LEARN SQL SERVER IN DEPTH

(A Step-by-Step Guide for Beginners)

109,WING A, SHOPPER'S ORBIT,
VISHRANTWADI, PUNE - 411015

7757012051/52

ENOSIS LEARNING

http://www.enosislearning.com

CONTENTS

CHAPTER 1: Introduction to SQL Server........ 4
Lesson 1: Understanding the SQL Server........ 4
Lesson 2: SQL Server Version History........ 6
Lesson 3: WHAT IS DATABASE........ 11
LESSON 4 : SQL SERVER ARCHITECTURE........ 12
CHAPTER 1: SUMMARY / CHEAT SHEET........ 14
CHAPTER 1: EXERCISE (ASSIGNMENTS)........ 14
CHAPTER 1: QUESTIONS........ 14
CHAPTER 2: TableS and SQL Statements........ 15
Lesson 1: Table creation........ 16
Lesson 2: SQL Statements/COMMANDS........ 18
CHAPTER 2: SUMMARY / CHEAT SHEET........ 23
CHAPTER 2: EXERCISE (ASSIGNMENTS)........ 24
CHAPTER 2: QUESTIONS........ 25
chapter 3: DATA INTEGRITY & CONSTRAINTS........ 26
Lesson 1: Constraints........ 26
CHAPTER 3: SUMMARY / CHEAT SHEET........ 31
CHAPTER 3: EXERCISE (ASSIGNMENTS)........ 31
CHAPTER 3: QUESTIONS........ 32
chapter 4: CLAUSES........ 33
Lesson 1: IN-BUILT FUNCTIONS........ 33
LESsON 2: CLAUSES DEFINATION WITH EXAMPLES........ 41
CHAPTER 4: SUMMARY / CHEAT SHEET........ 51
CHAPTER 4: EXERCISE 1(ASSIGNMENTS)........ 52
CHAPTER 4: questions........ 54
CHAPTER 5: RELATIONAL DATABASE........ 55
LESSON 1: RELATIONAL DATABASE........ 55
LESSON 2: TYPES OF RELATIONSHIP........ 60
Implementing one-to-one relation........ 61
Lesson 3: Normalization........ 63
Lesson 4: Types Of Joins........ 65
CHAPTER 5: SUMMARY / CHEAT SHEET........ 69
CHAPTER 5: EXERCISE (ASSIGNMENTS)........ 70
CHAPTER 5: EXERCISE (ASSIGNMENTS)........ 72

CHAPTER 5: EXERCISE (ASSIGNMENTS) 74
CHAPTER 5: QUESTIONS 75
CHAPTER 6: advance OPERATOR in SQL server 76
Lesson 1: union, EXcept & intersect 76
Lesson 2: pivot / unpivot 79
CHAPTER 6: SUMMARY / CHEAT SHEET 81
CHAPTER 6: QUESTIONS 81
CHAPTER 7 : Operators, Identifiers Variables & STATEMENTS 82
Lesson 1: Operators 82
Lesson 2: Identifiers 83
Lesson 3: Variables 83
Lesson 4: IF-ELSE STATEMENT 84
Lesson 5: CASE STATEMENT 85
Lesson 6: LOOP STATEMENT 86
CHAPTER 7: SUMMARY / CHEAT SHEET 87
CHAPTER 7: exercise(assignments) 88
CHAPTER 7: QUESTIONS 88
CHAPTER 8: STORED PROCEDURE & FUNCTIONS 89
Lesson 1: Implementing Stored Procedures 89
Lesson 2: Implementing User-Defined Functions 99
CHAPTER 8: summary / CHEATSHEET 101
CHAPTER 8: EXERCISE 1 (ASSIGNMENT) 101
CHAPTER 8: EXERCISE 2 (ASSIGNMENT) 102
chapter 8: QUESTIONS 105
CHAPTER 9: IMPLEMENTING Transactions, Error handling 105
LESSON 1: IMPLEMENTING ERROR HANDLING 105
Lesson 2: Managing Transactions 113
CHAPTER 9: summary / cheatsheet 116
CHAPTER 9:questions 116
CHAPTER 10: TABLE TYPES 117
Lesson 1: Temporary Tables 117
LESsON 2: Table Variable 120
Lesson 3: CTE 120
Lesson 4: subquery 121

CHAPTER 10: summary / CHEATSHEET 124
CHAPTER 10: EXERCISE (ASSIGNMENTS) 126
CHAPTER 10: questions 128
CHAPTER 11: Indexes 128
Lesson 1: Indexes 128
Lesson 2: Creating an index 131
CHAPTER 11 : SUMMARY / CHEATSHEET 133
CHAPTER 11: QUESTIONS 133
CHAPTER 12 : Views 134
Lesson 1: Creating and Implementing View 134
Chapter 12: SUMMARY / cheatsheet 137
Chapter 12: QUESTIONs 137
CHAPTER 13: Triggers 138
Lesson 1: Trigger 138
CHAPTER 13 : summary /cheatsheet 147
CHAPTER 13 : ASSIGNMENTS 147
CHAPTER 13 : questions 149

CHAPTER 1

CHAPTER 1: INTRODUCTION TO SQL SERVER

SQL Server is database server invented by Microsoft in the year 1989.It is a software to store and retrieve data. These software's are also called as Database Management System (DBMS).
The database management system is a software product which is used to stores and retrieves data requested by the user.
Lessons in this chapter:

- Lesson 1: Understanding the SQL server
- Lesson 2: Database
- Lesson 3: Version of Sql Server
- Lesson 4: Architecture

LESSON 1: UNDERSTANDING THE SQL SERVER

A database server (SQL SERVER) is a computer program that provides database services to other programs or computers, as defined by the client-server model. The database server will have a server which will store the data and it will be accessed by clients through different interfaces likes management studio, application etc. Therefore, a SQL Server is a database server that implements the Structured Query Language (SQL). SQL is a special-purpose programming language designed to handle data in a relational database management system.

Database Management system (DBMS)

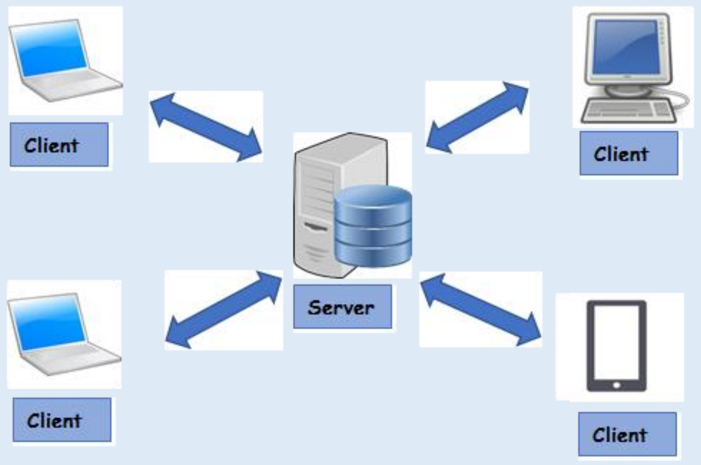
A DBMS makes it possible for end users to create, read, update and delete data in a database.

The DBMS essentially serves as an interface between the database and end users or application programs, ensuring that data is consistently organized and remains easily accessible.

DBMS is that it lets end users and application programmers access and use the same data while managing data integrity. Data is better protected and maintained when it can be shared using a DBMS instead of creating new iterations of the same data stored in new files for every new application. The DBMS provides a central store of data that can be accessed by multiple users in a controlled manner.

Why Database Management System.

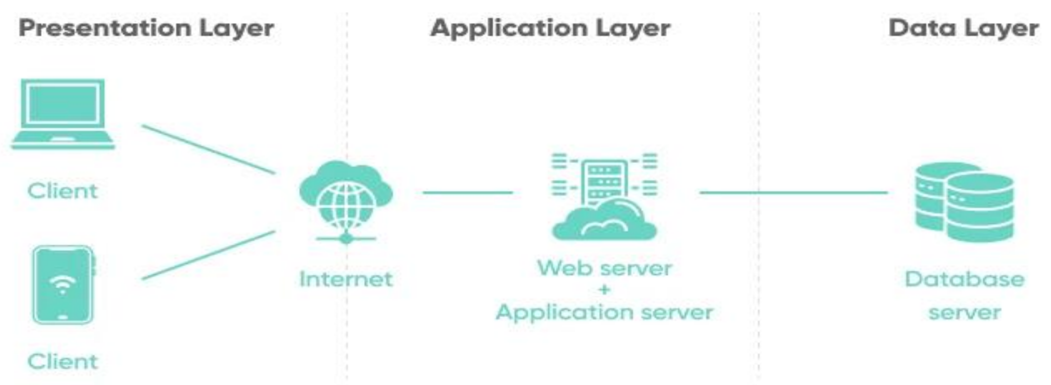
- It stores data in centralized server. [e.g. : single data server]
- **Client – Server architecture – single server – multiple clients.**
  Examples : Mail server, Banking, Reservation: centralized



- Data is being updated instantly.
- It supports disaster recovery. We can back up the data in remote locations.
- We can backup and restore the data instantly.
- Security: we can give access to different users depending upon roles.
- Transactions: data can be rolled back. Either all the tasks should be completed or it should be rolled back to the original position.

Examples of Database Management System :-> Oracle, My-SQL, SQL SERVER, MongoDB

Use of Database Management System in Application



Application cannot store the data permanently, so either the application will use the file system or database to store the data permanently. Since file-system have lots of drawback to store the data so application mostly use database for storing the data.

To use SQL SERVER, we need to install the server component and the client component. Whenever we install sql server on the machine, an instance of the server is installed on the machine.

SQL SERVER have different versions and editions which are explained in details in Lesson 2.

LESSON 2: SQL SERVER VERSION HISTORY

In this Lesson we will discuss different versions of SQL SERVER. It started in the year 1989 and released it next versions subsequently.

☑ 1st Generation

SQL Server Version	Features
SQL Server 1.0 (1989)	Developed by Microsoft, Sybase, and Ashton-Tate for OS/2.
SQL Server 4.2 (1992)	Developed for Windows NT 3.1.
SQL 6.0/6.5 (1995)	First version designed specifically for Windows NT.

☑ 2ND Generation

SQL Server Version	Features
SQL Server 7.0 (1999)	• Restructure of Relational Server. • Data Transformation Services. • Online Analytical Processing.
SQL2000	• Focus on Performance and Scalability • XML support • Data Mining • Reporting Services

3rd Generation (From SQL 2005 to SQL2022)

SQL Server Version	Features
SQL 2005	• High Availability (includes Database Mirroring). • Security Enhancements. • Integration Services. • Developer Productivity and CLR. • Native XML and Web services supports.
SQL 2008/SQL2008 R2	• Multi-Server Queries. • Compressions. • Database Encryption. • Spatial Data Types.
SQL 2012	• SQL Server Always On. • Power View. • Always on Failover Cluster Instance
SQL 2014	• In-memory OLTP Engine. • Always on Enhancements. • Buffer pool Extension. • Power view for multidimensional Model. • Integration to Azure • Memory Optimized Table
SQL 2016	• New security Features. • Dynamic data masking. • Stretch database. • Real time operational analysis. • Introduction of Polybase support • JSON Support
SQL 2017	• Can be installed in Linux and Docker • Resumable online index builder. • Automatic database tuning. • New graph database capabilities.
2019	• Memory-optimized TempDB Metadata • Accelerated Database Recovery (ADR) • Intelligent Query Processing (IDQ) Improvements • Replication in Linux • Always Encrypted with Secure Enclaves

2022	• Parameter Sensitive Plan Optimization • Query Store enhancements • Link to Azure SQL Managed Instance • Contained Availability Groups • Azure Synapse Link for SQL • Multi-Write Replication • Azure Active Directory authentication • Azure Purview integration • SQL Server Ledger • AWS S3 storage integration

Microsoft SQL SERVER also has multiple editions available within each version. There are 5 edition levels and each of the levels are geared towards different sized companies/applications. Below is a short description of each edition.

- **Enterprise** - contains all features with high end datacentre capabilities needed in large enterprises
- **Standard** - has basic data management which is geared towards departments or small companies
- **Web** - is a low cost option that web hosting companies can offer to their customers
- **Developer** - contains all of the functionality in enterprise edition, but is only licensed for development and test systems. This is an ideal version for developers who build applications.
- **Express** - free entry level database ideal for learning how to build a data driven application or for very small databases

The following table shows some examples of this.

Feature	Enterprise	Standard	Web	Express
Max Compute Capacity	OS max	Lesser of 4 sockets or 24 cores	Lesser of 4 sockets or 16 cores	Lesser of 1 socket or 4 cores
Max Buffer Pool Memory	OS max	128 GB	64 GB	1410 MB
Max Database Size	524 PB	524 PB	524 PB	10 GB

SQL SERVER EDITIONS FEATURES & LIMITATIONS

Free editions	
Express	Express edition is a **free** lightweight/lite edition of SQL Server with some limitations, that can be used in **production** environment. Main limitations: • Limited to lesser of **1 physical CPU** or **4 cores**. • No single database (.MDF file) can be over **10 GB**.
Developer	Developer edition is a fully-functional *Enterprise Edition* of SQL Server, licensed for use as a **development** and **test** database in a **non-production** environment.
Evaluation	Evaluation edition is a fully-functional **trial** *Enterprise Edition* of SQL Server but is limited to **180 days**, after which the tools will continue to run, but the server services will stop.
Paid editions	
Enterprise	Enterprise edition is the top-end edition with a full feature set.
Enterprise Core	Enterprise edition with the Core-based Licensing. The plain Enterprise one is limited to just 20 CPU cores.
Standard	Standard edition has less features than Enterprise, when there is no requirement of advanced features.
Azure SQL Database	Azure SQL Database is the cloud-based version of SQL Server, presented as a platform as a service (PaaS) offering on Microsoft Azure.
Specialized editions	
Web	Web edition is a low total-cost-of-ownership option for Web hosters.
Business Intelligence	Business Intelligence edition includes the Standard Edition capabilities and Business Intelligence tools.

Analytics Platform System (APS)	Formerly Parallel Data Warehouse (PDW). A massively parallel processing (MPP) SQL Server appliance optimized for large-scale data warehousing such as hundreds of terabytes.
Big Data Clusters (BDC)	Starting with SQL Server 2019, SQL Server Big Data Clusters allow you to deploy scalable clusters of SQL Server, Spark, and HDFS (Hadoop Distributed File System) containers running on Kubernetes.
LocalDB	SQL Server Express LocalDB is a lightweight version of Express that has all of its programmability features, runs in user mode and has a fast, zero-configuration installation and a short list of prerequisites.
Discontinued editions	
MSDE	Microsoft Data Engine or Microsoft Desktop Engine. Express edition predecessor.
Personal	Express edition predecessor.
Compact	Embedded database.
Workgroup	Cheaper version of the Standard edition with some limitations.
Datacenter	Datacenter edition was a full-featured edition of SQL Server and was designed for datacenters that need high levels of application support and scalability.

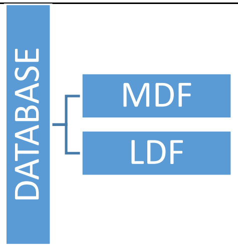
LESSON 3: WHAT IS DATABASE

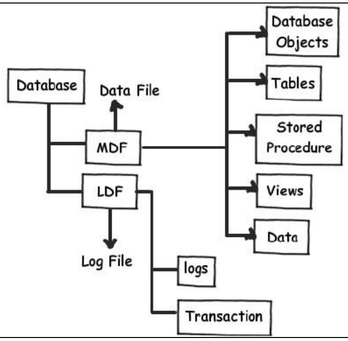
SQL SERVER IS A COLLECTION OF DATABASES. SQL SERVER STORES THE DATA IN A DATABASE.

A database is a collection of information that is organized so that it can be easily accessed, managed and updated. In database the data is stored in the form of tables. Table is nothing but the collection of rows and columns.

The structure of **table** is defined in the form of **columns** and data is defined in the form of **rows**.

Whenever we create a database it creates two files to store the data.





☑ Quick Facts

We can create 32767 databases in single instance

You can install 50 instances of SQL SERVER on a single machine.

1. .MDF (MASTER DATA FILE): [This is where the actual data is stored]
2. .LDF (LOG DATA FILE): [This stores the user queries, logs]

Note: While creating database, we have to specify the name of the file, location and size of the file.

- ***The MDF file is the heart of the database. All the data is stored in the MDF file.***
- ***The MDF file is divided into no of pages, each page size is of 8 KB.***
- ***1MB: 128 PAGES.* The MDF file is divided into pages.**

Databases are different types

User Defined Database	System Defined Database
The user defined databases are created by users to store data in tables etc	The system defined database are already created by sql server to stored system related information.

System Databases (System Defined Database)

SQL Server mainly contains four System Databases

MASTER	MODEL	MSDB	TEMPDB

Master Database : Master Database contains information about SQL server configuration. Without **Master** database, server can't be started. This will store the metadata information about all other objects (Databases, Stored Procedure, Tables, Views, etc.) which is Created in the SQL Server. It will contain login information of users. It is the heart of SQL SERVER.

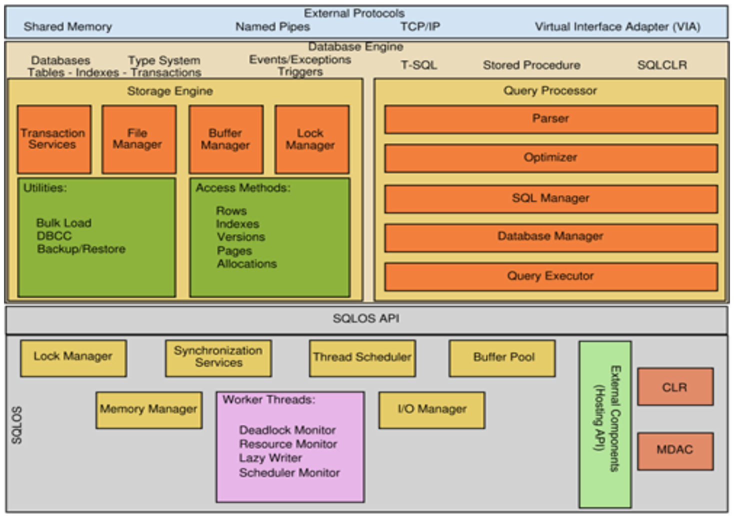
Model Database : The model database sets a template for every database that was newly created. It serves as a template for the SQL server in order to create a new database. When we create a new database, the data present in model database are moved to new database to create its default objects which include tables, stored procedures, etc. Primarily, the requirement of model database is not specific to creation of new database only. Whenever the SQL server starts, the **Tempdb** is created by using model database in the form of a template. By default it does not contain any data.

Msdb : The **msdb** database is used mainly by the SQL server Management Studio, SQL Server Agent to store system activities like SQL server jobs, mail, service broker, maintenance plans, user and system database backup history, Replication information, log shipping.

TempDB : It can be accessed by all the users in the SQL Server Instance. The tempdb is a temporary location for storing temporary tables (Global and Local) and temporary stored procedure that hold intermediate results during the sorting or query processing. This Database will be created by SQL Server instance when the SQL Server service starts. This database is created using model database. We cannot take a backup of temp Database.

LESSON 4 : SQL SERVER ARCHITECTURE

Like all database systems, the SQL Server performs extremely complex tasks like memory management, optimum CPU utilization, query optimization, execution planning etc. at a lightning fast speed.



The diagram gives you an overview of what the SQL Server architecture looks like.

There are 3 important items:

1. **Database Engine: Performs the storage and query processing tasks.**
2. **SQL Operating System: Works on I/O and other higher level management tasks.**
3. **Interfaces: The External Protocols and SQL Operating System API connect the dots.**

The Database engine is the heart of Microsoft SQL Server

It contains 2 major components:

- Query processor: optimizes and executes the query.
- Storage engine: Manages buffer and access methods

Going with the motorcycle analogy, the SQL Operating System API and External Protocols are the wires and dials. They connect the speedometer to the front wheel, show you the speed. This speed can then be reduced or increased using the accelerator. They simply provide ways to connect internal and external components together and manipulate the operations.

The major components of SQL Server are:

1. Relational Engine
2. Storage Engine
3. SQL OS

1) Relational Engine: Also called as the query processor, Relational Engine includes the components of SQL Server that determine what your query exactly needs to do and the best way to do it. It manages the execution of queries as it requests data from the storage engine and processes the results returned.

Different Tasks of Relational Engine:

1. Query Processing
2. Memory Management
3. Thread and Task Management
4. Buffer Management
5. Distributed Query Processing

2) Storage Engine: Storage Engine is responsible for storage and retrieval of the data on tothe storage system (Disk, SAN etc.). There are 2 types of files that are created at the disk level
Data file and Log file. Data file physically stores the data in data pages. Log files that are also known as write ahead logs, are used for storing transactions performed on the database.

CHAPTER 1: SUMMARY / CHEAT SHEET

SQL SERVER is a software for storing and retrieving data.
Once the sql server software is installed, it creates an instance which is called as server which can be used to stored and retrieve data.
We can install multiple instances (server) in a single physical machine.
In the instance we create multiple databases.
We can install 50 instances of SQL SERVER on a single machine.
A database is a collection of Sql Objects like Tables, Views , Stored Procedures etc
A database stores the data in mdf file internally.
A database stores the logs in ldf file internally.
We can create 32767 databases in single instance.

CHAPTER 1: EXERCISE (ASSIGNMENTS)

1	Install SQL SERVER COMPONENTS AND MANAGEMENT STUDIO are create a default instance or named instance.
2	Create a Database with an Initial Size of MDF file as 100 MB.
3	Create a Database with a Max Size of MDF file as 200 MB.
4	Create a Database by defining two file groups and separate Data file for each file.

CHAPTER 1: QUESTIONS

1	What is DBMS ?
2	What is Sql Server ?
3	What is different between DBMS, RDBMS & ORDBMS ?
4	What is an Instance?
5	What is the difference between default instance and named instance ?
6	What is MDF in database ?
7	What is LDF in database ?
8	What is Client Server Architecture ?

CHAPTER 2

CHAPTER 2: TABLES AND SQL STATEMENTS

Tables are used to store the data in Microsoft SQL server. Tables are created inside DATABASE of SQL SERVER to store data. In tables, data is logically organized in row and column format. Each row represents a new records and each column represents fields in that records. In this chapter, you will learn how to create and alter tables.

Types of Tables:

1. **User Defined Table**
   These tables are created by the user for storing data.
2. **System Defined Tables**
   These tables are already created by sql server which stores system related information.
3. **Memory optimized tables.**
   The main benefit of memory-optimized tables are that rows in the table are read from and written to memory which results in non-blocking transactions at super-fast speed. The second copy of the data is stored on the disk and during database recovery, data is read from the disk-based table. Memory-optimized tables are for specific types of workloads such as high volume OLTP applications.
4. **Temporal table.**
   A system-versioned temporal table is a type of user table designed to keep a full history of data changes and allow easy point in time analysis. This type of temporal table is referred to as a system-versioned temporal table because the period of validity for each row is managed by the system (i.e. database engine). Every temporal table has two explicitly defined columns, each with a **datetime2** data type. These columns are referred to as period columns. These period columns are used exclusively by the system to record period of validity for each row whenever a row is modified.
5. **External Table.**
   External Tables in SQL Server 2016 are used to set up the new Polybase feature with SQL Server. With this new feature (Polybase), you can connect to Azure blog storage or Hadoop to query non-relational or relational data from SSMS and integrate it with SQL Server relational tables.
6. **File Table.**
   The FileTable feature brings support for the Windows file namespace and compatibility with Windows applications to the file data stored in SQL Server. FileTable lets and application integrate its storage and data management components, and provides integrated SQL Server services - including full-text search and semantic search - over unstructured data and metadata.

Apart from this Sql Server also have some temporary tables like Local Temp Table, Global Temp Table, Table Variable etc. which are created temporarily and dropped automatically.

Lessons in this chapter:

- Lesson 1: Table creation.
- Lesson 2: SQL Statements/ COMMANDS

LESSON 1: TABLE CREATION

A table represents the structure of the data. In SQL server, first we have to define the structure. (First we have to create the table and then we can insert values into the table.) So in this lesson you learn about how to create table.

In SQL server tables are the database object that contains all the data in a database. For example ,a table that contain students data for a class it contain a row for each students and columns representing student information such as roll number, name, gender, math's marks, and science marks.

Creating a Table: You can create a table in SQL server in two ways.

1. By using the CREATE TABLE statement.
2. By using SELECT INTO statement.

1. By using the CREATE TABLE statement.

Using create table command you can create new table. Basic syntax for creating a table is:

```
CREATE TABLE TABLE_NAME
(
Column_name1 datatype,
Column_name2 datatype,
Column_name3 datatype
)
```

☑ Quick Facts

A table can have up to 1,024 columns.

What is a Datatype?
A Datatype is used to define the type of data to be used by the column.
SQL Server has a predefined set of datatypes.

SQL Server	Data Precision
BIT	Integer: 0 or 1
TINYINT	Positive Integer 0 -> 255
SMALLINT	Signed Integer -32,768 -> 32,767
INT	Signed Integer -2^31 -> 2^31-1
BIGINT	Signed Integer -2^63 -> 2^63-1
REAL	Floating precision -1.79E + 308 -> 1.79E + 308
FLOAT	Floating precision -3.40E + 38 -> 3.40E + 38
MONEY	4 decimal places, -2^63/10000 -> 2^63-1/10000
SMALLMONEY	4 decimal places, -214,748.3648 -> 214,748.3647
DECIMAL	Fixed precision -10^38 + 1 -> 10^38 – 1
NUMERIC	Fixed precision -10^38 + 1 -> 10^38 – 1
DATETIME	Date+Time 1753-01-01 -> 9999-12-31, accuracy of 3.33 ms
SMALLDATETIME	Date+Time 1900-01-01 -> 2079-06-06, accuracy of one minute

CHARn	Fixed-length non-Unicode string to 8,000 characters
NCHARn	Fixed-length Unicode string to 4,000 characters
VARCHARn	Variable-length non-Unicode string to 8,000 characters
NVARCHARn	Variable-length Unicode string to 4,000 characters
TEXT	Variable-length non-Unicode string to 2,147,483,647 characters
NTEXT	Variable-length Unicode string to 1,073,741,823 characters
BINARY	Fixed-length binary data up to 8,000 characters
VARBINARY	Variable-length binary data up to 8,000 characters
ROWVERSION	hexadecimal binary data up to 8 bytes
HIERARCHYID	Variable-length fanouts (0-7) the typical storage is about 6 x Log A * n bits
UNIQUEIDENTIFIER	Variable length binary data up to 16 byte
GEOMETRY	Variable-length binary data up to 8 Bytes

EXAMPLE OF A TABLE STRUCTURE: CREATE A TABLE EMPLOYEE

COLUMNNAME	DATATYPE	SQL SERVER TABLE CREATION
EMPID	INT	CREATE TABLE EMPLOYEE (EMPID INT, EMPNAME VARCHAR(100), SALARY FLOAT, DEPARTMENT VARCHAR(100), DESIGNATION VARCHAR(100), JOININGDATE DATETIME, STATUS BIT)
EMPNAME	VARCHAR(100)	
SALARY	FLOAT	
DEPARTMENT	VARCHAR(100)	
DESIGNATION	VARCHAR(100)	
JOININGDATE	DATETIME	
STATUS	BIT	

Other data types: There are few data types as well that can be used as per the requirement:

- Cursor: It is useful for variables or stored procedure OUTPUT parameter referencing to a cursor
- Rowversion: It returns automatically generated, unique binary numbers within a database
- Hierarchyid: **it is a system data type with variable length. We use it to represent a** position in a hierarchy
- Uniqueidentifier: It provides 16 bytes GUID
- XML: It is a special data type for storing the XML data in SQL Server tables
- Spatial Geometry type: We can use this for representing data in a flat (Euclidean) coordinate system
- Spatial Geography Types: We can use Spatial Geography type for storing ellipsoidal (round-earth) data, such as GPS latitude and longitude coordinates. It represents data in a round-earth coordinate system
- Table: It is a special data type useful for storing result set temporarily in a table-valued function. We can use data from this for processing later. It can be used in functions, stored procedures, and batches

2. By using SELECT INTO statement.

The SELECT INTO statement creates a new table and populates it with the result set of the SELECT statement. SELECT INTO can be used to combine data from several tables or views into one table. It can also be used to create a new table base upon existing table.

SYNTAX	**SELECT INTO** New_table_name **FROM** Old_table_name.
EXAMPLE	**SELECT INTO** Employee_new **FROM** Employee This will create a new table and copy the data from the existing table Select * from Employee_New

LESSON 2: SQL STATEMENTS/COMMANDS

SQL server provides two types of statements.

1. DML (Data Manipulation Language)
2. DDL (Data Definition Language)

DML is used for selecting/storing /updating/deleting the data. **DDL** command is used for creating structure. In this lesson you will learn how to do the operations like- create, select, insert, delete, update, truncate, alter and drop.

DDL	DML
CREATE	INSERT
ALTER	UPDATE
DROP	DELETE
	TRUNCATE
	SELECT

DML commands are insert, update, delete, select and truncate. (SYNTAX & EXAMPLES)

COMMAND	DESCRIPTION
INSERT	To add a new record into the table • **SYNTAX: INSERT INTO table_name values(value1, value2, value3)** • **To insert values into all the columns** EXAMPLE : INSERT INTO EMPLOYEE VALUES(1,'SAMIR',20000,'IT','MANAGER','10/10/2022',1) **INSERT INTO table_name(colname1, colname2, colname3) values(value1, value2, value3)** To insert values into selected columns **EXAMPLE :** INSERT INTO EMPLOYEE(EMPID,EMPNAME) VALUES (2,'RAJESH') INSERT INTO EMPLOYEE(EMPNAME,SALARY,STATUS) VALUES ('KAPIL',60000,1)

UPDATE	To update the values in the table **SYNTAX** **UPDATE table_name set col_name1 = newvalue1,col_name2=newvalue2 where condition** **EXAMPLE :** Update Employee set Salary= 50000 where EMPID=1 Update Employee set DEPARTMENT='IT',DESIGNATION='DEVELOPER' where EMPID=2 (IF WE SKIP WHERE CONDITION THEN ALL THE RECORDS WILL BE UPDATED) Update Employee set Salary= 0
DELETE	To remove the records from the table **SYNTAX : Delete from table_name where condition** It will delete those records which matches the condition. **Delete from table_name: It will delete all the records.** **EXAMPLE :** **DELETE FROM EMPLOYEE WHERE EMPID =1** **NOTE :[IF WE SKIP WHERE CONDITION ,IT will delete all the records in the table]** **DELETE FROM EMPLOYEE**
SELECT	To view the records of the table **SYNTAX** **Select * from table_name where condition** **Select colname1, colname2, colname3 from table_name where condition** **EXAMPLE :** **SELECT * FROM EMPLOYEE** SELECT EMPID,EMPNAME,SALARY FROM EMPLOYEE **Full Syntax** **SELECT** [DISTINCT] [TOP *n* [PERCENT]] *select list* **FROM** *table_source* [**WHERE** *search_condition*] [**GROUP BY** *group_by_expression*] [**HAVING** *search_condition*] [**ORDER BY** *order_expression* [**ASC** | **DESC**]]
TRUNCATE	Removes all rows from a table **SYNTAX: TRUNCATE TABLE** table_name **Example:** To delete all the rows from employee table, the query would be like: **TRUNCATE TABLE** employee;

DDL commands are create, drop and alter.(SYNTAX & EXAMPLES)

COMMAND	DESCRIPTION
CREATE	Is used to create. Like table, database, views, stored procedure, triggers. **SYNTAX:** CREATE TABLE TABLE_NAME (Column_name1 datatype, Column_name2 datatype, Column_name3 datatype) **Example:** CREATE TABLE EMPLOYEE (EMPID INT, EMPNAME VARCHAR(100), SALARY FLOAT, DEPARTMENT VARCHAR(100), DESIGNATION VARCHAR(100), JOININGDATE DATETIME, STATUS BIT
ALTER	Is used to make changes to the structures in the existing table. • To add a new column. • To drop existing column. • To alter a column.

Alter Syntax

COMMANDS	SYNTAX & Example	Description
ADD COLUMN	**SYNTAX** **ALTER TABLE TABLE_NAME ADD COLUMN_NAME DATATYPE** **EXAMPLE** ALTER TABLE EMPLOYEE ADD GENDER VARCHAR(100)	Adding a new column to the existing table
DROP COLUMN	**SYNTAX** **ALTER TABLE TABLE_NAME DROP COLUMN COLUMN_NAME** **EXAMPLE** ALTER TABLE EMPLOYEE DROP COLUMN GENDER	Drop a column from the table
ALTER COLUMN	**SYNTAX** **ALTER TABLE TABLE_NAME ALTER COLUMN COLUMN_NAME DATATYPE** **EXAMPLE** ALTER TABLE EMPLOYEE ALTER COLUMN EMPID VARCHAR(50)	MAKING CHANGES INTO THE EXISTING COLUMNS

DROP	Removes all rows along with structure of table. **SYNTAX:** **DROP TABLE table_name** **Example:** DROP TABLE employee;

☑ **Quick Facts / NOTE : Difference between Delete, Truncate and Drop.**

DELETE	TRUNCATE	DROP
IN **DELETE** WE CAN DELETE A SINGLE RECORD OR MULTIPLE RECORDS BASED UPON CONDITIONS. **Eg : Delete from table_name**	**TRUNCATE** IS USED TO DELETE THE COMPLETE DATA, WE CANNOT SPECIFY ANY CONDITIONS FOR IT. **Eg : Truncate table table_name**	**DROP** WILL REMOVE THE STRUCTURES ALONG WITH THE DATA. **Eg : Drop table table_name**

DELETE Statement: This command deletes only the rows from the table based on the condition given in the where clause or deletes all the rows from the table if no condition is specified. But it does not free the space containing the table.

TRUNCATE statement: This command is used to delete all the rows from the table and free the space containing the table.

☑ IMP : CREATE A TABLE STRUCTURE SIMILAR TO THIS

EMPID	EMPNAME	GENDER	DATEOFBIRTH	DATEOFJOINING	SALARY	TAX	DEPARTMENT	DESIGNATION	CITY

CREATE THE TABLE STRUCTURE AND RUN THE INSERT SCRIPTS FOR SAMPLE DATA, WHICH WILL BE USED IN NEXT CHAPTERS

```
CREATE TABLE EMPLOYEES
(
EMPID INT, EMPNAME VARCHAR(100), GENDER VARCHAR(10),
DATEOFBIRTH DATETIME, DATEOFJOINING DATETIME,
SALARY FLOAT, TAX FLOAT,
DEPARTMENT VARCHAR(100), DESIGNATION VARCHAR(100), CITY VARCHAR(100)
)
SELECT * FROM EMPLOYEES
INSERT INTO EMPLOYEES VALUES(1,'SANDEEP','MALE','01/15/1999','01/18/2021',20000,1000,
'IT','DEVELOPER','MUMBAI')
INSERT INTO EMPLOYEES VALUES(2,'KAILASH','MALE','05/20/1998','01/18/1999',30000,1000,
'IT','DEVELOPER','PUNE')
INSERT INTO EMPLOYEES VALUES(3,'SONALI','FEMALE','08/21/1999','01/20/2021',15000,1000,
'ADMIN','OFFICER','MUMBAI')
INSERT INTO EMPLOYEES VALUES(4,'ROSHINI','FEMALE','05/20/1997','01/21/2000',16000,1000,
'ADMIN','OFFICER','NASHIK')
INSERT INTO EMPLOYEES VALUES(11,'UMESH','MALE','02/25/1996','04/11/2021',25000,1000,
'HR','EXECUTIVE','MUMBAI')
INSERT INTO EMPLOYEES VALUES(12,'JAYESH','MALE','05/11/1995','01/18/2018',35000,1000,
'HR','MANAGER','PUNE')
INSERT INTO EMPLOYEES VALUES(13,'ISHITA','FEMALE','07/07/1999','01/20/2011',18000,1000,
'FINANCE','ACCOUNTANT','MUMBAI')
INSERT INTO EMPLOYEES VALUES(14,'KAVITA','FEMALE','09/08/1997','01/21/2012',19000,1000,
'FINANCE','MANAGER','PUNE')
INSERT INTO EMPLOYEES VALUES(15,'RADHA','FEMALE','05/25/1998','01/19/1999',16500,1000,
'ADMIN','OFFICER','NASHIK')
```

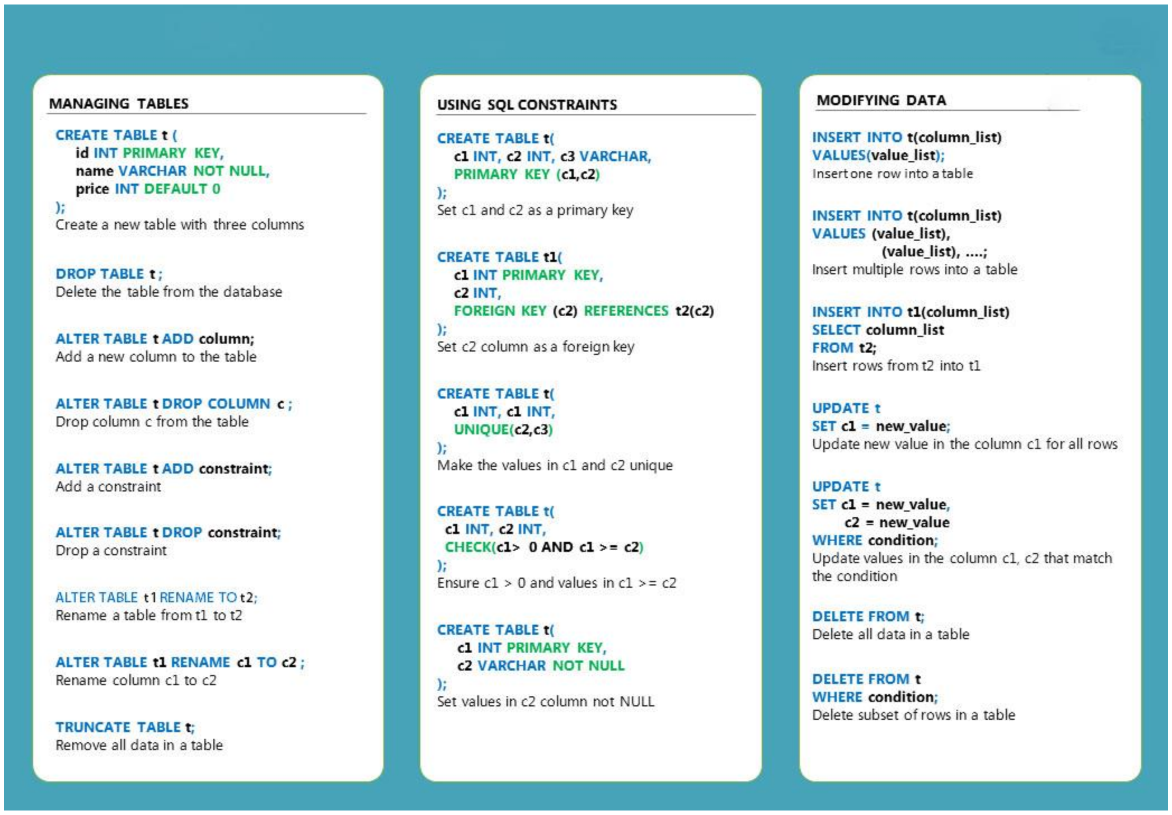
SYSTEM DEFINED TABLES

There are some predefined system tables already created in sql server which stores informations about the databases, files, tables etc.

