
35. SQL Server Transaction.
36. Variable in SQL.
37. Error Handling in SQL Server with Try Catch.
38. SQL Server Stored Procedure.
39. SQL Server Trigger.
40. SQL Server Function

----------  ----------

1. What is SQL Server?

SQL Server is a relational database management system, or RDBMS, developed and marketed by Microsoft.

2. What is SQL?

SQL is a database computer language designed for the retrieval and management of data in relational database.

SQL stands for Structured Query Language.

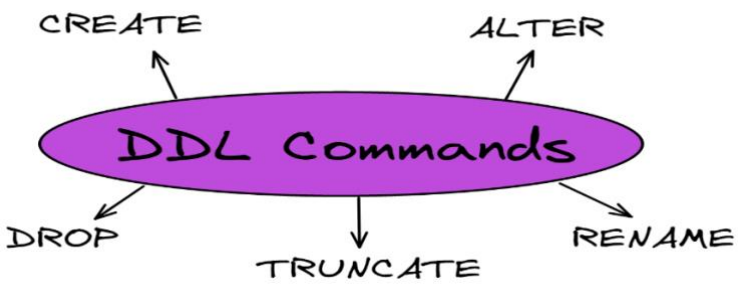
- **What Can SQL do?**
    - SQL can execute queries against a database
    - SQL can retrieve data from a database
    - SQL can insert records in a database
    - SQL can update records in a database
    - SQL can delete records from a database
    - SQL can create new databases
    - SQL can create new tables in a database
    - SQL can create stored procedures in a database
    - SQL can create views in a database
    - SQL can set permissions on tables, procedures, and views

3. SQL Commands.

The standard SQL commands to interact with relational databases are CREATE, SELECT, INSERT, UPDATE, DELETE and DROP. These commands can be classified based on their nature:

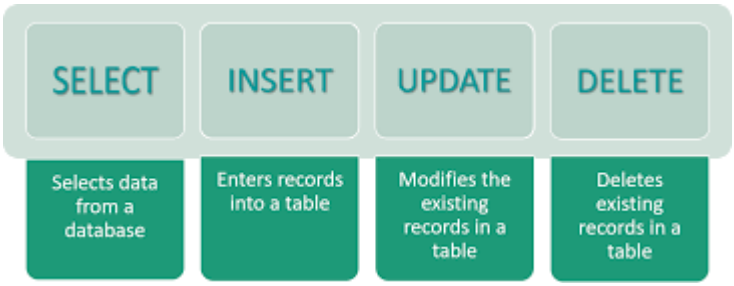
- **DDL - Data Definition Language:**

It is a syntax for creating and modifying database objects such as tables, indices, and users.



- **DML - Data Manipulation Language:**

  It is used for adding (inserting), deleting, and modifying (updating) data in a database.



- **DCL - Data Control Language:**

  It is used to control access to data stored in a database (authorization).

- **DQL - Data Query Language:**

  It is used to make various queries in information systems and databases.

  Select Statement is a DQL Command.

4. **What is Database?**

   A database is an organized collection of structured information, or data, typically stored electronically in a computer system.

5. **What is RDMS?**

   A relational database management system (RDBMS) is a program used to create, update, and manage relational databases.

- **What is Non-Relational Database?**

  A non-relational database stores data in a non-tabular form, and tends to be more flexible than the traditional, SQL-based, relational database structures. It does not follow the relational model provided by traditional relational database management systems.

6. **How to Create New Database In SQL?**

The CREATE DATABASE statement is used to create a new SQL database.

Syntax: -

```
CREATE DATABASE [Database Name] ;
```

7. **Drop Database.**

The DROP DATABASE statements are used to drop an existing

SQL database.

Syntax: -

```
DROP DATABASE [Database Name] ;
```

8. **Create Schema.**

A schema is a collection of database objects including tables, views, triggers, stored procedures, indexes, etc. A schema is associated with a username which is known as the schema owner, who is the owner of the logically related database objects.

A schema always belongs to one database. On the other hand, a database may have one or multiple schemas. For example, in our BikeStores sample database, we have two schemas: sales and production. An object within a schema is qualified using the schema_name.object_name format like sales.orders. Two tables in two schemas can share the same name so you may have hr.employees and sales.employees.

Built-in schemas in SQL Server

SQL Server provides us with some pre-defined schemas which have the same names as the built-in database users and roles, for example: dbo, guest, sys, and INFORMATION_SCHEMA.

Note that SQL Server reserve the sys and INFORMATION_SCHEMA

schemas for system objects,therefore, you cannot create or drop any objects in these schemas.

SQL Server CREATE SCHEMA **statement overview**

The CREATE SCHEMA statement allows you to create a new schema in the current database.

```
CREATE SCHEMA [schema_name]
```

9. Drop Schema.

The DROP SCHEMA statement allows you to delete a schema from a database. The following shows the syntax of the DROP SCHEMA statement:

```
DROP SCHEMA [IF EXISTS] [schema_name] ;
```

10. Create New Table.

Tables are used to store data in the database. Tables are uniquely named within a database and schema. Each table contains one or more columns. And each column has an associated data type that defines the kind of data it can store e.g., numbers, strings, or temporal data.

To create a new table, you use the CREATE TABLE statement as follows:

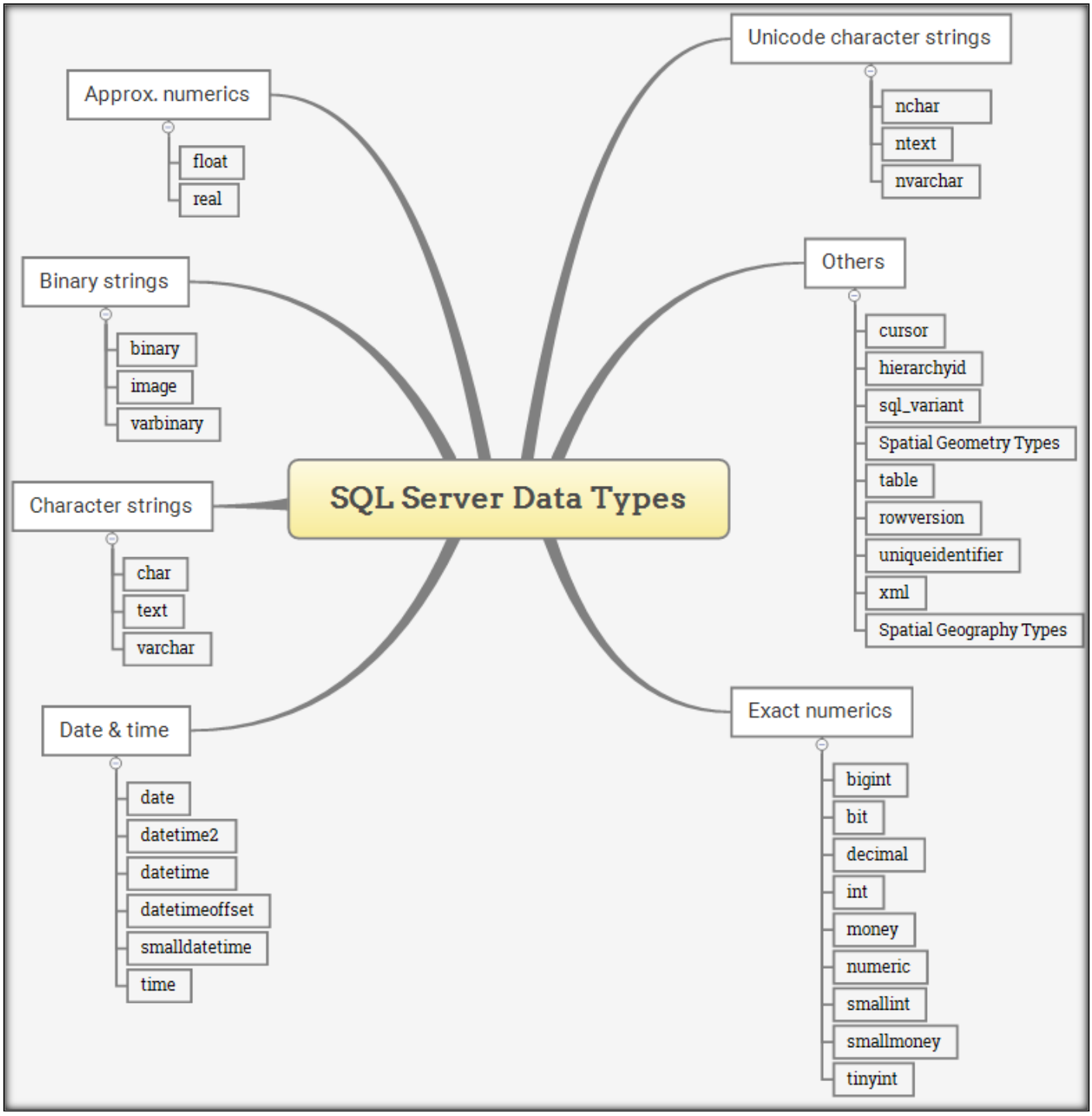
```
CREATE TABLE sales.visits (

    visit_id INT PRIMARY KEY IDENTITY (1, 1),
    first_name VARCHAR (50) NOT NULL,
    last_name VARCHAR (50) NOT NULL,
    visited_at DATETIME,
    phone VARCHAR(20),
    store_id INT NOT NULL,
    FOREIGN KEY (store_id) REFERENCES sales.stores (store_id)

);
```

11. Data Types.

In SQL Server, a column, variable, and parameter holds a value that associated with a type, or also known as a data type. A data type is an attribute that specifies the type of data that these objects can store. It can be an integer, character string, monetary, date and time, and so on.

SQL Server provides a list of data types that define all types of data that you can use e.g., defining a column or declaring a variable.

The following picture illustrates the SQL Server data types system:



Notice that SQL Server will remove **ntext**, **text**, and **image** data types in its future version. Therefore, you should avoid using these data types and use **nvarchar(max)**, **varchar(max)**, and **varbinary(max)** data types instead.

12. **Identity Column.**

To create an identity column for a table, you use the IDENTITY property as follows:

IDENTITY[(seed,increment)]

In this syntax:

- The seed is the value of the first row loaded into the table.
- The increment is the incremental value added to the identity value of the previous row.

The default value of seed and increment is 1 i.e., (1,1). It means that the first row, which was loaded into the table, will have the value of one, the second row will have the value of 2 and so on.

Suppose you want the value of the identity column of the first row is 10 and incremental value is 10, you use the following syntax:

IDENTITY (10,10)

Note that SQL Server allows you to have only one identity column per table.

The following statement creates a new table using the IDENTITY property for the personal identification number column:

```
CREATE TABLE hr.person (
        person_id INT IDENTITY(1,1) PRIMARY KEY,
        first_name VARCHAR(50) NOT NULL,
        last_name VARCHAR(50) NOT NULL,
        gender CHAR(1) NOT NULL
);
```

First, insert a new row into the person table:

```
INSERT INTO hr.person(first_name, last_name, gender)

OUTPUT inserted.person_id

VALUES('John','Doe', 'M');
```

13. Add Column.

SQL Server ALTER TABLE ADD Column :-

The following ALTER TABLE ADD statement appends a new column to a table:

```
ALTER TABLE [table_name]
ADD [column_name] [data_type] [column_constraint];
```

The following statement creates a new table named sales.quotations:

```
CREATE TABLE sales.quotations (
    quotation_no INT IDENTITY PRIMARY KEY,
    valid_from DATE NOT NULL,
    valid_to DATE NOT NULL
);
```

To add a new column named description to the sales.quotations table, you use the following statement:

```
ALTER TABLE sales.quotations
ADD description VARCHAR (255) NOT NULL;
```

14. Modify Column.

SQL Server ALTER TABLE ALTER COLUMN statement to modify a column of a table.

