1. **Data Types3**

In a Database, each column, local variable, expression, and parameters has a related data type. A data type is an attribute that specifies the type of data that the object can hold: integer data, character data, monetary data, data and time data, binary strings, and so on.

**Integer Types** : To hold the integer values it provides with **tinyint, smallint, int** and **bigint** data types with sizes 1,2,4 and 8 bytes respectively.

**Boolean Types** : To hold the Boolean values it provides with **bit** data type that can take a values of 1,0, or NULL.

Note : The string values TRUE and FALSE can be converted to **bit** values : TRUE is converted to 1 and FALES is converted to 0.

**Decimal Types** :To hold the decimal values it provides with the following types :

-**Decimal**[ **(**p[ ,s]) ]and **numeric** [ (p[ ,s]) ]

*p*(precision)

The maximum total number of decimal digits that can be stored, both to the left and to the right of the decimal point. The precision must be a value from 1 through the maximum precision of 38. The default precision is 18.

*s*(scale)

The maximum number of decimal digits that can be stored to the right of the decimal point. Scale must be a value from 0 through p. Scale can be specified only if precision is specified. The default scale is 0.

Storage sizes of Decimal and Numeric types vary, based on the precision

**Precision Storage byte**

1 – 9 5

10 – 19 9

20 – 28 13

29 – 38 17

Note: numeric is functionally equivalent to decimal

**-float [(n)] and real**

**-**Approximate-number data types for use with floating point numeric data. Floating point data is

Approximate; therefore, not all values in the data type range can be represented exactly. Where *n* is the

Number of bits that are used to store the mantissa of the **float** number in scientific notation and,

Therefore, dictates the precision and storage size. If *n* is specified, it must be a values between 1 and 53.

The default value of *n* is 53.

***n* value Precision Storage size**

1-24 7 digits 4 bytes

25-53 15 digits 8 bytes

**Monetary or Currency Types:** To hold the Currency values it provides with the following types which takes a scale of 4 by default:

**money - 922,337,203,685,477.5808** to **922,337,203,685,477.5807** 8 bytes

**smallmoney - 214,748.3648** to **214,748.3647** 4 bytes

**Data and Time values:** To hold the Date and Time values of a day it provides with the following types:

**Data type Range Accuracy**

datetime January 1, 1753, through December 31, 9999 3.33 milliseconds

smalldatetime January 1, 1900, through June 6, 2079 1 minute

Values with the **datetime** data type are stored internally by the Microsoft SQL Server 2005 Database

Engine as two 4-byte integers. The first 4 bytes store the number of days before or after the base date: January 1, 1990. The base date is the system reference date. The other 4 bytes store the time of day represented as the number of milliseconds after midnight.

The **smalldatetime** data type store date and time of day with less precision than **datetime.** The Database Engine as store **smalldatetimes**  values as two 2-byte integer. The first 2 bytes store the number of days after January 1,1900. The other 2 bytes store the number of minutes since midnight.

**String values:** To hold the string values it provides with following types:

**char[ ( *n* ) ]**

Fixed-length, non-Unicode character data with a length of *n* bytes. *n* must be a value from 1 through 8,000. The storage size is *n* bytes.

**varchar[ ( *n |* max) ]**

Variable-length, non-Unicode character data. *n* can be a value from 1 through 8,000. **max** indicates that the maximum storage size is 2^31-1 bytes. The Storage size is the actual length of data entered +2 bytes.

**text**

It was equal to varchar(max) this data type will be removed in a future version of Microsoft SQL Server. Avoid using these data types in new development work use varchar(max) instead.

Unicode Data types for storing Multilingual Character are nchar, nvarchar and ntext where n stands for national.

n**char[ (** n **) ]**

Fixed-length Unicode character data of *n* must be a value from 1 through 4,000. The storage size is two times *n* bytes.

**nvarchar[ ( n | max** **) ]**

Variable-length Unicode character data. *n* can be a value from 1 through 4,000. **max** indicates that the maximum storage size 2^31-1 bytes. The Storage size, in bytes, is two times the number of characters entered +2 bytes.

**ntext**

It was equal to nvarchar(max) this data type will be removed in a future version of Microsoft SQL Server. Avoid using these data types in new development work use nvarchar(max) instead.

**Binary Values:** To hold the binary values likes images, audio clips and video clips we use the following types.

**binary [( *n* )]**

Fixed-length binary data with a length of *n* bytes, where *n* is a value from 1 through 8,000. The storage size is *n* bytes.

**varbinary[( *n |* max)]**

Variable-length binary data. *n* can be a value from 1 through 8,000. **max** indicates that the maximum storage size is 2^31-1 bytes. The Storage size is the actual length of data entered +2 bytes.

**Image**

It was equal to varbinary(max) this data type will be removed in a future version of Microsoft SQL Server. Avoid using these data types in new development work use varbinary(max) instead.

1. Use **char, nchar, binary** when the sizes of the column data entries are consistent.
2. Use **varchar, nvarchar, varbinary** when the sizes of the column data entries vary considerably.
3. Use **varchar(max), nvarchar(max), varbinary(max)** when the sizes of the column data entries vary considerably, and the size might exceed 8,000 bytes.

**Other Types:** Apart from the above it provides some additional types like-

**timestamp:** Is a data type that exposes automatically generated, unique binary numbers with in a database. The storage size is 8 bytes. You can use the **timestamp** column of a row to easily determine whether any value in the row has changed since the last time it was read. If any change is made to the row, the timestamp value is updated. If no change is made to the row, the **t**imestamp value is the same as when it was previously read.

**Unique identifier:** Is a 16-byte GUID which is initialized by using the newid() function or converting a string constant in the form of xxxxxxxx-xxxx-xxxx-xxxxxxxxxxxx which is used to guarantee that rows are uniquely identified across multiple copies of the table.

**Xml:** Is the data type that store XML, data. You can store **xml** instances in a column, or a variable of **xml** type. The stored representation of **xml** data type instances cannot exceed 2 gigabytes (GB) in size.

1. **Database:**

SQL Server manages the objects in a container known as Database, where we can have multiple database present in it, database when created create 2 files internally those or .mdf and .ldf file.

Syntax for creating a database:

**-CREATE DATABASE<db\_name>**

-Database names must be unique with in an intance of SQL Server.

-Any Object name in sqlserver can be of 1 through 128 characters

1. **Tables:**

-It is the object, which will store the information in database in form of rows and column.

Syntax for creating Table:

**-CREATE TABLE<table\_name>**

**(column\_name1<dtype>[width],**

**column\_name1<dtype>[width],**

**…............................**

**column\_namen<dtype>[width])**

-Table names must be unique with in the database.

-Column name must be unique with in the table.

-Every table can have maximum of 1024 and minimum of 1 column,

-CREATE TABLE Bank(Custid int, Cname varchar(50), Bal decimal(7,2))

**Populating Data into Tables:** After the table gets created to populate the data into it we use

**insert Statement:**

Syntax for insert statement:

-**INSERT INTO<table name>[(col1,col2,........coln)]**

**VALUES(val1,val2,..........value)**

Examples:

**INSERT INTO BANK VALUES (101, 'BASITH', 6000)**

-In this case we need to provide the value for all the columns in the same order they are present in the table.

-String and Date values have to be enclosed in single quotes.

**INSERT INTO BANK (CUSTID, CNAME, BAL) VALUES (102, 'THANVEER', 5500)**

**-**this statement is same as above statement.

-If we want to change the order of columns while inserting:

**INSERT INTO BANK (CNAME, CUSTID, BAL) VALUES('HAMMATH', 103, 6500)**

-If we want to insert data only into required column then:

**INSERT INTO BANK (CUSTID, BAL) VALUES (104,3600)**

-If this case the column into which values are not supplied are filled with Null values.

-We can also insert Null's explicitly into the column in the following way:

**INSERET INTO BANK VALUES (105, NULL, 5400)**

**Retrieving the data from Tables:** if we want retrieve the information from the table use

**Select Statement:**

Basic Syntax for select statement:

**SELECT < \* | COLLIST > FROM < TNAME >[CONDITION]**

1. '\*' Represents all the columns of the table in the same order.
2. COLLIST is used for specifying the required no of column and in required order.
3. CONDITIONS are optionl which can be used for retrieving the required rows
4. SELECT \* FROM BANK
5. SELECT CUSTID, CNAME, BAL FROM BANK
6. SELECT CNAME, BAL, CUSTID FROM BANK
7. SELECT CUSTID AS ACCNO, CNAME, BAL FROM BANK

-We can specify an alias name for any required column while retrieving know as column Alias.

-If we want to retrieve required rows then we use a conditions statement where:

- SELECT \* FROM BANK WHERE CUSTID=104

1. SELECT CUSTID, BAL FROM BANK WHERE CNAME='HAMMATH'

-SQL Server does not have any case restrictions while writing the conditions.

**Handling Null Values:** The value NULL means the data value for the column is unknown or not available, so we cannot use equality conditions while getting the data based on null values.

SELECT \* FROM EMP WHERE CNAME= NULL

-The above statement will not get any result because no 2 null values can be compared so to get the data based on Null values we should use the IS NULL operators as following:

SELECT \* FROM EMP WHERE CNAME IS NULL

**Updating data present on the tables:** if we want to Update the data existing in the table we use

**Update statement:**

Syntax:

**UPDATE<TNAME> SET <CNAME> = <VALUE> [, …..] [CONDITIONS]**

Note: We can modify a single column using the update statement all the rows that satisfy the conditions gets affected.

-UPDATE BANK SET CNAME='MOHAMED' WHERE CUSTID=104

-UPDATE BANK SET CNAME='HUDA', BAL=3000 WHERE CUSTID=105

**Deleting data present in the table:** if we want to delete rows of data present in the table we use

**Delete Statement:**

Syntax:

**-DLETE FROM <TNAME> [CONDITIONS]**

-DLETE FROM BANK WHERE CUSTID=105

-DLETE FROM BANK

1. **Constraints**

**Constraints:** used to enforce the integrity of the data in the column, SQL Server 2005 provides'the following mechanisms to enforce the integrity of the data in column:

-Not Null

-Unique

-Primary Key

-Check

-Default

-Foreign Key

* 1. **Not Null:** If it is imposed on a column that column will not allow Null values into it; this can be imposed on any no of column.

**-CREATE TABLE<table\_name>**

**(column\_name1<dtype>[width] [NOT NULL],**

**column\_name1<dtype>[width] [NOT NULL],**

**…............................**

**column\_namen<dtype>[width] [NOT NULL])**

Recreating the bank table with Not Null Constraint on it:

**CREATE TABLE BANK ( Custid int Not Null, Cname varchar(50), Bal decimal(7,2) Not Null)**

After creating this if we try to insert a null value into the Custid or Bal column it will restrict us:

INSERT INTO Bank VALUES (NULL, 'RAJA', 3500)

INSERT INTO Bank (CUSTID, CNAME) VALUES (101, 'BASITH')

The drawback with Not Null Constraint is even if it restrict duplicate values, if they has to be restricted we usethe **Unique Constraints.**

* 1. **Unique:** If it is imposed on a column or column they will not allow duplicate Values into it.

Note: Unique, Primary Key, Check and Foreign Key Constraints can be imposed in two different ways:

-Column Level Definition

-Table Level Definition

Column Level Definition: In this case the constraint definition is immediately followed after the column definition. The Syntax is:

**-CREATE TABLE<table\_name>**

**(column\_name1<dtype>[width] [ [CONSTRAINT <Name>] <Type>],**

**column\_name1<dtype>[width] [ [CONSTRAINT <Name>] <Type>],**

**…............................**

**column\_namen<dtype>[width] [ [CONSTRAINT <Name>] <Type>],**

Recreating the bank table with Unique Constraint on it:

**CREATE TABLE Bank (Custid int Unique, Cname varchar(50), Bal decimal(7,2) Not Null)**

After creating this if we try to insert a duplicate values into the Custid column it will restrict us:

INSERT INTO Bank VALUES (101, 'RAJA', 3500)

INSERT INTO Bank VALUES (101, 'BASITH', 5555)

Note: Internally Unique, Primary Key, Check and Foreign Key Constraints are identified by using some unique name which has to be given by us or else the system will give a name, so when we violate these constraints it will show the name of the constraint in the error message, by seeing which we require to identify on which column we are getting the problem, but if the table is not created by us or we don't remember the structure of the table we cannot identify. So it is advised to give a name to the constraint so that when it violates the error message shows the name of the constraint using which we can easily identify where the violation has been done.

While giving a name to the Constraint they follow some conventions like:

<ColumnName\_ConstraintType>

<TableName\_ColumnName\_ConstraintType>

Recreating the bank table with Unique constraint by giving a name to it:

**CREATE TABLE Bank(Custid int Constraint Cusitd\_UQ Unique, Cname varchar(50), Bal decimal(7,2) Not Null)**

Table Level Definition: In this case the constraint definition is immediately followed after the column definition. The Syntax is

**-CREATE TABLE<table\_name>**

**(column\_name1<dtype>[width],**

**column\_name1<dtype>[width],**

**…............................**

**column\_namen<dtype>[width]),**

**[ [Constraint <Name>] <Type> (<Collist>)],**

**…........................)**

**Note: A Not Null Constraint Cannot be defined table level.**

**CREATE TABLE Bank(Custid int, Cname varchar(50), Bal decimal(7,2) Not Null, Constraint Custid\_UQ Unique(Custid))**

-In this case because the constraint is definedin the end of all columns it cannot understand to which column the constraint depends so we need to specify the column name beside the constraint for identification.

-There will not be any difference in execution whether the constraint is defined in the table level or column level.

-When we define a constraint in table level we can define composite constraint i.e a single constraint on multiple column.

**CREATE TABLE BankDetails(CityCode varchar(10), BranchCode varchar(10), Constraint CC\_BC\_UQ Unique(CityCode, BranchCode))**

**INSERT INTO BankDetails Values('C1', 'B1')**

**INSERT INTO BankDetails Values('C1', 'B2')**

**INSERT INTO BankDetails Values('C1', 'B3')**

**INSERT INTO BankDetails Values('C2', 'B1')**

**INSERT INTO BankDetails Values('C2', 'B2')**

**INSERT INTO BankDetails Values('C2', 'B3')**

-In this case all the statement are valid because a composite unique constraint check the uniqueness on the combination of column, but not on a single column.

The drawback with Unique Constraint is even if it restricts duplicate values it will allow a single null value in to the column. If we want to restricted duplicate values as well as null values we need to use **Primary Key Constraint.**

-While Creating a primary key constraint we need to keep this in mind i.e a table can contain only a single primary key present on it which can be present on a single column or multiple column also.

Creating a Primary Key in column level:

**CREATE TABLE Bank(Custid int Constraint Cusitd\_PK Primary Key, Cname varchar(50), Bal decimal(7,2) Not Null)**

Creating a Primary Key in table level:

**CREATE TABLE Bank(Custid int, Cname varchar(50), Bal decimal(7,2) Not Null, Constraint Cusitd\_PK Primary Key (Custid))**

Creating a Composite Primary Key in table level:

**CREATE TABLE BankDetails(CityCode varchar(10), BranchCode varchar(10), Constraint CC\_BC\_PK Primary Key(CityCode, BranchCode))**

* 1. **Check Constraint:** If we want to check the values present in a column to be according to a specified value we use this constraint.

-If we want to restrict the Bal in the bank table should be some specified values then we can use the constraint as following:

CHECK (Bal>=1000)-> Checking Bal should be some greater than equal to 1000

CHECK (Bal BETWEEN 1000 AND 9999)-> Checking Bal should be with in the range of 1000 and 9999

CHECK (Bal IN(3000,5000,7000)-> Checking Bal should be with in any of the three values only

Creating Check Constraint in Column level:

**CREATE TABLE Bank(Custid int, Cname varchar(50), Bal decimal(7,2),**

**Constraint Bal\_CK Check (Bal>=1000))**

Creating Check Constraint in Table level:

**CREATE TABLE Bank(Custid int, Cname varchar(50), Bal decimal(7,2),**

**Constraint Bal\_CK Check (Bal BETWEEN 1000 AND 9999))**

* 1. **Default value:** The default value for any column if a not null constraint is not present on it is “NULL”, which can be changed by using the Default Clause while creating the table as following:

**CREATE TABLE Bank(Custid int, Cname varchar(50), Bal decimal(7,2),**

**Default 1000)**

-In the above case if have not specified any value to the Bal column while inserting then it takes1000 as default

**INSERT INTO Bank (Custid,Cname) VALUES(101, 'BASITH')**

**Identity Function:** Generally for any column if we want to insert only unique values then we can hand over the task to the identity function, so that it takes the responsibility of inserting a unique value in to the column as following:

**Colname<dtype> [width] identify [(Seed,Incr)]**

Seed – It is the starting value for the identity function.

Incr – It is the difference between to subsequent values generated by the functions.

-Both of them are optional, if not specified 1 and 1 are taken as values.

-When we use the identity function on a column we cannot explicitly insert any values into the column using the insert statement.

-A table can have only one identity column present in it.

-Generally we this on Primary Key Columns.

**CREATE TABLE Bank(Custid int identity(101, 1), Cname varchar(50), Bal decimal(7,2))**

-In this case when we insert rows into the table then it automatically generates a identity value starting from 101.

**INSERT INTO Bank (Cname, Bal) values (‘Raja’, 3500)**

* 1. **Foreign Key Constraint:** it is column or combination of column that is used to establish and enforce a link between the data in two tables. In a foreign key reference, a link is created between two tables when the column(s) in a table reference the column(s) that hold the primary key of other table, which becomes a foreign key in the first table.

