SQL Server 2008(T-SQL)

Pragati Software Pvt. Ltd.

www.pragatisoftware.com

Index

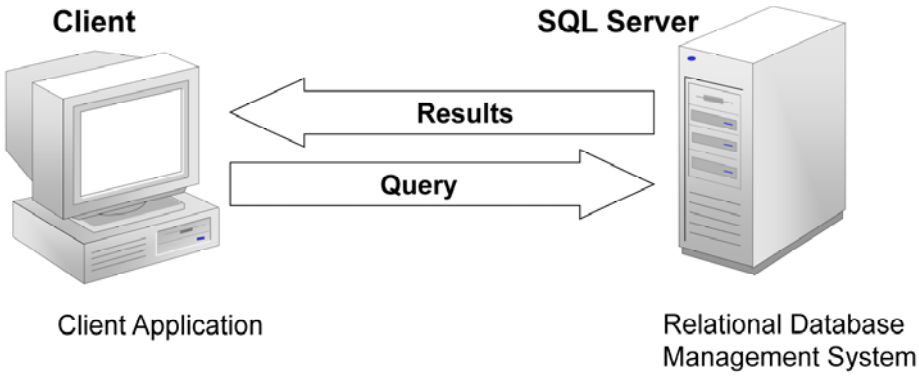
Index

Module 1:Overview of SQL Server

- Overview
  - What is SQL Server?
  - SQL Server Databases

What is SQL Server?

- High-performance, client/server relational database management system (RDBMS)
- Part of the core family of integrated products, including development tools, systems management tools, distributed system components and development interfaces

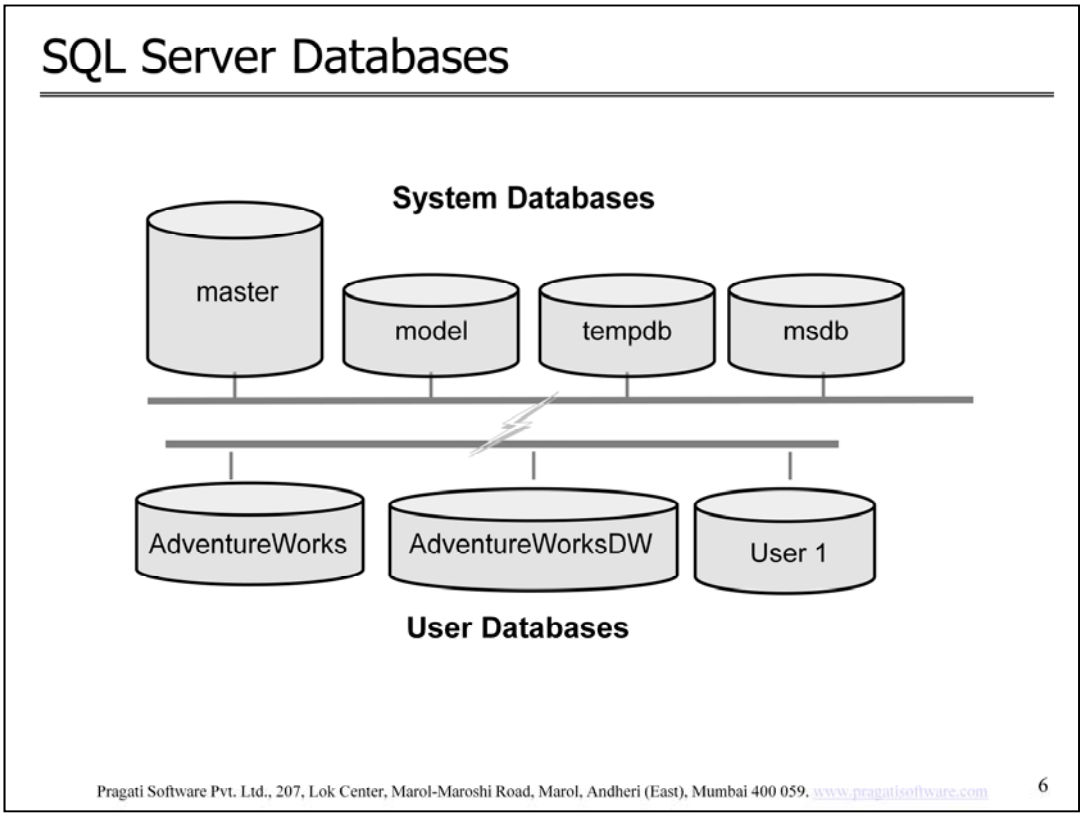


Pragati Software Pvt. Ltd., 207, Lok Center, Marol-Maroshi Road, Marol, Andheri (East), Mumbai 400 059. www.pragatisoftware.com 5

What is Microsoft SQL Server?

