TABLE OF CONTENT

SQL SERVER

INTRODUCTION TO SQL SERVER:

SQL is a language used to communicate with DATABASE SERVER.
Due to its powerful database administration and programming language capabilities, SQL Server has become one of the most commonly used RDBMSs of today.

SQL Server is capable of supporting thousands of users simultaneously. Therefore, it is widely used as server database system. A server database system is database system that runs on central server and allows multiple users to access the same data at a time.

Initially SQL was called SQUARE and later SEQUEL and now it is called SQL.

SQUARE – SPECIFICATIONS OF QUERY AS RELATIONAL EXPRESSIONS

SEQUEL – STRUCTURED ENGLISH QUERY LANGUAGE

SQL – STURCTURED QUERY LANGUAGE

Points To Be Remembered

1. SQL was introduced in the year 1989 by MICROSOFT as SQL SERVER 1.0
2. SQL is a Common Database Language, since it gets understand by every RDBMS
3. SQL is a Command Based Language
4. SQL is a Insensitive Language
5. SQL is a Non Procedural Language
6. SQL is a GUI based RDBMS.
7. SQL is a CLIENT/ SERVER technology.
8. It is associated with DATABASE ENGINE (set of compilers).

FEATURES OF SQL SERVER

1. GUI based RDBMS(Commands, Wizards)
2. Provides High Security
3. Provides High Scalability
4. Provides High Availability
5. Provides High Performance
6. Supports XML

VERSIONS OF SQL SERVER

SQL SERVER 1.0 ---------- 1989
SQL SERVER 4.0 ---------- 1993-95
SQL SERVER 4.5 ---------- 1993-95
SQL SERVER 6.0 ---------- 1995-97
SQL SERVER 6.5 ---------- 1995-97

SQL SERVER 7.0 ---------- 1998
SQL SERVER 2000
SQL SERVER 2005
SQL SERVER 2008
SQL SERVER 2008R2
SQL SERVER 2012
SQL SERVER 2014
SQL SERVER 2016

Differences between Oracle & SQL Server

ORACLE	SQL SERVER
It was Introduced In 1979	In 1989 It was Introduced
It Is CUI Based RDBMS	It Is GUI Based RDBMS
It can be Installed on any OS	It Requires Specific OS
Provides High Security	It Is Less Security
It Is High Priced RDBMS	It Is Low Priced RDBMS

DATABASE MANAGEMENT SYSTEM (DBMS)

RDBMS = R + DBMS {R Stands for Referential Integrity Constraint}
Relation can be defined as Referential Integrity Constraint, with which we create relationships One to One & One to Many (Direct Implementation) Many to Many (Indirect Implementation)

DR EF CODD has outlined 12 rules often called as CODD'S rules if any **Database Software Satisfies 8 or 8.5 rules**, it can be considered as pure RDBMS Database.

SQL Server Satisfies 10 CODD's Rules,
Oracle satisfies all 12 CODD's Rules,
MySQL Satisfies 8 rules,
MS Access is not a pure DBMS, because it satisfies only 6 rules, so it is called as Partial / Semi Database

Some other points:

1. RDBMS always presents the data in 2 Dimensional formats, i.e. in Rows and Columns, further to tables.
2. Collection of characters "or" we call as IDENTIFIERS /FIELD/ ATTRIBUTE.
3. Collection of columns: ROW or RECORD or TUPLE.
4. Collection of Rows & Columns: Table or Entity or Objects or Relations.
5. RDBMS always represent the data in a Normalized way.

Data Base Management System is defined as collection of Software Programs used to store the data, manage the data, and retrieve the data. Data retrieval is based on three points
1. Accuracy – Data Free from Errors.
2. Timelines – With in a time
3. Relevancy – Related Data

Maintaining of the data can be done in two ways.
1. File System (like Flat Files)
2. Data Base

Why Database Management System and why not flat files to maintain the data.

Flat Files has many disadvantages, they are
1. No Security
2. Difficulty in accessing the data
3. Redundancy
4. Data Inconsistency
5. Sequential Search Technique (Time Consuming Process)
6. Indexing is not present in file system
7. No Identifier (Related Information)

Features Of Relational DBMS:
1. Data Is Stored In Tables.
2. Intersection Of Rows And Columns Will Give Only One Value.
3. Relation Among Data Is Established Logically.
4. There Is No Physical Link Among Data.
5. There Is No Data Redundancy.
6. High Security For Data.
7. It Supports Any Type Of Data (Ex: Numbers, Numeric, Data, Character, Date Images Etc).
8. It Supports Null Values.
9. It Supports E.F Codd Rules.
10. It Supports Integrity Constraints.
11. Multiple Users Can Access Data From Any Location.

Good Example to understand SQL Language, User, Data, Database and DBMS:.

A handicapped Person 'A' is owner of a curry point and can't walk. Now he is at bit longer away from curries device and eating food. He needs a curry now. If he says to device to send one curry, it can't send because it a device so it is unable to understand his language. So there must be a person who understands owner's language and that person must be at device to store the curries, manage the curries and giving that curries to the customers.

Here:

Curries	=	Data
Curries device	=	Database
Another person	=	DBMS
Person 'A'	=	User
Language	=	SQL Language statements

SQL SERVER MANAGEMENT STUDIO (SSMS):

When we install SQL Server on our computer, a number of tools are also installed along with it. So these tools can be broadly categorized into three groups, development tools, configuration tools and performance tools.

Configuration tools allow us to configure SQL Server and to start and stop the many services. Like SSIS, SSRS and SSAS.

Eg: SQL Server Configuration manager and
Reporting services configuration manager

Performance tools help in improving the performance of SQL Server on our computer by allowing you to examining the query performance and modifying the physical design.

Eg: SQL Server Profiler and
Database engine tuning advisor

Development tools are useful for development purpose such as T-SQL queries and developing BI solutions.

Eg: SSTD and SSMS

SSMS is a development toll which was introduced in SQL Server 2005 as a replacement for three SQL Server tools- Enterprise Manager, Query Analyzer and Analysis manager.

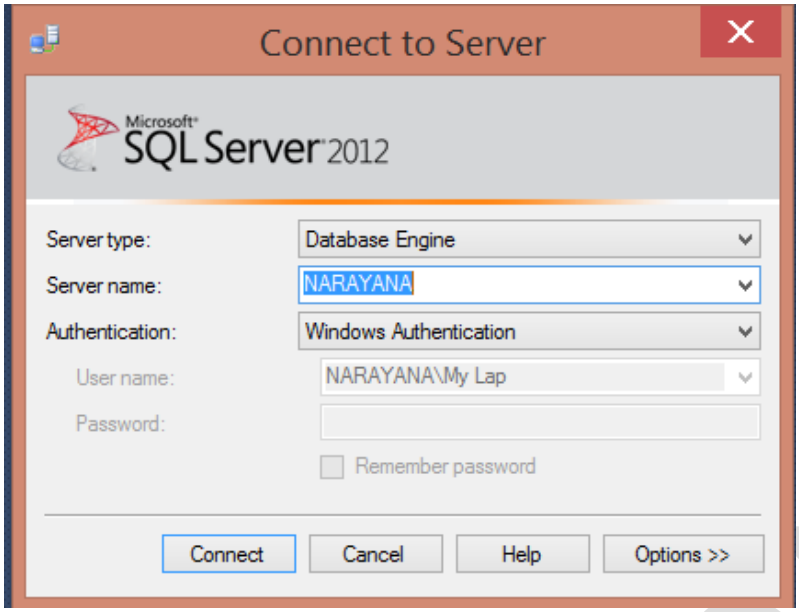
It was developed with the idea of providing single tool that can meet the needs of both developer and administrators.

A central feature of SSMS is the Object Explorer, which allows the user to browse, select, and act upon any of the objects within the serve

SSMS consists of several easy-to-use graphical tools and editors such as Object explorer and Query editor.

SSMS allows us to write T-SQL queries using Query editor. In this way it acts as simple even though it is a comprehensive platform in performing both administrative and development tasks.

To open SSMS:
Start→ All programs→ Microsoft SQL server 2012→SSMS



***SERVER TYPE*:** When we install SQL Server, automatically 4 servers are installed.

Those are

1. Database Engine
2. Analysis Services
3. Reporting Services
4. Integration Services

So here we need to choose Database Engine when we need to work with SQL Server.

SERVER NAME: Server name is the name of computer.

AUTHENTICATION MODES:

SQL Server supports two types of authentication modes,

1. Windows Authentication Mode:
   a. Windows Authentication Mode is also called Trusted Authentication Mode
   b. Windows authentication mode is the default authentication mode. That means SQL Server does not ask for password.
   c. When a user connects through a windows user account than SQL Server validates the account name and password using the windows principal token in the operating system. This means that the user name and password is confirmed by the windows.

2. SQL Server Authentication Mode:
   a. SQL Server Authentication Mode is also called Mixed-Mode Authentication Mode.
   b. When user wants to connect through SQL Server Authentication Mode than he must provide his credentials (username and password) otherwise SQL Server will not allow.
   c. Both username and password are created by using SQL Server and stored in SQL Server.

Things to Observe:

1. While writing the Queries using T-SQL in **SQL Server Management Studio** we need not to follow any particular case. Because T-SQL is case **insensitive** language.
2. After writing the Query, we need to select that query using either mouse or keyboard.
3. Now Press **F5** (Execute Key).
4. Then the results are displayed in a separate window called **Result window or Result Pane.**
5. Use **Ctrl+R** to Hide/Show the Result window or Result Pane.
6. Use **F8** for Object Explorer

DATA AND INFORMATION

DATA: Whatever we enter through the key board than that is called data.

Data is nothing but raw facts which are in unorganized manner.

Eg.: ON SERVER SQL WE WORKING ARE

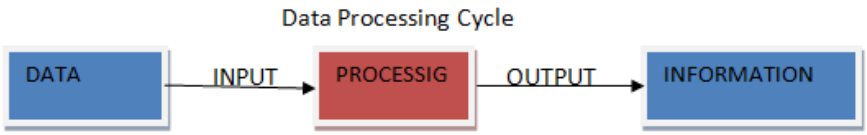
The above are raw facts which are in unorganized manner.

INFORMATION: When DATA is organized, structured or presented so as to make them meaningful then they are called INFORMATION .

Eg.: WE ARE WORKING ON SQL SERVER

Here the above example is giving meaning. So it is called information.

Simply we can say that the input to the processor is the DATA and output from the processor is the INFORMATION.

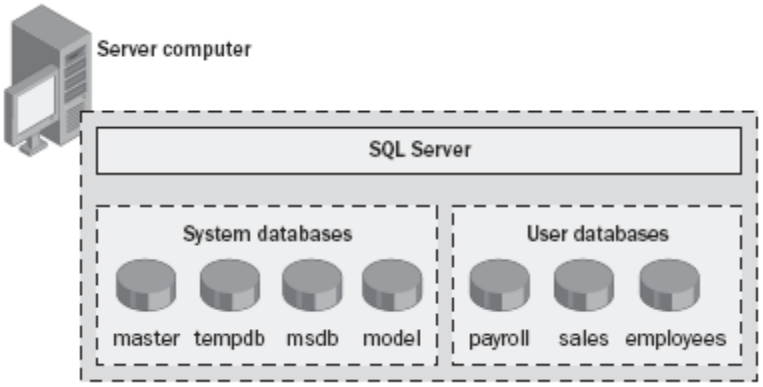


DATA BASE:

A Database is a collection of information that is organized so that it can be easily accessed, managed and updated. Databases are generally classified according to their organizational approach. The most relevant approach is Relational Database, a tabular database in which data is defined so that it can be reorganized and accessed in many different ways.

A database is basically a collection of information organized in such a way that a computer program can quickly select desired pieces of data.

.



Data bases are two types in SQL Server,

a. System Defined Data Bases.
b. User Defined Data Bases.

System Defined Data Bases:

These Data Bases are defined by system (means by Microsoft).

1. **Master:** It stores all configuration settings of SQL Server where it is installed.

1) The master database is the brain of SQL Server.
2) Great care should be taken when modifying any information contained in the master database.
3) We should get in the habit of backing up our master database whenever we make environmental changes to our server, including changing the sizes of databases or adding users.
4) By default in current versions of Microsoft SQL Server, the master is set to 25MB. It can be increased 5MB more if needed.

2. **Model**:
    1) It gives it's template to user defined databases, means it copies whatever it has to all user defined databases.
    2) Be careful when placing things in the model.
    3) This action will increase the Minimum size of your databases and may add unnecessary objects to databases.

3. **Temp DB**:
    1) It is used to store data temporarily.
    2) Tempdb is referred as very small scratch paper that we use for a short period of time and then discard when we no longer need the information on each piece of paper.

4. **MSDB**:

    1) It is used by SQL Server agent for scheduling alerts and jobs,
    2) You can add tasks to this database that will be performed on a scheduled recurring basis.
    3) You can also view the history of the defined tasks and their execution results.

5. **Resource**(msSQLsystemresource):

    1) The resource database is a hidden system database.
    2) This is where system objects are stored.
    3) It isn't possible to see the resource database.

However you can see the data file by navigating to

C:\Program Files\Microsoft SQL Server\MSSQL10.MSSQLSERVER\MSSQL\Binn.

User Defined Data Bases: Data Bases which are defined by user called user defined databases.
like: Sales table, Customer table, Employee table…

Creating a database:

Syn: Create database <Data base name>

Eg: Create database MyDatabase

After writing the above Query in SQL Server Management Studio then select the complete statement and press F5. Then server creates database with name MyDatabase, that we can check it in Database list box in **SSMS.**

Database files and File groups

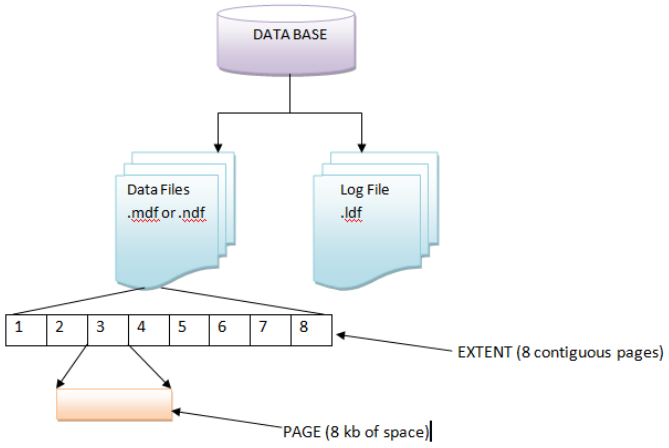
Every SQL Server database has two operating system files: a data file and a log file. Data files contain data and objects such as tables, indexes, stored procedures, and views. Log files contain the information that is required to recover all transactions in the database. Data files can be grouped together in filegroups for allocation and administration purposes.

SQL Server allows three types of database files.

Primary data files: Every database has one primary data file. This file contains the startup information for the database and is used to store data. The name of primary database file has the extension *MDF(master data file). By default the size of .mdf file is 1.18 mb*

Secondary data files: These files hold all of the data that does not fit into the primary data file. A database can have zero or more Secondary data files. The name of secondary database file has the extension *NDF(next data file).*

Transaction Log files: These files hold the log information used to recover the database. There must be at least one log file for each database. The name of a Log file has the extension *LDF(log data file). By default the size of .ldf file is 504 kb.*



Explanation:

1. When we create database, automatically two files will be created.
2. Those are .mdf and .ldf files.
3. Each file has multiple Extents which has 8 data pages.
4. Each data page size is 8kb.
5. In these data pages the data will be stored.

DATA BASE OBJECTS: In a relational database, Database object is a data structure which is used to store or reference the data.
Eg: Table, View, Stored Procedure, Functions, Sequence, Synonym, Constraint…

Note:

1. SQL SERVER can handle nearly 32767 Databases
2. Each Database can handle nearly 2 billion Database Objects.
3. Each Table can handle nearly 1024 columns
4. Each Table can handle nearly 1 million Rows.

Points To Be Remembered About Table:

1. A database table is a collection of columns(also known as fields or attributes) and rows(also known as tuples or records) where data is actually stored within a database.
2. No two columns in the table can't have same name.
3. In addition, each column in the table must contain data of a single type.
4. In other words, every column stores a specific category of information; for example, in a table named customers, the CustomerID column stores a unique ID for the customers, CustomerName column stores the name of the customers.
5. Column name should have space in the middle like EMP NAME. In this case we need to use braces like [EMP NAME].
6. Each table can handle nearly 1024 columns.
7. Each table can handle more number of records and size of each row is 8060 bytes.

To see list of tables in current database

```
select * from sys.tables
```

DATA INTEGRITY

Data Integrity means data validation or data checking process or Type Checking process. Before storing the user supplied information into the table, the server performs data integrity process in order to verify whether user supplying valid information or not.

If user supplies valid information then only it will store into the table otherwise server raises an error message like 'Data Type Mismatch'.

We can achieve this Data Integrity in Three ways

1. **Data Types**
2. **Constraints**
3. **Triggers**

DATA TYPES IN SQL SERVER:

Data Type means the type of data which users provide to a specific column.

SQL Server supports 2 types of data types:

1. **System Defined Data Types**
2. **User Defined Data Types.**

System Defined Data Types:

SQL Server already contains some data types called System Defined data types or Predefined Data types or Built-in Data types.

System Defined Data Types again different types

a. **Numeric Data types**
b. **String Data types**
c. **Date Time Data types**
d. **Special Data types**

Numeric Data types: These Data types are used to provide numeric information for the columns, these includes

Data Type	Size
BIT	0 or 1
TINYINT	1 BYTE
SMALLINT	2 BYTES
INT	4BYTES
BIGINT	8BYTES
REAL	4BYTES
FLOAT	8BYTES
DECIMAL (P, S)	5-17 BYTES

TINYINT, SMALLINT, INT, BIGINT are called **Exact Numerics** where as REAL, FLOAT are called **Approximate Numerics**.

String Data types: These Data types are used to provide character information for the columns.

Generally we use CHAR(N),Varchar(N),Nchar(N) and Nvarchar(N)

		Char	Nchar	Varchar	Nvarchar
Length	--	Fixed	Fixed	Variable	Variable
Storage	--	8000GB	4000GB	8000GB	4000GB
Code Type	--	Non UniCode	UniCode	Non UniCode	UniCode
No.Of bytes per char	--	1	2	1	2

CHAR [(N)]: It is a fixed length data type, which occupies by default 1 byte memory. When we give N value then it occupies N bytes of memory where N is an integer. It follows Static memory allocation process.

VARCHAR [(N)]: It is a variable length data type, which occupies by default 1 byte memory. When we give N value then it occupies N bytes of memory where N is an integer. It follows dynamic memory allocation process.

Note: The Maximum limit for N is 8000,if it is more than 8000 characters we will use TEXT or VARCHAR(MAX).

Date Time Data Type:

These data types are used to provide date oriented information to the columns, these includes

Data Type	Size	Range
SMALLDATETIME	2 BYTES	1900-01-01 TO 2079-06-06
DATETIME	4BYTES	1753-01-01 TO 9999-12-31

Special Data types:

These data types are used to provide miscellaneous information to the columns, these includes

Data Type	Size
SMALLMONEY	4 BYTE
MONEY	8 BYTES
IMAGE	16 BYTES
VARBINARY (MAX)	Unlimited

Syn.: CREATE TABLE <TAB_NAME>(Col1 DataType1, Col2 DataType, Col3 DataType3,..)

Eg.: CREATE TABLE EMP(ID Int, Ename Varchar(Max),Sal Money, MobileNumber Bigint, Dob Datetime)

II. User Defined Data Types:

When user create a data type then that data type is called user defined data type

Syntax: **CREATE TYPE USER_DEFINED_DATATYPE FROM SYS_DEFINED_DATATYPE**

Ex:

CREATE TYPE MYINT FROM INT
CREATE TYPE MYFLOAT FROM FLOAT
CREATE TYPE CASH FROM MONEY

LANGUAGES OR COMMANDS:

Whenever user wants to interact with SQL Server, he has to interact with SQL Server through SQL Languages. Those Sub Languages are,

1. DDL (Data Definition Language)
2. DML (Data Manipulation Language)
3. DRL (Data Retrieval Language) or DQL (Data Query Language)
4. DCL (Data Control Language)
5. TCL (Transaction Control Language)

DATA DEFINITION LANGUAGE (DDL): These DDL commands are used to create and modify the structure of data base objects.

