**SQL Server Important Interview Questions**

## <https://www.c-sharpcorner.com/UploadFile/puranindia/sql-server-interview-questions/>

<https://www.sqlservertutorial.net/sql-server-aggregate-functions/>

[https://www.javatpoint.com/dbms-sql-command#:~:text=SQL%20commands%20are%20instructions.,table%2C%20set%20permission%20for%20users](https://www.javatpoint.com/dbms-sql-command" \l ":~:text=SQL%20commands%20are%20instructions.,table%2C%20set%20permission%20for%20users" \t "_blank)

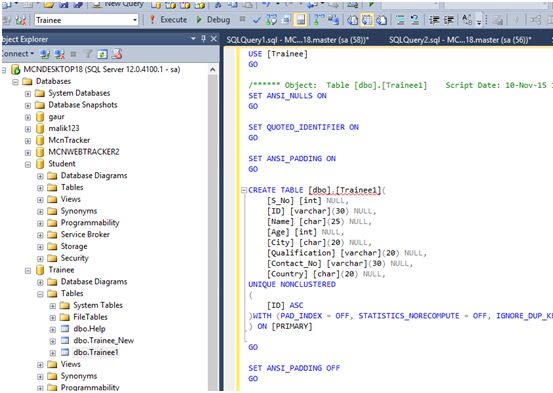
<https://www.javatpoint.com/common-table-expression-in-sql-server>

[SQL Roadmap - roadmap.sh](https://roadmap.sh/sql)

## **1. What is SQL?**

Structured Query Language (SQL) is a programming language for accessing and manipulating Relational Database Management Systems (RDBMSs). SQL is widely used in popular RDBMSs such as SQL Server, Oracle, and MySQL. The smallest unit of execution in SQL is a query. A SQL query is used to select, update, and delete data.

In RDBMSs, all the data is stored in tables, with each table consisting of rows and columns. The following is an example of a SQL query, CREATE DATABASE.



## **2. What is RDBMS?**

RDBMS is referred to as Relation Database Management Systems (RDBMS). RDBMS possesses the following characteristics:

* **Write-intensive operations-** The RDBMS is frequently written to and is often used in transaction-oriented applications.
* **Data in flux or historical data-** The RDBMS is designed to handle frequently changing data. Alternatively, RDBMS can also store vast amounts of historical data, which can later be analyzed or "mined".
* **Application-specific schema-** The RDBMS is configured per application, and a unique schema exists to support each application.
* **Complex data models-**The relational nature of the RDBMS makes it suitable for handling sophisticated, complex data models that require many tables, foreign key values, complex join operations, and so on.
* **Data integrity-**The RDBMS features many components designed to ensure data integrity. This includes rollback operations, referential integrity, and transaction-oriented operations.

## **3. What is a Check in SQL?**

A Check Constraint is a rule that identifies valid values for columns of data. A Check Constraint helps to enforce Domain Integrity. If the condition in a Check Constraint is not satisfied then prevents the value from entering the database.

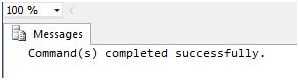
**Syntax**

Create table tableName(Column1 dataType Check(expression), Column2, columnN)

**Example**

create table emp(empId int check(empId >10),empName varchar(15))

**Output**



insert into emp values(8,'d')

**Output**

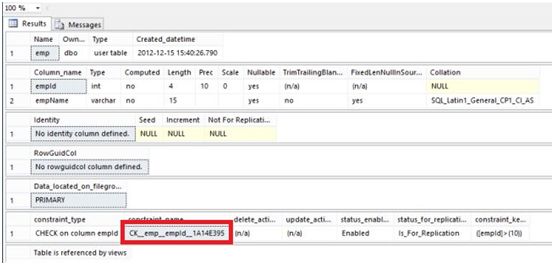


**Dropping the Check Constraint**

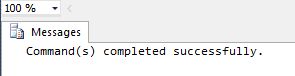
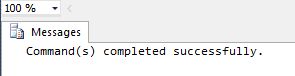
First, we can determine the name of the constraint using the following command:

exec sp\_help emp

**Output**



**Now execute the following command:**

alter table emp drop constraint CK\_\_emp\_\_empId\_\_1A14E395 **Output:**  
  
  
**Adding the Check Constraint:**  
  
alter table emp add check(empiD>15) **Output:  
  
  
  
Limitation:**The Check Constraint rejects the values that are invalid or we can say which does not satisfy the Check Conditions. But in the case of a null, a Check Constraint will allow it to be insert into the database.

## **4. What is a constraint in SQL?** **What is a default in SQL?**

Constraints are rules that decide what kind of data can be entered into the database tables. SQL server has six types of constraints and we will explore all these constraints here with suitable examples. The constraints that we are going to explore are listed below:

1. Primary Key Constraint
2. Foreign Key Constraint
3. Not Null Constraint
4. Unique constraint
5. Default Constraint
6. Check Constraint

**First, create two tables**

To explain these constraints, we need two tables. Firstly, let us create these tables. Run the scripts shown below to create the tables. Copy and paste the code into the new Query Editor window, then execute it.

CREATE TABLE Student(StudId smallint, StudName varchar(50), Class tinyint);

CREATE TABLE TotalMarks(StudentId smallint, TotalMarks smallint);

Go

Note that there are no constraints at present on these tables. We will add the constraints one by one.

**Primary Key Constraint**

A table column with this constraint is called the key column for the table. This constraint helps the table to make sure that the value is not repeated and also that there are no null entries. We will mark the StudId column of the Student table as the primary key. Follow these steps:

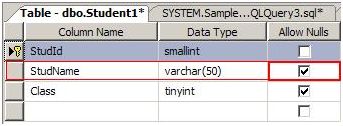
* Right-click the student table and click on the modify button
* From the displayed layout, select the StudId row by clicking the Small Square-like button on the left side of the row.
* Click on the Set Primary Key toolbar button to set the StudId column as the primary key column.

Now, this column does not allow null values and duplicate values. You can try inserting values to violate these conditions and see what happens. A table can have only one Primary key. Multiple columns can participate in the primary key column. Then the uniqueness is considered among all the participant columns by combining their values.

**Not Null Constraint**

This constraint is useful to stop storing the null entries in the specified columns. We will mark the student name column as the not null column. This allows us to always have some entries in the student name column of the student table without having NULL. Follow the steps below:

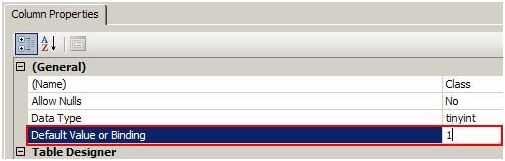
* As you did previously, bring up the table design view by clicking the modify context menu for the table.
* Remove the checkmark as shown in the picture below. This action will enable the Not Null constraint for the StudName column.



The default constraint allows you to set a default value for the column. That means when a row is created for the first time, and there is no entry specified for the column that has a default constraint on it, then the default value is stored in the column. Note that this is not a Not Null constraint, and do not confuse the default value constraint with disallowing the Null entries.

The default value for the column is set only when the row is created for the first time, and the column value is ignored on the Insert. Modification to the column with a NULL value or even the Insert operation specifying the Null value for the column is allowed. Let us set the Default value of 1 for the Class. Here are the steps:

* Bring up the table designer
* Select the Class Row as you already did.
* At the bottom of the layout, you will see a Column properties as shown in the below picture. Set the default as shown below:



## **5. How do I define constraints in SQL?**

Constraints are rules and restrictions applied to a column, or table so that unwanted data can't be inserted into tables. This ensures the accuracy and reliability of the data in the database. We can create constraints on single or multiple columns of any table. Constraints maintain the data integrity and accuracy in the table. Constraints can be classified into the following two types.

**Column Types Constraints**

Definitions of these types of constraints are given when the table is created.

Create Table My\_Constraint

(

ID int NOT NULL,

Salary int CHECK(Salary > 5000)

)

**Table Types Constraints**

Definitions of these types of constraints are given after the creation of the table using the Alter Command.

Alter Table My\_Cosntraint

Add constraint Check\_Constraint Check(Age>50)

SQL Server contains the following six types of constraints:

* Not Null Constraint
* Check Constraint
* Default Constraint
* Unique Constraint
* Primary Constraint
* Foreign Constraint

Let us understand each constraint briefly.

**Not Null Constraint**

A Not Null constraint restricts the insertion of null values into a column. If we are using a Not Null Constraint for a column, we cannot ignore this column's value during the data insertion into the table.

**Column Level**

**Syntax**

CREATE TABLE Table\_Name

(

Column\_Name Datatype CONSTRAINT Constraint\_Name NOT NULL,

);

**Example**

Create Table My\_Constraint

(

ID int NOT NULL,

Name nvarchar(50) CONSTRAINT Cons\_NotNull not null,

Age int Not Null,

)

**Table Level**

**Syntax**

ALTER TABLE Table\_Name

ALTER COLUMN Column\_Name Datatype NOT NULL

**Example**

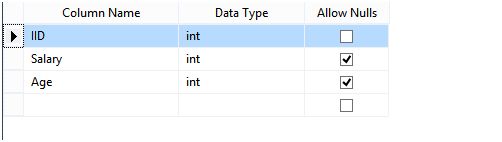
Alter Table My\_Constraint

Alter Column Id int Not Null

**Without SQL Command**

We can also create a Not Null constraint in Microsoft SQL Server without the execution of a SQL query.

First, right-click on the table and select and click on the design option. Now check all the columns in the “Allow Nulls” option that should have a Null Value.



A Check constraint checks for a specific condition before inserting data into a table. If the data passes all the Check constraints, then the data will be inserted into the table. Otherwise the data for insertion will be discarded. The CHECK constraint ensures that all values in a column satisfy certain conditions.

## **Check Constraint**

A Check constraint checks for a specific condition before inserting data into a table. If the data passes all the Check constraints, then the data will be inserted into the table; otherwise, the data for insertion will be discarded. The CHECK constraint ensures that all values in a column satisfy certain conditions.

**Column Level**

**Syntax**

Create Table Table\_Name

(

Column\_Name Datatype Constraint Constraint\_Name Check(Condition)

)

**Example**

Create Table Constraint\_

(

IId int Constraint Constraint\_Name Check(IId>100)

)

**Table Level**

**Syntax**

Alter Table Table\_Name

Add Constraint Constraint\_Name Check(Condition)

**Example**

Alter table Constraint\_

Add constraint Cons\_Name Check(IId>150)

## **Default Constraint**

Specifies a default value when a value is not specified for this column. If in an insertion query, any value is not specified for this column, then the default value will be inserted into the column.

**Column Level**

**Syntax**

Create Table Table\_Name

(

Column\_Name DataType Constraint Constraint\_Name Default(Value),

)

**Example**

Create Table My\_Table1

(

IId int default(1500),

Name Nvarchar(50)Constraint Name\_Default Default('Pankaj'),

Age Int,

Salary Int Default(100)

)

**Table Level**

**Syntax**

Alter Table Tabel\_Name

Add Constraint Constraint\_Name Default(Value) for[Column\_Name]

**Example**

Alter Table My\_Table1

Add Constraint cons\_Default Default(40) for[Age]

## **Without SQL Command**

Go to Table Design, click on the specific column name that should have a default value, and go to the column Property and provide the default value.

## **Unique Constraint**

It ensures that each row for a column must have a unique value. It is like a Primary key but can accept only one null value. In a table, one or more columns can contain a Unique Constraint.

**Column Level**

**Syntax**

Create Table Table\_Name

(

Column\_Name Datatype Constraint Constraint\_Name Unique

)

**Example**

Create Table MY\_Tab

(

IId int constraint Unique\_Cons Unique ,

Name nvarchar(50)

)

**Table Level**

**Syntax**

Alter Table\_Name

Add Constraint Constraint\_Name Unique(Column\_Name)

**Example**

Alter Table My\_Tab

Add Constraint Unique\_Cons\_ Unique(Name)

## **Without SQL Command**

First, go to the Table definition, select a column, and right-click on that column. Now select the option Index/Keys. Add a constraint and mark its "Is Unique" option as True. Now a window will be shown.

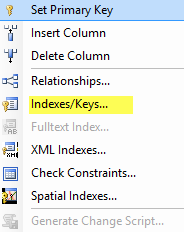


Figure 5 - Indexes & Keys

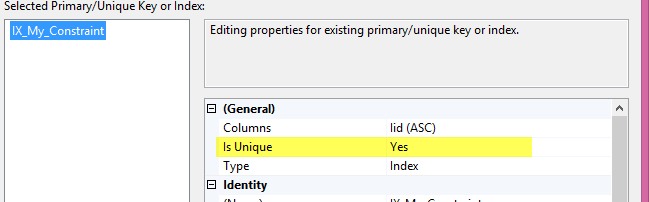


Figure 6 - Select Indexes

## **Primary Key Constraint**

A Primary key uniquely identifies each row in a table. One or more of the columns of a table can contain a Primary key. It cannot accept null and duplicate data.