SQL Server allows you to perform the following changes to an existing column of a table:

- Modify the data type
- Change the size
- Add a NOT NULL constraint

Modify column's data type

To modify the data type of a column, you use the following statement:

```
ALTER TABLE [table_name ]
ALTER COLUMN column_name new_data_type (size);
```

Note :- The new data type must be compatible with the old one, otherwise, you will get a conversion error in case the column has data and it fails to convert.

See the following example.

First, create a new table with one column whose data type is INT:

```
CREATE TABLE t1 (c INT);
```

Second, insert some rows into the table:

```
INSERT INTO t1
  VALUES
    (1),
    (2),
    (3);
```

Second, modify the data type of the column from INT to VARCHAR:

```
ALTER TABLE t1 ALTER COLUMN c VARCHAR (20);
```

Third, insert a new row with a character string data:

```
INSERT INTO t1
VALUES ('Kanhaiya');
```

Fourth, modify the data type of the column from VARCHAR back to INT:

```
ALTER TABLE t1 ALTER COLUMN c INT;
```

SQL Server issued the following error:

```
Conversion failed when converting the varchar value 'Kanhaiya' to data type int.
```

Change the size of a column :-

The following statement creates a new table with one column whose data type is VARCHAR(10):

```
CREATE TABLE t2 (c VARCHAR(10));
```

Let’s insert some sample data into the t2 table:

```
INSERT INTO t2
VALUES
    ('SQL Server'),
    ('Modify'),
    ('Column')
```

You can increase the size of the column as follows:

```
ALTER TABLE t2 ALTER COLUMN c VARCHAR (50);
```

However, when you decrease the size of the column, SQL Server checks the existing data to see if it can convert data based on the new size. If the conversion fails, SQL Server terminates the statement and issues an error message.

For example, if you decrease the size of column c to 5 characters:

```
ALTER TABLE t2 ALTER COLUMN c VARCHAR (5);
```

SQL Server issued the following error:

```
String or binary data would be truncated.
```

Add a NOT NULL constraint to a nullable column :-

The following statement creates a new table with a nullable column:

```
CREATE TABLE t3 (c VARCHAR(50));
```

The following statement inserts some rows into the table:

```
INSERT INTO t3
VALUES
    ('Nullable column'),
    (NULL);
```

If you want to add the NOT NULL constraint to the column c, you must update NULL to non-null first for example:

```
UPDATE t3
SET c = ''
WHERE
   c IS NULL;
```

And then add the NOT NULL constraint:

```
ALTER TABLE t3 ALTER COLUMN c VARCHAR (20) NOT NULL;
```

15. Drop Column.

Sometimes, you need to remove one or more unused or obsolete columns from a table. To do this, you use the ALTER TABLE DROP COLUMN statement as follows:

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

In this syntax:

- First, specify the name of the table from which you want to delete the column.
- Second, specify the name of the column that you want to delete.

If the column that you want to delete has a CHECK constraint, you must delete the constraint first before removing the column. Also, SQL Server does not allow you to delete a column that has a PRIMARY KEY or a FOREIGN KEY constraint.

If you want to delete multiple columns at once, you use the following syntax:

```
ALTER TABLE table_name
DROP COLUMN column_name_1, column_name_2,...;
```

In this syntax, you specify columns that you want to drop as a list of comma-separated columns in the DROP COLUMN clause.

SQL Server ALTER TABLE DROP COLUMN examples

Let's create a new table named sales.price_lists for the demonstration.

```
CREATE TABLE sales.price_lists(
    product_id int,
    valid_from DATE,
    price DEC(10,2) NOT NULL CONSTRAINT ck_positive_price CHECK(price >= 0),
    discount DEC(10,2) NOT NULL,
    surcharge DEC(10,2) NOT NULL,
    note VARCHAR(255),
    PRIMARY KEY(product_id, valid_from)
);
```

The following statement drops the note column from the price_lists table:

```
ALTER TABLE sales.price_lists
DROP COLUMN note;
```

The price column has a CHECK constraint, therefore, you cannot delete it. If you try to execute the following statement, you will get an error:

```
ALTER TABLE sales.price_lists
DROP COLUMN price;
```

Here is the error message:

```
The object 'ck_positive_price' is dependent on column 'price'.
```

To drop the price column, first, delete its CHECK constraint:

```
ALTER TABLE sales.price_lists
DROP CONSTRAINT ck_positive_price;
```

And then, delete the price column:

```
ALTER TABLE sales.price_lists
DROP COLUMN price;
```

The following example deletes two columns discount and surcharge at once:

```
ALTER TABLE sales.price_lists
DROP COLUMN discount, surcharge;
```

16. Computed Columns.

Introduction to SQL Server computed columns

Let’s create a new table named persons for the demonstrations:

```
CREATE TABLE persons
(
    person_id  INT PRIMARY KEY IDENTITY,
    first_name NVARCHAR(100) NOT NULL,
    last_name  NVARCHAR(100) NOT NULL,
    dob        DATE
);
```

And insert two rows into the the persons table:

```
INSERT INTO
    persons(first_name, last_name, dob)
VALUES
    ('John','Doe','1990-05-01'),
    ('Jane','Doe','1995-03-01');
```

To query the full names of people in the persons table, you normally use the CONCAT() function or the + operator as follows:

```
SELECT
    person_id,
    first_name + ' ' + last_name AS full_name,
    dob
FROM
    persons
ORDER BY
    full_name;
```

OUTPUT:-

person_id	full_name	dob
2	Jane Doe	1995-03-01
1	John Doe	1990-05-01

Adding the full_name expression first_name + ' ' + last_name in every query is not convenient.

Fortunately, SQL Server provides us with a feature called computed columns that allows you to add a new column to a table with the value derived from the values of other columns in the *same* table.

For example, you can add the full_name column to the persons table by using the ALTER TABLE ADD column as follows:

```
ALTER TABLE persons
ADD full_name AS (first_name + ' ' + last_name);
```

Every time you query data from the persons table, SQL Server computes the value for the full_name column based on the expression first_name + ' ' + last_name and returns the result.

Here is the new query, which is more compact:

```
SELECT
        person_id,
        full_name,
        dob
FROM
        persons
ORDER BY
        full_name;
```

17. Rename Table.

SQL Rename table using Transact SQL

SQL Server does not have any statement that directly renames a table. However, it does provide you with a stored procedure named **sp_rename** that allows you to change the name of a table.

The following shows the syntax of using the **sp_rename** stored procedure for changing the name of a table:

```
EXEC sp_rename 'old_table_name', 'new_table_name'
```

Note that both the old and new name of the table whose name is changed must be enclosed in single quotations.

18. Drop Table.

SQL Server DROP TABLE

Sometimes, you want to remove a table that is no longer in use. To do this, you use the following **DROP TABLE** statement:

```
DROP TABLE [IF EXISTS]  [database_name.][schema_name.]table_name;
```

In this syntax:

- First, specify the name of the table to be removed.
- Second, specify the name of the database in which the table was created and the name of the schema to which the table belongs. The database name is optional. If you skip it, the DROP TABLE statement will drop the table in the currently connected database.
- Third, use **IF EXISTS** clause to remove the table only if it exists.
- The IF EXISTS clause has been supported since SQL Server 2016 13.x. If you remove a table that does not exist, you will get an error. The IF EXISTS clause conditionally removes the table if it already exists.

When SQL Server drops a table, it also deletes all data, triggers, constraints, permissions of that table. Moreover, SQL Server does not explicitly drop the views and stored procedures that reference the dropped table. Therefore, to explicitly drop these dependent objects, you must use the **DROP VIEW** and **DROP PROCEDURE** statement.

SQL Server allows you to remove multiple tables at once using a single **DROP TABLE** statement as follows:

```
DROP TABLE [database_name].[schema_name].[table_name_1]
```

19. Insert Data into Table.

- **The SQL INSERT INTO Statement :**

  The Insert into statement is used to insert new records in a table.

  Insert Into Syntax :

  It is possible to write the INSERT INTO statement in two ways:

  1. Specify both the column names and the values to be inserted:

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

  2. 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.

  Here the INSERT INTO syntax would be as follows:

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

  Example:- Table Name Customer

Name	Address	Mobile
Kanhaiya	**Patna**	**234566**
Ravali	**Delhi**	**345677**

Insert New Row into table :-

```
Insert into Customer (Name, Address, Mobile)
Values (‘Tarun’, ‘Rajsthan’, 234567)
```

OR

```
Insert into Customer Values(‘Tarun’,’Rajsthan’,234567)
```

Name	Address	Mobile
Kanhaiya	Patna	234566
Ravali	Delhi	345677
Tarun	Rajsthan	234567

Insert Data Only in Specified Columns :

It is also possible to only insert data in specific columns.

```
INSERT INTO Customer ( Name, Address)
Values ('Abhishek', 'Delhi')
```

Name	Address	Mobile
Kanhaiya	Patna	234566
Ravali	Delhi	345677
Tarun	Rajsthan	234567
Abhishek	Delhi	NULL

Create Copy of Any Table With all Data :

```
Select * into Newtable_name from Oldtable_name Where 1=1
```

Create Empty Table Copy of Any Table:

```
Select * into Newtable_name from Oldtable_name Where 1=0
```

20. Update Table.

The Update Statement is used to modify the existing records in a table.