For example, the **Dept.Deptno** table below has a link to the **Emp.Deptno** table because there is a logical relationship between **Dept** table and **Emp** table. The **Deptno** column in the **Em**p table matches the primary key column of the Dept table. The **Deptno** column in the **Emp** table is the foreign key to the **Dept** table. In this case the value that is going to be inserted into the **Deptno** column of the **Emp** table should be present in the **Deptno** column of the **Dept** table or should be a null values.

- In this case the Deptno table is called as parent table and Emp table is called as Child table.

- Dept.Deptno is called as Reference Key column on which either Primary Key Constraint or Unique Constraint has to be imposed.

- Emp.Depto is called as Foreign Key column on which the Foreign Key Constraint has to be

imposed. with this only the link gets established between the 2 tables.

**Create Table Dept (Deptno int Constraint Deptno\_PK Primary Key, Dname varchar(50),**

**Loc varchar(50))**

**1.Insert into Dept values(10, 'Marketing', 'Mumbai')**

**2.Insert into Dept values(20, 'Sales', 'Chennai')**

**3.Insert into Dept values(30, 'Finance', 'Delhi')**

**4.Insert into Dept values(40, 'Production', 'Kolkota')**

- Creating Foreign Key Constraint in column level:

**Create table** **Emp (Empno int, Ename varchar(50), Job varchar(50),**

**Mgr int, HireDate datetime, Sal Money, Comm Money,**

**Deptno int Constraint Deptno\_Refrence Dept (Deptno))**

- Creating Foreign Key Constraint in column level: while defining constraint in table level we need to explicitly use the Foreign Key caluse to specify the Foreign Key Column:

**Create table** **Emp (Empno int, Ename varchar(100), Job varchar(100),**

**Mgr int, HireDate datetime, Sal Money, Comm Money,**

**Deptno int Constraint Deptno\_Ref Foreign Key Dept (Deptno) Refrence Dept (Deptno))**

- Now when we try to insert values into the Emp table the Deptno what we give should be only the 4 values (10,20,30,40) present in the dept table or null value, if we try to insert any other value the insert statement fails.

**1.Insert into Emp Values (1001,'Basith', 'President', NULL, '10/14/1991',**

**5000, NULL, 10)**

**2.Insert into Emp Values (1002,'Abdul', 'Manager', 1001, '10/14/1991',**

**4000, NULL, 20)**

**3.Insert into Emp Values (1003,'Abu', 'Manager', 1001, '10/14/1991',**

**3500, NULL, 30)**

**4.Insert into Emp Values (1004,'Basi', 'Manager', 1001, '10/14/1991',**

**4000, NULL, 40)**

**5.Insert into Emp Values (1005,'Raja', 'Salesman', 1003, '10/14/1991',**

**3000, NULL, 50)**

- In this case the first 4 statement gets executed but last statement fails because the Deptno given is not present in the dept table.

The Foreign Key constraint enforce referential integer by guaranteeing that change cannot be made to data in the primary key table if those changes invalidate the link to data in the foreign key table. If an attempt is made to delete the row in a primary key table or to change a primary key value, the action will fail when the deleted or changed primary key value corresponds to a value in the FOREIGN KEY constraint of another table. To successfully change or delete a row in a FOREIGN KEY constraint, you must first either delete the foreign key table or change the foreign key data in the foreign key table, which links the foreign key to different primary key data.

By using cascading referential integrity constraint, you can define the action that the SQL Server 2005 takes when a user tries to delete or update a key value in the master table to which existing foreign keys point.

The REFERENCES clauses of the CREATE TABLE statement support the ON DELETE and ON UPDATE clauses:

-ON DELETE<NO ACTION | CASCADE | SET NULL | SET DEFAULT>

-ON UPDATE<NO ACTION | CASCADE | SET NULL | SET DEFAULT>

NO ACTION is the default if ON DELETE OR ON UPDATE is not specified.

ON DELETE NO ACTION: Specifies that if an attempt is made to delete a key value in the master table, which is referenced by foreign key in other tables, an error is raised and the DELETE statement will not execute.

ON UPDATE NO ACTION: Specifies that if an attempt is made to delete a key value in the master table, which is referenced by foreign key in other tables, an error is raised and the UPDATE statement will not execute.

when NO ACTION is Specified we can delete or update or delete rows in the master table, which is referenced by child table rows, but we can perform those operations when we use CASCADE, SET NULL AND SET DEFAULT CLAUSES:

ON DELETE CLAUSES: Specifies that if an attempt is made to delete a key value in the master table, which is referenced by foreign key in other tables, all rows that contain those foreign key in child table are also deleted.

ON UPDATE CLAUSES: Specifies that if an attempt is made to update a key value in the master table, which is referenced by foreign key in other tables, all the foreign key values will also be updated to the new value specified for the key.

ON DELETE SET NULL: Specifies that if an attempt is made to delete a key value in the master table, which is referenced by foreign key in other tables, all rows that contain those foreign key in child table are set to NULL, provides the foreign key column allow NULL values into it.

ON UPDATE SET NULL: Specifies that if an attempt is made to update a key value in the master table, which is referenced by foreign key in other tables, all rows that contain those foreign key in child table are set to NULL, provides the foreign key column allow NULL values into it.

ON DELETE SET DEFAULT: Specifies that if an attempt is made to delete a key value in the master table, which is referenced by foreign key in other tables, all rows that contain those foreign key in child table are set to default value. All foreign key columns of the target table must have a default definition for this constraint to execute. If there is no explicit default value set, NULL become the implicit default value of the column.

ON UPDATE SET DEFAULT: Specifies that if an attempt is made to update a key value in the master table, which is referenced by foreign key in other tables, all rows that contain those foreign key in child table are set to default value. All foreign key columns of the target table must have a default definition for this constraint to execute. If there is no explicit default value set, NULL become the implicit default value of the column.

We can use any of the rule beside the column as following:

**Deptno int Constraint Deptno\_Ref references Dept (Deptno) on delete cascade on update cascade**

In the same way you can any rule there and also not mandatory to specify both the delete and update rule we can use any rule what we require.

**Alter Command:** After creating a table we want to make any modification to the structure of the table we use the alter Alter Command. Using alter command we can perform the following:

- Increase/Decrease the width of column.

- Change the data type of column.

- Change Null to Not Null and Not Null to Null

- Add a new column to the table.

- Drop an existing column from the table.

- Add a constraint to a column of the table.

- Drop an existing constraint present on a column from the table.

- To perform the first 3 operations the syntax is:

**ALTER TABLE <TNAME> ALTER COLUMN <COLNAME> <DTYPE> [WIDTH] [NULL | NOYNULL]**

First create a table as following:

**CREATE TABLE Students (SNO int, Sname varchar(50), Class int)**

Increasing the width of a column:

**ALTER TABLE Student ALTER COLUMN Sname varchar(100)**

Decreasing the width of a column:

**ALTER TABLE Student ALTER COLUMN Sname varchar(25)**

Changing the data type of the column:

**ALTER TABLE Student ALTER COLUMN Sname nvarchar(25)**

Adding a Not Null Constraint:

**ALTER TABLE Student ALTER COLUMN Sname nvarchar(25) Not Null**

Removing a Not Null Constraint:

**ALTER TABLE Student ALTER COLUMN Sname nvarchar(25) Null**

Syntax to add a new column:

**ALTER TABLE <TNAME> ADD <COLNAME> <DTYPE> | <WIDTH> | [ [CONSTRAINT <CONS NAME>]<CONS TYPE> ]**

Adding a column to the table with out any constraint:

**ALTER TABLE Students ADD Fees Money**

Adding a column to the table with a constraint:

**ALTER TABLE Students ADD Sid int Constraint Sid\_UQ UNIQUE**

Syntax to drop a existing column:

**ALTER TABLE <TNAME> DROP COLUMN <COL NAM>**

Dropping the Sid Column:

**ALTER TABLE Students DROP COLUMN Sid**

Syntax to Add a constraint:

**ALTER TABLE <TNAME> ADD [CONSTRAINT <CONS NAME>]<CONS TYPE> (COLLIST)**

Adding a check constraint on the Fees column:

**ALTER TABLE Student ADD Constraint Fees\_CK Check (Fees>1500)**

Adding a primary key constraint on the Sno column:

**ALTER TABLE Student ALTER COLUMN SNO INT NOT NULL**

**ALTER TABLE Student ADD Constraint Sno\_PK Primary Key(SNO)**

Syntax to Drop a constraint:

**ALTER TABLE <TNAME> DROP CONSTRAINT <CONS NAME>**

Dropping the check constraint present on the Fees column:

**ALTER TABLE Student DROP Constraint Fees\_CK**

**Drop Command:**If we want to destroy the existing tables persent in the database we use the Drop Command.

Syntax: **DROP TABLE <TNAME>**

Dropping the students Table:

**DROP TABLE Student**

**Truncate Command:** Removes all rows from a table. TRUNCATE TABLE is functionally the same as the DELETE statement with no WHERE clause specified.

Syntax: **TRUNCATE TABLE <TNAME>**

Truncating the EMP Table:

**TRUNCATE TABLE Student**

The difference between Truncate and Delete is:

1. Truncate table is faster in execution.
2. Truncate will reset the identity function if present on the table to initial value again which will not happen in delete.
3. **FUNCTIONS:** SQL Server 2005 provides built-in function that can be used to perform certain operations. Functions can be used or included in the following:

- The select list of a query that uses a SELECT statement to return a value.

- A WHERE clauses search conditions of a SELECT statement to limit the rows that qualify for the query.

Syntax for executing a function:

SELECT<Fun Name>([<expressions>])

- The expressions can be a constant values or a name of a column.

Functions can be classified into 2 types:

- Single Row Functions

- Group Functions

A single row function executes once for each row that is present in the table where as group function take multiple rows into consideration and returns a single value as output.

Single Row Function Categories:

- Mathematical Function

- String Function

- Date and time Function

- System Function

Mathematical Function: These function perform a calculation, usually based on input values that are provided as arguments, and return a numeric value; they “n” as input where n is numeric expression.

ABS (n): A mathematical function that returns the absolute (positive) value of the specified numeric expression.

Select ABS(10) Output: 10

Select ABS(-10) Output: 10

CEILING (n):Returns the smallest integer greater than, or equal to, the specified numeric expression.

SELECT CEILING(15.6) Output: 16

SELECT CEILING(15.6) Output: -15

CEILING (n):Returns the largest integer less than, or equal to, the specified numeric expression.

SELECT CEILING(15.6) Output: 15

SELECT CEILING(15.6) Output: -16

LOG(n):Return the natural logarithm of the specified expression, i.e base-e

SELECT LOG(10) OUTPUT:2.30258509299405

LOG10(n):Return base-10 logarithm of the specified expression, i.e base-e

SELECT LOG10(10) OUTPUT: 1

P1(): Return the constant value of P1.

SELECT P1() OUTPUT: 3.14159265358979

POWER(n, m): Return the value of the specified expression n to the specified power m.

SELECT POWER(10, 3) OUTPUT: 1000

RAND ([SEED]): Returns a random **float** value from 0 through 1.

- SEED: Is an integer expression that gives the seed value. If seed is not specified, the Database Engine assigns a seed value at random. For a specified seed value, the result returned is always the same.

SELECT RAND() - Each time we execute we get a random value.

SELECT RAND(100) - Each time we execute we get the same value.

ROUND (n, length [,function]): Returns a numeric expression, rounded to the specified length or precision.

SELECT ROUND(156.567,2)OUTPUT: 156.57

SELECT ROUND(156.567,1)OUTPUT: 156.6

SELECT ROUND(156.567,0)OUTPUT: 157

- If the seed is positive rounding will be done after the decimal, if it is negative rounding will be done before the decimal:

SELECT ROUND(156.567,-1)OUTPUT: 160

SELECT ROUND(156.567,-2)OUTPUT: 200

- If we specified the optional parameter function that is an integer value we can decide to truncate the value or round the value. If it is 0(default) rounds the value and value greater than 0 truncates the value.

SELECT ROUND(156.567,2, 1)OUTPUT: 156.56

SELECT ROUND(156.567,-2, 1)OUTPUT: 100

SIGN(n): Returns the positive (+1), zero (0), or negative (-1) sign of the specified expression.

- If n<0 it returns -1

- If n=0 it returns 0

- If n>0 it returns 1

SELECT SIGN(-100) OUTPUT: -1

SELECT SIGN(0) OUTPUT: 0

SELECT SIGN(100) OUTPUT: 1

SQRT(n): Returns the square root of the specified expression.

SELECT SQRT(81) OUTPUT: 9

SELECT SQRT(30) OUTPUT: 5.47722557505166

SQUARE(n): Returns the square root of the specified expression.

SELECT SQUARE(35) OUTPUT: 1225

- Apart from the above it provides with trigonometric function like COS, COT, SIN, TAN, ACOS, ASIN, ATAN for which we need to provide the degrees.

String Function these functions perform an operator on a string input value and return a string or numeric value.

ASCII(n):Return the ASCII code value of the leftmost characters of the expression.

ASCII('A') OUTPUT: 65

ASCII('BCD') OUTPUT: 66

CHAR(n): Converts the given ASCII code to character.

CHAR(97) OUTPUT: a

NCHAR(n): Return the Unicode character with the specified integer code ranging between 0 to 65,535, as defined by the Unicode standard.

CHAR(300) OUTPUT: Ĭ

CHARINDEX(search exp, string exp, [, start\_location ] ): Return the starting position of the search exp in the string exp which can also be a column name.

CHARINDEX('O', 'HELLO WORLD') OUTPUT: 5

- In this case it return 5 as output because it start its search form the beginning of the string, we can change it by using the start location option parameter.

CHARINDEX('O', 'HELLO WORLD', 6) OUTPUT: 8

- WAQ to get the details of employees whose name contains the character 'M' in it.

Sol: SELECT \* FROM EMP WHERE CHARINDEX('M', ENAME)>0

LEFT( , n): Return the left part of the string with the specified number of characters.

SELECT LEFT('HELLO', 3) OUTPUT: HEL

- WAQ to get the details of employees whose name contains the first 2 characters as 'VE'.

Sol: SELECT \* FROM EMP WHERE LEFT(ENAME, 2)> 'VE'

RIGHT( , n): Return the RIGHT part of the string with the specified number of characters.

SELECT RIGHT('HELLO', 3) OUTPUT: LLO

- WAQ to get the details of employees whose name ends with characters 'TT'.

Sol: SELECT \* FROM EMP WHERE RIGHT(ENAME, 2)= 'TT'

SUBSTRING(s, start, length): Returns a part of a string from string s starting from start position, where length is the no of chars to be picked.

SELECT SUBSTRING('HELLO', 1, 3) OUTPUT: HEL

SELECT SUBSTRING('HELLO', 3, 3) OUTPUT: LLO

SELECT SUBSTRING('HELLO', 2, 3) OUTPUT: ELL

1. WAQ to get the details of employees whose names 3rd and 4rd Characters are 'TI'.

Sol: SELECT \* FROM EMP WHERE RIGHT(LEFT(ENAME. 4), 2)= 'TI'

Sol: SELECT \* FROM EMP WHERE SUBSTRING(ENAME. 3, 2)= 'TI'

LEN(s): Returns the number of the specified string expression, excluding trailing blanks.

SELECT LEN('HELLO') OUTPUT: 5

SELECT LEN(' HELLO') OUTPUT: 8

- WAQ to get the details of employees whose name was 5 characters in length

Sol: SELECT \* FROM EMP WHERE LEN(ENAME)= 5

SELECT LEN('HELLO ') OUTPUT: 5

LOWER(s): Return a character expression after converting the given character data to lowercase.

SELECT LOWER('HELLO') OUTPUT: hello

UPPER(s): Return a character expression after converting the given character data to uppercase.

SELECT UPPER('Hello') OUTPUT: HELLO

LTRIM(s): Return a character expression after removes leading blanks.

SELECT LEN(LTRIM(' HELLO') OUTPUT: 5

SELECT 'HELLO' + LTRIM(' WORLD') OUTPUT: HELLO WORLD

RTRIM(s): Return a character expression after removes trailing blanks.

SELECT RTRIM ('HELLO') + 'WORLD' OUTPUT: HELLO WORLD

REPLACE(s1, s2, s3): Replace all occurrence of the s2 in s1 with s3.

SELECT REPLACE('HELLO', 'L', 'X') OUTPUT: HEXXO

REPLICATE(s, n): Repeat the expression 's' for specified 'n' number of times.

SELECT REPLICATE('HEL', 2) OUTPUT: HELHEL

REVERSE(S): Return the reverse of the given string 's'.

SELECT REVERSE ('HELLO') OUTPUT: OLLEH

SOUNDEX(s): Returns a four-character (SOUNDEX) code to evaluate the similarity of two strings. SOUNDEX converts an alphanumeric string to a four-character code to find similar-sounding words or names. The first character of the code is the first character of string and the second through fourth character of the code are numbers.

SELECT SOUNDEX ('Smith'), SOUNDEX ('smyth')

- Generally we use then when we perform comparison of the word, which are sounded in the same way but have different spelling like color & colour. Suppose in a table the ename of a person is smith we will get the result even if the statement is written as following:

SELECT \* FROM EMP WHERE SOUNDEX(ENAME)=SOUNDEX('SMYTH')

DIFFERENCE(S1, S2): Return an integer value that indicates the difference between the SOUNDEX value of two character expressions. The return value ranges from 0 through 4:0 indicates weak or no similarity, and 4 indicates strong similarity or the same values.