DDL Commands are Create, Alter, Truncate and Drop

Note: Here we can add ***sp_rename*** system defined stored procedure under DDL as it can be used to rename the columns.

Tip: We can remember those commands as ***Dr.CAT***

Dr means Drop.
C means Create.
A means Alter.
T means Truncate.

Create : Create statement is used to create data base objects, Like Tables, Views, Indexes, User Defined Stored Procedures, User Defined Functions, Triggers, Rules, Defaults..

Syn: create table tablename(col1 datatype1,col2 datatype2,col3 datatype3….)

Eg: CREATE TABLE TABNAME (ID INT,NAME VARCHAR(MAX),SAL INT)

Alter: Alter is used to modify the structure of data base objects.

With alter we can,

1. Add new column to existing table

Syn: alter table TableName add ColName DataType(Size)

Eg: ALTER TABLE TABNAME ADD LOC VARCHAR(MAX)

2. Drop column from existing table

Syn : Alter table TableName drop column ColumnName

Eg: ALTER TABLE TABNAME DROP COLUMN LOC

3. Change data type of existing column in the table

Syn: Alter table TableName alter column ColumnName TargetDataType(size)

Eg : ALTER TABLE TABNAME ALTER COLUMN NAME CHAR(50)

4. Change the data type size of the column in the table

Syn : Alter table TableName Alter column ColumnName Datatype(TargetSize)

Eg : ALTER TABLE TABNAME ALTER COLUMN NAME CHAR(100)

Truncate: Truncate is used to remove data from table

Syn: Truncate table TableName

Eg : TRUNCATE TABLE TABNAME

Drop: Drop command is used to drop the database objects permanently.

Syn: Drop Table TableName

Eg: DROP TABLE TABNAME

Sp_rename *:* This system defined stored procedure is used to rename the columns and tables

Renaming Column:

Syn: sp_rename 'TabName.ColName','NewColName'

EG.: SP_RENAME 'EMP.SAL','SALARY'

Renaming Table:

Syn: sp_rename 'OldTableName','NewTableName'

EG: SP_RENAME 'EMP','EMPLOYEE'

DATA MANIPULATION LANGUAGE (DML): These are used to manipulate the data in the database objects.

DML Commands are: Insert
Delete
Update

Tip: we can remember these commands as **UID** (pronounced User ID)

Insert: Insert command is used to insert the data into table.

There are different ways to insert data,

Type1: Using multiple INSERTs for Multiple records

Syn : Insert into TableName values(Val1,val2,val3,…)

Eg : INSERT INTO TABNAME VALUES(1,'SAI',10000)
INSERT INTO TABNAME VALUES(2,'RAM',20000)
INSERT INTO TABNAME VALUES(3,'TEJA',10000)

Type 2: Using single INSERT statement for multiple records

Syn: Insert into TbleName(col1,col2,col3..) values(val1,val2,val3…)

Eg: INSERT INTO TABNAME VALUES(4,'SATYA',4000),(5,'RAJA',5000)

Type 3: Inserting data for specific columns

Syn: Insert into TbleName(col1,col2,col3..) values(val1,val2,val3…)

Eg: INSERT INTO TABNAME (ID, NAME, SAL) VALUES (6,'SATYA RAJ', 4000), (7,'SYAM', 5000)

Type 4: Inserting Data using union all operator

Syn: Insert into TableName
select Val1, val2, val3
union all
select val1, val2, val3

Eg: INSERT INTO TABNAME
SELECT 8,'DEVI', 3000
UNION ALL
SELECT 9,'CHANDU', 4000

UNION ALL
SELECT 10,'NANI', 1000

Delete: Delete command is used to remove the data from table.

We can use WHERE clause in delete statement to remove specific record(s).

Syn: Delete from TabelName <where clause is optional>

Eg: DELETE FROM TABNAME where sal<10000

E.g.: Write a query to delete all the employees from the table

DELETE FROM EMP

This statement deletes all records from EMP table without disturbing its structure (columns).

E.g.: Write a Query to delete all employees who are working under 10th department.

DELETE FROM EMP WHERE DEPTNO=10

E.g.: Write a Query to delete all details of an employee who is working under 20th department and employee number 33.

DELETE FROM EMP WHERE DEPTNO=20 AND EMPNO=33

Update: Update command is used to modify the data in the table.

Syn: Update tableName set ColumnName= TagetValue where <condition>

Eg: UPDATE TABNAME SET SAL=3000 WHERE ID=8

E.g.: Write a Query to increase the salaries of all the employees in EMP table

UPDATE EMP SET SAL=SAL+1000

The above statement increases all employees' salaries by 1000.

E.g.: Write a Query to modify (increase) the salaries of all employees who are working under 10th department.

UPDATE EMP SET SAL=SAL+500 WHERE DEPTNO=10

E.g.: Write a Query to modify the salary of an employ whose employ number 11 and who is working under 20th department.

UPDATE EMP SET SAL= SAL+300 WHERE EMPNO=11 AND DEPTNO=20

DATA RETRIEVAL/QUERY LANGUAGE (DRL/DQL) : The command in DQL/DRL is used to retrieve the data from data base object, Like table.

DRL/DQL command is SELECT

Syn : Select */ColList from TableName

Eg : SELECT * FROM TABNAME

E.g.: Write a Query to select employee details who are working under 10^{th} department.

SELECT * FROM EMP WHERE DEPTNO=10

E.g.: Write a Query to select employ details who are earning salaries between 5000 and 25000.

SELECT * FROM EMP WHERE SAL>5000 AND SAL<25000

E.g.: Write a Query to select employ details whose employ number is 22.

SELECT * FROM EMP WHERE EMPNO=22

E.g.: Write a Query to select employee details whose department is null.

SELECT * FROM EMP WHERE DEPTNO IS NULL

Note: In the above example we used a special operator called IS operator, which used to compare NULLs.

Complete SELECT Statement:

```
SELECT
  [TOP ( top_expression ) ]
  [ALL | DISTINCT]
  {* | column_name | expression} [,...n ]
  [FROM { table_source } [ ,...n ] ]
  [WHERE <search_condition> ]
  [GROUP BY <group_by_clause> ]
  [HAVING <search_condition> ]
  [ORDER BY <order_by_expression> ]
```

Important Points To Be Remembered:

1. In a single select statement data can be retrieved of 4096 columns and 255 tables.
2.SELECT statement should always be executed from left to right.

TRANSACTION CONTROL LANGUAGE (TCL)

A transaction is a set of T-SQL Statements that are executed together as a unit like a single T-SQL Statement. If all these statements are executed successfully than that transaction is committed and the changes made by SQL Statements permanently saved to database. If any one of those T-SQL Statements within a transaction fail than the complete transaction is cancelled/ rolled back.

Syn:
Begin tran/transaction
<T-SQL Statements>

CREATE TABLE TRAN1(ID INT,NAME VARCHAR(MAX),SAL INT)

E.g.:
begin tran
insert into tran1 values(1,'satya',1000)

Now we have inserted one record into the Tran1 table under the transaction. This record is not saved in the table completely, because still the transaction is running.
If we run the following select statement in the same session than we can see the data in that table but if we run the same SELECT statement in the other session we don't get the data.

SELECT * FROM TRAN1

So we have some commands to save the changes on the database.

TCL language has three commands:

1. Commit
2. Rollback
3. Save point

COMMIT:

1. This TCL Command is used to make a transaction permanent in a database.
2. When multiple isolate transactions are created requires equal number of COMMITS to make them permanent in a DATABASE.

Syn: COMMIT

E.g.: COMMIT

If we commit the transaction than the T-SQL Statements affect the database.
As we inserted the data in the above example, that insert statement will affect the database and we can get the data throughout all sessions.

ROLLBACK:

1. This TCL Command is used to cancel the Complete Transaction or Partial Transaction.
2. When isolated Multiple transactions are created in a Single Session.

3. A Single ROLLBACK command would cancel all the transactions.

Syn: ROLLBACK [TRAN / TRANSACTION] NAME

E.g.: **ROLLBACK**

If we run Rollback than all the T-SQL Statements will rolled back, that means the database will not affect. We can't get the data in any sessions.

SAVE:

1. This TCL Command is also used to create Sub-Transactions under a Main Transaction.
2. The Created Sub Transaction can be cancelled partially.
3. Random Selection from Cancellation of Sub Transaction is not allowed

Syn:
save [tran / transaction] name

E.g.:

```
BEGIN TRAN
INSERT INTO TRAN1 VALUES(1,'SALES',3000)
SAVE TRAN F_T
UPDATE TRAN1 SET NAME='MGR' WHERE ID=1
SAVE TRAN S_T
INSERT INTO TRAN1 VALUES(2,'HR',4000)
SAVE TRAN T_T
INSERT INTO TRAN1 VALUES(3,'ANALYST',6000)
SAVE TRAN F_T
DELETE FROM TRAN1 WHERE ID=3
```

In the above transaction we have four sub transactions like f_t, s_t, t_t and f_t,
sub transaction f_t takes care of update statement,
sub transaction s_t takes care of insert statement,
sub transaction t_t takes care of insert statement and
sub transaction f_t takes care of delete statement

If we commit/rollback f_t than only that sub transaction (delete statement)will be affected,
If we commit/rollback t_t than both t_t and f_t sub transactions (insert and delete statements)will be affected,
If we commit/rollback s_t than three s_t, t_t and f_t sub transactions (Insert, insert and delete statements)will be affected,
If we commit/rollback f_t than all f_t, s_t, t_t and f_t sub transactions (update, Insert, insert and delete statements)will be affected,

We have one more INSERT statement that is below main transaction and above f_t sub transaction, to affect this INSERT Statement we need to commit/rollback main transaction.

DATA CONTROL LANGUAGE (DCL): DCL commands are used to enforce database security in multiple users' database environment.

SQL Server supports two types of commands:

1. GRANT
2. REVOKE

GRANT: Grant command is used for giving a privileges or permissions (INSERT, SELECT, UPDATE, DELETE) for a user to perform operations on the database objects.

Syntax: GRANT < ALL/SPECIFIC PERMISSIONS > on <object name> To {User} [With GRANT OPTION]

Privilege Name: it's a privilege or permission to the users for using SELECT, DELETE, UPDATE, ALL, EXECUTE and SELECT.

Object Name: It is the name of database objects like Table, Views and Stored Procedure etc….

User: It's name of the user who needs privilege.

With Grant Option: Allows a user to grant access rights to other users.

E.g.:
GRANT ALL ON EMP TO NARAYANA WITH GRANT OPTION

Explanation: From the above statement NARAYANA user account got all permissions on EMP table. Mean time NARAYANA can give the permissions on EMP to another user account because he got the permissions WITH GRANT OPTION.

E.g.: **GRANT INSERT, SELECT ON EMP TO SATYA**

Now SATYA can perform select and insert operations on EMP table.
But SATYA cannot perform update and delete operations on EMP table because he does not have the corresponding permissions.

REVOKE: Revoke command removes privileges / permissions (INSERT, SELECT, UPDATE, DELETE) from the user OR taking back the permission that is given to a user.

Syntax: Revoke <all/privilege name> on <object name > from {user}

Eg: **REVOKE SELECT ON EMP FROM SATYA**

Now Satya is unable to perform SELECT operation on emp table. Because that permission is revoked from Satya.

CLAUSES:

SQL Server provides different types of clauses in SELECT statement.

From: FROM clause is used to fetch data from database object.

Eg : From EMP,
From Customer,
From Dept

SELECT * **FROM EMP**

E.g.: Display the details of all the departments

SELECT * FROM DEPT

Where: Where clause is used to filter the data which is fetched from database object.
Where clause is also used in UPDATE and DELETE statements along with SELECT statement.

Eg: Where empno=1001
Where deptname='Sales'

SELECT * FROM EMP **WHERE SAL >30000**

E.g.: Display the details of employees who are working in department number 30

SELECT * FROM EMP WHERE DEPTNO=30

E.g.: Display the employee names of whose who are drawing more commission.

SELECT ENAME FROM EMP WHERE COMM IS NOT NULL

SELECT ENAME FROM EMP WHERE COMM != NULL

Note: when we set ansi_nulls off, we can use = operator otherwise we have to use 'is' operator

E.g.: Display the names of all employees working as clerks and drawing a salary more than 3000

SELECT * FROM EMP WHERE JOB='CLERK' AND SAL>3000

E.g.: Display the emp names who are working under dept numbers 10 and 20 as sales mans

SELECT * FROM EMP WHERE JOB='SALESMAN' AND DEPTNO IN(10,20)

NOTE: IN operator is used to compare two or more values. If it is only one value than we use = operator.

Group by: It is used to arrange identical values into groups.

It involves Aggregate functions : sum() ,min(),max(),avg(),count(), etc.

It returns one record from each group.

It returns the result in ascending order.

it is written after where clause and before having clause.

Eg : group by loc

group by gender

SELECT LOC, SUM(SAL) AS SUMSAL FROM TABNAME **GROUP BY LOC**

E.g.: Display the total salary of each department.

SELECT SUM(SAL) TOTAL_SAL,DEPTNO FROM EMP GROUP BY DEPTNO

Output:

TOTAL_SAL	**DEPTNO**
8750.00	10
8700.00	20
3700.00	30

E.g.: Display the average salary in each job

SELECT JOB,AVG(SAL) AVG_SAL FROM EMP GROUP BY JOB

Output:

JOB	**AVG_SAL**
ANALYST	3000.000000
CLERK	1116.666666
MANAGER	2450.000000
PRESIDENT	5000.000000
SALESMAN	1450.000000

Note: Whatever columns we use in select list, those all must be used in group by clause.

Whatever columns we use in the group by clause, all those columns may not be required in the select list and those group by columns works automatically.

If we want to use group by clause in the SELECT list, must have any one of aggregate functions.

Having: Having is used to filter the grouped data.

If we want to apply condition on the result of group by clause then we use having clause.

Having is written after group by clause.

Eg: having sum(sal)>20000

having count(*)>2

SELECT SUM(SAL) FROM TABNAME GROUP BY LOC **HAVING SUM(SAL)>30000**

E.g.: Display the total salary of each department, whose department total salary is greater than 5000.

```
SELECT SUM(SAL) TOTAL_SAL,DEPTNO
FROM EMP
GROUP BY DEPTNO
HAVING SUM(SAL)>5000
```

Output:

TOTAL_SAL	DEPTNO
8750.00	10
8700.00	20

E.g.: Display the average salary in each job, whose job average salary is less than 3500.

```
SELECT JOB,AVG(SAL) AVG_SAL
FROM EMP
GROUP BY JOB
HAVING AVG(SAL)<3500
```

Output:

JOB	AVG_SAL
ANALYST	3000.000000
CLERK	1116.666666
MANAGER	2450.000000
SALESMAN	1450.000000

Order by : Order by clause is used to sort the data in either ascending or descending order.
if we don't specify the order than it takes ascending order by default.

Eg : order by sal asc
order by sal desc
order by sal ----(If we don't specify order type then it takes ascending order by default)

SELECT * FROM TABNAME **ORDER BY SAL DESC**

E.g.: Display employee details and names is alphabetical order

```
SELECT *
FROM EMP
ORDER BY ENAME
```

E.g.: Display emp names who are working under dept 20 and their sal in desc order

```
SELECT ENAME,SAL
```

FROM EMP
WHERE DEPTNO =20
ORDER BY SAL DESC

Output:

ENAME	**SAL**
SCOTT	3000.00
FORD	3000.00
ALLEN	1600.00
ADAMS	1100.00

NOTE: In order by clause we can also use **alias names of columns**, **functions** and **Column positions** of the columns which are already in the select statement.

Distinct: Distinct is used to restrict the duplicate values.

Distinct is used only after SELECT word.

Distinct always arranges the data in ascending order.

Distinct includes null values.

When multiple columns are used with DISTINCT, it eliminates only if all columns contain duplicates.

Eg : **SELECT DISTINCT SAL FROM EMP**

SELECT DISTINCT SAL,COMM FROM TABNAME

Top

1. We can use TOP to perform operations on top N no.of rows of a table.
2. We can mention specific number or percentage.
3. Upto SQL Server 2000 it was used only in SELECT statement but later onwards it is being used with UPDATE, DELETE along with SELECT Statement.

E.g:

SELECT TOP 10 * FROM EMP

SELECT TOP 10 PERCENT * FROM EMP

NOTE: To get top rows without using TOP keyword and WHERE clause, we need to use SET ROWCOUNT function

SET ROWCOUNT 10

Execution steps in complete SELECT Statement

1. From
2.Where
3.On
4.Join
5.Group by
6.Cube/Rollup
7.Having
8. Select
9.Distinct
10.Order by
11.Top

Practice Queries:

1. Display the dept details from department table.
2. Display the details of all employees
3. Display employee number and total salary for each employee.
4. Display employee name and annual salary for all employees.
5. Display the names of all employees who are working in department number 10
6. Display the names of all employees working as clerks and drawing a salary more than 3000
7. Display employee number and names for employees who earn commission
8. Display names of employees who do not earn any commission.
9. Display the names of employees who are working as either clerk or salesman or analyst and drawing a salary more than 3000.
10. Display the names of employees in order of salary i.e. the name of the employee earning lowest salary should appear first.
11. Display the names of employees in descending order of salary.
12. Display empno, ename, deptno and sal .Sort the output first based on name and within name by deptno and within deptno by sal;
13. Display the name of the employee along with their annual salary (sal*12).The name of the employee earning highest annual salary should appear first.

IDENTITY PROPERTY:

Identity(seed,step) property is used to generate unique number automatically.
It's a property of a column.
Seed is the starting value and step is the increment value.

CREATE TABLE TABNAME(ID INT IDENTITY(1,1),NAME VARCHAR(MAX),SAL MONEY)

Here we don't need to insert values for id column because we applied identity property to id column.

INSERT INTO TABNAME VALUES('SAI',10000),('SATYA',20000)

SELECT * FROM TABNAME

ID	NAME	SAL
1	sai	10000.00
2	satya	20000.00

Here IDs 1 and 2 are auto generated because of identity property.
Seed and step values are optional, if we don't mention also it takes seed=1 and step=1 by default. And we can give different values for both seed and step.

NOTE: the difference between delete and truncate regarding identity property is,
If we delete the table and insert new record then the new record will continue from the last record before deleted.
If we truncate the table and insert a new record then the new record starts from 1 onwards

E.g.:

DELETE:

```
CREATE TABLE TAB1(ID INT IDENTITY,NAME VARCHAR(MAX))

INSERT INTO TAB1 VALUES('SAI'),('RAJ'),('CHANDU'),('SATYA'),('NANI')

SELECT * FROM TAB1

DELETE FROM TAB1          --if we delete the table
SELECT * FROM TAB1        --no records are available now
INSERT INTO TAB1 VALUES('RAM')     --if we insert new record

SELECT * FROM TAB1
```

NOTE:now we can see data started from 6th records onwards.

TRUNCATE:

```
CREATE TABLE TAB2(ID INT IDENTITY,NAME VARCHAR(MAX))

INSERT INTO TAB2 VALUES('SAI'),('RAJ'),('CHANDU'),('SATYA'),('NANI')

SELECT * FROM TAB2

TRUNCATE TABLE TAB2       --if we truncate the table
SELECT * FROM TAB2        --no records are available now
INSERT INTO TAB2 VALUES('RAM')     --if we insert new record
```

SELECT * FROM TAB2

NOTE: Now we can see data started from 1st record onwards.

TYPES OF COLUMNS: Generally SQL provides three types of columns

1.Physical or Normal columns
2.Identity Column
3.Computed Columns

Physical Columns: The columns which are existing in the table.
Identity Column: The column which is having identity property.
Computed Columns: These are the columns which completely depends on physical columns (EMPNO, ENAME, SAL …).Computed columns can automatically arrange their values according to the user specified formula or definition. For this computed columns user no need to supply any values or any data types. These are always associated with CREATE statement.

E.g.: **CREATE TABLE Student (SNO INT IDENTITY (1, 1), SNAME VARCHAR (20), M1 INT, M2 INT, M3 INT, TOT_MAR AS (M1+M2+M3), AVG_MAR AS (M1+M2+M3)/3)**

The above create statement creates a table student with 4 Physical columns, 1 identity column and 2 Computed Columns.