UPDATE Syntax

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

Example:- Table Name Customer (Update Kanhaiya Mobile Number 234566 to 111111)

Name	Address	Mobile
Kanhaiya	Patna	234566
Ravali	Delhi	345677

```
UPDATE CUSTOMER
SET Mobile= 111111
Where Name='Kanhaiya'
```

Name	Address	Mobile
Kanhaiya	Patna	111111
Ravali	Delhi	345677

Practice Update Statement :-

First, create a new table named taxes for demonstration.

```
CREATE TABLE sales.taxes (
	tax_id INT PRIMARY KEY IDENTITY (1, 1),
	state VARCHAR (50) NOT NULL UNIQUE,
	state_tax_rate DEC (3, 2),
	avg_local_tax_rate DEC (3, 2),
	combined_rate AS state_tax_rate + avg_local_tax_rate,
	max_local_tax_rate DEC (3, 2),
	updated_at datetime
);
```

Second, execute the following statements to insert data into the taxes table:

```
INSERT INTO sales.taxes(state,state_tax_rate,avg_local_tax_rate,max_local_tax_rate)
VALUES('Alabama',0.04,0.05,0.07);

INSERT INTO sales.taxes(state,state_tax_rate,avg_local_tax_rate,max_local_tax_rate)
VALUES('Alaska',0,0.01,0.07);

INSERT INTO sales.taxes(state,state_tax_rate,avg_local_tax_rate,max_local_tax_rate)
VALUES('Arizona',0.05,0.02,0.05);

INSERT INTO sales.taxes(state,state_tax_rate,avg_local_tax_rate,max_local_tax_rate)
VALUES('Arkansas',0.06,0.02,0.05);

INSERT INTO sales.taxes(state,state_tax_rate,avg_local_tax_rate,max_local_tax_rate)
VALUES('California',0.07,0.01,0.02);

INSERT INTO sales.taxes(state,state_tax_rate,avg_local_tax_rate,max_local_tax_rate)
VALUES('Colorado',0.02,0.04,0.08);

INSERT INTO sales.taxes(state,state_tax_rate,avg_local_tax_rate,max_local_tax_rate)
VALUES('Connecticut',0.06,0,0);

INSERT INTO sales.taxes(state,state_tax_rate,avg_local_tax_rate,max_local_tax_rate) VALUES('Delaware',0,0,0);

INSERT INTO sales.taxes(state,state_tax_rate,avg_local_tax_rate,max_local_tax_rate)
VALUES('Florida',0.06,0,0.02);

INSERT INTO sales.taxes(state,state_tax_rate,avg_local_tax_rate,max_local_tax_rate)
VALUES('Georgia',0.04,0.03,0.04);

INSERT INTO sales.taxes(state,state_tax_rate,avg_local_tax_rate,max_local_tax_rate) VALUES('Hawaii',0.04,0,0);

INSERT INTO sales.taxes(state,state_tax_rate,avg_local_tax_rate,max_local_tax_rate)
VALUES('Idaho',0.06,0,0.03);

INSERT INTO sales.taxes(state,state_tax_rate,avg_local_tax_rate,max_local_tax_rate)
VALUES('Illinois',0.06,0.02,0.04);

INSERT INTO sales.taxes(state,state_tax_rate,avg_local_tax_rate,max_local_tax_rate) VALUES('Indiana',0.07,0,0);
```

```
INSERT INTO sales.taxes(state,state_tax_rate,avg_local_tax_rate,max_local_tax_rate)
VALUES('Iowa',0.06,0,0.01);
INSERT INTO sales.taxes(state,state_tax_rate,avg_local_tax_rate,max_local_tax_rate)
VALUES('Kansas',0.06,0.02,0.04);

INSERT INTO sales.taxes(state,state_tax_rate,avg_local_tax_rate,max_local_tax_rate)
VALUES('Kentucky',0.06,0,0);

INSERT INTO sales.taxes(state,state_tax_rate,avg_local_tax_rate,max_local_tax_rate)
VALUES('Louisiana',0.05,0.04,0.07);

INSERT INTO sales.taxes(state,state_tax_rate,avg_local_tax_rate,max_local_tax_rate) VALUES('Maine',0.05,0,0);

INSERT INTO sales.taxes(state,state_tax_rate,avg_local_tax_rate,max_local_tax_rate)
VALUES('Maryland',0.06,0,0);

INSERT INTO sales.taxes(state,state_tax_rate,avg_local_tax_rate,max_local_tax_rate)
VALUES('Massachusetts',0.06,0,0);

INSERT INTO sales.taxes(state,state_tax_rate,avg_local_tax_rate,max_local_tax_rate)
VALUES('Michigan',0.06,0,0);

INSERT INTO sales.taxes(state,state_tax_rate,avg_local_tax_rate,max_local_tax_rate)
VALUES('Minnesota',0.06,0,0.01);

INSERT INTO sales.taxes(state,state_tax_rate,avg_local_tax_rate,max_local_tax_rate)
VALUES('Mississippi',0.07,0,0.01);

INSERT INTO sales.taxes(state,state_tax_rate,avg_local_tax_rate,max_local_tax_rate)
VALUES('Missouri',0.04,0.03,0.05);

INSERT INTO sales.taxes(state,state_tax_rate,avg_local_tax_rate,max_local_tax_rate) VALUES('Montana',0,0,0);

INSERT INTO sales.taxes(state,state_tax_rate,avg_local_tax_rate,max_local_tax_rate)
VALUES('Nebraska',0.05,0.01,0.02);

INSERT INTO sales.taxes(state,state_tax_rate,avg_local_tax_rate,max_local_tax_rate)
VALUES('Nevada',0.06,0.01,0.01);

INSERT INTO sales.taxes(state,state_tax_rate,avg_local_tax_rate,max_local_tax_rate) VALUES('New
Hampshire',0,0,0);

INSERT INTO sales.taxes(state,state_tax_rate,avg_local_tax_rate,max_local_tax_rate) VALUES('New
Jersey',0.06,0,0);

INSERT INTO sales.taxes(state,state_tax_rate,avg_local_tax_rate,max_local_tax_rate) VALUES('New
Mexico',0.05,0.02,0.03);

INSERT INTO sales.taxes(state,state_tax_rate,avg_local_tax_rate,max_local_tax_rate) VALUES('New
York',0.04,0.04,0.04);

INSERT INTO sales.taxes(state,state_tax_rate,avg_local_tax_rate,max_local_tax_rate) VALUES('North
Carolina',0.04,0.02,0.02);

INSERT INTO sales.taxes(state,state_tax_rate,avg_local_tax_rate,max_local_tax_rate) VALUES('North
Dakota',0.05,0.01,0.03);

INSERT INTO sales.taxes(state,state_tax_rate,avg_local_tax_rate,max_local_tax_rate)
VALUES('Ohio',0.05,0.01,0.02);
```

```
INSERT INTO sales.taxes(state,state_tax_rate,avg_local_tax_rate,max_local_tax_rate) VALUES('Oregon',0,0,0);
```

Practice Questions

1) Update a single column in all rows example

The following statement updates a single column for all rows in the taxes table:

```
UPDATE sales.taxes
SET updated_at = GETDATE();
```

In this example, the statement changed the values in the updated_at column to the system date time returned by the **GETDATE()** function.

2) Update multiple columns example

The following statement increases the max local tax rate by 2% and the average local tax rate by 1% for the states that have the max local tax rate 1%.

```
UPDATE sales.taxes
SET max_local_tax_rate += 0.02,
    avg_local_tax_rate += 0.01
WHERE
    max_local_tax_rate = 0.01;
```

21. Truncate Table.

Sometimes, you want to delete all rows from a table. In this case, you typically use the DELETE statement without a WHERE clause.

Eg:- To delete all rows from the customer_groups table, you use the DELETE statement as follows:

```
DELETE FROM sales.customer_groups;
```

Besides the DELETE FROM statement, you can use the TRUNCATE TABLE statement to delete all rows from a table.

The following illustrates the syntax of the TRUNCATE TABLE statement:

```
TRUNCATE TABLE [database_name.][schema_name.]table_name;
```

```
TRUNCATE TABLE sales.customer_groups;
```

22. Delete Row from Table.

To remove one or more rows from a table completely, you use the DELETE statement. The following illustrates its syntax:

```
DELETE [ TOP ( expression ) [ PERCENT ] ]
FROM table_name
[WHERE search_condition];
```

Some Specific Method for Delete rows from table :-

1) Delete the number of random rows example

The following DELETE statement removes 21 random rows from the product_history table:

```
DELETE TOP (21)
FROM production.product_history;
```

2) Delete the percent of random rows example

The following DELETE statement removes 5 percent of random rows from the product_history table:

```
DELETE TOP (5) PERCENT
FROM production.product_history;
```

3) Delete some rows with a condition example

The following DELETE statement removes all products whose model year is 2017:

```
DELETE  FROM
        production.product_history
WHERE
        model_year = 2017;
```

4) Delete all rows from a table example

The following DELETE statement removes all rows from the product_history table:

```
DELETE FROM  production.product_history;
```

23. Synonym.

In SQL Server, a synonym is an alias or alternative name for a database object such as a table, view, stored procedure, user-defined function, and sequence. A synonym provides you with many benefits if you use it properly.

SQL Server CREATE SYNONYM statement syntax

To create a synonym, you use the CREATE SYNONYM statement as follows:

A) Creating a synonym within the same database example

The following example uses the CREATE SYNONYM statement to create a synonym for the sales.orders table:

```
CREATE SYNONYM orders FOR sales.orders;
```

B) Creating a synonym for a table in another database

```
CREATE SYNONYM suppliers FOR
test.purchasing.suppliers;
```

Listing all synonyms of a database:-

A) Listing synonyms using Transact-SQL command

To list all synonyms of the current database, you query from the sys.synonyms catalog view as shown in the following query:

```
SELECT
        name,
       base_object_name,
        type
FROM
       sys.synonyms
ORDER BY
       name;
```

Removing a synonym

To remove a synonym, you use the DROP SYNONYM statement with the following syntax:

```
DROP SYNONYM [ IF EXISTS ] [schema.] synonym_name
```

When to use synonyms :-

You will find some situations which you can effectively use synonyms.

1) Simplify object names

If you refer to an object from another database (even from a remote server), you can create a synonym in your database and reference to this object as it is in your database.

2) Enable seamless object name changes

When you want to rename a table or any other object such as a view, stored procedure, user-defined function, or a sequence, the existing database objects that reference to this table need to be manually modified to reflect the new name. In addition, all current applications that use this table need to be changed and possibly to be recompiled. To avoid all of these hard work, you can rename the table and create a synonym for it to keep existing applications function properly.

Benefits of synonyms

Synonym provides the following benefit if you use them properly:

- Provide a layer of abstraction over the base objects.
- Shorten the lengthy name e.g., a very_long_database_name.with_schema.and_object_name with a simplified alias.
- Allow backward compatibility for the existing applications when you rename database objects such as tables, views, stored procedures, user-defined functions, and sequences.

24. SQL Constraints :-

Constraints are the rules enforced on data columns on 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. Constraints could be column level or table level.

Column level constraints are applied only to one column, whereas table level constraints are applied to the whole table.

Following are commonly used constraints available in SQL:

NOT NULL Constraint: Ensures that a column cannot have NULL value.

DEFAULT Constraint: Provides a default value for a column when none is specified.

UNIQUE Constraint: Ensures that all values in a column are different.

PRIMARY Key: Uniquely identified each rows/records in a database table.

FOREIGN Key: Uniquely identified a rows/records in any another database table.

CHECK Constraint: The CHECK constraint ensures that all values in a column satisfy certain conditions.

INDEX: Use to create and retrieve data from the database very quickly.

NOT NULL Constraint :

By default, a column can hold NULL values.

If you do not want a column to have a NULL value, then you need to define such constraint on this column specifying that NULL is now not allowed for that column.

A NULL is not the same as no data, rather, it represents unknown data.

Example: For example, the following SQL creates a new table called

CUSTOMERS and adds five columns, three of which, ID and NAME and AGE, specify not to accept NULLs:

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

DEFAULT Constraint:

The DEFAULT constraint provides a default value to a column when the INSERT INTO statement does not provide a specific value.

Example : For example, the following SQL creates a new table called CUSTOMERS and adds five columns. Here, SALARY column is set to 5000.00 by default, so in case INSERT INTO statement does not provide a value for this column. then by default this column would be set to 5000.00.

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

UNIQUE Constraint:

The UNIQUE Constraint prevents two records from having identical values in a particular column.

