**Use Database:** USE master

**Database:** CREATE DATABASE database\_name

**Use Database:** USE database\_name

**Drop Database:** DROP DATABASE database\_name

**Constraints:** NOT NULL, UNIQUE, DEFAULT (default value), PRIMARY KEY, FOREIGN KEY (FOREIGN KEY REFERENCES table\_name (col\_name)), CHECK (check (condition)), IDENTITY (identity (1,1) aka IDENTITY (start, step))

**Data Types:**

*Numeric & Float:*

bit: 0, 1 or null (1 byte)

tinyint: 0 to 255 (1 byte)

smallint: -32768 to +32767 (2 bytes)

int: -2,147,483,648 to +2,147,483,647 (4 bytes)

bigint: -9,223,372,036,854,775,808 to +9,223,372,036,854,775,807 (8 bytes)

decimal (7,3): 7-digit whole numbers and 3 digits after decimal (XXXX.XXX) (5 to 17 bytes)

*Text:*

char (255): Alphanumeric

nchar (255): Unicode

varchar (8000): Alphanumeric

nvarchar (4000): Unicode

text: 62000+ Chars (Product Feedback, Comment)

ntext: 62000+ Special Characters (Product Feedback, Comment)

varchar(max): Alphanumeric

nvarchar(max): Unicode

*Datetime (YYYY-MM-DD):*

date: YYYY-MM-DD (3 bytes)

smalldatetime: YYYY-MM-DD HH:MM:SS: precision is up to seconds (4 bytes)

datetime2: YYYY-MM-DD HH:MM:SS:XXXXXXX: precision is up to milliseconds (6 to 8 bytes)

datetime: YYYY-MM-DD HH:MM:SS: precision is up to seconds (8 bytes)

Money:

smallmoney: -214,748.3648 to 214,748.3647 (4 bytes)

money: -922,337,203,685,477.5808 to 922,337,203,685,477.5807 (8 bytes)

**System Defined Functions:**

*String Functions:*

SELECT upper('rajat') – converts the text to upper case

SELECT lower('RAJAT') – returns the text in lower case

SELECT len(' rajat123 ') as lengthofstring – returns the length of the text

SELECT 'Rajat' + ' ' + 'Dutta' – concatenates the strings

SELECT left('rajat',2) – returns the 2 characters for the left of the string

SELECT right('rajat',4) – returns the 4 characters from the right of the string

SELECT trim(' Rajat Dutta ') – o/p 'Rajat Dutta', removes leading and trailing spaces from the string

SELECT replace('Rajat', 'aja', 'j') as op – replaces a part of the string with another value

SELECT reverse('Rajat') – reverses a string

SELECT substring('Rajat',3,10) – returns the part of text from starting position to the number of characters specified

SELECT charindex('j','Rajat') – tries to search for the expression and returns its first location

SELECT concat('Rajat',' ', 'Dutta') – concatenates multiple strings

*DateTime Functions:*

SELECT getdate() – This function returns the current server date time

SELECT getutcdate() - This gets us the UTC date and time (Universal Time Coordinated/Greenwich Mean Time)

SELECT getdate() + 2 – This will add 2 days to my current date

SELECT getdate() - 2 – This will subtract 2 days from my current date

SELECT dateadd(hour, 2, getdate()) – This will add 2 hours to now

SELECT dateadd(hour, -2, getdate()) – This will deduct 2 hours from the current date time

SELECT dateadd(year, 2, getdate()) – This adds 2 years to the current date time

SELECT dateadd(year, -2, getdate()) – This will deduct 2 years from the current date time

SELECT dateadd(month, 2, getdate()) – This adds 2 months to the current date time

SELECT dateadd(month, 2, dateadd(hour, 3, getdate())) – This adds 3 hours and 2 months to the current date

SELECT year(getdate()) – Returns the year of the specified date

SELECT month(getdate()) – Returns the month of the specified date

SELECT datepart(day, getdate()) – Returns the date part of the current datetime

SELECT datepart(hour, getdate()) – Returns the hour part of current datetime

SELECT datepart(minute, getdate()) – Returns the minute part of current datetime

SELECT datepart(year, getdate()) – Returns the year part of current datetime

SELECT datepart(month, getdate()) – Returns the month part of current datetime

SELECT datename(weekday, getdate()) – Get name of weekday

SELECT datename(month, getdate()) – Get name of month

SELECT datediff(day, '09/03/2021', '09/07/2021') - returns the difference in days between the start and end date

SELECT datediff(hour, '09/03/2021', '09/07/2021')

SELECT datediff(minute, '09/07/2021 20:00', '09/07/2021 23:00')’

**Table Creation:** CREATE TABLE table\_name (col\_name1 datatype [constraint1|constraint2 …], col\_name2 datatype [constraint1 | constraint2 …] …)