**Column Level**

**Syntax**

Create Table Table\_Name

(

Column\_Name Datatype Constraint Constraint\_Name Primary Key,

)

**Example**

Create Table Employee

(

IId int constraint Const\_primary\_IId primary key,

Name nvarchar(50)

)

**Table Level**

**Syntax**

Alter Table Table\_Name

Add constraint Constraint\_Name Primary Key(Column\_Name)

**Example**

Alter Table Employee

Add constraint Constraint\_Name Primary Key(IId)

## **Without SqlQuery**

First, go to the table design, right-click Column, and select the "Set Primary Key" Option.

## **Foreign Key Constraint**

A Foreign Key is a field in a database table that is a Primary key in another table. A Foreign key creates a relation between two tables. The first table contains a primary key, and the second one contains a foreign one.

**Column Level**

**Syntax**

Create Table Table\_Name

(

Column\_Name Datatype Constraint Constraint\_Name References Reference\_Table\_Name(Reference\_Column\_Name)

)

**Example**

Create Table Employee\_

(

IId int constraint Cons\_Reference References My\_Constraint(IId),

Age int,

Salary int

)

**Table Level**

**Syntax**

ALTER TABLE Table\_Name

ADD CONSTRAINT Constraint\_Name FOREIGN KEY(Column\_Name)

REFERENCES Reference\_Table (Column\_Name)

**Example**

ALTER TABLE Employee\_

ADD CONSTRAINT Cons\_Emp\_Foreign FOREIGN KEY(IId)

REFERENCES My\_Constraint(IId)

## **Without SQL Command**

First, go to the table design, right-click on the column, and select the "Relationship" option. Now a window will be shown. In this window, click on the "Table and Column Specification" option and select Primary Key table, Column name, and Column name for the foreign key.

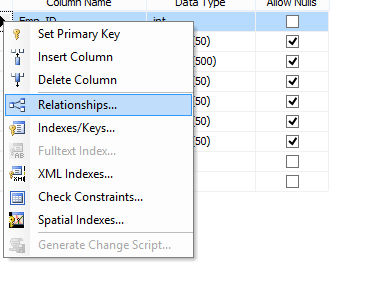


Figure 8 - Column Relationships

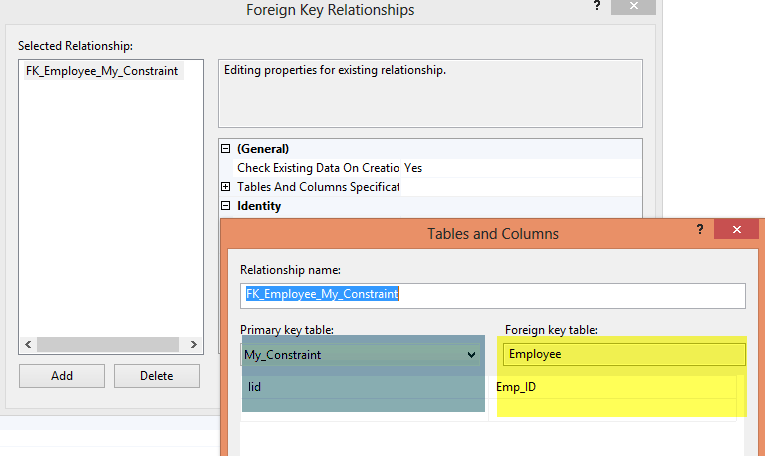
****

Figure 9 - Foreign Key Relationships

## **6. What is a database table?**

**Database table:** The table contains records in the form of rows and columns. A permanent table is created in the database you specify and remains in the database permanently until you delete it.

# **SQL Server Database Basics Query Commands**

**Create New Database**

Commands:

1. **Create** **database** [your **database** **name**]
2. **create** **database** SqlDataTable

**Database Rename**

Commands:

1. **Exec** sp\_Renamedb ‘your **current** **Database** **Name**’, ‘New **Database** **Name**’
2. **exec** sp\_renamedb 'SqlDataTable','SqlDatabase'

**Remove/Delete Database**

Commands:

1. **Drop** **Database** YourDatabaseName
2. **drop** **database** SqlDatabase

This section describes database table creation, delete and rename with SQL queries.

**Create Table in Database**

Command:

Create table TableName (columnName1 datatype, columnName2 datatype)

The following is an example of creating a simple table.

create table Info

(

Name varchar(20),

BirthDate date,

Phone nvarchar(12),

City varchar(20)

)

**Table Rename/Change Table Name**

Commands:

1. Sp\_rename ‘**Current** **Table**’,’New **Table** **Name**’
2. sp\_rename 'Info','PersonInfo'

**Table Remove/Delete**

Commands:

1. **Drop** **Table** TableName
2. **drop** **Table** PersonInfo

**Add new Column in Table**

Commands:

1. **Alter** **Table** Tablename
2. **Add** **Column** datatype
4. **alter** **table** Info
5. **add** Surname **varchar**(20)

**Column Rename in Table**

Commands:

1. Sp\_rename ‘**Table**.CurentColumnName’,‘NewColumnName’,‘**Column**’
2. sp\_rename 'Info.Name','FullName','column'

**Delete Column In table**

Commands:

1. **Alter** **Table** YourTable
2. **Drop** **Column** TableColumnName
3. **alter** **table** Info
4. **drop** **column** surname

**Column Identity**

It's important for any database table column, when we create an auto-increment number column in the database table.

Command:

ColumnName datatype Identity(n,n)  
  
Here, Id is the auto-incremented number.

1. **create** **table** Info
2. (
3. id **int** identity (1,1) not null,
4. **Name** **varchar**(20),
5. BirthDate **date**,
6. Phone nvarchar(12),
7. City **varchar**(20)
8. )

This section describes the SQL data process of inserting, updating and selecting data and a delete data query.

**Data Insert in Table**

Commands:

1. **Insert** **into** TableName **values** (column1, column2)
2. **insert** **into** Info **values** ('Rakesh','05-11-1986','9924194054','Ahmedabad')
3. or
4. **Insert** **into** info (**Name**,BirthDate,Phone,City) **values** ('Sagar','11-07-1990','9924141099','Surat')

**Data Insert With inserted Data Record Display**

It will display the inserted data while you are inserting data in table

Commands:

1. **Insert** **Into** TableName **output** inserted.\* **values** (column1,column2)
2. **insert** **into** info **output** inserted.\* **values** ('Rakesh','05-11-1986','9924194054','Ahmedabad')

**Select Data From Database Table**

Commands:

1. **Select** \* **from** TableName
2. **select** \* **from** info

**Update Data in Database Table**

Commands:

1. **Update** TableName **SET** ColumnName = NewData **where** Condition
2. **Update** info **Set** City = 'Baroda' **where** id = 2

**Delete Data Record From Database Table**

Commands:

**Delete All Table Records**

1. **Delete** TableName
2. **DELETE** info

**Delete Any One Record**

1. **Delete** **from** TableName **where** condition
2. **Delete** **from** Info **where** **Name**='Sagar'

**Unique**

This is for identifying each record.

Commands:

1. **Column** datatype **unique**
3. **create** **table** Employee
4. (
5. EmpId **int** **unique** not null,
6. EmpName **varchar**(20)
7. )

**Primary Key**

* Uniquely identify record
* it's not null
* Only one Primary Key in a SQL Table

Commands:

1. ColumnName datatype **Primary** **key**
3. **create** **table** Employee
4. (
5. EmpID **int** **Primary** **Key**,
6. EmpName **varchar**(20)
7. )

**FOREIGN KEY**

A Foreign Key is a field in a database table that is a Primary key in another table. A Foreign key creates a relation between two tables. The first table contains a primary key, and the second one contains a foreign one.  
  
Commands:  
  
ColumnName datatype Foreign Key References ExitingTableName(ExitingTable ColumnName)

1. **Create** **Table** Department
2. (
3. DepartmentId **int** **Primary** **Key**,
4. Department **varchar**(20),
5. EmpID **int** **Foreign** **Key** **References** Employee(EmpID)
6. )

**Unique Key Constraint**

Unique Constraint defined for a single or multiple columns.  
  
Commands:

Constraint ConstraintName Unique (Column1, Column2)

1. **Create** **table** Employee
2. (
3. EmpId **int** not null,
4. EmpName **varchar**(20) not null,
5. City **varchar**(20),
6. **constraint** UC\_Employee **Unique**(EmpId, EmpName)
7. )

**Primary Key Constraint**  
  
Commands:

Constraint ConstraintName Primary Key (Column)

1. **Create** **table** Employee
2. (
3. EmpId **int** not null,
4. EmpName **varchar**(20),
5. City **varchar**(20),
6. **constraint** PK\_EmpID **Primary** **Key**(EmpId)
7. )

**Foreign Key Constraint**  
Commands:

Constraint ConstraintName Foreign key (Column) References Table(Column)

1. **create** **table** Department
2. (
3. DepartmentId **int** not null **primary** **key**,
4. Department **varchar**(20),
5. EmpId **int**,
6. **constraint** FK\_Emp\_ID **Foreign** **key** (EmpId) **References** Employee(EmpId)

**Check Constraint**

Commands:

Constraint ConstraintName CHECK (Column condition)

1. **create** **table** Employees
2. (
3. EmpId **int**,
4. EmpName **varchar**(20),
5. Salary smallmoney,
6. **CONSTRAINT** CK\_Salary **CHECK** (Salary > 4000)
7. )

## **7. What are the relationships in the SQL Server database?**

Relationships are created by linking the column in one table with the column in another table. There are three different types of relationships that can be created. The relationships are listed below:

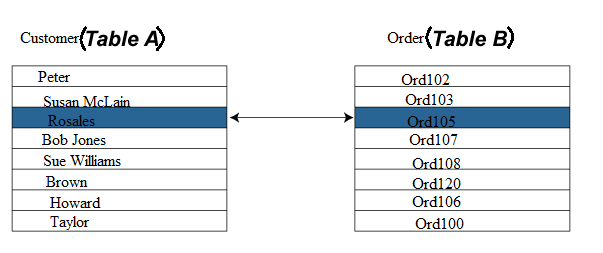
1. One-to-One Relationship
2. One -to-Many or Many-to-One Relationship
3. Many-to-Many Relationship

**Redundancy** means having multiple copies of the same data in the database. This problem arises when a database is not [normalized](https://www.geeksforgeeks.org/normal-forms-in-dbms/).

An **anomaly** is a deviation from the norm, a glitch or an error that doesn’t fit in with the rest of the pattern of the database.

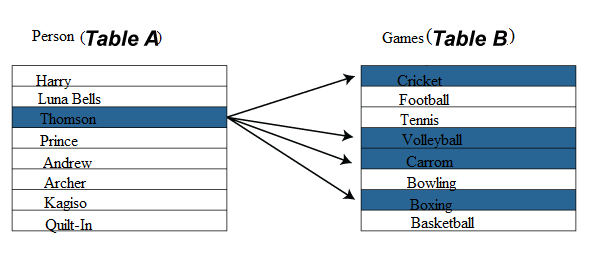
**One to One Relationship (1:1):** It is used to create a relationship between two tables in which a single row of the first table can only be related to one and only one records of a second table. Similarly, the row of a second table can also be related to anyone row of the first table.

Following is the example to show a relational database, as shown below.

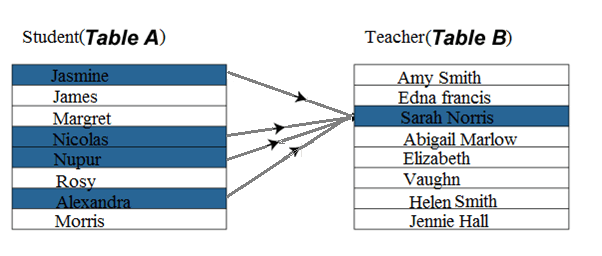


**One to Many Relationship:** It is used to create a relationship between two tables. Any single rows of the first table can be related to one or more rows of the second tables, but the rows of second tables can only relate to the only row in the first table. It is also known as a **many to one** relationship.