In the CUSTOMERS table, for example, you might want to prevent two or more people from having identical age.

Example: For example, the following SQL creates a new table called CUSTOMERS and adds five columns.

Here, Mobile_Number column is set to UNIQUE, so that you can not have two records with same Mobile_Number :

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

PRIMARY Key: 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, which may consist of single or multiple fields.

When multiple fields are used as a primary key, they are called a composite key.

If a table has a primary key defined on any field(s), then you can not have two records having the same value of that field(s).

Note: You would use these concepts while creating database tables.

Create Primary Key: Here is the syntax to define ID attribute as a primary key in a CUSTOMERS table.

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

FOREIGN Key:

A foreign key is a key used to link two tables together.

This is sometimes called a referencing key. Foreign Key is a column or a combination of columns whose values match a Primary Key in a different table.

The relationship between 2 tables matches the Primary Key in one of the tables with a Foreign Key in the second table. If a table has a primary key defined on any field(s), then you can not have two records having the same value of that field(s).

Example: Consider the structure of the two tables as follows:

CUSTOMERS table:

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

ORDERS table:

```
CREATE TABLE ORDERS ( ID INT NOT NULL,
        DATE DATETIME,
        CUSTOMER_ID INT references CUSTOMERS(ID),
        AMOUNT double,
        PRIMARY KEY (ID) );
```

If ORDERS table has already been created, and the foreign key has not yet been set, use the syntax for specifying a foreign key by altering a table.

```
ALTER TABLE ORDERS ADD FOREIGN KEY (Customer_ID) REFERENCES
CUSTOMERS (ID);
```

CHECK Constraint:

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 isn't entered into the table.

Example: For example, the following SQL creates a new table called CUSTOMERS and adds five columns. Here, we add a CHECK with AGE column, so that you can not have any CUSTOMER below 18 years:

```
CREATE TABLE CUSTOMERS( 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) );
```

INDEX:

The INDEX is used to create and retrieve data from the database very quickly.

Index can be created by using single or group of columns in a table.

When index is created, it is assigned a ROWID for each row before it sorts out the data.

Proper indexes are good for performance in large databases, but you need to be careful while creating index.

Selection of fields depends on what you are using in your SQL queries.

Example: For example, the following SQL creates a new table called CUSTOMERS and adds five columns:

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

Now, you can create index on single or multiple columns using the following syntax:

```
CREATE INDEX index_name ON table_name ( column1, column2.....);
```

25 SQL Syntax :-

SQL is followed by unique set of rules and guidelines called Syntax. This tutorial gives you a quick start with SQL by listing all the basic SQL Syntax:

Note :- Important point to be noted is that SQL is case insensitive, which means 'SELECT' and 'select' have same meaning in SQL statements.

SQL SELECT Statement:

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

SQL DISTINCT Clause:

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

SQL WHERE Clause:

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

SQL AND/OR Clause:

SELECT column1, column2....columnN FROM table_name WHERE CONDITION-1

{AND|OR} CONDITION-2;

SQL IN Clause:

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

SQL BETWEEN Clause:

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

SQL LIKE Clause:

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

SQL ORDER BY Clause:

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

SQL GROUP BY Clause:

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

SQL COUNT Clause:

SELECT COUNT(column_name) FROM table_name WHERE CONDITION;

SQL HAVING Clause:

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

SQL Operators

What is an Operator in SQL?

An operator is a reserved word or a character used primarily in an SQL statement's WHERE clause to perform operation(s), such as comparisons and arithmetic operations. Operators are used to specify conditions in an SQL statement and to serve as conjunctions for multiple conditions in a statement.

1. Arithmetic operators
2. Comparison operators
3. Logical operators

Operators used to negate conditions.

SQL Arithmetic Operators:

Assume variable a holds 10 and variable b holds 20, then:

Operator	Description	Example
+	Addition - Adds values on either side of the operator	a + b will give 30
-	Subtraction - Subtracts right hand operand from left hand operand	a - b will give -10
*	Multiplication - Multiplies values on either side of the operator	a * b will give 200
/	Division - Divides left hand operand by right hand operand	b / a will give 2
%	Modulus - Divides left hand operand by right hand operand and returns remainder	b % a will give 0

SQL Comparison Operators:

Assume variable a holds 10 and variable b holds 20, then:

Operator	Description	Example
=	Checks if the values of two operands are equal or not, if yes then condition becomes true.	(a = b) is not true.
!=	Checks if the values of two operands are equal or not, if values are not equal then condition becomes true.	(a != b) is true.
<>	Checks if the values of two operands are equal or not, if values are not equal then condition becomes true.	(a <> b) is true.
>	Checks if the value of left operand is greater than the value of right operand, if yes then condition becomes true.	(a > b) is not true.
<	Checks if the value of left operand is less than the value of right operand, if yes then condition becomes true.	(a < b) is true.
>=	Checks if the value of left operand is greater than or equal to the value of right operand, if yes then condition becomes true.	(a >= b) is not true.
<=	Checks if the value of left operand is less than or equal to the value of right operand, if yes then condition becomes true.	(a <= b) is true.
!<	Checks if the value of left operand is not less than the value of right operand, if yes then condition becomes true.	(a !< b) is false.
!>	Checks if the value of left operand is not greater than the value of right operand, if yes then condition becomes true.	(a !> b) is true.

SQL Logical Operators:

Here is a list of all the logical operators available in SQL.

Operator	Description
ALL	The ALL operator is used to compare a value to all values in another value set.
AND	The AND operator allows the existence of multiple conditions in an SQL statement's WHERE clause.
ANY	The ANY operator is used to compare a value to any applicable value in the list according to the condition.
BETWEEN	The BETWEEN operator is used to search for values that are within a set of values, given the minimum value and the maximum value.
EXISTS	The EXISTS operator is used to search for the presence of a row in a specified table that meets certain criteria.
IN	The IN operator is used to compare a value to a list of literal values that have been specified.
LIKE	The LIKE operator is used to compare a value to similar values using wildcard operators.
NOT	The NOT operator reverses the meaning of the logical operator with which it is used. Eg: NOT EXISTS, NOT BETWEEN, NOT IN, etc. **This is a negate operator.**
OR	The OR operator is used to combine multiple conditions in an SQL statement's WHERE clause.
IS NULL	The NULL operator is used to compare a value with a NULL value.
UNIQUE	The UNIQUE operator searches every row of a specified table for uniqueness (no duplicates).

SQL LIKE Clause

The SQL LIKE clause is used to compare a value to similar values using wildcard operators.

There are two wildcards used in conjunction with the LIKE operator:

- The percent sign (%)
- The underscore (_)

The percent sign represents zero, one, or multiple characters. The underscore represents a single number or character.

Example:

Here are number of examples showing WHERE part having different LIKE clause with '%' and '_' operators:

Statement	Description
WHERE SALARY LIKE '200%'	Finds any values that start with 200
WHERE SALARY LIKE '%200%'	Finds any values that have 200 in any position
WHERE SALARY LIKE '_00%'	Finds any values that have 00 in the second and third positions
WHERE SALARY LIKE '2 % %'	Finds any values that start with 2 and are at least 3 characters in length
WHERE SALARY LIKE '%2'	Finds any values that end with 2
WHERE SALARY LIKE '_2%3'	Finds any values that have a 2 in the second position and end with a 3
WHERE SALARY LIKE '2 3'	Finds any values in a five-digit number that start with 2 and end with 3

The SQL GROUP BY Statement :-

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.

GROUP BY Syntax:

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

SOME SQL AGGREGATE FUNCTIONS :-

COUNT counts how many rows are in a particular column.

SUM adds together all the values in a particular column.

MIN and **MAX** return the lowest and highest values in a particular column, respectively.

AVG calculates the average of a group of selected values.

Difference between Where and Having Clause in SQL :-

1. **WHERE Clause:**
   WHERE Clause is used to filter the records from the table or used while joining more than one table.Only those records will be extracted who are satisfying the specified condition in WHERE clause. It can be used with SELECT, UPDATE, DELETE statements.

   ```
   SELECT Name, Age FROM Student WHERE Age >=18
   ```

   2. HAVING Clause:

   HAVING Clause is used to filter the records from the groups based on the given condition in the HAVING Clause. Those groups who will satisfy the given condition will appear in the final result. HAVING Clause can only be used with SELECT statement.

   Let us consider Student table mentioned above and apply having clause on it:

```
SELECT Age, COUNT(Roll_No) AS No_of_Students
FROM Student GROUP BY Age
HAVING COUNT(Roll_No) > 1
```

27. 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.

Consider the following two tables,

(a)CUSTOMERS table is as follows:

ID	NAME	AGE	ADDRESS	SALARY
1	Ramesh	32	Ahmedabad	2000.00
2	Khilan	25	Delhi	1500.00
3	kaushik	23	Kota	2000.00
4	Chaitali	25	Mumbai	6500.00
5	Hardik	27	Bhopal	8500.00
6	Komal	22	MP	4500.00
7	Muffy	24	Indore	10000.00

(b) Another table is ORDERS as follows:

OID	DATE	CUSTOMER_ID	AMOUNT
102	2009-10-08 00:00:00	3	3000
100	2009-10-08 00:00:00	3	1500
101	2009-11-20 00:00:00	2	1560
103	2008-05-20 00:00:00	4	2060

SQL Join Types:

There are different types of joins available in SQL:

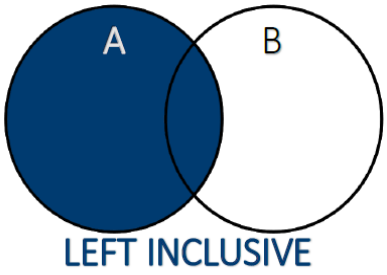
INNER JOIN: returns rows when there is a match in both tables.

LEFT JOIN: returns all rows from the left table, even if there are no matches in the right table.

RIGHT JOIN: returns all rows from the right table, even if there are no matches in the left table.

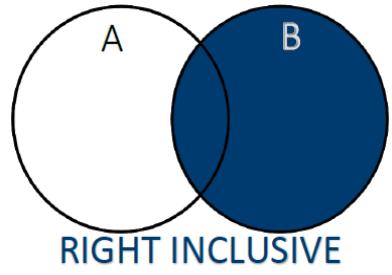
FULL JOIN: returns rows when there is a match in one of the tables.

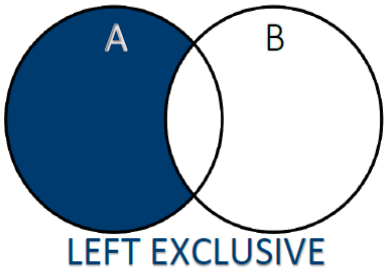
SELF JOIN: is used to join a table to itself as if the table were two tables, temporarily renaming at least one table in the SQL statement.