ALIAS NAMES:

It is an alternative or another name for the Tables and their columns.

These Alias names are mainly used to hide the original name of the table or column.

These are always associated with SELECT statement

Note: Alias names are only for temporary displaying purpose. Those names not stored permanently into the database.

E.g.: **SELECT EMPNO *AS ENO*, ENAME *AS EMPLOYNAME*, SAL *AS BASICSAL*, DEPTNO *AS DNO* FROM EMP**

Note: While providing alias names for the columns we can use a keyword called 'AS' between original column name and alias column name

Alias names are also used to provide alternative names for the newly derived values

COLUMN ALIAS:

1. It is the other name provided for a COLUMN.
2. It is a temporary name provided for a COLUMN.
3. It provides its effect only in the output of a query.

4. It provides its usage till the query is in execution.
5. It cannot be used in other clauses of SELECT statement except ORDER BY clause.
6. When COLUMN ALIAS is specified directly, it does not allow blank spaces and special Characters except underscores.
7. In order to have blank spaces and special characters it should be enclosed in **Double Quotes " " or Square Brackets [ ]**
8. It can be specified in two ways:
   With using "AS" Key word
   Without using "AS" Key word

SELECT ENO AS EMPNO, ENAME AS EMPNAME FROM EMP
SELECT EMPNO, SAL AS MONTHLY_SALARY FROM EMP
SELECT EMPNO, SAL MONTLY_SALARY FROM EMP
SELECT EMPNO, SAL AS "MONTHLY SALARY" FROM EMP
SELECT EMPNO, SAL AS [MONTHLY SALARY] FROM EMP

USING LITERALS IN A SELECT STATEMENT

1. Literals refer to a constraint which means input provided for a query the same is provided in the output.
2. A literal can be of numeric, string and date type data

Examples:

SELECT ENAME FROM EMP
SELECT 'ENAME' FROM EMP
SELECT 'ENAME' L1 FROM EMP
SELECT 'ENAME'
SELECT 'SQL' + 'SERVER' + '2005'
SELECT 'SQL'L1, 'SERVER'L2, '2005' L3
SELECT ENAME + ' WORKING AS ' + JOB RESULT FROM EMP

TYPES OF FUNCTIONS:

1. SYSTEM DEFINED FUNCTIONS
2. USER DEFINED FUNCTIONS

SYSTEM DEFIEND FUNCTIONS:

A. SCALAR OR SINGLE-VALUED FUNCTIONS
B. GROUP OR AGGREGATE FUNCTIONS

SCALAR OR SINGLE VALUED FUNCTIONS: These functions takes single value as input and returns single value as output

CLASSIFICATION OF SINGLE VALUED FUNCTIONS:

1. Mathematical Functions

2. String Functions
3. Date & Time Functions

MATHEMATICAL FUNCTIONS:

Absolute(n): It returns the absolute value of n.

SELECT ABS(-10.37) **ANS: 10.37**
SELECT ABS(10.37) **ANS: 10.37**

Power (m, n): It returns the m power n value.

SELECT POWER(3,2) **ANS: 9**
SELECT POWER(5,3) **ANS: 125**

Sqrt(n): It returns the square root value of n.

SELECT SQRT(4) **ANS: 2**
SELECT SQRT(25) **ANS: 5**

Square (n): it returns the square value of n.

SELECT SQUARE(5) **ANS: 25**
SELECT SQUARE(2) **ANS: 4**

Round (m, n): it will round thalue of m to nearest whole number of it will around.

SELECT ROUND(4.59,1) **ANS: 4.60**
SELECT ROUND(7.6275,3) **ANS: 7.6280**
SELECT ROUND(76,275) **ANS: 76**

Ceiling(n): It returns the smallest integer greater than 'n'.

SELECT CEILING(12.8) **ANS: 13**
SELECT CEILING(12.4) **ANS: 13**

Floor (n): It returns the largest integer less than 'n'.

SELECT FLOOR(12.8) **ANS: 12**
SELECT FLOOR(12.4) **ANS: 12**

STRING FUNCTIONS:

Ascii (ch): It returns the ascii value of given character.

SELECT ASCII('A') **ANS: 65**
SELECT ASCII('A') **ANS: 97**

SELECT ASCII('1') **ANS: 49**
SELECT ASCII(' ') **ANS: 32**

Char (n): It returns the character for given ASCII value.

SELECT CHAR(65) **ANS: A**
SELECT CHAR(97) **ANS: A**
SELECT CHAR(49) **ANS: 1**
SELECT CHAR(32) **ANS: --' ONE SPACE'**

Space(n): It returns n bytes space.

SELECT SPACE(5)

Lower (string): It converts the upper case letters in string into lower cases.

SELECT LOWER('NARAYANA') **ANS: NARAYANA**

Upper (string): it converts the lower cases into upper cases.

SELECT UPPER('NARAYANA') **ANS: NARAYANA**

Len (string): it returns the length of given string.

SELECT LEN('SQLNARAYANA') **ANS: 11**
SELECT LEN('SQL NARAYANA') **ANS: 12**

Left(string,no of characters): It returns no of characters from left side of the given string.

SELECT LEFT('NARAYANA',4) **ANS: NARA**

Right(String,no.of characters): It returns no of characters from right side of the given string.

SELECT RIGHT('NARAYANA',4) **ANS: YANA**

Reverse (string): it will reverse the given string.

SELECT REVERSE('NARAYANA') **ANS: ANAYARAN**

Replace (string, 'searching string', 'replacing string'): It will replace a sub string in astring with given string.

SELECT REPLACE('NARAYANA','NARA','SQL') **ANS: SQLYANA**

Substring ('string', 'starting position', no.of characters required): It will take a part of string from main string.

SELECT SUBSTRING('SQL NARAYANA',1,3) **ANS: SQL**

Charindex('search_string','main_string'): It searches for search_string in the main string and returns search_straing's position.

SELECT CHARINDEX('Y','NARAYANA') **ANS: 5**

Stuff(given_string,starting_position, no.of characters required,replacing_string): It works like replace function but it needs starting position and no of characters to be required.

SELECT STUFF('NARAYANA',1,4,'SQL') **ANS: SQLYANA**

DATE FUNCTIONS:

date part	abbreviation
year	yy
quarter	qq
month	mm
week	ww
day	dd
hour	hh
minute	mm
size	ss
millisec	ms

getdate (): It returns the current date and time.

SELECT GETDATE() **ANS: 2016-06-28 07:11:13.743**

dateadd (date part, number, date): it returns the date according to date part.

SELECT DATEADD(MM,5,GETDATE()) **ANS: 2016-11-28 07:14:57.363**

datediff (datepart, start_date, end-date): it returns the difference between the dates according to the date part.

SELECT DATEDIFF(YY,'03/10/1989',GETDATE()) ANS: 27

datepart(part,date): It returns the part of the given date.

SELECT DATEPART(DD,GETDATE()) **ANS: 28**
SELECT DATEPART(MM,GETDATE()) **ANS: 6**
SELECT DATEPART(YY,GETDATE()) **ANS: 2016**

datename(part,date): It returns the name of the given string.

SELECT DATENAME(DW,GETDATE()) **ANS: TUESDAY**

SELECT DATENAME(MM,GETDATE()) **ANS: JUNE**

datefromparts(year,month,day): It returns date from given arguments.

SELECT DATEFROMPARTS(2010,05,01) **ANS: 2010-05-01**

datetimefromparts(year,month,day,hours,minutes,seconds,milli seconds): It returns date and time from given arguments.

SELECT DATETIMEFROMPARTS(2010,05,01,11,20,30,420) **ANS: 2010-05-01 11:20:30.420**

Practice queries

1. Display the name of employees in Uppercase.
2. Display the name of employees in Lower case.
3. Find out the length of your name using appropriate function.
4. Display the length of all employees names.
5. Display the name of the employee concatenate with empno.
6. Use appropriate function and extract 4 characters staring from 2 characters from the following string 'SQL SERVER' i.e the output should be 'QL S'.
7. Find the first occurrence of character a from the following string 'computer maintenance corporation'.
8. Replace every occurrence of alphabet A with B in the string Allen's(user translate function).
9. Display the information from emp table. Wherever job 'manager' is found it should be displayed as boss(replace function).
10. Display your age in days.
11. Display your age in months.
12. Display current date.
13. Display the following output for each row from emp table as 'scott has joined the company on Wednesday 13th august nineteen ninety'.
14. Find the date of nearest Saturday after Current day.
15. Display current time.
16. Display the date three months before the current date.
17. Delete the details of employees who are working in the company for more than 3 years.
18. Display the names of employees who are working in the company for the past 5 years.

AGGRIGATE OR GROUP FUNCTIONS: These functions takes multiple values as input and returns single value as output, these includes.

ENO	ENAME	SAL
1001	sai	1000
1002	satya	4000
1003	narayana	4000
1004	subbu	3000
1005	nani	3000

SUM(ColName): It returns the sum of all values.

SELECT SUM(SAL) FROM EMP **OUTPUT:15000**

MAX(ColName): It rerurns maximum value from all the values.

SELECT MAX(SAL) FROM EMP **OUTPUT:4000**

MIN(ColName): It returns minimum value from all values.

SELECT MIN(SAL) FROM EMP **OUTPUT:1000**

AVG(ColName): It returns average of all values.

SELECT AVG(SAL) FROM EMP **OUTPUT: 3000**

COUNT(ColName): It returns count of all the values, that means no of values in that column.

SELECT COUNT(SAL) FROM EMP **OUTPUT: 5**
SELECT COUNT(*) FROM EMP **OUTPUT:5**

COUNT_BIG(ColName): It also returns count of all values like count(), but mainly it is useful when we have huge amount of data.

SELECT COUNT_BIG(*) FROM EMP **OUTPUT: 5**

Practice queries

1. Display the total number of employees working in the company.
2. Display the total salary being paid to all employees.
3. Display the maximum salary from emp table.
4. Display the minimum salary from emp table.
5. Display the average salary from emp table.
6. Display the maximum salary being paid to CLERK.
7. Display the min salary being paid to any SALESMAN
8. Display the average salary drawn by managers.

9. Display the total salary drawn by analyst working in dept no 40.
10. Display the department name along with total salary in each department
11. Display the department no and total number of employees in each department
12. Display the various jobs and total number of employees with each job group.
13. Display department numbers and total salary for each department.
14. Display department numbers and maximum salary for each department.
15. Display the various jobs and total salary for each job.
16. Display each jobs along with minimum sal being paid in each job group.
17. Display the department numbers with more than three employees in each dept.
18. Display the various jobs along with total sal for each of the jobs where total sal is greater than 40000
19. Display the various jobs along with total number of employee in each job. The output should contain only those jobs with more than three employees.
20. Display avg sal figure for the dept
21. Find all dept's which have more than 3 employees.

CONVERSION FUNCTIONS:

Cast(FieldName as target_datatype): It converts datatype of particular column at runtime. This conversion does not affect in the database.

SELECT CAST(EMPNAME AS CHAR(100)) FROM EMP

Convert(Target_datatype, ColName,(style)): It also converts datatype of particular column like cast(), but this convert() can take third argument as style which displays date in required format.

SELECT CONVERT(CHAR(100),EMPNAME,112) FROM EMP

CONSTRAINTS:

Constraints are used to specify conditions on table columns and these can be applied while creating table or after creating table.

Different types of constraints:

1.Null
2.Not Null
3.Primary Key
4.Unique Key
5.Check
6.Default
7.Foreign Key
8.Candidate Key
9.Composite Key

NOTE: we can apply constraints while creating table and also after creating the table

Null:
Null is used to all allow null values. Null constraint is default constraint in SQL.

--While creating table
Eg : **CREATE TABLE TABNAME(ID INT, NAME VARCHAR(MAX) NULL)**

NOTE: In the above example we have two columns id and name, we didn't apply null to Id and we applied null to Name column. But two columns have null values because Id also has null by default.

--After creating table

CREATE TABLE TABNAME(ID INT, NAME VARCHAR(MAX))

ALTER TABLE TABNAME ALTER COLUMN NAME VARCHAR(MAX) NULL

NOT NULL:
Not Null is used to restrict null values.

--While creating table:

CREATE TABLE TABNAME(ID INT NOT NULL,NAME VARCHAR(MAX))

NOTE : We can't give null values for Id column because we have taken null for Id column.

--After creating table:

CREATE TABLE TABNAME(ID INT NOT NULL,NAME VARCHAR(MAX))

ALTER TABLE TABNAME ALTER COLUMN ID INT NOT NULL

PRIMARY KEY:

Primary key is mainly used to identify a record uniquely.
Primary key is used to restrict duplicate and null values.
We can apply only one primary key constraint in the table.
Generally primary key is applied on only one column, If we apply on more than one column then it is called composite primary key.

Eg :

--While creating table

CREATE TABLE PK_TAB(ID INT PRIMARY KEY,NAME VARCHAR(MAX),DEPTID INT)

--After creating table

CREATE TABLE PK_TAB1(ID INT ,NAME VARCHAR(MAX))

NOTE: Here id column has null by default, so on nullable column we cannot create primary key. Now we have to change null to not null.

ALTER TABLE PK_TAB1 ALTER COLUMN ID INT NOT NULL

NOTE : now id is having not null constraint so we can apply primary key.

ALTER TABLE PK_TAB1 ADD PRIMARY KEY(ID)

UNIQUE KEY: Unique constraint is used to identify records restrict duplicate values but it allows one null value.

--While creating table

CREATE TABLE UK_TAB(ID INT UNIQUE, NAME VARCHAR(MAX))

--After creating table

CREATE TABLE UK_TAB1(ID INT, NAME VARCHAR(MAX))

NOTE : By default Id column has null constraint in the . However unique constraint allows one null value, So directly we can create unique constraint on id column without changing to not null.

ALTER TABLE UK_TAB1 ADD UNIQUE(ID)

CHECK CONSTRAINT*:* Check constraint is used to specify specific condition on each and every row.

-- While creating table

CREATE TABLE CK_TAB(ID INT,NAME VARCHAR(MAX), SAL INT CHECK(SAL<=30000))

--After creating table

CREATE TABLE CK_TAB1(ID INT,NAME VARCHAR(MAX), SAL INT)

ALTER TABLE CK_TAB1 ADD CHECK(SAL<=3000)

Eg:
Sometimes we can see some of our friends getting more than 100 marks for 100 marks in the university exams, because the university people entered it by mistake. So here if we keep check constraint on marks like check(marks<=100) then they it will restrict when we enter more than 100 marks

DEFAULT CONSTRAINT*:* Default constraint is used to set some default value for each and every row.

--While creating table

CREATE TABLE DEF_TAB(ID INT,NAME VARCHAR(MAX), COUNTRY VARCHAR(MAX) DEFAULT 'INDIA')

--After creating table

CREATE TABLE DEF_TAB1(ID INT,NAME VARCHAR(MAX), COUNTRY VARCHAR(MAX))

ALTER TABLE DEF_TAB1 ADD DEFAULT 'INDIA' FOR COUNTRY

Eg: A MNC company went to one university to recruit the students, it decided to recruit more than 500+ students from different branches. So here they wanted to take the students information like Student Name, Branch, Section, Mobile Number, Mail Id and Collage name.
Here Collage Name is same for all the students, so entering collage name for all 500+ students is time taken process. Here if they set that collage name as default in the TABLE CREATE statement than without entering collage name for students they can get it in the table.

FOREIGN KEY*:* Foreign key is used to link between two table that implements the referencial integrity.

1. Foreign Key must be Primary Key
2. Foreign Key can accept duplicate values and Null values
3. Foreign Key has to take the values from its corresponding Primary Key

--While creating table

CREATE TABLE FK_TAB(ID INT, NAME VARCHAR(MAX), DID INT FOREIGN KEY REFERENCES PK_TAB1(ID))

--After creating table

CREATE TABLE FK_TAB1(ID INT, NAME VARCHAR(MAX), DID INT)

ALTER TABLE FK_TAB1 ADD FOREIGN KEY(DID) REFERENCES PK_TAB1(ID)

Note: To apply foreign key in one table, we must have a primary key/unique key in another table. Otherwise referential integrity is not possible.

Some imp points about PK and FK

1. Only one PRIMARY KEY is allowed per table.
2. PRIMARY KEY table is called parent table and FOREIGN KEY table is called child table.
3. PRIMARY KEY column is KEY COLUMN and the rest of the columns in the same table are called NON-KEY COLUMNS.
4. While providing FOREIGN KEY we should give the reference of its corresponding PRIMARY KEY.
5. One PRIMARY KEY can be placed on more than one column then that primary is called COMPOSITE PRIMARY KEY.

COMPOSITE KEY: If we apply any constraint on more than one column then it's called composite key.

CREATE TABLE COM_TAB(ID INT,NAME VARCHAR(MAX), DID INT PRIMARY KEY(ID,DID))

NOTE: In the above example we applied primary key on both ID and DID columns, so that combination is called composite key. Here we used primary key so it's called composite primary key.

CREATE TABLE COM_TAB1(ID INT,NAME VARCHAR(MAX), DID INT UNIQUE(ID,DID))

NOTE: In the above example we used unique key constraint, so it is called composite unique key.

SOME OTHER KEYS

SUPER KEY: super keys are used to identify tuples(records) uniquely. This super key can be applied on more than one column. Means we can also identifying recods by using StudentID and StudentName. This super key also allows single attribute to identify tuple uniquely, like StudentID

CANDIDATE KEY: if a table has EmpID, Ename, Mobile_Num, PAN_Num, AADHAR_Num, Dept_Num Here EmpID, Mobile_Num, PAN_Num, AADHAR_Num are used to identify records uniquely so among these any one can be Primary key and remaining all are alternative keys (also called secondary keys). For Eg, here EmpID is Primary key and remaining Mobile_Num, PAN_Num, AADHAR_Num are alternative keys.

ALTERNATIVE KEY(SECONDARY KEY): Alternative key is key which is also used to identify the records uniquely and all the candidate keys except primary are called alternative keys.

Constraints can be added to table in two levels.

1. **Column Level**
2. **Table Level**

1. Column level constraints: Here constraints are placed on columns, after the definition of each and every individual column and their corresponding data type.

```
CREATE TABLE DEPT
        (DEPTNO INT PRIMARY KEY,
        DNAME VARCHAR (20) UNIQUE,
        LOC VARCHAR (10) DEFAULT 'HYD')
```

2. Table level constraints: Here constraints are placed on columns after the definition of all columns and their corresponding data types. It means at the end of the table definition constraints will be placed on columns.

Note: In Table Level constraints **Primary Key Constraint**, **Unique Constraint, Check Constraint, Foreign key Constraint** *are allowed.*

E.g.:

```
CREATE TABLE DEPT
        (DEPTNO INT,
        DNAME VARCHAR (20),
        LOC VARCHAR (20),  PRIMARY KEY (DEPTNO), UNIQUE (DNAME))
```

CONSTRAINT NAMES

If you create any table without any constraint names then server will arrange the constraint names in its own format. Those constraint names will be displayed when you execute a stored procedure SP_HELPCONSTRAINT Table_Name.

SP_HELPCONSTRAINT:
This Stored Procedure is used to display the description of constraints which have been placed on different columns of a specific table.

Syntax:
SP_HELPCONSTRAINT Table-Name

Ex:
SP_HELPCONSTRAINT EMP

We can also give names to the constraints which help us while dropping the constraints.

EG: CREATE TABLE TABNAME(ID INT CONSTRAINT TABNAME_ID_PK PRIMARY KEY, NAME VARCHAR(MAX))

In the above example we have given constraint name tabname_id_pk for primary key on id column.

Generally the naming convention for constraint names is tablename_columnname_constrainttype

We can give names to all types of constraints.

CASCADING RULES:

When we have referential integrity between parent and child tables then we can't delete and update the primary key column values directly in the parent table. To delete or update the primary key vales, first we need to delete or update the foreign key values which are depended on the primary key values than only we can delete or update the primary key values.

To delete or update the primary key values directly without delete or update the foreign key values, we have cascading rules in SQL server.