Microsoft SQL Server 2008 is a high-performance, client/server relational database management system (RDBMS). It was designed to support high-volume transaction processing (such as that for online order entry, inventory, accounting, or manufacturing) as well as data warehousing and decision-support applications (such as sales analysis applications).

SQL Server 2008 also provides many client tools and networking interfaces for other Microsoft operating systems, such as Windows 3.1 and MS-DOS. And because of SQL Server's open architecture, other systems (for example, UNIX-based systems) can interoperate with it as well. SQL Server is part of the core of a family of integrated products, including development tools, systems management tools, distributed system components, and open development interfaces.



Types of Databases

Each SQL Server has two types of databases: *system databases* and *user databases*. System databases store information about SQL Server as a whole. SQL Server uses system databases to operate and manage the system. User databases are databases that users create. When SQL Server is installed, SQL Server Setup creates system databases and sample user databases. The Distribution Database is installed when you configure SQL Server for replication activities. The following table describes each database.

Database Description

master : Controls the user databases and operation of SQL Server as a whole by keeping track of information such as user accounts, configurable environment variables, and system error messages

model : Provides a template or prototype for new user databases

tempdb : Provides a storage area for temporary tables and other temporary working storage needs

msdb : Provides a storage area for scheduling information and job history

AdventureWorks : Provides a sample database as a learning tool

AdventureWorksDW : Provides a sample database as a learning tool

User1 : Identifies a user-defined database

Module 2:Data Retrieval

- Overview
  - Retrieving Data Using the SELECT Statement
  - Filtering Data
  - Formatting Result Sets

Retrieving Data Using the SELECT Statement

Syntax:

```
SELECT <select_list>
FROM {<table_source>} [,...n]
WHERE <search_condition>
```

Examples:

SELECT * from Products

SELECT **ProductID, ProductName, UnitPrice, UnitsInStock, Discontinued**
FROM Products

SELECT * FROM Products **WHERE** Discontinued = 1

SELECT **ProductID, ProductName, UnitPrice, UnitsInStock, Discontinued**
FROM Products **WHERE** Discontinued = 1

Pragati Software Pvt. Ltd., 207, Lok Center, Marol-Maroshi Road, Marol, Andheri (East), Mumbai 400 059. www.pragatisoftware.com 8

Viewing all rows & all columns FROM a table

SELECT * from Products

Viewing all rows & selected columns FROM a Table

SELECT ProductID, ProductName, UnitPrice, Discontinued FROM Products

Conditional Retrieval

The WHERE clause is used in a SELECT statement to specify the criteria for retrieval of data. It enable us to lay down certain condition on the basis of which rows are retrieved FROM the table.

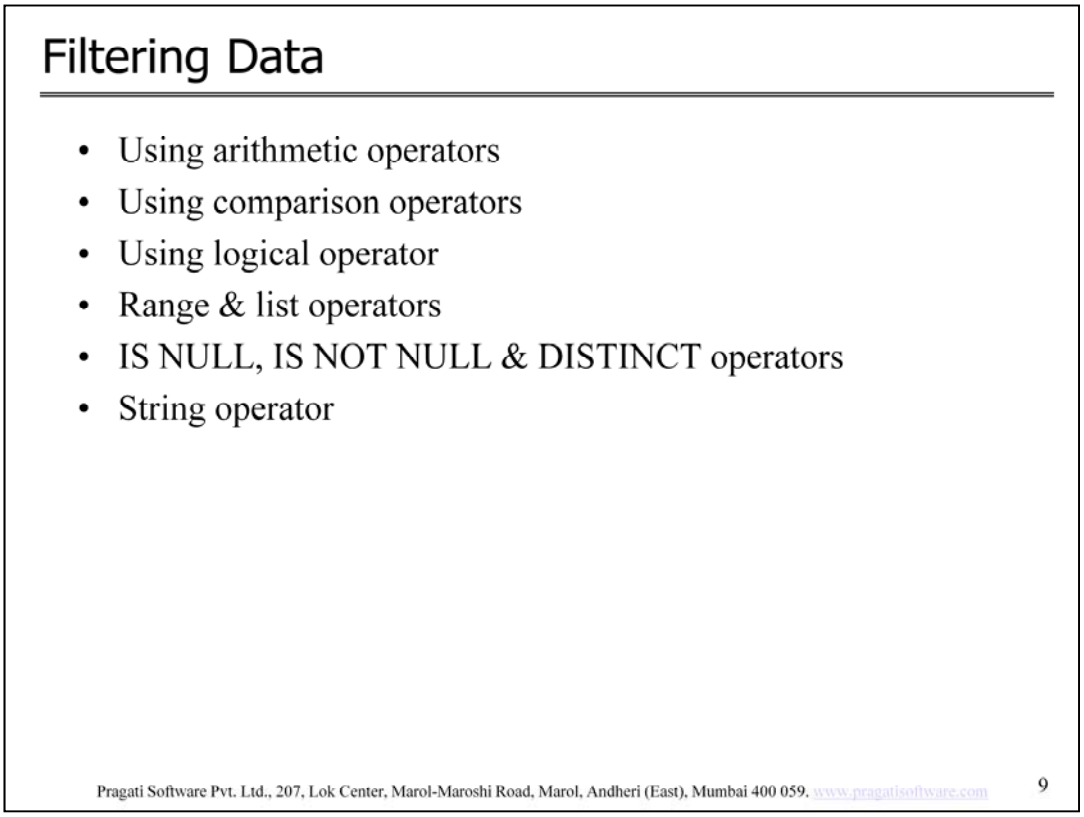
We make use of a platter of Operators, in conjunction with the WHERE clause, that enables us to retrieve the data conditionally. The list of Operators are mentioned in detail towards the end of this chapter.