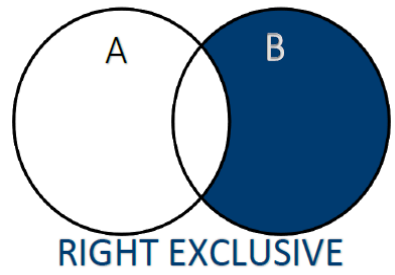


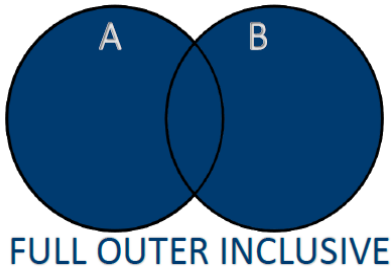
SQL JOINS

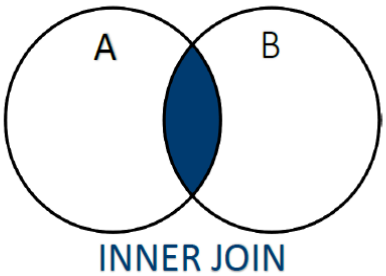
LEFT INCLUSIVE	RIGHT INCLUSIVE
SELECT *[Select List]* FROM TableA A LEFT OUTER JOIN TableB B ON A.Key= B.Key	SELECT *[Select List]* FROM TableA A RIGHT OUTER JOIN TableB B ON A.Key= B.Key
LEFT EXCLUSIVE	**RIGHT EXCLUSIVE**
SELECT *[Select List]* FROM TableA A LEFT OUTER JOIN TableB B ON A.Key= B.Key WHERE B.Key IS NULL	SELECT *[Select List]* FROM TableA A LEFT OUTER JOIN TableB B ON A.Key= B.Key WHERE A.Key IS NULL
FULL OUTER INCLUSIVE	**FULL OUTER EXCLUSIVE**
SELECT *[Select List]* FROM TableA A FULL OUTER JOIN TableB B ON A.Key = B.Key	SELECT *[Select List]* FROM TableA A FULL OUTER JOIN TableB B ON A.Key = B.Key WHERE A.Key IS NULL OR B.Key IS NULL
INNER JOIN	
SELECT *[Select List]* FROM TableA A INNER JOIN TableB B ON A.Key = B.Key	

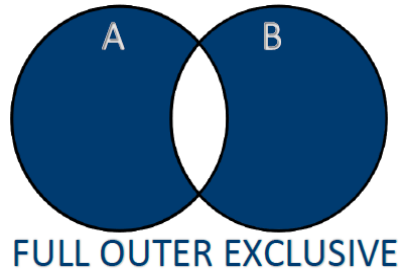












INNER JOIN

The most frequently used and important of the joins is the INNER JOIN. They are also referred to as an

EQUI JOIN.

The INNER JOIN creates a new result table by combining column values of two tables (table1 and table2) based upon the join-predicate.

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

RESULT :-

```
+----+----------+--------+---------------------+
| ID | NAME     | AMOUNT | DATE                |
+----+----------+--------+---------------------+
|  3 | kaushik  |   3000 | 2009-10-08 00:00:00 |
|  3 | kaushik  |   1500 | 2009-10-08 00:00:00 |
|  2 | Khilan   |   1560 | 2009-11-20 00:00:00 |
|  4 | Chaitali |   2060 | 2008-05-20 00:00:00 |
+----+----------+--------+---------------------+
```

LEFT JOIN :-

The SQL 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 right table, the join will still return a row in the result, but with NULL in each column from 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 ID, NAME, AMOUNT, DATE
FROM CUSTOMERS
LEFT JOIN ORDERS
ON CUSTOMERS.ID = ORDERS.CUSTOMER_ID;
```

RESULT :

ID	NAME	AMOUNT	DATE
1	Ramesh	NULL	NULL
2	Khilan	1560	2009-11-20 00:00:00
3	kaushik	3000	2009-10-08 00:00:00
3	kaushik	1500	2009-10-08 00:00:00
4	Chaitali	2060	2008-05-20 00:00:00
5	Hardik	NULL	NULL
6	Komal	NULL	NULL
7	Muffy	NULL	NULL

RIGHT JOIN :-

The SQL **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 left table, the join will still return a row

in the result, but with NULL in each column from 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 ID, NAME, AMOUNT, DATE
        FROM CUSTOMERS
        RIGHT JOIN ORDERS
        ON CUSTOMERS.ID = ORDERS.CUSTOMER_ID;
```

RESULT :

ID	NAME	AMOUNT	DATE
3	kaushik	3000	2009-10-08 00:00:00
3	kaushik	1500	2009-10-08 00:00:00
2	Khilan	1560	2009-11-20 00:00:00
4	Chaitali	2060	2008-05-20 00:00:00

FULL JOIN :-

The SQL FULL JOIN combines the results of both left and right outer joins. The joined table will contain all records from both tables, and fill in NULLs for missing matches on either side.

Syntax:

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

RESULT :

ID	NAME	AMOUNT	DATE
1	Ramesh	NULL	NULL
2	Khilan	1560	2009-11-20 00:00:00
3	kaushik	3000	2009-10-08 00:00:00
3	kaushik	1500	2009-10-08 00:00:00
4	Chaitali	2060	2008-05-20 00:00:00
5	Hardik	NULL	NULL
6	Komal	NULL	NULL
7	Muffy	NULL	NULL
3	kaushik	3000	2009-10-08 00:00:00
3	kaushik	1500	2009-10-08 00:00:00
2	Khilan	1560	2009-11-20 00:00:00
4	Chaitali	2060	2008-05-20 00:00:00

SELF JOIN :-

The SQL **SELF JOIN** is used to join a table to itself as if the table were two tables, temporarily renaming at least one table in the SQL statement.

SYNTAX :

```
SELECT a.ID, b.NAME, a.SALARY
FROM CUSTOMERS a, CUSTOMERS b
WHERE a.SALARY < b.SALARY;
```

RESULT :

ID	NAME	SALARY
2	Ramesh	1500.00
2	kaushik	1500.00
1	Chaitali	2000.00
2	Chaitali	1500.00
3	Chaitali	2000.00
6	Chaitali	4500.00
1	Hardik	2000.00
2	Hardik	1500.00
3	Hardik	2000.00
4	Hardik	6500.00

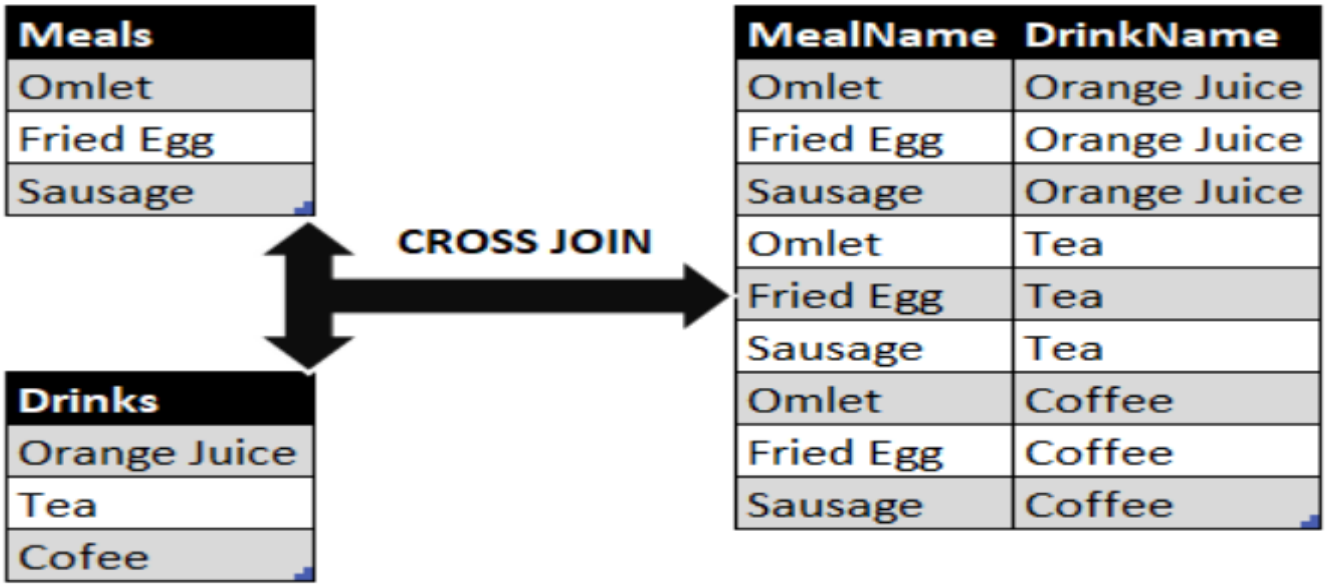
CROSS JOIN:

The CROSS JOIN is used to generate a paired combination of each row of the first table with each row of the second table.

This join type is also known as cartesian join.

SYNTAX :

```
SELECT * FROM Meals CROSS JOIN Drinks
```



SQL Union Operator :

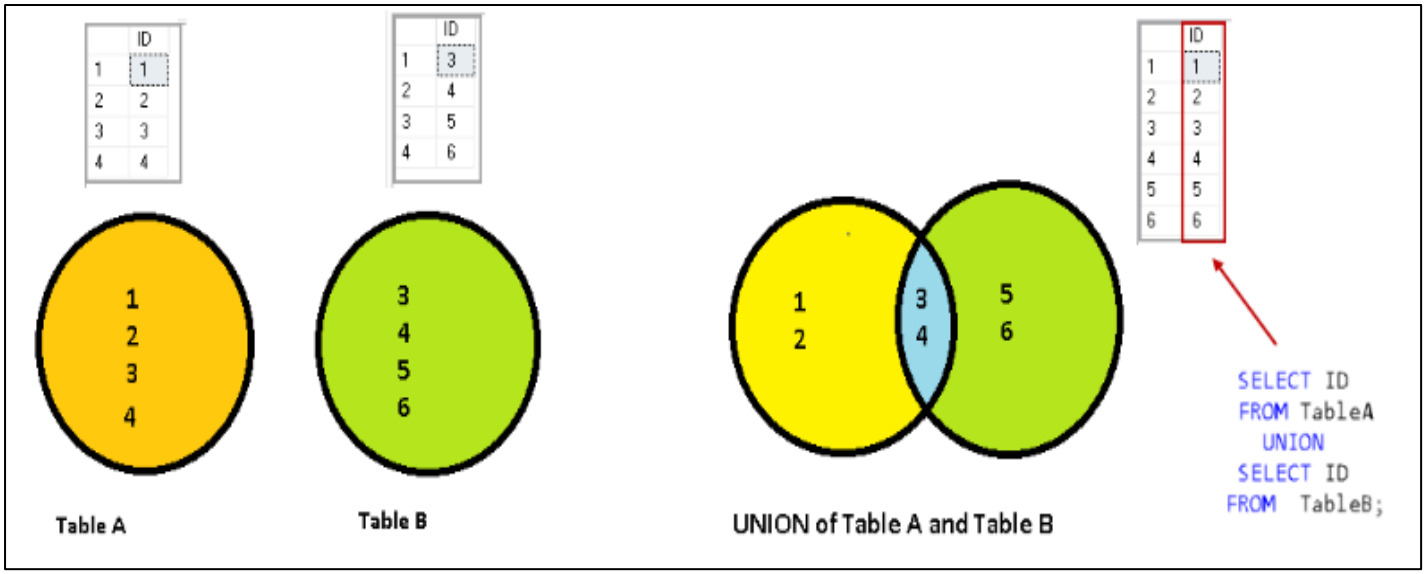
In the relational database, we stored data into SQL tables.