Some of the important tables are as follow

TABLE NAME	DESCRIPTION
SELECT * FROM sys.databases	Contains information about all sql server databases.
SELECT * FROM sys.all_objects	Contains information about all sql objects like tables, constraints etc
SELECT * FROM sys.database_files	Contains information about mdf / ldf files of the database
SELECT * FROM sys.tables	Contains information about tables of sql server. Contains a row for each user table in SQL Server.
SELECT * FROM sys.all_columns	Contains information about each columns of sql server. Contains a row for each column of the table in SQL Server.
SELECT * FROM sys.check_constraints	Contains information about check constraints added in the tables of sql server. Contains a row for each check constraints in SQL Server.
SELECT * FROM sys.procedures	Contains information about all procedures in the databases

CHAPTER 2: SUMMARY / CHEAT SHEET



MANAGING TABLES

```
CREATE TABLE t (
    id INT PRIMARY KEY,
    name VARCHAR NOT NULL,
    price INT DEFAULT 0
);
```
Create a new table with three columns

```
DROP TABLE t ;
```
Delete the table from the database

```
ALTER TABLE t ADD column;
```
Add a new column to the table

```
ALTER TABLE t DROP COLUMN c ;
```
Drop column c from the table

```
ALTER TABLE t ADD constraint;
```
Add a constraint

```
ALTER TABLE t DROP constraint;
```
Drop a constraint

```
ALTER TABLE t1 RENAME TO t2;
```
Rename a table from t1 to t2

```
ALTER TABLE t1 RENAME c1 TO c2 ;
```
Rename column c1 to c2

```
TRUNCATE TABLE t;
```
Remove all data in a table

USING SQL CONSTRAINTS

```
CREATE TABLE t(
   c1 INT, c2 INT, c3 VARCHAR,
   PRIMARY KEY (c1,c2)
);
```
Set c1 and c2 as a primary key

```
CREATE TABLE t1(
   c1 INT PRIMARY KEY,
   c2 INT,
   FOREIGN KEY (c2) REFERENCES t2(c2)
);
```
Set c2 column as a foreign key

```
CREATE TABLE t(
   c1 INT, c1 INT,
   UNIQUE(c2,c3)
);
```
Make the values in c1 and c2 unique

```
CREATE TABLE t(
  c1 INT, c2 INT,
  CHECK(c1> 0 AND c1 >= c2)
);
```
Ensure c1 > 0 and values in c1 >= c2

```
CREATE TABLE t(
   c1 INT PRIMARY KEY,
   c2 VARCHAR NOT NULL
);
```
Set values in c2 column not NULL

MODIFYING DATA

```
INSERT INTO t(column_list)
VALUES(value_list);
```
Insert one row into a table

```
INSERT INTO t(column_list)
VALUES (value_list),
             (value_list), ....;
```
Insert multiple rows into a table

```
INSERT INTO t1(column_list)
SELECT column_list
FROM t2;
```
Insert rows from t2 into t1

```
UPDATE t
SET c1 = new_value;
```
Update new value in the column c1 for all rows

```
UPDATE t
SET c1 = new_value,
       c2 = new_value
WHERE condition;
```
Update values in the column c1, c2 that match the condition

```
DELETE FROM t;
```
Delete all data in a table

```
DELETE FROM t
WHERE condition;
```
Delete subset of rows in a table

CHAPTER 2: EXERCISE (ASSIGNMENTS)

CREATE A TABLE FOR COMPANY

COLUMN_NAME	DATATYPE
COMPANYID	INT
COMPANYNAME	VARCHAR(100)
WEBSITE	VARCHAR(100)
EMAIL	VARCHAR(100)

ADD TWO COMPANY RECORDS TO THE TABLE using INSERT STATEMENTS

CREATE A TABLE FOR DEPT

COLUMN_NAME	DATATYPE
DEPTID	INT
DEPTNAME	VARCHAR(100)
DEPTLOCATION	VARCHAR(100)

ADD FIVE DEPT RECORDS TO THE TABLE using INSERT STATEMENTS

CREATE A TABLE FOR DESIGNATION

COLUMN_NAME	DATATYPE
DESGID	INT
DESGNAME	VARCHAR(100)
CREATEDDATE	DATETIME

ADD FIVE DESIGNATION RECORDS TO THE TABLE using INSERT STATEMENTS

CREATE A TABLE FOR EMPLOYEE

COLUMN_NAME	DATATYPE
EMPID	INT
EMPNAME	VARCHAR(100)
BASICSALARY	FLOAT
INCENTIVE	FLOAT
BONUS	FLOAT
DOB	DATETIME
DOJ	DATETIME
GENDER	VARCHAR(100)
EMAIL	VARCHAR(100)
MOBILE	VARCHAR(10)
CREATEDDATE	DATETIME

ADD 10 EMPLOYEES RECORDS TO THE TABLE USING INSERT STATEMENTS

CREATE A STUDENT TABLE SIMILAR TO THIS

ROLLNO	NAME	ENGMARKS	SCNMARKS	MATHMARKS

ADD AROUND 10 STUDENTS WITH ALL THE DETAILS
THE ALTER THE TABLE AND ADD SOME NEW COLUMNS AS SHOWN BELOW TO THE STUDENTS TABLE.

TOTALMARKS	PERCENTAGE	GRADE

UPDATE THE STUDENTS TO CALULATE THE TOTALMARKS, PERCENTAGES & GRADE.

CHAPTER 2: QUESTIONS

What is the difference between DDL statements and DML statements
What is the difference between DELETE and TRUNCATE? Which one is faster?
What is the difference between varchar and nvarchar datatype ?
How to copy one table into another table?
Which command is used to change the datatype of an existing table?
How many columns can be there in a table?

CHAPTER 3

CHAPTER 3: DATA INTEGRITY & CONSTRAINTS

Data integrity refers to the accuracy, consistency, and reliability of data that is stored in the database. Both database designers and database developers are responsible for implementing data integrity within one or a set of related databases.

There are four types of data integrity:

1. Row integrity
2. Column integrity
3. Referential integrity
4. User-defined integrity

Data Integrity is implemented by creating constraints on the columns of the table.

This lesson covers the types of constraints that you can create on tables that help you enforce data integrity.

Lessons in this chapter:

1. Lesson 1: Constraints

LESSON 1: CONSTRAINTS.

The best way to enforce data integrity in SQL Server tables is by creating or declaring constraints on tables. You apply these constraints to a table and its columns by using the CREATE TABLE or ALTER TABLE statements.

Constraint are the rules enforced on the data columns of a table. These are used to limit the type of data that can go into a table.

SQL server provides the following constraints.

- Primary Key, Composite Primary Key
- Unique
- Check
- Default
- Foreign key
- Not Null

Primary Key Constraint

This will make the column as unique and cannot accept null value. When you specify a primary key constraint for a table, the Database Engine enforces data uniqueness by automatically creating a unique index for the primary key columns.

This index also permits fast access to data when the primary key is used in queries. If a primary key constraint is defined on more than one column, values may be duplicated within one column, but each combination of values from all the columns in the primary key constraint definition must be unique.

To create a primary key on a column, there are a number of requirements:

- The column or columns cannot allow NULL. If the column or columns allow NULL, the constraint command will fail.
- Any data already in the table must have unique values in the primary key column or columns. If there are any duplicates, the ALTER TABLE statement will fail.
- There can be only one primary key constraint at a time in a table. If you try to create two primary key constraints on the same table, the command will fail.

☑ Quick Facts

Only one primary key is allowed in a table.

SYNTAX to create a primary key constraint is:

Column_name1 datatype constraint constraint_name PRIMARY KEY

Example: Empid int constraint pk_empid PRIMARY KEY

ADDING / DROPING CONSTRAINTS USING ALTER COMMAND

Another way of declaring a column as a primary key is to use the ALTER TABLE statement, which you could write as follows. Alter statement can be used for adding or dropping any type of constraints.

1. **ALTER TABLE TABLE_NAME ADD CONSTRAINT**
   Add a constraint to the table, if the table is already defined.
   SYNTAX:
   ALTER TABLE table_name ADD CONSTRAINT constraint_name CONSTRAINT_TYPE (column name)
   EXAMPLE:
   Alter table EMPLOYEE add constraint pk_empid PRIMARY KEY (EMPID)

2. **ALTER TABLE TABLE_NAME DROP CONSTRAINT**
   Drop a constraint from the table.
   SYNTAX:
   ALTER TABLE TABLE_NAME DROP CONSTRAINT constraint_name
   EXAMPLE:
   Alter table EMPLOYEE drop constraint pk_empid

COMPOSITE PRIMARY KEY

Composite key, or composite primary key, refers to cases where more than one column is used to specify the primary key of a table. In such cases, all foreign keys will also need to include all the columns in the composite key. Note that the columns that make up a composite key can be of different data types.

SYNTAX	EXAMPLE
CREATE TABLE table_name (column_name1 data_type(size) NOT NULL, column_name2 data_type(size) NOT NULL, CONSTRAINT Constraint_name **PRIMARY KEY** (column_name1, column_name2) etc…);	CREATE TABLE EMPLOYEE (EMPID INT NOT NULL, EMPNAME VARCHAR (100) NOT NULL, CONSTRAINT EMPIDNAME PRIMARY KEY (EMPID, NAME)) alter table EMPLOYEE add CONSTRAINT CPK_EMP PRIMARY KEY (EMPID, NAME)

Unique Key Constraint

This will make the column as unique and can accept one null value. In a table there can be multiple unique columns. The Database Engine automatically creates a UNIQUE index to enforce the uniqueness requirement of the UNIQUE constraint.

Therefore, if an attempt to insert a duplicate row is made, the Database Engine returns an error message.

☑ Quick Facts

A Table can have multiple unique key constraints but can have only one primary key

SYNTAX: Column_name1 datatype constraint constraint_name UNIQUE

EXAMPLE: EMPNAME NVARCHAR (100) constraint u_name **UNIQUE**

Another way of declaring a column as a unique key is to use the ALTER TABLE statement, which you could write as follows.

1. **ALTER TABLE TABLE_NAME ADD CONSTRAINT**
   Add a constraint to the table, if the table is already defined.
   SYNTAX:
   Alter table table_name add constraint constraint_name CONSTRAINT_TYPE (column name)
   EXAMPLE:
   Alter table EMPLOYEE add constraint uk_empname (empname)

2. **ALTER TABLE TABLE_NAME DROP CONSTRAINT**
   Drop a constraint from the table.
   SYNTAX:
   ALTER TABLE TABLE_NAME DROP CONSTRAINT constraint_name
   EXAMPLE:
   Alter table EMPLOYEE drop constraint uk_empname

Check Constraint

Check will put a limit/condition on the column that it should accept only those values defined in the check condition.

The values are already constrained by the data type, so a check constraint adds some additional constraints on the ranges, or set of allowable values. When you create a check constraint, you specify some expression so that SQL Server can constrain the valid values.

SYNTAX: Column_name datatype constraint constraint_name **CHECK** (condition)

EXAMPLE:

GENDER NVARCHAR (100) constraint C_GENDER **CHECK** (GENDER IN ('M','F'))

SALARY FLOAT constraint C_SAL **CHECK** (SALARY > 10000)

Another way of declaring a column as a check constraint is to use the ALTER TABLE statement, which you could write as follows.

1. **ALTER TABLE TABLE_NAME ADD CONSTRAINT**
   Add a constraint to the table, if the table is already defined.
   SYNTAX:
   Alter table table_name add constraint constraint_name CONSTRAINT_TYPE (condition)
   EXAMPLE:
   Alter table EMPLOYEE add constraint ck_salary empname (salary>5000)

2. **ALTER TABLE TABLE_NAME DROP CONSTRAINT**
   Drop a constraint from the table.
   SYNTAX: ALTER TABLE TABLE_NAME DROP CONSTRAINT constraint_name
   EXAMPLE: Alter table EMPLOYEE drop constraint ck_salary

Default Constraint

Default will put a default value on the column if user is not inserting any values.

SYNTAX: Column name datatype constraint constraint_name DEFAULT (value)

EXAMPLE: Joining_Date Datetime constraint d_def **default** ('10/10/2014')

1. **ALTER TABLE TABLE_NAME ADD CONSTRAINT**
   Add a constraint to the table, if the table is already defined.
   SYNTAX:
   Alter table table_name add constraint constraint_name CONSTRAINT_TYPE (value) FOR (Colum_name)
   EXAMPLE:
   Alter table EMPLOYEE add constraint df_employee DEFAULT 100 FOR (Salary)

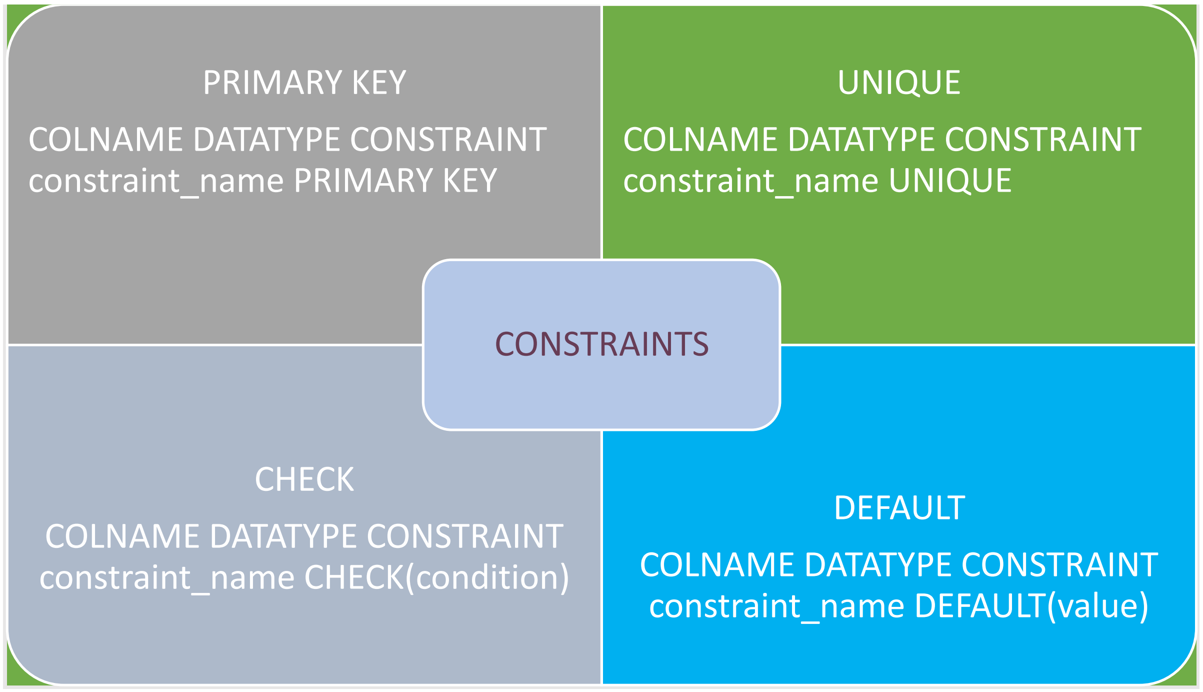
2. **ALTER TABLE TABLE_NAME DROP CONSTRAINT**
   Drop a constraint from the table.
   SYNTAX:
   ALTER TABLE TABLE_NAME DROP CONSTRAINT constraint_name
   EXAMPLE:
   Alter table EMPLOYEE drop constraint df_employee.

Example using constraints

☑ IMP: CREATE A TABLE STRUCTURE SIMILAR TO THIS

```
CREATE TABLE EMPLOYEE
(
EMPID INT CONSTRAINT PK_EMP PRIMARY KEY,
EMPNAME VARCHAR(100) CONSTRAINT U_NAM UNIQUE,
GENDER VARCHAR(10) constraint C_GENDER CHECK (GENDER IN (‘MALE’,’FEMALE’)),
DATEOFBIRTH DATETIME,
DATEOFJOINING DATETIME CONSTRAINT D_DJ DEFAULT GETDATE(),
SALARY FLOAT constraint C_SAL CHECK (SALARY > 10000),
TAX FLOAT CONSTRAINT D_DT DEFAULT 100,
DEPARTMENT VARCHAR(100),
DESIGNATION VARCHAR(100),
CITY VARCHAR(100)
)
INSERT INTO EMPLOYEE VALUES(1,'SANDEEP','MALE','01/15/1999','01/18/2021',20000,1000,
'IT','DEVELOPER','MUMBAI')
INSERT INTO EMPLOYEE VALUES(2,'KAILASH','MALE','05/20/1998','01/18/1999',30000,1000,
'IT','DEVELOPER','PUNE')
INSERT INTO EMPLOYEE VALUES(3,'SONALI','FEMALE','08/21/1999','01/20/2021',15000,1000,
'ADMIN','OFFICER','MUMBAI')
INSERT INTO EMPLOYEE VALUES(4,'ROSHINI','FEMALE','05/20/1997','01/21/2000',16000,1000,
'ADMIN','OFFICER','NASHIK')
INSERT INTO EMPLOYEE VALUES(11,'UMESH','MALE','02/25/1996','04/11/2021',25000,1000,
'HR','EXECUTIVE','MUMBAI')
INSERT INTO EMPLOYEE VALUES(12,'JAYESH','MALE','05/11/1995','01/18/2018',35000,1000,
'HR','MANAGER','PUNE')
INSERT INTO EMPLOYEE VALUES(13,'ISHITA','FEMALE','07/07/1999','01/20/2011',18000,1000,
'FINANCE','ACCOUNTANT','MUMBAI')
INSERT INTO EMPLOYEE VALUES(14,'KAVITA','FEMALE','09/08/1997','01/21/2012',19000,1000,
'FINANCE','MANAGER','PUNE')
INSERT INTO EMPLOYEE VALUES(15,'RADHA','FEMALE','05/25/1998','01/19/1999',16500,1000,
'ADMIN','OFFICER','NASHIK')
```

CHAPTER 3: SUMMARY / CHEAT SHEET


PRIMARY KEY
COLNAME DATATYPE CONSTRAINT constraint_name PRIMARY KEY
UNIQUE
COLNAME DATATYPE CONSTRAINT constraint_name UNIQUE
CONSTRAINTS
CHECK
COLNAME DATATYPE CONSTRAINT constraint_name CHECK(condition)
DEFAULT
COLNAME DATATYPE CONSTRAINT constraint_name DEFAULT(value)

CHAPTER 3: EXERCISE (ASSIGNMENTS)

CREATE A TABLE FOR COMPANY

COLUMN_NAME	DATATYPE	CONSTRAINT
COMPANYID	INT	PRIMARY KEY, IDENTITY(1,1)
COMPANYNAME	VARCHAR(100)	
WEBSITE	VARCHAR(100)	
EMAIL	VARCHAR(100)	

ADD TWO COMPANY RECORDS TO THE TABLE using INSERT STATEMENTS

CREATE A TABLE FOR DEPT

COLUMN_NAME	DATATYPE	CONSTRAINT
DEPTID	INT	PRIMARY KEY, IDENTITY(1,1)
DEPTNAME	VARCHAR(100)	
DEPTLOCATION	VARCHAR(100)	

ADD FIVE DEPT RECORDS TO THE TABLE using INSERT STATEMENTS

CREATE A TABLE FOR DESG

COLUMN_NAME	DATATYPE	CONSTRAINT
DESGID	INT	PRIMARY KEY
DESGNAME	VARCHAR(100)	
CREATEDDATE	DATETIME	DEFAULT - GETDATE()

ADD FIVE DESG RECORDS TO THE TABLE using INSERT STATEMENTS

CREATE A TABLE FOR EMPLOYEE

COLUMN_NAME	DATATYPE	CONSTRAINT
EMPID	INT	PRIMARY KEY
EMPNAME	VARCHAR(100)	UNIQUE
BASICSALARY	FLOAT	
INCENTIVE	FLOAT	
BONUS	FLOAT	
DOB	DATETIME	
DOJ	DATETIME	DEFAULT GETDATE()
GENDER	VARCHAR(100)	
EMAIL	VARCHAR(100)	UNIQUE
MOBILE	VARCHAR(10)	
CREATEDDATE	DATETIME	DEFAULT GETDATE()

ADD 10 EMPLOYEE RECORDS TO THE TABLE USING INSERT STATEMENTS

CREATE A STUDENT TABLE SIMILAR TO THIS

ROLLNO	NAME	ENGMARKS	SCNMARKS	MATHMARKS

Add around 10 records into the table USING INSERT STATEMENTS.

After adding the records add the following constraints to the table

COLUMN_NAME	CONSTRAINTS
ROLLNO	PRIMARY KEY
NAME	UNIQUE
ENGMARKS	CHECK CONSTRAINT TO ALLOW ONLY VALUES BETWEEN 0 TO 100
SCNMARKS	CHECK CONSTRAINT TO ALLOW ONLY VALUES BETWEEN 0 TO 100
MATHMARKS	CHECK CONSTRAINT TO ALLOW ONLY VALUES BETWEEN 0 TO 100

CHAPTER 3: QUESTIONS

	What is the difference between Primary Key and Unique Constraint?
	How to add a Constraint into an existing table?
	Can we add multiple constraint into a single column of a table?
	How to drop a constraint from a table?

CHAPTER 4

CHAPTER 4: CLAUSES

SQL server provides the different types of clauses. These are mention below.

Lesions in this chapter

- Lesson 1 : InBuilt Functions
- Lesson 2 : Clauses definition with example

LESSON 1: IN-BUILT FUNCTIONS

There are lots of functions which are already created in sql – server which can be used for different purpose.

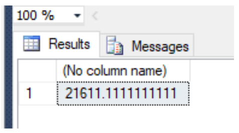
IN-BUILT FUNCTIONS are broadly categorized into the following fields:

- Aggregate Functions
- Numeric Functions
- Date Functions
- String Functions

☑Aggregate Functions

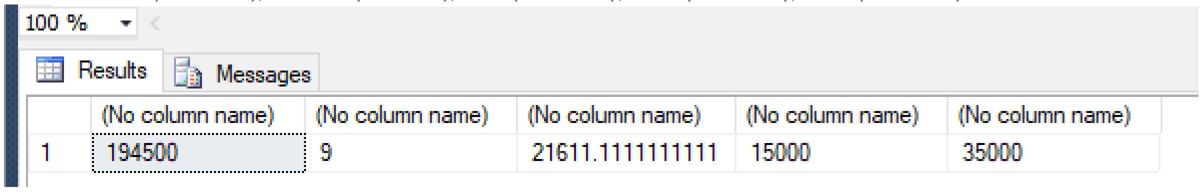
- **AVG** - Calculates the arithmetic mean (average) of the data values contained within a column. The column must contain numeric values.
- **MAX and MIN** - Calculate the maximum and minimum data value of the column, respectively. The column can contain numeric, string, and date/time values.
- **SUM** - Calculates the total of all data values in a column. The column must contain numeric values.
- **COUNT** - Calculates the number of (non-null) data values in a column. The only aggregate function not being applied to columns is COUNT (*). This function returns the number of rows (whether or not particular columns have NULL values)

EXAMPLE : SELECT AVG(SALARY) FROM EMPLOYEE



	(No column name)
1	21611.1111111111

SELECT SUM(SALARY),COUNT(EMPID),AVG(SALARY), MIN(SALARY),MAX(SALARY) FROM EMPLOYEE



	(No column name)	(No column name)	(No column name)	(No column name)	(No column name)
1	194500	9	21611.1111111111	15000	35000

☑ Numeric Functions

Numeric functions within Transact-SQL are mathematical functions for modifying numeric values. The following numeric functions are available:

Function	Explanation
ABS(n)	Returns the absolute value (i.e., negative values are returned as positive) of the numeric expression **n**. Example: SELECT ABS(–5.767) = 5.767, SELECT ABS(6.384) = 6.384
ACOS(n)	Calculates arc cosine of **n**. **n** and the resulting value belong to the FLOAT data type.
ASIN(n)	Calculates the arc sine of **n**. **n** and the resulting value belong to the FLOAT data type.
ATAN(n)	Calculates the arc tangent of **n**. **n** and the resulting value belong to the FLOAT data type.
ATN2(n, m)	Calculates the arc tangent of **n/m. n, m,** and the resulting value belong to the FLOAT data type.

CEILING(n)	Returns the smallest integer value greater or equal to the specified parameter. Examples:
	SELECT CEILING(4.88) = 5
	SELECT CEILING(–4.88) = –4

COS(n)	Calculates the cosine of **n. n** and the resulting value belong to the FLOAT data type.
COT(n)	Calculates the cotangent of **n. n** and the resulting value belong to the FLOAT data type.

DEGREES(n)	Converts radians to degrees. Examples:
	SELECT DEGREES(PI()/2) = 90.0
	SELECT DEGREES(0.75) = 42.97

EXP(n)	Calculates the value **e^n**. Example: SELECT EXP(1) = 2.7183
FLOOR(n)	Calculates the largest integer value less than or equal to the specified value **n**. Example: SELECT FLOOR(4.88) = 4

LOG(n)	Calculates the natural (i.e., base e) logarithm of **n**. Examples:
	SELECT LOG(4.67) = 1.54
	SELECT LOG(0.12) = –2.12

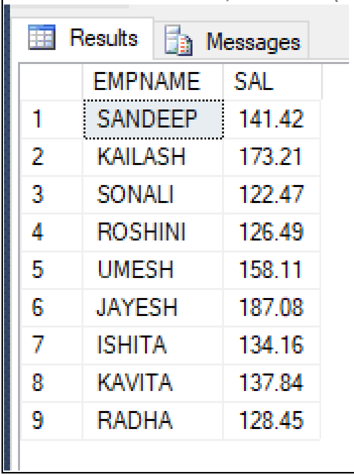
LOG10(n)	Calculates the logarithm (base 10) for **n**. Examples:
	SELECT LOG10(4.67) = 0.67
	SELECT LOG10(0.12) = –0.92

PI()	Returns the value of the number pi (3.14).
POWER(x,y)	Calculates the value **x^y**. Examples: SELECT POWER(3.12,5) = 295.65 SELECT POWER(81,0.5) = 9

RADIANS(n)	Converts degrees to radians. Examples:
	SELECT RADIANS(90.0) = 1.57
	SELECT RADIANS(42.97) = 0.75

RAND	Returns a random number between 0 and 1 with a FLOAT data type.

ROUND(n, p,[t])	Rounds the value of the number **n** by using the precision **p**. Use positive values of **p** to round on the right side of the decimal point and use negative values to round on the left side. An optional parameter **t** causes **n** to be truncated. Examples:
	SELECT ROUND(5.4567,3) = 5.4570
	SELECT ROUND(345.4567,–1) = 350.0000
	SELECT ROUND(345.4567,–1,1) = 340.0000

ROWCOUNT_BIG	Returns the number of rows that have been affected by the last Transact-SQL statement executed by the system. The return value of this function has the BIGINT data type.
SIGN(n)	Returns the sign of the value **n** as a number (+1 for positive, –1 for negative, and 0 for zero). Example: SELECT SIGN(0.88) = 1
SIN(n)	Calculates the sine of **n. n** and the resulting value belong to the FLOAT data type.
SQRT(n)	Calculates the square root of **n**. Example: SELECT SQRT(9) = 3 SELECT EMPNAME, ROUND(SQRT(SALARY),2) AS SAL FROM EMPLOYEE  Results Messages | | EMPNAME | SAL | | 1 | SANDEEP | 141.42 | | 2 | KAILASH | 173.21 | | 3 | SONALI | 122.47 | | 4 | ROSHINI | 126.49 | | 5 | UMESH | 158.11 | | 6 | JAYESH | 187.08 | | 7 | ISHITA | 134.16 | | 8 | KAVITA | 137.84 | | 9 | RADHA | 128.45 |
SQUARE(n)	Returns the square of the given expression. Example: SELECT SQUARE(9) = 81
TAN(n)	Calculates the tangent of **n. n** and the resulting value belong to the FLOAT data type.

☑ Date Functions

Date functions calculate the respective date or time portion of an expression or return the value from a time interval. Transact-SQL supports the following date functions:

Function	Explanation
GETDATE()	Returns the current system date and time. Example: SELECT GETDATE() = 2008-01-01 13:03:31.390

DATEPART(item, date)	Returns the specified part **item** of a date **date** as an integer. Examples:
	SELECT DATEPART(month, '01.01.2005') = 1 (1 = January)
	SELECT DATEPART(weekday, '01.01.2005') = 7 (7 = Sunday)

DATENAME(item, date)	Returns the specified part **item** of the date **date** as a character string. Example: SELECT DATENAME(weekday, '01.01.2005') = Saturday

DATEDIFF(item,dat1,dat2)	Calculates the difference between the two date parts **dat1** and **dat2** and returns the result as an integer in units specified by the value **item**. Example:
	SELECT DATEDIFF (year, BirthDate, GETDATE ()) AS age FROM employee; -> returns the age of each employee.
	
	SELECTDATEDIFF(YEAR,'05/10/1984',GETDATE())
	
	SELECTDATEDIFF(MONTH,'05/10/1984',GETDATE())
	
	SELECTDATEDIFF(DAY,'05/10/1984',GETDATE())
	
	SELECTDATEDIFF(HOUR,'05/10/1984',GETDATE())
	
	SELECTDATEDIFF(MINUTE,'05/10/1984',GETDATE())

DATEADD(i,n,d)	Adds the number **n** of units specified by the value **i** to the given date **d**. Example: SELECT DATEADD(DAY,3,HireDate) AS age FROM employee; -> adds three days to the starting date of employment of every employee (see the **sample** database).
DAY()	This function returns an integer representing the day part of the specified date. select DAY(<date>)
MONTH()	This function returns an integer representing the month part of the specified date. select MONTH(<date>)

YEAR()	This function returns an integer representing the year part of the specified date. select YEAR(<date>)

☑ String Functions

String functions are used to manipulate data values in a column, usually of a character data type. Transact-SQL supports the following string functions:

Function	Explanation
ASCII(character)	Converts the specified character to the equivalent integer (ASCII) code. Returns an integer. Example: SELECT ASCII('A') = 65
CHAR(integer)	Converts the ASCII code to the equivalent character. Example: SELECT CHAR (65) = 'A'.
CHARINDEX(z1,z2)	Returns the starting position where the partial string **z1** first occurs in the string **z2**. Returns 0 if **z1** does not occur in **z2**. Example: SELECT CHARINDEX ('bl', 'table') = 3.
DIFFERENCE(z1,z2)	Returns an integer, 0 through 4, that is the difference of SOUNDEX values of two strings **z1** and **z2**. (SOUNDEX returns a number that specifies the sound of a string. With this method, strings with similar sounds can be determined.) Example: SELECT DIFFERENCE('spelling', 'telling') = 2 (sounds a little bit similar, 0 = doesn't sound similar)
LEFT(z, length)	Returns the first **length** characters from the string **z**. Select left('Ravi Singh',4)
LEN(z)	Returns the number of characters, instead of the number of bytes, of the specified string expression, excluding trailing blanks. Select Len('Ravi Singh',4)
LOWER(z1)	Converts all uppercase letters of the string **z1** to lowercase letters. Lowercase letters and numbers, and other characters, do not change. Example: SELECT LOWER('BIG') = 'big'
LTRIM(z)	Removes leading blanks in the string **z**. Example: SELECT LTRIM(' String') = 'String'
NCHAR(i)	Returns the Unicode character with the specified integer code, as defined by the Unicode standard.
QUOTENAME(char_string)	Returns a Unicode string with the delimiters added to make the input string a valid delimited identifier.

PATINDEX(%p%,expr)	Returns the starting position of the first occurrence of a pattern **p** in a specified expression **expr**, or zeros if the pattern is not found. Examples: 1) SELECT PATINDEX('%gs%', 'longstring') = 4; 2) SELECT RIGHT(ContactName, LEN(ContactName)-PATINDEX('% %',ContactName)) AS First_name FROM Customers; (The second query returns all first names from the **customers** column.)
REPLACE(str1,str2,str3)	Replaces all occurrences of the **str2** in the **str1** with the **str3**. Example: SELECT REPLACE('shave' , 's' , 'be') = behave
REPLICATE(z,i)	Repeats string **z** **i** times. Example: SELECT REPLICATE('a',10) = 'aaaaaaaaaa'
REVERSE(z)	Displays the string **z** in the reverse order. Example: SELECT REVERSE('calculate') = 'etaluclac'
RIGHT(z ,length)	Returns the last **length** characters from the string **z**. Example: SELECT RIGHT('Notebook',4) = 'book'
RTRIM(z)	Removes trailing blanks of the string **z**. Example: SELECT RTRIM('Notebook ') = 'Notebook'
SOUNDEX(a)	Returns a four-character SOUNDEX code to determine the similarity between two strings. Example: SELECT SOUNDEX('spelling') = S145
SPACE(length)	Returns a string with spaces of length specified by **length**. Example: SELECT SPACE = ' '
STR(f,[len [,d]])	Converts the specified float expression **f** into a string. **Len** is the length of the string including decimal point, sign, digits, and spaces (10 by default), and **d** is the number of digits to the right of the decimal point to be returned. Example: SELECT STR(3.45678,4,2) = '3.46'
STUFF(z1,a,length,z2)	Replaces the partial string **z1** with the partial string **z2** starting at position a, replacing **length** characters of **z1**. Examples: SELECT STUFF('Notebook',5,0, ' in a ') = 'Note in a book' SELECT STUFF('Notebook',1,4, 'Hand') = 'Handbook'
SUBSTRING(z,a,length)	Creates a partial string from string **z** starting at the position with a length of **length**. Example: SELECT SUBSTRING('wardrobe',1,4) = 'ward'
UNICODE	Returns the integer value, as defined by the Unicode standard, for the first character of the input expression.
UPPER(z)	Converts all lowercase letters of string **z** to uppercase letters. Uppercase letters and numbers do not change. Example: SELECT UPPER('loWer') = 'LOWER'

CONVERT ()

Explicitly converts an expression of one data type to another.

CAST and CONVERT provide similar functionality.

SYNTAX:
CONVERT (data_type [(length)] , expression [, style])

Here 'EXPRESSION' can be any valid expression, 'DATA_TYPE' is the target data type and 'LENGTH' is an optional integer that specifies the length of the target data type. The default value is 30. STYLE is an integer expression that specifies how the CONVERT function is to translate expression. If style is NULL then NULL is returned. The range is determined by data_type.

Example: Converting DATETIME to VARCHAR

SELECTCONVERT (VARCHAR, GETDATE (), 0)
Output-- May 4 2013 2:14PM

Example: Converting VARCHAR to DATETIME

The style code is equally important when converting a VARCHAR to a DATETIME value. I'm using the output from the previous SQL code and different style codes, let's see how it works.

SELECT CONVERT (DATETIME, 'May 4 2013 2:14PM', 0)
Output-- 2013-05-04 14:14:00.000

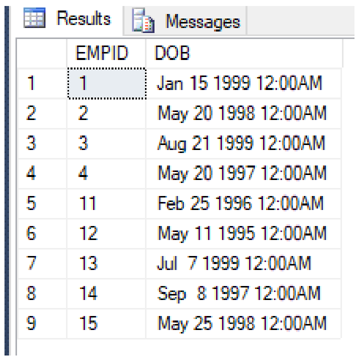
SELECT CONVERT (DATETIME, 'May 4 2013 2:14PM', 130)
Output -- Conversion failed when converting date and/or time from character string.

Date Format	Standard	SQL Statement	Sample Output
Mon DD YYYY [1] HH:MIAM (or PM)	Default	SELECT CONVERT(VARCHAR(20), GETDATE(), 100)	Jan 1 2005 1:29PM [1]
MM/DD/YY	USA	SELECT CONVERT(VARCHAR(8), GETDATE(), 1) AS [MM/DD/YY]	11/23/98
MM/DD/YYYY	USA	SELECT CONVERT(VARCHAR(10), GETDATE(), 101) AS [MM/DD/YYYY]	11/23/1998
YY.MM.DD	ANSI	SELECT CONVERT(VARCHAR(8), GETDATE(), 2) AS [YY.MM.DD]	72.01.01
YYYY.MM.DD	ANSI	SELECT CONVERT(VARCHAR(10), GETDATE(), 102) AS [YYYY.MM.DD]	1972.01.01
DD/MM/YY	British/French	SELECT CONVERT(VARCHAR(8), GETDATE(), 3) AS [DD/MM/YY]	19/02/72
DD/MM/YYYY	British/French	SELECT CONVERT(VARCHAR(10), GETDATE(), 103) AS [DD/MM/YYYY]	19/02/1972

DD.MM.YY	German	SELECT CONVERT(VARCHAR(8), GETDATE(), 4) AS [DD.MM.YY]	25.12.05
DD.MM.YYYY	German	SELECT CONVERT(VARCHAR(10), GETDATE(), 104) AS [DD.MM.YYYY]	25.12.2005
DD-MM-YY	Italian	SELECT CONVERT(VARCHAR(8), GETDATE(), 5) AS [DD-MM-YY]	24-01-98
DD-MM-YYYY	Italian	SELECT CONVERT(VARCHAR(10), GETDATE(), 105) AS [DD-MM-YYYY]	24-01-1998
DD Mon YY [1]	-	SELECT CONVERT(VARCHAR(9), GETDATE(), 6) AS [DD MON YY]	04 Jul 06 [1]
DD Mon YYYY [1]	-	SELECT CONVERT(VARCHAR(11), GETDATE(), 106) AS [DD MON YYYY]	04 Jul 2006 [1]
Mon DD, YY [1]	-	SELECT CONVERT(VARCHAR(10), GETDATE(), 7) AS [Mon DD, YY]	Jan 24, 98 [1]
Mon DD, YYYY [1]	-	SELECT CONVERT(VARCHAR(12), GETDATE(), 107) AS [Mon DD, YYYY]	Jan 24, 1998 [1]
HH:MM:SS	-	SELECT CONVERT(VARCHAR(8), GETDATE(), 108)	03:24:53
Mon DD YYYY HH:MI:SS:MMMAM (or PM) [1]	Default + milliseconds	SELECT CONVERT(VARCHAR(26), GETDATE(), 109)	Apr 28 2006 12:32:29:253PM [1]
MM-DD-YY	USA	SELECT CONVERT(VARCHAR(8), GETDATE(), 10) AS [MM-DD-YY]	01-01-06
MM-DD-YYYY	USA	SELECT CONVERT(VARCHAR(10), GETDATE(), 110) AS [MM-DD-YYYY]	01-01-2006
YY/MM/DD	-	SELECT CONVERT(VARCHAR(8), GETDATE(), 11) AS [YY/MM/DD]	98/11/23
YYYY/MM/DD	-	SELECT CONVERT(VARCHAR(10), GETDATE(), 111) AS [YYYY/MM/DD]	1998/11/23
YYMMDD	ISO	SELECT CONVERT(VARCHAR(6), GETDATE(), 12) AS [YYMMDD]	980124
YYYYMMDD	ISO	SELECT CONVERT(VARCHAR(8), GETDATE(), 112) AS [YYYYMMDD]	19980124
DD Mon YYYY HH:MM:SS:MMM(24h) [1]	Europe default + milliseconds	SELECT CONVERT(VARCHAR(24), GETDATE(), 113)	28 Apr 2006 00:34:55:190 [1]
HH:MI:SS:MMM(24H)	-	SELECT CONVERT(VARCHAR(12), GETDATE(), 114) AS [HH:MI:SS:MMM(24H)]	11:34:23:013

YYYY-MM-DD HH:MI:SS(24h)	ODBC Canonical	SELECT CONVERT(VARCHAR(19), GETDATE(), 120)	1972-01-01 13:42:24
YYYY-MM-DD HH:MI:SS.MMM(24h)	ODBC Canonical (with milliseconds)	SELECT CONVERT(VARCHAR(23), GETDATE(), 121)	1972-02-19 06:35:24.489
YYYY-MM-DDTHH:MM:SS:MMM	ISO8601	SELECT CONVERT(VARCHAR(23), GETDATE(), 126)	1998-11-23T11:25:43:250
DD Mon YYYY HH:MI:SS:MMMAM [1]	Kuwaiti	SELECT CONVERT(VARCHAR(26), GETDATE(), 130)	28 Apr 2006 12:39:32:429AM [1]
DD/MM/YYYY HH:MI:SS:MMMAM	Kuwaiti	SELECT CONVERT(VARCHAR(25), GETDATE(), 131)	28/04/2006 12:39:32:429AM

```
SELECT EMPID, CONVERT(VARCHAR(20),DATEOFBIRTH,100) AS DOB FROM EMPLOYEE
```



```
   EMPID  DOB
1  1      Jan 15 1999 12:00AM
2  2      May 20 1998 12:00AM
3  3      Aug 21 1999 12:00AM
4  4      May 20 1997 12:00AM
5  11     Feb 25 1996 12:00AM
6  12     May 11 1995 12:00AM
7  13     Jul  7 1999 12:00AM
8  14     Sep  8 1997 12:00AM
9  15     May 25 1998 12:00AM
```

Here are some more date formats that does not come standard in SQL Server as part of the **CONVERT** function.

Date Format	**SQL Statement**	**Sample Output**

YY-MM-DD	SELECT SUBSTRING(CONVERT(VARCHAR(10), GETDATE(), 120), 3, 8) AS [YY-MM-DD]
	SELECT REPLACE(CONVERT(VARCHAR(8), GETDATE(), 11), '/', '-') AS [YY-MM-DD]
		99-01-24

Mon-YY [1]	SELECT REPLACE(RIGHT(CONVERT(VARCHAR(9), GETDATE(), 6), 6), ' ', '-') AS [Mon-YY]	Sep-02 [1]

LESSON 2: CLAUSES DEFINATION WITH EXAMPLES

This List below shows important clauses of SQL SERVER with syntax and examples.