Representation of **One to Many** relational databases:

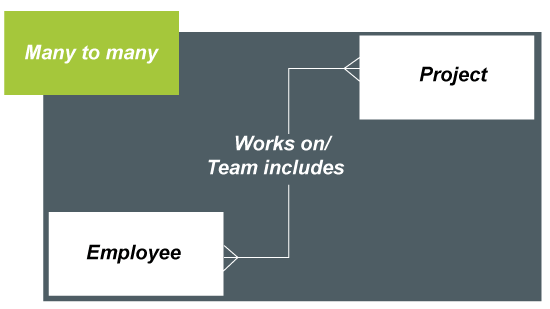


Representation of **many to one** relational database



**Many to Many Relationship:** It is **many to many** relationships that create a relationship between two tables. Each record of the first table can relate to any records (or no records) in the second table. Similarly, each record of the second table can also relate to more than one record of the first table. It is also represented an **N:N** relationship.

For example, there are **many** people involved in each **project,** and every person can involve more than one project.



### **Difference between a database and a relational database**

|  |  |
| --- | --- |
| **Relational Database** | **Database** |
| A relational database can store and arrange the data in the tabular form like rows and columns. | It is used to store the data as files. |
| The data normalization feature is available in the relational database. | It does not have a normalization. |
| It supports a distributed database. | It does not support the distributed database. |
| In a relational database, the values are stored as tables that require a primary keys to possess the data in a database. | Generally, it stores the data in the hierarchical or navigational form. |
| It is designed to handle a huge collection of data and multiple users. | It is designed to handle the small collection of data files that requires a single user. |
| A relational database uses integrity constraints rules that are defined in ACID properties. | It does not follow any integrity constraints rule nor utilize any security to protect the data from manipulation. |
| Stored data can be accessed from the relational database because there is a relationship between the tables and their attributes. | There is no relationship between data value or tables stored in files. |

### **Advantages of relational databases**

1. **Simple Model:**The simplest model of the relational database does not require any complex structure or query to process the databases. It has a simple architectural process as compared to a hierarchical database structure. Its simple architecture can be handled with simple SQL queries to access and design the relational database.
2. **Data Accuracy:**Relational databases can have multiples tables related to each other through primary and foreign keys. There are fewer chances for duplication of data fields. Therefore the accuracy of data in relational database tables is greater than in any other database system.
3. **Easy to access Data:**The data can be easily accessed from the relational database, and it does not follow any pattern or way to access the data. One can access any data from a database table using SQL queries. Each table in the associated database is joined through any relational queries such as join and conditional descriptions to concatenate all tables to get the required data.
4. **Security:** It sets a limit that allows specific users to use relational data in RDBMS.
5. **Collaborate:** It allows multiple users to access the same database at a time.

## **8. What is the primary key of a database?**

A table column with this constraint is called the key column for the table. This constraint helps the table to make sure that the value is not repeated and also that there are no null entries.

Now, this column does not allow null values and duplicate values. You can try inserting values to violate these conditions and see what happens. A table can have only one Primary key. Multiple columns can participate in the primary key.

## **9. What are database normalization forms?**

**Normalization** is the process of organizing data into a related table; it also eliminates redundancy and increases the integrity which improves performance of the query. To normalize a database, we divide the database into tables and establish relationships between the tables.

**Redundancy** means having multiple copies of the same data in the database. This problem arises when a database is not [normalized](https://www.geeksforgeeks.org/normal-forms-in-dbms/).

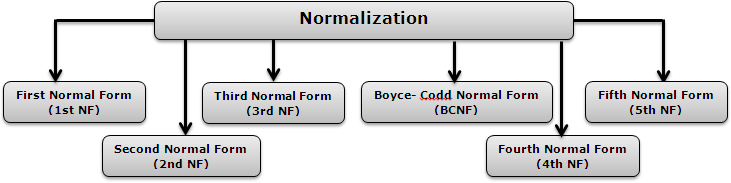
An **anomaly** is a deviation from the norm, a glitch or an error that doesn’t fit in with the rest of the pattern of the database.

## **De-Normalization**

De-normalization is the process of adding redundant data to speed up complex queries involving multiple table JOINS. One might just go to a lower form of Normalization to achieve De-normalization and better performance. Data is included in one table from another in order to eliminate the second table which reduces the number of JOINS in a query and thus achieves performance.

Normalization is a Six stage process - After the first stage, the data is said to be in first normal form, after the second, it is in second normal form, after the third, it is in third normal form and so on.

## **What are the four 4 types of database normalization?**



### **Important Points Regarding Normal Forms in DBMS**

* **First Normal Form (1NF):** This is the most basic level of normalization. In 1NF, each table cell should contain only a single value, and each column should have a unique name. The first normal form helps to eliminate duplicate data and simplify queries.
* **Second Normal Form (2NF):** 2NF eliminates redundant data by requiring that each non-key attribute be dependent on the primary key. This means that each column should be directly related to the primary key, and not to other columns.
* **Third Normal Form (3NF):** 3NF builds on 2NF by requiring that all non-key attributes are independent of each other. This means that each column should be directly related to the primary key, and not to any other columns in the same table.
* **Boyce-Codd Normal Form (BCNF):** BCNF is a stricter form of 3NF that ensures that each determinant in a table is a candidate key. In other words, BCNF ensures that each non-key attribute is dependent only on the candidate key.
* **Fourth Normal Form (4NF):** 4NF is a further refinement of BCNF that ensures that a table does not contain any multi-valued dependencies.
* **Fifth Normal Form (5NF):** 5NF is the highest level of normalization and involves decomposing a table into smaller tables to remove data redundancy and improve data integrity.

Normal forms help to reduce data redundancy, increase data consistency, and improve database performance. However, higher levels of normalization can lead to more complex database designs and queries. It is important to strike a balance between normalization and practicality when designing a database.

## **Advantages of Normal Form**

* **Reduced data redundancy:** Normalization helps to eliminate duplicate data in tables, reducing the amount of storage space needed and improving database efficiency.
* **Improved data consistency:** Normalization ensures that data is stored in a consistent and organized manner, reducing the risk of data inconsistencies and errors.
* **Simplified database design:** Normalization provides guidelines for organizing tables and data relationships, making it easier to design and maintain a database.
* **Improved query performance:**Normalized tables are typically easier to search and retrieve data from, resulting in faster query performance.
* **Easier database maintenance:** Normalization reduces the complexity of a database by breaking it down into smaller, more manageable tables, making it easier to add, modify, and delete data.

Overall, using normal forms in DBMS helps to improve data quality, increase database efficiency, and simplify database design and maintenance.

## **10. What is Schema in SQL Server?**

A **schema** is a collection of database objects like tables, triggers, stored procedures, etc. A schema is connected with a user which is known as the schema owner. Database may have one or more schema.

[SQL Server](https://www.geeksforgeeks.org/sql-tutorial/" \l "sql-server" \t "_blank) have some built-in schema, for example: dbo, guest, sys, and INFORMATION\_SCHEMA.

dbo is default schema for a new database, owned by dbo user. While creating a new user with CREATE USER command, user will take dbo as its default schema.

CREATE SCHEMA statement used to create a new schema in current database.

**Syntax:**

CREATE SCHEMA schemaname

[AUTHORIZATION ownername]

GO

**Example –**

CREATE SCHEMA geeks\_sch;

GO

**To select SQL Server SCHEMA :**  
To list all schema in the current database, use query as shown below:

SELECT \*

FROM sys.schemas

# **Q: What is Magic Tables in SQL Server?**

## **Magic Tables in SQL Server**

SQL Server allows you to define a Magic Table. Magic Tables are invisible tables or virtual tables. You can see them only with the help of[Triggers in the SQL Server](https://www.c-sharpcorner.com/UploadFile/a53f1a/triggers-in-sql-server/). Magic Tables are those tables that allow you to hold inserted, deleted and updated values during insert, delete and update DML operations on a table in SQL Server. So let's have a look at a practical example of how to use Magic Tables in SQL Server. The example is developed in SQL Server using the[SQL Server Management Studio.](https://www.c-sharpcorner.com/article/installation-way-of-sql-server-management-studio-2017/)

## **Types of Magic Tables in SQL Server**

These are the two Magic Tables

1. Inserted
2. Deleted

Generally, Magic Tables are invisible tables, we can only see them with the help of Triggers in SQL Server.

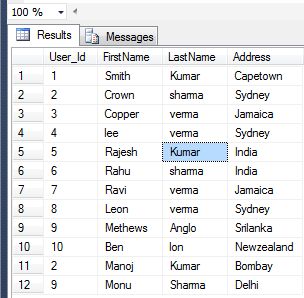
### **Use with Triggers**

If you have implemented a trigger for a table then:

1. Whenever you Insert a record on that table, that record will be shown in the INSERTED Magic Table.
2. Whenever you Delete the record on that table, that record will be shown in the DELETED Magic Table Only.
3. Whenever you Update the record on that table, that existing record will be shown in the DELETED Magic Table and Updated new data will be shown in the INSERTED Magic Table.

### **Creating Table in SQL Server**

Assume a table that looks as in the following figure.



### **Insert the Record in Table**

**Inserted Virtual Table**

The Inserted table holds the recently inserted values. Hence that record will be shown in INSERTED Magic Table. Suppose we have a UserDetails table as shown in the following figure. Now We need to create a trigger to see data within Inserted virtual tables. Now creating a trigger to see the data in the Inserted virtual table.

Create TRIGGER Trigger\_ForInsertmagic

ON [UserDetails]

FOR INSERT

AS

begin

SELECT \* FROM INSERTED

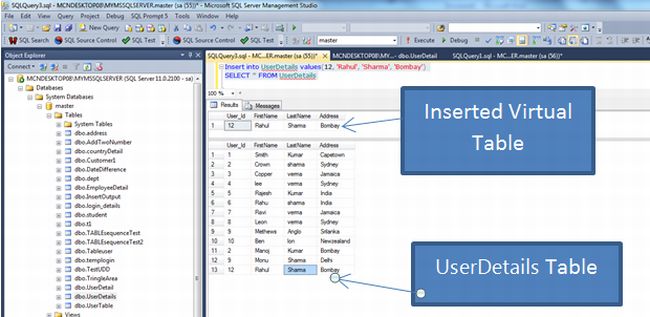
end

Now insert a new record in the UserDetails table to see data within Inserted virtual tables.

insert into UserDetails values(12, 'Rahul', 'Sharma', 'Bombay')

SELECT \* FROM UserDetails

Now press F5 to execute it.



### **Delete the Record in Table**

**Deleted Virtual Table**

The Deleted table holds the recently deleted values. Hence that record will be shown in the DELETED Magic Table. Suppose we have a UserDetails table as shown in the above figure. Now we need to create a trigger to see the data in the deleted virtual tables. To create a trigger to see the data in the deleted virtual table use the following:

Create TRIGGER Trigger\_Fordeletemagic

ON [UserDetails]

FOR DELETE

AS

begin

SELECT \* FROM Deleted

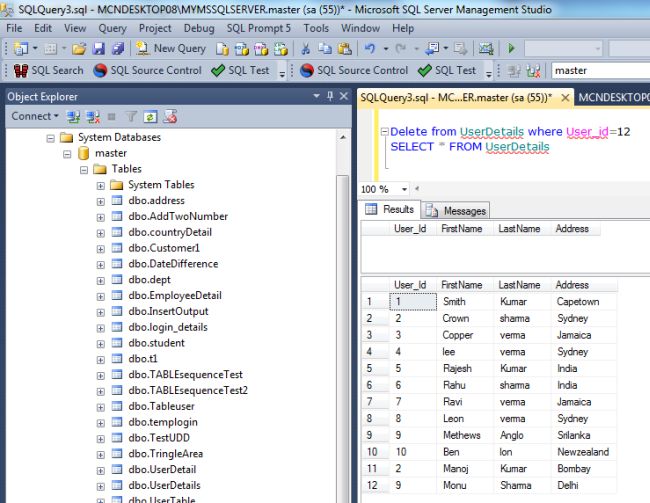
end

Now delete a record in the UserDetails table to see the data in the Deleted virtual tables.

Delete from UserDetails where User\_id=12

SELECT \* FROM UserDetails

Now press F5 to execute it.



### **Update the Record in Table**

To update the record in the UserDetails table, we use it for both virtual tables. One shows the inserted table and the other shows the deleted table. The following trigger defines both the inserted table and the deleted table:

Create TRIGGER Trigger\_ForInsertdeletemagic

ON [UserDetails]

FOR UPDATE

AS

begin

SELECT \* FROM INSERTED

SELECT \* FROM DELETED

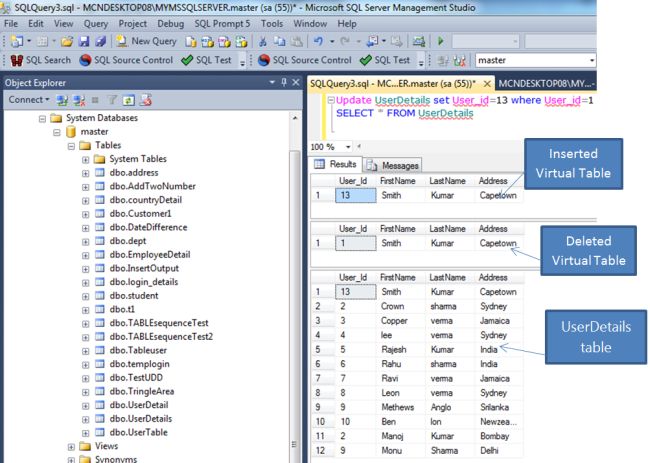
end

Now update the records in the UserDetails table to see the data in the inserted and deleted virtual tables.