Sometimes we need to Select data from multiple tables and combine result set of all Select statements.

We use the SQL Union operator to combine two or more Select statement result set.

SYNTAX :

```
SELECT column1, Column2 ...Column (N) FROM tableA
UNION
SELECT column1, Column2 ...Column (N) FROM tableB;
```

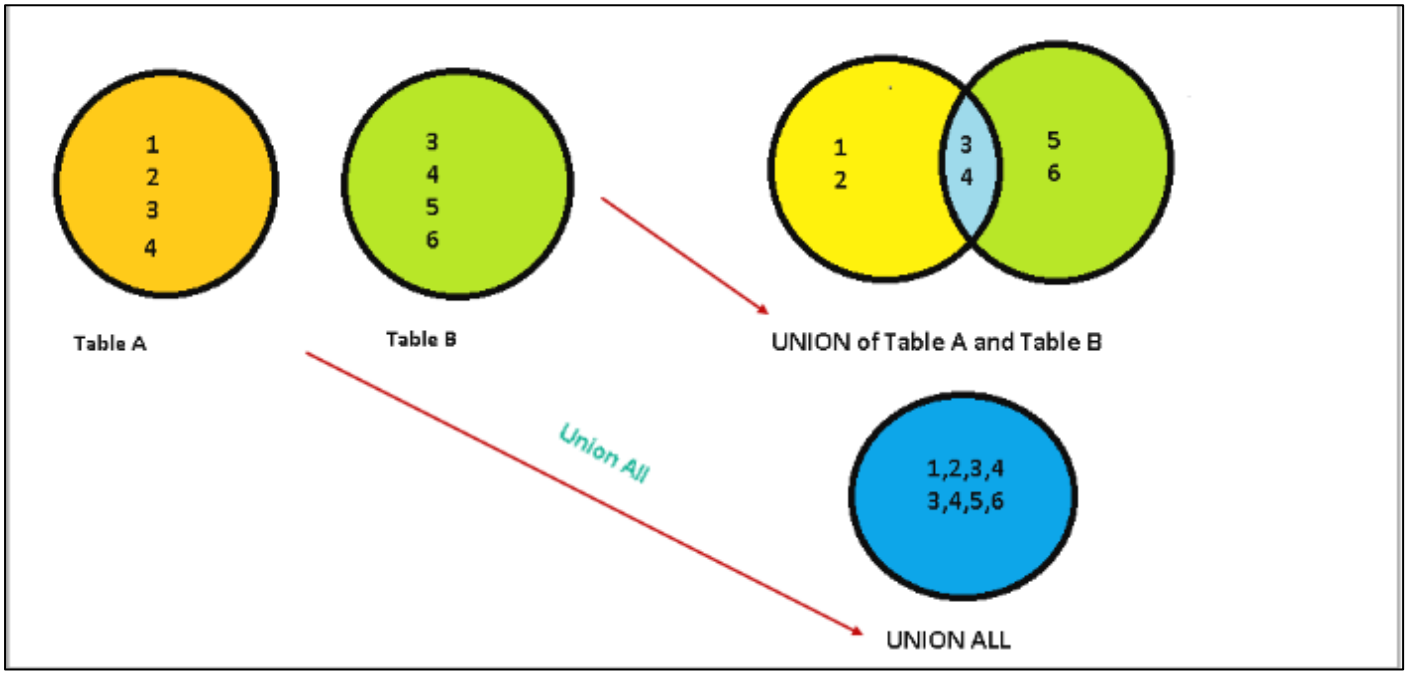


SQL Union All Operator :

The SQL Union All operator combines the result of two or more Select statement similar to a SQL Union operator with a difference. The only difference is that it does not remove any duplicate rows from the output of the Select statement.

syntax

```
SELECT column1, Column2 ...Column (N) FROM tableA
Union All
SELECT column1, Column2 ...Column (N) FROM tableB;
```



28. The SQL CASE Expression

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 Syntax

```
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;
```

29. SQL Server Window Functions :

1. SQL Server ROW_NUMBER Function

The ROW_NUMBER() is a window function that assigns a sequential integer to each row within

the partition of a result set. The row number starts with 1 for the first row in each partition.

The following shows the syntax of the ROW_NUMBER() function:

```
ROW_NUMBER() OVER (
        [PARTITION BY partition_expression, ... ]
ORDER BY sort_expression [ASC | DESC], ...
)
```

PARTITION BY

The PARTITION BY clause divides the result set into partitions (another term for groups of rows).

The ROW_NUMBER() function is applied to each partition separately and reinitialized the row number for each partition.

The PARTITION BY clause is optional. If you skip it, the ROW_NUMBER() function will

treat the whole result set as a single partition.

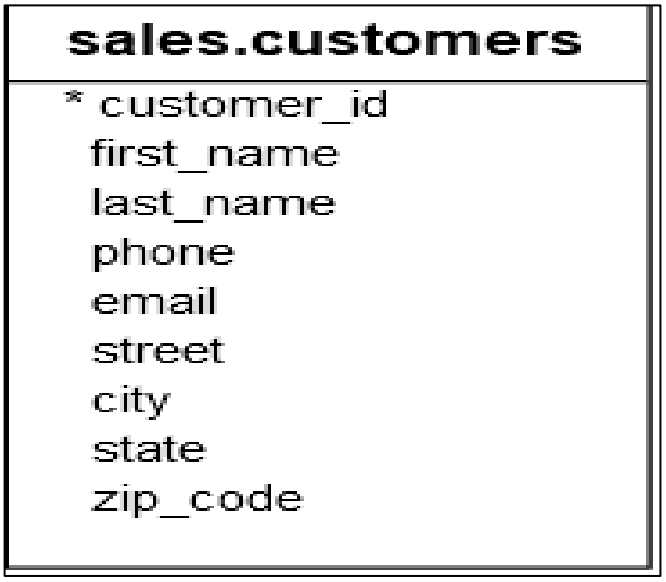
ORDER BY

The ORDER BY clause defines the logical order of the rows within each partition of the result set.

The ORDER BY clause is mandatory because the ROW_NUMBER() function is order sensitive.

SQL Server ROW_NUMBER() examples

We'll use the sales.customers table from the sample database to demonstrate the ROW_NUMBER() function.



The following statement uses the ROW_NUMBER() to assign each customer row a sequential number:

```
SELECT
    ROW_NUMBER() OVER (ORDER BY first_name) row_num,
    first_name,
    last_name,
    city
FROM
sales.customers;
```

RESULT :-

row_num	first_name	last_name	city
1	Aaron	Knapp	Yonkers
2	Abbey	Pugh	Forest Hills
3	Abby	Gamble	Amityville
4	Abram	Copeland	Harlingen
5	Adam	Henderson	Los Banos
6	Adam	Thornton	Central Islip

SQL Server DENSE_RANK Function

The DENSE_RANK() is a window function that assigns a rank to each row within a partition of a result set.

Unlike the RANK() function, the DENSE_RANK() function

returns consecutive rank values. Rows in each partition receive the same ranks if they have the same values.

```
SELECT  DENSE_RANK () OVER ( ORDER BY v ) my_dense_rank FROM sales.rank_demo;
```

SQL Server RANK Function

The RANK() function is a window function that assigns a rank to each row within a partition of a result set.

The rows within a partition that have the same values will receive the same rank. The rank of the first row within a partition is one. The RANK() function adds the number of tied rows to the tied rank to calculate the rank of the next row, therefore, the ranks may not be consecutive.

```
SELECT v, RANK () OVER ( ORDER BY v ) my_rank FROM sales.rank_demo;
```

v	my_dense_rank	my_rank
A	1	1
B	2	2
B	2	2
C	3	4
C	3	4
D	4	6
E	5	7

SUBQUERY IN SQL :-

A Subquery or Inner query or a Nested query is a query within another SQL query and embedded within

clauses, most commonly in the WHERE clause. It is used to return data from a table, and this data

will be used in the main query as a condition to further restrict the data to be retrieved.

Subqueries can be used with the SELECT, INSERT, UPDATE, and DELETE statements along with the

operators like **=, <, >, >=, <=, IN, BETWEEN,** etc.

There are a few rules that subqueries must follow –

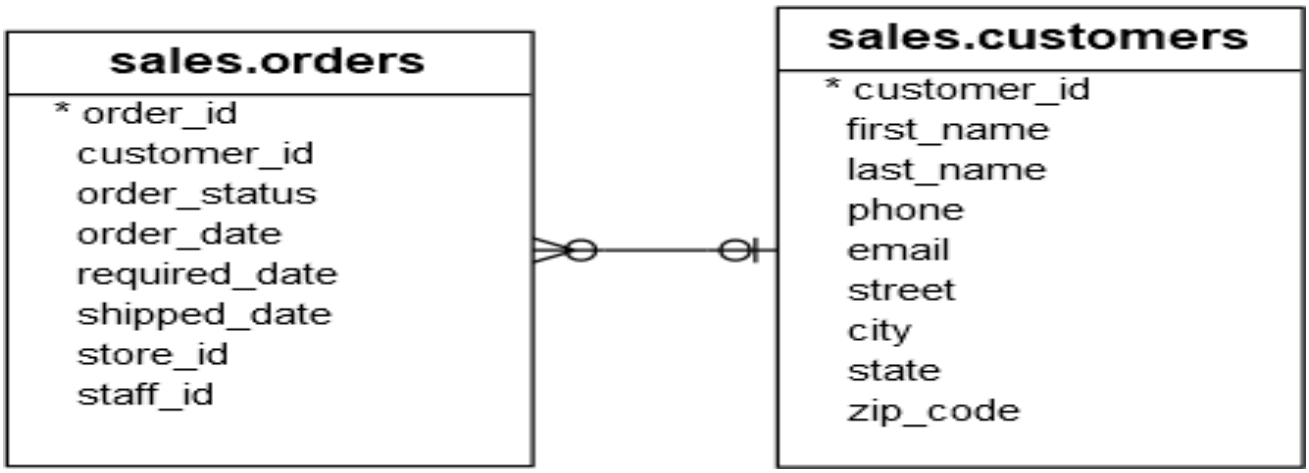
- Subqueries must be enclosed within parentheses.
- A subquery can have only one column in the SELECT clause, unless multiple columns are in the main query

  for the subquery to compare its selected columns.
- An ORDER BY command cannot be used in a subquery, although the main query can use an ORDER BY.

  The GROUP BY command can be used to perform the same function as the ORDER BY in a subquery.
- Subqueries that return more than one row can only be used with multiple value operators such as the IN operator.
- The SELECT list cannot include any references to values that evaluate to a BLOB, ARRAY, CLOB, or NCLOB.
- A subquery cannot be immediately enclosed in a set function.
- The BETWEEN operator cannot be used with a subquery. However, the BETWEEN operator can be used within the subquery.

Let’s see the following example.