**Insertion:** INSERT INTO table\_name [(col\_name1, col\_name2 …)] values (value1, value2 …) **OR**

INSERT INTO table\_name [(col\_name1, col\_name2 …)] select value1, value2 … **OR**

INSERT INTO table\_name [(col\_name1, col\_name2 …)] output inserted.col\_name1, inserted.col\_name2 …) values (value1, value2 …)

**Explicit Insertion into Identity Column:** SET IDENTITY\_INSERT table\_name ON (after insertion) SET IDENTITY\_INSERT table\_name OFF

**Backup Table:** SELECT \* INTO backup\_table\_name FROM table\_name ***–*** *Entire Table* **OR**

SELECT \* INTO backup\_table\_name FROM table\_name WHERE condition1 ***–*** *Selected Rows* **OR**

SELECT col\_name1, col\_name2 INTO backup\_table\_name FROM table\_name [WHERE condition] ***–*** *Selected Columns* [and Rows]

**Alter Table:** ALTER TABLE table\_name ADD col\_name datatype [constraints] **OR**

ALTER TABLE table\_name ALTER COLUMN col\_name datatype [constraints] **OR**

ALTER TABLE table\_name RENAME COLUMN old\_col\_name to new\_col\_name **OR**

ALTER TABLE ADD CONSTRAINT [constraint1 | constraint2 …] (col\_name1, col\_name2 …) **OR**

ALTER TABLE table\_name DROP COLUMN [IF EXISTS] col\_name1, col\_name2 …

**Update Table:** UPDATE table\_name SET col\_name = value [WHERE condition]

**Delete Records from Table:** DELETE table\_name [WHERE condition]

**Truncate Table:** TRUNCATE TABLE table\_name

**Drop Table:** DROP TABLE [IF EXISTS] table\_name

**Check & Drop Table:** IF object\_id('table\_name') IS NOT NULL DROP TABLE table\_name

**Check & Drop Temp Table:** IF object\_id('tempdb..#table\_name') IS NOT NULL DROP TABLE #table\_name

**Listing all tables in a DB:** SELECT \* FROM sys.tables **OR** SELECT \* FROM INFORMATION\_SCHEMA.TABLES

**Computed Column:**

*Creating Computed Column:*CREATE TABLE Products (ProductID int IDENTITY (1,1) NOT NULL, QtyAvailable smallint, UnitPrice money, InventoryValue AS QtyAvailable \* UnitPrice) [PERSISTED]

*Altering existing table:*ALTER TABLE Products ADD RetailValue AS (QtyAvailable \* UnitPrice \* 1.5) [PERSISTED]

*Dropping computed column:* ALTER TABLE Products DROP COLUMN RetailValue

**Joins:** SELECT [table1.]col\_name1, [table2.].col\_name2, [table1.]col\_name3 FROM Orders [AS <alias1>] [INNER | LEFT | RIGHT | CROSS (By Default INNER)] JOIN table2 [AS <alias2>] ON table1/alias1.col\_name = table2/alias2.col\_name

**Query Process Steps**

1. Getting Data (FROM, JOIN)

2. Row Filter (WHERE)

3. Grouping (GROUP BY)

4. Group Filter (HAVING)

5. Return Expressions (SELECT)

6. Order & Paging (ORDER BY & DISTINCT / TOP /OFFSET-FETCH)

**Derived Table:** SELECT \* FROM (<inner SELECT statement>) AS <alias>

**Variables:**

*Declaring a variable:*

DECLARE @variable\_name1 [AS] datatype, @variable\_name2 [as] datatype … **OR**

DECLARE @variable\_name1 [AS] datatype [=''], @variable\_name2 [as] datatype [=''] … **OR**

DECLARE @variable\_name1 [AS] datatype [=value1], @variable\_name2 [as] datatype [=value2] …

*Setting a variable:*

SET @variable\_name1 = value1

SET @variable\_name2 = value2

:

**OR**

SELECT @variable\_name1 = value1

:

**Table Variable:** DECLARE @table\_name table (col\_name1 datatype [PRIMARY KEY | UNIQUE | NOT NULL | CHECK (condition)], col\_name2 datatype [UNIQUE | NOT NULL | CHECK (condition)] …)

*Using a Table Variable:* SELECT \* FROM @table\_name [AS alias]

**Common Table Expression (CTE):** ;WITH cte\_name1 [(col\_names)] [, cte\_name2 [(col\_names)]] AS (SQL query)

SELECT \* FROM cte\_name1 [UNION SELECT \* FROM cte\_name2]

*CTE’s Usage is Mandatory right after creation in the same batch and it can be used just once using SELECT, INSERT, UPDATE or DELETE Statement*