SELECT SOUNDEX('SMITH'), SOUNDEX('SMYTH'),

DIFFERENCE('SMITH', 'SMYTH')

SPACE(n): Return a string with specified 'n' number of repeated space.

SELECT 'HELLO'+SPACE(1)+'WORLD' OUTPUT: HELLO WORLD

STUFF(s, start, length, replace\_str): Replace specified length of characters from specified starting point with replace\_str in the string 's'

SELECT STUFF('ABXXCDXX', 3, 3, 'YY') OUTPUT: ABYYDXX

Date and time Functions: The following function perform an operation on date and time input value and return a string, numeric, or date and time value.

GETDATA(): Return the current date and time of the server in SQL Server standard internal format.

SELECT GETDATA()

DAY(date): Returns as integer representing the DAY of the specified date, which has to be specified in standard SQL Server date format 'mm/ddd/yyyy'.

SELECT DAY(GETDATA())

SELECT DAY('07/10/20013') OUTPUT: 10

MONTH(date): Returns as integer representing the MONTH of the specified date, which has to be specified in standard SQL Server date format 'mm/ddd/yyyy'.

SELECT DAY(GETDATA())

SELECT MONTH('07/10/20013') OUTPUT: 07

YEAR(date): Returns as integer representing the YEAR of the specified date, which has to be specified in standard SQL Server date format 'mm/ddd/yyyy'.

SELECT DAY(GETDATA())

SELECT YEAR('07/10/20013') OUTPUT: 2013

DATENAME(datepart, date): Returns a character string representing the specified datepart of the specified date, datepart is the parameter that specifies that part of the date to return. The following table list datepart and abbreviations recognized by Sql Server:

|  |  |
| --- | --- |
| **Datepart** | **Abbreviations** |
| **year** | **yy, yyyy** |
| **quarter** | **qq, q** |
| **month** | **mm, m** |
| **dayofyear** | **dy, y** |
| **day** | **dd, d** |
| **week** | **wk, ww** |
| **weekday** | **dw** |
| **hour** | **hh** |
| **minute** | **mi, n** |
| **second** | **ss, s** |
| **millisecond** | **ms** |

SELECT DATENAME(mm, '07/10/20013') OUTPUT: October

SELECT DATENAME(dd, '07/10/20013') OUTPUT: 10

DATEPART(datepart, date): This is same as DATENAME function but the only difference is weekday(dw) of DATEPART function returns a number that corresponds to the day of the week, for example: Sunday = 1, Saturday = 7, where as in the case of DARENAME returns the value in string format that is Sunday, Monday,......Saturday.

DATEADD(datepart, number, date): Return a new datetime value based on adding an interval to the specified date, datepart is the value that has to be added and number is the interval.

SELECT DATEADD(dd, 30, GETDATA())-Adds 30 days to GETDATA()

SELECT DATEADD(mm, 16, GETDATA())-Adds 16 months to GETDATA()

DATEDIFF(datepart, startdate, enddate): Returns the difference between the start and end dates in the give datepart format.

SELECT DATEDIFF(yy,'07/10/20013', GETDATA())

GETUTCDATA()-Returns the datetime value respresenting the current UTC time (Coordinated Universal Time or Greenwich Mean Time).

SELECT GETUTCDATA()

Conversion Functions: Explicitly converts an expression of one data type to another. We has two conversion functions CAST and CONVERT, both provide similar functionality.

Syntax for CAST:

CAST(expression AS data\_type[ (length) ] )

SELECT CAST (10.6496 AS INT) OUTPUT: 10

SELECT CAST (10.3496847 AS money) OUTPUT: 10.3497

Syntax for CONVERT:

CONVERT(data\_type[ (length) ], expression [ , style] )

SELECT CONVERT (INT10.6496) OUTPUT: 10

SELECT CONVERT (VARCHAR(50), GETDATA())

Style is an optional parameter that can be used to specify a date format used to convert **datetime** or **smalldatetime** data to character. When style is NULL, the result returned is also NULL, Style is NULL, the result returned is also NULL, Style can be used as following:

SELECT CONVERT (VARCHAR(50), GETDATE(), 101)

SELECT CONVERT (VARCHAR(50), GETDATE(), 102)

- Each style will give the output of the date in a different format the default style it uses is 100. The style values can be ranging between 100-114, 120, 121, 126, 127, 130 and 131 or 0 to 8, 10, 11, 12 and 14 in this case century part will not returned.

SELECT CONVERT (VARCHAR(50), GETDATA(), 1)

System Function:

ISNUMERIC(expression): Determines whether an expression is a valid numeric type. If it is numeric it returns 1 else return 0.

SELECT ISNUMERIC(100) OUTPUT: 1

SELECT ISNUMERIC('100') OUTPUT: 1

SELECT ISNUMERIC('100A') OUTPUT: 0

ISDATE(expression): Determines whether an input expression is a valid date or not. If it is a valid date it returns 1 else return 0. Valid date in the sense the expression, which is present in mm/dd/yyyy format.

SELECT ISDATE('07/10/20013') OUTPUT: 1

SELECT ISDATE('21/07/20013') OUTPUT: 0

ISNULL(expression1,expression2): If expression1 is null then it returns expression2.

SELECT ISNULL(100, 200) OUTPUT: 100

SELECT ISNULL(NULL, 200) OUTPUT: 200

SELECT EMPNO, ENAME, SAL, COMM, SAL + COMM AS [TOTAL SAL] FROM EMP

- In above case if any of the value in the comm. Is null in the Total Sal because any arithmetic operations performed on null value results to null only at this time the statement has to be written as following:

SELECT EMPNO, ENAME, SAL, COMM, SAL + ISNULL(COMM, 0) AS [TOTAL SAL] FROM EMP

COALESCE (expression1,expression2, …... expression n): Return the first not null expression in the list of expressions given, similar to isnull but we can give multiple values here.

SELECT COALESCE(NULL, 100, NULL, 200) OUTPUT: 100

SELECT EMPNO, ENAME, SAL, COMM, SAL + COALESCE(COMM, 0) AS

[TOTAL SAL] FROM EMP

DATALENGTH (expression): Return the number of bytes used to represent any expression.

SELECT DATALENGTH(100) OUTPUT: 4

SELECT DATALENGTH('HELLO') OUTPUT: 5

HOST\_NAME(): Return the name of the workstation.

SELECT HOST\_NAME()

IDENT\_CURRENT('table\_name'): Return the last identity value generated for a specified table by the identity function.

SELECT IDENT\_CURRENT('BANK')

IDENT\_SEED('table\_name'): Return the seed value that was specified when the identity function in a table was created.

SELECT IDENT\_SEED('BANK')

IDENT\_INCR('table\_name'): Return the increment value that was specified when the identity function in a table was created.

SELECT IDENT\_INCR('BANK')

NEWID(): Create a unique value of type uniqueidentifier.

SELECT NEWID()

NULLIF(expression1,expression2): Return the first expression if the two expression are not equivalent. If the expression are equivalent, returns a null value.

SELECT NULLIF(100, 200) OUTPUT: 100

SELECT NULLIF(100, 100) OUTPUT: NULL

ROWCOUNT\_BIG(): Return the number of rows affected by the last statement executed. If we use this after a select statement it will return us the number of rows the select statement has returned.

SELECT \* FROM EMP

SELECT ROWCOUNT\_BIG FROM EMP

APP\_NAME(): Returns the name of the application from where the statement is executed.

SELECT APP\_NAME()

1. CASE: Evaluates a list of conditions and return one of multiple possible result expressions. It has two formats:

- The simple CASE function compares as expression to set of simple expressions to determine the result.

- the searched CASE function evaluates a set of Boolean expression to determine the result.

- Both formats support an optional ELSE argument.

CASE <expression>

WHEN when\_expression THEN result\_expression

WHEN when\_expression THEN result\_expression

…......................................

ELSE else\_result\_expression

END

- In this case if the expression matches with any of the when\_expression it returns the corresponding result\_expression, if it does not match any then it returns else\_result\_expression.

SELECT EMPNO, ENAME, SAL, JOB,

(CASE JOB

WHEN 'PRESIDENT' THEN 'BIG BOSS'

WHEN 'MANAGER' THEN 'BOSS'

WHEN 'ANALYST' THEN 'SCIENTIST'

ELSE 'EMPLOYEE'

END) AS COMMENTS FROM EMP

SELECT EMPNO, ENAME, JOB, SAL,

(CASE SING(SAL-3000)

WHEN 1 THEN 'ABOVE TARGET'

WHEN 0 THEN 'ON TARGET'

WHEN -1 THEN 'BELOW TARGET'

END) AS COMMENTS FROM EMP

- The above statement can be written in one more way also by using the second format of the CASE function.

CASE

WHEN condition THEN result\_expression

WHEN condition THEN result\_expression

…......................................

ELSE else\_result\_expression

END

SELECT EMPNO, ENAME, JOB, SAL,

(CASE

WHEN SAL>3000 THEN 'ABOVE TARGET'

WHEN SAL=3000 THEN 'ON TARGET'

WHEN SAL<3000 THEN 'BELOW TARGET'

END) AS COMMENTS FROM EMP

1. Set Operators:

COUNT(expression): Return the number of items in a group.

SELECT COUNT(\*)FROM EMP

SELECT COUNT(\*)FROM EMP WHERE DEPTNO=20

SELECT COUNT(COMM)FROM EMP

COUNT\_BIG(expression): COUNT\_BIG work like the COUNT function. The only difference between the two functions is their return value. COUNT\_BIG always return a **bigint** data type value. COUNT always returns an **int** data type value.

SELECT COUNT\_BIG(\*)FROM EMP

SUM(expression): Return the sum of all the value. SUM can be used with numeric columns only. Null values are ignored.

SELECT SUM(SAL) FROM EMP

AVG(expression): Return the average of the values in group. Null values are ignored.

SELECT AVG(SAL) FROM EMP

MAX(expression): Return the maximum value in the expression.

SELECT MAX(SAL) FROM EMP

MIN(expression): Return the minimum value in the expression.

SELECT MIN(SAL) FROM EMP

STDEV(expression): Return the statistical standard deviation of all values in the specified expression.

SELECT STDEV(SAL) FROM EMP

VAR(expression): Return the statistical variance of all values in the specified expression.

SELECT VAR(SAL) FROM EMP

Operators: An operator is symbol specifying an action that is performed on one or more expression. The lists the operator categories that SQL Server support:

- Arithmetic Operators

- Assignment Operators

- Comparison Operators

- Logical Operators

- Concatenation Operators

1. Arithmetic Operators: Arithmetic Operators perform mathematical operations on two expression of one or more of the data types of the numeric data type category.

Those are:

+ - Addition

- - Subtraction

\* - multiplication

/ - Division

% - Modulo

1. Assignment Operators: The equal sign (=) is the only assignment operator.
2. Comparison Operators: Comparison operator test whether two expression are the same. Comparison operators Can be used on all expression of the **text**, **ntext** or **image** data types.

Those are:

= - Equal to

> - Greater than

< - Less than

>= - Greater than or Equal to

<= - Less than or Equal to

!= - not equal to

!> - not greater than

!< - not less than

1. Logical Operators: Logical operators test for the truth of some condition. Logical operators, like comparison operators, return a **Boolean** value of TRUE or FALSE.

Those are:

1. ALL TRUE if all of a set comparison are TRUE
2. AND TRUE if both Boolean expression are TRUE
3. ANY TRUE if any one of a set of comparison are TRUE
4. BETWEEN TRUE if the operand is within a range
5. EXISTS TRUE if a subquery contains any rows
6. IN TRUE if the operand equal to one of a list of expressions.
7. LIKE TRUE if the operand matches a pattern
8. NOT Reverses the value of any other Boolean operator
9. OR TRUE if either Boolean expression is TRUE
10. SOME TRUE is some of a set of comparison are TRUE

String Concatenation Operators: The plus sign (+) is the string concatenation operator that enables string concatenation.

- WAQ to find the details of employees whose job is CLERK.

Sol: SELECT \* FROM EMP WHERE JOB='CLERK'

- WAQ to find the details of employees except SALESMAN

Sol: SELECT \* FROM EMP WHERE JOB !='SALESMAN'

**(OR)**

Sol: SELECT \* FROM EMP WHERE JOB <>'SALESMAN'

- WAQ to find details of employees who are earning more than 3000

Sol: SELECT \* FROM EMP WHERE SAL<3000

- WAQ to find details of employees who are earning less than 2500

Sol: SELECT \* FROM EMP WHERE SAL<2500

- WAQ to find details of employees who are earning with in a range of 2500 and 4000

Sol: SELECT \* FROM EMP WHERE SAL>=2500 AND SAL<=4000

Sol: SELECT \* FROM EMP WHERE SAL BETWEEN 2500 AND 4000

- Between operator is used for specifying with a range of values to test.

- WAQ to find details of employees who are earning less than 1500 as well as more than 3500

Sol: SELECT \* FROM EMP WHERE SAL<1500 OR SAL>3500

Sol: SELECT \* FROM EMP WHERE SAL NOT BETWEEN 1500 AND 3500

- WAQ to find details of employees whose jobs are CLERK, MANAGER AND SALESMAN

Sol: SELECT\* FROM EMP WHERE JOB='CLERK' OR JOB='MANAGER' OR JOB='SALESMAN'

Sol: SELECT \* FROM EMP WHERE JOB IN ('CLERK', 'MANGER', 'SALESMAN')

- In Operator Determines whether a specified value matches any value in the list.

- WAQ to find details of employees expect PRESIDENT AND MANAGER.

Sol: SELECT\* FROM EMP WHERE JOB != 'MANAGER' AND JOB != 'PRESIDENT'

Sol: SELECT\* FROM EMP WHERE JOB NOT IN ('MANAGER', 'PRESIDENT')

- WAQ to find details of employees who name starts with character S.

Sol: SELECT\* FROM EMP WHERE ENAME LIKE 'S%'

- Like Operator determines whether a specific character string matches a specified pattern. A pattern can include regular characters and wildcard characters. During pattern matching, regular character must exactly match the characters specified in the character string. However, wildcard characters can be matched with arbitrary fragment the character string. Using wildcard characters makes the LIKE operator more flexible than using the = and != string comparison operators.

% - it represents any of zero or more characters.

- WAQ to find details of employees whose name contains M in it.

Sol: SELECT\* FROM EMP WHERE ENAME LIKE '%M%'

- WAQ to find details of employees whose name is SMITH, when the spelling of the name is not known exactly as SMITH OR SMYTH.

Sol: SELECT \* FROM EMP WHERE ENAME LIKE 'SM\_TH'

**(OR**)

Sol: SELECT \* FROM EMP WHERE SOUNDEX(ENAME)=('SMYTH')

\_(underscore)- it represents any single character.

- WAQ to find details of employees whose name start with a characters between A to S.

Sol: SELECT \* FROM EMP WHERE ENAME LIKE '[A-S]%'

[] - I represents any signal character within the specified range ([a-f]) or set ([abcdef]).

WAQ to find the details of employees whose name starts with any of the character “ABCDE”

Sol: SELECT \* FROM EMP WHERE ENAME LIKE '[ABCDE]%'

WAQ to find the details of employees whose name starts with a characters not between A to S.

Sol: SELECT \* FROM EMP WHERE ENAME LIKE '[^A-S]%'

WAQ to find the details of employees whose name starts with characters apart from “ABCDE”

Sol: SELECT \* FROM EMP WHERE ENAME LIKE '[^ABCDE]%'

OR

Sol: SELECT \* FROM EMP WHERE ENAME NOT LIKE '[ABCDE]%'

WAQ to find the details of employees whose job is CLERK and earning 3000

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

WAQ to find the details of employees whose job is MANAGER as well as earning more than 3000

Sol: SELECT \* FROM EMP WHERE JOB='MANAGER' OR SAL=3000

WAQ to find the details of employees whose salary is not equal to 3000

Sol: SELECT \* FROM EMP WHERE NOT SAL=3000

Set operators: Combines the results of two or more queries into a single result set.

The following are basic rules for combining the result sets of two queries by using SET Operators:

1. The number and the order of the columns must be the same in all queries.
2. The data types must be compatible.

UNION: Combines the result of two or more queries into a single result set that includes all the rows that belong to all queries in the union.

SELECT JOB FROM EMP WHERE DEPTNO=10

UNION

SELECT JOB FROM EMP WHERE DEPTNO=30

UNION ALL: These is name as UNION but in this case duplicates will not be eliminated.

SELECT JOB FROM EMP WHERE DEPTNO=10

UNION ALL

SELECT JOB FROM EMP WHERE DEPTNO=30

INTERSECT: Returns any distinct values that are returned by both the query on the left and right sides of the INTERSECT operand.

SELECT JOB FROM EMP WHERE DEPTNO=10

INTERSECT

SELECT JOB FROM EMP WHERE DEPTNO=30

EXCEPT: Returns any distinct values from the query to the left of the EXCEPT operand that are not also returned from the right query.

SELECT JOB FROM EMP WHERE DEPTNO=10

EXCEPT

SELECT JOB FROM EMP WHERE DEPTNO=30

1. CLAUSES: SQL Server provides with the following clauses that can be used in the SELECT statement:
2. WHERE
3. GROUP BY
4. HAVING
5. ORDER BY

The complete syntax of the SELECT statement look as following:

SELECT <select\_list> FROM <tname>

[WHERE search\_condition]

[GROUP BY group\_by\_condition]

[HAVING search\_condition]

[ORDER BY order\_expression [ASC | DESC] ]

WHERE Clause: The WHERE clause is a filter that defines the conditions each row in the source table must meet to qualify for the SELECT. Only rows that meet the conditions contribute data to the result set. Data from rows that do not meet the conditions is not used.