☑ (Note: Will use the EMPLOYEE TABLE created in previous chapters for the examples of clauses)

CLAUSES	Description
SELECT	**Is used to view the columns values of the table.** **Syntax :** Select * from table_name Select * *from* *EMPLOYEE*
WHERE	**Is used to apply filter to the select clause** *Note : The WHERE clause is evaluated before the SELECT clause* **SYNTAX : SELECT * FROM TABLE_NAME WHERE CONDITION** EXAMPLE : SELECT * FROM EMPLOYEE WHERE SALARY > 20000
ORDER BY	**To sort the table based upon a column value in ascending or descending order.** **SYNTAX : SELECT * FROM TABLE_NAME ORDER BY COLUMN_NAME (ASC/DESC) (DEFAULT ASC)** **EXAMPLE : SELECT * FROM EMPLOYEE ORDER BY SALARY DESC**

AS	**Used to add an alias name(new name) to the column in the query**
	
	SYNTAX : SELECT COLUMNNAME AS NEW-COLUMNNAME FROM TABLENAME
	
	EXAMPLE : SELECT EMPID, FIRSTNAME + ' ' + LASTNAME AS FULLNAME FROM EMPLOYEES;
	
	SELECT EMPID,EMPNAME AS EMPLOYEENAME,SALARY,CITY FROM EMPLOYEE

IN	**To search multiple values in where condition.** **SYNTAX : SELECT * FROM TABLE_NAME WHERE COL IN(VALUE,VALUE2)** **EXAMPLE : SELECT * FROM EMPLOYEE WHERE DEPARTMENT IN('IT','HR')**
DISTINCT	**Used to select distinct values of the rows / column from the table.** **SYNTAX : SELECT DISTINCT * from table_name / SELECT DISTINCT colname from table_name** **EXAMPLE : SELECT DISTINCT DEPARTMENT FROM EMPLOYEE**
TOP	**Is used to find top number of records from the query** **SYNTAX : SELECT TOP [N] * FROM TABLENAME** **EXAMPLE : SELECT TOP 3 * FROM EMPLOYEE ORDER BY SALARY DESC**
OFFSET & FETCH	OFFSET and FETCH clause is used to skip certain number of rows from the query and fetch certain number of rows resultset generated by the query. The OFFSET and FETCH clauses appear right after the ORDER BY clause.You first specify the OFFSET clause indicating how many rows you want to skip (0 if you don't want to skip any); you then optionally specify the FETCH clause indicating how many rows you want to filter.

SYNTAX : SELECT * FROM TABLENAME ORDER BY COLNAME OFFSET N ROWS FETCH NEXT N ROWS ONLY;

EXAMPLE : GET THE 3rd & 4th Highest Salary Holder

SELECT * FROM EMPLOYEE ORDER BY SALARY DESC OFFSET 2 ROWS FETCH NEXT 2 ROWS ONLY;

LIKE

Is used to search string patterns in the query.

Wildcard Meaning Example

% (percent sign) Any string including an empty one.

'D%': string starting with D

_ (underscore) A single character

'_D%': string where second character is D

[<*character list*>] A single character from a list

'[AC]%': string where first character is A or C

[<*character range*>] A single character from a range

'[0-9]%': string where first character is a digit

[^<*character list or range*>] A single character that is not in the list or range

'[^0-9]%': string where first character is not a digit

SYNTAX : SELECT * FROM TABLE_NAME WHERE COLNAME LIKE 'STRINGPATTERN'

EXAMPLE : Show all name starting K

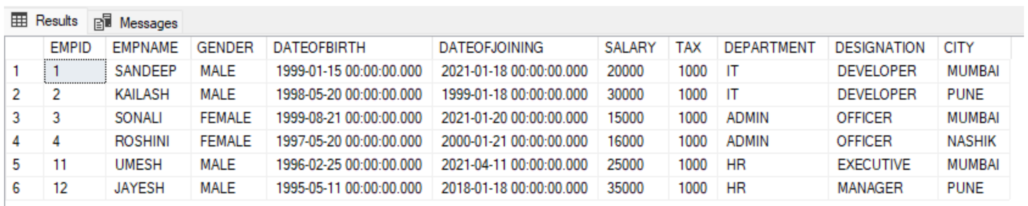
SELECT * FROM EMPLOYEE WHERE EMPNAME LIKE 'K%'

Show all name starting K or S

SELECT * FROM EMPLOYEE WHERE EMPNAME LIKE '[KS]%'

Show all names not ending with A.

SELECT * FROM EMPLOYEE WHERE EMPNAME LIKE '%[^A]'



Results | Messages

	EMPID	EMPNAME	GENDER	DATEOFBIRTH	DATEOFJOINING	SALARY	TAX	DEPARTMENT	DESIGNATION	CITY
1	1	SANDEEP	MALE	1999-01-15 00:00:00.000	2021-01-18 00:00:00.000	20000	1000	IT	DEVELOPER	MUMBAI
2	2	KAILASH	MALE	1998-05-20 00:00:00.000	1999-01-18 00:00:00.000	30000	1000	IT	DEVELOPER	PUNE
3	3	SONALI	FEMALE	1999-08-21 00:00:00.000	2021-01-20 00:00:00.000	15000	1000	ADMIN	OFFICER	MUMBAI
4	4	ROSHINI	FEMALE	1997-05-20 00:00:00.000	2000-01-21 00:00:00.000	16000	1000	ADMIN	OFFICER	NASHIK
5	11	UMESH	MALE	1996-02-25 00:00:00.000	2021-04-11 00:00:00.000	25000	1000	HR	EXECUTIVE	MUMBAI
6	12	JAYESH	MALE	1995-05-11 00:00:00.000	2018-01-18 00:00:00.000	35000	1000	HR	MANAGER	PUNE

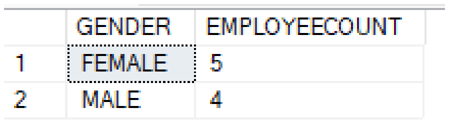
GROUP BY

Used to group the summation based up a column.

SYNTAX : SELECT COLNAME, AGGREGRATE_FUNCTION FROM TABLE_NAME GROUP BY COLNAME

EXAMPLE : TOTAL EMPLOYEE COUNT GROUPED BY GENDER

SELECT GENDER,COUNT (EMPID) AS EMPLOYEECOUNT FROM EMPLOYEE GROUP BY GENDER



	GENDER	EMPLOYEECOUNT
1	FEMALE	5
2	MALE	4

SELECT GENDER,SUM(SALARY) AS TOTALSALARY FROM EMPLOYEE GROUP BY GENDER

SELECT DEPARTMENT,SUM(SALARY) AS TOTALSALARY FROM EMPLOYEE GROUP BY DEPARTMENT

HAVING

USED TO APPLY FILTER AFTER GROUP BY.MOSTING SOME CONDITIONS ARE NOT POSSIBLE IN WHERE CLAUSES LIKE FOR EG(AGGREGRATE FUNCTIONS) , FOR THIS WE HAVE TO USE HAVING CLAUSE.

SYNTAX : SELECT COLNAME, AGGREGRATE_FUNCTION FROM TABLE_NAME GROUP BY COLNAME HAVING CONDITION

NOTE : WE CAN GIVE AGGREGRATE_FUNCTION CONDITION IN HAVING CLAUSE

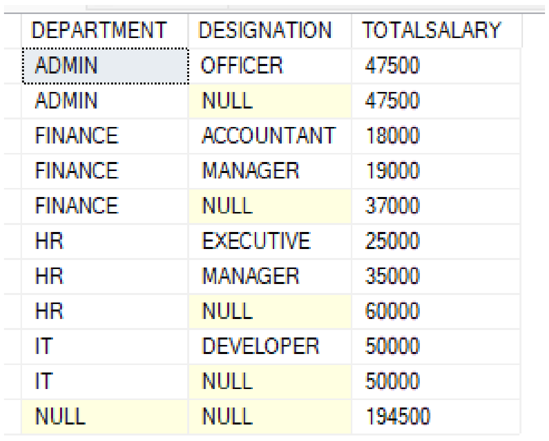
EXAMPLE : SELECT DEPARTMENT,SUM(SALARY) AS TOTALSALARY FROM EMPLOYEE GROUP BY DEPARTMENT HAVING SUM(SALARY) > 40000

ROLLUP CLAUSE

ROLLUP generates a result set that shows aggregates for a hierarchy of values in the selected columns.It is used with Group by clause.

SYNTAX : SELECT COLNAME, AGGREGRATE_FUNCTION FROM TABLE_NAME GROUP BY COLNAME WITH ROLLUP

EXAMPLE : SELECT DEPARTMENT,DESIGNATION,SUM(SALARY) AS TOTALSALARY FROM EMPLOYEE GROUP BY DEPARTMENT,DESIGNATION WITH ROLLUP



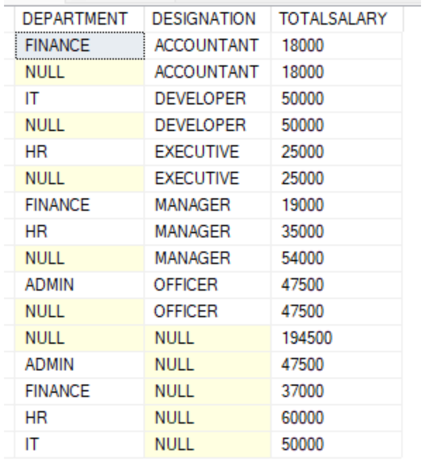
DEPARTMENT	DESIGNATION	TOTALSALARY
ADMIN	OFFICER	47500
ADMIN	NULL	47500
FINANCE	ACCOUNTANT	18000
FINANCE	MANAGER	19000
FINANCE	NULL	37000
HR	EXECUTIVE	25000
HR	MANAGER	35000
HR	NULL	60000
IT	DEVELOPER	50000
IT	NULL	50000
NULL	NULL	194500

CUBE CLAUSE

CUBE generates a result set that shows aggregates for all combinations of values in the selected columns.It is used with GROUP BY.

SYNTAX : SELECT COLNAME, AGGREGRATE_FUNCTION FROM TABLE_NAME GROUP BY COLNAME WITH CUBE

```
SELECT DEPARTMENT,DESIGNATION,SUM(SALARY) AS TOTALSALARY FROM EMPLOYEE
GROUP BY DEPARTMENT,DESIGNATION WITH CUBE
```



DEPARTMENT	DESIGNATION	TOTALSALARY
FINANCE	ACCOUNTANT	18000
NULL	ACCOUNTANT	18000
IT	DEVELOPER	50000
NULL	DEVELOPER	50000
HR	EXECUTIVE	25000
NULL	EXECUTIVE	25000
FINANCE	MANAGER	19000
HR	MANAGER	35000
NULL	MANAGER	54000
ADMIN	OFFICER	47500
NULL	OFFICER	47500
NULL	NULL	194500
ADMIN	NULL	47500
FINANCE	NULL	37000
HR	NULL	60000
IT	NULL	50000

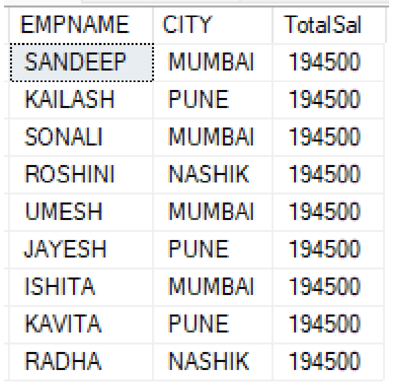
OVER CLAUSE

To add aggregate functions on the query

OVER allows you to get aggregate information without using a GROUP BY. In other words, you can retrieve detail rows, and get aggregate data alongside it.

SYNTAX : SELECT COLNAME,AGG-FUNCTIONS OVER () AS NEWCOLUMNNAME FROM TABLENAME

EXAMPLE SELECT EMPNAME,CITY,SUM(SALARY) OVER () AS TotalSal FROM EMPLOYEE

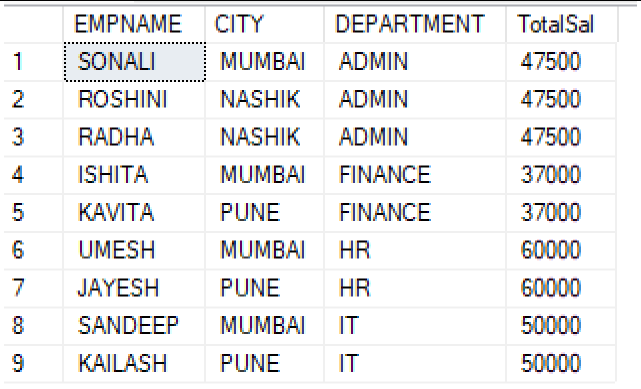


EMPNAME	CITY	TotalSal
SANDEEP	MUMBAI	194500
KAILASH	PUNE	194500
SONALI	MUMBAI	194500
ROSHINI	NASHIK	194500
UMESH	MUMBAI	194500
JAYESH	PUNE	194500
ISHITA	MUMBAI	194500
KAVITA	PUNE	194500
RADHA	NASHIK	194500

OVER (PARTITION BY) ; OVER, as used in our previous example, exposes the entire resultset to the aggregation…"Salary" was the sum of all [Sal] in the result set. We can BREAK UP that result set into partitions with the use of PARTITION BY:

```
SELECT EMPNAME,CITY,DEPARTMENT,SUM(SALARY) OVER (PARTITION BY DEPARTMENT) AS TotalSal
FROM EMPLOYEE
```

<table>
<tr><td></td><td>

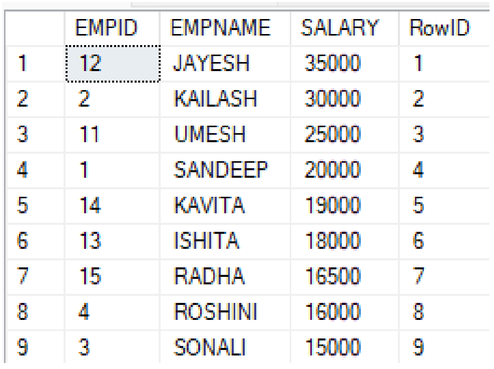


	EMPNAME	CITY	DEPARTMENT	TotalSal
1	SONALI	MUMBAI	ADMIN	47500
2	ROSHINI	NASHIK	ADMIN	47500
3	RADHA	NASHIK	ADMIN	47500
4	ISHITA	MUMBAI	FINANCE	37000
5	KAVITA	PUNE	FINANCE	37000
6	UMESH	MUMBAI	HR	60000
7	JAYESH	PUNE	HR	60000
8	SANDEEP	MUMBAI	IT	50000
9	KAILASH	PUNE	IT	50000

</td></tr>
<tr><td>ROW_NUMBER ()</td><td>

TO SPECIFY THE ROW NUMBER OF THE TABLE IN A INCREMENT FASHION FOR EACH ROW

```
SYNTAX : SELECT COLUMNAME, ROW_NUMBER() over(order by COLNAME desc) AS
'RowID' FROM TABLENAME
EXAMPLE : SELECT EMPID,EMPNAME,SALARY,ROW_NUMBER() over(order by Salary desc)
AS 'RowID' FROM EMPLOYEE
```



	EMPID	EMPNAME	SALARY	RowID
1	12	JAYESH	35000	1
2	2	KAILASH	30000	2
3	11	UMESH	25000	3
4	1	SANDEEP	20000	4
5	14	KAVITA	19000	5
6	13	ISHITA	18000	6
7	15	RADHA	16500	7
8	4	ROSHINI	16000	8
9	3	SONALI	15000	9

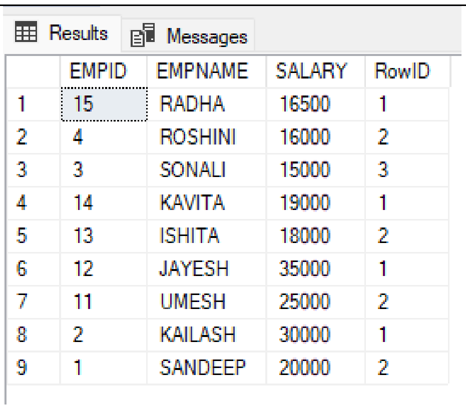
</td></tr>
<tr><td>PARTITION BY</td><td>

PARTITION BY – If you supply this parameter, then the row number will reset based on the value changing in the columns supplied. This is kind of like using a GROUP BY.

ORDER BY – This is the specified order of the row number that you would like. If you wanted the order of the row number to increment by an employee name (alphabetically), you do that here.

```
SELECT COLNAME1 ,ROW_NUMBER() over(partition by COLUMNNAME order by
COLUMNNAME) FROM TABLENAME

SELECT EMPID,EMPNAME,SALARY,ROW_NUMBER() over(partition by DEPARTMENT order
by Salary desc) AS 'RowID' FROM EMPLOYEE
```

</td></tr>
</table>



Results | Messages

	EMPID	EMPNAME	SALARY	RowID
1	15	RADHA	16500	1
2	4	ROSHINI	16000	2
3	3	SONALI	15000	3
4	14	KAVITA	19000	1
5	13	ISHITA	18000	2
6	12	JAYESH	35000	1
7	11	UMESH	25000	2
8	2	KAILASH	30000	1
9	1	SANDEEP	20000	2

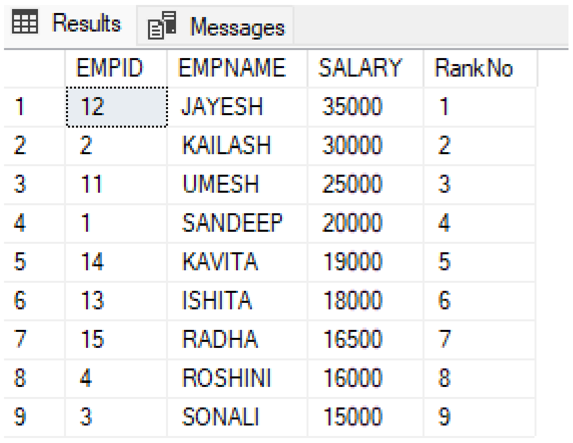
Rank

This function will assign a unique number to each distinct row, but it leaves a gap between the groups.

Syntax :

SELECT Col_Value, Rank() OVER (ORDER BY Col_Value) AS 'RowID' FROM @Table;

```
SELECT EMPID,EMPNAME,SALARY,RANK() over(order by Salary desc) AS
'RankNo' FROM EMPLOYEE
```



Results | Messages

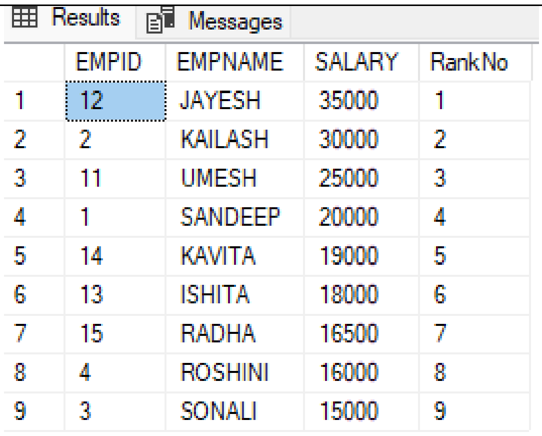
	EMPID	EMPNAME	SALARY	RankNo
1	12	JAYESH	35000	1
2	2	KAILASH	30000	2
3	11	UMESH	25000	3
4	1	SANDEEP	20000	4
5	14	KAVITA	19000	5
6	13	ISHITA	18000	6
7	15	RADHA	16500	7
8	4	ROSHINI	16000	8
9	3	SONALI	15000	9

DENSE_RANK ()

This function is similar to Rank with only difference, this will not leave gaps between groups.

SELECT   Col_Value, DENSE_RANK() OVER (ORDER BY Col_Value) AS 'RowID' FROM @Table;

```
SELECT EMPID,EMPNAME,SALARY,DENSE_RANK() over(order by Salary desc) AS
'RankNo' FROM EMPLOYEE
```



	EMPID	EMPNAME	SALARY	RankNo
1	12	JAYESH	35000	1
2	2	KAILASH	30000	2
3	11	UMESH	25000	3
4	1	SANDEEP	20000	4
5	14	KAVITA	19000	5
6	13	ISHITA	18000	6
7	15	RADHA	16500	7
8	4	ROSHINI	16000	8
9	3	SONALI	15000	9

SEQUENCE OBJECTS IN SQL SERVER

Sequence objects are used to sequentially generate numeric values. They were introduced in SQL Server 2012. Sequence objects are similar to the IDENTITY column in any SQL table. However, unlike the IDENTITY column, they are independent and are not attached to any table. Sequence objects are used both independently and within the DML statements i.e. INSERT, UPDATE and DELETE.

SYNTAX

```
CREATE SEQUENCE [schema].[Name_of_Sequence]   [ AS <data type> ]

[ START WITH <value> ]   [ INCREMENT BY <value> ]  [ MINVALUE <value > | NO MINVALUE ]
[ MAXVALUE <value> | NO MAXVALUE ]   [ CYCLE | NO CYCLE ]   [ CACHE value | NO CACHE ];
```

CREATE SEQUENCE [dbo].[NewCounter] AS INT START WITH 5 INCREMENT BY 5

SELECT NEXT VALUE FOR [dbo].[NewCounter]

PARAMETER	DESCRIPTION
CREATE SEQUENCE	Used to create a sequence followed by a database schema and the name of the sequence
AS	Specifies the data type of the sequence. Data types can be Decimal, Int, SmallInt, TinyInt, and BigInt. The default value for the data type is BigInt
START WITH	Sets the starting value for the sequence object
INCREMENT BY	Sets the amount that you want your sequence object to increment by
MIN VALUE	This is an optional parameter that specifies the minimum value for the sequence object
MAX VALUE	This is an optional parameter that sets the maximum value for the sequence object
CYCLE	This specifies if the sequence object should be restarted once it has reached its maximum or minimum value. It is an optional parameter for which the default value is NO CYCLE
CACHE	This is used to cache sequence object values. It is also optional parameter with the default value of NO CACHE

SYNTAX & EXAMPLE

Using a sequence object in a single table example

CREATE A TABLE	**CREATE A SEQUENCE :** order_number that starts with 1 and is incremented by 1:
CREATE TABLE purchase_orders(order_id INT PRIMARY KEY, vendor_id int NOT NULL, order_date date NOT NULL);	**CREATE SEQUENCE order_number AS INT START WITH 1 INCREMENT BY 1**

After that, insert three rows into the procurement.purchase_orders table and uses values generated by the order_number sequence:

```
INSERT INTO  purchase_orders  (order_id,   vendor_id,   order_date)  VALUES
   (NEXT VALUE FOR order_number,1,'2019-04-30');

INSERT INTO purchase_orders   (order_id,   vendor_id,   order_date)
VALUES  (NEXT VALUE FOR order_number,2,'2019-05-01');

INSERT INTO purchase_orders    (order_id,    vendor_id,   order_date) VALUES  (NEXT VALUE FOR
order_number,3,'2019-05-02');
```

Finally, view the content of the purchase_orders table:

```
SELECT     order_id,    vendor_id,    order_date FROM    purchase_orders;
```

A sequence object once created can also be used with multiple tables.

Sequence vs. Identity columns : Sequences, different from the identity columns, are not associated with a table. The relationship between the sequence and the table is controlled by applications. In addition, a sequence can be shared across multiple tables.

The following table illustrates the main differences between sequences and identity columns:

Property/Feature	**Identity**	**Sequence Object**
Allow specifying minimum and/or maximum increment values	No	Yes
Allow resetting the increment value	No	Yes
Allow caching increment value generating	No	Yes
Allow specifying starting increment value	Yes	Yes
Allow specifying increment value	Yes	Yes
Allow using in multiple tables	No	Yes

When to use sequences (Use sequence in following cases):

- The application requires a number before inserting values into the table.

- The application requires sharing a sequence of numbers across multiple tables or multiple columns within the same table.
- The application requires to restart the number when a specified value is reached.
- The application requires multiple numbers to be assigned at the same time. Note that you can call the stored procedure sp_sequence_get_range to retrieve several numbers in a sequence at once.
- The application needs to change the specification of the sequence like maximum value.

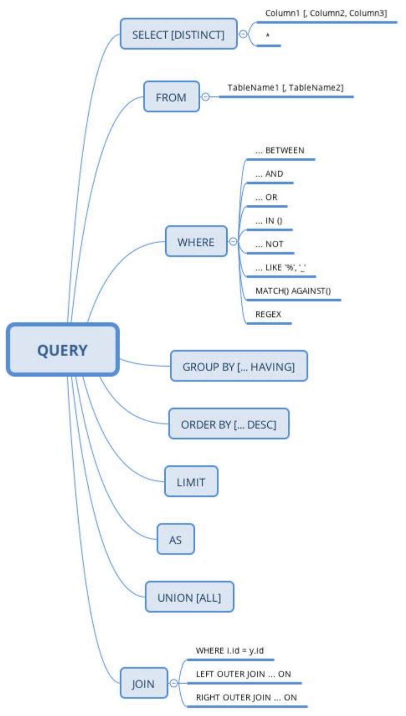
Getting sequences information

You use the view sys.sequences to get the detailed information of sequences.

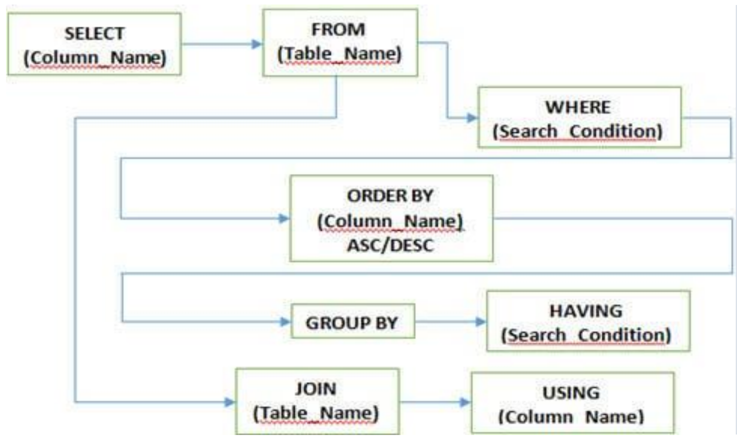
```
SELECT   *  FROM  sys.sequences;
```

CHAPTER 4: SUMMARY / CHEAT SHEET

Different Types of clauses :



The order of clause is as follows:



CHAPTER 4: EXERCISE 1(ASSIGNMENTS)

CREATE An EMPLOYEE TABLE BY EXECUTING THE BELOW SQL CREATE TABLE COMMAND AND INSERT DATA USING THE INSERT QUERIES.WRITE ANSWERS(USING SELECT QUERIES) FOR THE BELOW QUESTIONS USING THE ABOVE TABLE

```
CREATE TABLE [dbo].[EMPLOYEES](
        [EMPID] [int] IDENTITY(1,1) NOT NULL,
        [EMPNAME] [varchar](100) NULL,
        [SALARY] [float] NULL,
        [TAX] [float] NULL,
        [DOJ] [datetime] NULL,
        [DEPT] [varchar](100) NULL,
        [DESG] [varchar](100) NULL,
        [DEPTMANAGERNAME] [varchar](100) NULL,
        [DEPTLOCATION] [varchar](100) NULL, CONSTRAINT [pk_empid] PRIMARY KEY CLUSTERED
(       [EMPID] ASC ))
GO
SET IDENTITY_INSERT [dbo].[EMPLOYEES] ON
GO
INSERT [dbo].[EMPLOYEES] VALUES (1, N'AKASH', 70000, 2000, CAST(N'2017-05-20' AS DateTime), N'IT',
N'SOFTWARE DEVELOPER', N'SURESH', N'PUNE')
INSERT [dbo].[EMPLOYEES]  VALUES (2, N'AKSHAY', 69000, 1900, CAST(N'2017-07-14' AS DateTime), N'IT',
N'ARCHITECT', N'SURESH', N'PUNE')
INSERT [dbo].[EMPLOYEES] VALUES (3, N'ASHWIN', 77000, 2100, CAST(N'2017-09-14' AS DateTime), N'IT',
N'ARCHITECT', N'SURESH', N'PUNE')
INSERT [dbo].[EMPLOYEES] VALUES (4, N'CHINMAY', 88000, 2900, CAST(N'2017-09-10' AS DateTime), N'HR',
N'MANAGER', N'ANITA', N'MUMBAI')
INSERT [dbo].[EMPLOYEES]  VALUES (5, N'DEBAYAN', 65000, 2100, CAST(N'2017-10-18' AS DateTime), N'HR',
N'EXECUTIVE', N'ANITA', N'MUMBAI')
INSERT [dbo].[EMPLOYEES] VALUES (6, N'MANGESH', 99000, 2900, CAST(N'2017-10-10T00:00:00.000' AS
DateTime), N'HR', N'MANAGER', N'ANITA', N'MUMBAI')
INSERT [dbo].[EMPLOYEES] VALUES (7, N'NIKHIL', 56000, 2100, CAST(N'2017-07-18T00:00:00.000' AS DateTime),
N'HR', N'EXECUTIVE', N'ANITA', N'MUMBAI')
INSERT [dbo].[EMPLOYEES] VALUES (10, N'SANDEEP', 45000, 460, CAST(N'2017-07-25T00:00:00.000' AS
DateTime), N'IT', N'SOFTWARE DEVELOPER', N'SURESH', N'PUNE')
INSERT [dbo].[EMPLOYEES] VALUES (11, N'NIRAJ', 48000, 460, CAST(N'2017-07-25T00:00:00.000' AS DateTime),
N'ADMIN', N'MANAGER', N'DELHI', NULL)
INSERT [dbo].[EMPLOYEES] VALUES (12, N'VIVEL', 89000, 2000, CAST(N'2017-01-20T00:00:00.000' AS
DateTime), N'SECURITY', N'MANAGER', N'KOLKATTA', NULL)
INSERT [dbo].[EMPLOYEES]  VALUES (13, N'DILLIP', 47000, 400, CAST(N'2017-09-25T00:00:00.000' AS DateTime),
N'IT', N'TRAINER', N'SURESH', N'PUNE')
SET IDENTITY_INSERT [dbo].[EMPLOYEES] OFF
GO
```

THE ABOVE SCRIPTS WILL CREATE A TABLE LIKE THIS

EMPID	EMPNAME	SALARY	TAX	DOJ	DEPT	DESG	DEPT MANAGERNAME	DEPTLOCATION
1	AKASH	70000	2000	20-05-2017	IT	SOFTWARE DEVELOPER	SURESH	PUNE
2	AKSHAY	69000	1900	14-07-2018	IT	ARCHITECT	SURESH	PUNE
3	ASHWIN	77000	2100	14-09-2017	IT	ARCHITECT	SURESH	PUNE
4	CHINMAY	88000	2900	10-09-2017	HR	MANAGER	ANITA	MUMBAI
5	DEBAYAN	65000	2100	18-10-2017	HR	EXECUTIVE	ANITA	MUMBAI
6	MANGESH	99000	2900	10-10-2017	HR	MANAGER	ANITA	MUMBAI
7	NIKHIL	56000	2100	18-07-2018	HR	EXECUTIVE	ANITA	MUMBAI
10	SANDEEP	45000	460	25-07-2017	IT	SOFTWARE DEVELOPER	SURESH	PUNE
11	NIRAJ	48000	460	25-07-2017	ADMIN	MANAGER	DELHI	NULL
12	VIVEL	89000	2000	20-01-2019	SECURITY	MANAGER	KOLKATTA	NULL
13	DILLIP	47000	400	25-09-2017	IT	TRAINER	SURESH	PUNE

WRITE ANSWERS (USING SELECT QUERIES) FOR THE BELOW QUESTIONS USING THE ABOVE TABLE

1	WRITE A QUERY TO FIND THE details of employee having the 3rd Higher Salary from the table
2	WRITE A QUERY TO FIND THE employee having the second lowest salary from the table.
3	WRITE A QUERY TO FIND THE details of employees having the 5th & 6th Higher Salary from the table.
4	Write a query to find the details of employee having the highest salary in each department
5	Write a query to find the details of employee having the 3rd highest salary in each department
6	Write a query to find the details of employee having the lowest salary in each department
7	Write a query to find the details of employee having 2nd highest salary holder in each department.
8	Write a query to find the number of employees in each department and designation
9	Write a query find the percentage of salary for each employee. (Calculate the total salary and calculate each one percentage)
10	Write a query to calculate the experience of each employee in terms of years and months.

CHAPTER 4: QUESTIONS

What is the difference between RANK() and DENSE_RANK() ?
What is the use of Having Clause ?
What is default sorting order of Order By Clause (ASC or DESC)?
Can we order a table based upon multiple columns?
What is the use of OFFSET clause?

CHAPTER 5: RELATIONAL DATABASE

Lessons in this chapter

- Lesson 1: Relational Database
- Lesson 2: Types of Relationship
- Lesson 3: Normalization
- Lesson 4: Types of Joins

LESSON 1: RELATIONAL DATABASE

WHAT IS RELATIONAL DATABASE

If we store data in single table, then it create duplicate data which take extra space which increases the size of the database.

For Example, if we store country and state data together then there will be duplicate data as shown below : COUNTRYDETAILS TABLE

COUNTRYID	COUNTRY_NAME	CAPITAL	CONTINENT	STATEID	STATE_NAME	STATE_CAPITAL
1	INDIA	DELHI	ASIA	1	MH	MUMBAI
1	INDIA	DELHI	ASIA	2	UP	LUCKNOW
1	INDIA	DELHI	ASIA	1	GUJURAT	AHMEDABAD

So it better to divide in multiple tables to increase efficiency.

Advantages of Relational Database

- Saving of Data
- Consistency of Data

WE CAN BREAK THE STRUCTURE INTO COUNTRY & STATE TABLE AS SHOWN BELOW

Parent table: COUNTRY (NEED TO DEFINE THE PRIMARY KEY)

COUNTRYID	**COUNTRY_NAME**	**COUNTRY_CAP**	**CONTINENT**
1(PRIMARY KEY	**INDIA**	**DELHI**	**ASIA**

The Primary of the Parent table will become the Foreign Key of the child table to create a relationship.

Child table: STATE

STATEID	**STATE_NAME**	**STATE_CAPITAL**	**COUNTRY_ID(FOREIGNKEY)**
1	**MH**	**MUMBAI**	**1**
2	**UP**	**LUCKNOW**	**1**
3	**GUJURAT**	**AHMEDABAD**	**1**

FOREIGN KEY CONSTRAINTS:

SQL server is a relational database management system. Foreign Key is used to create relationship between two tables. We can have relation between two tables. That relation is done by putting a common column between two tables.

A parent child relationship is created between two tables By having the primary key column in the parent table and the same column as Foreign key column in child table.

Rules of Foreign Key

- Only those values which are there in parent table can be inserted in child table.
- The parent record cannot be deleted until and unless all its children have been deleted.

SYNTAX:

Column_name datatype constraint constraint_name FOREIGN KEY references PRIMARY TABLE (column_name).

EXAMPLES: Primary Key & Foreign Key

☑ CREATE THE FOLLOWING TABLE FOR DOING PRACTICALS (VERY IMP)

```
CREATE TABLE COUNTRY
(
COUNTRY_ID INT CONSTRAINT
pk_country PRIMARY KEY,
COUNTRY_NAME NVARCHAR (100),
COUNTRY_CAPITAL NVARCHAR (100)
)
```

Note : We have created a country table with COUNTRY_ID AS PRIMARY KEY

```
INSERT INTO COUNTRY VALUES (1,'INDIA','DELHI'),
('2','CHINA','BEIJING'),
(3,'U.S.A','WASHINGTON')
```

```
CREATE TABLE STATES
(
STATE_ID INT CONSTRAINT pk_state PRIMARY KEY,
STATE_NAME NVARCHAR (100),
STATE_CAPITAL NVARCHAR (100),
COUNTRY_ID INT CONSTRAINT FK_STATE FOREIGN
KEY REFERENCES COUNTRY (COUNTRY_ID)
)
```

Note : We have created a STATES table with COUNTRY_ID AS FOREIGN KEY

```
INSERT INTO STATES VALUES
(1,'MH','MUMBAI',1),(2,'MP','BHOPAL',1),
(3,'UP','LUCKNOW',1),
(4,'KARNATAKA','BANGALORE',1),(5,'ALASKA','JUNEAU',3),(6,'N
EW YORK','ALBANY',3),(7,'BERLIN','BERLIN',NULL)
```

```
Create table CITY
(
CITY_ID INT  PRIMARY KEY,
CITY_NAME NVARCHAR(100),
STATE_ID  INT  CONSTRAINT  FK_CITY  FOREIGN  KEY REFERENCES STATES(STATE_ID)
)
```

Note : We have created a CITY table with STATE_ID AS FOREIGN KEY

```
INSERT INTO CITY VALUES(1,'PUNE',1)
INSERT INTO CITY VALUES(2,'NAGPUR',1)
```

ANOTHER EXAMPLE OF FOREIGN KEY:

CREATE TABLE SCHOOL (**SCHOOLID INT CONSTRAINT PK_SCHOOLID PRIMARY KEY,** SCHOOLNAME NVARCHAR(100), LOCATION NVARCHAR(100), WEBSITE NVARCHAR(100)) Insert into SCHOOL values(1,'JOSEPH SCHOOL','PUNE','www.joseph.com') DELETE FROM SCHOOL WHERE SCHOOLID=1	CREATE TABLE CLASS (CLASSID INT CONSTRAINT PK_CLASSID PRIMARY KEY, CLASSNAME NVARCHAR(100), **SCHOOLID INT CONSTRAINT FK_SCHOOLID FOREIGN KEY REFERENCES SCHOOL(SCHOOLID)**) Insert into CLASS values(1,'CLASS-I',NULL) Insert into CLASS values(2,'CLASS-II',1)

ON DELETE CASCADE

ON DELETE CASCADE clause which tells SQL Server to delete the corresponding records in the child table when the data in the parent table is deleted. **A foreign key with cascade delete means that if a record in the parent table is deleted, then the corresponding records in the child table will automatically be deleted.** This is called a cascade delete in SQL Server.A foreign key with cascade delete can be created using either a CREATE TABLE statement or an ALTER TABLE statement.

Syntax : Create Foreign key with cascade delete - Using CREATE TABLE statement

```
CREATE TABLE child_table
(
  column1 datatype [ NULL | NOT NULL ],
  column2 datatype [ NULL | NOT NULL ],
  ...
  CONSTRAINT fk_name  FOREIGN KEY (child_col1, child_col2, ... child_col_n)
    REFERENCES parent_table (parent_col1, parent_col2, ... parent_col_n)
    ON DELETE CASCADE
    [ ON UPDATE { NO ACTION | CASCADE | SET NULL | SET DEFAULT } ]
);
```

☑ EXAMPLES OF ON DELETE CASCADE

CREATE TABLE Countries (CountryID INT PRIMARY KEY, CountryName VARCHAR(50), CountryCode VARCHAR(3))	CREATE TABLE StatesDetails (StateID INT PRIMARY KEY, StateName VARCHAR(50), StateCode VARCHAR(3), CountryID INT)

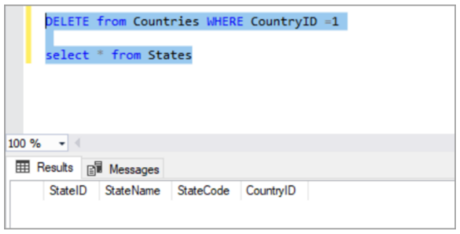
ALTER TABLE StatesDetails ADD CONSTRAINT [FK_States_Countries] FOREIGN KEY([CountryID]) REFERENCES Countries ([CountryID]) ON DELETE CASCADE

Insert some sample data using below T-SQL script.

```
INSERT INTO Countries VALUES (1,'United States','USA')
INSERT INTO Countries VALUES (2,'United Kingdom','UK')

INSERT INTO StatesDetails VALUES (1,'Texas','TX',1)
INSERT INTO StatesDetails VALUES (2,'Arizona','AZ',1)
```

Now I deleted a row in the parent table with CountryID =1 which also deletes the rows in the child table which has CountryID =1.

```
DELETE FROM COUNTRIES WHERE COUNTRYID = 1
```



```
DELETE from Countries WHERE CountryID =1

select * from States
```

```
StateID  StateName  StateCode  CountryID
```

When we create a foreign key using this option, it deletes the referencing rows
in the child table when the referenced row is deleted in the parent table which has a primary key.

FOREIGN KEY WITH CASCADE UPDATE IN SQL SERVER

When we create a foreign key using UPDATE CASCADE the referencing rows are updated
in the child table when the referenced row is updated in the parent table which has a primary key.

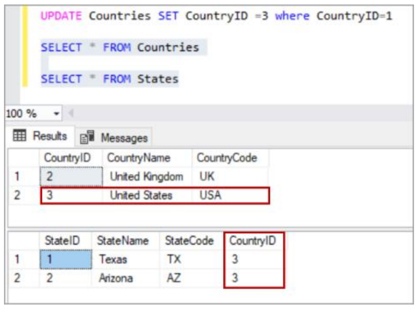
Following is the T-SQL script which creates a foreign key with cascade as UPDATE and DELETE rules.

```
ALTER TABLE StatesDetails  ADD CONSTRAINT FK_States_Countries FOREIGN KEY(CountryID)
REFERENCES Countries (CountryID) ON UPDATE CASCADE ON DELETE CASCADE
ALTER TABLE [dbo].[States] CHECK CONSTRAINT [FK_States_Countries]
```

EXAMPLES OF ON UPDATE CASCADE

Now update CountryID in the Countries for a row which also updates the referencing rows in the child table States.

```
UPDATE Countries SET CountryID =3 where CountryID=1
```



```
UPDATE Countries SET CountryID =3 where CountryID=1

SELECT * FROM Countries

SELECT * FROM States
```

```
   CountryID  CountryName     CountryCode
1  2          United Kingdom  UK
2  3          United States   USA
```

```
   StateID  StateName  StateCode  CountryID
1  1        Texas      TX         3
2  2        Arizona    AZ         3
```

Relationship Terminology: The term ***relation*** is sometimes used to refer to a table in a relational database. However, it is more often used to describe the relationships that exist between the tables in a relational database.

A ***relationship*** between two database tables presupposes that one of them has a foreign key that references the primary key of another table.