SQL Server supports two types of cascading rules,

ON DELETE set null/set default/cascade/ no action

ON UPDATE set null/set default/cascade/ no action

ON DELETE SET NULL:

Without specifying the ON DELETE SET NULL it is not possible to delete the record in the parent table if there are dependent records from the child table for that record. If we use ON DELETE SET NULL on child table, when the record in the parent table is deleted than all the dependent records (only values of particular foreign key column) in the child table will be set null values.

CREATE TABLE FK1(DID INT PRIMARY KEY,NAME VARCHAR(MAX), DNO INT FOREIGN KEY REFERENCES PK1(ID) ON DELETE SET NULL)

ON UPDATE SET NULL:

Without specifying the ON UPDATE SET NULL it is not possible to update the record in the parent table if there are dependent records from the child table for that record. If we use ON UPDATE SET NULL on the child table, when the record in the parent table is updated than all the dependent records(only values of particular foreign key column) in the child table will be set null values

CREATE TABLE FK1(DID INT PRIMARY KEY,NAME VARCHAR(MAX), DNO INT FOREIGN KEY REFERENCES PK1(ID) ON UPDATE SET NULL)

ON DELETE SET DEFAULT:

Without specifying the ON DELETE SET DEFAULT it is not possible to delete the record in the PARENT table if there are dependent records from the child table for that record. If we use ON DELETE SET DEFAULT on the child table, when the record in the parent table is deleted all the dependent records (only the values of particular foreign key column) in the child table will be set default values which we specify at foreign key column with the help of default constraint.

CREATE TABLE FK1(DID INT PRIMARY KEY,NAME VARCHAR(MAX), DNO INT DEFAULT 2 CONSTRAINT FK1_DNO_FK REFERENCES PK1(ID) ON DELETE SET DEFAULT)

ON UPDATE SET DEFAULT:

Without specifying the ON UPDATE SET DEFAULT it is not possible to update the record in the PARENT table if there are dependent records from the child table for that record. If we use ON UPDATE SET DEFAULT on the child table, when the record in the parent table is updated than all the dependent records in the child table will be set default value which we specify at foreign key column with the help of default constraint.

CREATE TABLE FK1(DID INT PRIMARY KEY,NAME VARCHAR(MAX), DNO INT DEFAULT 2 CONSTRAINT FK1_DNO_FK REFERENCES PK1(ID) ON UPDATE SET DEFAULT)

ON DELETE CASCADE:

Without specifying the ON DELETE CASCADE it is not possible to delete the record in the parent table if there are dependent records from the child table for that record. If we use on delete cascade on child table, when the record in the parent table is deleted than all the dependent records in the child table will be also be deleted.

CREATE TABLE FK1(DID INT PRIMARY KEY,NAME VARCHAR(MAX), DNO INT FOREIGN KEY REFERENCES PK1(ID) ON DELETE CASCADE)

ON UPDATE CASCADE:

Without specifying the ON UPDATE CASCADE, it is not possible to update the record in the parent table if there are dependent records from the child table for that record. If we use ON UPDATE CASCADE on the child table, when the record in the parent table is updated than all the dependent records(only the values of particular foreign key column) in the child table will be also be updated

CREATE TABLE FK1(DID INT PRIMARY KEY,NAME VARCHAR(MAX), DNO INT FOREIGN KEY REFERENCES PK1(ID) ON UPDATE CASCADE)

ON DELETE SET NO ACTION:

This is the default action on the child table that's why we are unable to delete records in the parent table if there are dependent records from the child table for that record

ON UPDATE SET NO ACTION:

This is the default action on the child table that's why we are unable to update records in the parent table if there are dependent records from the child table for that record.

OPERATORS: Operator is a symbol which performs specific operation on operands or expressions.

These operators are classified into different types in SQL.

1. Assignment operators -- =
2. Arithmetic operators -- +, -, *, /, %
3. Comparison operators --<, >, <=, >=, !=, !<, !> (Relational Operators)
4. Logical operators – AND, OR, NOT, SOME, ANY, ALL
5. Set operators – UNION, UNION ALL, INTERSECT, EXCEPT
6. Special operators—IN, BETWEEN, LIKE, EXISTS,
7. Not Special operators—NOT IN, NOT BETWEEN, NOT LIKE, NOT EXISTS
8. Compound operators -- +=, -=, *=, /=, %=

Assignment Operator: It has only one operator that is '=' (equal) operator. We use this operator more frequently compare to other operators.

E.g:

SELECT * FROM EMP WHERE LOC='DALLAS'

Here we are assigning 'DALLAS' to LOC column to fetch the data that belongs to only 'DALLAS'

SELECT EMPNO,ENAME,SAL FROM EMP WHERE SAL<100000

Here we are assigning 100000 to SAL column in the table to get the data who is drawing below 100000

Arithmetic Operators: Arithmetic Operators perform mathematical operations between two or more operands of same type or different type numeric data.
There are different types of Arithmetic operators:

1. + (add)
2. – (subtract)
3. * (multiply)
4. / (divide)
5. % (module) -- it returns the remainder of the division

SELECT SAL+ISNULL(COMM,0) FROM EMP
SELECT SAL-ISNULL(COMM.,0) FROM EMP
SELECT SAL*5/100 AS COMMISION FROM EMP

SELECT 150%7

Comparison Operators: These operators are used to compare between two values.

There are different types of comparison operators:

!=	(not equal to)
>	(greater than)
<	(less than)
>=	(greater than or equal to)
<=	(less than or equal to)
!>	(not greater than)
!<	(not less than)

SELECT * FROM EMP DEPTNO LOC != 30
DELETE FROM EMP WHERE SAL >10000
UPDATE EMP SET LOC='HYD' WHERE SAL <5000
SELECT * FROM EMP WHERE SAL >= 20000
SELECT ENAME,SAL FROM EMP COMM<=1000
SELECT SAL,DETNO FROM EMP WHERE SAL !> 40000
SELECT * FROM EMP WHERE SAL !< 20000

Logical Operators: These operators are used to test two or more conditions at a time.

There are three logical operators:

AND – if we want to select the rows that must satisfy all the given conditions than we use AND.

SELECT * FROM EMP WHERE LOC='HYD' AND DEPTNO=10

Here AND Operator checks two conditions and if two conditions satisfy then SELECT returns only those rows wherever these tow conditions are met.

That means it returns all the records which belong to Location HYD and dept number 10.

OR – if we want to select the rows that satisfy any of the given conditions than we use OR.

DELETE FROM DEPT WHERE DNAME='SALES' OR LOC='DALLAS'

It deletes all the records which belong to both 'sales' dname and 'dallas' loc.

NOT -- if we want to select the rows that do not satisfy the given condition than we use NOT

NOT operator works reverse to the condition that means if the condition satisfies then it does not return rows, if the condition fails than it returns rows

SELECT * FROM DEPT WHERE NOT LOC = 'CHICAGO'

It returns all the records which are not belong to 'CHICAGO'

ALL: ALL operator compares all the values of result sub query with the values in the where clause of outer query. If that condition satisfies all the values then outer SELECT returns result set otherwise outer query will not return result set.

The ALL must be preceded by the comparison operators

SELECT * FROM EMP WHERE DEPTNO > ALL(SELECT DEPTNO FROM DEPT)

Dept table has only 10 and 20 deptnos. now inner query returns only 10 and 20.

In the outer query, deptno in the where clause compares ALL the deptnos returned in the inner query. And the outer query returns the records of deptnos which are greater than ALL the deptnos in the inner query.

In the above example, emp table has 10, 20, 30 and 40, dept table has 10 and 20
So in the emp table deptno 10 is not greater than all the deptnos (10, 20) in the dept table, so it will not return.
Deptno 20 in the emp table is not greater than all the deptnos (10, 20) in the dept table,so it will not return.
Deptno 30 in the emp table is greater than all the deptnos (10, 20) in the dept table, so it will return.
Deptno 40 in the emp table is greater than all the deptnos (10, 20) in the dept table, so it will return.

ANY: ANY operator compares the values of inner queries result set and if the condition satisfies then the outer query returns result set.

If it satisfies any value in the result set of inner query then the outer query returns records.

ANY must be preceded by comparison operators.

SELECT * FROM EMP WHERE DEPTNO > ANY (SELECT DEPTNO FROM DEPT)

In the above example, emp table has 10, 20, 30 and 40. Dept table has 10 and 20

The inner query returns 10 and 20. The deptnos in the where clause of outer query compares with 10 and 20.

If emp table deptnos are greater than any one of the deptnos than those records will be returned

Deptno 10 in the emp table is not greater than any value in the deptnos(10 and 20), so it will not return.
Deptno 20 in the emp table is greater than 10 value in the deptnos(10 and 20), so it will return.
Deptno 30 in the emp table is greater than both 10 and 20 values in the deptnos(10 and 20), so it will return.

Deptno 40 in the emp table is greater than both 10 and 20 value in the deptnos(10 and 20), so it will return.

SOME: SOME operator works like ANY operator.

Special Operators—IN, BETWEEN, LIKE, EXISTS, SOME, ANY, ALL

IN – it is used to compare a value to a list of values.

NOTE: If we compare a value to single value then we use assignment operator

SELECT * FROM EMP WHERE DEPTNO IN(10,20,30)

BETWEEN – this operator is used to search the values between the given minimum and maximum values.

SELECT * FROM EMP WHERE SAL BETWEEN 10000 AND 50000

Here it returns all the employees who are drawing salary between 10000 and 50000

Like : this operator is used to compare a value with similar values using wildcard operators. We will see these wildcard operators in the following concepts

EXISTS : The EXISTS checks the existence of a result of a Sub query. The EXISTS sub query tests whether a sub query fetches at least one row. When no data is returned then this operator returns 'FALSE'.
If it is FALSE than the outer query will not return result set.

SELECT * FROM EMP WHERE EXISTS (SELECT DEPTNO FROM DEPT)

Here sub query returns more than one row so it is valid and the outer query returns rows.

NOT SPECIAL OPERATORS—NOT IN, NOT BETWEEN, NOT LIKE, NOT EXISTS

These operators work opposite to special operators like

IN -- NOT IN
BETWEEN -- NOT BETWEEN
LIKE -- NOT LIKE
EXISTES -- NOT EXISTS

COMPOUND OPERATORS -- +=, -=, *=, /=, %=

SQL Server 2008 has introduced the new feature compound operator. Compound operators are available in other programming languages like C# etc.
Compound operators are a combination of operator with another operator.
Compound operators execute some operation and set an original value to the result of the operation.

Generally we use compound operators with variables.

+=

```
DECLARE @VAR INT = 10;
SET @VAR += 5 ;
PRINT 'ADD VALUE :' + CAST(@VAR AS VARCHAR);
```

Here variable @Var equals 10, then @Var += 5 takes the original value of @x, add 5 and sets @Var to that new value 15.

-=

```
DECLARE @VAR INT = 10;
SET @VAR -= 5 ;
PRINT 'ADD VALUE :' + CAST(@VAR AS VARCHAR);
```

Here variable @Var equals 10, then @Var -= 5 takes the original value of @x, subtracts 5 and sets @Var to that new value 5.

Like this every compound operator works based on its behavior.

SET OPERATORS:

Set operators in SQL Server are used to combine the output of multiple queries.

Whenever we want to combine the output of multiple queries we have to identify three factors.

1. Multiple queries contain equal number of columns.
2. Data types of all columns in all queries must be same.
3. Output column names must come from first query.

Eg: We have EMP and DEPT table with the following data

EMP:	Dept:
Empno	Dno
10	10
20	20
30	300
40	400
50	

In SQL Server there are four types of Set Operators

1. **UNION ALL**
2. **UNION**
3. **INTERSECT**
4. **EXCEPT**

UNION ALL: It combines the output of multiple queries including duplicate values.

Syntax: SELECT COLUMN1, COLUMN2… FROM TABLE1
UNION ALL
SELECT COLUMN1, COLUMN2… FROM TABLE2

E.g.: **SELECT EMPNO FROM EMP**
UNION ALL
SELECT DNO FROM DEPT

Output:

EMPNO
10
20
30
40
50
60
10
20

300
400

2. **UNION**: It combines the output of multiple queries without considering duplicate values, and also it arranges output data in ascending order.

Syntax: SELECT COLUMN1, COLUMN2… FROM TABLE1
UNION
SELECT COLUMN1, COLUMN2… FROM TABLE2

E.g.: **SELECT EMPNO FROM EMP**
UNION
SELECT DNO FROM DEPT

Output:

EMPNO
10
20
30
40
50
60
300
400

4. INTERSECT: It selects the common values from given set of queries.

Syntax: SELECT COLUMN1, COLUMN2… FROM TABLE1
INTERSECT
SELECT COLUMN1, COLUMN2… FROM TABLE2

E.g.: **SELECT EMPNOO FROM EMP**
INTERSECT
SELECT DNO FROM DEPT

Output:

EMPNO
10
20

4. **EXCEPT**: It selects particular values from the first query which are not available in second query..

Syntax: SELECT COLUMN1, COLUMN2… FROM TABLE1
EXCEPT
SELECT COLUMN1, COLUMN2… FROM TABLE2

E.g.: **SELECT EMPNO FROM EMP**
EXCEPT
SELECT DNO FROM DEPT

Output:

EMPNO
30
40
50
60

Practice Queries

1. Display the common jobs from department number 10 and 20.
2. Display the jobs found in department number 10 and 20 eliminate duplicate jobs.
3. Display the jobs which are unique to dept no 10.

TAKING CLONING THE TABLE:

It can be done in two different ways.

1. If the destination table is existed
2. If the destination table is not existed

How to take cloning of table's structure only:

Syn.: SELECT <*/RequiredCols> INTO NEW_TABLE FROM OLD_TABLE WHERE <False Condition>

Eg.: **SELECT * INTO NEW_TAB FROM OLD_TAB WHERE 1=2**

1. If the destination table is existed:

Syn.: INSERT INTO <NEW_TABLE> SELECT <*/RequiredCols> FROM <OLD_TABLE>

Eg.: **INSERT INTO EMP_DUMP SELECT * FROM EMP**

Here EMP is the source table and EMP_DUMP is the destination table. EMP_DUMP is having only structure but not data, so here we are simply moving data from EMP table to EMP_DUMP table.

The structure of both the tables should be same.

INSERT INTO EMP_DUMP(EMPNO,ENAME) SELECT EMPNO,ENAME FROM EMP

Here we will get data for required columns from EMP to EMP_DUMP

2. If the destination table is not existed

Syn.: SELECT <*/RequiredCols> FROM <NEW_TABLE> FROM <OLD_TABLE>

Eg.: **SELECT * INTO EMP_DUMP1 FROM EMP**

Here EMP is the source table and EMP_DUMP1 is the destination table. Initially EMP_DUMP1 table is not existed in the database. With the above query, the EMP_DUMP table will be created and also the data is copied from EMP table.

SELECT EMPNO,ENAME,SAL INTO EMP_DUMP3 FROM EMP

Here we can get only required columns from source table to destination table.

JOINS:

Joins are used to fetch the data from multiple tables.
There are different types of joins in SQL Server.
1. Inner Join
2.Outer Join
    a. Left Outer Join
    b. Right Outer Join
    c. Full Outer Join
3. Cross Join
4. Self-Join

Note: To do these joins we should maintain one common column in all the tables.

INNER JOIN: It is used to fetch matching records from all datasets

Syn: Select table1.column1, table1.column2,…….,
table2.column1, table2.column2,……
from table1 inner join table2
on table1.common column=table2.common column

Eg: **SELECT EMPNO,ENAME,SAL,**
DNO,DNAME,DLOC
FROM EMP INNER JOIN DEPT
ON EMPNO=DEPT.DNO

Naming:

Sometimes column names can be ambiguous.

For example we take two table EMP and CUSTOMERS, lets assume that each table has a column called FirstName

Select FirstName from EMP inner join Customers on eid=cid

This query fails as FirstName is in both tables, DBMS can't work out on which FirstName we want.

We can tell the DBMS by putting the table name infront of column name separated by dot

Eg: **SELECT EMP.EMPNO,EMP.ENAME,EMP.SAL,**
DEPT.DNO,DEPT.DNAME,DEPT.DLOC
FROM EMP INNER JOIN DEPT
ON EMP.EMPNO=DEPT.DNO

Aliases

If you are writing a big query, you can find yourself typing the same long table names again and again. This can lead to errors.

SQL allows us to rename tables for the duration of query.

We need to put the new name immediately after the table name in FROM clause, separated by space.

Eg: **SELECT E.EMPNO,E.ENAME,E.SAL,**
D.DNO,D.DNAME,D.DLOC
FROM EMP E INNER JOIN DEPT D
ON E.EMPNO=D.DNO

Now it looks like simple and small query

Output:

EMPNO	ENAME	SAL	DNO	DNAME	DLOC
10	SAI	10000	10	SALES	HYD
20	RAMA	20000	20	MARKETING	BANG

LEFT(OUTER) JOIN: It fetches all (matched and unmatched) records from left table and matched records from right table and right table gives null values for unmatched records from left table.

or

It fetches all matching records from both the tables and also it takes all other unmatched records from left table and for these unmatched records right table gives null values.

Syn: SELECT TABLE1.COLUMN1, TABLE1.COLUMN2,…….,
TABLE2.COLUMN1, TABLE2.COLUMN2,……
FROM TABLE1 LEFT OUTER JOIN TABLE2
ON TABLE1.COMMON COLUMN =TABLE2.COMMON COLUMN

Eg: **SELECT E.EMPNO,E.ENAME,E.SAL,**

```
        D.DNO,D.DNAME,D.DLOC
FROM EMP E LEFT JOIN DEPT D
ON E.EMPNO=D.DNO
```

Output:

EMPNO	ENAME	SAL	DNO	DNAME	DLOC
10	SAI	10000	10	SALES	HYD
20	RAMA	20000	20	MARKETING	BANG
30	SHIVA	20000	NULL	NULL	NULL
40	SATYA	30000	NULL	NULL	NULL
50	NARI	40000	NULL	NULL	NULL
60	CHINNA	50000	NULL	NULL	NULL

RIGHT (OUTER) JOIN: It fetches all (matched and unmatched) records from right table and matched records from left table and left table gives null values for unmatched records from right table.

or

It fetches all matching records from both the tables and also it takes all other unmatched records from right table and for these unmatched records left table gives null values

Syn: SELECT TABLE1.COLUMN1, TABLE1.COLUMN2,…….,
TABLE2.COLUMN1, TABLE2.COLUMN2,……
FROM TABLE1 RIGHT OUTER JOIN TABLE2
ON TABLE1.COMMON COLUMN = TABLE2.COMMON COLUMN

Eg:
```
SELECT E.EMPNO,E.ENAME,E.SAL,
D.DNO,D.DNAME,D.DLOC
FROM EMP E RIGHT JOIN DEPT D
ON E.EMPNO=D.DNO
```

EMPNO	ENAME	SAL	DNO	DNAME	DLOC
10	**SAI**	**10000**	**10**	**SALES**	**HYD**
20	**RAMA**	**20000**	**20**	**MARKETING**	**BANG**
NULL	**NULL**	**NULL**	**300**	**ANALYST**	**CHE**
NULL	**NULL**	**NULL**	**400**	**REPORTING**	**PUNE**

FULL OUTER JOIN:

It is just combination of Left outer Join + Right outer join. It selects matched records as well as unmatched records from the given tables.

SYNTAX:

SELECT TABLE1.COLUMN1, TABLE1.COLUMN2,…….,
TABLE2.COLUMN1, TABLE2.COLUMN2,……
FROM TABLE1 FULL OUTER JOIN TABLE2
ON TABLE1.COMMON COLUMN =TABLE2.COMMON COLUMN

Eg:

SELECT E.EMPNO,E.ENAME,E.SAL,
D.DNO,D.DNAME,D.DLOC
FROM EMP E FULL OUTER JOIN DEPT D
ON E.EMPNO=D.DNO

EMPNO	ENAME	SAL	DNO	DNAME	DLOC
10	sai	10000	10	sales	hyd
20	rama	20000	20	marketing	bang
30	shiva	20000	null	null	null
40	satya	30000	null	null	null
50	nari	40000	null	null	null
60	chinna	50000	null	null	null
null	null	null	300	analyst	che
null	null	null	400	reporting	pune

CROSS JOIN: Cross Join does multiplication operation between two tables and returns the result. Every record from left table multiplies with all records from right table.
If left table has 'n' records and right table has 'm' records the cross join returns m*n records

It is also known as CROSS PRODUCT or CARTESIAN PRODUCT because it produces the product of multiple tables.