Consider the orders and customers tables



The following statement shows how to use a subquery in the WHERE clause of a SELECT statement to find The sales orders of the customers located in New York:

```
SELECT
    order_id,
    order_date,
    customer_id
FROM
    sales.orders
WHERE
    customer_id IN (
        SELECT
            customer_id
        FROM
            sales.customers
        WHERE
            city = 'New York'
    )
ORDER BY
    order_date DESC;
```

outer query

subquery

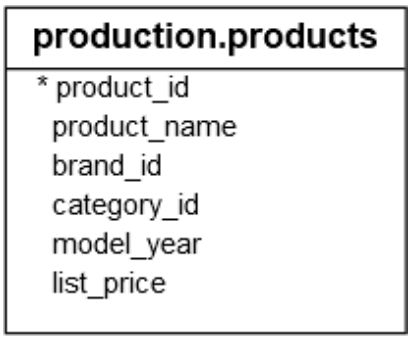
SQL Server correlated subquery

A correlated subquery is a subquery that uses the values of the outer query. In other words, the correlated subquery depends on the outer query for its values.

Because of this dependency, a correlated subquery cannot be executed independently as a simple subquery.

Moreover, a correlated subquery is executed repeatedly, once for each row evaluated by the outer query. The correlated subquery is also known as a repeating subquery.

Consider the following products table



The following example finds the products whose list price is equal to the highest list price of the products within the same category:

```
SELECT
        product_name,
        list_price,
        category_id
FROM
        production.products p1
WHERE
        list_price IN (
                SELECT
                        MAX (p2.list_price)
                FROM
                 production.products p2
                WHERE
                 p2.category_id = p1.category_id
                GROUP BY
                        p2.category_id
                 )
ORDER BY
         category_id,
        product_name;
```

Here we are using P1 alias to filter P2 alias table so its correlated subquery

31. CTE in SQL Server

CTE stands for common table expression. A CTE allows you to define a temporary named result set that available temporarily in the execution scope of a statement such as SELECT, INSERT, UPDATE, DELETE, or MERGE.

The following shows the common syntax of a CTE in SQL Server:

```
WITH expression_name[(column_name [,...])]
        AS
        (CTE_definition)
        SQL_statement;
```

In this syntax:

- First, specify the expression name (expression_name) to which you can refer later in a query.
- Next, specify a list of comma-separated columns after the expression_name. The number of columns must be the same as the number of columns defined in the CTE_definition.
- Then, use the AS keyword after the expression name or column list if the column list is specified.
- After, define a SELECT statement whose result set populates the common table expression.
- Finally, refer to the common table expression in a query (SQL_statement) such as SELECT, INSERT, UPDATE, DELETE, or MERGE.
- We prefer to use common table expressions rather than to use subqueries because common table expressions are more readable. We also use CTE in the queries that contain analytic functions (or window functions)

SQL Server CTE examples

Let’s take some examples of using common table expressions.

A) Simple SQL Server CTE example

This query uses a CTE to return the sales amounts by sales staffs in 2018:

```
WITH cte_sales_amounts (staff, sales, year) AS (
    SELECT
        first_name + ' ' + last_name,
        SUM(quantity * list_price * (1 - discount)),
        YEAR(order_date)
    FROM
    sales.orders o
    INNER JOIN sales.order_items i ON i.order_id = o.order_id
    INNER JOIN sales.staffs s ON s.staff_id = o.staff_id
    GROUP BY
        first_name + ' ' + last_name,
        year(order_date)
)

SELECT staff, sales
FROM
    cte_sales_amounts
WHERE
year = 2018;
```

B. Using multiple SQL Server CTE in a single query example

```
WITH cte_category_counts (
    category_id,
    category_name,
    product_count
)
AS (
    SELECT
        c.category_id,
        c.category_name,
        COUNT(p.product_id)
    FROM
        production.products p
        INNER JOIN production.categories c
            ON c.category_id = p.category_id
    GROUP BY
        c.category_id,
        c.category_name
),
cte_category_sales(category_id, sales) AS (

    SELECT
        p.category_id,
        SUM(i.quantity * i.list_price * (1 - i.discount))
    FROM
        sales.order_items i
        INNER JOIN production.products p
            ON p.product_id = i.product_id
        INNER JOIN sales.orders o
            ON o.order_id = i.order_id
    WHERE order_status = 4 -- completed
    GROUP BY
        p.category_id
)
SELECT
    c.category_id,
    c.category_name,
    c.product_count,
    s.sales
FROM
    cte_category_counts c
    INNER JOIN cte_category_sales s
        ON s.category_id = c.category_id
ORDER BY
    c.category_name;
```

32. SQL Server recursive CTE :-

A recursive common table expression (CTE) is a CTE that references itself. By doing so, the CTE repeatedly executes, returns subsets of data, until it returns the complete result set.
A recursive CTE is useful in querying hierarchical data such as organization charts where one employee reports to a manager or multi-level bill of materials when a product consists of many components, and each component itself also consists of many other components.
The following shows the syntax of a recursive CTE:

```
WITH expression_name (column_list)
AS
(
                    -- Anchor member
    initial_query
    UNION ALL
                -- Recursive member that references expression_name.
    recursive_query
)
                -- references expression name
SELECT *
FROM   expression_name
```

In general, a recursive CTE has three parts:

1. An initial query that returns the base result set of the CTE. The initial query is called an anchor member.

2. A recursive query that references the common table expression, therefore, it is called the recursive member. The recursive member is union-ed with the anchor member using the UNION ALL operator.

3. A termination condition specified in the recursive member that terminates the execution of the recursive member.

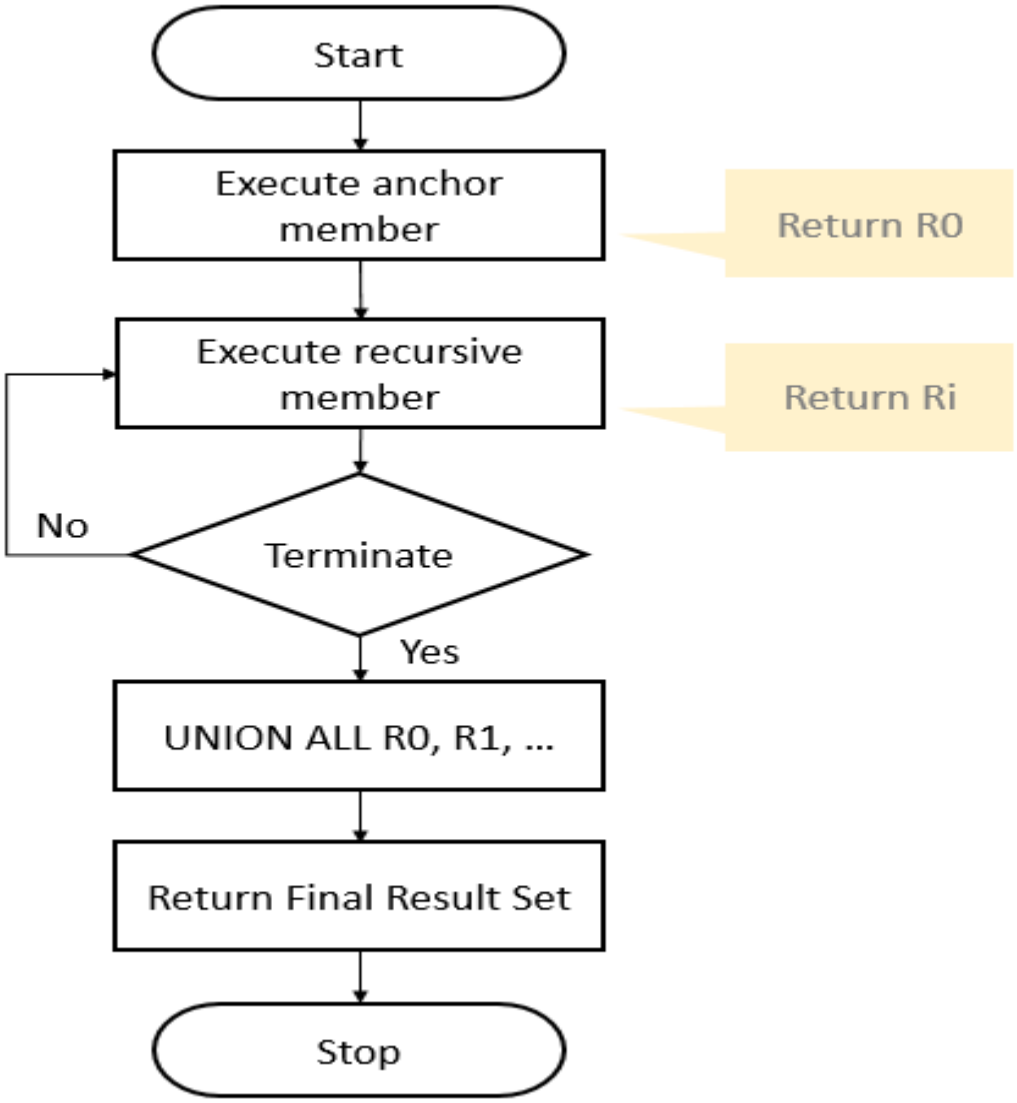
The execution order of a recursive CTE is as follows:

1. First, execute the anchor member to form the base result set (R0), use this result for the next iteration.

2. Second, execute the recursive member with the input result set from the previous iteration (Ri-1) and return a sub-result set (Ri) until the termination condition is met.

3. Third, combine all result sets R0, R1, … Rn using UNION ALL operator to produce the final result set.

The following flowchart illustrates the execution of a recursive CTE:



SQL Server Recursive CTE examples

Let's take some examples of using recursive CTEs

A) Simple SQL Server recursive CTE example

This example uses a recursive CTE to returns weekdays from Monday to Saturday:

```
WITH cte_numbers(n, weekday)
AS (
    SELECT
        0,
        DATENAME(DW, 0)

    UNION ALL
    SELECT
        n + 1,
        DATENAME(DW, n + 1)
    FROM
        cte_numbers
    WHERE n < 6
)
SELECT
    weekday
FROM
    cte_numbers;
```

Result :-

weekday
Monday
Tuesday
Wednesday
Thursday
Friday
Saturday
Sunday

Note :- The DATENAME() function returns the name of the weekday based on a weekday number.

33. SQL Server Temporary Tables.

Temporary tables are tables that exist temporarily on the SQL Server.

The temporary tables are useful for storing the immediate result sets that are accessed multiple times.

Creating temporary tables

SQL Server provided two ways to create temporary tables via **SELECT INTO** and **CREATE TABLE** statements.

Create temporary tables using SELECT INTO statement

The first way to create a temporary table is to use the SELECT INTO statement as shown below:

The name of the temporary table starts with a hash symbol (#). For example, the following statement

creates a temporary table using the SELECT INTO statement:

```
SELECT product_name, list_price
        INTO #trek_products --- temporary table
        FROM
        production.products
        WHERE
        brand_id = 9;
```

Create temporary tables using CREATE TABLE statement

The second way to create a temporary table is to use the CREATE TABLE statement:

```
CREATE TABLE #haro_products (
 product_name VARCHAR(MAX),
 list_price DEC(10,2)
);
```

This statement has the same syntax as creating a regular table. However, the name of the temporary table starts with a hash symbol (#)

After creating the temporary table, you can insert data into this table as a regular table:

```
INSERT INTO #haro_products
        SELECT
          product_name,
         list_price
        FROM
         production.products
        WHERE
         brand_id = 2;
```

34. **SQL Views :-**

SQL CREATE VIEW Statement :

In SQL, a view is a virtual table based on the result-set of an SQL statement.