Database entity—strictly speaking — is a person, place, thing, object, or any item about which data is stored in the database. However, the term is usually used to refer to the database table as tables are, in fact, the physical implementation of entities.

An ***entity-relationship diagram***, also known as ERD, ER diagram, or ER model, comprises a graphical representation of how entities relate to each other within a database.
ER models are widely used in database design as they are fairly abstract and are easy to view and analyze.
How Data Integrity is implemented by database constraints.
IMP NOTES

Data integrity type	**Enforced by database constraint**
Row integrity	• Primary key constraint • Unique constraint
Column integrity	• Foreign key constraint • Check constraint • Default constraint • Data type constraint
Referential Integrity	• Foreign key constraint
User-defined integrity	• Check constraint

LESSON 2: TYPES OF RELATIONSHIP

There are 3 main types of relationship in a database:

- **ONE-TO-ONE**
- **ONE-TO-MANY**
- **MANY-TO-MANY.**

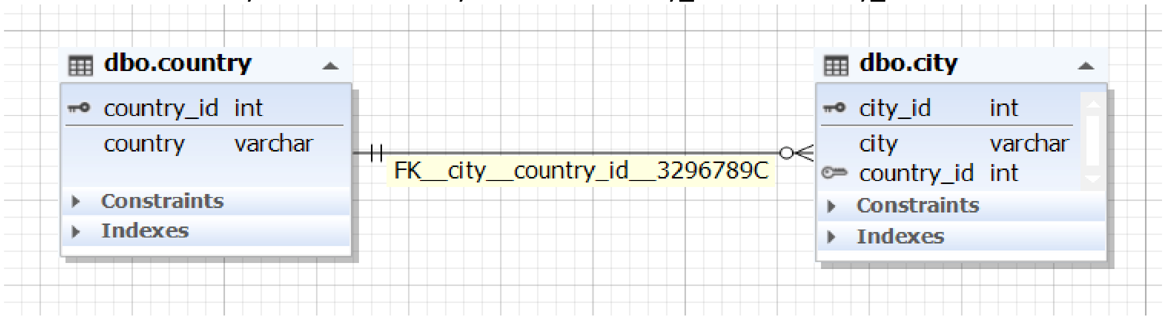
However, you may also encounter references to a many-to-one relationship which, in fact, is a special case of a one-to-many relationship and self-referencing relationship which occurs when only one table is involved.

One-to-many relationship

Let's start with a one-to-many relationship as it is the most commonly used type. So, what is one-to-many relationship in SQL? A one-to-many relationship occurs when one record in table 1 is related to one or more records in table 2. However, one record in table 2 cannot be related to more than one record in table 1

Let's query two SQL tables having one-to-many relationship.

SELECT   * FROM country c INNER JOIN city c1   ON c.country_id = c1.country_id



Example of one-to-many relation in SQL Server

How to implement one-to-many relationships when designing a database:

1. Create two tables (table 1 and table 2) with their own primary keys.
2. Add a foreign key on a column in table 1 based on the primary key of table 2. This will mean that table 1 can have one or more records related to a single record in table 2.

Step 1

CREATE TABLE dbo.city (city_id int IDENTITY PRIMARY KEY, city varchar(50) NOT NULL,
 country_id int NOT NULL)

CREATE TABLE dbo.country (country_id int IDENTITY PRIMARY KEY, country varchar(50) NOT NULL)

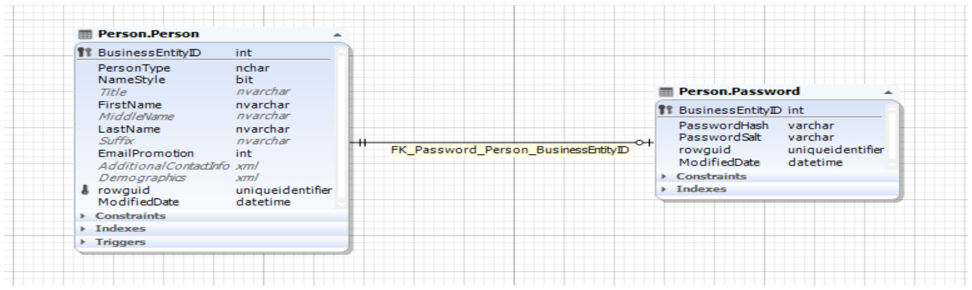
Step 2

ALTER TABLE dbo.city WITH NOCHECK ADD FOREIGN KEY (country_id) REFERENCES dbo.country (country_id)

One-to-one relationship

A one-to-one relationship in a database occurs when each row in table 1 has only one related row in table 2. For example, a department may have only one head manager, a husband — only one wife, an employee — one company car, etc.

One-to-one relationship example in SQL:



Person.Person	
BusinessEntityID	int
PersonType	nchar
NameStyle	bit
Title	*nvarchar*
FirstName	nvarchar
MiddleName	*nvarchar*
LastName	nvarchar
Suffix	*nvarchar*
EmailPromotion	int
AdditionalContactInfo	*xml*
Demographics	*xml*
rowguid	uniqueidentifier
ModifiedDate	datetime

Person.Password	
BusinessEntityID	int
PasswordHash	varchar
PasswordSalt	varchar
rowguid	uniqueidentifier
ModifiedDate	datetime

Implementing one-to-one relation

How to create one-to-one relationship in SQL Server? For this, you will need to create two tables, then define a simple primary foreign key relationship between them, and set the foreign key column to be unique.

CREATE TABLE Employee (ID int PRIMARY KEY, Name VARCHAR(50));

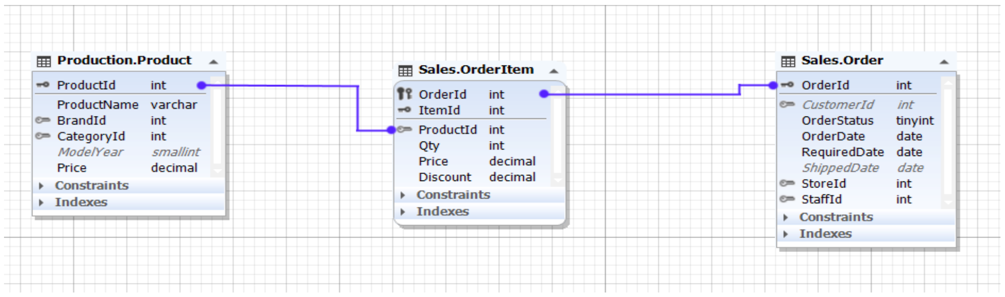
CREATE TABLE Salary (EmployeeID int UNIQUE NOT NULL,   SalaryAmount int);

ALTER TABLE Salary ADD CONSTRAINT FK_Salary_Employee FOREIGN KEY (EmployeeID) REFERENCES Employee (ID);

Many-to-many relationship

A many-to-many relationship occurs when multiple records in one table are related to multiple records in another table. For example, products and suppliers: one supplier may deliver one or many products and at the same time, the company may order one product from one or many suppliers.

A many-to-many relationship example in SQL:



The relationship between the Product entity and Order entity is many-to-many, as one product may be in many orders and many orders may contain the same product.

Example of creating many-to-many relation in SQL

Relational databases don't support direct many-to-many relationships between two tables. Then, how to implement many-to-many relationships in SQL? To create a many-to-many relationship in a database, you'll need to create a third table to connect the other two. This new table (also known as a *linking*, *joining*, *bridging*, or *junction* table) will contain the primary key columns of the two tables you want to relate and will serve as an intermediate table between them.

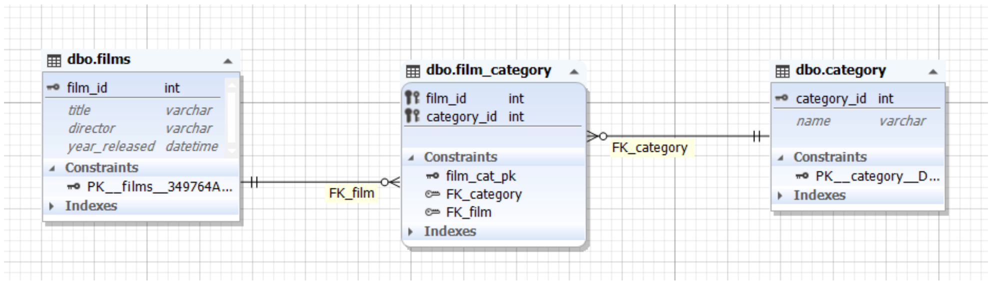
Let's consider the following example of how to create many-to-many relationship in SQL. Suppose, we want to establish a many-to-many relationship between two tables: *films* and *category*. First, we create the two tables.

```
CREATE TABLE films (   film_id INT PRIMARY KEY  ,title VARCHAR(50)  ,director VARCHAR(50)
 ,year_released DATETIME  );

CREATE TABLE category (   category_id INT PRIMARY KEY  ,name VARCHAR(50) );
```

Next, we create a junction table *film_category* that will map these two tables together by referencing the primary keys of both tables.

```
CREATE TABLE film_category (  film_id INT ,category_id INT
 ,CONSTRAINT film_cat_pk PRIMARY KEY (film_id, category_id)
 ,CONSTRAINT FK_film  FOREIGN KEY (film_id) REFERENCES films (film_id)
 ,CONSTRAINT FK_category   FOREIGN KEY (category_id) REFERENCES category (category_id)
);
```

A many-to-many relationship between the films and category tables has been successfully established.



LESSON 3: NORMALIZATION

What is Normalization?
Normalization is a process of **eliminating redundant** data and **storing** the **related information** in a table.

1. Eliminating redundant data.
2. Faster update
3. Improve performance
4. Performance in indexes

☑ Normalization forms

1. First Normal Form (1NF)

If a Table is said to be 1NF then it should satisfy following rules.

- Each cell must have one value
- Eliminating Duplicate Columns
- Create a separate table for group of related data and each row must be identify by primary key.

That means each cell must have single value and each row should be uniquely identified by Primary key

Name	**Department**	**Phone Number**
Rajesh	Computer	3452342,1234563,2345612
Suresh	Electronics	2398521,2323177,5302994
Praba	Civil	3958218

In the above we can see the duplicate columns phone numbers have more than one value, we have to eliminate that and create a group of related data with unique row identification by specifying a primary key for the table

Rule 1. By applying above rule each cell must have one value above table changes like below

Name	**Department**	**Phone Number**	**Phone Number**	**Phone Number**
Rajesh	Computer	3452342	1234563	2345612
Suresh	Electronics	2398521	2323177	5302994
Praha	Civil	3958218		

Rule 2 &3. By applying second rule and third rule no more duplicate columns and each row must be unique is applied to above table.

Id	**Name**	**Department**	**Phone Number**
1	Rajesh	Computer	3452342
2	Rajesh	Computer	1234563
3	Rajesh	Computer	2345612
4	Suresh	Electronics	2398521
5	Suresh	Electronics	2323177
6	Suresh	Electronics	5302994
7	Praha	Civil	3958218

2.Second Normal Form (2NF)

The Table must be in second normal form, then it should satisfy the following rules.

- **It should satisfy first normal form**
- **Separate the particular columns ,values are duplicated in each row should be place in separate table**
- **Create the relationship between the tables**

From the above table we can see the column name and department are repeated in each row, this two columns can be maintained in another table and make a relationship between these two tables

EmpId	**Name**	**Department**	**DeptLocation**
1	Rajesh	Computer	
2	Suresh	Electronics	
3	Praha	Civil	
4	Anil	Computer	

Id	**EmpId**	**PhoneNumber**
1	1	3452342
2	1	1234563
3	1	2345612
4	2	2398521
5	2	2323177
6	2	5302994
7	3	3958218

In the above table Empid is played as Primary key for the first table and foreign key for the second table.

3. Third Normal Form (3NF)

The table must be in 3NF, if it is satisfying the following rules

- Must be in 2NF
- Separate the columns that are not dependent upon the primary key of the table.

Product	**Price**	**Tax**
LED	23000	20%
AC	15000	10%
Fridge	12000	15%

From the above table you can see that Tax Column is not dependent on Product Primary key column, it is dependent on Price so we separate that in to two different table.

Product	**Price**
LED	23000
AC	15000
Fridge	12000

Price	**Tax**
23000	20%
15000	10%
12000	15%

4. Fourth Normal Form (4NF)

- It should be in 3NF
- The non key columns should be dependent on full primary key instead of partial key, if then separate it.

From the following table "EmployeeName" Non-Key column not dependent on full primary key "ManagerId, EmployeeId, TaskID" it depends upon the EmployeeId Partial Key so it can be separated.

ManagerId	**EmployeeId**	**TaskID**	**EmployeeName**
M1	E1	T1	Rajesh
M2	E1	T1	Rajesh

ManagerId	**EmployeeId**	**TaskID**
M1	E1	T1
M2	E1	T1

EmployeeId	**EmployeeName**
E1	Rajesh

LESSON 4: TYPES OF JOINS

Joins are used to get data from multiple tables. SQL server provides the following types of join these are

- Inner join
- Left outer join
- Right outer join
- Full outer join
- Self-join
- Cross join

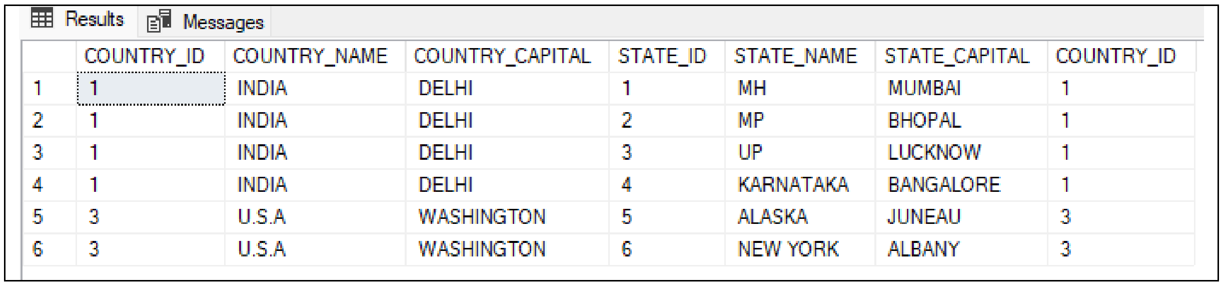
A join condition defines the way two tables are related in a query by:

- Specifying the column from each table to be used for the join. A typical join condition specifies a foreign key from one table and its associated key in the other table.
- Specifying a logical operator (for example, = or <>,) to be used in comparing values from the columns.

In this lesion we will discuss about the types of joins.

INNER JOIN: **I**t gets the common records from both the tables.

Syntax: Select * from table1 **inner join** table2 **on** table1.colname (p.k) = table2.colname (f.k)
Example: Select * from country inner join states on country.country_id = states.country_id
Output

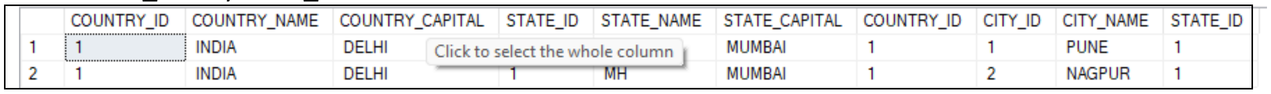


Results | Messages

	COUNTRY_ID	COUNTRY_NAME	COUNTRY_CAPITAL	STATE_ID	STATE_NAME	STATE_CAPITAL	COUNTRY_ID
1	1	INDIA	DELHI	1	MH	MUMBAI	1
2	1	INDIA	DELHI	2	MP	BHOPAL	1
3	1	INDIA	DELHI	3	UP	LUCKNOW	1
4	1	INDIA	DELHI	4	KARNATAKA	BANGALORE	1
5	3	U.S.A	WASHINGTON	5	ALASKA	JUNEAU	3
6	3	U.S.A	WASHINGTON	6	NEW YORK	ALBANY	3

Syntax: For more than two tables

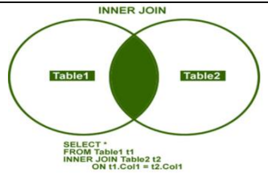
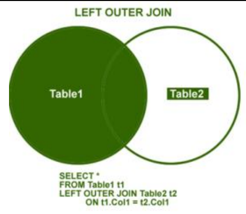
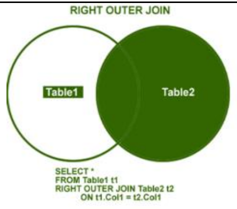
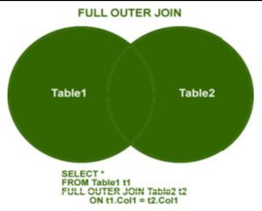
Select * from table1 **inner join** table2 on table1.colname (p.k) = table2.colname (f.k) **inner join** table3 on table2.colname = table3.colname
Example:
Select * from country inner join states on country.country_id = states.country_id inner join city on states.state_id=city.state_id



	COUNTRY_ID	COUNTRY_NAME	COUNTRY_CAPITAL	STATE_ID	STATE_NAME	STATE_CAPITAL	COUNTRY_ID	CITY_ID	CITY_NAME	STATE_ID
1	1	INDIA	DELHI	[illegible]	[illegible]	MUMBAI	1	1	PUNE	1
2	1	INDIA	DELHI	1	MH	MUMBAI	1	2	NAGPUR	1

Click to select the whole column

TYPES OF JOINS

INNER	LEFT OUTER	RIGHT OUTER	FULL OUTER	CROSS	SELF
INNER JOIN gets the Common records from both the tables.	**LEFT OUTER JOIN** gets the Common records from both the tables + all records from left tables.	**RIGHT OUTER JOIN** gets the Common records from both the tables + all records from right tables.	**FULL OUTER JOIN** gets the Common records from both the tables + all records from left tables and all records from right table.	In **Cross** join Each record of table1 joins all the records of table1	**self-Join :** Joins the same table with itself.
 INNER JOIN Table1 Table2 SELECT * FROM Table1 t1 INNER JOIN Table2 t2 ON t1.Col1 = t2.Col1	 LEFT OUTER JOIN Table1 Table2 SELECT * FROM Table1 t1 LEFT OUTER JOIN Table2 t2 ON t1.Col1 = t2.Col1	 RIGHT OUTER JOIN Table1 Table2 SELECT * FROM Table1 t1 RIGHT OUTER JOIN Table2 t2 ON t1.Col1 = t2.Col1	 FULL OUTER JOIN Table1 Table2 SELECT * FROM Table1 t1 FULL OUTER JOIN Table2 t2 ON t1.Col1 = t2.Col1		
SELECT * FROM TABLE1(left) INNER JOIN TABLE2(right) ON TABLE1.COLNAME(P.k) = TABLE2.COLNAME(f.K)	**SELECT * FROM TABLE1(left) LEFT OUTER JOIN TABLE2(right) ON TABLE1.COLNAME(P.k) = TABLE2.COLNAME(f.K)**	**SELECT * FROM TABLE1(left) RIGHT OUTER JOIN TABLE2(right) ON TABLE1.COLNAME(P.k) = TABLE2.COLNAME(f.K)**	**SELECT * FROM TABLE1(left) FULL OUTER JOIN TABLE2(right) ON TABLE1.COLNAME(P.k) = TABLE2.COLNAME(f.K)**	**SELECT * FROM table1 CROSS JOIN table2**	**SELECT * FROM table1 t1 Join table1 t2 On t1.col=t2.col**

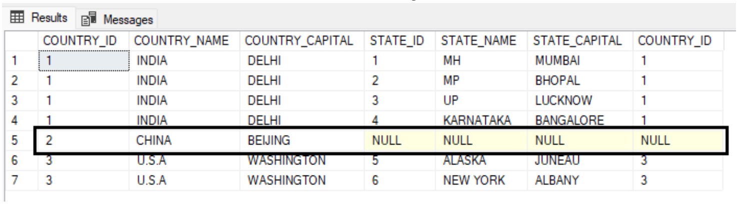
EXAMPLES OF JOINS

L EFT OUTER JOIN

It gets the Common records from both the tables + all records from left tables.

Syntax:
Select * from Table1 (Left) **LEFT OUTER JOIN** Table2 **ON** Tabl1.Column_name (P.K.) = Table2.Column_name (F.K.)

Example:
Select * from COUNTRY **left outer join** STATES **on** COUNTRY.COUNTRY_ID = STATES.COUNTRY_ID



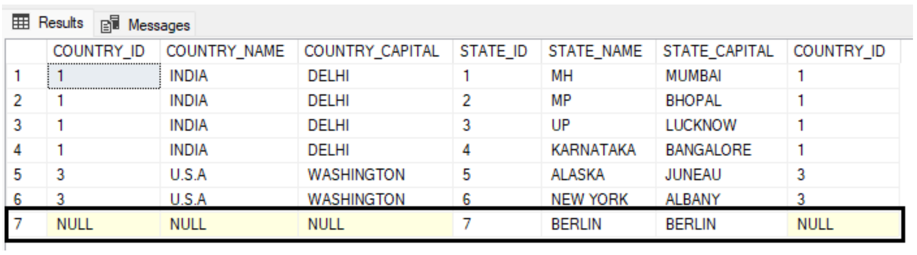
	COUNTRY_ID	COUNTRY_NAME	COUNTRY_CAPITAL	STATE_ID	STATE_NAME	STATE_CAPITAL	COUNTRY_ID
1	1	INDIA	DELHI	1	MH	MUMBAI	1
2	1	INDIA	DELHI	2	MP	BHOPAL	1
3	1	INDIA	DELHI	3	UP	LUCKNOW	1
4	1	INDIA	DELHI	4	KARNATAKA	BANGALORE	1
5	2	CHINA	BEIJING	NULL	NULL	NULL	NULL
6	3	U.S.A	WASHINGTON	5	ALASKA	JUNEAU	3
7	3	U.S.A	WASHINGTON	6	NEW YORK	ALBANY	3

R IGHT OUTER JOIN

It gets the Common records from both the tables + all records from right tables.

Syntax: Select * from Table1 (Left) **RIGHT OUTER JOIN** Table2 **ON** Tabl1.Column_name (P.K.) = Table2.Column_name (F.K.)

Example: Select * from COUNTRY **right outer join** STATES **on** COUNTRY.COUNTRY_ID = STATES.COUNTRY_ID



	COUNTRY_ID	COUNTRY_NAME	COUNTRY_CAPITAL	STATE_ID	STATE_NAME	STATE_CAPITAL	COUNTRY_ID
1	1	INDIA	DELHI	1	MH	MUMBAI	1
2	1	INDIA	DELHI	2	MP	BHOPAL	1
3	1	INDIA	DELHI	3	UP	LUCKNOW	1
4	1	INDIA	DELHI	4	KARNATAKA	BANGALORE	1
5	3	U.S.A	WASHINGTON	5	ALASKA	JUNEAU	3
6	3	U.S.A	WASHINGTON	6	NEW YORK	ALBANY	3
7	NULL	NULL	NULL	7	BERLIN	BERLIN	NULL

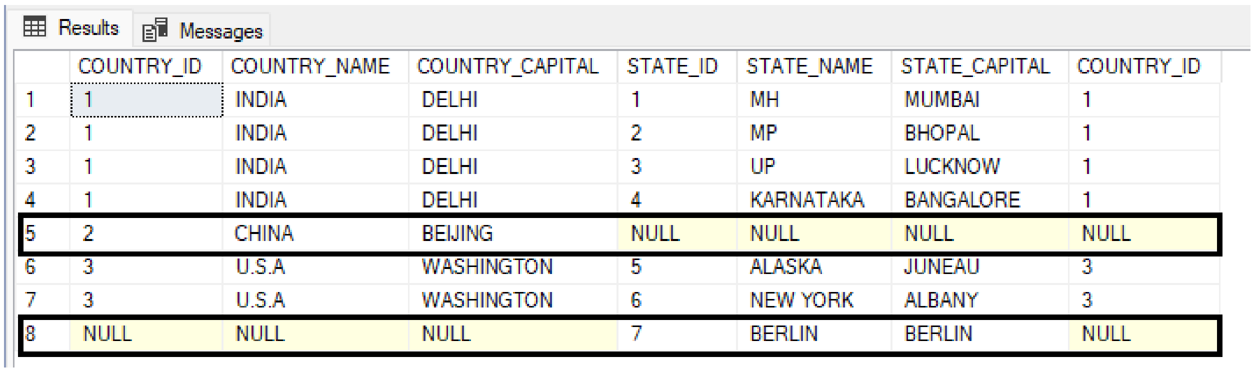
F ULL OUTER JOIN

It gets the Common records from both the tables + all records from left tables and all records from right table.

Syntax: Select * from Table1 (Left) **FULL OUTER JOIN** Table2 **ON** Tabl1.Column_name (P.K.) = Table2.Column_name (F.K.)

Example:

Select * from COUNTRY **full outer join** STATES **on** COUNTRY.COUNTRY_ID = STATES.COUNTRY_ID



Results | Messages

	COUNTRY_ID	COUNTRY_NAME	COUNTRY_CAPITAL	STATE_ID	STATE_NAME	STATE_CAPITAL	COUNTRY_ID
1	1	INDIA	DELHI	1	MH	MUMBAI	1
2	1	INDIA	DELHI	2	MP	BHOPAL	1
3	1	INDIA	DELHI	3	UP	LUCKNOW	1
4	1	INDIA	DELHI	4	KARNATAKA	BANGALORE	1
5	2	CHINA	BEIJING	NULL	NULL	NULL	NULL
6	3	U.S.A	WASHINGTON	5	ALASKA	JUNEAU	3
7	3	U.S.A	WASHINGTON	6	NEW YORK	ALBANY	3
8	NULL	NULL	NULL	7	BERLIN	BERLIN	NULL

C ROSS JOIN

In Cross join each record of table1 joins all the records of table1. This join is a Cartesian join that does not necessitate any condition to join. The result set contains records that are multiplication of record number from both the tables.

Syntax: Select * from Table1 **CROSS JOIN** Table2

Example: Select * from COUNTRY **cross join** STATE

S ELF JOIN

In self join it joins the same table with itself.

Syntax: Select * from Table1 T1 **JOIN** Table1 T2 **ON** T1.table1=T2.table1

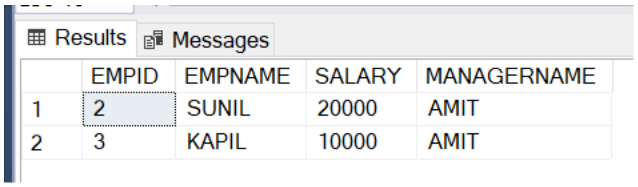
Example of SELF JOIN

Create table empmanagers
(
EMPID INT PRIMARY KEY,
EMPNAME VARCHAR (100),
SALARY FLOAT,
MGRID INT
)
	ALTER TABLE empmanagers ADD FOREIGN KEY (MGRID) REFERENCES EMPMANAGERS (EMPID)
	
	INSERT INTO EMPMANAGERS VALUES (1,'AMIT', 30000, NULL)
	INSERT INTO EMPMANAGERS VALUES (2,'SUNIL', 20000, 1)
	INSERT INTO EMPMANAGERS VALUES (3,'KAPIL', 10000, 1)

QUERY TO JOIN ABOVE THREE TABLE USING DIFFERENT TYPES OF JOINT

WRITE A QUERY TO FIND THE NAME OF THE MANAGER FOR THE EMPLOYEE

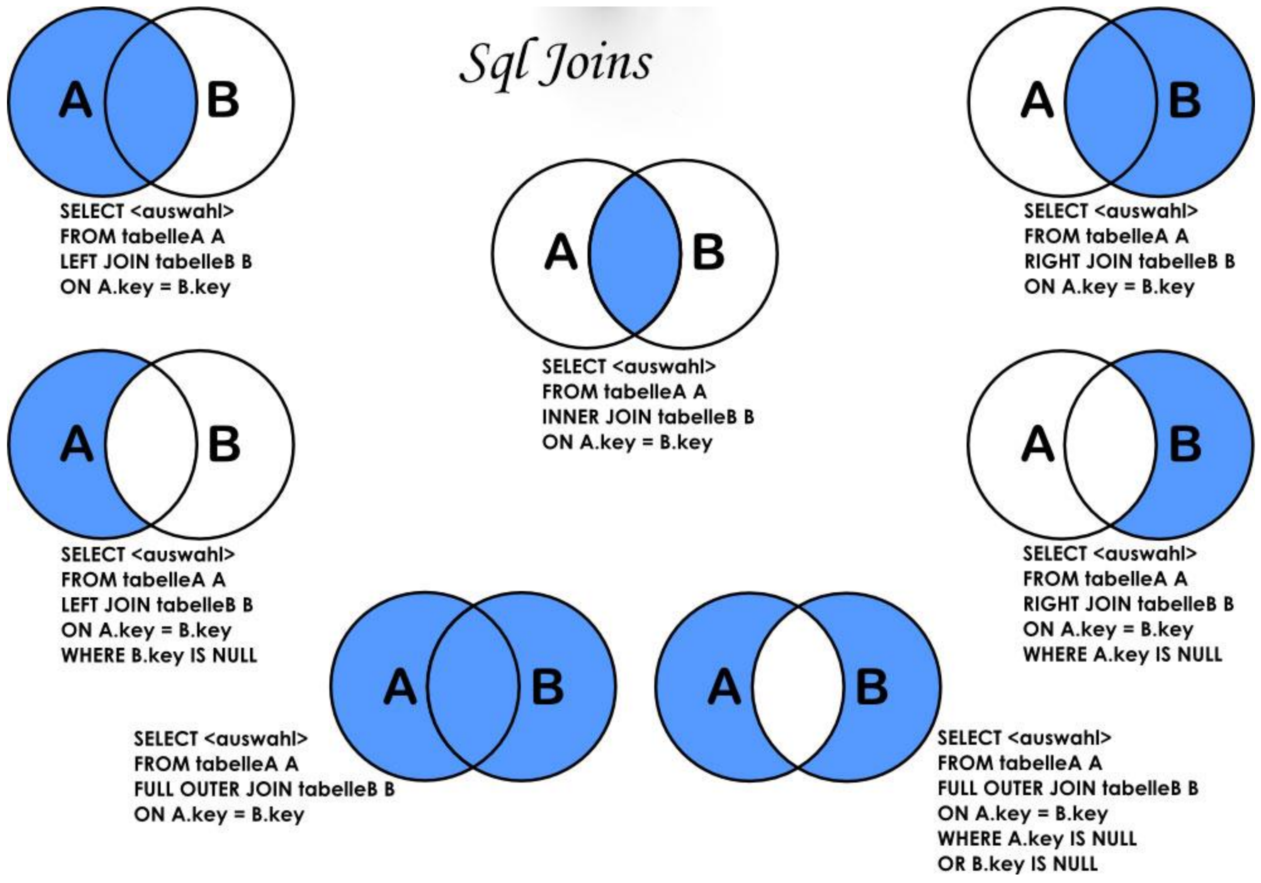
SELECT E1.EMPID, E1.EMPNAME, E1.SALARY, E2.EMPNAME AS MANAGERNAME FROM empmanagers E1 INNER JOIN empmanagers E2 ON E1.MGRID= E2.EMPID



Results | Messages

	EMPID	EMPNAME	SALARY	MANAGERNAME
1	2	SUNIL	20000	AMIT
2	3	KAPIL	10000	AMIT

CHAPTER 5: SUMMARY / CHEAT SHEET



```
SELECT <auswahl>
FROM tabelleA A
LEFT JOIN tabelleB B
ON A.key = B.key
```

```
SELECT <auswahl>
FROM tabelleA A
INNER JOIN tabelleB B
ON A.key = B.key
```

```
SELECT <auswahl>
FROM tabelleA A
RIGHT JOIN tabelleB B
ON A.key = B.key
```

```
SELECT <auswahl>
FROM tabelleA A
LEFT JOIN tabelleB B
ON A.key = B.key
WHERE B.key IS NULL
```

```
SELECT <auswahl>
FROM tabelleA A
RIGHT JOIN tabelleB B
ON A.key = B.key
WHERE A.key IS NULL
```

```
SELECT <auswahl>
FROM tabelleA A
FULL OUTER JOIN tabelleB B
ON A.key = B.key
```

```
SELECT <auswahl>
FROM tabelleA A
FULL OUTER JOIN tabelleB B
ON A.key = B.key
WHERE A.key IS NULL
OR B.key IS NULL
```

CHAPTER 5: EXERCISE (ASSIGNMENTS)

CREATE A TABLE FOR COMPANY

COLUMN_NAME	DATATYPE	CONSTRAINT
COMPANYID	INT	PRIMARY KEY, IDENTITY(1,1)
COMPANYNAME	VARCHAR(100)	
WEBSITE	VARCHAR(100)	
EMAIL	VARCHAR(100)	

ADD TWO COMPANY RECORDS TO THE TABLE using INSERT STATEMENTS

CREATE A TABLE FOR DEPT

COLUMN_NAME	DATATYPE	CONSTRAINT
DEPTID	INT	PRIMARY KEY, IDENTITY(1,1)
DEPTNAME	VARCHAR(100)	
DEPTLOCATION	VARCHAR(100)	

ADD FIVE DEPT RECORDS TO THE TABLE using INSERT STATEMENTS

CREATE A TABLE FOR DESG

COLUMN_NAME	DATATYPE	CONSTRAINT
DESGID	INT	PRIMARY KEY
DESGNAME	VARCHAR(100)	
CREATEDDATE	DATETIME	DEFAULT - GETDATE()

ADD FIVE DESG RECORDS TO THE TABLE using INSERT STATEMENTS

CREATE A TABLE FOR EMPLOYEE

COLUMN_NAME	DATATYPE	CONSTRAINT
EMPID	INT	PRIMARY KEY
EMPNAME	VARCHAR(100)	UNIQUE
BASICSALARY	FLOAT	
INCENTIVE	FLOAT	
BONUS	FLOAT	
DOB	DATETIME	
DOJ	DATETIME	DEFAULT GETDATE()
GENDER	VARCHAR(100)	
EMAIL	VARCHAR(100)	UNIQUE
MOBILE	VARCHAR(10)	
CREATEDDATE	DATETIME	DEFAULT GETDATE()
COMPANYID	INT	F.K
DEPTID	INT	F.K
DESGID	INT	F.K

ADD 10 EMPLOYEES RECORDS TO THE TABLE USING INSERT COMMAND

WRITE A QUERY TO DISPLAY THE TOTALSALARY FOR EACH DEPTNAME.

DEPTNAME	TOTALSALARY

WRITE A QUERY TO DISPLAY THE NOOFEMPLOYEES FOR EACH COMPANYNAME.'

COMPANYNAME	NOOFEMPLOYEES

WRITE A QUERY TO FIND HIGHEST SALARY HOLDER IN EMPLOYEE TABLE

WRITE A QUERY TO FIND 3RD HIGHEST SALARY HOLDER IN EMPLOYEE TABLE

WRITE A QUERY TO FIND FIFTH HIGHEST SALARY HOLDER IN EMPLOYEE TABLE

WRITE A QUERY TO DISPLAY THE DESGNAME,NOOFEMPS,TOTALSALARY

DESGNAME	NOOFEMPS	TOTALSALARY

CREATE A TABLE COUNTRY AND ADD THE VALUES AS SHOWN BELOW USING INSERT COMMAND

COUNTRYID(P.K)	COUNTRYNAME(UNIQUE)	COUNTRYCAPITAL
1	INDIA	DELHI
2	CHINA	BEIJING
3	U.S.A	Washington

CREATE A TABLE STATE AND ADD THE VALUES AS SHOWN BELOW USING INSERT COMMAND

STATEID(P.K)	STATENAME	STATECAPITAL	COUNTRYID
1	MH	MUMBAI	1
2	MP	BHOPAL	1
3	UP	LUCKNOW	1
4	KARNATAKA	BANGALORE	1
5	Alaska	Juneau	3
6	New York	Albany	3
7	Berlin	Berlin	NULL

CREATE A TABLE CITY AND ADD THE VALUES AS SHOWN BELOW USING INSERT COMMAND

CITYID(P.K)	CITYNAME	STATEID
1	NASHIK	1
2	PUNE	1
3	NAGPUR	1

WRITE THE QUERIES USING THE COUNTRY, STATES AND CITY TABLE

1. WRITE A QUERY TO DO an INNER JOIN BETWEEN COUNTRY AND STATE TABLE.
2. WRITE A QUERY TO DO an LEFT OUTER JOIN BETWEEN COUNTRY AND STATE TABLE.
3. WRITE A QUERY TO DO an RIGHT OUTER JOIN BETWEEN COUNTRY AND STATE TABLE.

4. WRITE A QUERY TO DO an FULL OUTER JOIN BETWEEN COUNTRY AND STATE TABLE.
5. WRITE A QUERY TO DO A CROSS JOIN BETWEEN COUNTRY AND STATE TABLE
6. WRITE A QUERY TO FIND THE NAME OF COUNTRIES WHICH ARE NOT PRESENT IN THE INNER JOIN. (HINT : USE LEFT JOIN EXCEPT INNER JOIN)

WRITE A QUERY TO FIND THE NUMBER OF STATES IN EACH COUNTRY

COUNTRYNAME	NOOFSTATES

SELF JOIN ASSIGNMENT

CREATE A TABLE EMPLOYEE AND ADD THE VALUES AS SHOWN BELOW
IN THE EXAMPLE, THE MANAGERID WILL BE FOREIGN KEY WHICH WILL HAVE REFERENCE TO THE PRIMARY KEY EMPID (PRIMARY KEY)

EMPID (PRIMARY KEY)	EMPNAME	SALARY	TAX	MANAGERID (FOREIGN KEY)
1	SAMIR	98000	1000	NULL
2	YOGESH	88000	2000	1
3	NEERAJ	58000	2000	1
4	KAMLESH	48000	1000	2

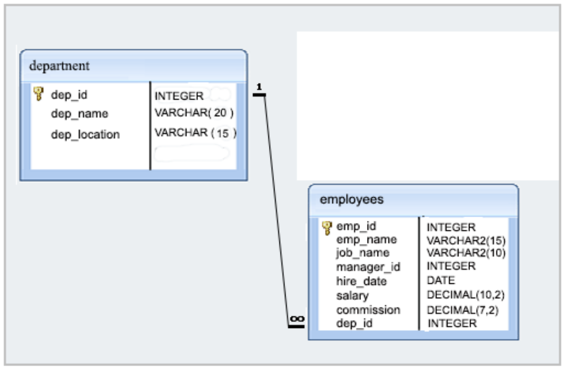
WRITE A QUERY TO FIND THE MANAGERNAME OF THE EMPLOYEES. OUTPUT WILL BE

EMPNAME	EMPSALARY	MANAGERNAME	MANAGER_SALARY

CHAPTER 5: EXERCISE (ASSIGNMENTS)

CREATE THE FOLLOWING TABLE DESCRIBED BELOW AND INSERT THE DATA AS GIVE BELOW.