Syn: SELECT TABLE1.COLUMN1, TABLE1.COLUMN2,…….
TABLE2.COLUMN1,TABLE2.COLUMN2,…….
FROM TABLE1 CROSS JOIN TABLE2

SELECT E.EMPNO,E.ENAME,E.SAL,D.DNO,D.DNAME,D.DLOC FROM EMP E CROSS JOIN DEPT D

(OR)

SELECT E.EMPNO,E.ENAME,E.SAL,D.DNO,D.DNAME,D.DLOC FROM EMP E, DEPT D

EMPNO	ENAME	SAL	DNO	DNAME	DLOC
10	SAI	10000	10	SALES	HYD
20	RAMA	20000	10	SALES	HYD
30	SHIVA	20000	10	SALES	HYD
40	SATYA	30000	10	SALES	HYD
50	NARI	40000	10	SALES	HYD

60	CHINNA	50000	10	SALES	HYD
10	SAI	10000	20	MARKETING	BANG
20	RAMA	20000	20	MARKETING	BANG
30	SHIVA	20000	20	MARKETING	BANG
40	SATYA	30000	20	MARKETING	BANG
50	NARI	40000	20	MARKETING	BANG
60	CHINNA	50000	20	MARKETING	BANG
10	SAI	10000	300	ANALYST	CHE
20	RAMA	20000	300	ANALYST	CHE
30	SHIVA	20000	300	ANALYST	CHE
40	SATYA	30000	300	ANALYST	CHE
50	NARI	40000	300	ANALYST	CHE
60	CHINNA	50000	300	ANALYST	CHE
10	SAI	10000	400	REPORTING	PUNE
20	RAMA	20000	400	REPORTING	PUNE
30	SHIVA	20000	400	REPORTING	PUNE
40	SATYA	30000	400	REPORTING	PUNE
50	NARI	40000	400	REPORTING	PUNE
60	CHINNA	50000	400	REPORTING	PUNE

SELF JOIN: Join a table with itself by providing two table alias names is called **SELF-JOIN.**
A self join is where the same table is used twice in FROM clause.

```
SELECT T1.COLUMN1, T1.COLUMN2,…….
        T2.COLUMN1,T2.COLUMN2,…….
        FROM TABLE1 T1 CROSS JOIN TABLE1 T2 WHERE CONDITION
```

Eg: We need to display emp name along with his manager name.

SELECT E.EMPNO AS EMPNO,E.ENAME AS EMPNAME,D.EMPNO AS MGRNUM, D.ENAME AS MANAGERNAME FROM EMP E, EMP D WHERE E.MGR=D.EMPNO

Q1: How to display only unmatched records from emp table

SELECT * FROM EMP LEFT JOIN DEPT ON EMP.DEPTNO=DEPT.DEPTNO WHERE DEPT.DEPTNO IS NULL

Explanation: To display only unmatched records from emp table, we have to look on dept table. If we are getting null values in the dept table means there is no matching record for the emp tables record that means it is unmatched record in the emp table. So if we take dept.deptno is null in the where clause of left join then it displays only unmatched records in the emp table

Q2: How to display unmatched records from dept table

SELECT * FROM EMP RIGHT JOIN DEPT ON EMP.DEPTNO=DEPT.DEPTNO WHERE EMP.DEPTNO IS NULL

Explanation: to display only unmatched records from dept table, we have to look on emp table. If we are getting null values in the emp table means there is no matching record for the dept table records that means it is unmatched records in the dept table. So if take emp.deptno is null in the where clause of right join then it displays only unmatched records in the dept table.

Q3: How to display only unmatched records from both emp and dept tables

SELECT * FROM EMP FULL JOIN DEPT ON EMP.DEPTNO=DEPT.DEPTNO WHERE EMP.DEPTNO IS NULL OR DEPT.DEPTNO IS NULL

Explanation: to display only unmatched records from both tables then we have to look on two tables and set emp.deptno is null or dept.deptno is null in the full join

SUB QUERIES:

A query which is defined under another query is called sub query.
A sub query is also called an inner query or inner select.
A Sub query can appear in the WHERE clause of SELECT, UPDATE and DELETE.

The following guidelines provide details about how to implement sub queries :

1. A sub query must be enclosed in parenthesis.
2. A sub query must include a SELECT clause and a FROM clause.
3. A sub query can include optional WHERE, GROUP BY, and HAVING clauses.
4. A sub query can include an ORDER BY clause only when a TOP clause is included.

Sub queries are defined two types :

1. Normal Sub queries(simply called sub queries)
2. Correlated Sub queries

Normal Sub queries: In normal sub queries, first inner select executes and based on that value the outer select will executes.

Eg:

SELECT * FROM EMP WHERE DEPTNO= (SELECT DEPTNO FROM DEPT WHERE LOC='DALLAS')

Explanation: Here first inner query that is (SELECT DEPTNO FROM DEPT WHERE LOC='DALLAS') will execute and it will return a value. Based on that value the outer query SELECT * FROM EMP WHERE DEPTNO will execute.
In the above example, in inner query, FROM clause takes all data from DEPT table and WHERE clause filters the data and it allows only 'DALLAS' data and based on that where clause the SELECT clause displays DEPTNO that is 30. So finally the result of inner query is 30.
The outer query will execute as same as inner query. FROM clause takes all the data from EMP table and WHERE clause filters the data based on the output of inner query means it takes the output of inner

select as input in the outer query like WHERE deptno=30. Finally SELECT clause displays all columns from EMP table based on the WHERE clause.

Normal sub queries are types: 1.Single value Sub queries and
2.Multi value sub queries

Single value sub query: if inner select returns single value then it is called single value sub query.

Eg:

SELECT * FROM EMP WHERE DEPTNO = (SELECT DEPTNO FROM DEPT WHERE LOC='DALLAS')

Multi value sub query: If Inner select returns more than one value then it is called multi value sub query.

SELECT * FROM EMP WHERE DEPTNO IN (SELECT DEPTNO FROM DEPT WHERE LOC='DALLAS' OR LOC='CHICAGO')

Here inner select returns more than one value so it is called Multi value sub query.

NOTE: If inner select returns only one value than we use '=' in where clause in the outer query, if inner select returns more than one value than we need to use 'in' in the where clause in the outer clause.

Some examples on sub queries:

1. **Display second highest salary.**

SELECT MAX(SAL) AS SECOND_HIGHEST_SAL FROM EMP WHERE SAL<(SELECT MAX(SAL) FROM EMP)

2. **Display all employees who are working in 'DALLAS'**

SELECT * FROM EMP WHERE DEPTNO = (SELECT DEPTNO FROM DEPT WHERE LOC='DALLAS')

3. **Display all the employees who are working in SALES department**

SELECT * FROM EMP WHERE DEPTNO=(SELECT DEPTNO FROM DEPT WHERE DNAME='SALES')

4. Display all the employees who are working under 'RESEARCH' department and drawing more than 2000 salary.

SELECT * FROM EMP WHERE SAL>2000 AND DEPTNO=(SELECT DEPTNO FROM DEPT WHERE DNAME='RESEARCH')

Explanation: In above queries we are taking only DEPTNO in inner SELECT clause and also in outer WHERE clause, because it is the only column that is common in both the tables.
When we want to work with more than one table than there must be a common column in those tables.

CORRELATED SUBQUERY:

A sub-query that uses values from the outer query. In this case the inner query has to be executed for every row of outer query. So the outer query will execute first and then the inner query will execute and based on the value from inner query the outer query will execute again. This bidirectional execution repeats like a loop.

SELECT * FROM EMP E WHERE 5=(SELECT COUNT(DISTINCT SAL) FROM EMP D WHERE E.SAL<=D.SAL)

Differences between Normal Sub Queries(Nested Queries) and Co-related Sub Queries:

In case of Co-related sub-query, sub query executes on each iteration of main query. Whereas in case of Nested-query; subquery executes first then outer query executes next.

In the case of co-related subquery, inner query depends on outer query for processing whereas in normal sub-query, Outer query depends on inner query.

Using Co-related sub-query performance decreases, since sub query executes on each iteration of main query

Practice Queries

1. Display the names of employees who are not working as managers.
2. Display the name of emp who earns highest sal.
3. Display the employee number and name of employee working as CLERK and earning highest salary among CLERKS.
4. Display the name of the salesman who earns a salary more than the highest salary of any clerk.
5. Display the names of employees who earn a Sal more than that of James or that of salary greater than that of Scott.
6. Display the employee names who are working in accountings dept.
7. Display the employee names who are working in CHICAGO.
8. Display the department where there are no employees;
9. write a query of display against the row of the most recently hired employee.
10. Display employees who can earn more than lowest sal in dept no 30
11. Find employees who can earn more than every employees in dept no 30

WILDCARD OPERATORS:

In SQL, wild card operators are used for searching the values in the table

There are two types of wild card operators in SQL: % (Percent),
_ (underscore)

% is substitute for one or more than one character.

_ is substitute for only one character.

Eg: display all the employees names along with their jobs whose name starts with 'a'

SELECT ENAME,JOB FROM EMP WHERE ENAME LIKE 'A%'

Output:

ENAME	**JOB**
ALLEN	SALESMAN
ADAMS	CLERK

Explanation: Here like operator compares 'a' with all the employee names and picks up the names which are started with 'a'. The % sign substitutes all other letters after 'a'

Eg: Display all the employees whose job ends with 'n'

SELECT EMPNO,ENAME,JOB FROM EMP WHERE JOB LIKE '%N'

Output:

EMPNO	**ENAME**	**JOB**
7499	ALLEN	SALESMAN
7521	WARD	SALESMAN
7654	MARTIN	SALESMAN
7844	TURNER	SALESMAN

Explanation: Here like operator compares 'n' with all the jobs and picks up the jobs which are ended with 'n'. The % sign substitutes all other letters before 'n'

Eg: Display all the names which are having only four letters.

SELECT ENAME FROM EMP WHERE ENAME LIKE '____'

Output:

ENAME
WARD

KING
FORD

Explanation: Here like operator compares all employee names whose names have only four characters. Because in the above example we have given four underscores. And each underscore (_) substitutes one character.

Eg: Display all employees whose name has ‘a’ in second position.

SELECT ENAME FROM EMP WHERE ENAME LIKE '_A%'

Output:

ENAME
WARD
MARTIN
JAMES

Explanation: here like operator leaves first character because one underscore is given and searches for ‘a’ in second character position and leaves all other characters after ‘a’ because % substitutes one or more characters.

Practice Queries:

1. Display all the employee names whose name starts with other than ‘a’
2. Display all the employee names whose name ends with other than ‘s’
3. Display all the employees whose name starts with 'a' or 'b' or 'c'
4. Display all the names whose name starts with 'AD'
5. Display all the employee names start between 'a' and 'm'
6. Display all the employee names whose name have 'k' any wherein the name
7. Display all the employee names whose name starts with 's' and ends with either 'h' or 't'
8. Display the names of employees whose name starts with alphabet S.
9. Display the names of employees whose name ends with alphabet S.
10. Display the names of employees whose name have second alphabet A in their names.
11. Display the names of employees whose name is exactly five characters in length.
12. List all employees which starts with either J or T.
13. List all employee names and jobs, whose job title includes M or P.
14. Display all employee names and salary whose salary is greater than minimum salary of the company and job title starts with ‘M’.

RANKING FUNCTIONS: These functions returns rank for each and every row in the result set.

SQL supports four kinds of ranking function:

1. ROW_NUMBER()
2. RANK()
3. DENSE_RANK()
4. NTILE()

Syn: Ranking_function OVER(Partition by ColumnName Order by ColumnName asc/desc)

NOTE: In the above syntax Partition by clause is optional and over() clause and order by clauses are mandatory.

By default order by clause keeps the data in ascending order.

ROW_NUMBER(): This function returns serial number to every row irrespective of duplicates values.

Eg: **SELECT ENAME,DEPTNO,SAL,ROW_NUMBER() OVER(ORDER BY SAL) AS ROW_NUM FROM EMP**

Output:

ENAME	DEPTNO	SAL	ROW_NUM
SMITH	20	800.00	1
JAMES	30	950.00	2
ADAMS	20	1100.00	3
WARD	30	1250.00	4
MARTIN	30	1250.00	5
ALLEN	30	1300.00	6
TURNER	30	1300.00	7
MILLER	10	1300.00	8
CLARK	10	2450.00	9
BLAKE	30	2850.00	10
JONES	20	2975.00	11
SCOTT	20	3000.00	12
FORD	20	3000.00	13
KING	10	5000.00	14

Explanation: Here FROM clause takes all data (all columns and all rows) from EMP table and ORDER BY clause sorts the data in ascending order based on SAL column and ROW_NUMBER() function gives serial numbers to each and every row and this serial number column we named as ROW_NUM and finally SELECT displays ENAME DEPTNO and SAL along with new column ROW_NUM.

Execution Steps:

FROM
ORDER BY
ROW_NUMBER()
SELECT

Eg: **SELECT ENAME,DEPTNO,SAL,ROW_NUMBER() OVER(PARTITION BY DEPTNO ORDER BY SAL) AS ROW_NUM FROM EMP**

ENAME	DEPTNO	SAL	ROW_NUM
MILLER	10	1300.00	1
CLARK	10	2450.00	2
KING	10	5000.00	3
SMITH	20	800.00	1
ADAMS	20	1100.00	2
JONES	20	2975.00	3
SCOTT	20	3000.00	4
FORD	20	3000.00	5
JAMES	30	950.00	1
WARD	30	1250.00	2
MARTIN	30	1250.00	3
ALLEN	30	1300.00	4
TURNER	30	1300.00	5
BLAKE	30	2850.00	6

Explanation: Here FROM clause takes all data(all columns and all rows) from EMP table and PARTITION BY clause divides the table into small partitions based on the DEPTNOs and ORDER BY clause sorts the data of each partitions, that means each partition starts sorting order from 1. ROW_NUMBER() function gives serial numbers to each partition. Finally SELECT displays ENAME, DEPTNO,SAL and ROW_NUM columns.

Execution Steps:

FROM
PARTITION BY
ORDER BY
ROW_NUMBER()
SELECT

RANK(): This function gives same rank for duplicate values and also gives gaps between ranks.

Eg: **SELECT ENAME,DEPTNO,SAL,RANK() OVER(ORDER BY SAL) AS RANK_FUN FROM EMP**

OutPut:

ENAME	DEPTNO	SAL	RANK_NUM
SMITH	20	800.00	1
JAMES	30	950.00	2
ADAMS	20	1100.00	3
WARD	30	1250.00	4
MARTIN	30	1250.00	4
ALLEN	30	1300.00	6
TURNER	30	1300.00	6
MILLER	10	1300.00	6
CLARK	10	2450.00	9
BLAKE	30	2850.00	10
JONES	20	2975.00	11
SCOTT	20	3000.00	12
FORD	20	3000.00	12
KING	10	5000.00	14

Explanation: Here FROM clause takes all data(all columns and all rows) from EMP table. Order by clause keeps the data in sorting order and RANK() gives numbers to the every row but it gives same number for duplicate values and also gives gaps.

Execution Steps:

FROM
ORDER BY
RANK()
SELECT

Eg: **SELECT ENAME,DEPTNO,SAL,RANK() OVER(PARTITION BY DEPTNO ORDER BY SAL) AS RANK_FUN FROM EMP**

OUTPUT:

ENAME	DEPTNO	SAL	RANK_FUN
MILLER	10	1300.00	1
CLARK	10	2450.00	2
KING	10	5000.00	3
SMITH	20	800.00	1
ADAMS	20	1100.00	2
JONES	20	2975.00	3
SCOTT	20	3000.00	4
FORD	20	3000.00	4
JAMES	30	950.00	1

WARD	30	1250.00	2
MARTIN	30	1250.00	2
ALLEN	30	1300.00	4
TURNER	30	1300.00	4
BLAKE	30	2850.00	6

Explanation: Here FROM clause takes all data(all columns and all rows) from EMP table. Partition by clause divides the table into small partitions based on the DEPTNOs and ORDER BY clause keeps the data of each partition in sorting order and RANK() gives numbers to each value in the each partition. RANK() gives same number to duplicate values and also gives gaps.

Execution Steps:

FROM
PARTITION BY
ORDER BY
RANK()
SELECT

DENSE_RANK(): This function gives same rank for duplicate values and does not give gaps between ranks.

Eg: **SELECT ENAME,DEPTNO,SAL,DENSE_RANK() OVER(ORDER BY SAL) AS DEN_RANK FROM EMP**

OutPut:

ENAME	**DEPTNO**	**SAL**	**DEN_RANK**
SMITH	20	800.00	1
JAMES	30	950.00	2
ADAMS	20	1100.00	3
WARD	30	1250.00	4
MARTIN	30	1250.00	4
ALLEN	30	1300.00	5
TURNER	30	1300.00	5
MILLER	10	1300.00	5
CLARK	10	2450.00	6
BLAKE	30	2850.00	7
JONES	20	2975.00	8
SCOTT	20	3000.00	9
FORD	20	3000.00	9
KING	10	5000.00	10

Explanation: Here FROM clause takes the data from EMP table and ORDER BY clause keeps the data in sorting order based on SAL column. Dense_Rank() gives numbers to the sorted data and it gives same number to duplicate values but it does not give gaps. This new column is named as DEN_RANK. Finally SELECT displays ENAME, DEPTNO, SAL and DEN_RANK columns.

Execution Steps:

FROM
ORDER BY
DENSE_RANK()
SELECT

Eg: **SELECT ENAME,DEPTNO,SAL,DENSE_RANK() OVER(PARTITION BY DEPTNO ORDER BY SAL) AS DEN_RANK FROM EMP**

OUTPUT:

ENAME	DEPTNO	SAL	DEN_RANK
MILLER	10	1300.00	1
CLARK	10	2450.00	2
KING	10	5000.00	3
SMITH	20	800.00	1
ADAMS	20	1100.00	2
JONES	20	2975.00	3
SCOTT	20	3000.00	4
FORD	20	3000.00	4
JAMES	30	950.00	1
WARD	30	1250.00	2
MARTIN	30	1250.00	2
ALLEN	30	1300.00	3
TURNER	30	1300.00	3
BLAKE	30	2850.00	4

Explanation: Here FROM clause takes all data(all columns and all rows) from EMP table and PARTITION BY clause divides the data into small partitions based on the DEPTNOs. ORDER BY clause keeps the data of each partition in sorting order. DENSE_RANK() gives numbers for each partition starting from 1 and it also gives same number for duplicate values and does not give gaps.

NTILE(n): This function divides the data into n no.of parts.

Eg: **SELECT ENAME,DEPTNO,SAL,NTILE(4) OVER(ORDER BY SAL) AS NTILE_FUN FROM EMP**

OUTPUT:

ENAME	DEPTNO	SAL	NTILE_FUN
SMITH	20	800.00	1

JAMES	30	950.00	1
ADAMS	20	1100.00	1
WARD	30	1250.00	1
MARTIN	30	1250.00	2
ALLEN	30	1300.00	2
TURNER	30	1300.00	2
MILLER	10	1300.00	2
CLARK	10	2450.00	3
BLAKE	30	2850.00	3
JONES	20	2975.00	3
SCOTT	20	3000.00	4
FORD	20	3000.00	4
KING	10	5000.00	4

Explanation: Here FROM clause takes all data(all columns and all rows) from EMP table. ORDER BY clause keeps the data in sorting order. NTILE(4) fun divides the whole data into 4 equal parts as per the number of records and also it gives same number to the whole part. For first part it gives 1 and for second part it gives 2 and so on. Finally SELECT displays ENAME, DEPTNO, SAL AND NTILE_FUN columns.