Delete from UserDetails where User\_id=12

SELECT \* FROM UserDetails

Now press F5 to execute it.



# **Q: What is CTE in SQL Server?**

**A CTE (Common Table Expression) is a one-time result set that only exists for the duration of the query**. It allows us to refer to data within a single SELECT, INSERT, UPDATE, DELETE, CREATE VIEW, or MERGE statement's execution scope. It is temporary because its result cannot be stored anywhere and will be lost as soon as a query's execution is completed. It first came with [SQL Server](https://www.javatpoint.com/sql-server-tutorial) 2005 version. A DBA always preferred CTE to use as an alternative to a Subquery/View. They follow the ANSI SQL 99 standard and are SQL-compliant.

### **CTE Syntax in SQL Server**

The CTE syntax includes a CTE name, an optional column list, and a statement/query that defines the common table expression (CTE). After defining the CTE, we can use it as a view in a SELECT, INSERT, UPDATE, DELETE, and MERGE query.

The following is the basic syntax of CTE in SQL Server:

1. **WITH** cte\_name (column\_names)
2. **AS** (query)
3. **SELECT** \* **FROM** cte\_name;

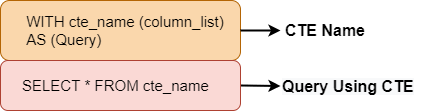
In this syntax:

* We have first specified the CTE name that will be referred to later in a query.
* The next step is to create a list of comma-separated columns. It ensures that the number of columns in the CTE definition arguments and the number of columns in the query must be the same. If we have not defined the CTE arguments' columns, it will use the query columns that define the CTE.
* After that, we'll use the AS keyword after the expression name and then define a SELECT statement whose result set populates the CTE.
* Finally, we will use the CTE name in a query such as SELECT, INSERT, UPDATE, DELETE, and MERGE statement.

It should keep in mind while writing the CTE query definition; we cannot use the following clauses:

1. ORDER BY unless you also use as TOP clause
2. INTO
3. OPTION clause with query hints
4. FOR BROWSE

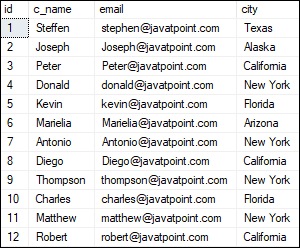
The below image is the representation of the CTE query definition.



Here, the first part is a CTE expression that contains a SQL query that can be run independently in SQL. And the second part is the query that uses the CTE to display the result.

### **Example**

Let us understand how CTE works in SQL Server using various examples. Here, we are going to use a table "**customer**" for a demonstration. Suppose this table contains the following data:

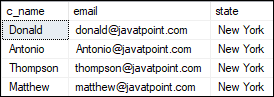


In this example, the CTE name is **customers\_in\_newyork**, the subquery that defines the CTE returns the three columns **customer name, email,** and **state**. As a result, the CTE customers\_in\_newyork will return all customers who live in New York State.

After defining the CTE customers\_in\_newyork, we have referenced it in the **SELECT** statement to get the details of those customers who are located in New York.

1. **WITH** customers\_in\_NewYork
2. **AS** (**SELECT** \* **FROM** customer **WHERE** state = 'New York')
3. **SELECT** c\_name, email, state **FROM** customers\_in\_NewYork;

After executing the above statement, it will give the following output. Here, we can see that the result returns only that customer information who are located in New York State.



## **Multiple CTE**

In some cases, we'll need to create multiple CTE queries and join them together to see the results. We may use multiple CTEs concept in this scenario. We need to use the comma operator to create multiple CTE queries and merge them into a single statement. The "," comma operator must be preceded by the CTE name to distinguish multiple CTE.

Multiple CTEs help us in simplifying complex queries that are eventually joined together. Each complex piece had its own CTE, which could then be referenced and joined outside the WITH clause.

#### **NOTE: The multiple CTE definition can be defined using UNION, UNION ALL, JOIN, INTERSECT, or EXCEPT.**

**The below syntax explains it more clearly:**

1. **WITH**
2. cte\_name1 (column\_names) **AS** (query),
3. cte\_name2 (column\_names) **AS** (query)
4. **SELECT** \* **FROM** cte\_name
5. **UNION** ALL
6. **SELECT** \* **FROM** cte\_name;

### **Example**

Let us understand how multiple CTE works in SQL Server. Here, we are going to use the above "**customer**" table for a demonstration.

In this example, we have defined the two CTE names **customers\_in\_newyork** and **customers\_in\_california**. Then the result set of subqueries of these CTEs populates the CTE. Finally, we will use the CTE names in a query that will return all customers who are located in **New York** and **California State**.

1. **WITH**
2. customers\_in\_NewYork
3. **AS** (**SELECT** \* **FROM** customer **WHERE** state = 'New York'),
4. customers\_in\_California
5. **AS** (**SELECT** \* **FROM** customer **WHERE** state = 'California')
6. **SELECT** c\_name, email, state **FROM** customers\_in\_NewYork
7. **UNION** ALL
8. **SELECT** c\_name, email, state **FROM** customers\_in\_California;

New York and California State.



## **Why do we need CTE?**

Like database views and derived tables, CTEs can make it easier to write and manage complex queries by making them more readable and simple. We can accomplish this characteristic by breaking down the complex queries into simple blocks that can reuse in rewriting the query.

**Some of its use cases are given below:**

* It is useful when we need to define a derived table multiple times within a single query.
* It is useful when we need to create an alternative to a view in the database.
* It is useful when we need to perform the same calculation multiple times on multiple query components simultaneously.
* It is useful when we need to use ranking functions like ROW\_NUMBER(), RANK(), and NTILE().

**Some of its advantages are given below:**

* CTE facilitates code maintenance easier.
* CTE increases the readability of the code.
* It increases the performance of the query.
* CTE makes it possible to implement recursive queries easily.

## **Types of CTE in SQL Server**

SQL Server divides the CTE (Common Table Expressions) into two broad categories:

1. Recursive CTE
2. Non-Recursive CTE

### **Recursive CTE**

A common table expression is known as recursive CTE that references itself. Its concept is based on recursion, which is defined as "**the application of a recursive process or definition repeatedly**." When we execute a recursive query, it repeatedly iterates over a subset of the data. It is simply defined as a query that calls itself. There is an end condition at some point, so it does not call itself infinitely.

A recursive CTE must have a **UNION ALL** statement and a second query definition that references the CTE itself in order to be recursive.

**Example**

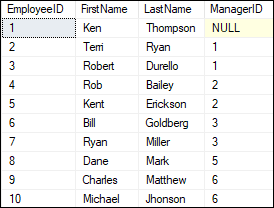
Let us understand how recursive CTE works in SQL Server. Consider the below statement, which **generates a series of the first five odd numbers:**

1. **WITH**
2. odd\_num\_cte (id, n) **AS**
3. (
4. **SELECT** 1, 1
5. **UNION** ALL
6. **SELECT** id+1, n+2 **from** odd\_num\_cte **where** id < 5
7. )
8. **SELECT** \* **FROM** odd\_num\_cte;

When we execute this recursive CTE, we will see the output as below:

CTE in SQL Server

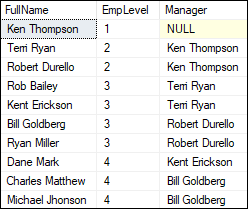
The below example is the more advanced recursive CTE. Here, we are going to use the "**jtp\_employees**" table for a demonstration that contains the below data:



This example will display the hierarchy of employee data. Here table provides a reference to that person's manager for each employee. The reference is itself an employee id within the same table.

1. **WITH** cte\_recursion (EmpID, FirstName, LastName, MgrID, EmpLevel)
2. **AS**
3. (
4. **SELECT** EmployeeID, FirstName, LastName, ManagerID, 1
5. **FROM** jtp\_employees **WHERE** ManagerID **IS** NULL
6. **UNION** ALL
7. **SELECT** emp.EmployeeID, emp.FirstName, emp.LastName, emp.ManagerID, r.EmpLevel + 1
8. **FROM** jtp\_employees emp **INNER** JOIN cte\_recursion r **ON** emp.ManagerID = r.EmpID
9. )
10. **SELECT**
11. FirstName + ' ' + LastName **AS** FullName, EmpLevel,
12. (**SELECT** FirstName + ' ' + LastName **FROM** jtp\_employees
13. **WHERE** EmployeeID = cte\_recursion.MgrID) **AS** Manager
14. **FROM** cte\_recursion **ORDER** **BY** EmpLevel, MgrID

This CTE will give the following output where we can see the hierarchy of employee data:



### **Non-Recursive CTE**

A common table expression that doesn't reference itself is known as a non-recursive CTE. A non-recursive CTE is simple and easier to understand because it does not use the concept of recursion. According to the CTE Syntax, each CTE query will begin with a "**With**" clause followed by the CTE name and column list, then AS with parenthesis.

## **Disadvantages of CTE**

The following are the limitations of using CTE in SQL Server:

* CTE members are unable to use the keyword clauses like Distinct, Group By, Having, Top, Joins, etc.
* The CTE can only be referenced once by the Recursive member.
* We cannot use the table variables and CTEs as parameters in stored procedures.
* We already know that the CTE could be used in place of a view, but a CTE cannot be nested, while Views can.
* Since it's just a shortcut for a query or subquery, it can't be reused in another query.
* The number of columns in the CTE arguments and the number of columns in the query must be the same.

SELECT TOP 5 \* FROM [INFORMATION\_SCHEMA].[COLUMNS] WHERE TABLE\_NAME='employee\_info'; --- show table column information

select \* from employee\_info

where startDate BETWEEN CAST('2021-01-01' as DATE) AND GETDATE();

UPDATE employee\_info set FirstName = 'Radha', City = 'Varanasi', Gender= 'F' Where EmployeeID = 7

It is necessary to use Group By clause with aggregate function (e.g. Count() ) to select list otherwise sql will throw an error.

SELECT City, COUNT (\*) FROM employee\_info WHERE State = 'MP' Group By City Order By City;

SELECT City, COUNT (\*) FROM employee\_info WHERE State = 'UP' Group By City HAVING COUNT (\*) > 2 Order By City;

**Order of Group By, Having and Order By Clause**

Group By 🡪 Having 🡪 Order By

If you don’t explicitly specify ASC or DESC, SQL Server uses ASC as the default sort order. Also, SQL Server treats NULL as the lowest value.

When processing the SELECT statement that has an ORDER BY clause, the ORDER BY clause is the very last clause to be processed.

SELECT

FirstName,

LastName

FROM

employee\_info

ORDER BY

FirstName DESC;

Sort a result set by multiple columns

SELECT

FirstName,

LastName,

City

FROM

employee\_info

ORDER BY

City,

FirstName;

**OFFSET, FETCH Clause**

The OFFSET and FETCH clauses are the options of the ORDER BY clause. They allow you to limit the number of rows to be returned by a query.

The OFFSET clause specifies the number of rows to skip before starting to return rows from the query.

The FETCH clause specifies the number of rows to return after the OFFSET clause has been processed.

The OFFSET clause is mandatory while the FETCH clause is optional. Also, the FIRST and NEXT are synonyms respectively so you can use them interchangeably.

**Note** that you must use the OFFSET and FETCH clauses with the ORDER BY clause. Otherwise, you will get an error.

To skip the first 4 employees and select the next 3 employees, you use both OFFSET and FETCH clauses as follows:

SELECT

FirstName,

LastName

FROM

employee\_info

ORDER BY

LastName,

FirstName

OFFSET 4 ROWS

FETCH NEXT 3 ROWS ONLY;

Group By 🡪 Having 🡪 Order By 🡪 OFFSET 🡪 FETCH

Using TOP WITH TIES to include rows that match the values in the last row

SELECT TOP 3 WITH TIES

FirstName,

LastName,

Salary

FROM

employee\_info

ORDER BY

Salary DESC

Both DISTINCT and GROUP BY clause reduces the number of returned rows in the result set by removing the duplicates.

However, you should use the GROUP BY clause when you want to apply an aggregate function on one or more columns.