Structure of employee Database:



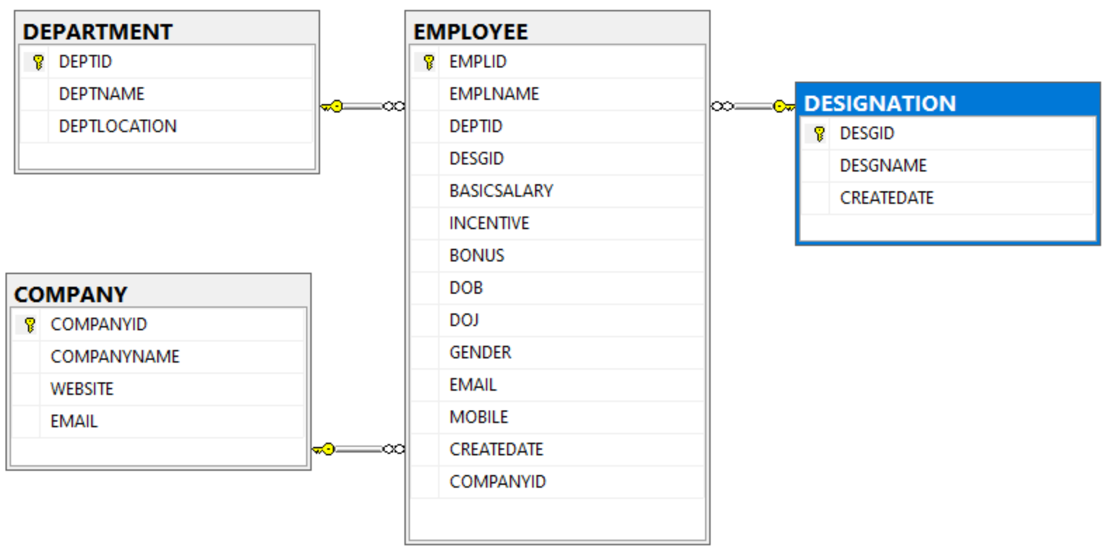
```
DROP TABLE DEPARTMENT
DROP TABLE EMPLOYEES
CREATE TABLE DEPARTMENT
(
DEPT_ID INT PRIMARY KEY,
DEPT_NAME VARCHAR(100),
DEPT_LOCATION VARCHAR(100)
)
INSERT INTO DEPARTMENT VALUES(1,'IT','PUNE')
INSERT INTO DEPARTMENT VALUES(2,'HR','MUMBAI')
INSERT INTO DEPARTMENT VALUES(3,'ADMIN','MUMBAI')
INSERT INTO DEPARTMENT VALUES(4,'SECURITY','PUNE')
INSERT INTO DEPARTMENT VALUES(5,'FINANCE','MUMBAI')
```

```
CREATE TABLE EMPLOYEES
(
EMP_ID INT PRIMARY KEY,
EMP_NAME VARCHAR(100),
JOB_NAME VARCHAR(100),
MANAGER_ID INT FOREIGN KEY REFERENCES EMPLOYEES(EMP_ID),
HIRE_DATE DATETIME,
SALARY DECIMAL(10,2),
COMMISSION DECIMAL(7,2),
DEPT_ID INT FOREIGN KEY REFERENCES DEPARTMENT(DEPT_ID)
)

INSERT INTO EMPLOYEES VALUES (1,'SUNIL','MANAGER',NULL,'10/10/2018',60001,1000,1)
INSERT INTO EMPLOYEES VALUES (2,'KAPIL','MANAGER',NULL,'05/10/2018',80012,1000,2)
INSERT INTO EMPLOYEES VALUES (3,'SAMIR','CLERK',2,'10/10/2018',11000,1000,2)
INSERT INTO EMPLOYEES VALUES (4,'KRUTIKA','CLERK',2,'05/10/2018',12000,1000,2)
INSERT INTO EMPLOYEES VALUES (5,'YASIKA','CLERK',2,'10/10/2019',13000,5000,2)
INSERT INTO EMPLOYEES VALUES (6,'SONAL','CLERK',2,'05/10/2019',14000,6000,2)

INSERT INTO EMPLOYEES VALUES (13,'RAJESH','CLERK',NULL,'10/10/2000',15000,1000,1)
INSERT INTO EMPLOYEES VALUES (14,'UMESH','CLERK',NULL,'05/10/1999',16000,1000,2)
INSERT INTO EMPLOYEES VALUES (15,'YAMIN','CLERK',NULL,'10/10/1998',17000,5000,1)
INSERT INTO EMPLOYEES VALUES (16,'KAMINI','CLERK',NULL,'05/10/1997',18000,6000,2)

INSERT INTO EMPLOYEES VALUES (113,'SUJANAN','CLERK',1,'10/10/2000',12500,1000,1)
INSERT INTO EMPLOYEES VALUES (114,'ANJANA','CLERK',1,'05/10/1999',13500,1000,1)
INSERT INTO EMPLOYEES VALUES (115,'TEJAL','CLERK',1,'10/10/1998',10500,5000,1)
INSERT INTO EMPLOYEES VALUES (116,'AKSHAY','CLERK',1,'05/10/1997',10600,6000,1)

INSERT INTO EMPLOYEES VALUES (213,'SUJANAN','CLERK',2,'10/10/2000',12001,1000,2)
INSERT INTO EMPLOYEES VALUES (214,'ANJANA','CLERK',2,'05/10/1999',13001,1000,2)
INSERT INTO EMPLOYEES VALUES (215,'TEJAL','CLERK',2,'10/10/1998',10001,5000,2)
INSERT INTO EMPLOYEES VALUES (216,'AKSHAY','CLERK',2,'05/10/1997',10045,6000,2)
INSERT INTO EMPLOYEES VALUES (313,'JAYESH','ANALYST',2,'10/10/2000',12011,1000,2)
INSERT INTO EMPLOYEES VALUES (314,'TAJESH','ANALYST',2,'05/10/1999',13011,1000,2)
INSERT INTO EMPLOYEES VALUES (315,'MANGESH','ANALYST',2,'10/10/1998',10451,5000,2)
INSERT INTO EMPLOYEES VALUES (316,'KALPESH','ANALYST',2,'05/10/1997',10565,6000,2)

SELECT * FROM EMPLOYEES
```

1. Write a query in SQL to display the unique designations for the employees.
2. Write a query in SQL to list the employees who joined before 1991
3. Write a query in SQL to display the average salaries of all the employees who works as ANALYST
4. Write a query in SQL to display all the details of the employees whose commission is more than their salary.
5. Write a query in SQL to list the employees who are either CLERK or MANAGER.
6. Write a query in SQL to list the employees who are working either MANAGER or ANALYST with a salary range between 2000 to 5000 without any commission
7. Write a query to list average salaries per department, highest salary per department.
8. Write a query to find departments having more than 20 employees.
9. query in SQL to list the employees who joined
   ---before 2001
   -- in month of Feb
10. find employees having salary greater than avg salary of their dept and second letter of their name has 'a'.
11. Find Highest Salary Holder Employee details of Each Dept in a single query
12. Find Second Highest Salary Holder Employee details of Each Dept in a single query
13. Find Top 3 Highest Salary Holder of Each Dept in a single query
14. Find Top 5 Highest Salary of Each Dept in a single query
15. Find Bottom 3 Highest Salary Holder of Each Dept in a single query

CHAPTER 5: EXERCISE (ASSIGNMENTS)

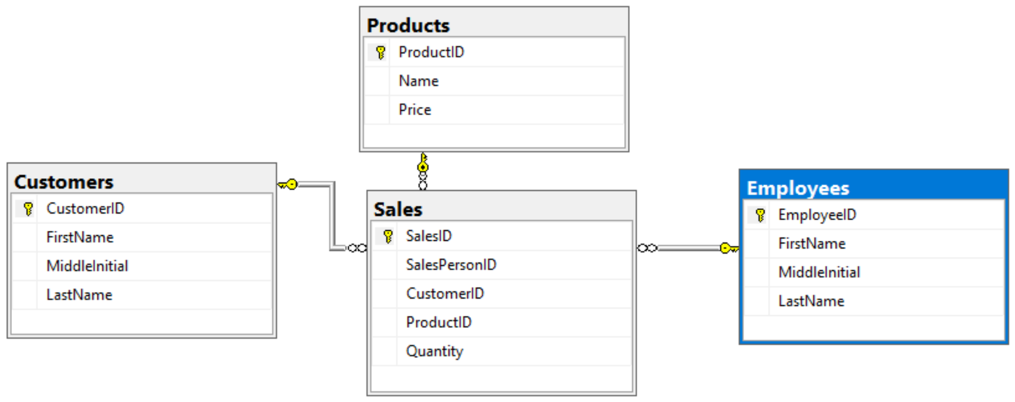
CREATE A COMPANY MANAGEMENT SYSTEM DATABASE SIMILAR TO THIS.



INSERT VALUES INTO ALL THE TABLES OF THE DATABASE DESIGNED

WRITE A QUERY TO JOIN ALL THE TABLES AND SHOW THE RESULTS
WRITE A QUERY TO COPY THE DATA OF ALL TABLES GENERATED USING JOINS INTO A NEW TABLE

CREATE A SALES MANAGEMENT SYSTEM DATABASE SIMILAR TO THIS.



After Creating the Data Structures.

Add some Products into the tables.
Then add some Customers into the tables
Then add some employees into the tables
Then add sales data into the sales table.

CHAPTER 5: QUESTIONS

Can we put multiple foreign key in a single table ?
What is the advantage of Relational Database Model ?
What is Normalization ?
What is the difference between inner join and left outer join ?
What is Self Join ?

CHAPTER 6

CHAPTER 6: ADVANCE OPERATOR IN SQL SERVER

Lesions in this chapter

- Lesion 1 : Union, Except & intersect
- Lesson 2: Pivot & UnPivot

LESSON 1: UNION, EXCEPT & INTERSECT

UNION & UNION ALL

SQL Server UNION is one of the set operations that allows you to combine results of two SELECT statements into a single result set which includes all the rows that belongs to the SELECT statements in the union.

The following illustrates the syntax of the SQL Server UNION: query_1 UNION query_2

The following are requirements for the queries in the syntax above:

- The number and the order of the columns must be the same in both queries.
- The data types of the corresponding columns must be the same or compatible.

UNION vs. UNION ALL :By default, the UNION operator removes all duplicate rows from the result sets. However, if you want to retain the duplicate rows, you need to specify the ALL keyword explicitly as shown below:
query_1 UNION ALL query_2

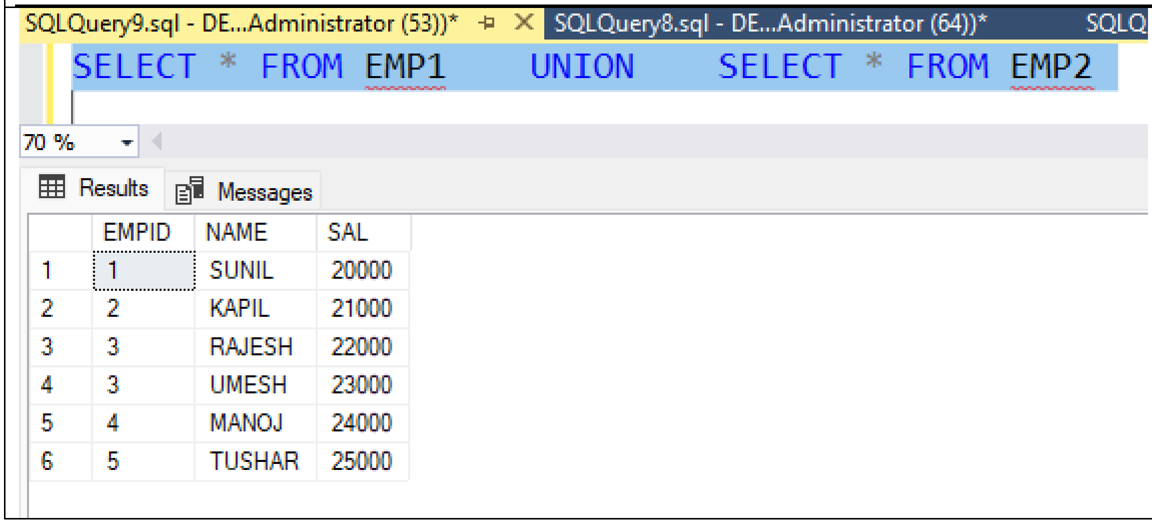
In other words, the UNION operator removes the duplicate rows while the UNION ALL operator includes the duplicate rows in the final result set.

```
CREATE TABLE EMP1
(
EMPID INT,
NAME VARCHAR(100),
SAL FLOAT
)
INSERT INTO EMP1
VALUES(1,'SUNIL',20000)
INSERT INTO EMP1
VALUES(2,'KAPIL',21000)
INSERT INTO EMP1
VALUES(3,'RAJESH',22000)
```

```
CREATE TABLE EMP2
(
EMPID INT,
NAME VARCHAR(100),
SAL FLOAT
)
INSERT INTO EMP2
VALUES(2,'KAPIL',21000)
INSERT INTO EMP2
VALUES(3,'UMESH',23000)
INSERT INTO EMP2
VALUES(4,'MANOJ',24000)
INSERT INTO EMP2
VALUES(5,'TUSHAR',25000)
```

```
CREATE TABLE EMP3
(
EMPID INT,
NAME VARCHAR(100),
SAL FLOAT,
DEPT VARCHAR(100)
)
INSERT INTO EMP3
VALUES(5,'TUSHAR',25000,'IT')
INSERT INTO EMP3
VALUES(6,'KAILASH',35000,'HR')
INSERT INTO EMP3
VALUES(7,'TEJAS',45000,'FINANCE')
```

SELECT * FROM EMP1 UNION SELECT * FROM EMP2



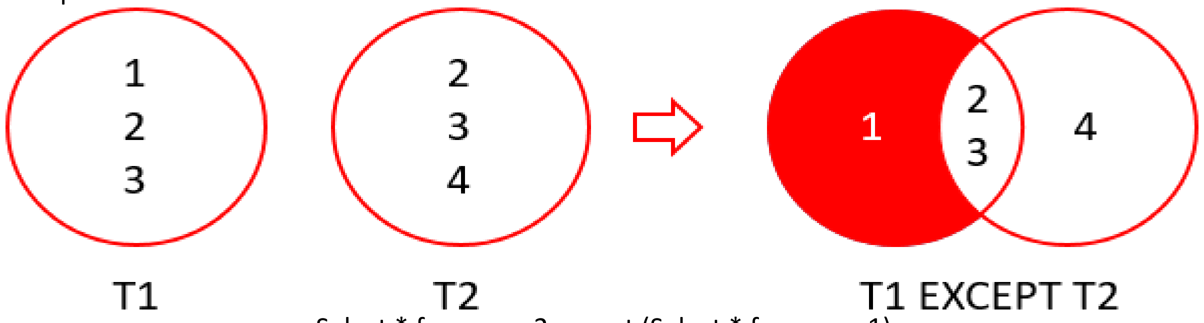
```
SELECT * FROM EMP1     UNION     SELECT * FROM EMP2
```

	EMPID	NAME	SAL
1	1	SUNIL	20000
2	2	KAPIL	21000
3	3	RAJESH	22000
4	3	UMESH	23000
5	4	MANOJ	24000
6	5	TUSHAR	25000

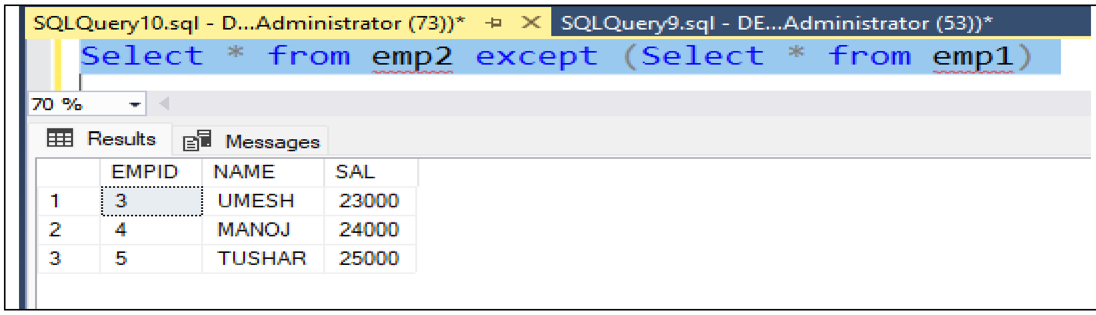
EXCEPT

The SQL Server EXCEPT compares the result sets of two queries and returns the distinct rows from the first query that are not output by the second query. In other words, the EXCEPT subtracts the result set of a query from another.

Except



Select * from emp2 except (Select * from emp1)



```
Select * from emp2 except (Select * from emp1)
```

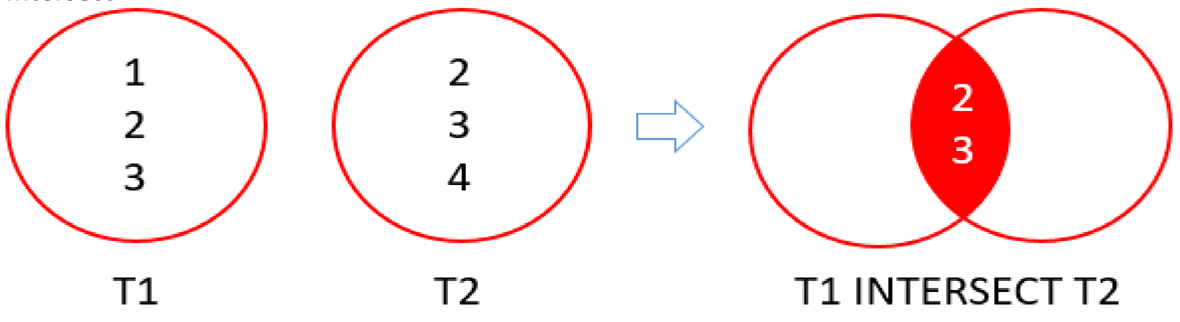
	EMPID	NAME	SAL
1	3	UMESH	23000
2	4	MANOJ	24000
3	5	TUSHAR	25000

INTERSECT

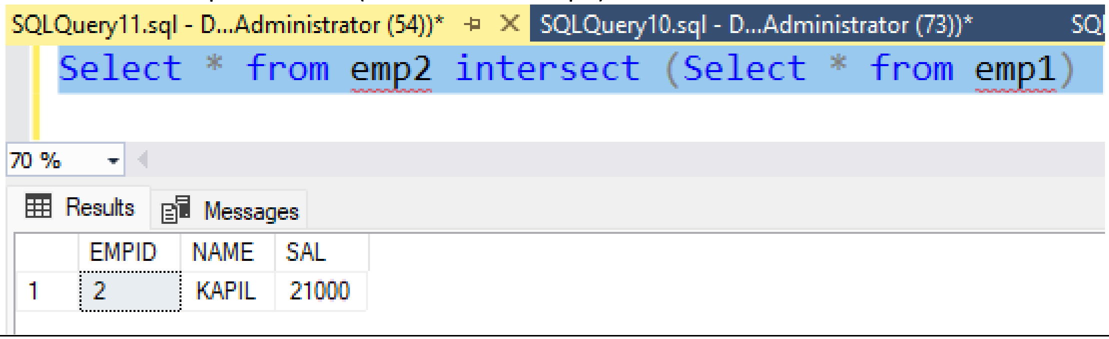
The SQL Server INTERSECT combines result sets of two or more queries and returns distinct rows that are output by both queries.

Syntax : query_1 INTERSECT query_2

Intersect



Select * from emp2 intersect (Select * from emp1)


SQLQuery11.sql - D...Administrator (54))*
SQLQuery10.sql - D...Administrator (73))*
70 %
Results
Messages

```sql
Select * from emp2 intersect (Select * from emp1)
```

```
     EMPID  NAME   SAL
1    2      KAPIL  21000
```

LESSON 2: PIVOT / UNPIVOT

Pivot and Unpivot in SQL are two relational operators that are used to convert a table expression into another.

PIVOT in SQL is used when we want to transfer data from row level to column level and UNPIVOT in SQL is used when we want to convert data from column level to row level.

PIVOT and UNPIVOT relational operators are used to generate a multidimensional reporting.

PIVOT in SQL Server

PIVOT relational operator converts data from row level to column level.

PIVOT rotates a table-valued expression by turning the unique values from one column in the expression into multiple columns in the output.

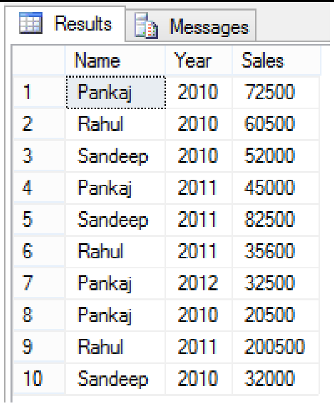
Using PIVOT operator, we can perform aggregate operation where we need them.

Syntax

```
SELECT <non-pivoted column>,
      <list of pivoted column>   FROM  (<SELECT query  to produces the data>)
   AS <alias name>
PIVOT  (  <aggregation function>(<column name>)
FOR
[<column name that  become column headers>]   IN ( [list of  pivoted columns])
 ) AS <alias name  for  pivot table>
```

☑ (Create a Sample Table Employee_Pivot to understand the Pivot Examples)

```
CREATE TABLE Employee_Pivot
          (
            Name [nvarchar](max),
            [Year] [int] ,
            Sales [int]
          )
INSERT INTO Employee_Pivot  values
('Pankaj',2010,72500),('Rahul',2010,60500),
('Sandeep',2010,52000),('Pankaj',2011,45000),
('Sandeep',2011,82500),('Rahul',2011,35600),
('Pankaj',2012,32500),('Pankaj',2010,20500),
('Rahul',2011,200500),('Sandeep',2010,32000)
```


Results | Messages

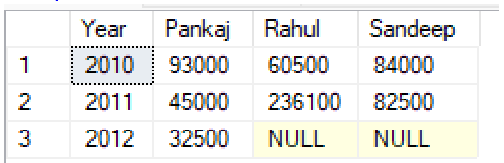
	Name	Year	Sales
1	Pankaj	2010	72500
2	Rahul	2010	60500
3	Sandeep	2010	52000
4	Pankaj	2011	45000
5	Sandeep	2011	82500
6	Rahul	2011	35600
7	Pankaj	2012	32500
8	Pankaj	2010	20500
9	Rahul	2011	200500
10	Sandeep	2010	32000

EXAMPLES OF PIVOT

```
select * from
(SELECT Name,Year,Sales FROM Employee_Pivot) t1
Pivot (  sum(Sales) for Name in(Pankaj,Rahul,Sandeep) ) t2
```

```
SELECT YEAR,[Pankaj],[Rahul],[Sandeep] from
(SELECT Name,Year,Sales FROM Employee_Pivot) Tab1
Pivot ( sum(sales) for name in (Pankaj,Rahul,Sandeep) )
as Tab2
```

Output



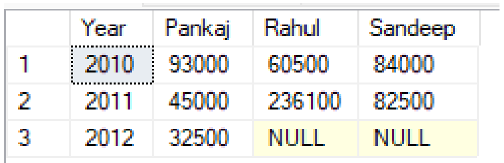
	Year	Pankaj	Rahul	Sandeep
1	2010	93000	60500	84000
2	2011	45000	236100	82500
3	2012	32500	NULL	NULL

UNPIVOT

UNPIVOT relational operator is reverse process of PIVOT relational operator. UNPIVOT relational operator convert data from column level to row level.

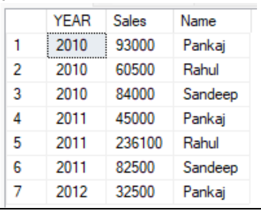
☑ Create a Employee_UnPivot table by copying the PIVOT results

```
SELECT YEAR,[Pankaj],[Rahul],[Sandeep] into Employee_UnPivot  from
(SELECT Name,Year,Sales FROM Employee_Pivot) Tab1
Pivot ( sum(sales) for name in (Pankaj,Rahul,Sandeep) )  as Tab2
```



	Year	Pankaj	Rahul	Sandeep
1	2010	93000	60500	84000
2	2011	45000	236100	82500
3	2012	32500	NULL	NULL

```
select * from
(select * from Employee_UnPivot) t1
UNPIVOT
(
Sales FOR Name IN (Pankaj,Rahul,Sandeep)
) AS TAb2
```



	YEAR	Sales	Name
1	2010	93000	Pankaj
2	2010	60500	Rahul
3	2010	84000	Sandeep
4	2011	45000	Pankaj
5	2011	236100	Rahul
6	2011	82500	Sandeep
7	2012	32500	Pankaj

CHAPTER 6: SUMMARY / CHEAT SHEET

UNION IS USED TO COMBINE THE ROWS.
Union will give only the distinct records and union all with give all the records
JOINS ARE USED TO COMBINE THE COLUMNS
INTERSECT IS USED TO GET COMMON ROWS BETWEEN TWO TABLES.
PIVOT is used to convert the column values as row header.

CHAPTER 6: QUESTIONS

What is the difference between Union and Join ?
What is the difference between Union and Union All ?
What is the difference between Union and Intersect?
What is the use of PIVOT ?
Can we apply Union on more than 2 tables ?

CHAPTER 7

CHAPTER 7 : OPERATORS, IDENTIFIERS VARIABLES & STATEMENTS

In this lesion we will discuss about the operators, identifiers and variables. What they are and how we can use it in SQL server. SQL SERVER statements that you can use to control the flow of your code. These constructs can be used in T-SQL scripts, and they are commonly used in stored procedures.

This chapter consists of the following lessons :

1. Lesson 1: Operators
2. Lesson 2: Identifiers
3. Lesson 3: Variables
4. Lesson 4 : IF-ELSE STATEMENT
5. Lesson 5 : CASE STATEMENT
6. Lesson 6 : LOOP STATEMENT

LESSON 1: OPERATORS

Operators are used to specify conditions in an SQL statement and to serve as conjunctions for multiple conditions in a statement. When a complex expression has multiple operators, operator precedence determines the sequence in which the operations are performed.

They are

- Arithmetic Operator
- Comparison Operator
- Logical Operator

Operators have the precedence levels shown in the following table. An operator on higher levels is evaluated before an operator on a lower level.

Level	Operators
1	~ (Bitwise NOT)
2	* (Multiply), / (Division), % (Modulo)
3	+ (Positive), - (Negative), + (Add), (+ Concatenate), - (Subtract), & (Bitwise AND), ^ (Bitwise Exclusive OR), | (Bitwise OR)
4	=, >, <, >=, <=, <>, !=, !>, !< (Comparison operators)
5	NOT
6	AND
7	ALL, ANY, BETWEEN, IN, LIKE, OR, SOME
8	= (Assignment)

LESSON 2: IDENTIFIERS

Identifiers are the names given to database objects such as such as tables, constraints, and stored procedures, views, columns, and data types.

All databases, servers, and database objects in SQL Server (such as tables, constraints, and stored procedures, views, columns, and data types) must have unique names, or identifiers.

They are assigned when an object is created, and used thereafter to identify the object. The identifier for the object may, if needed, be changed.

The following are the rules for creating identifiers:

- Identifiers may have between 1 and 128 characters.
- There are exceptions to this rule: certain objects are limited (for instance, temporary tables can have identifiers up to only 116 characters long).
- Before Microsoft SQL Server 7.0, identifiers were limited to 30 characters.
- The first character of the identifier must be a letter, underscore (_), at sign ([])

There are two types of identifiers:

- Regular Identifiers.
- Delimited Identifiers.

Regular identifiers : They are not delimited when they are used in Transact-SQL statements.

For example: SELECT * FROM Table1 WHERE Id = 101

Delimited identifiers : Are enclosed in double quotation marks (") or brackets ([ ]). Identifiers that comply with the rules for the format of identifiers might not be delimited.
For example: SELECT * FROM [Table1] --Delimiter is optional.
WHERE [Id] = 101 --Delimiter is optional.

LESSON 3: VARIABLES

A variable is an object that can hold a single data value of a specific type.
Declaring a Transact-SQL Variable
The **DECLARE** statement initializes a Transact-SQL variable by:
Assigning a name. The name must have a single @ as the first character.

- Assigning a system-supplied or user-defined data type and a length. For numeric variables, a precision and scale are also assigned. For variables of type XML, an optional schema collection may be assigned.
- Setting the value to NULL.

For example:

```
DECLARE @MyNumber int
DECLARE @First_Name varchar (10)
```

PROGRAMMING WITH OPERATORS

When two operators in an expression have the same operator precedence level, they are evaluated left to right based on their position in the expression. For example, in the expression that is used in the following SET statement, the subtraction operator is evaluated before the addition operator.

DECLARE @MyNumber int
SET @MyNumber = 4 - 2 + 27
- Evaluates to 2 + 27 which yields an expression result of 29.
- Then evaluate 4-29 which yields an expression result of -25

SELECT @MyNumber

LESSON 4: IF-ELSE STATEMENT

The IF/ELSE construct gives you the ability to conditionally execute code. You enter an expression after the IF keyword, and if the expression evaluates as true, the statement or block of statements after the IF statement will be executed. You can use the optional ELSE to add a different statement or block of statements that will be executed if the expression in the IF statement evaluates to false. The basic syntax of IF-ELSE statement is:

Example of IF-ELSE Statement

```
IF (Condition)
 BEGIN
          Statement
END
ELSE IF (Condition)
 BEGIN
          Statement
END
ELSE
BEGIN
          Statement
END
```

```
 Declare @Marks float
 Set @Marks=70
IF (@Marks >75)
      BEGIN
          Print 'Distinction'
END
ELSE IF (@Marks>60)
 BEGIN
          Print 'First Class'
END
ELSE IF (@Marks>40)
 BEGIN
          Print 'Second Class'
END

ELSE
BEGIN
          Print 'Fail'
END
```

LESSON 5: CASE STATEMENT

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

The basic syntax for a simple CASE expressions is shown below:

```
CASE expression
    WHEN expression1 THEN result1
    WHEN expression2 THEN result2
    ELSE expression
END
```

Example of Case Statements:

```
DECLARE @TestVal INT
SET @TestVal= 3
SELECT
CASE @TestVal
WHEN 1 THEN 'First'
WHEN 2 THEN 'Second'
WHEN 3 THEN 'Third'
ELSE 'Other'
END
```

```
CASE :  SELECT empID, empname, CASE gender
           WHEN 'MALE' THEN 'M'
         WHEN 'FEMALE' THEN 'F'
         ELSE 'NA'
        END
FROM employee
```

```
SELECT STUDENTID,PERCENTAGE,
                    Grade=CASE WHEN PERCENTAGE >=  75  THEN ‘DISTINCTION’
                      WHEN  PERCENTAGE >=  60   THEN ‘FIRST CLASS’
                      WHEN  PERCENTAGE >=  40   THEN ‘SECOND CLASS’
                      ELSE ‘FAIL’
                      END
FROM   STUDENT
```

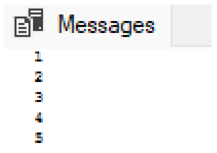
Update using case statements

```
UPDATE   STUDENT   SET
                    Grade=CASE   WHEN  percentage  >= 75 THEN  'DISTINCTION'
                      WHEN  percentage  >= 60  THEN 'FIRST CLASS'
                      WHEN  percentage  >= 40  THEN 'SECOND CLASS'
                      ELSE  ‘FAIL’
                    END
```

LESSON 6: LOOP STATEMENT

With the WHILE construct, you can create loops inside T-SQL in order to execute a statement block as long as a condition continues to evaluate to true. The keyword WHILE is followed by a condition that evaluates to either true or false. If the condition evaluates to true when it's first tested, the control of execution enters the loop, finishes the commands inside the loop the first time, and then tests the condition. Each time the loop is repeated, the WHILE condition is retested. As soon as the condition evaluates to false, the loop ends and execution control passes to the next statement following the WHILE loop. Is used to Iterate a statements multiple no of times.

Syntax:

Declaration **While(condition)** **Begin** **Statements** **incrimination** **end**	**Example** DECLARE @i INT SET @i= 1 WHILE (@i <=5) BEGIN PRINT @i SET @i = @i+ 1 END  1 2 3 4 5

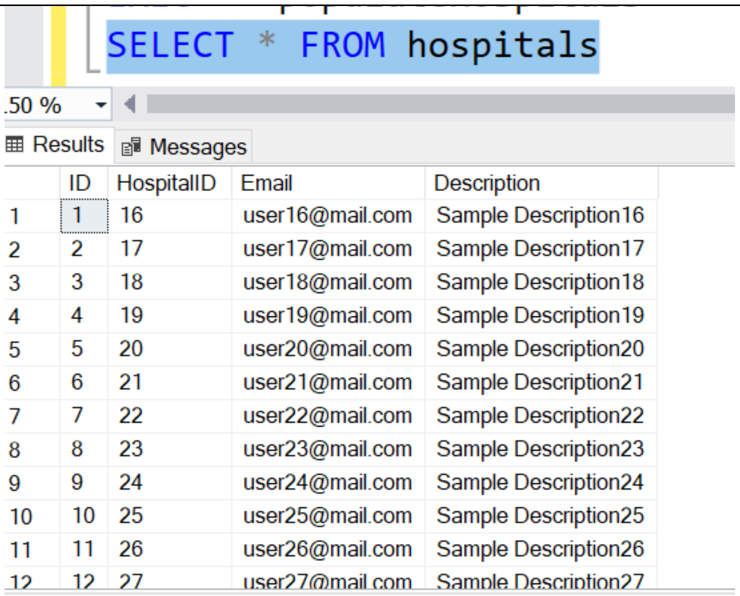
Example of table using while loop

Create table hospitals
(
ID int identity(1,1),
HospitalID int,
Email nvarchar(200),
Description nvarchar(200)
)
	CREATE PROCEDURE populateHospitals
	AS
	BEGIN
	DECLARE @hid INT
	SET @hid=16;
	WHILE @hid< 100
	BEGIN
	INSERT INTO hospitals([HospitalID], Email, Description)
	VALUES(@hid,'user'+LTRIM(STR(@hid))+'@mail.com','Sample Description'+LTRIM(STR(@hid)));
	SET @hid=@hid+ 1;
	END
	END

<table>
<tr><td>

```
EXEC    populateHospitals
SELECT * FROM hospitals
```

</td><td>



```
SELECT * FROM hospitals
```

50 %

Results | Messages

```
    ID  HospitalID  Email            Description
1   1   16          user16@mail.com  Sample Description16
2   2   17          user17@mail.com  Sample Description17
3   3   18          user18@mail.com  Sample Description18
4   4   19          user19@mail.com  Sample Description19
5   5   20          user20@mail.com  Sample Description20
6   6   21          user21@mail.com  Sample Description21
7   7   22          user22@mail.com  Sample Description22
8   8   23          user23@mail.com  Sample Description23
9   9   24          user24@mail.com  Sample Description24
10  10  25          user25@mail.com  Sample Description25
11  11  26          user26@mail.com  Sample Description26
12  12  27          user27@mail.com  Sample Description27
```

</td></tr>
</table>

Example of While Loop

```
DECLARE  @uniqueId int
DECLARE  @TEMPTABLE(uniqueIdint)

-- Insert into the temporary table a list of the records to be updated
INSERT INTO @TEMP(uniqueId)  SELECT uniqueId FROM myTable

-- Start looping through the records
WHILE EXISTS (SELECT* FROM@TEMP)
BEGIN
   -- Grab the first record out
   SELECT   Top1 @uniqueId = uniqueId FROM  @TEMP
   PRINT  'Working on @uniqueId = '+ CAST(@uniqueId as varchar(100))
       -- Perform some update on the record
   UPDATE myTable  SET myField = 'something or other' WHERE uniqueId = @uniqueId

   -- Drop the record so we can move onto the next one
   DELETE  FROM  @TEMP WHERE uniqueId = @uniqueId
END
```

CHAPTER 7: SUMMARY / CHEAT SHEET

Sql Server have various types of operators like arithmetic operator, conditional operator etc.
In Sql Server we can use variables to store data.
Sql Server support conditional statements like IF-ELSE statements and case-when statements.
Sql Server supports lopping statements like while loop.

CHAPTER 7: EXERCISE(ASSIGNMENTS)

T-SQL ASSIGNMENTS

1. Write a SQL statement to display numbers from 1 to 10.
2. Write a SQL statement to display even numbers from 1 to 10.
3. Write a SQL statement to add numbers from 1 to 10 and display the result.
4. Write a SQL statement to multiple numbers from 1 to 10 and display the result.
5. Write a SQL statement to create a multiple table for a number (eg : 9).
6. Write a SQL statement to generate a Triangle like this :

```
*
* *
* * *
* * * *
* * * * *
```

CHAPTER 7: QUESTIONS

1	What are operators in SQL server?
2	What are the different types of operator?
3	Which is better temp table or table variable?
4	What are identifiers in SQL server?

CHAPTER 8

CHAPTER 8: STORED PROCEDURE & FUNCTIONS

T-SQL code can be stored within Microsoft SQL Server databases by using T-SQL routines such as stored procedures, triggers, and functions. This helps make your T-SQL code more portable, because it stays with the database and these routines will be restored from a database backup along with the database data. These routines can then be executed from the database as well.

In this chapter, you learn about creating reusable T-SQL routines in stored procedures and user-defined functions.

Lessons in this chapter:

1. Lesson 1: Implementing Stored Procedures
2. Lesson 2: Implementing User-Defined Functions

LESSON 1: IMPLEMENTING STORED PROCEDURES

Sored Procedure is a group of SQL statements which is saved as program which can be reused when they required

Important features of stored procedures:

- They can be called from T-SQL code by using the EXECUTE command.
- You can pass data to them through input parameters, and receive data back through output parameters.
- They can return result sets of queries to the client application.
- They can modify data in tables.

SYNTAX FOR CREATE STORED PROCEDURE	**STORED PROCEDURE TO ADD TWO NUMBERS AND SHOW THE RESULTS**

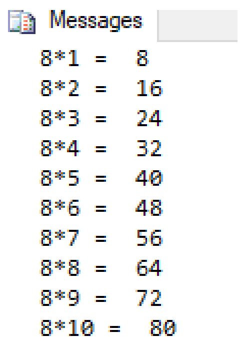
```
Create procedure procedure_name
(
Parameter1,
Parameter2,
Parameter3
)
As
Begin
Statements
End
```
SYNTAX FOR CALLING STORED PROCEDURE:
Exec sp_name parameters.
	```
	Create procedure sp_add
	(
	@n1,
	@n2
	)
	As
	Begin
	DECLARE @total int
	SET @total =@n1 + @n2
	PRINT @total
	End

	EXEC sp_add 20,10
	OUTPUT : 30
	```

WRITE A STORED PROCEDURE TO SHOW A MULTIPLICATION TABLE FOR A GIVEN NUMBER

```
CREATE PROCEDURE SP_MULTIPLYTBL
(
@NUM INT
)
AS
BEGIN
DECLARE @I INT = 1
DECLARE @T INT = 1
WHILE (@I <= 10)
BEGIN
SET @T =@NUM * @I
PRINT CAST(@NUM AS VARCHAR(10)) + '*' + CAST(@I AS
VARCHAR(10)) + ' =  ' + CAST(@T AS VARCHAR(10))
SET @I = @I +1
END
END
```

CALLING A STORED PROCEDURE

```
EXEC SP_MULTIPLYTBL 8
```



Messages

```
8*1 =  8
8*2 =  16
8*3 =  24
8*4 =  32
8*5 =  40
8*6 =  48
8*7 =  56
8*8 =  64
8*9 =  72
8*10 =  80
```

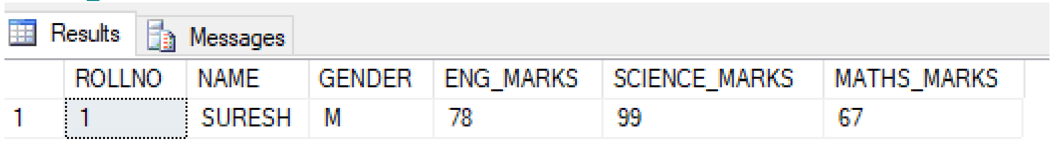
STORED PROCEDURE EXAMPLES

Step 1 : Create a Table

```
CREATE  TABLE  [STUDENTS]
(
            [ROLLNO]  [int]  IDENTITY(1,1)  CONSTRAINT  [PK_ROLLNO]  PRIMARY KEY,
            [NAME]  [nvarchar](100) NULL,
            [GENDER]  [char](1)  NULL,
            [ENG_MARKS]  [float]  NULL,
            [SCIENCE_MARKS]  [float]  NULL,
            [MATHS_MARKS]  [float]  NULL
)
```

Create a Procedure to Insert a student

```
CREATE  PROCEDURE  SP_INSERTSTUDENTS
(
@STUDNAME  NVARCHAR(50),
@GEN  CHAR(1),
@ENG  FLOAT,
@SCN FLOAT,
@MATHS  FLOAT
)
AS
BEGIN
INSERT  INTO STUDENTS VALUES(@STUDNAME,@GEN,@ENG,@SCN,@MATHS)
SELECT * FROM STUDENTS
END

EXEC SP_INSERTSTUDENTS 'SURESH','M',78,99,67
```



Results | Messages

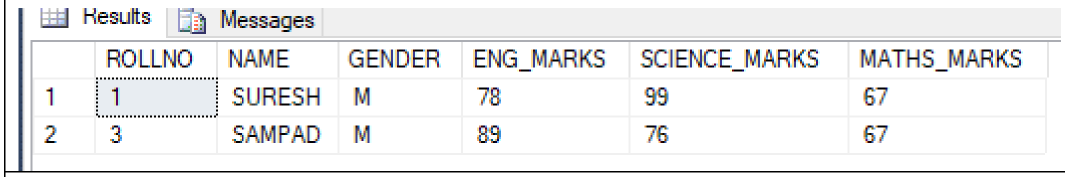
	ROLLNO	NAME	GENDER	ENG_MARKS	SCIENCE_MARKS	MATHS_MARKS
1	1	SURESH	M	78	99	67

Step 2: Create a Procedure to Insert a student if the students does not exists

```
CREATE  PROCEDURE  SP_ADDSTUDENTS
(
@STUDNAME  NVARCHAR(50),
@GEN  CHAR(1),
@ENG  FLOAT,
@SCN FLOAT,
@MATHS  FLOAT
)
AS
BEGIN
DECLARE @COU  INT= 0
SELECT   @COU=COUNT(*)  FROM  STUDENTS  WHERE NAME=@STUDNAME -- THIS WILL CHECK IF THE STUDENT IS
ALREADY PRESENT

IF (@COU> 0)
BEGIN
PRINT  'STUDENT ALREADY PRESENT, PLEASE ENTER A NEW NAME,NAME='+@STUDNAME
END
ELSE
BEGIN
INSERT  INTO STUDENTS VALUES(@STUDNAME,@GEN,@ENG,@SCN,@MATHS)
SELECT  *  FROM  STUDENTS
END
END
```

CALLING A STORED PROCEDURE

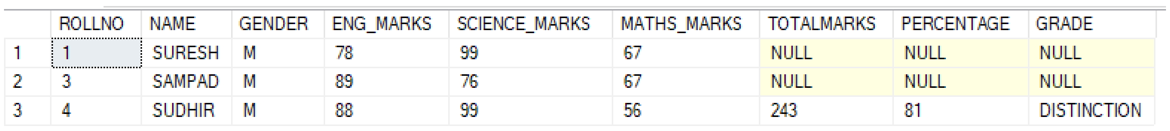
```
Exec  [SP_INSERTSTUDENTS]  'SAMPAD','M',89,76,67
```


Results | Messages

	ROLLNO	NAME	GENDER	ENG_MARKS	SCIENCE_MARKS	MATHS_MARKS
1	1	SURESH	M	78	99	67
2	3	SAMPAD	M	89	76	67

```
ALTER TABLE STUDENTS ADD TOTALMARKS FLOAT,PERCENTAGE FLOAT,GRADE VARCHAR(100)

CREATE  PROCEDURE  SP_SAVESTUDENTS
(
@STUDNAME  NVARCHAR(50),
@GEN  CHAR(1),
@ENG  FLOAT,
@SCN FLOAT,
@MATHS  FLOAT
)
AS
BEGIN
DECLARE @TOTAL FLOAT,@PER FLOAT
DECLARE @GRADE VARCHAR(100)
SET @TOTAL = @ENG + @SCN + @MATHS
SET @PER = @TOTAL /3
SET @GRADE = (CASE WHEN @PER >= 75 THEN 'DISTINCTION' WHEN @PER >= 60 THEN 'FIRST CLASS' WHEN @PER >=
40 THEN 'SECOND CLASS' ELSE 'FAIL' END)
INSERT  INTO STUDENTS VALUES(@STUDNAME,@GEN,@ENG,@SCN,@MATHS,@TOTAL,@PER,@GRADE)
```

```
SELECT * FROM STUDENTS
END
EXEC SP_SAVESTUDENTS 'SUDHIR','M',88,99,56
```



	ROLLNO	NAME	GENDER	ENG_MARKS	SCIENCE_MARKS	MATHS_MARKS	TOTALMARKS	PERCENTAGE	GRADE
1	1	SURESH	M	78	99	67	NULL	NULL	NULL
2	3	SAMPAD	M	89	76	67	NULL	NULL	NULL
3	4	SUDHIR	M	88	99	56	243	81	DISTINCTION

MORE EXAMPLES ON STORED PROCEDURES

FIRST CREATE THE TABLES (COUNTRY & STATE) AND THEN WILL DO AN EXAMPLE OF INSERTING VALUES INTO BOTH THE TABLES

```
CREATE  TABLE  COUNTRY
(
COUNTRY_ID  INT  IDENTITY(1,1) PRIMARY KEY,
COUNTRY_NAME  NVARCHAR(100),
COUNTRY_CAPITAL  NVARCHAR(100)
)
```

```
CREATE  TABLE  STATE
(
STATE_ID  INT  IDENTITY(1,1)  PRIMARY KEY,
STATE_NAME  NVARCHAR(100),
STATE_CAPITAL  NVARCHAR(100),
COUNTRY_ID INT  CONSTRAINT  FK_COUNTRY
FOREIGN KEY  REFERENCES
COUNTRY(COUNTRY_ID)
)
```

SP –Examples: Create a Stored Procedure to insert a COUNTRY AND STATE.

```
CREATE PROCEDURE  SP_INSERTCOUNTRY
(
@COUNTRYNAME  NVARCHAR(100),
@COUNTRYCAPITAL  NVARCHAR(100),
@STATENAME   NVARCHAR(100),
@STATECAPITAL  NVARCHAR(100)
)
AS
BEGIN
DECLARE  @COUNTRYID  INT
INSERT  INTO  COUNTRY  VALUES (@COUNTRYNAME,@COUNTRYCAPITAL)
SET @COUNTRYID=@@IDENTITY
INSERT  INTO  STATE VALUES (@STATENAME,@STATECAPITAL,@COUNTRYID)
SELECT  * FROM  COUNTRY
SELECT  * FROM  STATE
END
```

Stored Procedure Example:

```
CREATE  PROCEDURE  [SP_INSERTCOUNTRY]
(
@COUNTRYNAME      NVARCHAR(100),
@COUNTRYCAPITAL  NVARCHAR(100),
@STATENAME      NVARCHAR(100),
@STATECAPITAL  NVARCHAR(100)
)
AS
BEGIN
DECLARE  @COUNTRYID  INT=0
Select  @COUNTRYID=COUNTRY_ID  FROM  COUNTRY WHERE
COUNTRY_NAME=@COUNTRYNAME
IF (@COUNTRYID= 0)
BEGIN
INSERT  INTO COUNTRY  VALUES(@COUNTRYNAME,@COUNTRYCAPITAL)
SET  @COUNTRYID=@@IDENTITY
END
ELSE
BEGIN
PRINT  'COUNTRY ALREADY EXISTS'
END
INSERT  INTO  STATE VALUES(@STATENAME,@STATECAPITAL,@COUNTRYID)
SELECT *  FROM COUNTRY
SELECT *  FROM STATE
END

EXEC  [SP_INSERTCOUNTRY]  'INDIA','DELHI','GUJURAT','AHMEDABAD'

EXEC  [SP_INSERTCOUNTRY]  'INDIA','DELHI','MP','BHOPAL'

EXEC [SP_INSERTCOUNTRY]  'U.S.A','WASIGNTON','MS','BOSTON'
```

Output Parameters in Stored Procedures

Setting up output parameters for a stored procedure is basically the same as setting up input parameters, the only difference is that you use the OUTPUT clause after the parameter name to specify that it should return a value. The output clause can be specified by either using the keyword "OUTPUT" or just "OUT".