Viewing selected rows & all columns FROM a Table

SELECT * FROM Products WHERE Discontinued = 1

Viewing selected rows & selected columns FROM a Table

SELECT ProductID, ProductName, UnitPrice, Discontinued FROM Products
WHERE Discontinued = 1



You sometimes want to limit the results that a query returns. You can limit the results by specifying search conditions in a WHERE clause to filter data. There is no limit to the number of search conditions that you can include in a SELECT statement. The following table describes the type of filter and the corresponding search condition that you can use to filter data.

Type of filter Search condition

Comparison operators =, >, <, >=, <=, and <>

String comparisons LIKE and NOT LIKE

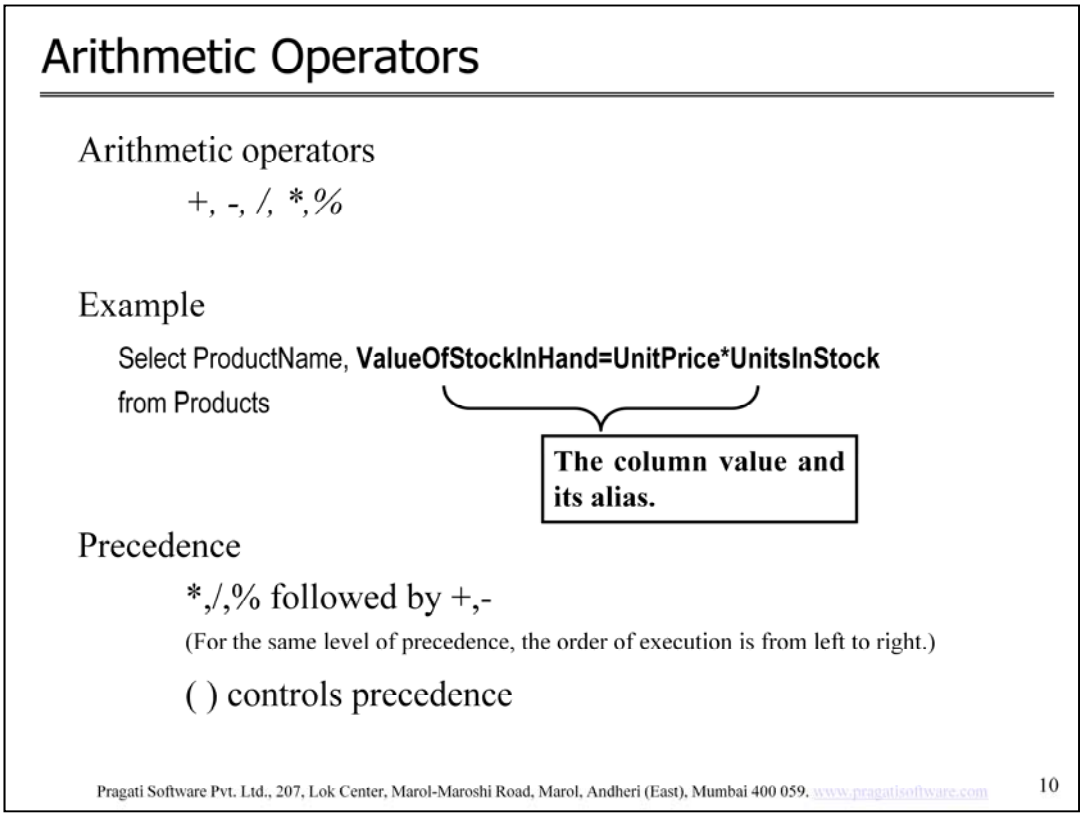
Logical operators: combination of conditions AND, OR