SELECT

City,

State

FROM

employee\_info

GROUP BY

City, State

ORDER BY

City, State

--It is equivalent to the following query that uses the DISTINCT operator:

SELECT DISTINCT

City,

State

FROM

employee\_info;

Finding rows whose values contain a string

SELECT

FirstName,LastName,Salary,Gender,City,State

FROM

employee\_info

WHERE

FirstName LIKE '%Sh%'

ORDER BY

Salary DESC;

Using the AND operator with other logical operators

SELECT

\*

FROM

employee\_info

WHERE

State='UP' or Salary>400000 and Gender='F'

ORDER BY

Salary DESC;

--In this example, we used both OR and AND operators in the condition.

--As always, SQL Server evaluated the AND operator first. However, you can use the parentheses to change the order of the evaluation.

SELECT

FirstName,LastName,Salary,Gender,State

FROM

employee\_info

WHERE

(Salary<300000 or Salary>600000) and Gender='F'

ORDER BY

Salary DESC;

The IN operator is equivalent to multiple OR operators.

The IN operator is a logical operator that allows you to test whether a specified value matches any value in a list.

Using SQL Server IN with a list of values example

SELECT

FirstName,LastName,Salary,Gender

FROM

employee\_info

WHERE

Salary IN (400000,500000,800000)

ORDER BY

Salary DESC;

To find the employees whose salaries are not one of the salaries above, you use the NOT IN operator as shown in the following query:

SELECT

FirstName,LastName,Salary,Gender

FROM

employee\_info

WHERE

Salary NOT IN (400000,500000,800000)

ORDER BY

Salary DESC;

Using SQL Server IN operator with a subquery example

SELECT

FirstName,LastName,Salary,Gender

FROM

employee\_info

WHERE

EmployeeID IN (

SELECT

EmployeeID

FROM

employee\_info

WHERE

Salary>600000

)

ORDER BY

FirstName;

The following statement retrieves the employees whose last name starts with the letter v and ends with the letter a:

SELECT

EmployeeID,FirstName,LastName,Salary,Gender

FROM

employee\_info

WHERE

LastName Like 'v%a'

ORDER BY

FirstName DESC;

If you want to search for 30% in the comment column

SELECT

feedback\_id, comment

FROM

sales.feedbacks

WHERE

comment LIKE '%30!%%' ESCAPE '!';

In this query, the ESCAPE clause specified that the character ! is the escape character.

It instructs the LIKE operator to treat the % character as a literal string instead of a wildcard.

Note that without the ESCAPE clause, the query would return an empty result set.

**Q:** **What are web services and why should we use them?**  
**Web services are a standardized way for developing interoperable applications i.e enabling an application to invoke a method of another application.** These applications can be on the same computer or different computers. Web services use open standards and protocols like **HTTP**, **XML**and **SOAP.**Since these are open and well known protocols, applications built on any platform can interoperate with web services. For example, a JAVA application can interoperate with a web service built using .NET. Similarly a web service built using JAVA can be consumed by a .NET application.  
  
**Hyper Text Transfer Protocol (HTTP)**is the protocol widely used by web services to send and receive messages.  
The messaging protocol is **SOAP.**SOAP stands for Simple Object Access Protocol. SOAP messages are in XML format.

----SQL Alias----

--SQL Aliases can be used to create a temporary name for columns and tables.

------SQL Server column alias-----

--To assign a column or an expression a temporary name during the query execution, you use a column alias.

SELECT

FirstName + ' ' + LastName AS 'Full Name'

FROM

employee\_info

ORDER BY

FirstName;

SELECT

FirstName, Salary 'Employee Salary'

FROM

employee\_info

ORDER BY

'Employee Salary';

SELECT

EmployeeID As Id, FirstName, LastName

From

employee\_info

---------------------------SQL Server table alias--------------------------------

----TABLE ALIASES: are used to shorten SQL query to make it easier to read or when there are more than one table is involved.

-----------Table Alias Example--------------------------

SELECT

E.Emp\_ID, E.Name, D.Dept\_ID, D.D\_Name

FROM

Employees AS E, Dept AS D

WHERE

E.Dept\_ID=D.Dept\_ID;

--In this query, E is the alias for the Employees table and D is the alias for the Dept table.

----------------SQL Server Inner Join(Like an intersection in Math)-------------------

-- Inner join selects matching records from both tables.

-- Here hr is a schema of the table’s candidates and employees.

SELECT

c.id candidate\_id,

c.fullname candidate\_name,

e.id employee\_id,

e.fullname employee\_name

FROM

hr.candidates c

INNER JOIN hr.employees e

ON e.fullname = c.fullname;

------------------------SQL Server Left Join----------------------------------

--Left join selects data starting from the left table and matching rows in the right table.

--The left join returns all rows from the left table and the matching rows from the right table.

--If a row in the left table does not have a matching row in the right table,

--the columns of the right table will have nulls.

--The left join is also known as the left outer join. The outer keyword is optional.

SELECT

c.id candidate\_id,

c.fullname candidate\_name,

e.id employee\_id,

e.fullname employee\_name

FROM

hr.candidates c

LEFT JOIN hr.employees e

ON e.fullname = c.fullname;

--To get the rows that are available only in the left table but not in the right table, you add a WHERE clause to the above query:

SELECT

c.id candidate\_id,

c.fullname candidate\_name,

e.id employee\_id,

e.fullname employee\_name

FROM

hr.candidates c

LEFT JOIN hr.employees e

ON e.fullname = c.fullname

WHERE

e.id IS NULL;

--------------------------SQL Server Right Join----------------------------

--The right join or right outer join selects data starting from the right table. It is a reversed version of the left join.

--The right join returns a result set that contains all rows from the right table and the matching rows in the left table.

--If a row in the right table does not have a matching row in the left table, all columns in the left table will contain nulls.

SELECT

c.id candidate\_id,

c.fullname candidate\_name,

e.id employee\_id,

e.fullname employee\_name

FROM

hr.candidates c

RIGHT JOIN hr.employees e

ON e.fullname = c.fullname;

--Notice that all rows from the right table (employees) are included in the result set.

--To get the rows that are available only in the right table but not in the left table, you add a WHERE clause to the above query:

SELECT

c.id candidate\_id,

c.fullname candidate\_name,

e.id employee\_id,

e.fullname employee\_name

FROM

hr.candidates c

RIGHT JOIN hr.employees e

ON e.fullname = c.fullname

WHERE

c.id IS NULL;

--------------------------------SQL Server full join----------------------------------

--The full outer join or full join returns a result set that contains all rows from both left and right tables,

--with the matching rows from both sides where available. In case there is no match, the missing side will have NULL values.

SELECT

c.id candidate\_id,

c.fullname candidate\_name,

e.id employee\_id,

e.fullname employee\_name

FROM

hr.candidates c

FULL JOIN hr.employees e

ON e.fullname = c.fullname;

--To select rows that exist either left or right table, you exclude rows that are common to both tables by adding a WHERE clause as shown in the following query:

SELECT

c.id candidate\_id,

c.fullname candidate\_name,

e.id employee\_id,

e.fullname employee\_name

FROM

hr.candidates c

FULL JOIN hr.employees e

ON e.fullname = c.fullname

WHERE

c.id IS NULL OR

e.id IS NULL;

--------------------------------SQL Server Cross Join--------------------------------

--The CROSS JOIN joined every row from the first table (T1) with every row from the second table (T2).

--In other words, the cross join returns a Cartesian product of rows from both tables.

--Unlike the INNER JOIN or LEFT JOIN, the cross join does not establish a relationship between the joined tables.

--The CROSS JOIN gets a row from the first table (T1) and then creates a new row for every row in the second table (T2).

--It then does the same for the next row for in the first table (T1) and so on.

--if the first table has n rows and the second table has m rows, the cross join will result in n x m rows.

--Note that GO command instructs the SQL Server Management Studio to send the SQL statements up to the GO statement to the server to be executed.

SELECT

Name, DeptName

From

crosses.employee CROSS JOIN crosses.department;

SELECT \* FROM crosses.department, crosses.employee;

-------------------------------SQL Server self join-------------------------------------

--A self join allows you to join a table to itself. It helps query hierarchical data or compare rows within the same table.

--A self join uses the inner join or left join clause. Because the query that uses the self join references the same table,

--the table alias is used to assign different names to the same table within the query.

--To get who reports to whom, you use the self join as shown in the following query:

SELECT

e.first\_name + ' ' + e.last\_name employee,

m.first\_name + ' ' + m.last\_name manager

FROM

sales.staffs e INNER JOIN sales.staffs m ON m.staff\_id = e.manager\_id

ORDER BY

manager;

--In this example, we referenced to the staffs table twice: one as e for the employees and the other as m for the managers. The join predicate matches employee and manager relationship using the values in the e.manager\_id and m.staff\_id columns.

--------------------------Control-of-flow statements----------------------------------

/\*

BEGIN...END statement:-

The BEGIN...END statement is used to define a statement block. A statement block consists of a set of SQL statements that execute together. A statement block is also known as a batch.

\*/

BEGIN

SELECT

FirstName,

Salary

FROM

employee\_info

WHERE

Salary > 800000;

IF @@ROWCOUNT = 0

PRINT 'No employee with salary greater than 800000 found';

END

/\*

To view the messages generated by the PRINT statement, in SQL Server Management Studio, you need to click the Messages tab.

By default, the Messages tab is hidden.

Note that the @@ROWCOUNT is a system variable that returns the number of rows affected by the last previous statement.

\*/

------------------------SQL Server Loops---------------------------------

/\* Looping structures allow a single command, or a group of statements, to be executed repeatedly until the specific condition result turn out to be false.

Types of Loops:

DO...WHILE Loop

FOR LOOP

WHILE LOOP:- SQL Server implements the WHILE Loop allowing us to repeat a certain code while the loop condition holds.

SQL Server only have while Loop. There is no For Loop and Do While Loop in SQL Server (Transact-SQL). You can simulate the FOR LOOP in SQL Server (Transact-SQL) using the WHILE LOOP.

Syntax

The syntax to simulate the FOR Loop in SQL Server (Transact-SQL) is:

DECLARE @cnt INT = 0;

WHILE @cnt < cnt\_total

BEGIN

{...statements...}

SET @cnt = @cnt + 1;

END;

cnt\_total:- The number of times that you want the simulated FOR LOOP (ie: WHILE LOOP) to execute.

statements:- The statements of code to execute each pass through the loop.

\*/

DECLARE @count INT;

SET @count = 10;

While (@count <= 15)

BEGIN

Print @count;

SET @count = @count + 1;

END

Print 'End Of While Loop'

DECLARE @count INT;

DECLARE @TableNO INT;

SET @count = 1;

SET @TableNO = 8;

while @count <=10

BEGIN

Print concat(@TableNo,' x ', @count, ' = ', @TableNO \* @count);

SET @count = @count + 1;

End

Print 'End Of While Loop'

--Table with break statement

DECLARE @count INT;

DECLARE @TableNO INT;

SET @count = 1;

SET @TableNO = 19;

while @count <=10

BEGIN

Print concat(@TableNo,' x ', @count, ' = ', @TableNO \* @count);

SET @count = @count + 1;

if @count =5

break;

End

Print 'End Of While Loop'

--------NESTED WHILE LOOP---------

DECLARE @TableNO INT;

SET @TableNO = 15;

while @TableNO <= 20

BEGIN

DECLARE @InnerCount int =1;

while @InnerCount <=10

Begin

Print CONCAT(@TableNo,' x ', @InnerCount, ' = ', @TableNO \* @InnerCount);

SET @InnerCount = @InnerCount + 1;

End

SET @TableNO = @TableNO + 1;

Print ''

End

Print 'End Of Nested Loop'

/\* SQL CASE STATEMENT \*/

-----Case Statement in SELECT Statement-----

Select \*,

JOB\_ROLE =

Case

WHEN Salary > 600000 THEN 'Manager'

WHEN Salary > 400000 AND Salary <= 600000 THEN 'Snr Developer'

WHEN Salary >=200000 AND Salary <= 400000 THEN 'Jnr Developer'

Else 'No Role'

End

from employee\_info

---If Else Condition Statement-----------------------

DECLARE @RollNo int;

DECLARE @ENGLISH INT;

DECLARE @MATHS INT;

DECLARE @SCIENCE INT;

DECLARE @TOTAL INT;

DECLARE @PERCENTAGE INT;

---INITILIAZE VARIABLSES

SET @RollNo = 99;

SET @ENGLISH = 45;

SET @MATHS = 50;

SET @SCIENCE = 56;

SET @TOTAL = @ENGLISH+@MATHS+@SCIENCE;

SET @PERCENTAGE = @TOTAL/3;

PRINT Concat('RollNo: ',@RollNo);

PRINT Concat ('Total: ',@TOTAL);

PRINT Concat('Percentage: ',@PERCENTAGE);

IF @PERCENTAGE > 80

BEGIN

PRINT 'GRADE A';

END

ELSE IF @PERCENTAGE > 60 AND @PERCENTAGE <=80

BEGIN

PRINT 'GRADE B';

END

ELSE IF @PERCENTAGE > 40 AND @PERCENTAGE <= 60

BEGIN

PRINT 'GRADE C';

END

ELSE

BEGIN

PRINT 'FAIL';

END

// Output

RollNo: 99

Total: 151

Percentage: 50

GRADE C

IF EXISTS (SELECT EmployeeID from employee\_info where EmployeeID =14)

BEGIN

PRINT 'EmployeeID EXISTS';

END

ELSE

BEGIN

PRINT ' EmployeeID DOES NOT EXIST ';

END

/\* SQL Server Exceptions \*/

/\*

System Defined Exceptions:-

In a system defined exception, the exceptions (errors) are generated by the system.