Example: Creating a Stored Procedure with output parameters

```
CREATE PROCEDURE SP_ADDITION
(
@n1 int,
@n2 int,
@t int output
)
as
BEGIN
set @t = @n1 + @n2
END
```

Calling the stored procedure with output parameters

```
DECLARE @tot int
exec SP_ADDITION 100,200 , @tot output
print @tot
```

Types of Stored Procedures

- **User Defined Stored procedure**: The user defined stored procedures are created by users and stored in the current database

- **Temporary Stored procedures**: The temporary stored procedures have names prefixed with the # symbol. Temporary stored procedures stored in the tempdb databases. These procedures are automatically dropped when the connection terminates between client and server.

  Temporary stored procedures on Microsoft SQL Server are prefixed with a pound sign #. One pound sign means that it’s temporary within the session, two pound signs ## means it’s a global temporary procedure, which can be called by any connection to the SQL server during its lifetime.

  Example of Temporary Stored Procedure:

```
CREATE PROC #SP_MULTIPLY
(
@N1 INT,
@N2 INT
)
AS
BEGIN
DECLARE @T INT = @N1 * @N2
PRINT @T
END

EXEC #SP_MULTIPLY 10,20
```

- **Remote Stored Procedures:** The remote stored procedures are procedures that are created and stored in databases on remote servers. These remote procedures can be accessed from various servers, provided the users have the appropriate permission

- **Extended Stored Procedures:** These are Dynamic-link libraries (DLL's) that are executed outside the SQL Server environment. They are identified by the prefix xp_

Creating Extended Stored Procedures

An extended stored procedure appears as a normal function with the following prototype:

sp_addextendedproc 'xp_average', 'xp_average.dll', where xp_ is optional and the xp name is case sensitive. All functions outside of the DLL must be exported, which can be done by listing its name in the EXPORTS section of .def file.

Let's examine a code snippet to create an **Extended stored procedure DLL in C# -** compiled and stored in 'C: \Program Files\Microsoft SQL Server\MSSQL12.SQL2014\MSSQL\Bin\

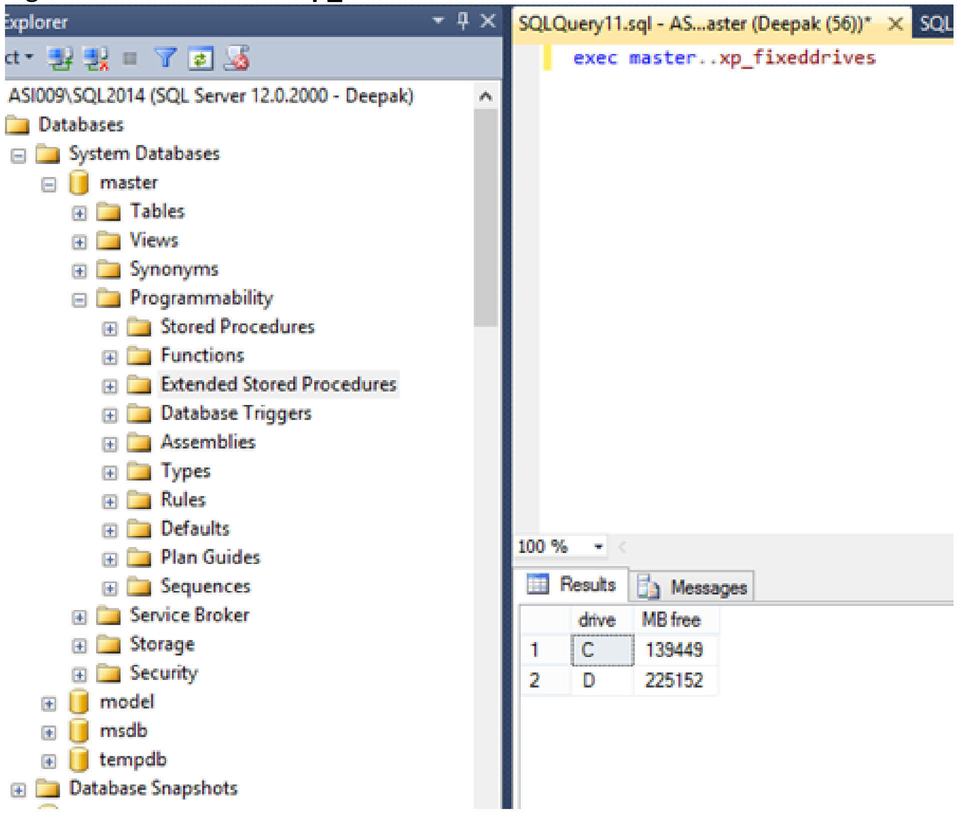
```
public class xp_sampleUser
{
Staticstring connection String =
 System.Configuration.ConfigurationManager.ConnectionString ["Connection String"].
ConnectionString.ToString ();
[Microsoft.SqlServer.Server.SqlProcedure]
publicstaticvoid HelloWorld ()
{
SqlContext.Pipe.Send ("Hello world! The Time is" + System.DateTime.Now.ToString ()
 + "\n");
Using (SqlConnection connection = newSqlConnection (connection String))
{
Connection. Open ();
SqlCommand command = newSqlCommand ("SELECT value FROM
 MSreplication_options where opt name='transactional'", connection);
SqlDataReader reader = command.ExecuteReader ();
SqlContext.Pipe.Send (reader);
}
}
}
```

To add the procedure, we are prompted with **sp_addextendedproc 'xp_average', 'xp_average.dll'**

Calling an XP is similar to calling a normal stored procedure, but it remains inside a master database. Therefore, a prefix 'master' DB name is mandatory. To demonstrate extended stored procedures, we'll use an example.

Sample: To find the free space of a drive

Signature **exec master..xp_fixeddrives**



```
exec master..xp_fixeddrives
```

	drive	MB free
1	C	139449
2	D	225152

Advantages

- Create DLL routines as regular Stored Procedures for any SQL Server version.
- Since they use API-Request/Response formats, they are capable of running remotely.
- In the case of non–SQL Server based applications, we can create gateways to tie the two programs together.
- We can interface with various applications, and this function will deliver the result set as a whole.

Disadvantages

- The reliability of the server may get reduced by extended stored procedures, sometimes leading to memory leaks.
- Granting permission to other users is mandatory - system admin reviews are a must after each update.
- If user data was directly appended as an input, SQL Injection may possible.

SPs - Auto Executing-startup procedures

You can designate stored procedures to execute every time the SQL Server is started. These types of procedures cannot accept any input parameters and have to be owned by a member of SYSADMIN fixed server role. To designate stored procedures for automatic execution, use the sp_procoption system stored procedure.

The only option allowed by this procedure is 'startup'. The procedure to be started automatically MUST reside in the Master database. The following example makes the procedure execute automatically every time the server starts up: **sp_procoption my_procedure,'startup', 'on'**

- This option could be useful if you have specific processing requirements or tasks that need to be perform at server startup - for instance you might wish to backup all of your user databases every time SQL Server is started.

```
    USE MASTER
        CREATE TABLE LOGGEDINUSERDETAILS
        (
        ID INT IDENTITY(1,1),
        USERNAME VARCHAR(100),
        LOGGEDINTIME DATETIME
        )
        SELECT * FROM LOGGEDINUSERDETAILS
        CREATE PROCEDURE SP_LOGIN
        AS
        BEGIN
        INSERT INTO LOGGEDINUSERDETAILS VALUES(SUSER_NAME(),GETDATE())
        END

sp_procoption    'SP_LOGIN','startup', 'on'
```

System Stored Procedure: The system stored procedure has names prefixed with sp_. Its manage SQL Server through administrative tasks. Which databases store system stored procedures are master and msdb database. E.g. : sp_rename

SYSTEM STORED PROCEDURES

System Stored Procedures are useful in performing administrative and informational activities in SQL Server. Here’s a bunch of System Stored Procedures that are used on a frequent basis (in no particular order):

System Stored Procedure	**Description**
sp_help	Reports information about a database object, a user-defined data type, or a data type
sp_helpdb	Reports information about a specified database or all databases
sp_helptext	Displays the definition of a user-defined rule, default, unencrypted Transact-SQL stored procedure, user-defined

	Transact-SQL function, trigger, computed column, CHECK constraint, view, or system object such as a system stored procedure
sp_helpfile	Returns the physical names and attributes of files associated with the current database. Use this stored procedure to determine the names of files to attach to or detach from the server
sp_spaceused	Displays the number of rows, disk space reserved, and disk space used by a table, indexed view, or Service Broker queue in the current database, or displays the disk space reserved and used by the whole database
sp_who	Provides information about current users, sessions, and processes in an instance of the Microsoft SQL Server Database Engine. The information can be filtered to return only those processes that are not idle, that belong to a specific user, or that belong to a specific session
sp_lock	Reports information about locks. This stored procedure will be removed in a future version of Microsoft SQL Server. Use the sys.dm_tran_locks dynamic management view instead.
sp_configure	Displays or changes global configuration settings for the current server
sp_tables	Returns a list of objects that can be queried in the current environment. This means any object that can appear in a FROM clause, except synonym objects.
sp_columns	Returns column information for the specified tables or views that can be queried in the current environment
sp_depends	Displays information about database object dependencies, such as the views and procedures that depend on a table or view, and the tables and views that are depended on by the view or procedure. References to objects outside the current database are not reported

LESSON 2: IMPLEMENTING USER-DEFINED FUNCTIONS

They simply return a single value or a table value. They can only be use to select records. However, they can be called very easily from within standard SQL

Understanding User-Defined Functions

The purpose of a user-defined function (UDF) is to encapsulate reusable T-SQL code and return a scalar value or a table to the caller. Like stored procedures, UDFs can accept parameters, and the parameters can be accessed inside the function as variables.

- UDFs cannot be executed by using the EXECUTE command.
- UDFs access SQL Server data, but they cannot perform any DDL—that is, they cannot create tables, and they cannot make modifications to tables, indexes, or other objects, or change any data in permanent tables by using a DML statement.

Syntax of Function:

```
Create function function_name ( Parameter1 datatype, Parameter2 datatype )
RETURNS DATATYPE
AS
BEGIN
STATEMENTS
RETURN VALUE
END

SYNTAX FOR CALLING A FUNCTION
SELECT dbo.functionname('Parameter1')
OR
SELECT Name, dbo.Functionname('Parameter1') FROM sysObjects
```

Function Examples

```
Create function fn_add (@n1 int, @n2 int ) returns INT
AS
BEGIN
Declare @tot int
Set @tot= @n1 + @n2
Return @tot
End

CALLING A FUNCTION :    SYNTAX    : Select  dbo.function_name(parameters)
EXAMPLE
Select dbo. fn_add (100,200)
```

For simple reusable select operations, functions can simply your code. Just be wary of using JOIN clauses in your functions. If your function has a JOIN clause and you call it from another select statement that returns multiple results that function call with JOIN those tables together for EACH line returned in the result set. So they can be helpful in sampling some logic, they can also be a performance bottleneck if they're not used properly.

There are two major types of UDFs:

- Scalar and table-valued.
- Table-valued function

The scalar function

Scalar function returns a single value back. . Scalar UDFs can consist of a single line of T-SQL code, or of multiple lines.

```
WRITE A FUNCTION TO GET THE NUMBER OF EMPLOYEES IN A DEPT
CREATE FUNCTION FN_GETEMPCOUNT
(
@DEPTNAME VARCHAR(100)
)
RETURNS INT
AS
BEGIN
DECLARE @COU INT
SELECT @COU = COUNT(*) FROM EMPLOYEE WHERE DEPARTMENT=@DEPTNAME
RETURN @COU
END

SELECT DBO.FN_GETEMPCOUNT('IT') AS EMPCOUNT
```

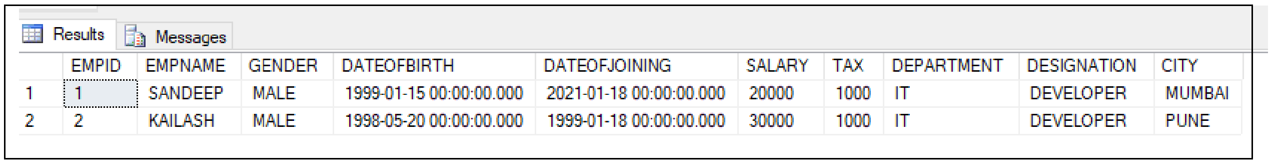
Table-Valued Functions

User-defined table-valued functions return a table data type**.** For an inline table-valued function, there is no function body; the table is the result set of a single SELECT statement.

The following example creates an inline table-valued function. The function takes one input parameter, a customer (store) ID, and returns the columns ProductID, Name, and the aggregate of year-to-date sales as YTD Total for each product sold to the store.

Example:

```
CREATE FUNCTION FN_GETEMP
(
@DEPTNAME VARCHAR(100)
)
RETURNS TABLE
RETURN (SELECT * FROM EMPLOYEE WHERE DEPARTMENT = @DEPTNAME)

SELECT * FROM FN_GETEMP('IT')
```



Results | Messages

	EMPID	EMPNAME	GENDER	DATEOFBIRTH	DATEOFJOINING	SALARY	TAX	DEPARTMENT	DESIGNATION	CITY
1	1	SANDEEP	MALE	1999-01-15 00:00:00.000	2021-01-18 00:00:00.000	20000	1000	IT	DEVELOPER	MUMBAI
2	2	KAILASH	MALE	1998-05-20 00:00:00.000	1999-01-18 00:00:00.000	30000	1000	IT	DEVELOPER	PUNE

Note: Functions cannot be used to update, delete, or add records to the database.

Difference between stored procedures and functions.

1. Unlike Stored Procedure, Function returns only single value.
2. Unlike Stored Procedure, Function accepts only input parameters.
3. Unlike Stored Procedure, Function is not used to Insert, Update, Delete data in database table(s).
4. Like Stored Procedure, Function can be nested up to 32 levels.
5. User Defined Function can have up to 1023 input parameters while a Stored Procedure can have up to 21000 input parameters.
6. User Defined Function can't returns XML Data Type.
7. User Defined Function doesn't support Exception handling.
8. User Defined Function can call only Extended Stored Procedure.
9. User Defined Function doesn't support set options like set ROWCOUNT etc.

CHAPTER 8: EXERCISE 1 (ASSIGNMENT)

PROCEDURE ASSIGNMENTS

1. Write a PROCEDURE to display a multiplication table for a number. User will enter a number and it should display the multiplication table.
2. Write a PROCEDURE to calculate the area of a rectangle. User will enter the length and breadth and it should display the area.
3. Write a PROCEDURE to check whether the number is prime number or not. User will enter a number and it should display whether it is a prime number or not.
4. Write a procedure which will bifurcate a fullname into firstname and lastname.User will enter fullname like Amit Patil and it should display FirstName : Amit and LastName : Patil
5. Write a PROCEDURE to generate a triangle similar to this.

```
*
* *
* * *
* * * *
* * * * *
```

FUNCTION ASSIGNMENTS

1. Write a function to add two numbers.
2. Write a function to add /sub/mul/div based upon the mode passed. (Eg: user will pass two numbers and mode, (5, 6,"Add") accordingly it should return value).
3. Write a function to calculate the age in year. User will pass the Date of Birth and it should return the age in terms of number.
4. Write a function to calculate the factorial of a number.
5. Write a Function to calculate the square of a number and return the result.
6. Write a Function to calculate the grade for the percentage enter. User will enter the percentage it should return the GRADE. (DISTINCTION >= 75, FIRSTCLASS>=60, SECONDCLASS>=40, FAIL)

CHAPTER 8: EXERCISE 2 (ASSIGNMENT)

CREATE A TABLE FOR COMPANY

COLUMN_NAME	DATATYPE	CONSTRAINT
COMPANYID	INT	PRIMARY KEY, IDENTITY(1,1)
COMPANYNAME	VARCHAR(100)	
WEBSITE	VARCHAR(100)	
EMAIL	VARCHAR(100)	

Write a Stored procedure to add/insert a row and then display the company row added.
Write a Stored Procedure to delete a company by using CompanyId

CREATE A TABLE FOR DEPT

COLUMN_NAME	DATATYPE	CONSTRAINT
DEPTID	INT	PRIMARY KEY, IDENTITY(1,1)
DEPTNAME	VARCHAR(100)	
DEPTLOCATION	VARCHAR(100)	

Write a Stored procedure to add/insert a row and then display the department row added.
Write a Stored Procedure to delete a department by using DepartmentId

CREATE A TABLE FOR DESG

COLUMN_NAME	DATATYPE	CONSTRAINT
DESGID	INT	PRIMARY KEY
DESGNAME	VARCHAR(100)	
CREATEDDATE	DATETIME	DEFAULT - GETDATE()

Write a Stored procedure to add/insert a row and then display the desg row added.
Write a Stored Procedure to delete a designation by using DesignationId

CREATE A TABLE FOR EMPLOYEE

COLUMN_NAME	DATATYPE	CONSTRAINT
EMPID	INT	PRIMARY KEY

EMPNAME	VARCHAR(100)	UNIQUE
BASICSALARY	FLOAT	
INCENTIVE	FLOAT	
BONUS	FLOAT	
DOB	DATETIME	
DOJ	DATETIME	DEFAULT GETDATE()
GENDER	VARCHAR(100)	
EMAIL	VARCHAR(100)	UNIQUE
MOBILE	VARCHAR(10)	
CREATEDDATE	DATETIME	DEFAULT GETDATE()
COMPANYID	INT	F.K
DEPTID	INT	F.K
DESGID	INT	F.K

Write a Stored procedure to add and display an Employee
Write a Stored Procedure to delete an Employee by using EmployeeId

CREATE A TABLE FOR STUDENT

COLUMN_NAME	DATATYPE	CONSTRAINT
ROLLNO	INT	PRIMARY KEY (IDENTITY(1,1)
NAME	VARCHAR(100)	UNIQUE
ENGMARKS	FLOAT	
SCNMARKS	FLOAT	
MATHMARKS	FLOAT	
TOTALMARKS	FLOAT	
PERCENTAGE	FLOAT	
GRADE	VARCHAR(100)	

```
CREATE TABLE STUDENTS
(
ROLLNO INT IDENTITY(1,1) PRIMARY KEY,
NAME VARCHAR(100) UNIQUE,
ENGMARKS FLOAT,
SCNMARKS FLOAT,
MATHMARKS FLOAT,
TOTAL FLOAT,
PERCENTAGE FLOAT,
GRADE VARCHAR(100)
)
```

- Part A

WRITE A PROCEDURE TO INSERT VALUES INTO STUDENTS
THE PROCEDURE SHOULD TAKE NAME, ENGMAKS, SCNMARKS, MATHMARKS AS INPUT PARAMETERS AND CALCULATE THE TOTALMARKS, PERCENTAGE & GRADE AND INSERT IT INTO THE STUDENTS TABLE

- Part B

IF THE NAME IS ALREADY PRESENT THEN IT SHOULD UPDATE THE VALUES INTO THE STUDENTS TABLE.

CREATE TWO TABLES COUNTRY & STATE

COUNTRY

COUNTRY_ID	COUNTRY_NAME	COUNTRY_CAPITAL
INT, IDENTITY, PRIMARY KEY	UNIQUE	

STATE

STATE_ID	STATE_NAME	STATE_CAPITAL	COUNRTYID
INT, IDENTITY, PRIMARY KEY	UNIQUE		INT (FOREIGN KEY)

```
CREATE TABLE COUNTRY
(
COUNTRYID INT IDENTITY(1,1) PRIMARY KEY,
COUNTRYNAME VARCHAR(100),
COUNTRYCAPITAL VARCHAR(100)
)

CREATE TABLE STATES
(
STATEID INT IDENTITY(1,1) PRIMARY KEY,
STATENAME VARCHAR(100),
STATECAPITAL VARCHAR(100),
COUNTRYID INT FOREIGN KEY REFERENCES COUNTRY(COUNTRYID)
)
```

WRITE A SINGLE PROCEDURE TO INSERT VALUES INTO BOTH COUNTRY & STATE TABLE.
PROCEDURE PARAMETERS

COUNTRY_NAME	COUNTRY_CAPITAL	STATE_NAME	STATE_CAPITAL

1	**WRITE A PROCEDURE TO TAKE THE BACKUPS OF ALL THE DATABASES IN THE SERVER**
2	**WRITE A PROCEDURE TO DISPLAY THE NAMES OF ALL THE TABLES IN THE DATABASE. THE USER WILL ENTER THE DATABASE NAME AND IT SHOULD DISPLAY THE LIST OF TABLES IN THE DATABASE**
3	**WRITE A PROCEDURE TO SEND A MAIL TO An EMAIL ID. THE USER WILL ENTER THE EMAILID AND IT SHOULD SEND A MAIL TO THE EMAILID**
4	**WRITE A PROCEDURE TO DISPLAY THE COLUMN NAMES OF A TABLE. THE USER WILL ENTER THE TABLENAME AND IT SHOULD DISPLAY THE LIST OF COLUMNS OF THE TABLE**
5	**WRITE A PROCEDURE TO DISPLAY THE SIZE OF EACH OF THE DATABASES IN THE SERVER**

CHAPTER 8: QUESTIONS

Define Stored Procedure along with its usage ?
What are the advantages of Stored Procedure ?
What are the disadvantages of Stored Procedure ?
What is a function in SQL Server?
What are the different types of functions in sql server ?
Advantages of functions in SQL SERVER ?
Difference between stored procedure and functions ?
Can we call stored procedure inside function ?
Can we call function inside stored Procedure ?

CHAPTER 9

CHAPTER 9: IMPLEMENTING TRANSACTIONS, ERROR HANDLING

Microsoft SQL Server is a relational database that strictly enforces transactional behavior on database changes in order to protect data integrity. This chapter presents the T-SQL code behind transactions, and extends that code to handling errors and using dynamic SQL.

Lessons in this chapter:

- Lesson 1: Implementing Error Handling
- Lesson 2: Managing Transactions

LESSON 1: IMPLEMENTING ERROR HANDLING

Exception handling are used to handle errors raised in the program while execution due to wrong data etc.
The error needs to be handled else the program will terminate abnormally.

All statements should be put inside the begin try and end try block to handle exceptions in SQL statements.

SYNTAX FOR HANDLING ERRORS

```
BEGIN TRY
    {SQL_statement | statement_block}
END TRY
BEGIN CATCH
    [{sql_statement | statement_block}]
END CATCH
```

In case of error in SQL statements, it will be redirected to the CATCH block.

sql_statement : Is any Transact-SQL statement.

statement_block : Any group of Transact-SQL statements in a batch or enclosed in a BEGIN…END block.

A TRY…CATCH construct catches all execution errors that have a severity higher than 10 that do not close the database connection.

A TRY block must be immediately followed by an associated CATCH block. Including any other statements between the END TRY and BEGIN CATCH statements generates a syntax error.

Errors encountered in a CATCH block are treated like errors generated anywhere else.

If the CATCH block contains a nested TRY…CATCH construct, any error in the nested TRY block will pass control to the nested CATCH block. If there is no nested TRY…CATCH construct, the error is passed back to the caller.

GOTO statements cannot be used to enter a TRY or CATCH block. GOTO statements can be used to jump to a label inside the same TRY or CATCH block or to leave a TRY or CATCH block.

Example:

```
BEGIN TRY
   -- Generate divide-by-zero error.
   SELECT 1/0;
END TRY
BEGIN CATCH
   -- Execute error retrieval routine.
   PRINT 'ERROR OCCURRED'
END CATCH;
```

The TRY…CATCH construct cannot be used in a user-defined function.

Retrieving Error Information

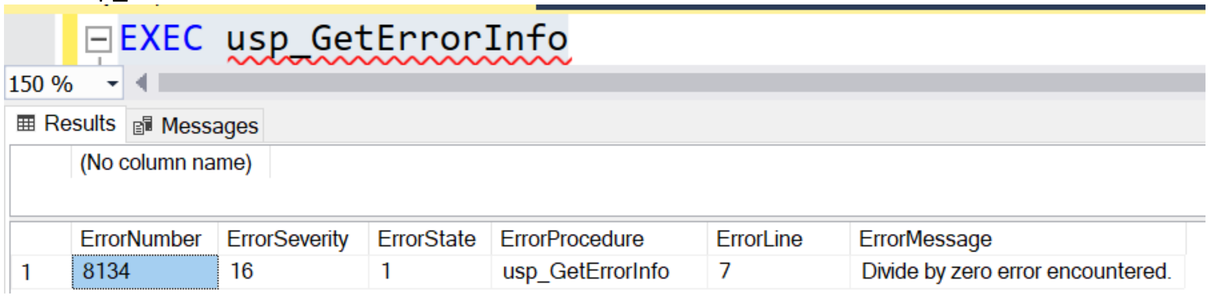
In the scope of a CATCH block, the following system functions can be used to obtain information about the error that caused the CATCH block to be executed:

ERROR_NUMBER()	Returns the number of the error.
ERROR_SEVERITY()	Returns the severity.
ERROR_STATE()	Returns the error state number.
ERROR_PROCEDURE()	Returns the name of the stored procedure or trigger where the error occurred.
ERROR_LINE()	Returns the line number inside the routine that caused the error.

ERROR_MESSAGE()	Returns the complete text of the error message. The text includes the values supplied for any substitutable parameters, such as lengths, object names, or times.

These functions return NULL if they are called outside the scope of the CATCH block. Error information can be retrieved by using these functions from anywhere within the scope of the CATCH block.

```
CREATE PROCEDURE usp_GetErrorInfo
AS
BEGIN
BEGIN TRY
   -- Generate divide-by-zero error.
   SELECT 1/0;
END TRY
BEGIN CATCH
   -- Execute error retrieval routine.
SELECT   ERROR_NUMBER() AS ErrorNumber ,ERROR_SEVERITY() AS ErrorSeverity
,ERROR_STATE() AS ErrorState,ERROR_PROCEDURE() AS ErrorProcedure
,ERROR_LINE() AS ErrorLine,ERROR_MESSAGE() AS ErrorMessage;
END CATCH

END

EXEC usp_GetErrorInfo
```



```
EXEC usp_GetErrorInfo
```

```
   (No column name)

   ErrorNumber  ErrorSeverity  ErrorState  ErrorProcedure    ErrorLine  ErrorMessage
1  8134         16             1           usp_GetErrorInfo  7          Divide by zero error encountered.
```

EXAMPLE OF ERROR HANDLING WHILE INSERT A VALUE INTO A TABLE THROUGH SP

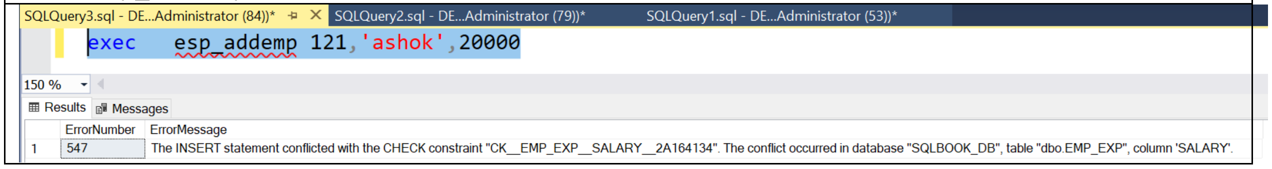
```
CREATE TABLE EMP_EXP
(
EMPID INT PRIMARY KEY,
EMPNAME VARCHAR(100) UNIQUE,
SALARY FLOAT CHECK(SALARY > 25000)
)
```

CREATE A TABLE AND STORED PROCEDURE AND CALL IT

```
CREATE procedure [dbo].[esp_addemp]
(
@id int,
@name nvarchar(200),
@salary int
)
as
begin
begin try
insert  into  EMP_EXP  values(@id,@name,@salary)
End try
Begin catch
SELECT ERROR_NUMBER()AS ErrorNumber,ERROR_MESSAGE()AS ErrorMessage
End catch
End
```

CALLING THE PROCEDURE WITH A VALUE LESS THAN 25000 WILL THROW AN ERROR SHOWN BELOW

```
exec   esp_addemp 1,'amit',20000
```



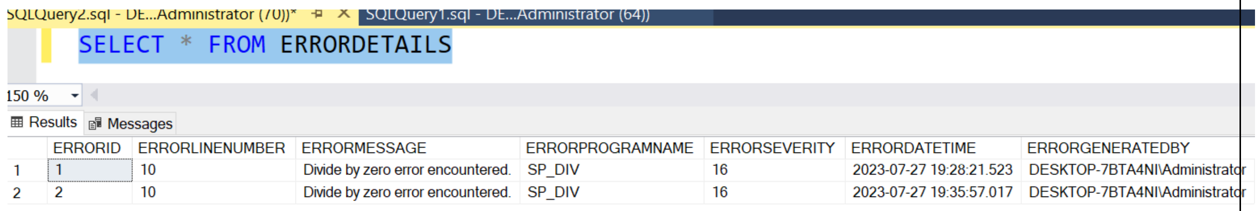
```
exec    esp_addemp 121,'ashok',20000
```

```
   ErrorNumber  ErrorMessage
1  547          The INSERT statement conflicted with the CHECK constraint "CK__EMP_EXP__SALARY__2A164134". The conflict occurred in database "SQLBOOK_DB", table "dbo.EMP_EXP", column 'SALARY'.
```

Logging Errors

```
CREATE TABLE ERRORDETAILS
(
ERRORID INT IDENTITY(1,1),
ERRORLINENUMBER INT,
ERRORMESSAGE VARCHAR(4000),
ERRORPROGRAMNAME VARCHAR(200),
ERRORSEVERITY INT,
ERRORDATETIME DATETIME,
ERRORGENERATEDBY VARCHAR(100)
)
CREATE PROCEDURE SP_DIV
(
@n1 int,
@n2 int
)
AS
BEGIN

BEGIN TRY
declare @t int = @n1 / @n2
print @t
END TRY
BEGIN CATCH
 PRINT 'AN ERROR OCCURED, PLEASE CHECK THE INPUT VALUES'
 SELECT ERROR_LINE() AS LINENUMBER,ERROR_MESSAGE(),ERROR_PROCEDURE() AS PROGRAMNAME,
 ERROR_SEVERITY() AS SEVERITY

 INSERT INTO ERRORDETAILS
 SELECT ERROR_LINE() AS LINENUMBER,ERROR_MESSAGE(),ERROR_PROCEDURE() AS PROGRAMNAME,
 ERROR_SEVERITY() AS SEVERITY,GETDATE(),SUSER_NAME()
END CATCH

END
```

```
EXEC SP_DIV 50,0

SELECT * FROM ERRORDETAILS
```


SQLQuery2.sql - DE...Administrator (70))*
SQLQuery1.sql - DE...Administrator (64))
150 %
Results
Messages

```
SELECT * FROM ERRORDETAILS
```

	ERRORID	ERRORLINENUMBER	ERRORMESSAGE	ERRORPROGRAMNAME	ERRORSEVERITY	ERRORDATETIME	ERRORGENERATEDBY
1	1	10	Divide by zero error encountered.	SP_DIV	16	2023-07-27 19:28:21.523	DESKTOP-7BTA4NI\Administrator
2	2	10	Divide by zero error encountered.	SP_DIV	16	2023-07-27 19:35:57.017	DESKTOP-7BTA4NI\Administrator

TRY…CATCH constructs do not trap the following conditions:

- Warnings or informational messages that have a severity of 10 or lower.
- Errors that have a severity of 20 or higher that stops the SQL Server Database Engine task processing for the session. If an error occurs that has severity of 20 or higher and the database connection is not disrupted, TRY…CATCH will handle the error.
- Attentions, such as client-interrupt requests or broken client connections.
- When the session is ended by a system administrator by using the KILL statement.

The following types of errors are not handled by a CATCH block when they occur at the same level of execution as the TRY…CATCH construct:

- Compile errors, such as syntax errors, that prevent a batch from running.
- Errors that occur during statement-level recompilation, such as object name resolution errors that occur after compilation because of deferred name resolution.

These errors are returned to the level that ran the batch, stored procedure, or trigger.

If an error occurs during compilation or statement-level recompilation at a lower execution level (for example, when executing sp_executesql or a user-defined stored procedure) inside the TRY block, the error occurs at a lower level than the TRY…CATCH construct and will be handled by the associated CATCH block.

The following example show how an object name resolution error generated by a SELECT statement is not caught by the TRY…CATCH construct, but is caught by the CATCH block when the same SELECT statement is executed inside a stored procedure.

```
BEGIN TRY
   -- Table does not exist; object name resolution
   -- Error not caught.
   SELECT  *  FROM NonexistentTable;
END TRY
BEGIN CATCH
   SELECT  ERROR_NUMBER() AS ErrorNumber ,ERROR_MESSAGE() AS ErrorMessage;
END CATCH
```

The error is not caught and control passes out of the TRY…CATCH construct to the next higher level.

Running the SELECT statement inside a stored procedure will cause the error to occur at a level lower than the TRY block. The error will be handled by the TRY…CATCH construct.

```
-- Create a stored procedure that will cause an  -- object resolution error.
CREATE PROCEDURE usp_ExampleProc
AS
   SELECT * FROM  NonexistentTable;
GO
BEGIN TRY
   EXECUTE  usp_ExampleProc;
END TRY
BEGIN CATCH
   SELECT  ERROR_NUMBER() AS ErrorNumber ,ERROR_MESSAGE() AS ErrorMessage;
END CATCH;
```

ADDING CUSTOM ERROR MESSAGES

SP_addmessage: We use the SP_admessage Stored Procedure to define a User Defined Custom Error Message.

This Stored Procedure adds a record to the sys.message system view.
A User Defined message should have a message number of 50000 or higher with a severity of 1 to 25.

Syntax:
sp_addmessage [@msgnum =] msg_id ,[@severity =] severity ,[@msgtext =] 'msg'[, [@lang =] language'][, [@with_log =] 'with_log']
[, [@replace =] 'replace']

Here mcg_id is the id of the message which can be between 50000 and 2147483647.

- The severity is the level of the message which can be between 1 and 25.
- For User Defined messages we can use it a value of 0 to 19.
- The severity level Between 20 to 25 can be set by the administrator. Severity levels from 20 through 25 are considered fatal.
- The actual error message is "msg", which uses a data type of nvarchar (255). The maximum characters limit is 2,047. Any more than that will be truncated.

The language is used if you want to specify any language. Replace is used when the same message number already exists, but you want to replace the string for that ID, you have to use this parameter.

RAISERROR

The RAISERROR statement generates an error message by either retrieving the message from the sys.messages catalog view or constructing the message string at runtime.

It is used to invoke the User Defined error message.
First we create a User Defined error message using sp_addmessage and after that we invoke that by the use of RAISERROR.

SYNTAX:
RAISERROR ({ msg_id }{ ,severity ,state }[,argument [,...n]])[WITH option [,...n]]
Example:

```
EXEC   sp_addmessage   500021, 10,'THis error message is created by ENOSIS LEARNING'
Go
RAISERROR (500021, 10, 1)
```

Replacement of Message.

```
EXEC sp_addmessage 500021, 10,'Previous error message is replaced by ENOSIS LEARNING',
@lang='us_english', @with_log='false', @replace='replace'
GO
RAISERROR (500021, 10, 1)
```

Altering the message:

Exec sp_altermessage 500021, @parameter='with_log', @parameter_value='true'
Output:
Dropping the message:

Exec sp_dropmessage

500021
Output: USE master: Adding error messages

```
EXEC  sp_addmessage 50001, 1,N'This message is not that big of a deal.
This is not caught by error handling, and prints this message to the screen.';
EXEC  sp_addmessage 50002, 16,N'This actually causes an error,
and is caught by error-handling';
EXEC  sp_addmessage 50003, 20,N'This causes an error, and stops any further
processing.  This is not caught by error handling.';
```

Using custom error messages
Now that my custom error messages are defined, I can use them inside my database engine. To invoke these errors, I'll use the RAISERROR TSQL construct. RAISERROR accepts an error number, a severity level, and a state number.

Select * from sys.messages   : To view the error messages.