Execution Steps:

FROM
ORDER BY
NTILE()
SELECT

Eg: **SELECT ENAME,DEPTNO,SAL,NTILE(2) OVER(PARTITION BY DEPTNO ORDER BY SAL) AS NTILE_FUN FROM EMP**

Output:

ENAME	DEPTNO	SAL	NTILE_FUN
MILLER	10	1300.00	1
CLARK	10	2450.00	1
KING	10	5000.00	2
SMITH	20	800.00	1
ADAMS	20	1100.00	1
JONES	20	2975.00	1
SCOTT	20	3000.00	2
FORD	20	3000.00	2
JAMES	30	950.00	1
WARD	30	1250.00	1
MARTIN	30	1250.00	1
ALLEN	30	1300.00	2
TURNER	30	1300.00	2
BLAKE	30	2850.00	2

Explanation: Here FROM clause takes all data from EMP table and PARTITION BY clause divides the data into small parts based on the DEPTNOs. ORDER BY clause keeps the data in sorting order in every partition. NTILE(2) divides each partition into 2 parts as per number of records in each partition and this new column is named as NTILE_FUN. Finally SELECT displays ENAME, DEPTNO, SAL, NTILE_FUN columns.

Execution Steps:

FROM
PARTITION BY
ORDER BY
NTILE()
SELECT

VIEWS

A View is nothing but it is a stored select statement.

(or)

A View is nothing but an imaginary table or virtual table, which is created for a base table.
A view can be created by taking all columns from the base table or by taking only selected columns from base table.
We treat views as like normal tables when writing queries.

There are two types of user defined views in SQL Server.

1. **Simple Views**
2. **Complex Views**

1. Simple Views: Creating View by taking only one single base table.

Note: If we perform any modifications in base table, then those modifications automatically effect in view and vice-versa.

Syntax:

Create View Viewname
As
Select * From Tablename

E.g.:

```
CREATE VIEW V1
AS
SELECT * FROM EMP

INSERT INTO V1 VALUES (1,'NARAYANA', 10000,10)

CREATE VIEW V2
AS
SELECT * FROM EMP WHERE DEPTNO=10

INSERT INTO V2 VALUES (2,'NANI', 25000,10)
```

The above insert statements insert the values into base table EMP as well as into view V2.

Insert Into V2 Values (3,'Chinna', 15000, 20)

Explanation: The above insert statement inserts the values into only base table EMP but not into view V2 because in the definition of view we have given WHERE Deptno=10, so view v2 takes only the values which are having 10 as Deptno. To stop this kind of operations we have to create the view with 'WITH CHECK OPTION'.

E.g.:

CREATE VIEW V3
AS
SELECT * FROM EMP WHERE DEPTNO=10 WITH CHECK OPTION

INSERT INTO V3 VALUES (4,'TEJA', 25000,20)

The above insert statement cannot insert the values into base table EMP as well as into view V3.

SP_HELPTEXT: This is a system stored procedure which is used to display the definition of a specific view.

syntax: sp_helptext viewname

e.g.: **SP_HELPTEXT V1**

output:

```
create view v1
        as
        select * from emp
```

WITH ENCRYPTION: Once we create any view with 'WITH ENCRYPTION' then we cannot see the definition of that particular view using SP_HELPTEXT stored procedure because this encryption option hides the definition.

E.g.:

CREATE VIEW V4 WITH ENCRYPTION
AS
SELECT * FROM EMP WHERE DEPTNO=20

SP_HELPTEXT V4

Output: The text for object v4 is encrypted

To decrypt the definition of view V4 we have to follow the below approach

1. Replace CREATE with ALTER
2. Remove WITH ENCRYPTION keyword
3. Select the query and press F5.

E.g.:

ALTER VIEW V4
AS SELECT * FROM EMP WHERE DEPTNO=20

SP_HELPTEXT V4

CREATE VIEW V4
AS
SELECT * FROM EMP WHERE DEPTNO=20

2. Complex Views: Creating View by taking multiple base tables or functions or group by clause or distinct etc,.

Ex:
CREATE VIEW EMP_DEPT_VIEW
AS
SELECT EMP.EMPNO, EMP.ENAME, DEPT.DEPTNO, DEPT.DNAME FROM EMP INNER JOIN DEPT WHERE EMP.DEPTNO=DEPT.DEPTNO

Syn To Create view based on another views:

SQL SERVER allows us to create views based on another view. We can create view based on another view up to 32 levels

Syntax:
Create View Viewname [With Encryption]
As
*Select * From Viewname [Where Condition] [With Check Option]*

e.g.: **CREATE VIEW V5**
AS
SELECT * FROM V1 WHERE DEPTNO=10

Syntax To Drop the Views:

drop view viewname […n]

e.g.: **DROP VIEW V1, V2, V3, V4, V5**

WITH SCHEMA BINDING : The main benefit of WITH SCHEMA BINDING is to avoid any accidental drop or change of an object that is referenced by view

Syntax:
Create View Viewname [with schemabinding]
As
Select ColumnList From SchemaName.TableName

NOTE: when we are working with WITH SCHEMABINDING, we should not use * in the select list and also we must use schema name before table name.

e.g.:

CREATE VIEW V2 WITH SCHEMABINDING

AS

SELECT EMPNO,ENAME,SAL FROM DBO.EMP

Now we can't drop or alter the table EMP because it is being referenced by view v2.

INDEXES

Indexes in SQL server is similar to index in text book. Indexes are used to improve the performance of queries.

INDEXES ARE GENERALLY CREATED FOR FOLLOWING COLUMNS

1. Primary key column
2. Foreign key column: frequently used in join conditions.
3. Column which are frequently used in where clause
4. Columns, which are used to retrieve the data in sorting order.

INDEXED CANNOT BE CREATED FOR FOLLOWING COLUMNS:

1. The columns which are not used frequently used in where clause.
2. Columns containing the duplicate and null values
3. Columns containing images, binary information, and text information.

TYPES OF INDEXES:

- **CLUSTERED INDEX**
- **NON-CLUSTERED INDEX**

CLUSTERED INDEX: only one clustered index is allowed per table. The order of values in a table order of values in index is also same. When cluster index is created on table data is arranged in ascending order cluster index will occupy 5% of the table.

Syntax:

CREATE [UNIQUE] CLUSTERED INDEX INDEXNAME ON TABLENAME (COLUMN)

E.g.:

CREATE CLUSTERED INDEX CI ON EMP (EMPNO)

Note: if we want to maintain unique values in clustered/non clustered indexed column then specify UNIQUE keyword along with CLUSTERED INDEX/NONCLUSTERD INDEX

NONCLUSTERED INDEX: It is the default index created by the server the physical order of the data in the table is different from the order of the values in index.

Max no. Of non-clustered indexed allowed for table is 249

Syntax:
CREATE [UNIQUE] NONCLUSTERED INDEX INDEXNAME
ON TABLENAME (COLUMN1,…)

E.g.:
CREATE NONCLUSTERED INDEX NCI ON EMP (ENAME, SAL)

Ex:
CREATE UNIQUE NONCLUSTERED INDEX UI ON DEPT (DNAME)

COMPOSITE INDEX: If a **Unique NonClustered index** is created on more than one column then that concept is called composite index.

CREATE UNIQUE NONCLUSTERED INDEX COI ON DEPT (DEPTNO, DNAME)

DEPTNO	**DNAME**
10	SALES
20	HR
30	IR
10	HR (Accepted)
20	SALES (Accepted)
30	IR (Repeated, Not accepted)

SP_HELPINDEX:

This system defined stored procedure is used to display the list of indexes, which have been placed on different columns of a specific table.

E.g.: SP_HELPINDEX EMP

Syntax to drop the index:

DROP INDEX TABLENAME.INDEX.NAME

E.g.:

DROP INDEX DEPT.UI

The **characteristics** of the indexes are:

1. They fasten the searching of a row.

2. They are sorted by the Key values.
3. They are small and contain only a small number of columns of the table.
4. They refer for the appropriate block of the table with a key value

Purpose of indexes.

1. Allow the server to retrieve requested data, in as few I/O operations
2. Improve performance
3. To find records quickly in the database

When an index is appropriate.

1. When there is large amount of data. For faster search mechanism indexes are appropriate.
2. To improve performance they must be created on fields used in table joins.
3. They should be used when the queries are expected to retrieve small data sets
4. When the columns are expected to a nature of different values and not repeated
5. They may improve search performance but may slow updates.

Create Index:
CREATE INDEX index_name ON table_name (col_1,col_2..);

Example:
CREATE INDEX INDEX_SAMPLE ON EMP(EMPNO)

Rename Index:
EXEC sp_rename 'Table_name.Index_name_old', 'Index_name_new'

Example:
EXEC SP_RENAME 'EMP.INDEX_SAMPLE','SAMPLE'

Delete index:
DROP INDEX INDEX_NAME ON TABNAME

Example:
DROP INDEX SAMPLE ON EMP

TEMP TABLES

Temp tables are used to store the data temporarily.
Temp tables are two types.
1. Local Temp Tables
2. Global Temp Tables

NOTE: these tables are same as normal tables but while creating temp tables we use # symbol before table name.
For Local Temp Table we use single # (like #TableName)
For Global Temp Table we use double #(like ##tableName)

Local Temp Tables: These are useful to store the data which can be used for that particular session only.

E.g.:

```
CREATE TABLE #TAB1(ID INT,NAME VARCHAR(MAX),SAL INT)
INSERT INTO #TAB1 VALUES(1,'SAI',1000),(2,'RAM',2000)
DELETE FROM #TAB1 WHERE ID=1
UPDATE #TAB1 SET SAL=3000 WHERE ID=2
ALTER TABLE #TAB1 ALTER COLUMN ID BIGINT
TRUNCATE TABLE #TAB1
DROP TABLE #TAB1
SELECT * FROM #TAB1
```

Like this we can use all DDL and DML operations on temp tables as same as normal tables.

Global Temp Tables: These are useful to store the data which can be used in all sessions.

NOTE: If we remove the session where the local and global temp tables are created then automatically those temp tables are deleted.

E.g,

```
CREATE TABLE ##TAB1(ID INT,NAME VARCHAR(MAX),SAL INT)
INSERT INTO ##TAB1 VALUES(1,'SAI',1000),(2,'RAM',2000)
SELECT * FROM ##TAB1
DELETE FROM ##TAB1 WHERE ID=1
UPDATE ##TAB1 SET SAL=3000 WHERE ID=2
ALTER TABLE ##TAB1 ALTER COLUMN ID BIGINT
TRUNCATE TABLE ##TAB1
DROP TABLE ##TAB1
SELECT * FROM ##TAB1
```

Like this we can all DDL and DML operations on temp tables as same as normal tables.

TABLE VARIABLES:

Microsoft introduced table variables in SQL Server 2000 for an alternative solution for temp tables. These table variables are used to store the data temporarily.

Table variables are created with DECLARE statement as like CREATE TABLE statement.

We can use table variables in stored procedures, user defined functions and batches. The table variables are no longer exist after the stored procedure, user defined functions or batches exists.

We can work with all DML operations in table variables as like in the normal tables.

Table variable expires automatically so we don't need to use DROP statement.

```
DECLARE @VAR TABLE
                ( ID INT,
                  NAME VARCHAR(MAX),
                  LOC VARCHAR(MAX)
                )

INSERT INTO @VAR VALUES (1,'SAI',1000),
            (2,'SATYA',2000),
                        (3,'KUMAR',3000),
                        (4,'ROSE',5000),
                        (5,'CHANDRA',2000)

SELECT * FROM @VAR

UPDATE @VAR SET NAME='RAJU' WHERE ID=5
SELECT * FROM @VAR

DELETE FROM @VAR WHERE ID=5
SELECT * FROM @VAR
```

NOTE: While running table variable we need to select the entire code that belongs to particular table variable.

MERGE STATEMENT: Merge statement was introduced in 2008. We can perform insert, delete and update operations in a single statement called Merge Statement.
Merge statement gives more efficient result because it is used instead of multiple operations.

Syn:
Merge [into] <Target Table> <TT alias name>
Using <Source Table> <ST alias name>
On<Join>
When matched then
Update set tt.clo1=st.col1 <UPDATE Statement>
tt.col2=st.col2
tt.col3=st.col3…

When not matched then <INSERT Statement>
insert (col1,col2,col3…) values(st.col1,st.col2,st.col3…)

When not matched by source then <DELETE Statement>
delete ;

NOTE: A MERGE statement must be terminated by a semi-colon (;).

Note: We are adding new column LOCATION to EMP Table and now we will update the this LOCATION column in the emp table with LOC column in DEPT Table

Alter table emp add Location varchar(max)

Eg:
```
MERGE INTO EMP
USING DEPT
ON DEPT.DEPTNO=EMP.DEPTNO
WHEN MATCHED THEN
UPDATE SET EMP.LOCATION=DEPT.LOC
WHEN NOT MATCHED THEN
INSERT (LOCATION) VALUES(DEPT.LOC)
WHEN NOT MATCHED BY SOURCE THEN
DELETE ;
```

Eg:

```
MERGE INTO EMP1 E1
USING EMP E
ON E.EMPNO=E1.EMPNO
WHEN MATCHED THEN
```

```
UPDATE SET E1.EMPNO=E.EMPNO,
           E1.ENAME=E.ENAME,
           E1.JOB=E.JOB,
           E1.SAL=E.SAL,
           E1.DEPTNO=E.DEPTNO
WHEN NOT MATCHED THEN
INSERT (EMPNO, ENAME, JOB, SAL, DEPTNO)
 VALUES(E.EMPNO,E.ENAME,E.JOB,E.SAL,E.DEPTNO)
WHEN NOT MATCHED BY SOURCE THEN
DELETE;
```

Note: The insert column list used in the MERGE statement cannot contain multi-part identifiers. We have to use single part identifiers instead.

Note: Merge Statement can be used in both OLTP and Data Warehouse environments
In OLTP, we can merge recent information from external source and
In DW, we can use merge statement for incremental updates of fact, slowly changing dimensions.

CTE (Common Table Expression)

The Common Table Expression (CTE) is called as a temporary named RESULT SET, within the scope of an executing statement that can be used within a SELECT, INSERT, UPDATE, or DELETE or even in a CREATE VIEW or MERGE statement.

CTEs are one of the beautiful and the most powerful feature of SQL Server. It is introduced in SQL Server 2005. This not only simplifies most of the complex and impossible queries in T-SQL, but also provides one medium to deal with recursive queries. Moreover, I can say it is mainly for recursive queries. It can be used in replacement of derived tables (sub queries), temp tables, table variables, inline user-defined functions etc.

CTE Syntax:

[WITH <CTE_Name>]
[Columns_list **)**]
AS
(Select Statement for CTE definition **)**
Select/delete/update/insert/create view statement

Eg 1:

```
WITH CTENAME
([EMPLOYEE NUMBER],[EMPLOYEE NAME],SALARY)
AS
```

```
(SELECT EMPNO,ENAME,SAL FROM EMP WHERE SAL>=3000)
SELECT * FROM CTENAME
```

OUTPUT:

Employee number	employee name	salary
7788	SCOTT	3000.00
7839	KING	5000.00
7902	FORD	3000.00

Eg 2:

```
WITH EMP_CTE
 AS
 (
   SELECT ENAME, EMPNO, SAL, DNAME,LOC
   FROM EMP A
   INNER JOIN
  DEPT D ON A.DEPTNO = D.DEPTNO
)
SELECT * FROM EMP_CTE WHERE LOC = 'DALLAS';
```

OUTPUT:

Ename	Empno	Sal	Dname	Loc
SMITH	7369	800.00	RESEARCH	DALLAS
JONES	7566	2975.00	RESEARCH	DALLAS
SCOTT	7788	3000.00	RESEARCH	DALLAS
ADAMS	7876	1100.00	RESEARCH	DALLAS
FORD	7902	3000.00	RESEARCH	DALLAS

Eg 3: CTE with Union All

```
with cte_union
(col1,col2)
as
(
select 1 as col1, 3 as col2
union all
select 5 as col11,7 as col22
)
select * from cte_union
```

OUTPUT:

COL1	COL2
1	3
5	7

DERIVED TABLES:

These tables are created on the fly with the help of the Select statement.

If the sub query is written in the FROM clause of outer query then that sub query is used as a source table to that outer query. So this table (sub query) is called derived table.

These derived tables are different from the temp tables because in case of temporary tables, we have to create, insert, select and drop the temporary tables. But in case of derived tables, SQL Server itself creates and populates the table in the memory and we can directly use these derived tables. We do not need to drop it. But it can only be referenced by the outer Select query who created it. Also since it is reside in the memory itself, it is faster than Temporary table which are created in the temp database.

Syn: select * from (select statement)

Explanation: Generally we use table name in FROM clause but here another select statement is used in place of table. So this select statement acts as a table to the outer query. That's why this inner select statement is treated as derived table.

Eg:

```
SELECT EMPNO, ENAME,  SAL, DEPTNO  FROM (
SELECT EMPNO,
        ENAME,
        SAL,
        DEPTNO,
        ROW_NUMBER() OVER(PARTITION BY DEPTNO ORDER BY SAL DESC) AS ROW FROM EMP)AS
DERIVED_TAB
WHERE ROW=1
```

Here the inner select gives the result like in a table format and that table is used as a source table to the outer query. So this inner query is called derived table

SEQUENCE :

Microsoft introduced SEQUENCE in SQL Server 2012.

Sequence is used to generate an auto-increment number.

Sequence is a database object.

It is like alternative for IDENTITY property.

We can create SEQUENCE with CREATE statement.

Syn:

create sequence <sequence_name>
as int data type
Start with <constant> -- represents the start values
increment by <constant> --represents the increment value means step size
minvalue <constant> /no minvalue --represents the minimum value of the sequence
maxvalue<constant> /no maxvalue --represents the maximum values of the sequence
cache <constant> /no cache --specifies how many sequence values will be stored in the memory for faster access.
Cycle /no cycle --specifies whether the sequence object should restart after the minimum or maximum values is exceeded or throw an exception.
By default it is no cycle

Eg:

```
CREATE SEQUENCE MY_SEQ
AS INT
START WITH 1
INCREMENT BY 1
MINVALUE 1
MAXVALUE 100
CACHE 10
CYCLE
```

Explanation: Here in the above example we are creating a sequence with name MY_SEQ.

MINVALUE of sequence is 1 and MAXVALUE is 100.

The START WITH value must be greater than or equal to minimum value and less than or equal to maximum value. So here we have given 1 for START value.

The value of the INCREMENT for sequence must be less than or equal to the difference between the minimum and maximum value of the sequence object. So here we have given 1.

The value of CACHE is given 10 that means 10 values will be stored in the memory for faster access. After first 10 rows, again it will take next 10 rows. Like this the CACHE takes all rows.

We have mentioned min value and max values, if want access the rows beyond this boundary also than we have to keep CYCLE. If we don't need values beyond the given boundary and need to throw an exception if exceeds the boundary than we have to set NO CYCLE.

```
CREATE TABLE MY_TAB( ID INT,
                     NAME VARCHAR(MAX),
                     SAL INT
                   )

INSERT INTO MY_TAB VALUES(NEXT VALUE FOR MY_SEQ,'A',1000)
INSERT INTO MY_TAB VALUES(NEXT VALUE FOR MY_SEQ,'B',2000)
INSERT INTO MY_TAB VALUES(NEXT VALUE FOR MY_SEQ,'C',3000)
INSERT INTO MY_TAB VALUES(NEXT VALUE FOR MY_SEQ,'D',5000)
INSERT INTO MY_TAB VALUES(NEXT VALUE FOR MY_SEQ,'E',2000)
INSERT INTO MY_TAB VALUES(NEXT VALUE FOR MY_SEQ,'F',3000)

SELECT * FROM MY_TAB
```

ID	NAME	SAL
1	A	1000
2	B	2000
3	C	3000
4	D	5000
5	E	2000
6	F	3000

We have to write NEXT VALUE FOR SEQUENCE_NAME to insert new value for every record like in the above example.
We can also enter numbers manually without using SEQUENCE in the same table.

```
INSERT INTO MY_TAB VALUES(7,'G',3000)
INSERT INTO MY_TAB VALUES(8,'H',1000)

SELECT * FROM MY_TAB
```

ID	NAME	SAL
1	A	1000
2	B	2000
3	C	3000

4	D	5000
5	E	2000
6	F	3000
7	G	3000
8	H	1000

And we can continue sequence for other records. So if we want to enter the records we can use sequence or else we can also enter manually.
If we use sequence again than it will start from 7 onwards because it was stopped at 6.