\*/

Declare @val1 int;

Declare @val2 int;

BEGIN TRY

Set @val1=8;

Print 'Line 1';

Print 'Line 2';

Set @val2=@val1/0; /\* Error Occur Here \*/

Print 'Line 3';

Print 'Line 4';

END TRY

BEGIN CATCH

Print 'Inside the Catch block';

Print Concat('ERROR\_MESSAGE()=',ERROR\_MESSAGE());

Print Concat('ERROR\_LINE()=',ERROR\_LINE());

Print Concat('ERROR\_NUMBER()=',ERROR\_NUMBER());

Print Concat('ERROR\_SEVERITY()=',ERROR\_SEVERITY());

Print Concat('ERROR\_STATE()=',ERROR\_STATE());

Print Concat('ERROR\_PROCEDURE()=',ERROR\_PROCEDURE());

END CATCH

// Output

Line 1

Line 2

Inside the Catch block

ERROR\_MESSAGE()=Divide by zero error encountered.

ERROR\_LINE()=7

ERROR\_NUMBER()=8134

ERROR\_SEVERITY()=16

ERROR\_STATE()=1

ERROR\_PROCEDURE()=

/\*

User Defined Errors

The THROW statement in SQL Server raises an exception and transfers the control to a CATCH block.

Syntax:

THROW @errorNumber,@errorMessage, @errorSeverity

Error No-> 50000 to 2147483647

ERROR Severity

13-> Indicates transaction deadlock errors.

14-> Indicates security-related errors, such as permission denied.

15-> Indicates syntax errors in the Transact-SQL command.

16-> Indicates general errors that can be corrected by the user

\*/

Declare @Age int;

BEGIN TRY

Set @Age=60;

Print 'Line 1';

Print 'Line 2';

IF @Age > 40

THROW 50000,'Age is greater than 40',16;

Print 'Line 3';

Print 'Line 4';

END TRY

BEGIN CATCH

Print 'Inside the Catch block';

Print Concat('ERROR\_MESSAGE()=',ERROR\_MESSAGE());

Print Concat('ERROR\_LINE()=',ERROR\_LINE());

Print Concat('ERROR\_NUMBER()=',ERROR\_NUMBER());

Print Concat('ERROR\_SEVERITY()=',ERROR\_SEVERITY());

Print Concat('ERROR\_STATE()=',ERROR\_STATE());

Print Concat('ERROR\_PROCEDURE()=',ERROR\_PROCEDURE());

END CATCH

// Output

Line 1

Line 2

Inside the Catch block

ERROR\_MESSAGE()=Age is greater than 40

ERROR\_LINE()=7

ERROR\_NUMBER()=50000

ERROR\_SEVERITY()=16

ERROR\_STATE()=16

ERROR\_PROCEDURE()=

/\*

FUNCTIONS IN PROGRAMMING

A function is a block of code that performs a specific task.

Functions usually "Take in" data, process it, and "return" a result.

Once a function is written, it can be used over and over again, it means functions can be reused.

FUNCTIONS IN SQL SERVER

SQL Server Functions are useful objects in SQL Server databases.

A function is a set of SQL statements that perform a specific task.

A SQL Server function is a code snippet that can be executed on a SQL Server.

If you have to repeatedly write large SQL scripts to perform the same task, you can create a function that performs that task.

Next time instead of rewriting the SQL, you can simply call that function.

A function accepts inputs in the form of parameters and returns a value.

SQL Server comes with a set of built-in functions that perform a variety of tasks.

In SQL Server, a function is a STORED PROGRAM that you can pass parameters into and return a value.

Ofcourse, you could create a stored procedure to group a set of SQL statements and execute them, however, stored procedures cannot be called within SQL statements.

Therefore, if you are using functions with large data sets, you can hit performance issues.

In T-SQL, a function is considered an object. Here are some of the rules when creating functions in SQL Server.

\* A function must have a name and a function name can never start with a special character such as @, $, #, and so on.

\* Functions only work with SELECT Statement.

\* Functions can be used anywhere in SQL, like AVG, COUNT, SUM, MIN, DATE and so on with select statements.

\* Functions compile every time.

\* Functions must return a value or result.

\* Functions only work with input parameters.

\* TRY and CATCH statements are not used in functions.

SQL SERVER FUNCTIONS TYPES

SQL Server supports two types of functions - user defined and system.

\* USER DEFINED FUNCTION: User defined functions are created by a user.

\* SYSTEM DEFINED FUNCTION: System functions are built in database functions.

There are three types of user-defined functions in SQL Server.

1. Scalar Functions

2. Inline Table Valued Functions

3. Multi-Statement Table Valued Functions

WHAT ARE SCALER FUNCTIONS

SQL Server scalar function takes one or more parameters and returns a single(scalar) value.

The returned value can be of any data type, except text, ntext, image, cursor and timestamp.

\*/

/\* Create a function without Parameter \*/

CREATE Function ShowMessage()

Returns varchar(100)

AS

BEGIN

return 'Welcome To Function'

END

Select dbo.ShowMessage();

/\* Create a function with a single Parameter \*/

CREATE Function TakeANumber(@num as int)

Returns int

AS

BEGIN

return (@num \* @num)

END

Select dbo.TakeANumber(5);

/\* Create a function with multiple Parameters \*/

CREATE Function Addition(@num1 as int, @num2 as int)

Returns int

AS

BEGIN

return (@num1 + @num2)

END

SELECT dbo.Addition(6,5);

SELECT dbo.Addition(4,5);

--------- Scalar function can call other functions ---------------

Create function GetMyDate()

returns DateTime

AS

BEGIN

return GetDate();

END

Select dbo.GetMyDate();

# Should I use CAST or CONVERT?

Unless you have some specific formatting requirements you're trying to address during the conversion. I would stick with using the CAST function. There are several reasons I can think of:

\* CAST is ANSI-SQL compliant; therefore, more apt to be used in other database implementation.

\* There is no performance penalty using CAST.

ANSI: - American National Standards Institute

SELECT EmployeeID, CAST(StartDate AS Varchar(12)) as join\_date, FirstName, LastName from employee\_info;

SELECT EmployeeID, CONVERT(Varchar(12), StartDate) as join\_date, FirstName, LastName from employee\_info;

/\*

INLINE TABLE VALUED FUNCTIONS

Contains a single T-SQL statement and returns a Table set.

SCALER FUNCTION: It returns a scalar value.

INLINE TABLE VALUED FUNCTIONS: It returns a table.

Steps to create Inline Table Valued Functions:-

Step 1: We have to specify TABLE as the return type, instead of any scalar data type like int, varchar etc.

Step 2: There is no BEGIN and END Blocks.

Step 3: The table that gets returned, is determined by the SELECT command within the function.

\*/

Create function fn\_GetStudents()

returns table

as

return (select \* from Student\_details);

select \* from fn\_GetStudents();

-------- With Parameter ----------------

Create function fn\_GetStudentsWithRollNo(@rollNO INT)

returns table

as

return (select \* from Student\_details Where RollNO >= @rollNO);

--drop function fn\_GetStudentsWithClass;

select \* from fn\_GetStudentsWithRollNo(5);

/\*

Multi-Statement Table Valued Functions

\* A multi-statement table valued function is a table-valued function that returns the result of multiple statements.

\* The multi-statement table-valued function is very useful because you can execute multiple queries within the function and aggregate results into the returned table.

\* To define a multi-statement table-valued function, you use a table variable as the return value. Inside the function, you execute one or more queries and insert data into this table variable.

Difference between Inline Table Valued Function and Multi-statement Table Valued Functions

INLINE TABLE-VALUED FUNCTIONS

\* In this, the returns clause cannot contain the structure of the table.

\* In this, there are no BEGIN and END Blocks.

\* Inline table-valued functions are better in performance as compared to multi-statement table-valued functions.

\* Internally, Inline table-valued function much like it would a view.

MULTI-STATEMENT TABLE-VALUED FUNCTIONS

\* In this, we specify the structure of the table with returns clause.

\* In this, we have to use BEGIN and END blocks.

\* There is no performance advantage in multi-statement table-valued functions.

\* Internally, multi-statement table-valued function much like it would a stored procedure.

SIMILARITIES

\* Inline statement table-valued functions and multi-statement table-valued functions both are table-valued functions.

\* Inline statement table-valued functions and multi-statement table-valued functions both are located in Table-Valued functions folder in SSMS.

\* Both are the types of user-defined functions in SQL Server.

\*/

--MULTI-STATEMENT TABLE-VALUED FUNCTIONS

Create function fn\_GetEmployeesByGender(@gender varchar(1))

returns @myTable table (empID int, empName varchar(50), gender varchar(1))

as

BEGIN

insert into @myTable

select EmployeeID, FirstName, Gender from employee\_info WHERE Gender = @gender

return

END

select \* from fn\_GetEmployeesByGender('F');

--INLINE TABLE-VALUED FUNCTIONS

Create function fn\_GetEmployeesByGender2(@gender varchar(1))

returns table

as

return

(select EmployeeID, FirstName, Gender from employee\_info WHERE Gender = @gender)

select \* from fn\_GetEmployeesByGender2('M');

/\*

Ques: - What is an Aggregate Function in SQL?

Ans: - An aggregate function in SQL returns one value after calculating multiple values of a column. We often use aggregate functions with the GROUP BY and HAVING clauses of the SELECT statement. SQL provides many aggregate functions that include avg, count, sum, min, max, etc.

An aggregate function ignores NULL values when it performs the calculation, except for the count function.

When using aggregate functions in SQL, it is crucial to understand column references.

A column reference is a name containing the data you want to aggregate. To use an aggregate function with a column reference, specify the column's name in the function's parentheses.

For example, to find the average salary of employees in a table called "employees",

you would use the AVG function with the column reference "salary" like this:

SELECT AVG (salary)

FROM employees;

There are 5 types of SQL aggregate functions:

Count(), Sum(), Avg(), Min(), Max()

\*/

-----------------------------------INDEXES--------------------------------------------

/\*

Indexes are special data structures associated with tables or views that help speed up the query. Indexes increases the search performance.

Search becomes faster because of Balance tree structure. Internally it creates Node and Leaf Nodes to reach to the data quickly.

WHAT IS INDEX?

\* An index is a pointer to data in a table.

\* An index in a database is very similar to an index in the back of a book.

\* An index helps to speed up SELECT queries and WHERE clauses.

\* Indexes can be created or dropped with no effect on the data.

TYPES OF INDEXES

1) CLUSTERED INDEX:-

\* Each table has only one clustered index because data rows can be only sorted in one order.

\* A clustered index is a special index which physically orders the data according to the indexed columns.

\* The leaf nodes of the index store the data for the rest of the columns in the table.

2) NON-CLUSTERED INDEX:-

\* A table may have one or more non-clustered.

\* A non-clustered index is just like the index of a book.

\* It points back to the actual page that contains the data. (In other words, it points back to the clustered index)

\*/

Create Table EmployeesIndex

(

Id int primary key identity,

[Name] nvarchar(50),

Email nvarchar(50),

Department nvarchar(50)

)

SET NOCOUNT ON

Declare @counter int = 1

While(@counter <= 10000)

Begin

Declare @Name nvarchar(50) = 'ABC ' + RTRIM(@counter)

Declare @Email nvarchar(50) = 'abc' + RTRIM(@counter) + '@startech.com'

Declare @Dept nvarchar(10) = 'Dept ' + RTRIM(@counter)

Insert into EmployeesIndex values (@Name, @Email, @Dept)

Set @counter = @counter +1

If(@Counter%1000 = 0)

Print RTRIM(@Counter) + ' rows inserted'

End

----In SQL Server Management Studio click on Include Actual Execution Plan icon and then execute the following query

Select \* from EmployeesIndex where Id = 9320

/\*

---------Non-Clustered Index in SQL Server-----------

In non-clustered index we do not have table data. We have key values and row locators.