SELECT \* FROM EMP WHERE JOB='MANAGER'

SELECT \* FROM EMP WHERE DEPTNO=20

GROUP BY Clause: The GROUP BY clause partitions the result set into groups based on the values in the column of the *group\_by\_list*. For example, the **Emp** table has 3 values in **Deptno** columm. A GROUP BY **Deptno** clause partitions the result set into 3 groups, one for each value of **Deptno.**

WAQ to find the highest salaries for each department.

Sol: SELECT DEPTNO, MAX(SAL) FROM EMP GROUP BY DEPTNO

WAQ to find the highest salaries for each job.

Sol: SELECT JOB, MAX(SAL) FROM EMP GROUP BY JOB

WAQ to find the highest salaries for each department in it for each job.

Sol: SELECT DEPTNO, JOB, MAX(SAL) FROM EMP GROUP BY DEPTNO, JOB

Note: While using the GROUP BY clause the select\_list of the query should contain only the following:

1. Group Function or Aggregate Functionality
2. Column used in the Group By Clause
3. Constants.

WAQ to find the number of employees working for each department.

Sol: SELECT DEPTNO, COUNT(\*) FROM EMP GROUP BY DEPTNO

WAQ to find the number of employees working for each department only if the number is greater than 3.

Sol: SELECT DEPTNO, COUNT(\*) FROM EMP GROUP BY DEPTNO HAVING COUNT(\*)>3

HAVING Clause: The HAVING clause is an additional filter that applied to the result set, Logically, the HAVING clause filter rows from the intermediate result set build from applying any FROM, WHERE, or GROUP BY clause in the SELECT statement. HAVING clause are typically used with a GROUP BY clause.

WAQ to find the number of Clerk's working for each department.

Sol: SELECT DEPTNO, COUNT(\*) FROM EMP WHERE JOB='CLERK' GROUP BY DEPTNO

WAQ to find the number of Clerk's working for each department only if the count is greater than 1.

Sol: SELECT DEPTNO, COUNT(\*) FROM EMP WHERE JOB='CLERK' GROUP BY DEPTNO HAVING COUNT(\*)>1

ORDER BY *order\_list*[ASC | DESC]

The ORDER BY clause define the order in which the rows in the result set are sorted. *Order\_list* specifies the result set columns that make up the sort list. The ASC and DESC keywords are used to specify if the rows are sorted in an ascending or descending sequence.

SELECT \* FROM EMP ORDER BY SAL

SELECT \* FROM EMP ORDER BY SAL DESC

SELECT \* FROM EMP ORDER BY SAL, COMM

1. SUBQUERY: A subquery is a query that is nested inside a SELECT, INSERT, UPDATE or DELETE statement, or inside another subquery. A subquery can be used anywhere an expression is allowed. In this case first the inner query executes and basing upon the result generated by it the outer query executes to generate the final output.

WAQ to find details of employees earning the highest salary.

Sol: SELECT \* FROM EMP WHERE SAL= (SELECT MAX(SAL) FROM EMP)

WAQ to find the details of employees earning the second highest salary.

Sol: SELECT \* FROM EMP WHERE SAL=

( SELECT MAX(SAL) FROM EMP WHERE SAL <

(SELECT MAX(SAL) FROM EMP) )

WAQ to find the details of employees working in sales department.

Sol: SELECT \* FROM EMP WHERE DEPTNO=

(SELECT DEPTNO FROM DEPT WHERE DNAME=’SALES’)

WAQ to find the details of employees working in Mumbai.

Sol: SELECT \* FROM EMP WHERE DEPTNO=

(SELECT DEPTNO FROM DEPT WHERE LOC=’MUMBAI’)

WAQ to find the details of employees who are earning more than the highest salary of deptno 30

Sol: SELECT \* FROM EMP WHERE SAL>

(SELECT MAX(SAL) FROM EMP WHERE DEPTNO=30)

OR

SELECT \* FROM EMP WHERE SAL>

ALL(SELECT SAL FROM EMP WHERE DEPTNO=30)

-In this case we can use the ALL operator which will compare an expression with set of values, where the expression has to satisfy the condition with all the values.

WAQ to find the details of employees who are earning less than the lowest salary of deptno 20

Sol: SELECT \* FROM EMP WHERE SAL<

(SELECT MIN(SAL) FROM EMP WHERE DEPTNO=20)

OR

SELECT \* FROM EMP WHERE SAL>

ALL(SELECT SAL FROM EMP WHERE DEPTNO=20)

WAQ to find the details of the employees who are earning less than highest salary of deptno 10

Sol: SELECT \* FROM EMP WHERE SAL <

(SELECT MAX(SAL) FROM EMP WHERE DEPTNO=10)

OR

SELECT \* FROM EMP WHERE SAL>

ANY SELECT SAL FROM EMP WHERE DEPTNO=20)

1. In the people of ANY we can use SOME operator also.
2. In this case we can use the ANY/SOME operators which will compare an expression with set of values, where the expression has to satisfy the condition with at least a single value.

WAQ to find the details of employees who are earning the highest salary in each department.

SELECT \* FROM EMP WHERE SAL IN

(SELECT MIN(HIREDATE)FROM EMP GROUP BY DEPTNO)

WAQ to find the details of seniors in each department.

SELECT \* FROM EMP WHERE HIREDATE IN

(SELECT MIN(HIREDATE)FROM EMP GROUP BY DEPTNO)

Correlated Subqueries: Many queries can be evaluated by executing the subquery once and substituting the resulting value or values into the WHERE clause of the outer query. In queries that include a correlated subquery (also know as a repeating subquery), the subquery depends on the outer query for its values. This means that the subquery is executed repeatedly, once for each row that might be selected by the outer query.

WAQ to find the details of employees earning the highest salary.

Sol: SELECT \* FROM EMP E WHERE 0=

(SELECT COUNT(DISTINCT SAL) FROM EMP WHERE SAL>ESAL)

WAQ to find the details of employees earning the second highest salary.

Sol: SELECT \* FROM EMP E WHERE 1=

(SELECT COUNT(DISTINCT SAL) FROM EMP WHERE SAL>ESAL)

- In this if we want the n th highest salary we need to substitute n-1 value in the where condition of the outer query.

WAQ to find the details of department in which employees are working.

Subquery: SELECT \* FROM DEPT WHERE DEPTNO IN

(SELECT DISTINCT DEPTNO FROM EMP)

Correlated Subquery: SELECT \* FROM DEPT WHERE EXISTS

(SELECT DEPTNO FROM EMP WHERE

EMP.DEPTNO DEPT.DEPTNO)

1. EXISTS is an operator which is used to specifies a subquery to test for the existence of rows.

WAQ to find the details of department in which employees are not working.

Subquery: SELECT \* FROM DEPT WHERE DEPTNO NOT IN

(SELECT DISTINCT DEPTNO FROM EMP)

Correlated Subquery: SELECT \* FROM DEPT WHERE NOT EXISTS

(SELECT DEPTNO FROM EMP WHERE

EMP.DEPTNO DEPT.DEPTNO)

WAQ to find the details of employees who have subordinates under them.

Subquery: SELECT \* FROM EMP WHERE EMPNO IN

(SELECT DISTINCT MGR FROM EMP)

Correlated Subquery: SELECT \* FROM EMP E WHERE EXISTS

(SELECT \* FROM EMP M WHERE E.EMPNO=M.MGR)

WAQ to find the details of employees who doesn't have any subordinates under them.

Subquery: SELECT \* FROM EMP WHERE EMPNO NOT IN

(SELECT DISTINCT MGR FROM EMP)

Correlated Subquery: SELECT \* FROM EMP E WHERE NOT EXISTS

(SELECT \* FROM EMP M WHERE E.EMPNO=M.MGR)

1. JOINS: By using joins, you can retrieve data from two or more tables based on logical relationships between the tables. Joins indicate how database should use data from one table to select the rows in another table.

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

1. 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.
2. Specifying a logical operator (for example, = or <>,) to be used in comparing values from the columns.

Types of Joins:

1. Equi-Joins
2. Non Equi-Joins
3. Self Joins
4. Cartesian Joins
5. Outer Joins
6. Left Outer Join
7. Right Outer Join

Equi-Joins: It return the specified column from both the table, returns only the rows for which there is an equal value in the join column.

SELECT

E.EMPNO, E.ENAME, E.SAL, E.DEPTNO,

D.DEPTNO, D.DNAME, D.LOC

FROM EMP E, DEPT D

WHERE E.DEPTNO=D.DEPTNO

- In above statement is know, as old-style join statement, which will combine the tables basing on equality condition i.e the column in the Deptno column in the Emp table, has to have an exact math of Deptno in the Dept table, then these 2 rows combine and get retrieved. In the new-style we call this as Inner Join where we write the statement as following:

SELECT

E.EMPNO, ENAME, E.SAL, E.DEPTNO,

D.DEPTNO, D.DNAME, D.LOC

FROM EMP E INNER JOIN DEPT D

ON E.DEPTNO=D.DEPTNO

- In the same way if we want to combine multiple table in old-style we write following:

SELECT

E.EMPNO, ENAME, E.SAL, E.DEPTNO,

D.DEPTNO, D.DNAME, D.LOC

DD.DID, DD.COMMENTS

FROM EMP E, DEPT D, DEPTDETAILS DD

WHERE E.DEPTNO=D.DEPTNO AND D.DEPTNO=DD.DEPTNO

- The same statement in the new-style we write as following:

SELECT

E.EMPNO, ENAME, E.SAL, E.DEPTNO,

D.DEPTNO, D.DNAME, D.LOC

DD.DID, DD.COMMENTS

FROM EMP E INNER JOIN DEPT D

ON E.DEPTNO=D.DEPTNO

INNER JOIN DEPTDETAILS DD

ON D.DEPTNO=DD.DEPTNO\

Non Equi-Joins: You can also values in two column that are not equal. The same operators and predicates used for equi-joins can be used for not-equi joins.

SELECT

E.EMPNO, ENAME, E.SAL,

S.SALGRADE, S.LOSAL, S.HISAL

FROM EMP E, SALGRADE S

WHERE E.SAL BETWEEN S.LOSAL AND S.HISAL

- We can write the above statement using inner join in the new style as following:

SELECT

E.EMPNO, ENAME, E.SAL,

S.SALGRADE, S.LOSAL, S.HISAL

FROM EMP E INNER JOIN SALGRADE S

ON E.SAL BETWEEN S.LOSAL AND S.HISAL

Self Join: if a table has a reflexive relationship in the database, you can join it to itself automatically which is known as self join.

SELECT

DISTINCT E.EMPNO, E.ENAME, E.SAL, E.DEPTNO

FROM EMP E, EMP M

WHERE E.EMPNO=M.MGR

-We can write the above statement using inner join in the new style as following:

SELECT

DISTINCT E.EMPNO, E.ENAME, E.SAL, E.DEPTNO

FROM EMP E INNER JOIN EMP M

ON E.EMPNO=M.MGR

Cartesian join: a Cartesian join that does not have a WHERE clause produces the Cartesian product of the tables involved in the join. The size of a Cartesian product result set is the number of rows in the first table multiplied by the number of rows in the second table. This is also known as cross-join. However, if a WHERE clause is added, the cross-join behaves as an inner join.

SELECT

E.EMPNO, E.ENAME, E.SAL, E.DEPTNO,

D.DEPTNO, D.DNAME, D.LOC

FROM EMP E, DEPT D

-We can write the above statement in the new style as following:

SELECT

E.EMPNO, E.ENAME, E.SAL, E.DEPTNO,

D.DEPTNO, D.DNAME, D.LOC

FROM EMP E CROSS JOIN DEPT D

Outer Join: By default, when we join multiple tables using inner join what we get is the matching data from the tables, if we want to include data rows in the result set that do not have a match in the joined table, we can us outer join.

The old-style of outer join have been classified into 2 types as Left Outer join and Right outer join

We use Left Outer join to get the matching information plus unmatched information from left hand side table, in the same way we use Right Outer join to get the matching information plus unmatched information from right hand side table.

Left hand side table and right hand side table are referred in the order we write in the from clause, first table is LHS table and second table is RHS table.

SELECT

E.EMPNO, E.NAME, E.SAL, E.DEPTNO,

D.DEPTNO, D.DNAME, D.LOC

FROM EMP E, DEPT D

WHERE E.DEPTNO=\*D.DEPTNO

-In the above case we get the matching information plus unmatched information from Dept table.

SELECT

E.EMPNO, E.NAME, E.SAL, E.DEPTNO,

D.DEPTNO, D.DNAME, D.LOC

FROM EMP E, DEPT D

WHERE E.DEPTNO\*=D.DEPTNO

-In the above case we get the matching information plus unmatched information from Emp table.

-suppose we have unmatched information in both the sides we cannot retrieve it at the same time to over come this in the new-style of join statement they have introduced Full Outer join. So the new -style support use the following:

1. LEFT OUTER JOIN
2. RIGHT OUTER JOIN
3. FULL OUTER JOIN

-Use Left Outer join to get the unmatched information from left hand side table as following:

SELECT

E.EMPNO, E.NAME, E.SAL, E.DEPTNO,

D.DEPTNO, D.DNAME, D.LOC

FROM EMP E LEFT OUTER JOIN DEPT D

ON E.DEPTNO=D.DEPTNO

-Use Right Outer join to get the unmatched information from right hand side table as following:

SELECT

E.EMPNO, E.NAME, E.SAL, E.DEPTNO,

D.DEPTNO, D.DNAME, D.LOC

FROM EMP E RIGHT OUTER JOIN DEPT D

ON E.DEPTNO=D.DEPTNO

-Use Full Outer join to get the unmatched information from both the table as following:

SELECT

E.EMPNO, E.NAME, E.SAL, E.DEPTNO,

D.DEPTNO, D.DNAME, D.LOC

FROM EMP E FULL OUTER JOIN DEPT D

ON E.DEPTNO=D.DEPTNO

Finally concluding in the new-style we have only 3 types of join those are inner joins, Cross Joins and Outer Joins in the place of Equi-Joins, Non Equi-Joins, Self Joins Cartesian Joins and Outer Joins which are present in the old-sytle.

1. **Transactions**
2. A transaction is a single unit of work.
3. If a transaction is successful, all of the data modifications made during the transaction are committed and become a permanent part of the database.
4. If transaction encounters errors and must be canceled or rolled back, then all of the data modifications are crased.

SQL Server operators in the following transaction modes:

1. Autocommit transaction: Each individual statement is a transaction.
2. Explicit transaction: Each transaction is explicitly started with the BEGIN TRANSACTION statement and explicitly ended with a COMMIT or ROLLBACK statement.
3. Implicit transaction: A new transaction is implicitly started when the prior transaction completes but each transaction is explicitly completed with a COMMIT or ROLLBACK statement.
4. By Default SQL Server uses AutoCommit Transaction i.e after executing each statement it will auto' Commit it.
5. If we want to use the Explicit Transaction before executing the statement we need to start with a Begin Transaction statement and then decide whether it has to be commited or rollback, until the transaction ends the records gets locked.
6. If we want to use the Implicit Transaction we should use the following Statement:

SET IMPLICIT\_TRANSACTIONS ON | OFF

1. When ON, SET, IMPLICIT\_TRANSACTIONS sets the connection into implicit transaction mode. When OFF, it returns the connection to autocommit transaction mode.
2. To Manage the Transaction we have the TCL (Transaction Control Language) with 3 commands in it Commit, Rollback and save Transaction.

Commit: Mark the end of a successful implicit or explicit transaction. COMMIT TRANSACTION makes all data modifications performed since the start of the transaction a permanent part of the transaction a permanent part of the database, frees the resource held by the transaction.

Begin Transaction

DELETE FROM EMP WHERE EMPNO=1015

COMMIT

Rollback: Rolls back an explicit transaction to the beginning of the transaction, or to a savepoint inside the transaction.

Begin Transaction

DELETE FROM EMP WHERE EMPNO=1014

COMMIT

Save Transaction: A user can set a savepoint, or maker, within a transaction. The savepoint defines a location to which a transaction can return if part of the transaction is conditionally canceled. If a transaction is rolled back to a savepoint, it must proceed to completion with more Transact-SQL statement if needed and a COMMIT TRANSACTION statement, or it must be canceled altogether by rolling the transaction back to its beginning. To cancel an transaction, use the ROLLBACK TRANSACTION statement.

BEGIN TRANSACTION

UPDATE EMP SET SAL=5000 WHERE EMP=1001

SAVE TRANSACTION S1

UPDATE EMP SET SAL=5000 WHERE EMP=1002

SAVE TRANSACTION S2

UPDATE EMP SET SAL=5000 WHERE EMP=1003

ROLLBACK TRANSACTION S2 OR ROLLBACK TRANSACTION S1

COMMIT

- In the above case either the last statement gets rolled back or the last 2 statement gets rolled back and commit the rest.

**Creating a table from an existing table:** We can create a table from an existing table maintain a copy of the actual table before manipulating the data.

Syntax: SELECT< \* | <COLLIST> INTO <NEW TNAME> FROM <OLD TNAME> [CONDITIONS]

SELECT \* INTO NEW\_EMP FROM EMP

- In this case it creates a table NEW\_EMP by copying all the rows and column of the EMP table.