**Recursive CTE:** ;WITH cte\_name (col\_names) AS

(

-- Anchor member

initial\_query

UNION ALL

-- Recursive member that references cte name

recursive\_query

)

-- references cte\_name

SELECT \* FROM cte\_name

**Index:** CREATE [OR ALTER] [CLUSTERED | NONCLUSTERED | UNIQUE] INDEX index\_name ON table\_name (col\_name1 [, col\_name2, col\_name3…])

**Ideal column to create Clustered Index:** Ideally the column should be Unique, Static, Narrow, Non-nullable, Fixed Width and Ever-increasing column

**Covering Index using INCLUDE:** CREATE [OR ALTER] [NONCLUSTERED | UNIQUE] INDEX index\_name ON table\_name (col\_name1) INCLUDE (col\_name2, col\_name3…)

**Filtered Index:** CREATE [OR ALTER] [NONCLUSTERED | UNIQUE] INDEX index\_name ON table\_name (col\_name1, col\_name2…) WHERE col\_name1 = value and col\_name2 = value2

**View:** CREATE [OR REPLACE] VIEW view\_name [WITH SCHEMABINDING] AS SELECT column1, column2 ... FROM table\_name [WHERE condition]

*Using a view:* SELECT \* FROM view\_name [AS ALIAS] [WHERE condition]

**Used Defined Functions:**

**Scaler-valued Functions:** CREATE [OR ALTER] FUNCTION function\_name

(

@param\_name1 datatype

[, param\_name2 datatype …]

)

RETURNS datatype AS

BEGIN

SQL statements

RETURN value

END

*Using a function:* SELECT dbo.function\_name (value1, value2 …) [AS alias] FROM table\_name [WHERE condition]

**Table-valued Functions:** CREATE [OR ALTER] FUNCTION function\_name (@param\_name1 datatype [, param\_name2 datatype …]) RETURNS TABLE AS

RETURN SQL statements **OR**

*TVF using Table Variable:* CREATE [OR ALTER] FUNCTION function\_name (@param\_name datatype) RETURNS @table\_variable\_name TABLE

(

col\_name1 datatype

[, col\_name2 datatype …]

)

AS

BEGIN

IF @param\_name = value1

INSERT INTO @table\_variable\_name select \* FROM Table1

ELSE IF @param\_name = value2

INSERT INTO @table\_variable\_name SELECT \* FROM Table2

RETURN

END

**Stored Procedure:** CREATE [OR ALTER] PROCEDURE (OR PROC) procedure\_name [@param\_name1 datatype [='' **OR** =value1] [output], param\_name2 datatype [='' **OR** =value2] [output] …] AS

[BEGIN]

SQL statements (SELECT, INSERT, DELETE, UPDATE statements)

[END]

*Executing a Procedure:* EXECUTE (OR EXEC) procedure\_name [‘param\_value1’, ‘param\_value2’ …]

*Checking for existing SPs:* SELECT \* FROM sys.procedures **OR** SELECT \* FROM sys.objects WHERE type='P' [AND name LIKE '%sp\_%']

**Dynamic SQL:** CREATE [OR ALTER] PROCEDURE (or PROC) procedure\_name @param\_name varchar(max) AS

DECLARE @SQL varchar(max)='SELECT \* FROM ' + @param\_name

EXEC (@SQL) **OR** sp\_execute (@SQL)

*Executing a Dynamic SQL:* EXECUTE (OR EXEC) procedure\_name [param\_value1, param\_value2 …]

**Synonym:** CREATE SYNONYM [schema\_name.]synonym\_name FOR db\_name.schema\_name.object\_name (objects can be Tables, Views, Functions & Stored Procedures)

**DDL Trigger Example:** CREATE TRIGGER trigger\_name ON {DATABASE | ALL SERVER} [WITH ENCRYPTION | EXECUTE] FOR

CREATE\_TABLE, ALTER\_TABLE, DROP\_TABLE, DDL\_TABLE\_EVENTS

AS

BEGIN

INSERT INTO TableLog (

EventVal,

DateChanged,

ChangedBy

)

VALUES (

EVENTDATA(),

GETDATE(),

USER

)

END

**DML Triggers:** CREATE [OR ALTER] TRIGGER trigger\_name ON table\_name {FOR, AFTER, INSTEAD OF} {INSERT, DELETE, UPDATE} [NOT FOR REPLICATION] AS SQL statements (SELECT, INSERT, UPDATE OR DELETE statements)

**Transactions:** BEGIN TRANSACTION (or TRAN)

SQL statements (INSERT, UPDATE, DELETE statements)

-- Check for error

IF (@@ERROR > 0)

BEGIN

ROLLBACK TRANSACTION (or TRAN)