We created a non-clustered index on the Name column, so the key values, in this case Employee names are sorted and stored in alphabetical order.

The row locators at the bottom of the tree contain Employee Names and cluster key of the row. In our example, Employee Id is the cluster key.

\*/

USE [DB\_Programming]

GO

CREATE NONCLUSTERED INDEX IX\_EmployeesIndex\_Name

ON [dbo].[EmployeesIndex] ([Name])

GO

----Execute the following query again with Include Actual Execution Plan turned ON

Select \* from EmployeesIndex Where Name = 'ABC 9320'

/\*

NON CLUSTERED INDEX

\* A non-clustered index doesn't sort the physical data inside the table.

\* In fact, a non-clustered index is stored at one place and table data is stored in another place.

\* This is similar to a textbook where the book content is located in one place and the index is located in another.

\* This allows for more than one non-clustered index per table.

SQL SERVER UNIQUE INDEX

\* A unique index ensures the index key columns do not contain any duplicate values.

\* A unique index may consist of one or many columns.

\* A unique index can be clustered or non-clustered.

WHERE TO APPLY INDEX

\* Indexes are meant to speed up the performance of a database, so use indexing whenever it significantly improves the performance of your database.

\* Check query and find reason for slow performance.

\* Find column in query which is used frequently for searching.

DISADVANTAGES OF INDEXING

\* In case of update (change in indexed column) and delete a record, the database might need to move the entire row into row into a new position to keep the rows in sorted order.

\*/

------------------------------------- TRIGGERS -------------------------------------

/\*

/\* What is Trigger \*/

\* Trigger is an Event-driven T-SQL Programming block. It runs automatically when a particular event occurs.

\* 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:

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

2) 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.

3) Logon triggers which fire in response to LOGON events

INSERT, UPDATE, DELETE --> DML Command --> DML TRIGGER

CREATE, ALTER, DROP --> DDL Command --> DDL TRIGGER

/\* Introduction to SQL Server CREATE TRIGGER statement \*/

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}

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

\*/

select \* from Employees;

/\*\*\*\*\*\*\*\*\*\*\*\* Audit Table \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

CREATE TABLE Employees\_Audit (

Emp\_ID INT,

Inserted\_By Varchar(100)

)

Go

/\*\*\*\*\*\*\*\*\*\*\*\* Creating TRIGGER For INSERT EVENT \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

CREATE TRIGGER TRG\_Insert\_Audit

On Employees

FOR INSERT

AS

BEGIN

Declare @emp\_ID int

Select @emp\_ID = Emp\_ID from inserted

insert into Employees\_Audit(Emp\_ID, Inserted\_By)

Values (@emp\_ID, ORIGINAL\_LOGIN())

PRINT 'Insert Trigger Executed Successfully'

END

GO

--------------- Insert a new data in Employees Table ------------------------

INSERT INTO Employees(Name,Age,Address,Salary,Dept\_ID) Values

('Arjun',24,'Jammu',400000,3)

Select \* from Employees;

Select \* from Employees\_Audit;

CREATE TRIGGER TRG\_Delete\_Rule

ON Employees

FOR DELETE

AS

BEGIN

RollBack

PRINT '\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*'

PRINT 'You can not delete records from this table'

PRINT '\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*'

END

GO

Delete from Employees Where Emp\_ID = 14;

SELECT \* From Employees;

/\*\*\*\*\*\*\*\*\*\*\*\* DDL TRIGGER \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

CREATE TRIGGER Trg\_SampleDB

ON DATABASE

FOR CREATE\_TABLE

AS

BEGIN

ROLLBACK

PRINT 'You are not allowed to create tables'

END

Create Table Tbl1 (ID INT)

-- DROP TRIGGER Trg\_SampleDB ON DATABASE;

/\*\*\*\*\*\*\*\*\*\*\*\* List All Triggers \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

Select \* from sys.triggers

------------------------------SQL Stored Procedures-----------------------------------

/\*

A Stored Procedure is a database object.

A stored Procedure is a series of declarative SQL statements.

A stored procedure is a prepared SQL code that you can save, so the code can be reused over and over again.

It is a pre-compiled code of SQL statement.

A stored procedure can be stored in the DB and can be reused over & over again.

Parameters can be passed to a stored procedure, so that the stored procedure can act based on the parameter value(s).

SQL server creates an execution plan & stores it in the cache.

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.

\*/

------How to create stored procedure-------

Create Procedure spEmployeeDetails

AS

BEGIN

SELECT \* from employee\_info where City='Jaunpur';

END

----drop proc spEmployeeDetails

-------------How to execute/call stored procedure---------------------

spEmployeeDetails

Execute spEmployeeDetails

EXEC spEmployeeDetails

Create Procedure spEmployeeDetails2

AS

BEGIN

SELECT \* from Employees where Dept\_ID=1;

END

EXEC spEmployeeDetails2

------How to Modified Stored Procedure-------

ALTER Proc spEmployeeDetails2

AS

BEGIN

SELECT \* from Employees where Dept\_ID=1;

SELECT \* from Employees where Dept\_ID=2;

END

EXEC spEmployeeDetails2

-----------------Parameters in STORED PROCEDURE------------------

------ Two Types: - (1) Input Parameter & (2) Output Parameter

Create Proc spEmployeeDetails2

@dept\_ID int,

@name Varchar(100)

AS

BEGIN

SELECT \* from Employees where Dept\_ID=@dept\_ID;

SELECT \* from Employees where Name=@name;

END

spEmployeeDetails2 1, 'Shubham'

---- Order of parameters and value is very important in SP.

----Named Parameter value

spEmployeeDetails2 @name = 'Shubham', @dept\_ID = 2

----In this case order of parameter is not important

--OUTPUT PARAMETER

Create Proc spAddDigits

@num1 INT,

@num2 INT,

@result INT OUTPUT

AS

BEGIN

SET @result = @num1 + @num2;

END

Declare @var money

EXEC spAddDigits 27, 23, @Var OUTPUT;

SELECT @var

sp\_helptext spAddDigits --- To display SP Texts

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

CREATE PROCEDURE uspFindEmployeeByGender (

@Gender VARCHAR(1),

@employee\_count INT OUTPUT

)

AS

BEGIN

SELECT

FirstName,

Salary

FROM

employee\_info

WHERE

Gender = @Gender;

SELECT @employee\_count = @@ROWCOUNT;

END;

----Note that the @@ROWCOUNT is a system variable that returns the number of rows read by the previous statement.

/\* Calling stored procedures with output parameters \*/

--First, declare the @count variable to hold the value of the output parameter of the stored procedure:

DECLARE @count INT;

--Then, execute the uspFindEmployeeByGender stored procedure and passing the parameters:

EXEC uspFindEmployeeByGender

@Gender = 'M',

@employee\_count = @count OUTPUT;

--Finally, show the value of the @count variable:

SELECT @count AS 'Number of employees found';

**Differences between CTE and View**

**CTE:**

CTE stands for Common Table expressions can be thought of as a temporary result set that is defined within the execution scope of a single SELECT, INSERT, UPDATE, DELETE, or CREATE VIEW statement. A CTE is like a derived table in that it is not stored as an object and lasts only for the duration of the query. Unlike a derived table, a CTE can be self-referencing and can be referenced multiple times in the same query. CTE improves readability and ease in maintenance of complex queries and sub-queries.

A CTE can be used to:

1.            Create a recursive query. For more information, see Recursive Queries Using Common Table Expressions.

2.            Substitute for a view when the general use of a view is not required; that is, you do not have to store the definition in metadata.

3.            Enable grouping by a column that is derived from a scalar sub select, or a function that is either not deterministic or has external access.

**Syntax structure for a CTE:**

WITH expression\_name [ ( column\_name [,...n] ) ]

AS  
  
( CTE\_query\_definition )

The list of column names is optional only if distinct names for all resulting columns are supplied in the query definition.

The statement to run the CTE is:

SELECT <column\_list>

FROM expression\_name;

**View:**

A view is a virtual table which doesn’t physically store any data, it consists of columns from one or more tables. So, whenever we query a view then it retrieves data from the underlying base tables. It is a query stored as an object. Views are used for security purpose in databases, views restrict the user from viewing certain column and rows means by using view we can apply the restriction on accessing the rows and columns for specific user. Views display only those data which are mentioned in the query, so it shows only data which is returned by the query that is defined at the time of creation of the View.

**Syntax structure for a View:**

CREATE VIEW view\_name AS

  SELECT columns

  FROM tables

  [WHERE conditions];

**Advantages of CTE over View are:**

Common Table Expressions offer the same functionality as a view, but are ideal for one-off usages where you don't necessarily need a view defined for the system. Even when a CTE is not necessarily needed, it can improve readability.

Using a CTE offers the advantages of improved readability and ease in maintenance of complex queries. The query can be divided into separate, simple, logical building blocks. These simple blocks can then be used to build more complex, interim CTEs until the result set is generated.

# What is the difference between varchar and nvarchar?

**Varchar** and **nvarchar** are variable-length character data types, which are used for declaring the data type of the variables used in the SQL server table. Although both of them serve the same purpose, there are still a few differences between them.

Here are a few important differences between varchar and nvarcharin the SQL server.

### Character data type

* Varchar stores Non-unicode or English character data types, and it can contain a maximum of 8000 characters. It only supports ASCII values.
* Nvarchar stores Unicode or Non-English character data types, and it can contain a maximum of 4000 characters. It supports ASCII values as well as special characters. To support multiple languages, nvarchar is a must.

### Literals

* Literals or values in varchar are enclosed within single inverted commas. For example:

Example of Varchar

INSERT INTO tableName VALUES ('Educative')

* Literals or values in nvarchar are also enclosed within single inverted commas, but they are prefixed with N. For example:

Example of NVarchar

INSERT INTO tableName VALUES (N'Educative')

### Storage

* Each character in thevarchar datatype occupies one byte of storage.
* Each character in thenvarchar datatype occupies two bytes of storage.

Thus, in terms of storage, nvarchar is twice as costly when compared to varchar.

To count the number of persons belonging to a particular age group in SQL Server, you can use a **SELECT** statement with a **WHERE** clause to filter the age group, and then use the **COUNT** function. Here's an example query:

DECLARE @AgeGroupToCount INT = YourAgeGroup;

-- Count the number of persons in the specified age group

SELECT COUNT(\*) AS NumberOfPersonsInAgeGroup

FROM persons

WHERE Age = @AgeGroupToCount;

To insert a large amount of data (such as 1,000,000 records) from a file into SQL Server, you can use the **BULK INSERT** statement or the **SqlBulkCopy** class in C#

Assuming you have a CSV file named **yourfile.csv**, and your table structure matches the data in the file:

BULK INSERT YourTableName

FROM 'C:\YourFilePath\yourfile.csv'

WITH (

FIELDTERMINATOR = ',',

ROWTERMINATOR = '\n',

BATCHSIZE = 10000, -- Adjust as needed

TABLOCK

);

Create table Student

(

Id int identity(1,1) primary key,

Name Varchar(100) null,

Email Varchar(100) null,

Grade Varchar(1) null,

Gender Varchar(1) null

)

Script to change the Grade column to GradeName in sql server:

EXEC sp\_rename 'Student.Grade', 'GradeName', 'COLUMN';

Script to change the data type of a column in sql server:

ALTER TABLE Student

ALTER COLUMN Gender CHAR(1);

insert into Student (Name, Email, Grade, Gender) Values

('Shubham', 'shubham@gmail.com', 'A', 'M'),

('Sohan', 'sohan@gmail.com', 'A', 'M')

Updated Table will be ----

Create table Student

(

Id int identity(1,1) primary key,

Name Varchar(100) null,

Email Varchar(100) null,

GradeName Varchar(1) null,

Gender char(1) null

)

**To rename a column:**

sp\_rename 'table\_name.old\_column\_name', 'new\_column\_name' , 'COLUMN';

**To rename a table:**

sp\_rename 'old\_table\_name', 'new\_table\_name';

**Creating a Table**

CREATE TABLE Table\_name(

col\_1 TYPE col\_1\_constraint,

col\_2 TYPE col\_2 constraint,

col\_3 TYPE UNIQUE,

col\_4 TYPE REFERENCES Table\_Name(col\_name),

.....

)

col: The name of the columns.