Logical operator: negations NOT

Range of values BETWEEN and NOT BETWEEN

Lists of values IN and NOT IN

Unknown values IS NULL and IS NOT NULL



+, -, /, *,%

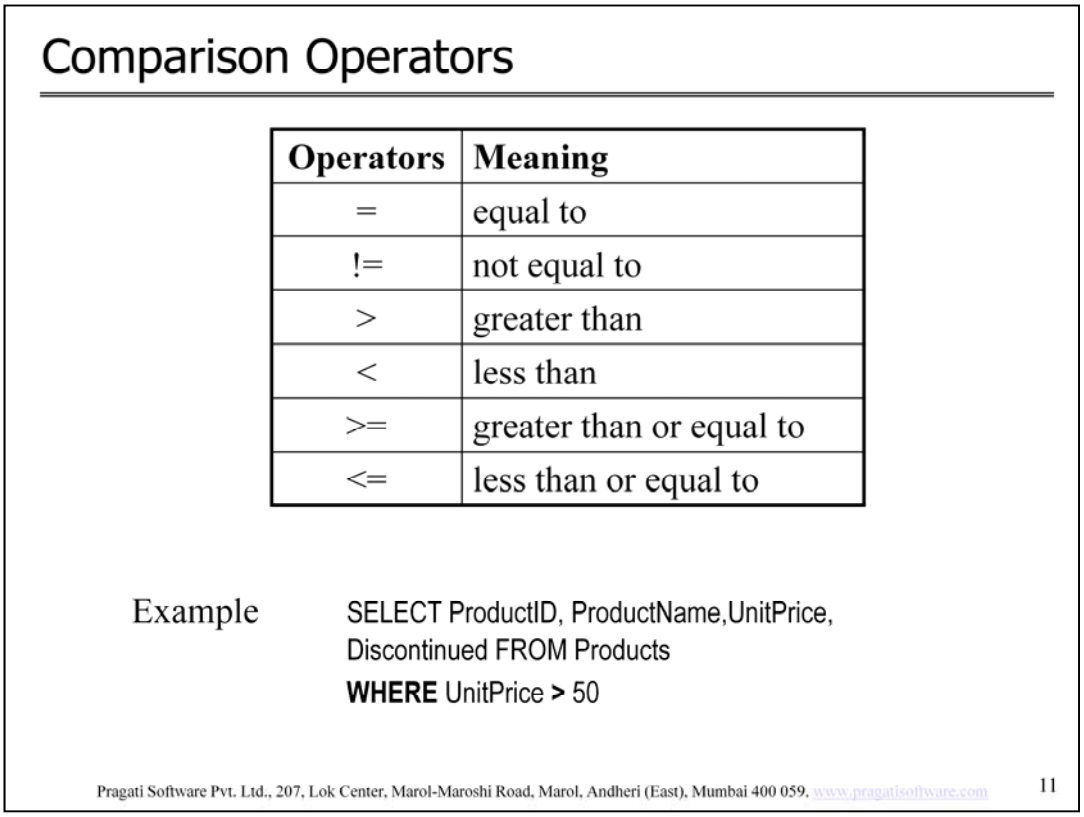
Can be used with numeric columns or constants.

The % operator cannot be used with money, smallmoney, float or real data types.

Precedence

*,/,% followed by +,-

When arithmetic expression uses same level of precedence, the order of execution is from left to right.



Use comparison operators to compare the values in a table to a specified value or expression. You also can use comparison operators to check for a condition.

Comparison operators compare columns or variables of compatible data types.

The comparison operators are listed in the following table.

Operator Description

= Equal to

> Greater than

< Less than

>= Greater than or equal to

<= Less than or equal to

<> Not equal to

Note

Avoid the use of NOT in search conditions. They may slow data retrieval because all rows in a table are evaluated.