SELECT EMP, ENAME, SAL,DEPTNO INTO TEST\_EMP FROM EMP

-In this case creates a table TEST\_EMP with only the specified column from the EMP table.

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

- In this case it creates a table SALES\_EMP with only the information of sales department from the EMP table.

SELECT \* INTO DUMMY\_EMP FROM EMP WHERE 1=2

- In this case it creates the DUMMY\_EMP table with out any data in it.

**Copying data from one existing table to another table:** We can copy the data from one table into another table by using a combination of insert and select statement as following:

Syntax: INSERT INTO <TNAME> [ (COLLIST) ]

SELECT< \* | <COLLEST> FROM <TNAME> [CONDITIONS]

INSERT INTO DUMMY\_EMP SELECT \* FROM EMP

-In this case all the rows of EMP table is copied into DUMMY\_EMP table.

INSERT INTO DUMMY\_EMP (EMPNO, ENAME, SAL, DEPTNO)

SELECT EMPNO, ENAME, SAL, DEPTNO FROM EMP WHERE DEPTNO=30

- In this case it copies only the selected columns into the DUMMY\_EMP table from the EMP table.

1. **VIEWS**
2. A view can be thought of as either a virtual table or a stored query, like a real table, a view consists of a set of named columns and rows of data.
3. Unless a view is indexed, its data is not stored in the database as a distinct object.
4. What is stored in the database is a SELECT statement.
5. The result set of the SELECT statement forms the virtual table returned by the view.
6. A user can use this virtual table by referencing the view name in Transact-SQL statements the same way a table is referened.
7. The rows and column of data come from tables referenced in the query defining the view and are produced dynamically when the view is referenced.
8. A view acts as a filter on the underlying tables referenced in the view.
9. The query that defines the view can be from one or more tables or from other views in the current or other database.
10. There are no restrictions on querying through views and few restriction on modifying data through them.

**Syntax: CREATE VIEW <view\_name> [(column[,....*n*])]**

**[WITH <view\_attribute> [,.....*n*] ]**

**AS select\_statement**

**[WITH CHECK OPTION]**

Under the view\_attribute we have the following options:

**[ENCRYPTION]**

**[SCHEMABINDING]**

Types of Views:

1. Simple Views
2. Complex Views

Simple Views:

- These views as based upon a single table, which access the data from the single table.

- They contain a Sub Query which retrieves the data from one base table.

CREATE VIEW SIMPLE\_VIEW

AS SELECT EMPNO, ENAME, SAL, DEPTNO FROM EMP

- Once the view is created we can access the data from it as if it was a table as following:

SELECT \* FROM SIMPLE\_VIEW

SELECT EMPNO, ENAME, SAL, SAL\*12 AS [ANNUAL SAL], DEPTNO

FROM SIMPLE\_VIEW

SELECT DEPTNO, MAX(SAL) FROM EMP GROUP BY DEPTNO

-We can also perform DML operations on the Simple Views which will effect on the base table.

INSERT INTO SIMPLE\_VIEW VALUES(1234, 'BASITH', 4300, 20)

DELETE FROM SIMPLE\_VIEW WHERE DEPTNO=20

UPDATE EMP SET SAL=5600 WHERE EMP=1001

- All the column that are referenced in the view can be modified through the view.

- We cannot perform insert operations on the viewif he view does not contain all the not null columns of the base table.

Complex Views:

- If the View is based on multiple tables it is a complex view

- If it is based on a single table with any of the following:

1. Group By Cluase
2. Having Cluase
3. Group Functions
4. Distinct Function
5. Function Calls

CREATE VIEW EMP\_DEPT

AS

SELECT E.EMPNO, E.ENAME, E.SAL, D.DEPTNO, D.DNAME, D.LOC

FROM EMP E INNER JOIN DEPT D

ON E.DEPTNO=D.DEPTNO

CREATE VIEW EMP\_GRADE

AS

SELECT E.EMPNO, E.ENAME, E.SAL, E.GRADE, S.LOSAL, S.HISAL

FROM EMP E INNER JON SALGRADE S

ON E.SAL BETWEEN S.LOSAL AND S.HISAL

CREATE VIEW EMP\_MANAGERS

AS

SELECT E.ENAME+'WORK UNDER'+ M.ENAME

FROM EMP E INNER JOIN EMP M

ON E.MGR=M.EMPNO

CREATE VIEW EMP\_SALARIES

AS

SELECT EMPNO, ENAME, DEPTNO, SAL AS MONTHLY, SAL \*12 AS

ANNUAL FROM EMP

CREATE VIEW EMP\_DESIGNATIONS

AS

SELECT JOB FROM EMP WHERE DEPTNO=10

UNION

SELECT JOB FROM EMP WHERE DEPTNO=20

UNION

SELECT JOB FROM EMP WHERE DEPTNO=30

CREATE VIEW EMP\_MAX\_SAL

AS

SELECT DEPTNO, MAX(SAL) AS [HIGH SAL] FROM EMP GROUP BY DEPTNO

- If we want to perform manipulations on the Complex Views we have the following restrictions:

1. Any modifications, including UPDATE, INSERT, and DELETE statement, must reference columns from only one base table.
2. The column being modified in the view must reference the underlying data in the table column directly. They cannot be derived in any other way, such as through:

1. An aggregate function
2. A computation; the column cannot be computed from as expression using other column. Columns formed using set operators amount to a computation and are also not updatable.

1. The column being modified cannot be affected by GROUP BY, HAVING, or DISTINCT clauses.

- We can also classify views aas Updatable Views and Non Updatable Views:

- A view, which allows manipulations on it, is known as Updateable View.

- A view, which will not allow manipulations on it, is known as Non Updateable View.

With Check Option:

- Forces all data modification statements executed against the view to follow the criteria set within select statement.

- The Clause specifies that DML operations are not allowed on rows that the view cannot Select

- When a row is modified through a view, the WITH CHECK OPTION makes sure the data remains visible through the view after the modification is committed.

CREATE VIEW SALES\_EMP

AS

SELECT EMPNO, ENAME, SAL, DEPTNO FROM EMP WHERE DEPTNO=20

INSERT INTO SALES\_EMP VALUES (1050, 'ABDUL', 3500, 30)

- The above insert statement executes even if it does not satisfy the condition in the View, if this has to be restricted the view has to be created by using With Check Option clause.

ALTER VIEW SALES\_EMP

AS

SELECT EMPNO, ENAME, SAL, DEPTNO FROM EMP WHERE DEPTNO=20

WITH CHECK OPTION

- If we want to make any modifications to the existing view we can use the alter view statement.

**View Attributes:**

Encryption: After creating a view if we want to see the definition that can be found in the SYSCOMMENTS System Table.

SELECT TEXT FROM SYSCOMMENTS WHERE OBJECT\_NAME(ID)='SALES\_EMP'

if we want to hide the definition from other persons we can use the Encryption option while creating the view or alter the view after creation to add the clause:

ALTER VIEW SALES\_EMP

WITH ENCRYPTION

AS

SELECT EMPNO, ENAME, SAL, DEPTNO FROM EMP WHERE DEPTNO=20

WITH CHECK OPTION

Schemabinding:

- When SHEMABINDING is specified, the base table or tables cannot be modified in a way that would affect the view definition.

- The view definition itself must first be modified or dropped to remove

dependencies on the table that is to be modified.

- When you use SCHEMABINDING, the select statement must include the two-part names (schema.object) of tables that are referenced.

- We need to specify the column names individual in the select statement, cannot use "\*" in the select statement.

- All referenced object must be in the same database.

- View or tables that participate in a view created with the SCHEMABINDING clause cannot be dropped unless that view is dropped or changed so that it no longer has schema binding.

CREATE VIEW EMP BIND

WITH SCHEMABINDING

AS

SELECT EMNO, ENAME, JOB, MGR FROM DBO.EMP

- After the view is create EMP table cannot be dropped are the column referred in the views cannot be altered using the alter command.

1. **INDEXES**

- Like an index in a book, an index in a database lets you quickly find specific information in a table or indexed view.

- An index contain keys build from one or more columns in the table, or view, and pointers that map to storage location of the specified data.

- These keys are stored in structure (B-tree) that enables. SQL Server to find the row or rows associated with the key value quickly and efficiently.

- We can significantly improve the performance of database queries and applications by creating well-designed index to support your queries.

- Index can reduce the amount if data that must be read to return the query result set.

- Index can also enforce uniqueness on the rows I a table, ensuring the data integrity of the table data.

Types of indexes:

Clustered:

1. Clustered indexes sort and store the data rows in the table or view based on their key values.
2. These are the columns included in the index definition.
3. There can be only one clustered index per table, because the data rows themselves can be sorted in only order.
4. The only time the data rows in a table are sorted in stored order is when the table contains a clustered index.
5. When a table has a clustered index the table is called a clustered table.
6. A table can have only 1 clustered index on it, which will be created when a primary key constraint is used in a table.

Nonclustered:

1. Nonclustered indexes have a structure separate from the data rows.
2. A nonclustered index contain the nonclustered index key values and each key value entry has a pointer to the data rows that contains the key value.
3. The pointer from an index row in nonclustered index to a data row is called a row locator.
4. If a table has no clustered index, its data rows are stored in an unordered structure called a heap.
5. The structure of the row locator depends on whether the data pages are stored in a heap or a clustered table.
6. For a heap, a row locator is a pointer to the row.
7. For a clustered table, the row locator is the clustered index key.
8. A table can have 249 Nonclustered indexes on it, which will be create whenever a unique constraint is used in the table.

How indexes are used:

We-designed indexes can reduce disk I/O operations and consume fewer system resource therefore improving query performance. Index can be helpful for a variety of queries that contain SELECT, UPDATE or DELETE statements. When this query is executed, the query optimizer evaluates each available method for retrieving the data and select the most may efficient method. The method may be a table scan, or may be scanning one or more indexes if they exist.

When performing a table scan, the query optimizer reads all the rows in the table, and extracts the rows that meet the criteria of the query. A table scan generates many disk I/O operations and can be resource intensive. However, a table scan could be the most efficient method if, for example, the result set of the query is a high percentage of rows from the table.

When the query optimizer uses an index, it searches the index key column, find the storage location of the rows needed by the query and extract the matching rows from that location. Generally, searching the index is much faster then searching the table because unlike a table, an index frequently contain very few column per row and the rows are in sorted order.

The query optimizer typically select the most efficient method when executing queries. However, if no indexes are available, the query optimizer must use a table scan. Your task is to design and create indexes that are best suited to your environment so that the query environment so that the query optimizer has a selection of efficient indexes from which to select.

**The following tasks make up SQL Server recommended strategy for creating indexes:**

1. **Design the index.**

Index design is a critical task. Index design includes determining which columns to use, selecting the

index type(for example, clustered or nonclustered), selecting appropriate index options, and

determining filegroup or partition scheme placemant. For more information, see Designning Imdexes.

**2. Determine the best creation method. Indexes are created in the following ways:**

1. By defining a PRIMARY KEY or UNIQUE constraint on a column by using CREATE TABLE or ALTER TABLE
2. The SQL Server 2005 Database Engine automatically create a unique index to uniqueness requirement of a PRIMARY KEY or UNIQUE constraint. By default , a unique clustered index is created to enforce a PRIMARY KEY constraint, unless a clustered index already exists on the table, or you specify a unique nonclustered index. By default, a unique nonclustered index created to enforce a UNIQUE constraint unless a unique clustered index is explicitly specified and a clustered index is specified and clustered index on the table does not exist.
3. An index created as part of PRIMARY KEY or UNIQUE constraint is automatically given the same name as the constraint name.
4. By creating an index independent of a constraint by using the CREATE INDEX statement, you must specify the name of the index, table and column to which the index applies. Index option and index location, filegroup or partition scheme, can also be specified. By default, a nonclustered, nonunique index is created if the clustered or unique options are not specified.

**3. Create the index:**

1. Whether the index will be created on an empty table or one that contains data is an important factor to consider. Creating an index on an empty table has no performance implications at the index is created; performance will be affected when data is added to the table.
2. Creating indexes on large tables should be planned carefully so database performance is not hindered. The preferred way to create indexes on large tables is to start with the clustered index and then build any nonclustered index and then build any nonclustered indexes.

Syntax for creating a Index:

**CREATE [UNIQUE] [CLUSTERED | NONCLUSTERED] INDEX**

***index\_name* ON <table\_name | view\_name>(column[ASC | DESC] [*,...n*])**

CREATE UNIQUE CLUSTERED INDEX ENO\_IND ON EMP(EMPNO)

-In this case it create a unique clustered index on the empno column.

CREATE INDEX ENAME\_IND ON EMP(ENAME)

-In this case it create a non-unique non-clustered index on the ename column.

INDEXED VIEWS

1. An indexed view is a view that has been materialized, this means it has been computed an stored.
2. You index a view by creating a unique clustered index on it.
3. Indexed views dramatically improve the performance of some types of queries.
4. Indexed views work best for queries that aggregate many rows.
5. They are not well-suited for underlying data sets that are frequently updated

View are also know as virtual tables. The result set of a standard view is not stored permanently in the database. For a standard view, the overhead of dynamically building the result set for each query that references a view can he significant for views that involve complex processing of large number of rows, such as aggregating lots of data, or joining many rows. If such views are frequently referenced in queries, you can improve performance by creating a unique clustered index on the view, which is know as indexed view. When a unique clustered index is created on a view, the result set is stored in the database just like a table with a clustered index is stored.

Another benfit to creating an index on a view is existing queries can benefit from improved efficiency of retrieving data from the indexed view without having to be recoded.

As modifications are made to the in the base tables, the data modifications are reflected in the data stored in the Indexed View. The requirement that the clustered index of the view be unique improves the efficiency with SQL Server can find the rows in the index that are affected by any data modification.

If we want to create an Indexed View we need to do the following:

1. Create a view by using the with SchemaBinding Option.
2. Create a Unique Clustered Index on the view

CREATE VIEW IND\_VIEW

WITH SCHEMABINDING

AS

SELECT DEPTNO, SUM(ISNULL(SAL, 0)) AS [TOTAL SAL], COUNT\_BIG(\*)

AS [TOTAL RECORDS] FROM DBO.EMP GROUP BY DEPTNO

CREATE UNIQUE CLUSTERED INDEX DEPTNO\_IND ON IND\_VIEW(DEPTNO)

-Once the index is created on the view it will internally the store the information of the View physicially in a location, any manipulating performed on the base table reflects to the View also.

1. **TSQL PROGRAMMING**
2. TSQL (Transact SQL) Programming is an Procedural Language Extension to SQL which is known as PL/SQL in Oracle.
3. It extends SQL by adding programming structures and subroutines available in any high level language.
4. It has syntax and rules that determine how programming statements work together.
5. We can control the program flow by using conditional statements like IF and While loop
6. Runtime Error Handling is provided using the try catch mecchanism.
7. Reusability of the code is available by defining object such as Procedures and Functions.
8. SQL Commands can be embedded inside the programs.
9. Program Blocks can be of 2 types:

1. Anonymous Blocks

2. Sub-program Blocks

Anonymous Blocks: They are unnamed block of code for execution which can be written at a point where they are to be executed. They can be written on a Query window and execute.

Sub-program Blocks: These are nothing but named block of code for execution, where the program blocks are given a name for identification. These will be stored on the database which provides the reusablility of code.

Program Blocks like in any other language provides option for variable declaration, program logic using conditional statements and displaying the result to the user, in the same way we can define programs in SQL Server also.

Declaring Variables: While declaring variable it has to be preceded with @ symbol.

Syntax: DECLARE @<var> [AS] <data\_type> [,....n]

DECLARE @X INT

DECLARE @SAL AS MONEY

DECLARE @ENAME VARCHAR(50), @JOB VARCHAR(50)

Assigned Values to Variables: Value can be assigned by using a SET ststement.

Syntax: SET @ <var> = <value>

SET @ X=100

SET @ ENAME='SCOTT'

Printing Values: If we want to print the values we can use the Print statement.

Syntax: Print @<var>

Print @X

Print @Ename

Conditional Statements:

* 1. **If … Else If …. Else:** Imposes condition on the execution of a Transact-SQL statement. The Transact-SQL statement that follows an IF keyword and its condition are executed if the condition is satisfied: The Boolean expression return TRUE. The optional ELSE key word introduces another Transact-SQL statement that is when the IF condition is not satisfied: The Boolean expression return FALSE.

IF Boolean\_expression

[BEGIN]

<sql\_statement | statement\_block>

[END]

[ELSE IF Boolean\_expression

[BEGIN]

<sql\_statement | statement\_block>

ELSE

[BEGIN]

<sql\_statement | statement\_block> ]

[END]

-If there are multiple statements being enclosed between each block then we can put them under Begin and End Statements.