TYPE: Data type whether an integer, variable character, etc

col\_constraint: Constraints in SQL like PRIMARY KEY, NOT NULL, UNIQUE, REFERENCES, etc.

col\_3: Defining an ALTERNATE KEY using constraint UNIQUE

col\_4: Defining an FOREIGN KEY using constraint REFERENCES

# SQL – ALTERNATE KEY

We can define the Alternate key as the set of Candidate Keys other than the Primary Key. There can be many Candidate Keys for a given table and out of all these the Database Administrators selects only one of these as the Primary Key. Hence, the other Candidate Keys which are not used as a Primary Key are the “Alternate Keys”.

Some important points about Alternate Keys are as follows:

1. A Primary Key can’t be an Alternate Key. For a table with a single Candidate Key which has to be the Primary Key will not contain any Alternate Key.
2. A Foreign Key can’t be an Alternate Key as it is only used to reference another table.
3. The alternate Key should be unique.
4. An Alternate Key can be a set of a single attribute or multiple attributes.
5. It can be NULL as well.
6. **Write sql query to select 7th lowest paid salary from Employees table in sql server?**

Select Salary

from Employees

order by Salary

Offset 6 rows

Fetch next 1 row only

1. **Write query to count total Employees of each Dept\_ID.**

Select Dept\_ID, Count(\*) as 'Total Employee'

From Employees

Group by Dept\_ID

1. **OOP Concepts with real-world examples**

[https://jeemariyana.medium.com/oop-concepts-with-real-world-examples-cda1cd277f4f](10) SQL Server Imp Interview Questions.docx)

1. **Show first name, last name and role of every person that is either patient or doctor. The roles are either "Patient" or "Doctor".**

**Here is the colums of patients table**

**patient\_id INT**

**first\_name TEXT**

**last\_name TEXT**

**gender CHAR(1)**

**birth\_date DATE**

**city TEXT primary key**

**province\_id CHAR(2)**

**allergies TEXT**

**height INT**

**weight INT**

**doctors table columns**

**doctor\_id INT**

**first\_name TEXT**

**last\_name TEXT**

**specialty TEXT**

Ans: select first\_name, last\_name, 'Patient' as role from patients

union all

select first\_name, last\_name, 'Doctor' as role from doctors

The main difference between UNION and UNION ALL is that:

****UNION****: only keeps *unique* records

****UNION ALL****: keeps all records, including *duplicates*

**5. Show all allergies ordered by popularity. Remove NULL values from query.**

SELECT

allergies,

COUNT(\*) AS Popularity

FROM patients

WHERE

allergies IS NOT NULL

GROUP BY allergies

ORDER BY Popularity DESC

1. **Show all patient's first\_name, last\_name, and birth\_date who were born in the 1970s decade. Sort the list starting from the earliest birth\_date.**

Select first\_name, last\_name, birth\_date

FROM patients

WHERE

YEAR(birth\_date) BETWEEN 1970 AND 1979

ORDER BY birth\_date ASC;

Or

Select first\_name, last\_name, birth\_date

FROM patients

WHERE

birth\_date >= '1970-01-01'

AND birth\_date < '1980-01-01'

ORDER BY birth\_date ASC

Or

Select first\_name, last\_name, birth\_date

from patients

Where

YEAR(birth\_date) >= 1970 and YEAR(birth\_date) < 1980

order by birth\_date asc

1. **We want to display each patient's full name in a single column. Their last\_name in all upper letters must appear first, then first\_name in all lower case letters. Separate the last\_name and first\_name with a comma. Order the list by the first\_name in decending order  
   EX: SMITH,jane**

SELECT

CONCAT(UPPER(last\_name), ',', LOWER(first\_name)) AS new\_name\_format

FROM patients

ORDER BY first\_name DESC;

Or

SELECT

UPPER(last\_name) + ',' + LOWER(first\_name) AS new\_name\_format

FROM patients

ORDER BY first\_name DESC;

**7. Show the province\_id(s), sum of height; where the total sum of its patient's height is greater than or equal to 7,000.**

SELECT province\_id, SUM(height) AS sum\_height

FROM patients

GROUP BY province\_id

HAVING sum\_height >= 7000

Or

select \*

from (select province\_id, SUM(height) as sum\_height FROM patients group by province\_id)

where sum\_height >= 7000;

**8. Show the difference between the largest weight and smallest weight for patients with the last name 'Maroni'**

select (max(weight) - Min(weight)) as weight\_difference

from patients

where last\_name = 'Maroni'

**9. Show all of the days of the month (1-31) and how many admission\_dates occurred on that day. Sort by the day with most admissions to least admissions.**

|  |  |  |
| --- | --- | --- |
| **IMG_256** | **patient\_id** | **INT** |
|  | **admission\_date** | **DATE** |
|  | **discharge\_date** | **DATE** |
|  | **diagnosis** | **TEXT** |
| **IMG_257** | **attending\_doctor\_id** | **INT** |

SELECT

DAY(admission\_date) AS day\_number,

COUNT(\*) AS number\_of\_admissions

FROM admissions

GROUP BY day\_number

ORDER BY number\_of\_admissions DESC

**10. Show all columns for patient\_id 542's most recent admission\_date.**

Select \*

from admissions

where patient\_id = 542

group by patient\_id

having max(admission\_date)

Or

SELECT \*

FROM admissions

WHERE patient\_id = 542

ORDER BY admission\_date Asc

Offset 1 rows

fetch next 1 rows only

1. **Show patient\_id, attending\_doctor\_id, and diagnosis for admissions that match one of the two criteria:  
   1. patient\_id is an odd number and attending\_doctor\_id is either 1, 5, or 19.  
   2. attending\_doctor\_id contains a 2 and the length of patient\_id is 3 characters.**

SELECT

patient\_id,

attending\_doctor\_id,

diagnosis

FROM admissions

WHERE

(

attending\_doctor\_id IN (1, 5, 19)

AND patient\_id % 2 != 0

)

OR

(

attending\_doctor\_id LIKE '%2%'

AND len(patient\_id) = 3

)

1. **Show first\_name, last\_name, and the total number of admissions attended for each doctor.  
   Every admission has been attended by a doctor.**

**admissions table**

|  |  |  |
| --- | --- | --- |
| **IMG_256** | **patient\_id** | **INT** |
|  | **admission\_date** | **DATE** |
|  | **discharge\_date** | **DATE** |
|  | **diagnosis** | **TEXT** |
| **IMG_257** | **attending\_doctor\_id** | **INT** |

**doctors**

|  |  |  |
| --- | --- | --- |
| **IMG_256** | **doctor\_id** | **INT** |
|  | **first\_name** | **TEXT** |
|  | **last\_name** | **TEXT** |
|  | **specialty** | **TEXT** |

Select

d.first\_name,

d.last\_name,

count(a.patient\_id) as total\_no\_of\_admissions\_attended

From

admissions a Inner Join doctors d on d.doctor\_id = a.attending\_doctor\_id

Group By a.attending\_doctor\_id

Or

SELECT

first\_name,

last\_name,

count(\*)

from

doctors p,

admissions a

where

a.attending\_doctor\_id = p.doctor\_id

group by p.doctor\_id;

1. **For each doctor, display their id, full name, and the first and last admission date they attended.**

Select

d.doctor\_id,

Concat(d.first\_name, ' ', d.last\_name) as full\_name,

Min(admission\_date) as first\_admission\_date,

Max(admission\_date) as last\_admission\_date

from doctors d join admissions a

on d.doctor\_id = a.attending\_doctor\_id

group by a.attending\_doctor\_id

1. **Display the total amount of patients for each province. Order by descending.**

SELECT

province\_name,

COUNT(\*) as patient\_count

FROM patients pa

join province\_names pr on pr.province\_id = pa.province\_id

group by pr.province\_id

order by patient\_count desc;

**15. For every admission, display the patient's full name, their admission diagnosis, and their doctor's full name who diagnosed their problem.**

**patients table**

|  |  |  |
| --- | --- | --- |
| **IMG_256** | **patient\_id** | **INT** |
|  | **first\_name** | **TEXT** |
|  | **last\_name** | **TEXT** |
|  | **gender** | **CHAR(1)** |
|  | **birth\_date** | **DATE** |
|  | **city** | **TEXT** |
| **IMG_257** | **province\_id** | **CHAR(2)** |
|  | **allergies** | **TEXT** |
|  | **height** | **INT** |
|  | **weight** | **INT** |

SELECT

CONCAT(p.first\_name, ' ', p.last\_name) as patient\_name,

diagnosis,

CONCAT(d.first\_name,' ',d.last\_name) as doctor\_name

FROM patients p

JOIN admissions a ON a.patient\_id = p.patient\_id

JOIN doctors d ON d.doctor\_id = a.attending\_doctor\_id;

**16. Display the first name, last name and number of duplicate patients based on their first name and last name.  
Ex: A patient with an identical name can be considered a duplicate.**

SELECT

first\_name,

last\_name,

COUNT(\*) AS duplicate\_count

FROM

patients

GROUP BY

first\_name,

last\_name

HAVING

COUNT(\*) > 1;

1. **Display patient's full name, height in the units feet rounded to 1 decimal,  
   weight in the unit pounds rounded to 0 decimals, birth\_date,**

**gender non abbreviated.  
Convert CM to feet by dividing by 30.48.  
Convert KG to pounds by multiplying by 2.205.**

SELECT

CONCAT(first\_name, ' ', last\_name) as patient\_name,

round(height/30.48, 1) as height\_in\_feet,

round(weight\*2.205, 0) as weight\_in\_pound,

birth\_date,

case

When gender='M' then 'Male'

Else 'Female'

End as gender\_type

FROM patients

**18. Show patient\_id, first\_name, last\_name from patients whose does not have any records in the admissions table. (Their patient\_id does not exist in any admissions.patient\_id rows.)**

SELECT

patients.patient\_id,

first\_name,

last\_name

from patients

where patients.patient\_id not in (

select admissions.patient\_id

from admissions

)

Or

SELECT

patients.patient\_id,

first\_name,

last\_name

from patients

left join admissions on patients.patient\_id = admissions.patient\_id

where admissions.patient\_id is NULL

**19. Show unique birth years from patients and order them by ascending.**

select

distinct Year(birth\_date) as unique\_birth\_year

from

patients

order by

unique\_birth\_year asc

**20. Show unique first names from the patients table which only occurs once in the list.  
For example, if two or more people are named 'John' in the first\_name column then don't include their name in the output list. If only 1 person is named 'Leo' then include them in the output.**

select

distinct first\_name

from

patients

group by

first\_name

having

count(first\_name) = 1

1. **Show patient\_id and first\_name from patients where their first\_name start and ends with 's' and is at least 6 characters long.**

select

patient\_id, first\_name

from

patients

where

first\_name like 's%s' and len(first\_name) >= 6

Or

SELECT

patient\_id,

first\_name

FROM patients

WHERE first\_name LIKE 's\_\_\_\_%s';

**22. Show patient\_id, first\_name, last\_name from patients whose diagnosis is 'Dementia'.  
Primary diagnosis is stored in the admissions table.**

select

patients.patient\_id, first\_name, last\_name

from

patients join admissions

on

patients.patient\_id = admissions.patient\_id

where diagnosis = 'Dementia'

Or

SELECT

patient\_id,

first\_name,

last\_name

FROM patients

WHERE patient\_id IN (

SELECT patient\_id

FROM admissions

WHERE diagnosis = 'Dementia'

);

**23. Display every patient's first\_name.  
Order the list by the length of each name and then by alphabetically.**

select

first\_name

from

patients

order by

len(first\_name), first\_name asc

**24. Show the total amount of male patients and the total amount of female patients in the patients table.  
Display the two results in the same row.**

select

(select Count(\*) from patients where gender = 'M') as total\_male\_patients,

(select count(\*) from patients where gender = 'F') as total\_female\_patients

Or

SELECT

SUM(Gender = 'M') as male\_count,

SUM(Gender = 'F') AS female\_count

FROM patients

Or

select

sum(case when gender = 'M' then 1 end) as male\_count,

sum(case when gender = 'F' then 1 end) as female\_count

from patients;

**25. Show first and last name, allergies from patients which have allergies to either 'Penicillin' or 'Morphine'. Show results ordered ascending by allergies then by first\_name then by last\_name.**

select

first\_name, last\_name, allergies

from

patients

where

allergies = 'Penicillin' or allergies = 'Morphine'

order by

allergies, first\_name, last\_name

**26. Show patient\_id, diagnosis from admissions. Find patients admitted multiple times for the same diagnosis.**

select

patient\_id, diagnosis

from

admissions

group by

patient\_id,

diagnosis

having

count(\*) > 1

**27. Show the city and the total number of patients in the city.  
Order from most to least patients and then by city name ascending.**

select

city, count(patient\_id) as 'total Patient in city'

from

patients

group by

city

order by

count(city) desc, city asc