INSERT INTO MY_TAB VALUES(NEXT VALUE FOR MY_SEQ,'I',2000)
INSERT INTO MY_TAB VALUES(NEXT VALUE FOR MY_SEQ,'J',6000)

SELECT * FROM MY_TAB

ID	**NAME**	**SAL**
1	A	1000
2	B	2000
3	C	3000
4	D	5000
5	E	2000
6	F	3000
7	G	3000
8	H	1000
7	I	2000
8	J	6000

If the ID value exceeds 100 than again it starts from 1 onwards because in CREATE SEQUENCE statement we used CYCLE.

We can alter sequence with ALTER in place of CREATE.

ALTER SEQUENCE MY_SEQ
RESTART WITH 1
INCREMENT BY 5
MINVALUE 1
MAXVALUE 50
CACHE 10
CYCLE

We can modify any value. In ALTER statement we can not use AS INT and we need to use RESTART in place of START.

We can also modify each and every value in individual ALTER statements.

ALTER SEQUENCE MY_SEQ RESTART WITH 10

ALTER SEQUENCE MY_SEQ INCREMENT BY 1

ALTER SEQUENCE MY_SEQ MINVALUE 10

ALTER SEQUENCE MY_SEQ MAXVALUE 500

ALTER SEQUENCE MY_SEQ CACHE 50

ALTER SEQUENCE MY_SEQ NO CYCLE

PAGINATION:

Microsoft introduced Pagination in SQL Server 2012.
In previous versions also we have done this data paging by writing stored procedures or complex queries, but SQL Server 2012 onwards we have this excellent feature for data paging but here we need to use two keywords like OFFSET and FETCH.

We must use ORDER BY clause in Pagination

Syn:

Select< ColList/*> from <TabName> Order by< ColName>
OFFSET <n> rows
FETCH next <m> rows only.

OFFSET indicates that number of rows to be skipped
FETCH NEXT indicates that number of rows to be fetched from that skipped rows onwards.

NOTE: I have added new column EID to the EMP table for sequence number purpose.

ALTER TABLE EMP ADD EID INT IDENTITY(1,1)

Eg: **SELECT EID,EMPNO,ENAME FROM EMP ORDER BY EMPNO**
OFFSET 4 ROWS
FETCH NEXT 3 ROWS ONLY

EID	**EMPNO**	**ENAME**
5	7654	MARTIN
6	7698	BLAKE
7	7782	CLARK

Here FROM clause takes all the data(all columns and rows) from EMP table and ORDER BY clause keeps the data in sorting order by using EMPNO.
OFFSET skips the top 4 rows and FETCH fetches next 3 rows starting from 5^{th} row onwards.
SELECT displays those three rows along with EID, EMPNO and ENAME columns.

Execution Order:

FROM
ORDER BY
OFFSET
FETCH
SELECT

SYNONYMS: Microsoft introduced Synonyms in SQL Server 2005.
A synonym is a database object that provides an alternative name for the database object.

A synonym provides a layer of abstraction that protects a client application from changes made to the name or location of the base object.

For example we have Server1, database1, schema1 and table1 and now we need to access the table1 in Server2 to perform some query operations.

You have to write fully qualified name (four part name) like Server1.database1.schema1.table1 so here we have to use Server name, database name, schema name and table name.

Instead of this, we can create synonym like *create synonym SYN1 for Server1.database1.schema1.table1*
So in Server2 you can access like *select * from table2 t2 inner join syn1 s1 on t1.id=s1.id*

So it gives the advantages of Abstraction, Ease of change, scalability.
Later on if you want to change Server name or Schema or table name, just you have to change the synonym alone and there is no need for you do search all and replace them.

A synonym is a single-part name which can replace multi part name in SQL Statement.
A synonym can be used with only SELECT, UPDATE, INSERT, DELETE, EXECUTE commands.

Multi part name means: Schema_name.Table_name --half qualified name
(OR)
Database_name.Schema_name.Table_name -- half qualified name
(OR)
Server_name.Database_name.Schema_name.Table_name -fully qualified name

Syn: CREATE SYNONYM Synonym_Name FOR Table_Name
Eg: **CREATE SYNONYM SYN1 FOR EMP**
Or
CREATE SYNONYM SYN2 FOR SATYA.SQLDB.DBO.EMP

We can create SYNONYMs for

1. Tables
2. Views
3. Assembly Stored Procedures, Table Valued Functions, Aggregations
4. SQL Scalar Functions
5. SQL Stored Procedures
6. SQL Table Valued Functions
7. SQL Inline-Table-Valued Functions
8. Local and Global Temporary Tables
9. Replication-filter-procedures
10. Extended Stored Procedures

STORED PROCEDURES:

Stored procedures are one of the great features in SQL.

Stored procedure is nothing but a set of precompiled code which can be used over and over again.

If we think that the code needs to be used over and over again in the future then we can save that code as a stored procedure and simply we can call that stored procedure when we need that code.

We can also pass parameters to the stored procedures.

We create stored procedures by using CREATE PROCEDURE statement.

Syn:

```
Create Procedure Proc_name
(
Input Parameters,
Output Parameters(if needed)
)

As
Begin
      SQL statements need to be used in Stored Procedure
End
```

Eg: Display employee name, job and salary for whose employee number is passed.

```
CREATE PROC PROC1(@ENO INT)
AS
BEGIN
SELECT ENAME,JOB,SAL FROM EMP WHERE EMPNO=@ENO
END
```

After writing stored proc we need to run this whole code so that it will be stored in the name of PROC1. Now we can all PROC1 again and again instead running the same code always.

Executing the Stored Proc:

Syn: EXEC/EXECUTE Proc_name <values>

NOTE: Without using EXEC/EXECUTE also we can run only proc name with parameter values.

Eg: **EXECUTE PROC1 7369**
OR
EXEC PROC1 7369
OR
PROC1 7369

Output:

ENAM	**E**	**JOB**	**SAL**
SMITH		CLERK	800.00

Eg: Write a procuedure with parameters

```
CREATE PROCEDURE PROC2(@A INT,@B INT)
AS
BEGIN
DECLARE @Z INT
SET @Z= @A+@B
PRINT 'SUM OF THE VALUES IS '+CAST(@Z AS VARCHAR(10))
END

EXEC PROC2 100,50
        OR
EXEC PROC2 @A=100, @B=50
        OR
EXEC PROC2 @B=50, @A=100
```

Eg: Write a procedure with default values to parameters

```
CREATE PROCEDURE PROC3(@X INT=100,@Y INT)
AS
BEGIN
DECLARE @Z INT
SET @Z=@X+@Y
PRINT 'SUM OF THE VALUES IS '+CAST(@Z AS VARCHAR(20))
END
```

EXEC PROC3 200,300 **OUTPUT: SUM OF THE VALUES IS 500**
EXEC PROC3 DEFAULT,400 **OUTPUT: SUM OF THE VALUES IS 500**
EXEC PROC3 @X=DEFAULT,@Y=100 **OUTPUT: SUM OF THE VALUES IS 200**
EXEC PROC3 @Y=100 **OUTPUT: SUM OF THE VALUES IS 200**

Note: Server will give highest priority to the user supplied values rather than default values

Q: Write procedure using both input and output parameters

```
CREATE PROCEDURE BOTHPARAMS_SP(@A INT,@B INT, @C INT OUT,@D INT OUT)
AS
BEGIN
SET @C=@A+@B
SET @D=@A-@B
END

DECLARE @X INT,@Y INT
EXEC BOTHPARAMS_SP 20,10, @X OUT,@Y OUT
PRINT @X
PRINT @Y
```

Eg: Write a procedure to display total salary employee whose employee number is passed.

```
CREATE PROC PROC3(@EMPNO INT)
AS
BEGIN
SELECT 'TOTAL SALARY OF'+SPACE(1)+ ENAME+SPACE(1)+'IS' +SPACE(1)+CAST(SAL+ISNULL(COMM,0)
AS VARCHAR(MAX)) FROM EMP WHERE EMPNO=@EMPNO
END

EXEC PROC3 7499
```

Output:

TOTAL SALARY OF ALLEN IS 1600.00

Eg: Write a procedure to insert employee name, job ,sal and deptnos into the EMP table

```
CREATE PROC PROC2
(
@NAME VARCHAR(MAX),
@JOB VARCHAR(MAX),
```

```
@SAL INT,
@DEPTNO INT
)
AS
BEGIN
INSERT INTO EMP(ENAME,JOB,SAL,DEPTNO) VALUES(@NAME,@JOB,@SAL,@DEPTNO)
END

EXEC PROC2 'SAI','ADMIN',2000,40
```

E.g.: Write a procedure to select the data from EMP table based on user supplied DEPTNO.

```
CREATE PROCEDURE P2 @X INT
AS
BEGIN
SELECT * FROM EMP WHERE DEPTNO=@X
END

EXEC P2 20
```

We can modify the code inside the stored proc by using ALTER keyword.

```
ALTER PROC PROC1(@ENO INT)
AS
BEGIN
SELECT ENAME,JOB,SAL,DEPTNO FROM EMP WHERE EMPNO=@ENO
END
```

Usages of Stored Procs:

1. We can compile Stored proc only once instead of compiling the code again and again.
2. We can provide stored procs to the users and grant permission to that stored proc instead of giving permission to the tables used in that stored proc.
3. Stored procs are very helpful in providing the reusability of the code. Because multiple user can use the same procedure.
4. By using stored proc we can reduce the amount code sent to the Server if the bandwidth of the Server is less

Some System Defined Stored Procedures:

SP_RENAMEDB: Here SP stands for Stored Procedure. This stored procedure is used to change the name of the existing database.

Syntax: SP_RENAMEDB 'OLD DATABASENAME', 'NEW DATABASENAME'

E.g.: SP_RENAMEDB 'NRSTT', 'NRSTTS'

The above statement renames (changes the database name) NRSTT to NRSTTS

SP_RENAME: This stored procedure is used for changing the name of the table and for changing the name of the column

i. Syntax to change the name of the table

SP_RENAME 'OLD TABLENAME', 'NEW TABLENAME'

E.g. **SP_RENAME 'EMP', 'EMPLOY'**

The above stored procedure changes the name of EMP table to EMPLOY

ii. Syntax to change the name of the column

SP_RENAME 'TABLE.OLDCOLUMN NAME', 'NEW COLUMNNAME'

E.g. SP_RENAME 'STUDENT.ADR', 'ADDRESS'

The above stored procedure changes the name of ADR column to ADDRESS in STUDENT table.

SP_HELP: This stored procedure is used to display the description of a specific table.

Syntax: SP_HELP TABLENAME

E.g.: SP_HELP EMP

The above stored procedure displays the description of EMP table

SP_DATADASES: This Stored procedure displays the list of databases available in SQL Server.

Syntax: SP_DATABASES

SP_TABLES: This stored procedure displays the list of tables available in the current database.

Syntax: SP_TABLES

SP_HELPDB: This stored procedure is used to display the description of master and log data file information of a specific database

Syntax: SP_HELPDB Database-Name

Ex: SP_HELPDB SAMPLE

SP_SPACEUSED: This stored procedure is used to find the memory status of the current database

Syntax: SP_SPACEUSED

SP_HELPCONSTRAINT

SP_HELPTEXT

FUNCTIONS: (user defined functions)

When we want perform complex logics which cannot be written in single statement then User Defined Functions are useful.

User defined functions accepts parameters and return single value or result set.

Some points on functions:

1. User defined functions compile every time.
2. User defined functions are helpful when we need to write complex logics.
3. User defined functions can have only input parameters.
4. In user defined functions we can't insert or delete or update the data in the database tables.
5. User defined functions do not support exception handling concept.
6. User defined functions can be nested up to 32 levels.

SQL Server supports two kinds of User defined functions based on the return values:

1. Scalar UDF
2. Table valued UDF

Scalar UDF: Scalar UDF returns single /scalar value of any datatype.

syn:

```
create function function_name
        ( input parameters
        )
Returns datatype
Begin
Return (select statements)
End
```

Calling function:

Syn: Select dbo.fun_name(value)

Eg: Write a function to display total salary of employee whose numbers is passed.

```
CREATE FUNCTION FUN1( @ENO INT)
RETURNS INT
AS
BEGIN
RETURN (SELECT SAL+ISNULL(COMM,0) AS TOTAL_SAL FROM EMP WHERE EMPNO=@ENO)
END
```

SELECT DBO.FUN1(7499) AS TOTAL_SAL

E.g.: Write a function to find the product of two numbers

```
CREATE FUNCTION F1 (@A INT, @B INT)
RETURNS INT
AS
BEGIN
DECLARE @C INT
SET @C = @A * @B
RETURN @C
END
```

SELECT/PRINT DBO.F1 (3,5)

Note: we can use print or select to call the function.

Table valued UDF: Table valued UDFs return more than a value or full result set.
Table valued UDFs are two types:

1. Inline table valued UDFs
2. Multi statement table valued UDFs

Inline table valued UDFs: the user defined inline table valued function returns a table variable as a result of actions performed by the function.
Inline table valued function does not have BEGIN-END block

syn:
Create function fun_name()
Returns table
As
Return (select statement)

Calling Function:
Syn: Select */col_list from schemaname.fun_name

Eg:

```
CREATE FUNCTION INLINE_FUN()
RETURNS TABLE
AS
RETURN (SELECT EMPNO,ENAME,SAL FROM EMP)
```

SELECT * FROM DBO.INLINE_FUN()

Multi statement table valued UDF: multi statement table valued UDF returns a table variable as a result of action performed by the function.

In multi statement table valued UDF, we need to create the table variable explicitly and also we need to define values which are derived from the multiple statements

Syn:
Create function fun_name()
Returns TableVariable_name Table(col list)
As
Begin
SQL Statements
End

Calling Function:
Syn: select */colList from Fun_Name()

Eg:

```
CREATE FUNCTION MULTI_FUN()
RETURNS @EMP TABLE(EMPNO INT,
                   ENAME VARCHAR(MAX),
                    JOB VARCHAR(MAX),
                   SAL INT
                  )
AS
BEGIN
INSERT INTO @EMP SELECT EMPNO,ENAME,JOB,SAL FROM EMP
DELETE FROM @EMP WHERE EMPNO IS NULL
UPDATE @EMP SET SAL=10000 WHERE EMPNO=7369
RETURN
END

SELECT * FROM MULTI_FUN()
```

Explanation:
Here we are creating table variable @EMP with some columns
In begin—end block, we are copying empno, ename,job and sal from emp table.
We are deleting records whose empno is null, this delete operation works only on table variable @EMP not on main table EMP.
The update statement also works on table variable @EMP but not on EMP table

CURSORS:

Generally T-SQL commands operate on all the rows at a time in the result set.
Cursor is a database object which is used to retrieve a row at a time from a set of rows together.
Cursors are useful when we need to apply DML operations on data in a row by row basis.

Cursor has 5 steps:

1. ***Declare*** Cursor: in DECLARE statement we need to define cursor by executing SELECT Statement
2. ***Open*** Cursor: in OPEN cursor statement we need to open the cursor which is already defined by using DECLARE statement.
3. ***Fetch*** from Cursor: here we need to fetch rows from the cursor which is opened by using OPEN statement. We do data manipulations on these rows.
4. ***Close*** Cursor: here we need to close the cursor after it is used. Once cursor is closed it can be reopened.
5. ***Deallocate*** Cursor: here we need to deallocate cursor definition so that it will release all system resources which are associated with cursor. Once cursor is deallocated, it can not be reallocated.

Syn to Cursor:

```
DECLARE cursor_name CURSOR
        [LOCAL | GLOBAL]
        [FORWARD_ONLY | SCROLL]
        [STATIC | KEYSET | DYNAMIC | FAST_FORWARD]
        [READ_ONLY | SCROLL_LOCKS | OPTIMISTIC]
         FOR select_statement
         FOR UPDATE [col1,col2,...coln]
OPEN [GLOBAL] cursor_name
FETCH [NEXT|PRIOR|FIRST|LAST|ABSOLUTE n|RELATIVE n]
         FROM [GLOBAL] cursor_name
         INTO @Variable_name[1,2,..n]
CLOSE cursor_name
DEALLOCATE cursor_name
```

Here:

Local/global	--these define the scope of cursor
forward_only/scroll	--these define the movement of cursor
static/keyset/dynamic/fast_forward	--these are the types of cursor
read_only/scroll_locks/optimistic	--these define locks

FIRST: Fetches first record from the cursor

NEXT: Fetches next record from the current position of the cursor

PRIOR: Fetches previous record from the current position of the cursor

LAST: Fetches last record from the cursor

ABSOLUTE N: Fetches nth record from the top of the cursor if n is positive fetches the nth record from bottom of the cursor if n is negative. Where n is an integer

RELATIVE N: Fetches nth next record from current position of the cursor if n is positive fetches nth previous record from the current position of the cursor if n is negative where n is an integer.

Types of cursors:

DYNAMIC: It reflects changes happened on the table while scrolling through the row.
STATIC: It works on snapshot of record set and disconnects from the server. This kind doesn't reflect changes happened on the table while scrolling through the row.
KEYSET: In this kind, new record is not reflected, but data modification can be seen
FORWARD_ONLY CURSOR: This is the most unused logical area because it supports only NEXT operation

Types of locks:

READ ONLY: This prevents any updates on the table.
SCROLL LOCK: This allows you to make changes to the table.
OPTIMISTIC: This checks if any change in the table record since the row fetched before updating. If there is no change, the cursor can update

Scrollable cursors:

We can use keyword SCROLL to make cursor Scrollable.
It can scroll to any row and can access the same row in the result set multiple times.
A non-scrollable cursor is also known as forward-only and each row can be fetched at most once

How cursor works:

When we **open** the cursor than the SELECT Statement will be sent to the SQL Server.
Server executes the SELECT statement and returns the data.
That data will store in a place called context area.
By using cursor we can ***fetch*** row by row from context area.
To fetch multiple rows we need to use the loop. In that loop we can manipulate the rows.
We need to ***close*** the cursor and finally we have to ***deallocate*** the cursor.

Eg: Write a cursor to print all employees along with their job and salary

```
DECLARE @ENO INT,@ENAME VARCHAR(MAX),@JOB VARCHAR(MAX),@SAL INT
DECLARE C1 CURSOR
FOR SELECT EMPNO,ENAME,JOB,SAL FROM EMP
OPEN C1
FETCH NEXT FROM C1 INTO @ENO,@ENAME,@JOB,@SAL
```

```
WHILE(@@FETCH_STATUS=0)
BEGIN
PRINT @ENAME +' EMPLOYEE NUMBER IS '+CAST(@ENO AS VARCHAR(MAX))+ ' AND WORKING AS
'+@JOB+     ' AND EARNS '+CAST(@SAL AS VARCHAR(MAX))
FETCH NEXT FROM C1 INTO @ENO,@ENAME,@JOB,@SAL
END
CLOSE C1
DEALLOCATE C1
```

Here @@FETCH_STATUS returns 0 if fetch is successful else returns 1.

Eg: write a cursor to display every third employee name in the emp table

```
DECLARE @NO INT=0,@ENAME VARCHAR(MAX)
DECLARE C1 CURSOR SCROLL FOR SELECT ENAME FROM EMP
OPEN C1
FETCH FIRST FROM C1 INTO @ENAME
WHILE(@@FETCH_STATUS=0)
BEGIN
PRINT @ENAME
SET @NO=@NO+3
FETCH ABSOLUTE @NO FROM C1 INTO @ENAME
END
CLOSE C1
DEALLOCATE C1
```

NOTE : The fetch typeS first and last AND PRIOR cannot be used with forward only cursors.
The fetch type Absolute cannot be used with dynamic cursors.

Read_only Cursor:

We know that processing of data through cursors is a not a choice, because SQL Server is designed to work best with sets of data and not one row at a time.

Still processing through cursors can be made faster by making little changes. Like, if we need to use cursor to process one row at a time and we don't need to update base table through this cursor, we MUST use read only cursor.