DECLARE @ WEEK INT

SET @ WEEK=DATEPART (DW, GETDATE())

IF @ WEEK=1

PRINT 'SUNDAY'

ELSE IF @ WEEK=2

PRINT 'MONDAY'

ELSE IF @ WEEK=3

PRINT 'TUESDAY'

ELSE IF @ WEEK=4

PRINT 'WEDNESDAY'

ELSE IF @ WEEK=5

PRINT 'THURSDAY'

ELSE IF @ WEEK=6

PRINT 'FRIDAY'

ELSE IF @ WEEK=7

PRINT 'SATURDAY'

* 1. **CASE FUNCTION:** The case function what we have discussed under the system Function can also be used here as following:

DECLARE @ WEEK INT

SET @ WEEK=DATEPART (DW, GETDATE())

SELECT CASE@ WEEK

WHEN 1 THEN'SUNDAY'

WHEN 2 THEN'MONDAY'

WHEN 3 THEN'TUESDAY'

WHEN 4 THEN 'WEDNESDAY'

WHEN 5 THEN'THURSDAY'

WHEN 6 THEN'FRIDAY'

WHEN 7 THEN'SATURDAY'

END

-This can be written in the second style of the CASE Statement also that has been discussed in the SQL as following:

DECLARE @ WEEK INT

SET @ WEEK=DATEPART (DW, GETDATE())

SELECT CASE

WHEN@ WEEK= 1 THEN'SUNDAY'

WHEN@ WEEK= 2 THEN'MONDAY'

WHEN@ WEEK= 3 THEN'TUESDAY'

WHEN@ WEEK= 4 THEN 'WEDNESDAY'

WHEN@ WEEK= 5 THEN'THURSDAY'

WHEN@ WEEK= 6 THEN'FRIDAY'

WHEN@ WEEK= 7 THEN'SATURDAY'

END

* 1. **While loop:** Sets a condition for the repeated execution of an SQL statement or statement block. The statements are executed repeatedly as long as the specified condition is true. The execution of statements in the WHILE loop can be controlled from inside the loop with the BREAK and CONTINUE keywords.

WHILE Boolean\_expression

[BEGIN]

<sql\_statement | statement\_block>

[ BREAK]

<sql\_statement | statement\_block>

[CONTINUE]

<sql\_statement | statement\_block>

[END]

-If there are multiple statements being enclosed then we can put them under Begin and end statements.

* 1. **BREAK:** Causes an exit from the innermost WHILE loop. Any statements that appear after the END keyword, marking the end of the loop, are executed.
  2. **CONTINUE:** Cause the WHILE loop to restart, ignoring any statements after the CONTINUE keyword.

Program 1:

DECLARE @X INT

SET @ X=0

WHILE @X<10

BEGIN

SET @X=@X+1

PRINT @X

END

Program 2:

DECLARE @X INT

SET @ X=0

WHILE @X<10

BEGIN

SET @X=@X+1

IF @X=6 BREAK

PRINT @X

END

-In this case the break statement bring the control out of the loop printing from 1 to 5.

Program 3 :

DECLARE @X INT

SET @ X=0

WHILE @X<10

BEGIN

SET @X=@X+1

IF @X=6 CONTINUE

PRINT @X

END

-In this case the continue statement will skip the print statement when the value of x is 6 so print from 1 to 5 and 7 to 10.

* 1. **Comment in TSQL:** Comment will be ignored will executing the program, they will increase the readability and aids understanding of the program.

1. Single Line Comment(- -)
2. Multi Line Comment (/\*......\*/)

Assinging values from column into variable: Till now we were assinging static valued to the variables using the SET statement, but we can also assign values from a column into the variables as following:

SELECT @<var>=<col\_name>[,.....n] FROM <table\_name>[COMDITIONS]

SELECT @ENAME=ENAME FROM EMP WHERE EMPNO=1001

-A simple TSQL program which takes the Empno and prints the Name and salary.

DECLARE @EMPNO INT, @ENAME VARCHAR(50), @SAL MONEY

SET @EMPNO=1005

SELECT @ENAME=ENAME, @SAL=SAL FROM EMP WHERE EMPNO = @EMPNO

PRINT @ENAME +'EARNS'+ CAST(@SAL AS VARCHAR)

- A Program which takes the Empno and increment the salary of the person on the following criteria:

If Job is President increment with 10 %

If Job is Manager increment with 8 %

If Job is Analyst increment with 6 %

If Job is any thing other increment with 5 %

DECLARE @EMPNO INT, @JOB VARCHAR(50)

SET @ EMPNO=1005

SELECT @JOB=JOB FROM EMP WHERE EMPNO = @EMPNO

IF @JOB='PRESIDENT'

UPDATE EMP SET SAL = SAL \* 0.1 WHERE

EMPNO = @EMPNO

ELSE IF @JOB='MANAGER'

UPDATE EMP SET SAL = SAL \* 0.08 WHERE

EMPNO = @EMPNO

ELSE IF @JOB='ANALYST'

UPDATE EMP SET SAL = SAL \* 0.06 WHERE

EMPNO = @EMPNO

ELSE

UPDATE EMP SET SAL = SAL \* 0.05 WHERE

EMPNO = @EMPNO

-In the above case which empno has been provided for the variable @EMPNO first it will check for the JOB of the employee and then it will increment the salary on the specified criteria.

-The problem in the above case is we can increment only one Employees Salary at a time but if we want to increase the Salary of Multiple employees at the same time it is not possible, as multiple rows cannot be effected within the program to over come this we use **Cursors.**

-operations in a relational database act on a complete set of rows. The set of rows returned by a SELECT statement consists of all the rows that satisfy the conditions in the WHERE clause of the statement. This complete set of rows returned by the statement is known as the result set. Applications, especially interactive online applications, cannot always work effectively with the entire result set as unit. These application need a mechanisms to work with one by one at a time. **Cursors** are an extension to result sets that provide that mechanism.

Cursor extent result processing by:

1. Allowing positioning at specific rows of the result set.
2. Retrieving one row or block of row from the current position in the result set.
3. Supporting data modifications to the rows at the current position in the result set.
4. Supporting different levels of visibility to changes made by other users to the database data that is presented in the result set.
5. Providing Transact-SQL statements in scripts, stored procedures, and triggers access to the data in a result set.
6. **Cursor Process:** Transact-SQL Cursors follow a general process that is used with all SQL Server cursors:
7. Associate a cursor with the result set of a Transact-SQL statement, and define characteristics of the cursor, such as whether the rows in the cursor can be updated.
8. Executed the Transact-SQL statement to populate the cursor.
9. Retrieve the rows in the cursor you want to see. The operation to retrieve one row or one block of rows from a cursor is called a fetch. Performing a series of fetches to retrieve rows in either a forward or backward direction is called scrolling.
10. Optionally, perform modification operations (update or delete) on the row at the current position in the cursor.
11. Close the Cursor

The Cursor Process has the following steps involved in it:

1. Declare a Cursor
2. Open a Cursor
3. Fetch data from the Cursor
4. close the Cursor
5. De-allocate the Cursor

**Declare a Cursor:** Define the attributes of a Transact-SQL Server cursor, such as its scrolling behavior and the query used to build the result set on which the cursor operates.

DECLARE *cursor\_name* CURSOR

[LOCAL|GLOBAL]

[FORWARD\_ONLY|SCROLL]

[STATIC|KEYSET|DYNAMIC|FAST\_FORWARD]

[READ\_ONLY|SCROOL\_LOCKS|OPTIMISTIC]

[TYPE\_WARNING]

FOR *select\_statement*

[FOR UPDATE [OF *column\_name*[,....*n*] ] ]

LOCAL: Specifies that the scope of the cursor is local to the program in which the cursor was created.

GLOBAL: Specifies that the scope of the cursor is global to the connection. The cursor name can be referenced in any program by the connection. The cursor is only implicitly deallocated at dis connect.

If neither GLOBAL nor LOCAL is specified, the default is taken as GLOBAL.

FORWARD\_ONLY: Specifies that the cursor can only be scrolled from the first to the last row. FETCH NEXT is the only supported fetch option. When neither FORWARD\_ONLY nor SCROLL is specified. FORWARD\_ONLY is the default, unless the keywords STATIC, KEYSET or DYNAMIC are specified. STATIC, KEYSET and DYNAMIC cursor default to SCROLL.

SCROLL: Specifies that the cursor can scroll from to the last row as well as last to first row also. It Supports 6 fetch methods like FETCH NEXT, FETCH PRIOR, FETCH FIRST, FETCH LAST, FETCH ASSOLUTE n and FETCH RELATIVE n.

STATIC: defines a cursor that makes a temporary copy of the data to be used by the cursor. All request to the cursor are answered from this temporary table in **tempdb**; therefore, modifications made to base table are not reflected in the data returned by fetches made to this cursor, and this cursor does not allow modifications.

KEYSET: Specifies that the membership and order of rows in the the cursor are fixed when the cursor is opened. The set of keys that uniquely identify the rows is build into a table **tempdb** known as the **keyset.** Changes to nonkey values in the base table, either made by the cursor or committed by other user, are not visible, are visible as we scroll around the cursor. Insert made by other user are not visible. If a row is deleted, an attempt to fetch the row returns an @@FETCH\_STATUS of -2. Updates of key values from outside the cursor resemble a delete of the old row followed by an insert of the new row.

DYNAMIC: Defines a cursor that reflects all data changes made to the rows in its result set as you scroll around the cursor. The data values, order and membership of the rows can change on each fetch.

FAST\_FORWARD: Specifies a FORWARD\_OMLY, READ\_ONLY cursor with performance optimizations enabled. FAST\_FORWARD cannot be specified if SCROLL or FOR\_UPDATE is also specified.

READ\_ONLY: Prevent update made through this cursor. This option overrides the default capability of a cursor to be updated.

SCROLL\_LOCKS: Specifies that positioned updates or deletes made through the cursor are guaranteed to succeed. Microsoft SQL Server locks the rows as they are read into the cursor to ensure their availability for later modification. SCROLL\_LOCKS cannot be specified if FAST\_FORWARD is also specified.

OPTMISTIC: Specifies that positioned update or delete made through the cursor do not succeed if the row has been updated in the table since it was read into the cursor. SQL Server does not lock rows as they are read into the cursor. It instead uses comparisons of **timestamp** column values , or a checksum value if the table has no  **timetamp** column, to determine whether the row was modified after it was read into the cursor. If the row was modified if FAST\_FORWARD is also specified.

TYPE\_WARNING: Specifies that a warning message is sent to the client if the cursor is implicitly converted from the requested type to another.

FOR UPDATE [OF *column name*[,...*n*]]: Define updatable column within the cursor. If OF  *column name*[,...*n*] is supplied, only the column listed allow modifications. If UPDATE is specified without a column list, all column can be updated, unless the READ\_ONLY concurrency option was specified.

Opening a Cursor: Opens a Transact-SQL Server cursor and populates the cursor by executing the Transact-SQL statement specified on the DECLARE CURSOR.

**Syntax: OPEN<cursor\_name>**

**Fetching data from the Cursor:** Retrieves a specific row from a Transact-SQL server cursor into specified variables.

Syntax:

**FETCH [NEXT | PRIOR | FIRST | LAST | ABSOLUTE n | RELATIVE n]**

**FROM <cursor\_name> INTO @ variable\_name [,....n]**

NEXT: Returns the result row immediately following the current row and increment the current row to the returned. If FETCH NEXT is the first fetch against a cursor, it return the first row in the result set. NEXT is the default cursor fetch option.

PRIOR: Returns the result row immediately preceding the current row, and decrements the current row to the row returned. If FETCH PRIOR is the first fetch against a cursor, no row is returned and the cursor is left positioned before the first row.

FIRST: Returns the first row in the cursor and makes it the current row.

LAST: Returns the last row in the cursor and makes it the current row.

ABSOLUTE n: If n is positive, it returns the specified nth row from the front of the cursor. If n is negative, it returns the specified nth row from the back of the cursor.

RELATIVE n: If n is positive, returns the row n rows beyond the current row. If n is negative , returns the row n rows prior to the current row.

If any of the used fetch statement is successful it returns the status of it which will be stored in a implict variable @@FETCH\_STATUS (this does not requires to be declared) which can be any of the following values:

0 - The FETCH statement was successful

-1 - The FETCH statement failed or the row was beyond the result set

-2 - The row fetched is missing

Closing a Cursor: Closes an open cursor by releasing the current result set and freeing any cursor locks held on the rows on which the cursor is positioned. CLOSE leaves the data structure available for reopening, but fetches and positioned updates are not allowed until the cursor is reopened. CLOSE must be issued on an open cursor; CLOSE is not allowed on cursor that have only been declared or are already closed.

**Syntax: Close <cursor\_name>**

**Deallocating a Cursor:** Removes a cursor reference. When the last cursor reference is deallocated, SQL Server releases the data structures comprising the cursor.

**Syntax: Deallocating <cursor\_name>**

**Using a Simple Cursor:**

DECLARE EMPCUR CURSOR FOR SELECT ENAME, SAL FROM EMP

DECLARE @ENAME VARCHAR(50), @SAL MONEY

OPEN EMPCUR

FETCH NEXT FROM EMPCUR INTO @ENAME, @SAL

WHILE @@FETCH\_STATUS=0

BEGIN

PRINT 'SALARY OF' +@ENAME + 'IS' + CAST(@SAL AS VARCHAR)

FETCH NEXT FROM EMPCUR INTO @ENAME, @SAL

END

CLOSE EMPCUR

DEALLOCATE EMPCUR

**Using a cursor to Update all the rows of the table:**

This program will explain you how we can update all the rows of the table basing on some conditions, similar to the program we have written before discussing cursor but there only a single row a single row has been modified

DECLARE EMPCUR CURSOR FOR SELECT EMPNO, JOB FROM EMP

DECLARE @EMPNO INT, @JOB VARCHAR(50)

OPEN EMPCUR

FETCH NEXT FROM EMPCUR INTO @EMPNO, @JOB

WHILE @@FETCH\_STATUS=0

BEGIN

IF @JOB='PRESIDENT'

UPDATE EMP SET SAL = SAL = SAL \* 0.1 WHERE [EMPNO = @EMPNO](about:blank)

ELSE IF @JOB='MANAGER'

UPDATE EMP SET SAL = SAL = SAL \* 0.08 WHERE [EMPNO = @EMPNO](about:blank)

ELSE IF @JOB='ANALYST'

UPDATE EMP SET SAL = SAL = SAL \* 0.06 WHERE [EMPNO = @EMPNO](about:blank)

ELSE

UPDATE EMP SET SAL = SAL = SAL \* 0.05 WHERE [EMPNO = @EMPNO](about:blank)

FETCH NEXT FROM EMPCUR INTO @ENAME, @SAL

END

CLOSE EMPCUR

DEALLOCATE EMPCUR

**Using a Global Cursor:**

**Program 1:**

DECLARE EMPCUR CURSOR GLOBAL

FOR SELECT ENAME, SAL, COMM FROM EMP

DECLARE @ENAME VARCHAR(50), @SAL MONEY, @COMM MONEY,

@TOTSAL MONEY

OPEN EMPCUR

FETCH NEXT FROM EMPCUR INTO @ENAME, @SAL, @COMM

WHILE @@FETCH\_STATUS=0

BEGIN

SET @TOTSAL=@SAL + ISNULL(@COMM,0)

PRINT @ENAME + 'EARNS' + CAST(@TOTSAL AS VARCHAR) +

'EVERY MONTH'

FETCH NEXT FROM EMPCUR INTO @ENAME, @SAL, @COMM

END

CLOSE EMPCUR

-In the above case because it was a global cursor we are not using any Deallocate cursor statement, now we use the same cursor in other programs with of declaring it as following:

**Program 2:**

DECLARE @ENAME VARCHAR(50), @SAL MONEY, @COMM MONEY,

@ANNSAL MONEY

OPEN EMPCUR

FETCH NEXT FROM EMPCUR INTO @ENAME, @SAL, @COMM

WHILE @@FETCH\_STATUS=0

BEGIN

SET @ANNSAL=(@SAL + ISNULL(@COMM,0)) \* 12

PRINT @ENAME + 'EARNS' + CAST(@ ANNSAL AS VARCHAR) +

'EVERY YEAR'

FETCH NEXT FROM EMPCUR INTO @ENAME, @SAL, @COMM

END

CLOSE EMPCUR

-We don't require to Deallocate the Cursor any where it gets deallocate when we close the connection.

**Using Static Cursor:**

DECLARE EMPCUR CURSOR STATIC

FOR SELECT SAL FROM EMP WHERE EMPNO=1005

DECLARE @SAL MONEY

OPEN EMPCUR

UPDATE EMP SET SAL=6000 WHERE EMPNO=1005

FETCH NEXT FROM EMPCUR INTO @SAL

PRINT @SAL

CLOSE EMPCUR

DEALLOCATE EMPCUR

-In this case after opening the cursor we have performed an update of Sal on the EMP table but still the cursor contains the old value but not the new value, so it prints the old Salary value only

**Using a Dynamic Cursor:**

DECLARE EMPCUR CURSOR DYNAMIC

FOR SELECT SAL FROM EMP WHERE EMPNO=1005

DECLARE @SAL MONEY

OPEN EMPCUR

UPDATE EMP SET SAL=4000 WHERE EMPNO=1005

FETCH NEXT FROM EMPCUR INTO @SAL

PRINT @SAL

CLOSE EMPCUR

DEALLOCATE EMPCUR

-In this case after opening the cursor we have performed an update of Sal on the EMP table but still the cursor contains the old value but not the new value, so it prints the old Salary value only

**Using Scroll Cursor:**

DECLARE EMPCUR CURSOR

SCROLL

FOR SELECT EMPNO FROM EMP

DECLARE @EMPNO INT

OPEN EMPCUR

FETCH NEXT FROM EMPCUR INTO @EMPNO

PRINT @EMPNO

FETCH LAST FROM EMPCUR INTO @EMPNO

PRINT @EMPNO

FETCH PRIORFROM EMPCUR INTO @EMPNO

PRINT @EMPNO

FETCH FIRST FROM EMPCUR INTO @EMPNO

PRINT @EMPNO

FETCH ABSOLUTE 12 FROM EMPCUR INTO @EMPNO

PRINT @EMPNO

FETCH ABSOLUTE 10 FROM EMPCUR INTO @EMPNO

PRINT @EMPNO

FETCH RELATIVE 3 FROM EMPCUR INTO @EMPNO

PRINT @EMPNO

FETCH RELATIVE 5 FROM EMPCUR INTO @EMPNO

PRINT @EMPNO

CLOSE EMPCUR

DEALLOCATE EMPCUR

-As we have declared the cursor as scroll all the Fetch methods can be used on it.