END

ELSE

BEGIN

COMMIT TRANSACTION (or TRAN)

END

**Cursors:** DECLARE cursor\_name CURSOR for

SELECT statement[s]

OPEN cursor\_name

FETCH NEXT FROM cursor\_name INTO @Variable [, @Variable1]

WHILE @@FETCH\_STATUS=0

BEGIN

print @Variable [,@Variable1]

FETCH NEXT FROM cursor\_name INTO @Variable [, @Variable1]

END

CLOSE cursor\_name

DEALLOCATE cursor\_name

**Error Handling:**

BEGIN TRY

SQL statements

END TRY

BEGIN CATCH

-- SQL statements if an error occurs

END CATCH

ERROR\_NUMBER – Returns the internal number of the error

ERROR\_STATE – Returns the information about the source

ERROR\_SEVERITY – Returns the information about anything from informational errors to errors user of DBA can fix, etc.

ERROR\_LINE – Returns the line number at which an error happened on

ERROR\_PROCEDURE – Returns the name of the stored procedure or function

ERROR\_MESSAGE – Returns the most essential information and that is the message text of the error

**Data Storage Structure/Allocation Units:**

IN\_ROW\_DATA

Every row in a table has an IN\_ROW\_DATA portion

All fixed width columns must be stored in the IN\_ROW\_DATA portion

IN\_ROW\_DATA can’t span pages

ROW\_OVERFLOW\_DATA

Tables with potential wider rows might have one or more columns that can overflow to ROW\_OVERFLOW\_DATA structure

Data types in this category: varchar(n), nvarchar(n), varbinary(n), sqlvariant

Vertical partitioning of tables basis most common use patterns can be a solution to avoid critical columns to overflow into this unit

LOB\_DATA

Tables with potential for extremely wide rows might have one or more columns that have been stored in LOB\_DATA structure

Data types in this category: text, ntext & image (legacy LOB datatypes till SQL Server 2005); varchar(max), nvarchar(max), varbinary(max), XML & CLR user defined types (new LOB datatypes)

**SQL Server Profiler Trace Templates:** Trace templates define the default configuration for a trace. SQL Server ships with a standard set of templates including:

*SP\_Counts:*

Captures stored-procedure executions over time

Can be used for tracking stored-procedure frequency of usage

SP:Starting:

*Standard (the default template):*

Generic starting point for creating new traces against SQL Server

Can be used for common troubleshooting of normal database activity

Audit Login, Audit Logout, ExistingConnection, RPC:Completed, SQL:BatchCompleted, SQL:BatchStarting

TSQL:

Captures all Transact-SQL statements that are submitted to SQL Server

Can be used for debugging client applications

Audit Login, Audit Logout, ExistingConnection, RPC:Starting, SQL:BatchStarting

TSQL\_Duration:

Captures all Transact-SQL statements executed by SQL Server and groups them by duration

Can be used to identify slow queries

RPC:Completed, SQL:BatchCompleted

TSQL\_Grouped:

Captures all Transact-SQL statements submitted to SQL Server and groups the events by the user or client that submitted the statement

Can be used to identify queries submitted by a specific user or client

Audit Login, Audit Logout, ExistingConnection, RPC:Starting, SQL:BatchStarting

TSQL\_Locks:

Captures all of the Transact-SQL statements that are executed by clients along with lock events from the engine

Can be used to troubleshoot deadlocks, lock time-outs, and lock escalation events

Blocked Process Report, SP:StmtCompleted, SP:StmtStarting, SQL:StmtCompleted, SQL:StmtStarting, Deadlock Graph, Lock:Cancel, Lock:Deadlock, Lock:Deadlock Chain, Lock:Escalation Lock:Timeout (timeout>0)

TSQL\_Replay:

Captures the events that are required for the trace to be used for workload replay operations

Can be used for performance tuning and benchmarking a database

CursorClose, CursorExecute, CursorOpen, CursorPrepare, CursorUnprepare, Exec Prepared SQL, Prepare SQL, Audit Login, Audit Logout, Existing Connection, RPC Output Parameter, RPC:Completed, RPC:Starting, SQL:BatchCompleted, SQL:BatchStarting

TSQL\_SPs:

Captures detailed information about all executing stored procedures

Can be used to analyze the individual statements of stored procedures

Audit Login, Audit Logout, ExistingConnection, RPC:Starting, SP:Completed, SP:Starting, SP:StmtStarting, SQL:BatchStarting

Tuning:

Captures information about stored procedures and Transact-SQL batch execution

Can be used to capture trace output that Database Engine Tuning Advisor can use as a workload for tuning analysis

RPC:Completed, SP:StmtCompleted, SQL:BatchCompleted

Customized templates can be created and saved to create a standard set of trace definitions to be used across clients.