The following snippet uses RAISERROR inside of a TRY...CATCH construct. I am including the WITH LOG option of the RAISERROR statement to write the error message to the application log so that I can review it later if necessary. (This particular error does not invoke the CATCH block due to the severity of the error.

```
BEGIN TRY
        RAISERROR (50001, 1, 1) WITH LOG
END TRY
BEGIN CATCH
        SELECT ERROR_MESSAGE (), ERROR_NUMBER ()
END CATCH
```

This statement invokes the second custom error message I define above. This message has a defined severity of 16, which will get caught by my CATCH statement. These types of error messages are some of the more commonly seen messages inside the SQL Server database engine.

```
BEGIN TRY
        RAISERROR (50002, 16, 1) WITH LOG
END TRY
BEGIN CATCH
        SELECT ERROR_MESSAGE (), ERROR_NUMBER ()
END CATCH
```

```
Msg 2745, Level 16, State 2, Line 2
Process ID 51 has raised user error 50003, severity 20. SQL Server is terminating this process.

Msg 50003, Level 20, State 1, Line 2
This causes an error, and stops any further processing.  This is not caught by error handling.

Msg 0, Level 20, State 0, Line 0
A severe error occurred on the current command.  The results, if any, should be discarded.
```

The error is marked as so severe that if I were to run the same statement again, I receive the following error:

```
Msg 233, Level 20, State 0, Line 0
A transport-level error has occurred when receiving results from the server. (Provider: Shared Memory Provider, error: 0 - No process is on the other end of the pipe.)
```

This error states that the connection has been terminated.

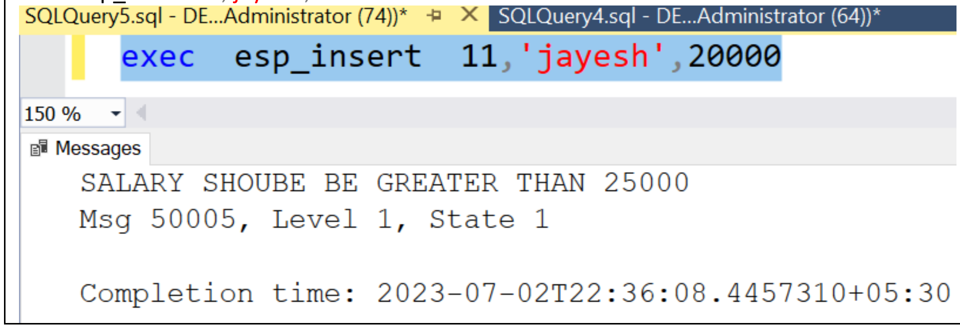
Example: (Create a stored procedure for inserting a value and if value is less than 25000 will call the custom error code though procedure)

First create a custom error code and then create a procedure and call it.

```
EXEC  sp_addmessage 50005, 1,'SALARY SHOUBE BE GREATER THAN 25000';
```

```
CREATE procedure [dbo].[esp_insert]
( @id int, @name nvarchar(200), @salary int )
as
begin
begin try
if (@salary <= 25000)
begin
RAISERROR (50005, 1, 1)WITH LOG
end
else
```

```
begin
insert  into  EMP_EXP  values(@id,@name,@salary)
end
End try
Begin catch
SELECT ERROR_NUMBER()AS ErrorNumber,ERROR_MESSAGE()AS ErrorMessage
End catch
end

exec  esp_insert  11,'jayesh',20000
```



SQLQuery5.sql - DE...Administrator (74))*
SQLQuery4.sql - DE...Administrator (64))*
150 %
Messages

```
exec  esp_insert  11,'jayesh',20000
```

```
SALARY SHOUBE BE GREATER THAN 25000
Msg 50005, Level 1, State 1

Completion time: 2023-07-02T22:36:08.4457310+05:30
```

LESSON 2: MANAGING TRANSACTIONS

- Transaction is a group of commands that is treated and executed as a single unit.
- If the one of the commands in the transaction fails, all the commands fail, and any data that was modified in the database is rolled back.
- Transaction maintain the integrity of data in a database.

For processing the transaction,steps are as follow :

1. Begin the transaction.
2. Process database commands
3. Check for error

```
If error occurred,
    Rollback the transaction
Else
    Commit the transaction
```

Syntax to create transaction

BEGIN TRY
BEGIN TRAN
STATEMENTS
COMMIT TRAN
END TRY
BEGIN CATCH
ROLLBACK TRAN
END CATCH

EXAMPLE OF TRANSACTION	
WITHOUT TRANSACTION	WITH TRANSACTION
CREATE TABLE EMP _T (EMPID INT PRIMARY KEY, EMPNAME VARCHAR(100), SAL FLOAT) Begin try **insert into EMP _T values(1,'Anil',51000)** **insert into EMP _T values(2,'Sunil',61000)** **insert into EMP _T values(1,'Kapil',81000)** insert into EMP _T values(16,'Jay',17000) End try Begin Catch Select ERROR_MESSAGE() AS **ErrorMessage** End Catch	Begin try BEGIN TRAN insert into EMP _T values(1,'Anil',51000) insert into EMP _T values(2,'Sunil',61000) insert into EMP _T values(1,'Kapil',81000) insert into EMP _T values(16,'Jay',17000) COMMIT TRAN End try Begin Catch ROLLBACK TRAN Select ERROR_MESSAGE() AS **ErrorMessage** End Catch
It will insert the record 1 and 2 and go to the error block.	**It will not commit any records since there is a error it will go the catch block and revert the transaction**

```
Begin try
Begin tran
insert into  EMP _T values(3,'ravi',56000)
insert into  EMP _T values(4,'sandeep',66000)
save transaction s1
insert into  EMP _T values(5,'anil',76000)
insert into  EMP _T values(4,'rajdeep',36000)
insert into  EMP _T values(6,'amitabh',26000)
commit tran
end try
begin catch
rollback transaction s1
            SELECT ERROR_NUMBER() AS ErrorNumber,ERROR_SEVERITY() AS ErrorSeverity
,ERROR_STATE() AS  ErrorState,ERROR_PROCEDURE() AS ErrorProcedure
,ERROR_LINE() AS ErrorLine,ERROR_MESSAGE() AS ErrorMessage
End catch
```

Additional Transaction Options

Additional remaining options for transactions are available that are

- **Savepoints:** These are locations within transactions that you can use to roll back a selective subset of work.
    - You can define a savepoint by using the SAVE TRANSACTION <savepoint name> command.

- The ROLLBACK statement must reference the savepoint. Otherwise, if the statement is unqualified, it will roll back the entire transaction.

Keep in mind that SQL updates and rollbacks generally are expensive operations. So savepoints are useful only in situations where errors are unlikely and checking the validity of an update beforehand is relatively costly.

The syntax for typical server savepoint usage (where SAVE TRANSACTION permissions default to any valid user) is:

```
 SAVE TRANSACTION SavepointName
IF @@error= some Error
BEGIN
 ROLLBACK TRANSACTION SavepointName
COMMIT TRANSACTION
END
```

- **Cross-database transactions:** A transaction may span two or more databases on a single SQL Server instance without any additional work on the user's part.
    - SQL Server preserves the ACID properties of cross-database transactions without any additional considerations.
    - However, there are limitations on database mirroring when using cross-database transactions. A cross-database transaction may not be preserved after a failover of one of the databases.
- **Distributed transactions**: It is possible to make a transaction span more than one server, by using a linked server. In that case, the transaction is known as a distributed (as opposed to local) transaction.
    - After a transaction spans multiple servers by using a linked server, the transaction is considered a distributed transaction and SQL Server invokes the Distributed Transaction Coordinator (MSDTC).
    - A transaction restricted to one database, or to a cross-database transaction, is considered a local transaction, as opposed to a distributed transaction, which crosses SQL Server instance boundaries.

CHAPTER 9: SUMMARY / CHEATSHEET

Error Handling is used to handle error generated in a program due to external conditions like wrong data, network failure etc. The error needs to be handled properly else the program will terminate abnormally.
Transaction is a group of commands that is treated and executed as a single unit.If the one of the commands in the transaction fails, all the commands fail, and any data that was modified in the database is rolled back.

CHAPTER 9:QUESTIONS

Explain try and catch?
Define the usage of the following functions. **a. ERROR_NUMBER()** **b. ERROR_SEVERITY()** **c. ERROR_STATE()** **d. ERROR_PROCEDURE()** **e. ERROR_LINE()** **f. ERROR_MESSAGE()**
Can we use two or more catch block with associated one try block?
What is transaction?
What are the steps to write transaction?

CHAPTER 10: TABLE TYPES

This chapter covers two main topics. It starts with a lesson about the cursor. It then continues with a lesson about the use of temporary objects like local temporary tables and table variables, explaining when you should consider using each kind.

Lessons in this chapter:

- Lesson 1: Temporary Tables
- Lesson 2: Table Variable
- Lesson 3: CTE
- Lesson 4: Subquery

LESSON 1: TEMPORARY TABLES

SQL Server provides the concept of **temporary table** which helps the developer in a great way.

Temporary table are temporary in nature and are dropped automatically as soon as the sessions closes or disconnects. These tables can be created at **runtime** and can do the all kinds of operations that one normal table can do. But, based on the table types, the scope is limited. These tables are created inside tempdb database.

SQL Server provides two types of temp tables based on the behavior and scope of the table. These are:

LOCAL TEMP TABLE	GLOBAL TEMP TABLE
Local temp table are available only in the session of user.	Local temp table are available other users till the session of the user who have created the table is alive.

Local Temp Table

Local temp tables are available only in the current session of the user. and they are automatically deleted when the user disconnects or closes the session.
Local temporary table name is stared with hash ("#") sign.

The **syntax** given below is used to create a **local Temp table**:

```
CREATE TABLE #LocalTempTable
(
UserID int,
UserName varchar (50),
UserAddress varchar (150)
)
```

The above script will create a temporary table in tempdb database. We can insert or delete records in the temporary table similar to a general table like:

Insert into #LocalTempTable values (1, 'Abhijit', 'India');

Now select records from that table:

Select* from #LocalTempTable

After execution of all these statements, if you close the query window and again execute "Insert" or "Select" Command, it will throw the following error:

Msg 208, Level 16, State 0, Line 1 Invalid object name '#LocalTempTable'.

This is because the **scope of Local Temporary table** is only bounded with the current connection of current user.

Global Temp Table

Global Temporary tables name starts with a double hash ("##"). Global Temp tables and also available to other users till the session in which the table is created. Once the session disconnects or closes then the global temp table is dropped automatically. We need to put "##" with the name of Global temporary tables. Below is the syntax for creating a Global Temporary Table:

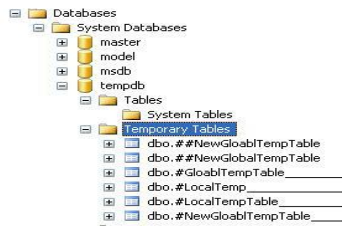
```
CREATE TABLE ##NewGlobalTempTable
(
UserID int, UserName varchar (50),  UserAddress varchar (150)
)
```

The above script will create a temporary table in **tempdb database**. We can insert or delete records in the temporary table similar to a general table like:

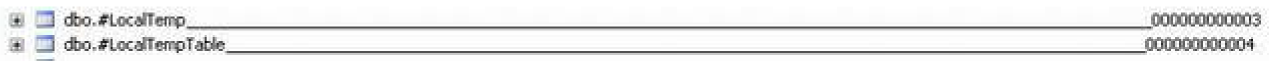
Insert into **##NewGlobalTempTable** values (1, 'Abhijit', 'India');
Now select records from that table: Select * from ##NewGlobalTempTable

Global temporary tables are visible to all SQL Server connections. When you create one of these, all the users can see it.

Storage Location of Temporary Table : Temporary tables are stored inside the **Temporary Folder of tempdb**. Whenever we create a temporary table, it goes to **Temporary folder of tempdb database**.



Now, if we deeply look into the name of Local Temporary table names, a 'dash' is associated with each and every table name along with an ID. Have a look at the image below:



```
dbo.#LocalTemp______________________________________________000000000003
dbo.#LocalTempTable_________________________________________000000000004
```

SQL server does all this automatically; we do not need to worry about this; we need to only use the table name.

When to Use Temporary Tables?
Below are the scenarios where we can use temporary tables:

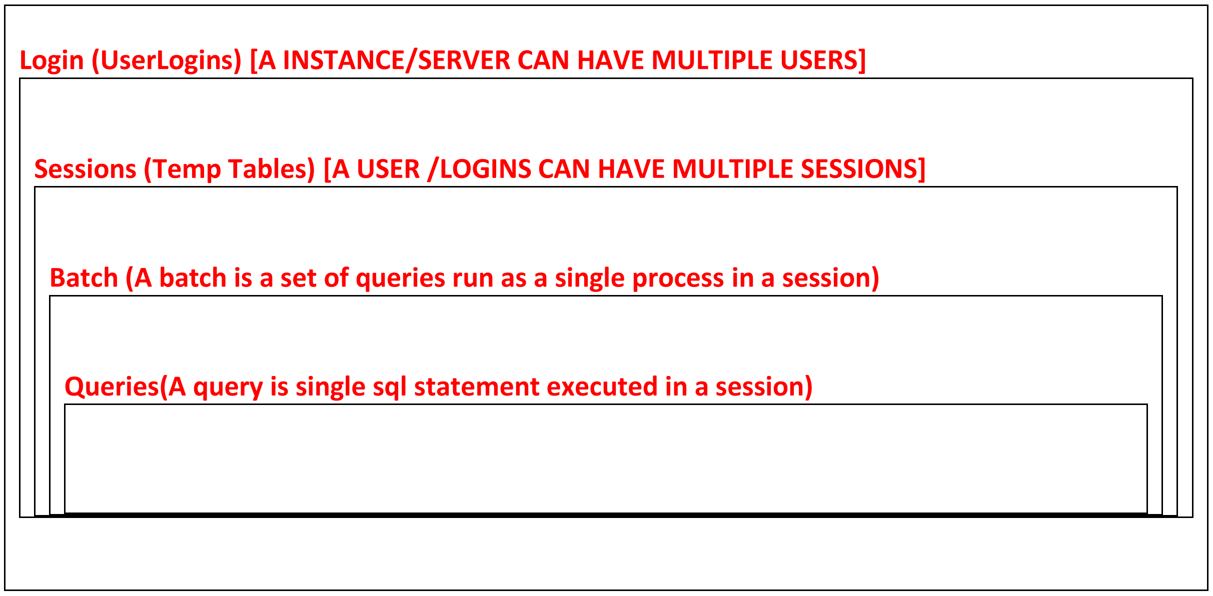
- When we are doing large number of row manipulation in stored procedures.
- This is useful to replace the cursor. We can store the result set data into a temp table, then we can manipulate the data from there.
- When we are having a complex join operation.

Points to Remember Before Using Temporary Tables

- **Temporary table created on tempdb of SQL Server.** This is a separate database. So, this is an additional overhead and can causes performance issues.
- Number of rows and columns need to be as minimum as needed.
- Tables need to be deleted when they are done with their work.

HIERARCHY STRUCTURE OF INSTANCES,LOGINS,SESSIONS,BATCH AND QUERIES

Instance (SERVER)


Login (UserLogins) [A INSTANCE/SERVER CAN HAVE MULTIPLE USERS]
Sessions (Temp Tables) [A USER /LOGINS CAN HAVE MULTIPLE SESSIONS]
Batch (A batch is a set of queries run as a single process in a session)
Queries(A query is single sql statement executed in a session)

LESSON 2: TABLE VARIABLE

Table Variable is a variable which can stored a table as a variable. Its scope is limited to a batch.

We can do all operations on a table variable inside the batch.

Alternative of Temporary table is the **Table variable** which can do all kinds of operations that we can perform in Temp table. Below is the syntax for using **Table variable**.

```
SYNTAX
CREATE @table_variable TABLE
(
COLNAME1 DATATYPE,
COLNAME2 DATATYPE,
COLNAME3 DATATYPE
)
Declare @USERDETAIL TABLE
 (
UserID  int,
UserName  varchar(50),
UserAddress  varchar(150)
)
The below scripts are used to insert and read the records for Tablevariables:
insert into @USERDETAIL  values ( 1, 'Abhijit', 'India');

Now select records from that table variable:
 select *  from  @USERDETAIL
```

When to Use Table Variable over Temp Table

- Table variable is always useful **for less data**.
- If the result set returns a large number of records, we need to go for temp table.

LESSON 3: CTE

CTE (Common Table Expression)

A common table expression (CTE) can be thought of as a temporary result set that is defined within the execution scope of a single SELECT, INSERT, UPDATE, DELETE, or CREATE VIEW statement. A CTE is similar to a derived table in that it is not stored as an object and lasts only for the duration of the query. Unlike a derived table, a CTE can be self-referencing and can be referenced multiple times in the same query.

Syntax

```
WITH EXPRESSION_NAME (COLUMNNAME1, COLUMNNAME2)
(
CTE QUERY DEFINATION
)
SELECT COLUMNNAME FROM EXPRESSION_NAME
```

EXAMPLE OF CTE

```
WITH cte_emp(EMPID,EMPNAME,SALARY,TAX)
(
SELECT EMPID,EMPNAME,SALARY,TAX FROM EMPLOYEE
)
SELECT * FROM cte_emp
```

```
WRITE A CTE TO FINE THE TOP 5 EMPLOYEES
WITH EMP_TOP5(EMPID,EMPNAME,SALARY,TAX)
(
SELECT TOP 5 EMPID,EMPNAME,SALARY,TAX FROM EMPLOYEE ORDER BY SALARY DESC
)
SELECT * FROM EMP_TOP5
```

Advantages

- Can be used to create a recursive query.
- Can be substituted for a view.
- Allow grouping by a column which might be derived from a scalar subset.
- Can reference itself multiple times.

Disadvantages

- CTE's members cannot use the following clauses of keywords Distinct, Group By, Having, Top, Joins limiting by this type of the queries that can be created and reducing their complexity.
- The Recursive member can refer to the CTE only once.
- Table Variables and CTE's cannot be passed as parameters in stored procedures.

LESSON 4: SUBQUERY

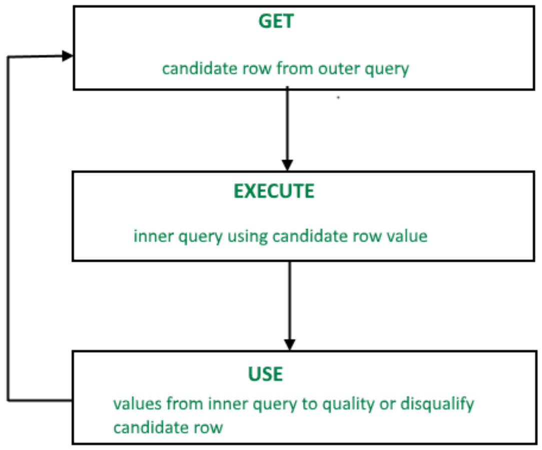
When a query is included inside another query, the Outer query is known as Main Query, and Inner query is known as Subquery.

- **Nested Query –**
  In Nested Query, Inner query runs first, and only once. Outer query is executed with result from Inner query.Hence, Inner query is used in execution of Outer query.
  Example – Find details of customers who have ordered.

  ```
  SELECT * FROM Customers WHERE
  CustomerID IN (SELECT CustomerID FROM Orders);
  ```

- **Correlated Query –**
  In Correlated Query, Outer query executes first and for every Outer query row Inner query is executed. Hence, Inner query uses values from Outer query.

Correlated subqueries are used for row-by-row processing. Each subquery is executed once for every row of the outer query.


GET
candidate row from outer query
EXECUTE
inner query using candidate row value
USE
values from inner query to quality or disqualify candidate row

- A correlated subquery is evaluated once for each row processed by the parent statement. The parent statement can be a **SELECT**, **UPDATE**, or **DELETE** statement.

```
SELECT column1, column2, ....
FROM table1 outer
WHERE column1 operator (SELECT column1, column2
                 FROM table2
                 WHERE expr1 =  outer.expr2);
```

- A correlated subquery is one way of reading every row in a table and comparing values in each row against related data. It is used whenever a subquery must return a different result or set of results for each candidate row considered by the main query. In other words, you can use a correlated subquery to answer a multipart question whose answer depends on the value in each row processed by the parent statement.
- With a normal nested subquery, the inner **SELECT** query runs first and executes once, returning values to be used by the main query.
- A correlated subquery, however, executes once for each candidate row considered by the outer query. In other words, the inner query is driven by the outer query.

NOTE : You can also use the **ANY** and **ALL** operator in a correlated subquery. EXAMPLE

```
SELECT empname, salary, department  FROM employees e  WHERE salary > (SELECT AVG(salary) FROM
employees  WHERE department = e.department);
```

```
SELECT empname, salary, department
 FROM employees e
 WHERE salary <
            (SELECT AVG(salary)
             FROM employees WHERE department = e.department);
```

Difference between Nested Query, Correlated Query and Join Operation

Parameters	Nested Query	Correlated Query	Join Operation
Definition	In Nested query, a query is written inside another query and the result of inner query is used in execution of outer query.	In Correlated query, a query is nested inside another query and inner query uses values from outer query.	Join operation is used to combine data or rows from two or more tables based on a common field between them.INNER JOIN, LEFT JOIN, RIGHT JOIN, FULL JOIN are different types of Joins.
Approach	Bottom up approach i.e. Inner query runs first, and only once. Outer query is executed with result from Inner query.	Top to Down Approach i.e. Outer query executes first and for every Outer query row Inner query is executed.	It is basically cross product satisfying a condition.
Dependency	Inner query execution is not dependent on Outer query.	Inner query is dependent on Outer query.	There is no Inner Query or Outer Query. Hence, no dependency is there.
Performance	Performs better than Correlated Query but is slower than Join Operation.	Performs slower than both Nested Query and Join operations as for every outer query inner query is executed.	By using joins we maximize the calculation burden on the database but joins are better optimized by the server so the retrieval time of the query using joins will almost always be faster than that of a subquery.

Nested Subqueries Versus Correlated SUBQUERIES

With a normal nested subquery, the inner **SELECT** query runs first and executes once, returning values to be used by the main query.

A correlated subquery, however, executes once for each candidate row considered by the outer query. In other words, the inner query is driven by the outer query.
NOTE : You can also use the **ANY** and **ALL** operator in a correlated subquery.

EXAMPLE of Correlated Subqueries :

Find all the employees who earn more than the average salary in their department.

```
SELECT last_name, salary, department_id
 FROM employees outer
 WHERE salary > (SELECT AVG(salary)
                FROM employees
                WHERE department_id =                    outer.department_id);
```

Other use of correlation are in **UPDATE** and **DELETE**

CORRELATED UPDATE :

```
UPDATE table1 alias1
 SET column = (SELECT expression
              FROM table2 alias2
              WHERE alias1.column =
                   alias2.column);
```

Use a correlated subquery to update rows in one table based on rows from another table.

CORRELATED DELETE :

```
DELETE FROM table1 alias1
 WHERE column1 operator
              (SELECT expression
               FROM table2 alias2
               WHERE alias1.column = alias2.column);
```

CHAPTER 10: SUMMARY / CHEATSHEET

Temp tables are used to create tables which are temporary and which will be dropped automatically as soon as the session is closed.
Temp Tables are physically created in the Tempdb database. These tables act as the normal table and also can have constraints, index like normal tables.
CTE is a named temporary result set which is used to manipulate the complex sub-queries data. This exists for the scope of statement. This is created in memory rather than Tempdb database. You cannot create any index on CTE.
Table Variable acts like a variable and exists for a particular batch of query execution. It gets dropped once it comes out of batch. This is also created in the Tempdb database but not the memory.

SCOPE & COMPARISION OF TEMPORARY OBJECTS

	Local Temp Tables	Global Temp Tables	CTE	TABLE VARIABLES	Derived Table/ SUBQUERY
Scope	Single user-session	Multiuser-session	Withinbatch	Within the batch	Within the query
Operations	Insert,update, delete,select	Insert,update, delete,select	Only select	Insert,update, delete,select	Only select

Deciding which type of table to use

Things to be aware of when using temp tables versus table variables versus derived tables:

- Temp Tables are created in the SQL Server TEMPDB database and therefore require more IO resources and locking. Table Variables and Derived Tables are created in memory.
- Temp Tables will generally perform better for large amounts of data that can be worked on using parallelism whereas Table Variables are best used for small amounts of data (I use a rule of thumb of 100 or less rows) where parallelism would not provide a significant performance improvement.
- You cannot use a stored procedure to insert data into a Table Variable or Derived Table. For example, the following will work: INSERT INTO #MyTempTable EXEC dbo.GetPolicies_sp whereas the following will generate an error: INSERT INTO @MyTableVariable EXEC dbo.GetPolicies_sp.
- Derived Tables can only be created from a SELECT statement but can be used within an Insert, Update, or Delete statement.
- In order of scope endurance, Temp Tables extend the furthest in scope, followed by Table Variables, and finally Derived Tables.

EXECUTE THE SQL QUERIES AND CREATE An EMPLOYEE TABLE IN THE DATABASE.

```
CREATE TABLE [dbo].[EMPLOYEES](
	[EMPID] [int] IDENTITY(1,1) NOT NULL,
	[EMPNAME] [varchar](100) NULL,
	[SALARY] [float] NULL,
	[TAX] [float] NULL,
	[DOJ] [datetime] NULL,
	[DEPT] [varchar](100) NULL,
	[DESG] [varchar](100) NULL,
	[DEPTMANAGERNAME] [varchar](100) NULL,
	[DEPTLOCATION] [varchar](100) NULL,
 CONSTRAINT [pk_empid] PRIMARY KEY CLUSTERED
(
	[EMPID] ASC
)) ON [PRIMARY]

GO
SET IDENTITY_INSERT [dbo].[EMPLOYEES] ON

GO
INSERT [dbo].[EMPLOYEES] VALUES (1, N'AKASH', 70000, 2000, CAST(N'2017-05-20' AS DateTime), N'IT',
N'SOFTWARE DEVELOPER', N'SURESH', N'PUNE')
INSERT [dbo].[EMPLOYEES]  VALUES (2, N'AKSHAY', 69000, 1900, CAST(N'2017-07-14' AS DateTime), N'IT',
N'ARCHITECT', N'SURESH', N'PUNE')
INSERT [dbo].[EMPLOYEES] VALUES (3, N'ASHWIN', 77000, 2100, CAST(N'2017-09-14' AS DateTime), N'IT',
N'ARCHITECT', N'SURESH', N'PUNE')
INSERT [dbo].[EMPLOYEES] VALUES (4, N'CHINMAY', 88000, 2900, CAST(N'2017-09-10' AS DateTime), N'HR',
N'MANAGER', N'ANITA', N'MUMBAI')
INSERT [dbo].[EMPLOYEES]  VALUES (5, N'DEBAYAN', 65000, 2100, CAST(N'2017-10-18' AS DateTime), N'HR',
N'EXECUTIVE', N'ANITA', N'MUMBAI')
INSERT [dbo].[EMPLOYEES] VALUES (6, N'MANGESH', 99000, 2900, CAST(N'2017-10-10T00:00:00.000' AS
DateTime), N'HR', N'MANAGER', N'ANITA', N'MUMBAI')
INSERT [dbo].[EMPLOYEES] VALUES (7, N'NIKHIL', 56000, 2100, CAST(N'2017-07-18T00:00:00.000' AS DateTime),
N'HR', N'EXECUTIVE', N'ANITA', N'MUMBAI')
INSERT [dbo].[EMPLOYEES] VALUES (10, N'SANDEEP', 45000, 460, CAST(N'2017-07-25T00:00:00.000' AS
DateTime), N'IT', N'SOFTWARE DEVELOPER', N'SURESH', N'PUNE')
INSERT [dbo].[EMPLOYEES] VALUES (11, N'NIRAJ', 48000, 460, CAST(N'2017-07-25T00:00:00.000' AS DateTime),
N'ADMIN', N'MANAGER', N'DELHI', NULL)
INSERT [dbo].[EMPLOYEES] VALUES (12, N'VIVEL', 89000, 2000, CAST(N'2017-01-20T00:00:00.000' AS
DateTime), N'SECURITY', N'MANAGER', N'KOLKATTA', NULL)
INSERT [dbo].[EMPLOYEES]  VALUES (13, N'DILLIP', 47000, 400, CAST(N'2017-09-25T00:00:00.000' AS DateTime),
N'IT', N'TRAINER', N'SURESH', N'PUNE')
SET IDENTITY_INSERT [dbo].[EMPLOYEES] OFF
GO
```

EMPID	EMPNAME	SALARY	TAX	DOJ	DEPT	DESG	DEPT MANAGERNAME	DEPTLOCATION
1	AKASH	70000	2000	20-05-2017	IT	SOFTWARE DEVELOPER	SURESH	PUNE
2	AKSHAY	69000	1900	14-07-2018	IT	ARCHITECT	SURESH	PUNE
3	ASHWIN	77000	2100	14-09-2017	IT	ARCHITECT	SURESH	PUNE
4	CHINMAY	88000	2900	10-09-2017	HR	MANAGER	ANITA	MUMBAI
5	DEBAYAN	65000	2100	18-10-2017	HR	EXECUTIVE	ANITA	MUMBAI
6	MANGESH	99000	2900	10-10-2017	HR	MANAGER	ANITA	MUMBAI
7	NIKHIL	56000	2100	18-07-2018	HR	EXECUTIVE	ANITA	MUMBAI
10	SANDEEP	45000	460	25-07-2017	IT	SOFTWARE DEVELOPER	SURESH	PUNE
11	NIRAJ	48000	460	25-07-2017	ADMIN	MANAGER	DELHI	NULL
12	VIVEL	89000	2000	20-01-2019	SECURITY	MANAGER	KOLKATTA	NULL
13	DILLIP	47000	400	25-09-2017	IT	TRAINER	SURESH	PUNE

WRITE ANSWERS TO THE QUERIES USING THE ABOVE TABLE

1. Write a CTE to find the details of employee having the 4th Highest salary
2. Write a Subquery to find the details of employee having the 4th Highest salary
3. Write a query to find names of employee of the each department whose salary is greater than the average salary of the department.
4. Write a query to find names of employee of the each department whose salary is less than the average salary of the department.

5. Write a query to find the number of employees recruited in each department in different years.(Use Pivot if required)

DEPT / YEAR	2017	2018	2019
IT			
HR			
ADMIN			

6. Write a stored procedure which will create two table from the single table (One is the department table and second is the employee table and create a relationship between them). [Hint : Use temp tables if required)
7. Write a stored to create three tables from the single table and also move the data to each tables. (One is department, employee and salary details)

DimDept	DeptId(Primary Key)
DimEmployee	EmpId(PKey) DeptId(FKey)
FactSalary	EmpId(FKey)

CHAPTER 10: QUESTIONS

What is temp table?
What are different types of temp table?
Where is temp tables stored?
Why CTE is used in SQL Server?
Is CTE faster than temp table?
Which is better CTE or temp table?
Advantages and disadvantages of CTE?
Can we use CTE multiple times?
Which is better CTE or subquery?
Can we use more than one CTE in a single select query?
What is difference between CTE and view?

CHAPTER 11

CHAPTER 11: INDEXES

Indexes are necessary for good performance of your databases. In order to create appropriate indexes, you need to understand how Microsoft SQL Server stores data in tables and indexes, and how it then accesses this data.

Lessons in this chapter:

1. Lesson 1: Indexes
2. Lesson 2: Creating an index

LESSON 1: INDEXES

An index is an on-disk structure associated with a table or views that speed retrieval of rows from the table or view. An index contains keys built from one or more columns in the table or view. These keys are stored in a structure (B-tree=) that enables SQL Server to find the row or rows associated with the key values quickly and efficiently.

An index is a database object created and maintained by DBMS. It is essentially a list of the contents of a column or group of columns. Indexes are ordered so that extremely first search can be computed through them to find data.
Why Use an Index? Use of SQL server indexes provide many facilities such as:

- Rapid access of information
- Efficient access of information
- Enforcement of uniqueness constraints

Correct use of indexes can improve the query performance.

Types of Indexes

SQL Server has two major types of indexes:

1. **Clustered Index**
2. **Non-Clustered Index**

The index type refers to the way the index is stored internally by SQL Server. So a table or view can contain the two types of indexes.

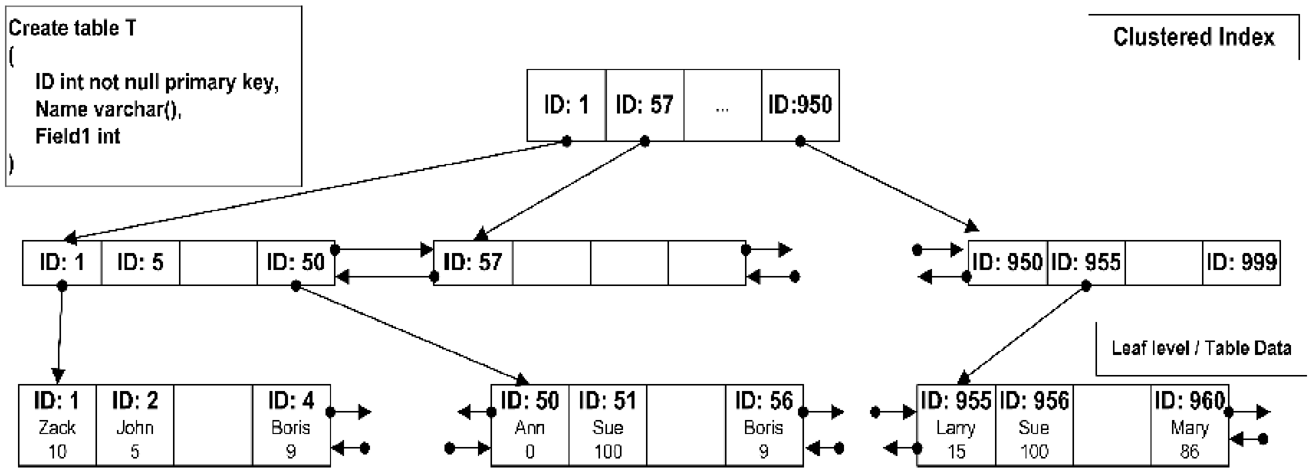
Clustered Index

An index defined as being clustered, defines the physical order that the data in a table is stored. Only one cluster can be defined per table. So it can be defined as:

Clustered indexes sort and store the data rows in the table or view based on their key values. These are the columns included in the index definition. There can be only one clustered index per table, because the data rows themselves can be sorted in only one order.

The only time the data rows in a table are stored in sorted order is when the table contains a clustered index. When a table has a clustered index, the table is called a clustered table. If a table has no clustered index, its data rows are stored in an unordered structure called a heap.

Internals of Clustered Index



The image above shows a small table with ID as the primary key (and clustered index). As the side note, SQL Server creates clustered index on the primary key field by default.

Leaf level (the bottom one) contains actual table data sorted by ID. As you can see, the data pages are linked into the double-linked list so SQL Server can scan the index in both directions.

Levels above the leaf level called "intermediate levels". Every index row on those levels points to the separate data pages in the level below. At the top level (root level) there is only one page. There could be several intermediate levels based on the table size. But the top root level of the index always has 1 data page.

So let's see how it actually works: Assuming you want to select the record with ID = 50. SQL Server start from the root level and find that first row contains ID=1 and the second row contains ID=57. It means that the row with ID=50 would be located on the data page started with ID=1 on the next level of the index. So the next step is analyzing **the first data page** on the intermediate level which contains IDs from 1 to 50. So SQL Server finds the row with ID=50 and jump on the leaf level page with the actual data.

Note :SQL Server does not require you to create the clustered indexes - tables without such indexes called heap tables.

Non-Clustered index

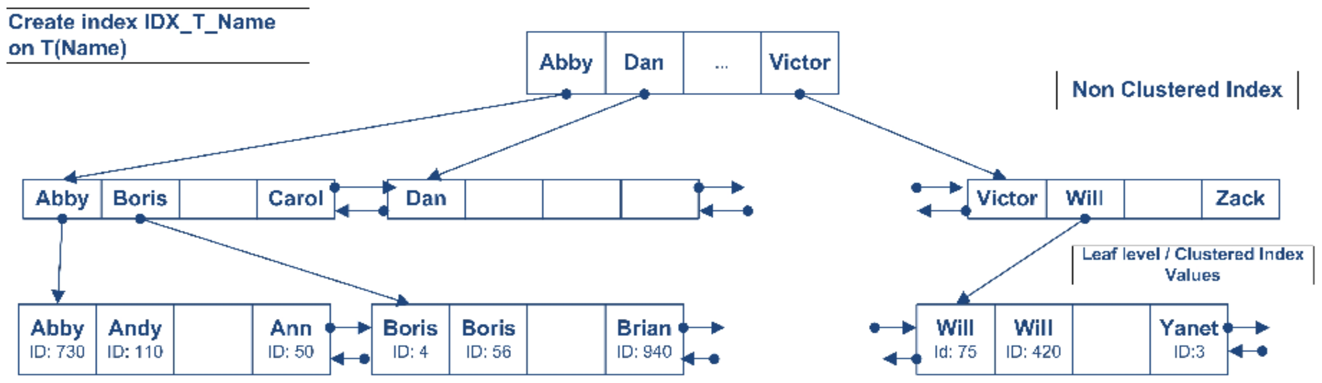
As a non-clustered index is stored in a separate structure to the base table, it is possible to create the non-clustered index on a different file group to the base table. So it can be defined as:

Non-Clustered indexes have a structure separate from the data rows. **A non-clustered index contains the non-clustered index key values and each key value entry has a pointer to the data row that contains the key value.**

The pointer from an index row in a non-clustered index to a data row is called a row locator. The structure of the row locator depends on whether the data pages are stored in a heap or a clustered table. For a heap, a row locator is a pointer to the row. For a clustered table, the row locator is the clustered index key.

You can add non key columns to the leaf level of the Non-Clustered index to by-pass existing index key limits, 900 bytes and 16 key columns, and execute fully covered, indexed, queries.

Internals of Non- Clustered Index



The structure of the index is exactly the same with the exception that leaf level does not contain table data but values for the clustered index. It does not really matter if you specify ID in the index definition, it would be there. For the heap tables, leaf level contains actual RID - Row id which consists of

- **FileId:**
- **PageNumber:**
- **RowNumber.**

Annotation at the end of the book is a good example. It does not include the actual paragraph from the book but the page # (clustered index)

Let's see how SQL Server works when it uses non-clustered index for the lookups on the tables with clustered index. As you can see, first it needs to find ID of the row(s) and next perform clustered index lookup in order to obtain the actual table data. This operation called "Key lookup" or "Bookmark lookup".

Uniqueness

An index can be defined either as unique or non-unique. **A unique index ensures that the data contained within the unique index columns appear only once within the table, including "NULL".** A unique index is commonly implemented to support the constraints.

SQL Server automatically enforces the uniqueness of the columns contained within a unique index. If an attempt is made to INSERT a value/data that already exists in the table, then an error will be generated by the SQL Server and finally the attempt to INSERT the data will fail.

A non-unique index is also applicable as there can be duplicate data; a non-unique index has more overhead than a unique index when retrieving data.

LESSON 2: CREATING AN INDEX

If we want to create an index by using Transact – SQL, we must know the columns detail for index creation. A sample syntax is given below.

Syntax:
```
CREATE  [ UNIQUE ] [ CLUSTERED | NONCLUSTERED ]  INDEX
index_name  ON  table ( column1, ... )
```

UNIQUE	: Indicates that a unique index is to be created.
CLUSTERED	: Indicates that the index created is a clustered index.
NONCLUSTERED	: Indicates that the index created is a non-clustered index.
index_name	: Is the name of the index.
Table	**:** The name of the table on which the index is to be created.
column1	: The column or columns to which the index is to be applied.

Create a single clustered index

Syntax:

```
        CREATE  [ UNIQUE ] [ CLUSTERED  ]
         INDEX  index_name  ON  table ( column1, ... )
```

Example:

```
CREATE CLUSTERED INDEX ix_empid ON EMPLOYEES (empid)
```

Create a single nonclustered index

Syntax:

```
CREATE  [ UNIQUE ] [ NONCLUSTERED  ]
 INDEX  index_name  ON  table ( column1, ... )
```

Example:

```
CREATE NONCLUSTERED INDEX ix_empname ON EMPLOYEES (EMPNAME)
```

Create a multi-column (composite) nonclustered index

Syntax:

```
CREATE  [ UNIQUE ] [ NONCLUSTERED  ]
 INDEX  index_name  ON  table ( column_name1, column_name2  )
```

Example:

```
CREATE UNIQUE NONCLUSTERD INDEX ix_nc_PresidentNumber, PresidentName ---specify index
ON dbo.Presidents (PresidentNumber, PresidentName) ---specify table and column names
```

Create a multi-column (composite) clustered index

Syntax:

```
CREATE  [ UNIQUE ] [ CLUSTERED  ]
 INDEX  index_name  ON  table ( column_name1, column_name2  )
```

Example:

```
CREATE UNIQUE CLUSTERD INDEX ix_nc_PresidentNumber, PresidentName ---specify index
ON dbo.Presidents (PresidentNumber, PresidentName) ---specify table and column names
```

Drop an Index

Removes one or more indexes from the current database. The DROP INDEX statement does not apply to indexes created by defining PRIMARY KEY or UNIQUE constraints (created by using the PRIMARY KEY or UNIQUE options of either the CREATE TABLE or ALTER TABLE statements, respectively). For more information about PRIMARY or UNIQUE KEY constraints, see "CREATE TABLE" or "ALTER TABLE" in this volume.