1. **SUB-PROGRAM**

A Sub-Program is a new block of code which can be reused. We have 2 types of Sub-programs in

SQL Server:

1. Procedures
2. Functions

Procedures :

1. A stored procedures is a saved collection of Transact-SQL statements or a reference to Microsoft .Net Framework common language runtime (CLR) method that can take return user-supplied parameters.
2. Procedures can be created for permanent use or for temporary use within a session, local temporary procedures, or for temporary use within all sessions, global temporary procedures.
3. Stored procedures can also be created to run automatically when an instance of SQL Server starts

Syntax:

CREATE | ALTER PROCEDURES <procedures\_name>

[ ( @parameter1 <data\_type>[=default] [OUT | OUTPUT]

( @parameter2 <data\_type>[=default] [OUT | OUTPUT]

…................................................

( @parametern <data\_type>[=default] [OUT | OUTPUT]

[ WITH <procedure\_options> ]

AS

BEGIN

<statement>

END

ALTER: Modifies a previously created procedure that was created by executing the CREATE PROCEDURE statement. ALTER PROCEDURE does not change permissions and does not affect any dependent stored procedure or triggers.

Procedure Options: the procedure provide to option that can be used while creating the procedure.

They are:

1. Encryption
2. Recompile

RECOMPILE: Indicates that the Database Engine does not cach a plan for this procedure and the procedure is compiled at run time. To instruct the Database Engine to discard plans for individual queries inside a stored procedure, use the RECOMPILE query hint when atypical or temporary values are used in only a subset of queries that belong to the stored procedure.

**Important:** Because the SQL Server 2005 query optimizer typically selects the best execution plan for a query, we recommend that hints, including <query\_hint>, be used only as a last resort by experienced developers and database administrators.

ENCRYPTION: Indicates that SQL Server 2005 will convert the original text of the CREATE PROCEDURE statement to an obfuscated format. The output of the obfuscation is not directly visible in any of the catalog views in SQL Server 2005. Users that have no access to system tables or database files cannot retrieve the obfuscated text.

-Procedure contain 2 parts in it: 1.Header 2.body

-Header part is the content above the AS keyword

-Body part is the content below the AS keyword

Passing Parameters to Procedures: As if we are passing parameters to functions in language, we can also parameters to Procedures. They are the means to pass a value to the procedure or returns from a procedure.

Parameter Modes: These will specify whether there parameter is passed into the procedure or returned out of the procedure. SQL Server supports to parameter Modes:

1. IN MODE (DEFAULT)
2. OUT OR OUTPUT MODE

IN MODE: Passes a value into the procedure for execution, this is best suitable for constants & expression. The value of it can be changed with in the program but cannot be returned. It is the default mode if nothing is specified.

OUT MODE: Passes a value back from the program after the execution of the procedure. The value of this option can be returned to the calling EXECUTE statement. Use OUTPUT parameters to return values to the caller of the procedure. **text, ntext** and **image** parameters cannot be used as OUTPUT parameters.

Syntax for executing the Procedure:

**EXEC | EXECUTE [ [@parameter=] <value> [OUTPUT] [DEFAULT] [,...n] ]**

A Simple Procedure:

CREATE PROCEDURE PROCI

AS

BEGIN

PART 'MY FIRST PROCEDURE'

END

-Executing the above procedure:

EXEC PROCI OR EXECUTE PROCI

A procedure Which accepts arguments:

ALTER PROCEDURE PROC2(@X INT, @Y INT)

AS

BEGIN

DECLARE @Z INT

SET @Z=@X+@Y

PRINT 'The SUM of the 2 Numbers is:' + CAST(@Z AS VARCHAR)

END

-Executing the above procedure:

1. EXEC PROC3 200 25
2. EXEC PROC3 @X=200, @Y=25
3. EXEC PROC3 X=DEFAULT, @Y=25
4. EXEC PROC3 @Y=25

-In the 3rd and 4th case it uses the default value of 100 to the variable X which has been given while creating the procedure.

A Procedure with OUTPUT parameter:

CREATE PROCEDURE PROC4(@X INT, @Y INT, @Z INT OUTPUT)

AS

BEGIN

SET @ @X @Y

END

-Executing the above procedure:

DECLARE @A INT

EXECUTE PROC4 500 250 @A OUTPUT

PRINT @A

-A Procedure for inserting values into the EMP Table:

CREATE PROCEDURE Insert Emp@Empno int, @Ename varchar(50), @Sal money, @Deptono int)

AS

Begin

INSERT INTO Emp (Empno, Ename, Sal, Deptno) VALUES (@Empno, @Ename, @Sal, @Deptno)

End

-Executing the above Procedure:

EXEC Insert\_Emp 1016, 'Abdul', 2500,10

-A Procedure for Inserting values into the Emp Table but with Validations:

- This is name as the previous one but with the following validations present in it:

- Empno cannot be NULL value.

- Empno cannot be duplicated.

- Salary cannot be less than 2500.

- Deptno should be present in the Dept Table.

CREATE PROCEDURE Insert\_Emp@Empno int, @Ename varchar(50), @Sal money, @Detno int)

AS

Begin

IF @Empno IS NULL

Begin

Print 'Empno cannot be NULL'

Return

End

IF Exists(SELECT \* FROM Emp WHERE Empno=@Empno)

Begin

Print 'Empno cannot be Duplicated'

Return

End

IF @Sal<2500 Begin

Print 'Salary cannot be less than 2500'

Return

End

IF Not Exists(SELECT \* FROM Dept WHERE Empno=@Empno)

Begin

Print 'Deptno not found in the Dept Table'

Return

End

INSERT INTO Emp (Empno, Ename, Sal, Deptno) VALUES (@Empno, @Ename, @Sal, @deptno)

End

-A Procedure which takes the Empno and return Provident Fund and Professional Tax at 12% and 5% respectively on the Salary.

CREATE PROCEDURE Deductions(@Empno int, @PF money OUTPUT, @PT money OUTPUT)

AS

Begin

Declare @Sal Money

SELECT @Sal=Sal FROM Emp WHERE Empno=@Empno

SET @PF=@Sal \* 0.12

SET @PF=@Sal \* 0.05

End

-Executing the above Procedure:

Declare@VPF money, @VPT money

EXEC Deductions 1005,@VPF OUTPUT,@VPT OUTPUT

Print @VPF

Print @VPT

-A Procedure which takes the Empno and prints the Net Salary of the Employee.

CREATE PROCEDURE Net\_Sal(@Empno int)

As

Begin

Declare @VSal money, @NSal money, @VPF money, @VPT money

EXEC Deductions @Empno, @VPF OUTPUT, @VPT OUTPUT

SELECT @Sal=Sal FROM Emp WHERE Empno=@Empno

SET @NSal=@VSal-@VPF-@VPT

Print 'Net Salary of the Employee is:' + Cast(@NSal as Varchar)

End

-Executing the above Procedure:

EXEC Net\_Sal 1005

-A Procedure which will insert values into the Dept table by generating a unique Deptno.

CREATE PROCEDURE Insert\_Dept(@Dname varchar(50), @Loc varchar(50)

As

Begin

Declare @Deptno int

Select @Deptno= ISNULL(MAX(Deptno), 0) + 10 FROM Dept

INSERT INTO Dept Values (@Deptno, @Dname, @Loc)

End

-Executing the above Procedure:

EXEC Insert\_Dept 'Research' , 'Hyderabad'

-A Procedure which is used from transferring amount from one account to the other

within the Bank table:

CREATE PROCEDURE Funds\_Transfer(@SrcID int, @DestID int, @Amt money)

As

Begin

UPDATE BANK SET Bal = Bal - @Amt WHERE CUSTID=@ScrID

UPDATE BANK SET Bal = Bal - @Amt WHERE CUSTID=@DestID

End

-Executing the above Procedure:

EXEC Funds\_Transfer 101, 102, 500

- In the above case if the SrcID or DestID are not present in the table then it will deduct the amount from the other or add the amount from the other to avoid this we neeed to use transaction management.

- To manage the tranaction first we need to identify which statement is executed and which failed for this we use the function @@**ROWCOUNT.**

**-** @@**ROWCOUNT** returns the number of rows affected by the last statement.

-Managing Transactions in the Procedure:

CREATE PROCEDURE Funds\_Transfer(@ScrID int, @DestID int, @Amt money)

As

Begin

Declare @Count int, @count2 int

Begin Transaction

UPDATE BANK SET Bal = Bal - @Amt WHERE CUSTID=@ScrID

Set @count1=@@ROWCOUNT

UPDATE BANK SET Bal = Bal - @Amt WHERE CUSTID=@DestID

Set @count2=@@ROWCOUNT]

IF @COUNT1= @COUNT2

Begin

COMMIT

PRINT 'TRANSACTION COMMITED'

End

ELSE

Begin

ROLLBACK

PRINT 'TRANSACTION ROLLBACK'

End

End

**Handling Error in Procedures:**

* + In SQL Server when a error occurs, the statement that caused the error is terminated, but the execution of the stored procedure or batch continues.
  + When stored procedures and batches are executed within the scope of a TRY block, batch abort errors can be handled by TRY....CATCH construt.
  + Errors in Transact-SQL code can be processed using a TRY....CATCH construct similar to the exception-handling features of the language.
  + A TRY....CATCH construct consists of two parts: a TRY block and a CATCH block.
  + When an error condition is detected in a Transact-SQL statement contained in a TRY block, control is passed to a CATCH block where it can be processed.
  + After the CATCH block handles the exception, control is then transferred to the first Transact-SQL statement that follows the END CATCH statement.
  + If the END CATCH statement is the last statement in a stored procedure or trigger, control is returned to the code that invoked the stored procedure or trigger.
  + Transact-SQL statement in the TRY block following the statement that generates an error will not get executed.
  + If there are no errors inside the TRY block, control passes to the statement immediately after the associated END CATCH statement.
  + If the END CATCH statement is the last statement in a stored procedure or trigger, control is passed to the statement that invoked the stored procedure or trigger.
  + A TRY block starts with the BEGIN TRY statement and ends with the END TRY statement.
  + One or more Transact-SQL statements can be specified between the BEGIN TRY and END TRY statements.
  + A CATCH block must follow a TRY block immediately.
  + A CATCH block starts with the BEGIN CATCH statement and ends with the END CATCH statement.
  + In Transact-SQL, each TRY block is associated with only one CATCH block.

- A Procedure which can cause Error:

* CREATR PROCEDURE Div(@X int, @Y int)
* As
* Begin
* Declare @Z int
* SET @Z=0
* SET @Z=@X/@Y
* PRINT 'The Output is:' + Cast(@Z as varchar)
* END
* - Executing the above procedure:
* EXEC DIVX 100, 20
* EXEC DIVX 100, 0
* - The first execution will print the result of 5 but the second time execution will raise an error because we cannot divide a number by zero, in this case still it will try to print the result as 0, because even if the error is encountered it will not stop the execution of the program, if we want to stop the execution of the program when an error raises the code has to be written in the following way:
* CREATE PROCEDURE Div(@X int, @Y int)
* As
* Begin
* Begin Try
* Declare @Z INT
* SET @Z=0
* SET @Z=@X/@Y
* PRINT 'The Output is:' + Cast(@Z as varchar)
* END Try
* Begin Catch
* Print Error\_Message()
* End Catch
* End
* - Executing the above procedure:
* EXEC DIVX 100, 20
* EXEC DIVX 100, 0
* - Every error has 4 properties to it, they are:
  + Msg id
  + Msg str
  + Severity
  + statements
* For Example try the following statement:
* Print 100/0
* - This will display the following error message:
* Msg 8134, Level 16, State 1,
* Divide by zero error encountered.
* - In this the Msg id is 8134, Msg str is “Divide by zero error encountered”, Severity Level is 16 and State is 1.
* Msg id: ID of the message, which is unique across server, Message Ids less than 50000 are system messages.
* Msg str: Error message that has be displayed when the error raises.
* Severity Level: Severity level that is associated with the error. Severity levels can range between 0 and 25, Severity level is encountered, the client connection is terminated after receiving the message, and the error is logged in the error and application logs.
* State: Is an arbitrary integer from 1 through 127. If the same user-defined error is raised at multiple locations, using a unique state number for each location can help find which section of code is raising the errors.
* Raising Errors Manually: We can also raise errors manually at some required situations. It is used to return message back to applications using the same format as a system error or warning message generated by the SQL Server Database Engine. For raising an error manually we use the Raiserror Statement.
* RAISERROR can return either:
* A user-defined error message that has been created using the **sp\_addmessage** system stored procedure.
* A message string specified in the RAISERROR statement.
* RAISERROR can also:
* Assign a specific error number, severity, and state.
* Request that the error be logged in the Database Engine error log and the Microsoft Windows application log.
* Substitute argument values into the message text, much like the C language printf function.
* Syntax: RAISERROR (msg\_id | msg\_str | @local\_variable, severity, state [,argument[,...n]])
* [ WITH option [,....n] ]
* The error is returned to the caller if RAISERROR is run:
* Outside the scope of any TRY block.
* With a severity of 10 or lower in a TRY block.
* With a severity of 10 or higher that terminates the database connection.
* CATCH blocks can use RAISERROR to rethrow the error that invoked the CATCH block by using system function such as ERROR\_NUMBER and ERROR\_MESSAGE to retrieve the original error information. @@ERROR is set to 0 by default for message with a severity from 1 through 10.
* A procedure to divide 2 numbers and will raise an error when the divisor is 1.
* CREATE PROCEDURE Divx(@X int, @Y int)
* As
* Begin
* Declare @Z INT
* SET @Z 0
* IF @Y=1
* RAISERROR ('CANNOT DIVIDE BY 1', 15, 1)
* SET @Z=@X/@Y
* PRINT ' The Output is:' + Cast(@Z as varchar)
* End
* - Executing the above procedure:
* EXEC DIVX 100, 20
* EXEC DIVX 100, 0
* - In the above case the RAISERROR statement raises the error but still next statement get executed. So if we want to stop the execution on the same line the code has to be enclosed with in the TRY and Catch blocks.
* CREATE PROCEDURE Divx(@X int, @Y int)
* As
* Begin
* Begin TRY
* Declare @Z INT
* SET @Z 0
* IF @Y=1
* RAISERROR ('CANNOT DIVIDE BY 1', 15, 1)
* SET @Z=@X/@Y
* PRINT ' The Output is:' + Cast(@Z as varchar)
* End Try
* Begin Catch
* PRINT ERROR\_MESSAGE()
* End
* - Executing the above procedure:
* EXEC DIVX 100, 1
* **Pre-defined Error**: All the predefined error list of sql server can be found in the SYS.Message Catalog View. Query on the database with the statement where we can view the list of predefined errors:
  + **SELECT \* FROM SYS.MESSAGE**
* **SP\_AddMessage:**  Stored a new user-defined error message in an instance of the SQL Server Database Engine. Message stored using sp\_addmessage can be viewed using the **sys.message** catalog view.
* Syntax: **sp\_addmessage [ @msgnum = ] msg\_id,**
* **[ @severity = ] severity,**
* **[ @msgtext = ] 'msg'**
* **[ , [ @lang = ] 'language' ]**
* **[ , [ @with\_log = ] 'with\_log'**
* **[ , [ @replace = ] 'replace'**
* **EXEC sp\_addmessage 50001, 16, 'Cannot Divide the Number by One'**
* - The statement will insert a record into the SYS.Message System catalog after it was inserted we can use the raiseerror statement as following in our previous procedure:
* **Raiserror(50001, 16, 1)**
* **-** So when the error is raised the corresponding error message is picked out from the catalog View and displayed to the user.
* - Add Procedure, which will delete a record from the dept table for the given deptno and will raise an error if the deptno has any child records in the emp table:
* CREATE PROCEDURE Delete\_Dept(@Deptno int)
* As
* Begin
* IF EXISTS (SELECT \* FROM Emp WHERE [Deptno=@Deptno](about:blank))
* Raiserror('Child Records Found' 15. 1)
* ELSE
* DELETE FROM Dept WHERE [Deptno=@Deptno](about:blank)
* End
* - After creating a Procedure at any time if we want to view the content of it write the following statement:
* **SP\_HELPTEXT <procedure\_name>**
* **SP\_HELPTEXT Delete \_Dept**
* **Creating a Procedure using With Encryption Option:**
* CREATE PROCEDURE Delete\_Dept(@Deptno int)
* WITH ENCRYPTION
* AS
* IF EXISTS(SELECT \* FROM Emp WHERE [Deptno=@Deptno](about:blank)
* Raiserror('Child Records Found', 15, 1)
* ELSE
* DELETE FROM Dept WHERE [Deptno=@Deptno](about:blank)
* End
* - If the Procedure is created by using the With Encryption Option even if we use the SP\_HELPTEXT also we cannot view the content of it.
* Function are 3 types:
* Scalar Functions
* Inline Table- valued Functionality
* Multistatement Table-valued Function
* Scalar Function: Functions are scalar-valued if the RETURNS clause specifies one of the scalar data types.
* Syntax: CREATE FUNCTION <function\_name>
* ([@parameter\_name[AS] data\_type [=default][,...n] ])
* RETURNS data\_type
* [WITH <function\_option>[,...n] ]
* [AS]
* BEGIN
* function\_body
* RETURN scalar-expression
* END
* function\_option can be any of these two:
* Encryption
* Schemabinding
* ENCRYPTION: Indicates that the Database Engine encrypts the catalog view column that contain the text of the CREATE FUNCTION statement.
* SCHEMABINDING: Specifies that the function is bound to the database object that it reference. The binding of the function to the object it references is moved only when one of the following actions occurs:
* The function is dropped.
* The function is modified by using the ALTER statement with the SCHEMABINDING option not specified.
* - A Function that takes the Empno and Returns the total salary of the employee.
* CREATE FUNCTION GET\_TSAL (@EMPNO INT)
* RETURNS MONEY
* AS
* BEGIN
* DECLARE @TSAL MONEY
* SELECT @TSAL=SAL + ISNULL (COMM, 0) FROM EMP WHERE EMPNO=@EMPNO
* RETURN @TSAL
* END
* Syntax for Calling a Scalar Function:
* SELECT<owner><function\_name>(<list of values>)
* Calling the above Function:
* SELECT DBO.GET\_TSAL(1005)
* Inline Table-valued Function: These function can return a table as an output. Inline table- valued function , the TABLE return value is defined through a single SELECT statement. Inline function do not have associated return variables.
* Syntax: CREATE FUNCTION <function\_name>
* ([@parameter\_name[AS] data\_type [=default][,...n] ])
* RETURNS TABLE
* [WITH <function\_option>[,...n] ]
* [AS]
* BEGIN
* function\_body
* RETURN [( ] select\_stmt [ )]
* END
* Multistatement Table-valued Functions: These function are same as Inline Table-valued but the body of this function can contain multiple statement in it and the structure of the table can be defined by us.
* Syntax: CREATE FUNCTION <function\_name>
* ([@parameter\_name[AS] data\_type [=default][,...n] ])
* RETURNS @return\_variable TABLE <table\_type\_definition>
* [WITH <function\_option>[,...n] ]
* [AS]
* BEGIN
* function\_body
* RETURN
* END