Write a cursor by using Read_only

```
DECLARE @EMPID INT
DECLARE CUREMP CURSOR READ_ONLY FOR SELECT EMPNO FROM EMP
OPEN CUREMP
FETCH NEXT FROM CUREMP INTO @EMPID
WHILE @@FETCH_STATUS = 0
BEGIN
    PRINT @EMPID
    FETCH NEXT FROM CUREMP INTO @EMPID
```

END
CLOSE CUREMP
DEALLOCATE CUREMP

CURSOR OPTIMIZATION TIPS.

1. Close cursor when it is not required.
2. We shouldn't forget to deallocate cursor after closing it.
3. We should fetch least number of records.
4. We should use FORWARD ONLY (Read-only) option when there is no need to update rows.

DISADVANTAGES OF CURSORS

1. Cursors use more resources because Each time we fetch a row from the cursor, it results in a network roundtrip
2. There are restrictions on the SELECT statements that can be used.
3. Because of the round trips, performance and speed is slow

DIFFERENT WAYS TO AVOID CURSORS:

1. **Using the SQL while loop**: Using a while loop we can insert the result set into the temporary table.
2. **User defined functions** : Cursors are sometimes used to perform some calculation on the resultant row set. This can also be achieved by creating a user defined function to suit the needs

TRIGGERS:

A trigger is a special type of stored procedure that executes automatically when an event occurs in the database server without being explicitly executed.

There are two types of triggers in SQL Server:

1. DML Triggers
2. DDL Triggers

DML Triggers: these triggers fire when any DML statements are executed on the database.
These triggers fire when any valid event is fired regardless of whether or not any table rows are affected or not.
These DML triggers can be used to enforce business rules and data integrity.
These triggers are similar to constraints in the way that they also enforce integrity.

Business rules are two types:

1. Declarative business rules
   These declarative business rules are implemented with constraints
2. Procedural business rules
   These procedural business rules are implemented with triggers.

If business rules are simple than we can use constraints
If business rules are complex than we can use triggers

DML Triggers are two types:

1. AFTER Triggers
2. INSTEAD OF Triggers

AFTER Triggers: These triggers fire after the DML Statements are executed on the database.

Eg: Create trigger on table which will restrict DML operations apart from business hours.

```
CREATE TRIGGER TRI1 ON EMP
AFTER INSERT,DELETE,UPDATE
AS
BEGIN
IF DATEPART(HH,GETDATE()) NOT BETWEEN 9 AND 17
BEGIN
ROLLBACK
RAISERROR('CAN NOT DO DML OPERATIONS',15,6)
END
END
```

Explanation: we have created a trigger on emp table to not to allow any DML operations before 9am and after 5pm everyday.

The datepart function checks the time whether the DML Statement execution time is between 9am and 5pm or not, if it is not between 9am and 5pm then that trigger will fire and user defined error will be thrown. If it is between 9am and 5pm then that trigger will not fire, that DML operation will execute on the table.

Eg: develop a trigger on emp table to not to allow insert statement on sunday

```
CREATE TRIGGER TRI2 ON EMP
AFTER INSERT
AS
BEGIN
IF DATENAME(DW,GETDATE())='SUNDAY'
BEGIN
ROLLBACK
RAISERROR('YOU CAN NOT DO DML OPERATIONS ON EMP TABLE ON SUNDAY',15,5)
END
END
```

Explanation: we have implemented trigger tri2 on emp table. This trigger does not allow insert statement on Sunday. If we try to insert a record on Sunday then it will throw an user defined error saying that 'YOU CAN NOT DO DML OPERATIONS ON EMP TABLE ON SUNDAY'. If is not Sunday than that INSERT statement will execute.

Write a trigger on dept table so that it will convert the name and loc into upper case when they are inserted into the table

```
CREATE TRIGGER T2 ON DEPT
AFTER INSERT
AS
BEGIN
DECLARE @DNO INT,@DNAME VARCHAR(MAX),@DLOC VARCHAR(MAX)
SELECT @DNO=DEPTNO,@DNAME=DNAME,@DLOC=LOC FROM INSERTED
UPDATE DEPT SET DNAME=UPPER(@DNAME),LOC=UPPER(@DLOC) WHERE DEPTNO=@DNO
SELECT * FROM INSERTED
END
```

NOW INSERT A RECORD:

```
INSERT INTO DEPT VALUES(10,'SALES','HYD')
```

Explanation: whenever a trigger fires because of an INSERT Statement, the values that are being inserted into the DEPT table will be captured in the trigger under a table called INSERTED. So that we can find out which values are being provided to the DEPT table by the user.
In the above case out trigger is AFTER Trigger so first the INSERT Statement will be executed and values will be inserted into the DEPT table and than that trigger executes so first those values are captured into the INSERTED table and than from the INSERTED Table we are updating the inserted values to the DEPT table.

CREATE A TRIGGER FOR INSERTING DATA ON THE FOLLOWING TIMINGS ONLY
MONDAY-FRIDAY TIMINGS ARE 10AM TO 6PM
SATURDAY TIMINGS ARE 10AM TO 2PM
SUNDAY, DATA CANT BE INSERTED

```
CREATE TRIGGER WDS ON TT1
AFTER INSERT,DELETE,UPDATE
AS
BEGIN

IF DATENAME(DW,GETDATE()) in( 'MONDAY','TUESDAY','WEDNESDAY','THURSDAY', 'FRIDAY')
BEGIN
IF DATENAME(HH,GETDATE()) NOT BETWEEN 10 AND 18
BEGIN
ROLLBACK
RAISERROR('MONDAY TO FRIDAY TIMINGS ARE 10 TO 18 ONLY',15,1)
END
END

IF DATENAME(DW,GETDATE())='SATURDAY'
BEGIN
IF DATENAME(HH,GETDATE()) NOT BETWEEN 10 AND 14
BEGIN
ROLLBACK
RAISERROR('SATURDAY TIMINGS ARE 10 TO 2PM ONLY',15,1)
END
END

IF DATENAME(DW,GETDATE())='SUNDAY'
BEGIN
ROLLBACK
RAISERROR('SUNDAY IS CLOSED COMPLETELY',15,1)
END
END
```

INSTAED OF Triggers: these triggers are used on Views.

DDL Triggers:

These DDL Triggers fire when we try to do any DDL Operations Like Create, Alter and Drop.

Create a trigger to restrict creating new table in the database

```
CREATE TRIGGER CT ON DATABASENAME
AFTER CREATE_TABLE
```

```
AS
BEGIN
ROLLBACK
RAISERROR('CAN NOT CREATE TABLE IN THIS DATABASE',15,1)
END
```

--Create a trigger to restrict dropping existing table in the database

```
CREATE TRIGGER DT ON DATABASE
AFTER DROP_TABLE
AS
BEGIN
ROLLBACK
RAISERROR('CAN NOT',15,1)
END
```

MAGIC TABLES(also called logical tables):

Magic table are virtual tables or invisible tables, we can use them through triggers only. These tables will be created, managed and dropped automatically by SQL Server. These magic tables hold recently inserted, updated and deleted (Like DML Operations) data on the table.

In SQL Server we have two types of Magic tables:

1. Inserted magic table
2. Deleted magic table

Inserted Magic Table:
This table will be created automatically when we insert data into table which is having DML Trigger.

Deleted Magic Table:
This table will be created automatically when we delete data from the table which is having DML Trigger.

When we update the data than both Deleted and Inserted tables will be created. Old data will be moved to Deleted Magic Table and new data will be moved to Inserted Magic Table.

Purpose of Magic tables:

Magic tables are used by triggers for the following purpose:

1. For finding the difference between the state of a table before and after the data modification and take actions based on that difference.
2. For testing data manipulation errors and taking suitable actions based on the errors.

To see list of triggers in current database:

```
select * from sys.triggers
```

ISOLATION LEVELS IN SQL SERVER

Following are the different types of isolations available in SQL Server.

- READ COMMITTED
- READ UNCOMMITTED
- REPEATABLE READ
- SERIALIZABLE
- SNAPSHOT

Before working on the isolations, let's create a table

```
CREATE TABLE STUDENT(ID INT,NAME VARCHAR(50),FEE INT)

  INSERT INTO STUDENT (ID,NAME,FEE)
  VALUES( 1,'NANI',10000)

  INSERT INTO STUDENT (ID,NAME,FEE)
  VALUES( 2,'SATYA',12000)

  INSERT INTO STUDENT (ID,NAME,FEE)
  VALUES( 3,'RENU',30000)
```

READ COMMITTED

In select query it will take only committed values of table. If any transaction is opened and uncompleted on table in others sessions then select query will wait till no transactions are pending on same table.

Read Committed is the default transaction isolation level.

E.g.: 1

Session 1

```
BEGIN TRAN
    UPDATE STUDENT SET FEE=15000 WHERE ID=1
    WAITFOR DELAY '00:00:15'
    COMMIT
```

Session 2

```
SET TRANSACTION ISOLATION LEVEL READ COMMITTED
    SELECT FEE FROM STUDENT WHERE ID=1
```

NOTE: Run both sessions side by side.

In second session, it returns the result only after execution of complete transaction in first session because of the lock on STUDENT table. We have used wait command to delay 15 seconds after updating the STUDENT table in transaction

E.g.: 2

Session1

```
BEGIN TRAN
SELECT * FROM STUDENT
WAITFOR DELAY '00:00:15'
COMMIT
```

Session2

```
SET TRANSACTION ISOLATION LEVEL READ COMMITTED
SELECT * FROM STUDENT
```

NOTE: Run both sessions side by side.

In session2, there won't be any delay in execution because in session1 Student table is used under transaction but it is not used update or delete command hence Student table is not locked.

READ UNCOMMITTED

If any table is modified (insert or update or delete) under a transaction and same transaction is not completed that is not committed or roll backed then uncommitted values will display (Dirty Read) in select query of "Read Uncommitted" isolation transaction sessions. There won't be any delay in select query execution because this transaction level does not wait for committed values on table.

E.g.:1

Session 1

```
BEGIN TRAN
UPDATE STUDENT SET FEE=15000 WHERE ID=1
WAITFOR DELAY '00:00:15'
ROLLBACK
```

Session 2

```
SET TRANSACTION ISOLATION LEVEL READ UNCOMMITTED
SELECT SALARY FROM STUDENT WHERE ID=1
```

NOTE: Run both sessions at a time one by one.

Select query in Session2 executes after update STUDENT table in transaction and before transaction rolled back. Hence 15000 is returned instead of 10000.

If you want to maintain Isolation level "Read Committed" but you want dirty read values for specific tables then use with(nolock) in select query for same tables as shown below.

```
SET TRANSACTION ISOLATION LEVEL READ COMMITTED
SELECT * FROM STUDENT WITH(NOLOCK)
```

REPEATABLE READ

Select query data of table that is used under transaction of isolation level "Repeatable Read" can not be modified from any other sessions till transaction is completed.

E.g.: 1

Session 1

```
SET TRANSACTION ISOLATION LEVEL REPEATABLE READ
BEGIN TRAN
SELECT * FROM STUDENT WHERE ID IN(1,2)
WAITFOR DELAY '00:00:15'
SELECT * FROM STUDENT WHERE ID IN (1,2)
ROLLBACK
```

Session 2

```
UPDATE STUDENT SET FEE=15000 WHERE ID=1
```

NOTE: Run both sessions side by side.

Output

Update command in session 2 will wait till session 1 transaction is completed because STUDENT table row with ID=1 has locked in session1 transaction.

E.g.: 2

Session 1

```
SET TRANSACTION ISOLATION LEVEL REPEATABLE READ
BEGIN TRAN
SELECT * FROM STUDENT
WAITFOR DELAY '00:00:15'
SELECT * FROM STUDENT
ROLLBACK
```

Session 2

```
INSERT INTO STUDENT(ID,NAME,FEE)
VALUES( 11,'CHANDU',11000)
```

Run both sessions side by side.

SERIALIZABLE

Serializable Isolation is similar to Repeatable Read Isolation but the difference is it prevents Phantom Read. This works based on range lock. If table has index then it locks records based on index range used in WHERE clause(like where ID between 1 and 3). If table doesn't have index then it locks complete table.

E.g.: 1

Assume table does not have index column.

Session 1

```
SET TRANSACTION ISOLATION LEVEL SERIALIZABLE
BEGIN TRAN
SELECT * FROM STUDENT
WAITFOR DELAY '00:00:15'
SELECT * FROM STUDENT
ROLLBACK
```

Session 2

```
INSERT INTO STUDENT(ID,NAME,FEE)
VALUES( 11,'CHANDU',11000)
```

Run both sessions side by side.

Output

Result in Session 1.

Serializable Isolation example table with no index
Complete STUDENT table will be locked during the transaction in Session 1. Unlike "Repeatable Read", insert query in Session 2 will wait till session 1 execution is completed. Hence Phantom read is prevented and both queries in session 1 will display same number of rows.

To compare same scenario with "Repeatable Read" read Repeatable Read Example 2.

E.g.: 2

Assume table has primary key on column "ID". In our example script, primary key is not added. Add primary key on column Emp.ID before executing below examples.

Session 1

```
SET TRANSACTION ISOLATION LEVEL SERIALIZABLE
BEGIN TRAN
SELECT * FROM STUDENT WHERE ID BETWEEN 1 AND 3
WAITFOR DELAY '00:00:15'
SELECT * FROM STUDENT WHERE ID BETWEEN 1 AND 3
ROLLBACK
```

Session 2

```
INSERT INTO STUDENT(ID,NAME,FEE)
VALUES( 11,'CHANDU',11000)
```

Run both sessions side by side.

Output

Since Session 1 is filtering IDs between 1 and 3, only those records whose IDs range between 1 and 3 will be locked and these records can not be modified and no new records with ID range between 1 to 3 will be inserted. In this example, new record with ID=11 will be inserted in Session 2 without any delay.

SNAPSHOT

Snapshot isolation is similar to Serializable isolation. The difference is Snapshot does not hold lock on table during the transaction so table can be modified in other sessions. Snapshot isolation maintains versioning in Tempdb for old data in case of any data modification occurs in other sessions then existing transaction displays the old data from Tempdb.

E.g.: 1

Session 1

```
SET TRANSACTION ISOLATION LEVEL SNAPSHOT
BEGIN TRAN
SELECT * FROM STUDENT
WAITFOR DELAY '00:00:15'
SELECT * FROM STUDENT
ROLLBACK
```

Session 2

```
INSERT INTO STUDENT(ID,NAME,FEE) VALUES( 11,'CHANDU',11000)
UPDATE STUDENT SET FEE=4444 WHERE ID=4
SELECT * FROM STUDENT
```

Run both sessions side by side.

Output

Result in Session 1.

Snapshot isolation Session 1 result
Result in Session 2.

Snapshot isolation Session 2 result
Session 2 queries will be executed in parallel as transaction in session 1 won't lock the table Emp.

TRANSACTIONS AND LOCKS IN SQL SERVER

TRANSACTIONS

Transaction is nothing but a logical unit of work, which contains set of operations.

Transaction and locks increase the complexity of database operations, but they guarantee valid data and query results in multi-user applications. This guarantee is expressed with ACID properties;

A --- Atomicity

C --- Consistency

I --- Isolation

D --- Durability

For a transaction to be Atomic, all the DML statements INSERT, UPDATE, DELETE must either all commit or rollback, which means a transaction can't be left in a half done state.

Consistency means a user should never see data changes in mid of transaction. Their view should return the data as it appears prior to beginning of the transaction or if the transaction is finished, then they should see the changed data.

Isolation is the heart of multi-user transactions. It means one transaction should not disrupt another transaction.

Durability implies that the changes made by the transaction are recorded permanently.

LOCKS

The Microsoft SQL Server Database Engine locks resources using different lock modes that determine how the resources can be accessed by concurrent transactions.

Locks are used by the Database Engine to synchronize access to a resource by multiple concurrent transactions

Terminology

Transaction – a unit of work performed within the database
Lock - the synchronization mechanism on a resource that protects changes amongst multiple concurrent transactions
Lock mode - defines the level of access that other transactions have while the resource is locked
Blocking - when a transaction requests a lock mode that conflicts with a currently held lock and has to wait for that lock to be released
Deadlock - when two transactions block each other trying to acquire locks on resources the other transaction holds in a conflicting mode

Lock Granularity

RID/KEY – a single row is locked

RID – row identifier for a single row in a heap

KEY – index key for a single row in a index

PAGE - a single page in the database is locked

HoBT – a heap or B-tree (index) partition is locked

TABLE – the entire table is locked

METADATA - the table schema definition is locked

Locks are acquired at multiple levels of granularity to fully protect the lowest-level resource

Locks are always acquired 'top-down', from the table level down to individual rows This forms the lock hierarchy

Lock Hierarchy

DELETE statement begins executing that affects one row in a table

A intent exclusive (IX) lock is acquired for the table

A intent exclusive (IX) lock is acquired for the page containing the row

A exclusive (X) lock is acquired for the row being modified

Shared (S) – acquired for read operations that do not modify data No transaction can modify data while the shared lock exists

Concurrent SELECT operations can read the data

Locks are released as soon as the read operation completes unless the isolation level is repeatable read or higher or hints are used

Update (U) – acquired for resources that will be updated Only one transaction can acquire an update (U) lock at a time

Prevents deadlocks caused by lock conversions from a shared (S) lock to an exclusive (X) lock

The update (U) lock is converted to an exclusive (X) lock to modify the data

Concurrent shared (S) locks are allowed

Exclusive (X) – acquired for data modifications Prevents access to a resource from concurrent transactions

Ensures that multiple changes cannot be made to the same resource at the same time

Reads can only occur using NOLOCK or read uncommitted, read committed snapshot, or snapshot isolation levels

Intent (I) – acquired to establish the lock hierarchy Acquired on higher-level resources to protect locks on lower-level resources E.g. a table-level intent exclusive (IX) lock is required before a page-level exclusive (X) or intent exclusive (IX) lock can be acquired

Prevents transactions from modifying/locking the higher-level resource with an incompatible lock for the lower-level lock being acquired

Intent shared (IS), intent exclusive (IX), shared with intent exclusive (SIX), intent update (IU), shared intent update (SIU) and update intent exclusive (UIX)

Schema (Sch) – acquired to protect or modify an object's definition Schema modification (Sch-M) locks are acquired during DDL operations to prevent access to the object while its definition is changed

Schema stability (Sch-S) locks are acquired when compiling and executing queries to prevent modification of the object definition by DDL operations

Key-range - acquired to protect a range of rows when using the serializable isolation level Prevents other transactions from inserting rows into the range to ensure that the data returned by a query would remain the same if it were run again within the transaction

Lock Escalation

The lock granularity is chosen during query compilation

Row, page, partition, or table

During query execution, the lock granularity may be escalated if the resources necessary for the lower-level locks are not available

Lock escalation can occur from row-to-table or page-to-table If partition-level escalation is enabled in SQL Server 2008 onwards, escalation can be row-to-partition or page-to-partition Locks NEVER escalate row-to-page-to-partition/table

Lock escalation can be desirable under some circumstances, but the higher-level locking reduces concurrency and can lead to blocking

Summary

Locking is an integral part of relational database management systems and protects data during concurrent activity in the database

Locks can be acquired at different levels of granularity and in different modes to control concurrent data access

Blocking occurs when a transaction attempts to acquire a lock with a mode that conflicts with a lock that is held by another transaction

Deadlocks occur when two transactions block each other trying to acquire locks on resources the other transaction holds in a conflicting mode

The following table shows the resource lock modes that the Database Engine uses.

Lock mode	Description
Shared (S)	Used for read operations those do not change or update data, such as a SELECT statement.
Update (U)	Used on resources that can be updated. Prevents a common form of deadlock that occurs when multiple sessions are reading, locking, and potentially updating resources later.
Exclusive (X)	Used for data-modification operations, such as INSERT, UPDATE, or DELETE. Ensures that multiple updates cannot be made to the same resource at the same time.
Intent	Used to establish a lock hierarchy. The types of intent locks are: intent shared (IS), intent exclusive (IX), and shared with intent exclusive (SIX).

Schema	Used when an operation dependent on the schema of a table is executing. The types of schema locks are: schema modification (Sch-M) and schema stability (Sch-S).
Bulk Update (BU)	Used when bulk copying data into a table and the **TABLOCK** hint is specified.
Key-range	Protects the range of rows read by a query when using the serializable transaction isolation level. Ensures that other transactions cannot insert rows that would qualify for the queries of the serializable transaction if the queries were run again.