Syntax:

DROP INDEX 'table.index | view.index'[...n]

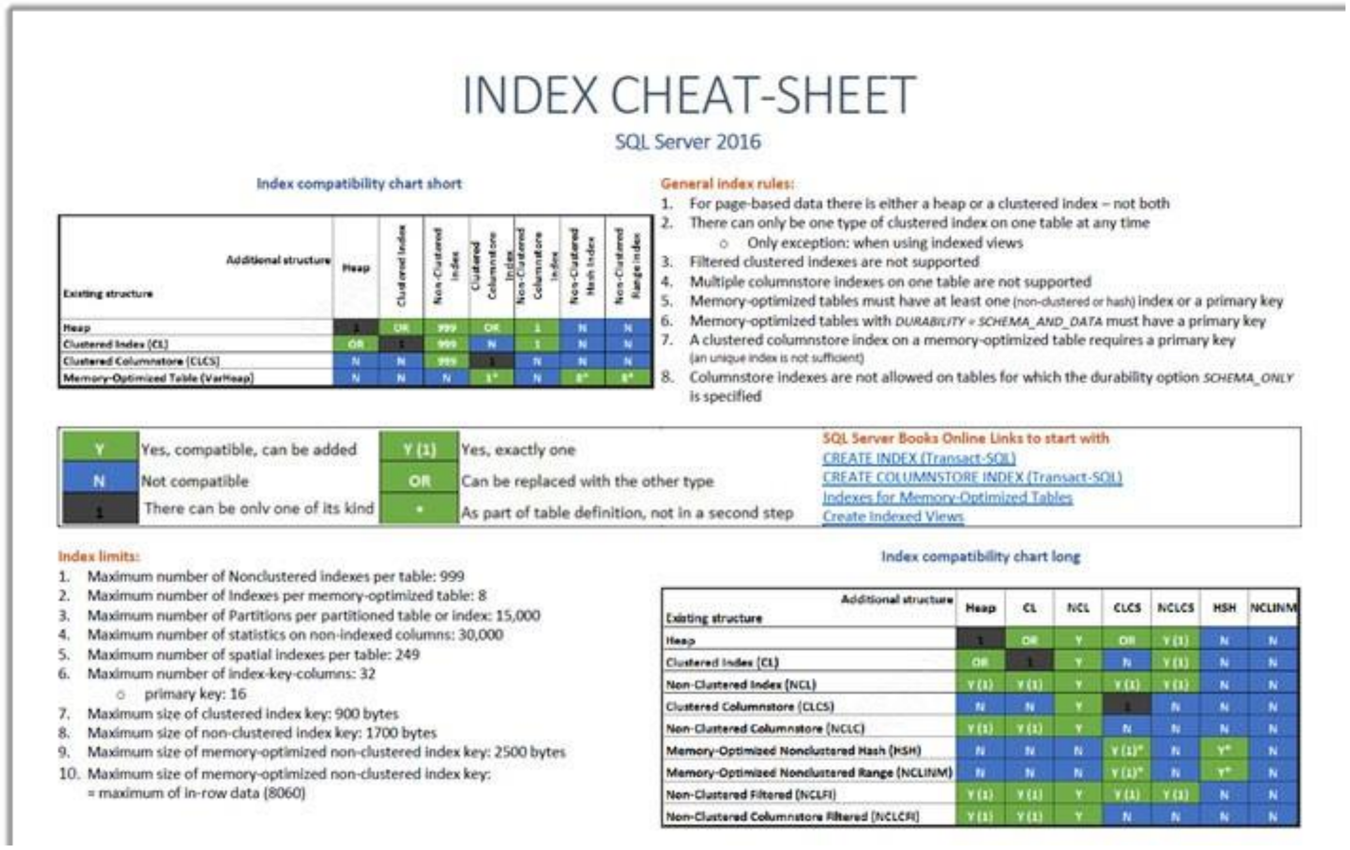
Example:

```
DROP INDEX ix_empid ON EMPLOYEES
```

You can also drop multiple indexes within a single transaction:

```
DROP INDEX IX_Product ON dbo.Product, IX_Customer ON dbo.Customer
```

CHAPTER 11 : SUMMARY / CHEATSHEET



INDEX CHEAT-SHEET

SQL Server 2016

Index compatibility chart short

Existing structure \ Additional structure	Heap	Clustered Index	Non-Clustered Index	Clustered Columnstore Index	Non-Clustered Columnstore Index	Non-Clustered Hash Index	Non-Clustered Range index
Heap	1	OR	999	OR	1	N	N
Clustered Index (CL)	OR	1	999	N	1	N	N
Clustered Columnstore (CLCS)	N	N	999	1	N	N	N
Memory-Optimized Table (VarHeap)	N	N	N	1*	N	8*	8*

General index rules:

1. For page-based data there is either a heap or a clustered index – not both
2. There can only be one type of clustered index on one table at any time
   - Only exception: when using indexed views
3. Filtered clustered indexes are not supported
4. Multiple columnstore indexes on one table are not supported
5. Memory-optimized tables must have at least one (non-clustered or hash) index or a primary key
6. Memory-optimized tables with *DURABILITY = SCHEMA_AND_DATA* must have a primary key
7. A clustered columnstore index on a memory-optimized table requires a primary key (an unique index is not sufficient)
8. Columnstore indexes are not allowed on tables for which the durability option *SCHEMA_ONLY* is specified

Y	Yes, compatible, can be added	Y (1)	Yes, exactly one
N	Not compatible	OR	Can be replaced with the other type
1	There can be only one of its kind	*	As part of table definition, not in a second step

SQL Server Books Online Links to start with

CREATE INDEX (Transact-SQL)
CREATE COLUMNSTORE INDEX (Transact-SQL)
Indexes for Memory-Optimized Tables
Create Indexed Views

Index limits:

1. Maximum number of Nonclustered indexes per table: 999
2. Maximum number of Indexes per memory-optimized table: 8
3. Maximum number of Partitions per partitioned table or index: 15,000
4. Maximum number of statistics on non-indexed columns: 30,000
5. Maximum number of spatial indexes per table: 249
6. Maximum number of index-key-columns: 32
   - primary key: 16
7. Maximum size of clustered index key: 900 bytes
8. Maximum size of non-clustered index key: 1700 bytes
9. Maximum size of memory-optimized non-clustered index key: 2500 bytes
10. Maximum size of memory-optimized non-clustered index key: = maximum of in-row data (8060)

Index compatibility chart long

Existing structure \ Additional structure	Heap	CL	NCL	CLCS	NCLCS	HSH	NCLINM
Heap	1	OR	Y	OR	Y (1)	N	N
Clustered Index (CL)	OR	1	Y	N	Y (1)	N	N
Non-Clustered Index (NCL)	Y (1)	Y (1)	Y	Y (1)	Y (1)	N	N
Clustered Columnstore (CLCS)	N	N	Y	1	N	N	N
Non-Clustered Columnstore (NCLC)	Y (1)	Y (1)	Y	N	N	N	N
Memory-Optimized Nonclustered Hash (HSH)	N	N	N	Y (1)*	N	Y*	N
Memory-Optimized Nonclustered Range (NCLINM)	N	N	N	Y (1)*	N	Y*	N
Non-Clustered Filtered (NCLFI)	Y (1)	Y (1)	Y	Y (1)	Y (1)	N	N
Non-Clustered Columnstore Filtered (NCLCFI)	Y (1)	Y (1)	Y	N	N	N	N

CHAPTER 11: QUESTIONS

Define Clustered Index ?
Define Non Clustered Index ?
How many clustered index can be created on a table ?
How many Non Clustered index can be created on a table ?

CHAPTER 12

CHAPTER 12 : VIEWS

Microsoft SQL Server provides three different ways to present a logical view of a table to user queries without having to expose the physical base table directly. Views behave just like tables but can hide complex logic.

Lessons in this chapter:

1. Lesson 1: Creating and Implementing View

LESSON 1: CREATING AND IMPLEMENTING VIEW

- A view is a virtual table that consists of columns from one or more tables.
- Though it is similar to a table, it is stored in the database.
- It is a query stored as an object.
- So a view is an object that derives its data from one or more tables.
- These tables are referred to as base or underlying tables.
- Once you have defined a view, you can reference it like any other table in a database.
  A view serves as a security mechanism.
- This ensures that users are able to retrieve and modify only the data seen by them.
- Users cannot see or access the remaining data in the underlying tables.

A view also serves as a mechanism to simplify query execution. Complex queries can be stored in the form as a view, and data from the view can be extracted using simple queries.

- A view consists of a SELECT statement that stored with a database. Because views are stored as part of the database, they can be managed independently of the applications that use them.
- A view behaves like a virtual table. Since you can code a view name anywhere you can code a table name. A view is sometimes called a viewed table.
- Views can be used to restrict the data that a user is allowed to access or to present data in a form that is easy for the user to understand. In some database users may be allowed to access data only through views.

Types of User Defined Views

1. User Defined Views
2. Materialized View
3. Indexed View
4. Partitioned Views

User Defined Views

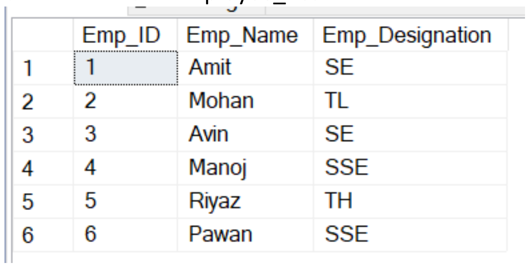
These types of view are defined by users. We have two types of user defined views.

1. Simple View
2. Complex View

Let's create a sample table as shown below for doing practical examples on views.

```
CREATE TABLE Employee_Test
(
Emp_ID INT,
Emp_Name VARCHAR(100),
Emp_Designation VARCHAR(100)
)
INSERT INTO Employee_Test VALUES
(1,'Amit','SE'),(2,'Mohan','TL'),(3,'Avin','SE'),
(4,'Manoj','SSE'),(5,'Riyaz','TH'),(6,'Pawan','SSE')
```

```
SELECT * FROM Employee_Test
```



	Emp_ID	Emp_Name	Emp_Designation
1	1	Amit	SE
2	2	Mohan	TL
3	3	Avin	SE
4	4	Manoj	SSE
5	5	Riyaz	TH
6	6	Pawan	SSE

SIMPLE VIEW

When we create a view on a single table, it is called simple view.

Syntax: **Create VIEW view_name** **AS** **Select column_name From Table_name**	**Example:** **Create VIEW vw_Employee_Test** **AS Select Emp_ID ,Emp_Name ,Emp_Designation From Employee_Test**
Syntax to call view : Select * from view_name	Example : Select * from vw_Employee_Test

In simple view we can insert, update, select and delete data.

We can only insert data in simple view if we have primary key and all not null fields in the view.

- Insert data to vw_Employee_Test

  Insert into vw_Employee_Test (Emp_name, Emp_Designation) values ('Shailu', 'SSE')

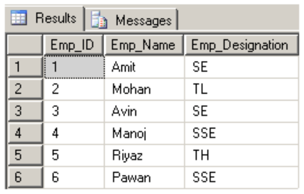
- To see the affected view

  Select * from vw_Employee_Test

- Update data to vw_Employee_Test

  Update vw_Employee_Test set Emp_Name = 'Pawan' where Emp_ID= 6

Result set of the above changes is:



	Emp_ID	Emp_Name	Emp_Designation
1	1	Amit	SE
2	2	Mohan	TL
3	3	Avin	SE
4	4	Manoj	SSE
5	5	Riyaz	TH
6	6	Pawan	SSE

- Delete data from vw_Employee_Test

  Delete from vw_Employee_Test **where** Emp_Id =6

Complex View

When we create a view on more than one table, it is called complex view.

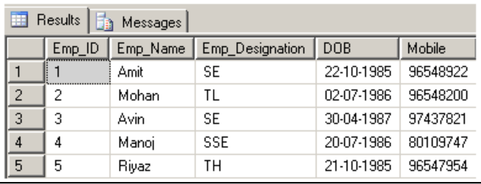
```
Create table Personal_Info ( Emp_Name varchar (55), FName varchar (55), DOB varchar (55), Address varchar
(55), Mobile int, State varchar (55) )
SELECT * FROM Personal_Info
 --Now insert data
Insert into Personal_Info values ('Amit','Patil','22-10-1985','Ghaziabad',96548922,'UP');
Insert into Personal_Info values ('Mohan','Kar','02-07-1986','Haridwar', 96548200,'UK');
Insert into Personal_Info values ('Avin','Das','30-04-1987','Noida', 97437821,'UP');
Insert into Personal_Info values ('Manoj','Yadav','20-07-1986','Rampur', 80109747,'UP');
Insert into Personal_Info values ('Riyaz','Rai', '21-10-1985','Delhi', 96547954,'Delhi');
```

-- Now create view on two tables Employee_Test and Personal_Info

Create VIEW vw_Employee_Personal_Info **as Select** e.Emp_ID, e.Emp_Name, e.Emp_Designation, p.DOB, p.Mobile **From** Employee_Test e **INNER JOIN** Personal_Info p **On** e.Emp_Name = p. Emp_Name

-- Now Query view like as table

Select * from vw_Employee_Personal_Info


Results | Messages

	Emp_ID	Emp_Name	Emp_Designation	DOB	Mobile
1	1	Amit	SE	22-10-1985	96548922
2	2	Mohan	TL	02-07-1986	96548200
3	3	Avin	SE	30-04-1987	97437821
4	4	Manoj	SSE	20-07-1986	80109747
5	5	Riyaz	TH	21-10-1985	96547954

We can only update data in complex view. We can't insert data in complex view.

- Update View : **Update** vw_Employee_Personal_Info **set** Emp_Designation = 'SSE' **where** Empid = 3
- To see affected view : **Select * from** vw_Employee_Personal_Info

Difference between SIMPLE and COMPLEX view

SIMPLE VIEW	COMPLEX VIEW
Contains only one single base table or is created from only one table.	Contains more than one base tables or is created from more than one tables.
We cannot use group functions like MAX(),COUNT(), etc.	We can use group functions.
Does not contain groups of data.	It can contain groups of data.
DML operations could be performed through a simple view.	DML operations could not always be performed through a complex view.
INSERT, DELETE and UPDATE are directly possible on a simple view.	We cannot apply INSERT, DELETE and UPDATE on complex view directly.
Simple view does not contain group by, distinct, pseudocolumn like rownum, columns defined by expressions.	It can contain group by, distinct, pseudocolumn like rownum, columns defined by expressions.
Does not include NOT NULL columns from base tables.	NOT NULL columns that are not selected by simple view can be included in complex view.

Indexed View

Views allow you to create a virtual table by defining a query against one or more tables. With a standard view, the result is not stored in the database. Instead, the result set is determined at the time a query utilizing that view is executed.

Creating a unique clustered index on a view changes it to an indexed view. The clustered index is stored in SQL Server and updated like any other clustered index, providing SQL Server with another place to look to potentially optimize a query utilizing the indexed view.

```
CREATE TABLE MyBigTable
(
ItemID INT PRIMARY KEY,
ItemDsc VARCHAR (20),
QTY INT
)
CREATE VIEW MyView WITH SCHEMABINDING  AS  SELECT ItemID, QTY   FROM dbo.MyBigTable
 WHERE QTY > 10

Create unique clustered index idx_myview on myview (qty)
```

CHAPTER 12: SUMMARY / CHEATSHEET

A view is a virtual table that consists of columns from one or more tables.
It is a query stored as an VIEW, which can be used as a table.
View can be used to hide the complexity. We can store a complex query inside a view and use it as a table.
A view also serves as a security mechanism. You can hide critical information of the table by creating a view out of it.

CHAPTER 12: QUESTIONS

1What are the differences between a table and a view in SQL Server?
How many types of views are there in SQL Server?
Why do we need Views in SQL Server?
What are the limitations of a View in SQL Server?

CHAPTER 13

CHAPTER 13: TRIGGERS

Lessons in this chapter:

In this chapter we will cover the following lessons.

Lesson 1: Trigger

LESSON 1: TRIGGER

What is a Trigger?

- A trigger is a special kind of a store procedure (set of sql statements) or program that executes in response to certain action on the table like insertion, deletion or updating of data.
- It is a database object which is bound to a table and is executed automatically.
- You can't explicitly invoke **triggers**.
- The only way to do this is by performing the required action on the table that they are assigned to.

Types of Triggers:

- DML Triggers
- DDL Triggers
- CLR Triggers
- LOGON Triggers

Syntax for Trigger

```
CREATE TRIGGER   trigger_name ON {table|view}
[WITH ENCRYPTION|EXECUTE AS]
{FOR|AFTER|INSTEAD OF} {[CREATE|ALTER|DROP|INSERT|UPDATE|DELETE]}
[NOT FOR REPLICATION]
AS
sql_statement [1...n]
END
```

DML Triggers

There are three action query types that you use in **SQL** which are INSERT, UPDATE and DELETE. So, there are three types of **triggers** and hybrids that come from mixing and matching the events and timings that fire them.

Triggers are classified into two main types:-

Instead Of **Triggers : Instead of Triggers are executed instead of insert, update or delete statements on the table**

After **Triggers** (For **Triggers**) : After Trigger are executed after insert,update or delete statements on the table.

INSTEAD OF TRIGGERS ARE OF THREE TYPES	AFTER TRIGGERS ARE OF THREE TYPES
1. INSTEAD OF INSERT This trigger will be invoked instead of the insert statements.	1. AFTER INSERT This trigger will be executed after the insert statements.
2. INSTEAD OF UPDATE This trigger will be invoked instead of the update statements.	2. AFTER UPDATE This trigger will be executed after the update statements.
3. INSTEAD OF DELETE This trigger will be invoked instead of the delete statements.	3. AFTER DELETE This trigger will be executed after the delete statements.

Examples: FIRST CREATE A TABLE Employee_Exp ON WHICH WE WILL CREATE A TRIGGER

```
CREATE TABLE EMPLOYEE_EXP
(
EMP_ID INT IDENTITY(1,1),
EMP_NAME VARCHAR (100),
EMP_SAL FLOAT,
TAX FLOAT
)
INSERT INTO EMPLOYEE_EXP VALUES ('AMIT', 1000,100),('SUNIL', 2000,200),('KAPIL', 5000,500)
```

INSTEAD OF TRIGGERS

- These can be used as an interceptor for anything that anyone tried to do on our table or view.
- These triggers are sometimes very dangerous as these will not allow that command to get executed and instead of that some other statements which is there inside the trigger will get executed.
- For Example If you define an *Instead Of trigger* on a table for the Delete operation, they try to delete rows, and they will not actually get deleted (unless you issue another delete instruction from within the trigger)

INSTEAD OF TRIGGERS can be classified further into three types as:

(a) INSTEAD OF INSERT Trigger.
(b) INSTEAD OF UPDATE Trigger.
(c) INSTEAD OF DELETE Trigger.

Syntax for Trigger

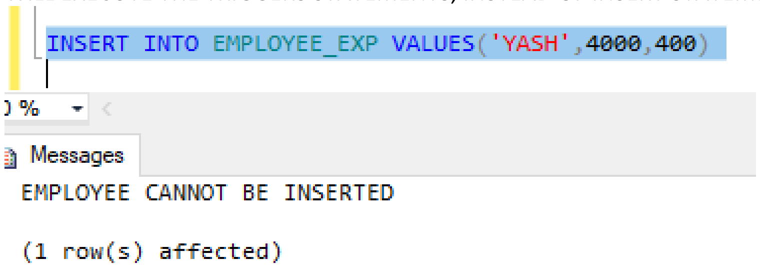
```
CREATE TRIGGER   trigger_name ON {table|view}
{ AFTER|INSTEAD OF} {INSERT|UPDATE|DELETE]}
 AS
BEGIN

END
```

EXAMPLE OF INSTEAD OF INSERT TRIGGER

WILL CREATE A INSTEAD OF INSERT TRIGGER ON THE TABLE EMPLOYEE_EXP.

```
CREATE TRIGGER TRGEMPEXP_INSERT ON  EMPLOYEE_EXP INSTEAD OF INSERT
AS
BEGIN
PRINT 'EMPLOYEE CANNOT BE INSERTED'
END
```

ONCE THE TRIGGER HAVE BEEN CREATED, IT WILL BE INVOKED WHENEVER THE INSERT STATEMENT IS FIRED, AND WILL EXECUTE THE TRIGGERS STATEMENTS, INSTEAD OF INSERT STATEMENT.



```
INSERT INTO EMPLOYEE_EXP VALUES('YASH',4000,400)
```

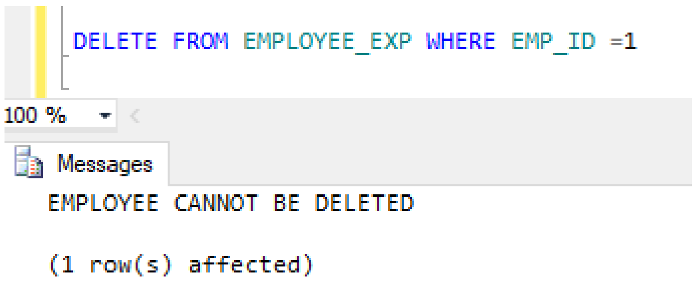
```
EMPLOYEE CANNOT BE INSERTED

(1 row(s) affected)
```

EXAMPLE OF INSTEAD OF DELETE TRIGGER

WILL CREATE A INSTEAD OF DELETE TRIGGER ON THE TABLE EMPLOYEE_EXP.

```
CREATE TRIGGER TRGEMPEXP_DEL ON  EMPLOYEE_EXP INSTEAD OF DELETE
AS
BEGIN
PRINT 'EMPLOYEE CANNOT BE DELETED'
END
```

THIS TRIGGER WILL BE INVOKED WHENEVER USER WILL FIRE THE DELETE STATEMENT



```
DELETE FROM EMPLOYEE_EXP WHERE EMP_ID =1
```

```
EMPLOYEE CANNOT BE DELETED

(1 row(s) affected)
```

Note: Inside the Trigger we will get the data from INSERTED & DELETED TABLE

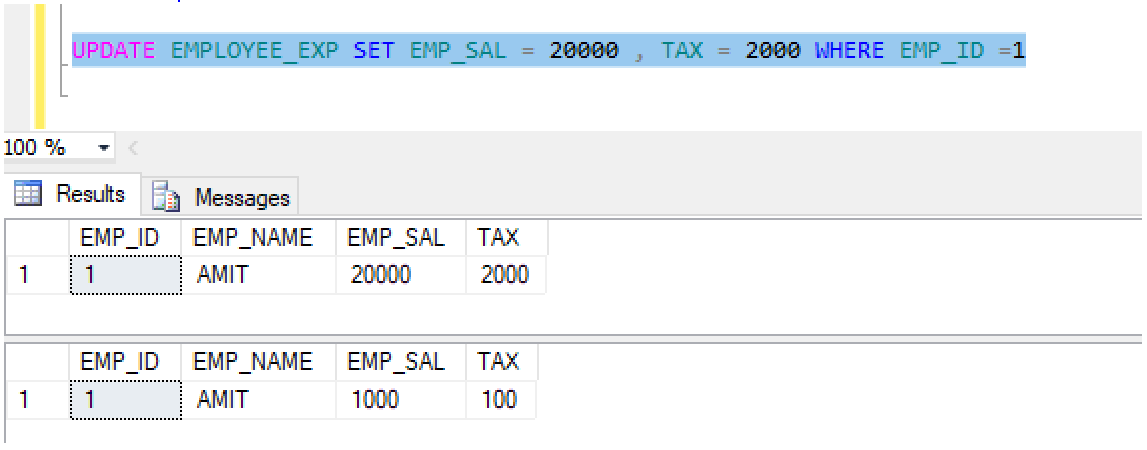
- INSERTED : WILL GIVE THE NEW DATA
- DELETED : WILL GIVE THE OLD DATA

EXAMPLE OF INSTEAD OF UPDATE TRIGGER

```
CREATE TRIGGER TRGEMPEXP_UPD ON  EMPLOYEE_EXP INSTEAD OF UPDATE
AS
BEGIN
```

```
SELECT * FROM INSERTED
SELECT * FROM DELETED

END
```

Let's execute a update statement and see the results



```
UPDATE EMPLOYEE_EXP SET EMP_SAL = 20000 , TAX = 2000 WHERE EMP_ID =1
```

	EMP_ID	EMP_NAME	EMP_SAL	TAX
1	1	AMIT	20000	2000

	EMP_ID	EMP_NAME	EMP_SAL	TAX
1	1	AMIT	1000	100

ENABLE / DISABLE TRIGGER

Specific **Triggers** can be enabled or disabled as:-

ALTER TABLE table_name {**ENABLE**| **DISABLE**} TRIGGER **triggername**

ALTER TABLE Employee_Test DISABLE TRIGGER trgAfterDelete

This disables the After Delete Trigger named trgAfterDelete on the specified table.

All the **triggers** can be enabled/disabled on the table using the statement

ALTER TABLE Employee_Test {ENABLE|DISBALE} TRIGGER ALL

```
ALTER TABLE EMPLOYEE_EXP DISABLE TRIGGER ALL
```

Dropping a DML trigger

DROP TRIGGER Trigger_name

Example:

DROP TRIGGER tr_employee

After Triggers

These **triggers** run after an insert, update or delete on a table. They are **not supported for views.** AFTER **TRIGGERS** can be classified further into three types as:

(a) AFTER INSERT Trigger.
(b) AFTER UPDATE Trigger.
(c) AFTER DELETE Trigger.

(a) AFTER INSERT Trigger EXAMPLE

This trigger is fired after an INSERT on the table. Let's create the trigger as:-
This trigger will show the records of the table and print a message once the records are deleted.

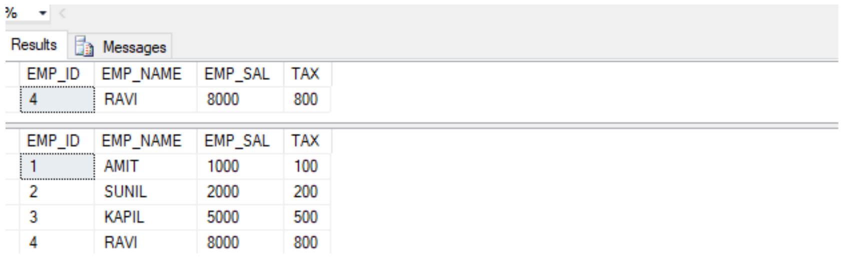
```
CREATE TRIGGER TRG_ADDEMP ON EMPLOYEE_EXP AFTER INSERT
AS
BEGIN
SELECT * FROM INSERTED
SELECT * FROM EMPLOYEE_EXP

PRINT 'EMPLOYEE RECORD ADDED SUCCESSFULLY'
END
```

- The **CREATE TRIGGER** statement is used to create the trigger.
- THE **ON** clause specifies the table name on which the trigger is to be attached.
- The **FOR INSERT** specifies that this is an **AFTER INSERT** trigger.
- In place of **FOR INSERT, AFTER INSERT** can be used. Both of them mean the same.

  In the trigger body, table named **inserted** has been used.
- This table is a logical table and contains the row that has been inserted.
- I have selected the fields from the logical inserted table from the row that has been inserted into different variables, and finally inserted those values into the Audit table.

To see the newly created trigger in action, let's insert a row into the main table as:

```
INSERT INTO EMPLOYEE_EXP VALUES('RAVI',8000,800)
```



EMP_ID	EMP_NAME	EMP_SAL	TAX
4	RAVI	8000	800

EMP_ID	EMP_NAME	EMP_SAL	TAX
1	AMIT	1000	100
2	SUNIL	2000	200
3	KAPIL	5000	500
4	RAVI	8000	800

Now, a record has been inserted into the Employee_Test table.

(b) AFTER UPDATE Trigger

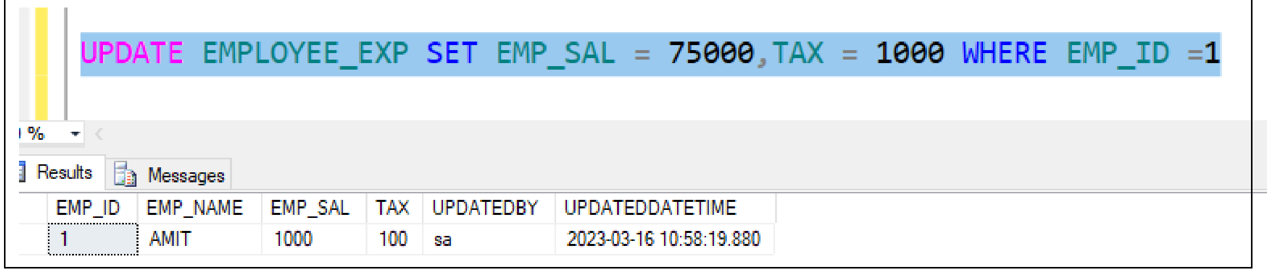
This trigger is fired after an update on the table. Let's create the trigger as:-
In this example, we will see how we can use trigger to maintain audit(history) of a table. Whenever a table is updated the previous value of the table will be moved into another table.

```
CREATE TABLE EMPLOYEE_EXP_AUDIT
(
EMP_ID INT,
EMP_NAME VARCHAR (100),
EMP_SAL   FLOAT,
TAX FLOAT,
UPDATEDBY VARCHAR(100),
UPDATEDDATETIME DATETIME
)
CREATE TRIGGER TRG_UPEMP ON EMPLOYEE_EXP AFTER UPDATE
AS
BEGIN
INSERT INTO EMPLOYEE_EXP_AUDIT SELECT *,SUSER_NAME(),GETDATE() FROM DELETED
SELECT * FROM EMPLOYEE_EXP_AUDIT WHERE EMP_ID IN(SELECT EMP_ID FROM DELETED)
END
```

- The **AFTER UPDATE** Trigger is created in which the updated record is inserted into the audit table.
- There is no logical table updated like the logical table inserted.
- We can obtain the updated value of a field from the **update (column name)** function.
- In our trigger, we have used, **if update (Emp_Name)** to check if the column Emp_Name has been updated.
- We have similarly checked the column Emp_Sal for an update.

Let's update a record column and see what happens.

```
UPDATE EMPLOYEE_EXP SET EMP_SAL = 75000,TAX = 1000 WHERE EMP_ID =1
```



```
UPDATE EMPLOYEE_EXP SET EMP_SAL = 75000,TAX = 1000 WHERE EMP_ID =1
```

EMP_ID	EMP_NAME	EMP_SAL	TAX	UPDATEDBY	UPDATEDDATETIME
1	AMIT	1000	100	sa	2023-03-16 10:58:19.880

This inserts the row into the audit table as:-

(c) AFTER DELETE Trigger

This trigger is fired after a delete on the table.

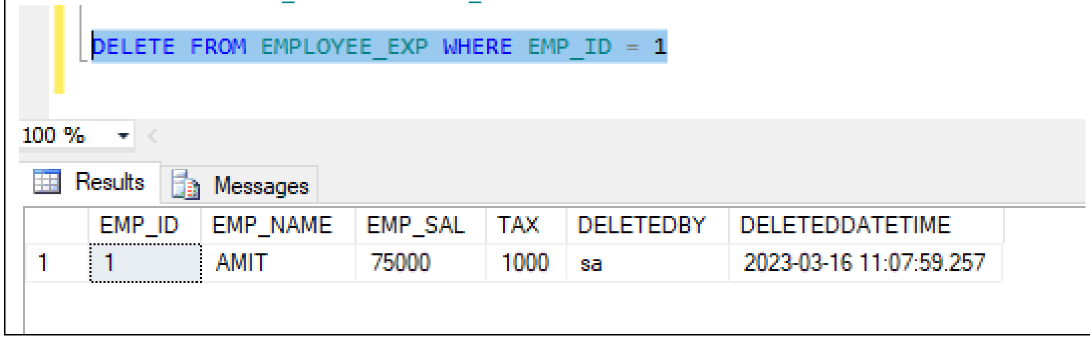
Example : In this example we will create a after delete trigger to store the deleted records in a table for reference purpose.

Let's create the trigger as:-

```
CREATE TABLE EMPLOYEE_EXP_DELETED
(
EMP_ID INT,
EMP_NAME VARCHAR (100),
EMP_SAL   FLOAT,
TAX FLOAT,
DELETEDBY VARCHAR(100),
DELETEDDATETIME DATETIME
)
CREATE TRIGGER TRG_DELEMP ON EMPLOYEE_EXP AFTER DELETE
AS
BEGIN
INSERT INTO EMPLOYEE_EXP_DELETED SELECT *,SUSER_NAME(),GETDATE() FROM DELETED
SELECT * FROM EMPLOYEE_EXP_DELETED
END
```

In this trigger, the deleted record's data is picked from the **logical deleted table** and inserted into the audit table.
Let's fire a delete on the main table.
A record has been inserted into the audit as:-

```
DELETE FROM EMPLOYEE_EXP WHERE EMP_ID = 1
```



```
DELETE FROM EMPLOYEE_EXP WHERE EMP_ID = 1
```

	EMP_ID	EMP_NAME	EMP_SAL	TAX	DELETEDBY	DELETEDDATETIME
1	1	AMIT	75000	1000	sa	2023-03-16 11:07:59.257

DDL Triggers

In SQL Server we can create triggers on DDL statements (like CREATE, Alter and Drop) and certain system defined stored procedures that perform DDL operations.

Example: If you are going to execute the CREATE LOGIN statement or the sp_addlogin stored procedure to create login user, then both these cases can execute/fire a DDL trigger that you can create on CREATE_LOGIN event of SQL Server.

We can use only FOR/ALTER clause in DDL triggers not INSTEAD OF clause means we can make only after trigger on DDL statements.

DDL triggers can be used to observe and control actions performed on the server, and to audit these operations. DDL triggers can be used to manage administrator task such as auditing and regulating database operations.

```
CREATE TRIGGER trigger1
ON DATABASE
FOR DROP_TABLE, ALTER_TABLE AS
PRINT 'You cannot drop or alter this table.'
ROLLBACK
```

```
CREATE TRIGGER PreventDropTable
ON DATABASE
FOR DROP_TABLE
AS
PRINT 'Tables cannot be dropped'
ROLLBACK
```

DDL Trigger Examples

```
USE SqlHints DDLTriggersDemo
GO
CREATE TRIGGER DDLTriggerToBlockTableDDL
ON DATABASE
FOR DROP_TABLE,ALTER_TABLE
AS
BEGIN
 PRINT 'Disable trigger DDLTriggerToBlockTableDDL to drop or alter tables'
 ROLLBACK
END
```

How to disable a ddl Trigger

DISABLE TRIGGER DDLTriggerToBlockTableDDL ON DATABASE

```
ENABLE TRIGGER DDLTriggerToBlockTableDDL ON DATABASE

DROP TRIGGER DDLTriggerToBlockTableDDL ON DATABASE
```

CLR Triggers

CLR triggers are special type of triggers that based on the CLR (Common Language Runtime) in .net framework. CLR integration of triggers has been introduced with SQL Server 2008 and allows for triggers to be coded in one of .NET languages like C#, Visual Basic and F#.

We coded the objects (like trigger) in the CLR that have heavy computations or need references to objects outside the SQL Server. We can write code for both DDL and DML triggers, using a supported CLR language like C#, Visual basic and F#. I will discuss CLR trigger later.

LOGON TRIGGERS

Logon triggers are special type of trigger that fire when LOGON event of SQL Server is raised. This event is raised when a user session is being established with SQL Server that is made after the authentication phase finishes, but before the user session is actually established. Hence, all messages that we define in the trigger such as error messages, will be redirected to the SQL Server error log. Logon triggers do not fire if authentication fails. We can use these triggers to audit and control server sessions, such as to track login activity or limit the number of sessions for a specific login.

Syntax for Logon Trigger

CREATE TRIGGER trigger_name ON ALL SERVER [WITH ENCRYPTION]

{FOR|AFTER} LOGON

AS

sql_statement [1...n]

END

Example: /* Create Audit Database */

```
CREATE DATABASE AuditDb
USE AuditDb


/* Create Audit Table */
CREATE TABLE ServerLogonHistory
(SystemUser VARCHAR(512),
DBUser VARCHAR(512),
SPID INT,
LogonTime DATETIME)
GO

/* Create Logon Trigger */
CREATE TRIGGER Tr_ServerLogon
ON ALL SERVER FOR LOGON
AS
BEGIN
INSERT INTO AuditDb.dbo.ServerLogonHistory
SELECT SYSTEM_USER,USER,@@SPID,GETDATE()
END
GO

alter TRIGGER Tr_ServerLogon
ON ALL SERVER after LOGON
AS
BEGIN
INSERT INTO AuditDb.dbo.ServerLogonHistory
SELECT SYSTEM_USER,USER,@@SPID,GETDATE()
END
```

CHAPTER 13 : SUMMARY /CHEATSHEET

Trigger are set of sql statements which are executed on different commands of the table.
Triggers are categorized as INSTEAD OF TRIGGER and AFTER TRIGGER.
INSTEAD OF TRIGGER are executed instead of INSERT, UPDATE & DELETE. There are three types of INSTEAD OF TRIGGER. • **INSTEAD OF INSERT** • **INSTEAD OF UPDATE** • **INSTEAD OF DELETE**
AFTER TRIGGER are executed after INSERT, UPDATE & DELETE. There are three types of AFTER TRIGGER. • **AFTER INSERT** • **AFTER UPDATE** • **AFTER DELETE**

CHAPTER 13 : ASSIGNMENTS

Assignment1 - BankAccounts Assignment

1. **Create a Table BankAccounts in SQL SERVER.**

CREATE A TABLE FOR BANKACCOUNT

COLUMN_NAME	DATATYPE	CONSTRAINT
ACCOUNTNO	INT	PRIMARY KEY, IDENTITY(1,1)
ACCOUNTNAME	VARCHAR(100)	
BALANCE	FLOAT	

- Part A

WRITE A TRIGGER TO MAINTAIN THE ACCOUNT HISTORY IN ACCOUNT_HISTORY TABLE

When there is any update into the Balance of the BankAccount table, it should be maintained in the history table like this
TABLENAME : ACCOUNTHISTORY

ACCOUNTNO	TRANSACTIONTYPE	AMT	LASTBALANCE	TRANSACTIONDATE
101	DEBT	100	6000	08/05/2020
101	CREDIT	500	6500	09/05/2020

- Part B

WHENEVER THERE IS AN UPDATE INTO THE BANKACCOUNT TABLE, IT SHOULD ALSO GENERATE ALSO GENERATE THE LAST 5 TRANSACTION DONE INTO THE ACCOUNT NO

Assignment2 - Employee Assignment

2. **Create a Table EMPLOYEE in SQL SERVER.**

CREATE A TABLE FOR EMPLOYEE

COLUMN_NAME	DATATYPE	CONSTRAINT
EMPNO	INT	PRIMARY KEY, IDENTITY(1,1)
EMPNAME	VARCHAR(100)	
SALARY	FLOAT	
TAX	FLOAT	
Designation	VARCHAR(100)	

- Part A

WRITE A TRIGGER TO MAINTAIN THE updates IN EMPLOYEE TABLE
When there is any update in the **EMPLOYEE** table, it should be maintained in the history table like this
TABLENAME : **EMPLOYEE_**HISTORY

EMPNO	EMPNAME	SALARY	TAX	DESIGNATION	UPDATED_DATETIME	UPDATED_BY
					Datetime at which the data have been updated	UserName who have updated the record

- Part B

WRITE A TRIGGER TO MAINTAIN THE deleted IN EMPLOYEE TABLE
When there is any delete in the **EMPLOYEE** table, it should be maintained in the deleted table like this
TABLENAME : **EMPLOYEE_**DELETE

EMPNO	EMPNAME	SALARY	TAX	DESIGNATION	DELETED_DATETIME	DELETED_BY
					Datetime at which the data have been deleted	UserName who have DELETED the record

Assignment3 - StudentAssignment

CREATE A STUDENT TABLE SIMILAR TO THIS

ROLLNO	NAME	ENGMARKS	SCNMARKS	MATHMARKS	TOTALMARKS	PERCENTAGE	GRADE
IDENTITY(1,1)							

WRITE A TRIGGER TO CALCULATE THE VALUES OF TOTALMARKS, PERCENTAGE AND GRADE AND INSERT INTO THE STUDENT TABLE.
THE USER WILL INSERT NAME,ENGMARKS,SCNMARKS,MATHMARKS.
THE TOTALMARKS,PERCENTAGE AND GRADE SHOULD BE CALCULATED THROUGH TRIGGER AND INSERT INTO THE TABLE.

CHAPTER 13 : QUESTIONS

Define TRIGGER ?
Explain any real-time examples where trigger can be used ?
What is INSTEAD OF TRIGGER?
What is AFTER TRIGGER?
What is use of INSERTED table in Trigger ?
What is use of DELETED table in Trigger ?