1. **TRIGGER**
2. Microsoft SQL Server 2005 provides two primary mechanism for enforcing business rules and data integrity: constraints and triggers.
3. A trigger is special type of stored procedure that automatically takes effect when a language event executes.
4. SQL Server includes two general types of trigger: DML trigger and DDL triggers.
5. DDL trigger are new to SQL Server 2005. These triggers are invoked when a data definition language (DDL) event takes place in the server or database.
6. DML triggersare invoked when a data manipulation language (DML) event takes place in the database. DML event inside INSERT, UPDATE OR DELETE statements that modifiy data in a specified table or view.
7. A DML trigger can query other table and can include complex Transact-SQL statement. The trigger and the statement that fires it are treated as a single transaction., which cab be rolled back from within the trigger. If a severe error is detected ( For example, insufficient disk space), the entire transaction automatically rolls back.

DML Triggers are useful in these ways:

1. They can cascade changes through related tables in the database; however, these changes can be executed more efficiently using cascading referential integrity constraints.
2. They can guard agaist malicious or incorrect INSERT, UPDATE and DELETE operations and enforce other restictions that are more complex than those defined with CHECK constraints.
3. Unlike CHECK constraints, DML triggers can reference columns in other tables. For example, a trigger can use a SELECT from another table to compare to the insert or updated data and to perform additional actions, such as modify the data or display a user-defined error message.
4. They can evaluate the state of a table before and after a data modification and take actions based on that difference.
5. Multiple DML triggers of the same type (INSERT, UPDATE OR DELETE) on a table allow multiple, different actions to take place in response to the same modification statement.

Types of DML Triggers:

**AFTER Triggers**: AFTER triggers are executed after the action of the INSERT, UPDATE OR DELETE statement is performed. Specifying AFTER is the same as specifying FOR, which is the only option available in earlier versions of Microsoft SQL Server. AFTER triggers can be specified only on tables.

**INSTEAD OF Triggers:** INSTEAD OF Triggers are executed in place of the usual triggering action. INSTEAD OF Triggers can also be defined on views with one or more base tables, where they can extent the types of updates a view can support.

A trigger is special kind of stored procedure that automatically executed when an event occurs in the database server. DML triggers execute when a user tries to modify data through a data manipulation language (DML) event. DML events are INSERT, UPDATE OR DELETE statements on a table or view. DDL triggers execute in response to a variety of data definition language (DDL) event. These are primarily CREATE, ALTER and DROP statements. DML and DDL triggers can be created in the SQL Server 2005 Database Engine directly from Transact-SQL. SQL Server allows for creating multiple triggers for specific statement.

Syntax: CREATE TRIGGER trigger\_name

ON table | view

[ WITH ENCRYPTION ]

FOR | AFTER | INSTEAD OF

[ INSERT ] [ , ] [ UPDATE ] [ , ] [ DELETE ]

AS

BEGIN

sql\_statements

END

-A Trigger that will restrict the operations to be performed before 9 A.M and after 5 P.M

CREATE TRIGGER EMP\_TRG

ON EMP AFTER INSERT, UPDATE, DELETE

AS

BEGIN

DECLARE @DT INT

SET @DT=DATENAME(HH, GETDATE( ))

IF @DT NOT BETWEEN 9 AND 16

BEGIN

ROLLBACK

RAISERROR('CANNOT PERFORM DML OPERATIONS NOW' 15, 1)

END

END

-After the trigger is created try to perform any DML Operations on the EMP table before 9 A.M and after 5 P.M the trigger will fire and restrict the operations.

-A trigger that will convert the DName and loc into uppercase when the user insert in lower case.

CREATE TRIGGER DEPT\_CONVERT\_TRG

ON DEPT AFTER INSERT

AS

BEGIN

DECLARE @DEPTNO INT

DECLARE @DNAME VARCHAR(50)

DECLARE @LOC VARCHAR(50)

SELECT @DEPTNO=DEPTNO, DNAME=@DNAME , @LOC=LOC FROM INSERTED

UPDATE DEPT SET DNAME=UPPER(@DNAME), LOC=UPPER(@LOC)

WHERE DEPTNO=@DEPTNO

END

-To test the trigger execute the folllowing statement which will convert the date into upper case in the table:

INSERT INTO DEPT VALUES (50, 'research', 'hyderabad')

-The above trigger can be written in the following way also:

CREATE TRIGGER DEPT\_CONVERT\_TRG

ON DEPT INSTEAD OF INSERT

AS

BEGIN

INSERT INTO DEPT

SELECT DEPTNO, UPPER(LOC) FROM INSERTED

END

-A trigger which will gensrate a unique Deptno when the user insert a record into the dept table only by specifying DName and Loc when a primary key constraint is present on the Deptno column.

CREATE TRIGGGER DEPT\_CONVERT\_TRG

ON DEPT INSTEAD OF INSERT

AS

BEGIN

INSERT INTO DEPT

SELECT DEPTNO, UPPER(LOC) FROM INSERTED

IF @DEPTNO=ISNULL(MAX(DEPTNO), 0) + FROM DEPT

INSERT INTO DEPT SELECT @DEPTNO, DNAME, LOC FROM INSERTED

END

-To test the following Trigger execute the following statement:

INSERT INTO DEPT(DNAME, LOC) VALUES('RESERCH, HYDERABAD')

-A program which will restrict the delete operation if the Job of the person is Manager.

ALTER TRIGGER EMP\_TRG

ON EMP AFTER DELETE

AS

BEGIN

DECLARE @JOB VARCHAR(50)

SELECT @JOB FROM DELETED

IF @JOB=;MANAGER'

BEGIN

ROLLBACK

RAISERROR ('CANNOT DELETE MANAGER FROM THE TABLE', 15, 1)

END

END

-To test the following Trigger execute the following statement:

DELETE FROM EMP WHERE EMPNO=1002

A Trigger which will restrict to update the salary of the Employee if the New Salary is less than the Old Salary.

CREATE TRIGGER EMP\_UPDATE\_TRG

ON EMP AFTER UPDATE

AS

BEGIN

DECLARE @OLDSAL MONEY

DECLARE @NEWSAL MONEY

SELECT @OLDSAL=SAL FROM DELETED

SELECT @NEWSAL=SAL FROM INSERTED

IF @OLDSAL > @NEWSAL

BEGIN

ROLLBACK

RAISERROR('NEW SAL CANNOT BE LESS THAN OLD SAL', 15, 1)

END

END

NESTED TRIGGERS: Triggers can be nested to a maximum of 32 levels. If a trigger changes a table on which there is another trigger, the second trigger is activated and can then call a third trigger, and so on. To disable nested triggers, set the nested triggers option of sp\_configure to 0(off). The default configuration allows for nested triggers.

-A Trigger which will fire when a record is inserted into the Emp table which verifies whether the given Deptno is present in the Dept table or not if not it will insert a record into it.

CREATE TRIGGER EMP\_NESTED\_TRG

ON DEPT AFTER INSERT

AS

BEGIN

DECLARE @DEPTNO INT

SELECT @DEPTNO=DEPTNO FROM INSERTED

IF NOT EXISTS(SELECT \* FROM DEPT WHERE DEPTNO=@DEPTNO)

INSERT INTO VALUES(@DEPTNO, NULL, NULL)

END

-A Trigger which will fire when a record is inserted into the Dept table which verifies whether the given Deptno is present in the DeptDetails table or not if not it will insert a record into it.

CREATE TRIGGER DEPT\_NESTED\_TRG

ON DEPT AFTER INSERT

AS

BEGIN

DECLARE @DEPTNO INT

DECLARE @DID INT

SELECT @DEPTNO=DEPTNO FROM INSERTED

IF NOT EXISTS(SELECT \* FROM DEPTDETAILS WHERE DEPTNO=@DEPTNO)

BEGIN

SELECT @DID=MAX(DID)+1 FROM DEPTDETAILS

INSERT INTO DEPTDETAILS VALUES (@DID, @DEPTNO, NULL)

END

END

-After the 2 Trigger if we try to insert a record into Emp table with a Deptno not present in the Dept table it will insert a record into the Dept table which will internally check whether the Deptno is present in the Deptdetails table or not and inserts a records into it if not present.

**Instead of Triggers on Complex Views which are not updatable:**

CREATE VIEW EMP\_DEPT

AS

SELECT E.EMPNO, E.ENAME, E.SAL, D.DEPTNO, D.DNAME, D.LOC

FROM EMP INNER JOIN DEPT D ON E.DEPTNO=D.DEPTNO

-After creating the view try to execute the following insert which will fail because complex views are by default non updatable:

INSERT INTO EMP\_DEPT VALUES(1100, 'ABU', 4500, 50, 'IT', 'BANGLORE')

-If the above statement has to execute we need to define a Instead of trigger on the view so that the View become updatable.

CREATE TRIGGER VIEW\_INSERT\_TRG

ON EMP\_DEPT INSTEAD OF INSERT

AS

BEGIN

INSERT INTO EMP(EMPNO, ENAME, SAL, DEPTNO)

SELECT EMPNO, ENAME, SAL, DEPTNO FROM INSERTED

INSERT INTO DEPT(DEPTNO, DNAME, LOC)

SELECT DEPTNO, DNAME, LOC FROM INSERTED

END

-In the same way a trigger which will allow delete operations to be performed on the View:

CREATE TRIGGER VIEW\_DELETE\_TRG

ON EMP\_DEPT INSTEAD OF DELETE

AS

BEGIN

DECLARE @DEPTNO INT

DECLARE @COUNT INT

SELECT @DEPTNO FROM DELETED

SELECT @COUNT(\*)FROM EMP WHERE DEPTNO=@DEPTNO

DELETE FROM EMP WHERE DEPTNO=@DEPTNO

IF @COUNT=1

DELETE FROM DEPT WHERE DEPTNO=@DEPTNO

END

**SAMPLE TABLES**

These are the samples table that are used in the examples in the above:

Dept Table with out Constraint:

Create Table Dept(Deptno int, DName varchar(50), Loc varchar(50))

Dept Table with Constraint:

Create Table Dept(Deptno int, Constraint Deptno\_Pk Primary Key, DName varchar(50),

loc varchar(50))

Emp Table with out Constraint:

Create table Emp(Empno int, ename varchar(50), Job varchar(50), Mgr int, HireDate datetime, Sal Money, Comm Money, Deptno int)

Emp Table with Constraint:

Create table Emp(Empno int Constraint Empno\_Pk Primary Key, Ename varchar(100), Job varchar(100), Mgr int constraint Mgr\_Ref References Emp(Empno), HireDate datetime, Sal Money constraint Sal\_Check Check (sal between 1500 and 7500), Comm Money, Deptno int, Constraint Deptno\_Ref Foreign Key(Deptno) References Dept(Deptno))

DeptDetails Table with out Constraint:

Create table DeptDetails (Did int, deptno int, comments varchar(8000))

DeptDetails Table with Constraint:

Create table DeptDetails (Did int primary key, deptno int references dept(deptno),

comments varchar(8000))

Salgrade Table:

Create Table SalGrade(Grade int Constraint Grade\_Pk Primary Key, LoSal Money, Hisal Money)

Data in Dept Table:

Insert into Dept values(10, 'Marketing', 'Mumbai')

Insert into Dept values(20, 'Sales', 'Chennai')

Insert into Dept values(30, 'Finance', 'Delhi')

Insert into Dept values(40, 'Production', 'Kolkota')

Data in Emp Table:

Insert into Emp (Empno, Ename, job, Mgr, HireDate, Sal, Comm, Deptno)

Values(1001, 'Scoot', 'President', NULL, '01/07/2013, 5000,NULL, 10)

Insert into Emp (Empno, Ename, job, Mgr, HireDate, Sal, Comm, Deptno)

Values(1002, 'Clark', 'Manager', 1001, '01/07/2013, 4000,NULL, 10))

Insert into Emp (Empno, Ename, job, Mgr, HireDate, Sal, Comm, Deptno)

Values(1003, 'Smith', 'Manager', 1001, '01/07/2013, 3500, 500, 20))

Insert into Emp (Empno, Ename, job, Mgr, HireDate, Sal, Comm, Deptno)

Values(1004, 'Vijay', 'Manager', 1001, '01/07/2013, 4000, ,NULL, 30))

Insert into Emp (Empno, Ename, job, Mgr, HireDate, Sal, Comm, Deptno)

Values(1005, 'Ajay', 'Salesman', 1003, '01/07/2013, 3000, 300, 20))

Insert into Emp (Empno, Ename, job, Mgr, HireDate, Sal, Comm, Deptno)

Values(1006, 'Sathis', 'Salesman', 1003, '01/07/2013, 4000, 600, 20))

Insert into Emp (Empno, Ename, job, Mgr, HireDate, Sal, Comm, Deptno)

Values(1007, 'Venkat', 'Salesman' 1003, '01/07/2013, 3300, 0, 20))

Insert into Emp (Empno, Ename, job, Mgr, HireDate, Sal, Comm, Deptno)

Values(1008, 'Vinod', 'Clerk', 1003, '01/07/2013, 2400,NULL, 20))

Insert into Emp (Empno, Ename, job, Mgr, HireDate, Sal, Comm, Deptno)

Values(1009, 'Suneel', 'Clerk', 1004, '01/07/2013, 2000,NULL, 30))

Insert into Emp (Empno, Ename, job, Mgr, HireDate, Sal, Comm, Deptno)

Values(1010, 'Srinivas', 'Analyst', 1004, '01/07/2013, 3400,NULL, 30))

Insert into Emp (Empno, Ename, job, Mgr, HireDate, Sal, Comm, Deptno)

Values(1011, 'Prakesh', 'Analyst', 1004, '01/07/2013, 3600,NULL, 30))

Insert into Emp (Empno, Ename, job, Mgr, HireDate, Sal, Comm, Deptno)

Values(1012, 'Madan', 'Analyst', 1004, '01/07/2013, 3100,NULL, 30))

Insert into Emp (Empno, Ename, job, Mgr, HireDate, Sal, Comm, Deptno)

Values(1013, 'Ravi', 'Clerk', 1002, '01/07/2013, 1800,NULL, 10))

Insert into Emp (Empno, Ename, job, Mgr, HireDate, Sal, Comm, Deptno)

Values(1014, 'Raju', 'Clerk', 1005, '01/07/2013, 2300,NULL, 20))

Insert into Emp (Empno, Ename, job, Mgr, HireDate, Sal, Comm, Deptno)

Values(1015, 'Ramesh', 'Clerk', 1011, '01/07/2013, 2500, NULL, 30))

Data in DeptDetails Table:

Insert into DeptDetails values(1, 10, 'This department is located in Mumbai and mainly involed in marketing')

Insert into DeptDetails values(2, 20, 'This department is located in Chennai and mainly involed in Sales')

Insert into DeptDetails values(3, 30, 'This department is located in Delhi and mainly involed in Finance')

Insert into DeptDetails values(4, 40, 'This department is located in Kolkota and mainly involed in Prodution')

Data in Salgrade Table:

Insert into SalGrade Values(1, 1300, 1800)

Insert into SalGrade Values(2, 1800, 2700)

Insert into SalGrade Values(3, 2700, 3500)

Insert into SalGrade Values(4, 3500, 5000)

Insert into SalGrade Values(5, 5000, 8000)