A view contains rows and columns, just like a real table. The fields in a view are fields from one or

more real tables in the database.

You can add SQL statements and functions to a view and present the data as if the data were

coming from one single table.

A view is created with the **CREATE VIEW** statement.

CREATE VIEW Syntax

```
CREATE VIEW view_name AS
SELECT column1, column2, ...
FROM table_name
WHERE condition;
```

Note: A view always shows up-to-date data! The database engine recreates the view, every time a user queries it.

35. SQL Server Transaction :-

Introduction to the SQL Server Transaction

A transaction is a single unit of work that typically contains multiple T-SQL statements.

If a transaction is successful, the changes are committed to the database. However, if a transaction has an error, the changes have to be rolled back.

When executing a single statement such as INSERT, UPDATE, and DELETE, SQL Server uses the autocommit transaction. In this case, each statement is a transaction.

To start a transaction explicitly, you use the BEGIN TRANSACTION or BEGIN TRAN statement first:

BEGIN TRANSACTION;

Then, execute one or more statements including INSERT, UPDATE, and DELETE.

Finally, commit the transaction using the COMMIT statement:

COMMIT;

Or roll back the transaction using the ROLLBACK statement:

ROLLBACK;

```
-- start a transaction
BEGIN TRANSACTION;
-- other statements
-- commit the transaction
COMMIT;
```

Summary

- Use the BEGIN TRANSACTION statement to start a transaction explicitly.
- Use the COMMIT statement to commit the transaction and ROLLBACK statement to roll back the transaction.

36. <u>What is a variable in SQL</u> :-

A variable is an object that holds a single value of a specific type e.g., integer, date, or varying character string.

We typically use variables in the following cases:

As a loop counter to count the number of times a loop is performed.

To hold a value to be tested by a control-of-flow statement such as WHILE.

To store the value returned by a stored procedure or a function

Declaring a variable.

To declare a variable, you use the DECLARE statement.

For example, the following statement declares a variable named @model_year:

```
DECLARE @model_year SMALLINT;
```

The DECLARE statement initializes a variable by assigning it a name and a data type.

The variable name must start with the @ sign.

In this example, the data type of the @model_year variable is SMALLINT.

By default, when a variable is declared, its value is set to NULL.

Between the variable name and data type, you can use the optional AS keyword as follows:

```
DECLARE @model_year AS SMALLINT;
```

To declare multiple variables, you separate variables by commas:

```
DECLARE @model_year SMALLINT,
        @product_name VARCHAR(MAX);
```

Assigning a value to a variable

To assign a value to a variable, you use the SET statement. For example, the following statement assigns 2018 to the @model_year variable:

```
SET @model_year = 2018;
```

Using variables in a query

The following SELECT statement uses the @model_year variable in the WHERE clause to find the products of a specific model year:

```
SELECT product_name, model_year, list_price FROM production.products
WHERE
    model_year = @model_year ORDER BY product_name;
```

37.Error Handling in SQL Server with Try Catch

```
BEGIN TRY
    -- Write statements here that may cause exception
END TRY
BEGIN CATCH
    -- Write statements here to handle exception
END CATCH
```

In SQL Server you can take advantage of TRY...CATCH statements to handle errors. When writing code that

handles errors, you should have a TRY block and a CATCH block immediately after it.

The TRY block starts with a BEGIN TRY statement and ends with an END TRY statement. Similarly,

the CATCH block starts with a BEGIN CATCH statement and ends with an END CATCH statement.

When an error occurs inside the TRY block, the control moves to the first statement inside the CATCH block.

On the contrary, if the statements inside a TRY block have completed execution successfully without an error,

the control will not flow inside the CATCH block. Rather, the first statement immediately after the END CATCH

statement will then be executed. In this article we'll take advantage of the Northwind database to run our queries.

Retrieving detailed information on the error

You can take advantage of various functions inside the CATCH block to get detailed information about an error.

These functions include the following:

- ERROR_MESSAGE() - you can take advantage of this function to get the complete error message.
- ERROR_LINE() - this function can be used to get the line number on which the error occurred.
- ERROR_NUMBER() - this function can be used to get the error number of the error.
- ERROR_SEVERITY() - this function can be used to get the severity level of the error.
- ERROR_STATE() - this function can be used to get the state number of the error.
- ERROR_PROCEDURE() - this function can be used to know the name of the stored procedure or trigger that has caused the error.

EXAMPLE:-

```
BEGIN TRY
  Insert Into Categories(CategoryID, CategoryName,
Description, Picture) Values (9, 'Test', 'Test Description', 'Test')
END TRY
BEGIN CATCH
  SELECT ERROR_MESSAGE() AS [Error Message]
      ,ERROR_LINE() AS ErrorLine
      ,ERROR_NUMBER() AS [Error Number]
      ,ERROR_SEVERITY() AS [Error Severity]
      ,ERROR_STATE() AS [Error State]
END CATCH
```

38. SQL Server Stored Procedures

SQL Server stored procedures are used to group one or more Transact-SQL statements into logical units. The stored procedure is stored as a named object in the SQL Server Database Server.

When you call a stored procedure for the first time, SQL Server creates an execution plan and stores it in the cache. In the subsequent executions of the stored procedure, SQL Server reuses the plan to execute the stored procedure very fast with reliable performance.

To create a stored procedure that wraps this query, you use the CREATE PROCEDURE statement as follows:

```
CREATE PROCEDURE uspProductList
AS
BEGIN
    SELECT
    product_name,
    list_price
    FROM
    production.products
    ORDER BY
    product_name;
END;
```

Executing a stored procedure :- To execute a stored procedure, you use the EXECUTE or EXEC statement followed by the name of the stored procedure:

```
EXECUTE sp_name;
    Or
EXEC sp_name;
```

Deleting a stored procedure

To delete a stored procedure, you use the DROP PROCEDURE or DROP PROC statement:

```
DROP PROCEDURE sp_name;
        or
DROP PROC sp_name;
```

SQL Server Stored Procedure Parameters

```
Create PROCEDURE uspFindProducts(@min_list_price AS DECIMAL)
AS
BEGIN
   SELECT
      product_name,
      list_price
   FROM
      production.products
   WHERE
      list_price >= @min_list_price
   ORDER BY
      list_price;
END;
```

To execute the uspFindProducts stored procedure, you pass an argument to it as follows:

```
EXEC uspFindProducts 100;
```

39. SQL Server Triggers

SQL Server triggers are special stored procedures that are executed automatically in response to the database object, database, and server events. SQL Server provides three type of triggers:

Data manipulation language (DML) triggers which are invoked automatically in response to INSERT, UPDATE, and DELETE events against tables.

Data definition language (DDL) triggers which fire in response to CREATE, ALTER, and DROP statements. DDL triggers also fire in response to some system stored procedures that perform DDL-like operations.

Logon triggers which fire in response to LOGON events

In this section, you will learn how to effectively use triggers in SQL Server.

Creating a trigger in SQL Server – show you how to create a trigger in response to insert and delete events.

Creating an INSTEAD OF trigger – learn about the INSTEAD OF trigger and its practical applications.

SQL Server CREATE TRIGGER

The CREATE TRIGGER statement allows you to create a new trigger that is fired automatically whenever an event such as INSERT, DELETE, or UPDATE occurs against a table.

The following illustrates the syntax of the CREATE TRIGGER statement:

```
CREATE TRIGGER [schema_name.]trigger_name
        ON table_name
        AFTER  {[INSERT],[UPDATE],[DELETE]}
        [NOT FOR REPLICATION]
        AS
        {sql_statements}
```

In this syntax:

The schema_name is the name of the schema to which the new trigger belongs. The schema name is optional.

The trigger_name is the user-defined name for the new trigger.

The table_name is the table to which the trigger applies.

The event is listed in the AFTER clause. The event could be INSERT, UPDATE, or DELETE. A single trigger can fire in response to one or more actions against the table.

The NOT FOR REPLICATION option instructs SQL Server not to fire the trigger when data modification is made as part of a replication process.

The sql_statements is one or more Transact-SQL used to carry out actions once an event occurs.

“Virtual” tables for triggers: INSERTED and DELETED

SQL Server provides two virtual tables that are available specifically for triggers called INSERTED and DELETED tables. SQL Server uses these tables to capture the data of the modified row before and after the event occurs.

The following table shows the content of the INSERTED and DELETED tables before and after each event:

DML event	INSERTED table holds	DELETED table holds
INSERT	rows to be inserted	empty
UPDATE	new rows modified by the update	existing rows modified by the update
DELETE	empty	rows to be deleted

40. User-Defined Functions

```
CREATE FUNCTION [database_name.]function_name (parameters)
RETURNS data_type AS
BEGIN
   SQL statements
   RETURN value
END;
```

```
ALTER FUNCTION [database_name.]function_name (parameters)
RETURNS data_type AS
BEGIN
   SQL statements
   RETURN value
END;
```

```
DROP FUNCTION [database_name.]function_name;
```

User-Defined Function Returning the Table

```
CREATE FUNCTION east_from_long (
        @long DECIMAL(9,6)
)
RETURNS TABLE AS
RETURN
        SELECT *
        FROM city
        WHERE city.long > @long;
```

SQL SERVER FUNCTIONS

Function Category	Function Name	Description
Aggregate	COUNT()	Returns the number of rows in a result set.
	SUM()	Calculates the sum of values in a column.
	AVG()	Computes the average of values in a column.
	MIN()	Returns the minimum value in a column.
	MAX()	Returns the maximum value in a column.
	GROUP_CONCAT()	Concatenates strings from a group into one string.
String	CONCAT()	Concatenates two or more strings.
	LEN()	Returns the length of a string.
	UPPER() / UCASE()	Converts a string to uppercase.
	LOWER() / LCASE()	Converts a string to lowercase.
	LEFT()	Extracts a specified number of characters from the beginning of a string.
	RIGHT()	Extracts a specified number of characters from the end of a string.
Date and Time	GETDATE()	Returns the current system date and time.
	DATEPART()	Extracts a specific part of a date/time.
	DATEADD()	Adds a specified interval to a date/time.
	DATEDIFF()	Calculates the difference between two dates/times.
	YEAR()	Extracts the year from a date.
	MONTH()	Extracts the month from a date.
	DAY()	Extracts the day of the month from a date.
Conversion	CAST()	Converts data from one data type to another.
	CONVERT()	Converts data from one data type to another, with specific formatting.
Conditional	CASE	Performs conditional logic in SQL queries.
	COALESCE()	Returns the first non-null value from a list of expressions.
	NULLIF()	Returns NULL if two expressions are equal; otherwise, returns the first expression.
	IIF()	Returns one of two values based on a Boolean expression.
Mathematical	ROUND()	Rounds a numeric value to a specified precision.
	CEILING()	Rounds up to the nearest integer.
	FLOOR()	Rounds down to the nearest integer.
	ABS()	Returns the absolute value of a numeric expression.
	POWER()	Raises a number to a specified power.
	SQRT()	Returns the square root of a number.
	RAND()	Generates a random number between 0 and 1.