Logical Operators

Used to combine multiple search conditions

Operator	When does it returns result?
OR	Any of the specified search conditions is true
AND	All the specified search conditions are true
NOT	The search conditions is false

Examples

```
SELECT productid, productname, unitprice, discontinued FROM products
WHERE unitprice > 50 AND Discontinued=0

SELECT productid, productname, unitprice, discontinued FROM products
WHERE ProductName='Chai' OR ProductName='Northwoods Cranberry Sauce'
```


Use the logical operators AND, OR, and NOT to combine a series of expressions and to refine query processing. The results of a query may vary depending on the grouping of expressions and the order of the search conditions.

When you use logical operators, consider the following guidelines:

- Use the AND operator to retrieve rows that meet all of the search criteria.
- Use the OR operator to retrieve rows that meet any of the search criteria.
- Use the NOT operator to negate the expression that follows the operator.

Using Parentheses

Use parentheses when you have two or more expressions as the search criteria. Using parentheses allows you to:

- Group expressions.
- Change the order of evaluation.
- Make expressions more readable.

Order of Search Conditions

When you use more than one logical operator in a statement, consider the following facts:

- Microsoft SQL Server 2008 evaluates the NOT operator first, followed by the AND operator and then the OR operator.
- The precedence order is from left to right if all operators in an expression are of the same level.

Range & List Operators

- **BETWEEN:** Specifies an inclusive range to search

  SELECT productname, unitprice FROM products
  WHERE unitprice **BETWEEN** 11 AND 20

- **NOT BETWEEN:** Excludes rows of specified range

  SELECT productname, unitprice FROM products
  WHERE unitprice **NOT BETWEEN** 11 AND 20

- **IN:** Allows rows that match any values in the list

  SELECT OrderID,OrderDate,ShipCountry FROM orders
  WHERE ShipCountry **IN (**'USA', 'UK'**)**

- **NOT IN:** Disallows rows that match any values in the list

  SELECT OrderID,OrderDate,ShipCountry FROM orders
  WHERE ShipCountry **NOT IN (**'USA', 'UK'**)**

Use the BETWEEN search condition in the WHERE clause to retrieve rows that are within a specified range of values. When you use the BETWEEN search condition, consider the following facts and guidelines:

•SQL Server includes the end values in the result set.

•Use the BETWEEN search condition rather than an expression that includes the AND operator with two comparison operators (> = x AND < = y). However, to search for an exclusive range in which the returned rows do not contain the end values, use an expression that includes the AND operator with two comparison operators (> x AND < y).

•Use the NOT BETWEEN search condition to retrieve rows outside of the specified range. Be aware that using NOT conditions may slow data retrieval.

```
SELECT productname, unitprice
FROM products WHERE unitprice BETWEEN 10 AND 20
```

•Use the IN search condition in the WHERE clause to retrieve rows that match a specified list of values. When you use the IN search condition, consider the following guidelines:

•Use either the IN search condition or a series of comparison expressions that are connected with an OR operator .SQL Server resolves them in the same way, returning identical result sets.

•Do not include a null value in the search condition. A null value in the search condition list evaluates to the expression, = NULL. This may return unpredicted result sets.

•Use the NOT IN search condition to retrieve rows that are not in your specified list of values. Be aware that using NOT conditions may slow data retrieval.

This example produces a list of companies from the **suppliers** table that are located in Japan or Italy.

```
SELECT companyname, country
FROM suppliers WHERE country IN ('Japan', 'Italy')
```

IS NULL,IS NOT NULL & DISTINCT Operators

NULL is an unknown value.

- When compared with any other value using comparison operators, the result is always NULL
- To check for NULL values you need to use IS NULL or IS NOT NULL keyword
- No two NULL values are equal
- NULL values are always first to be displayed in an ascending order

Example :

SELECT * from orders where shippeddate **is null**

Distinct keyword : removes duplicate rows from the result set

Example : SELECT **distinct** shipcountry FROM orders

A column has a null value if no value is entered during data entry and no default values are defined for that column. A null value is not the same as entries with a zero (a numerical value) or a blank (a character value).

Use the IS NULL search condition to retrieve rows in which information is missing from a specified column. When you retrieve rows that contain unknown values, consider the following facts and guidelines:

•Null values fail all comparisons because they do not evaluate equally with one another.

•You define whether columns allow null values in the CREATE TABLE statement.

•Use the IS NOT NULL search condition to retrieve rows that have known values in the specified columns.

This example retrieves a list of companies from the **suppliers** table for which the **fax** column contains a null value.

```
SELECT companyname, fax FROM suppliers WHERE fax IS NULL
```

Viewing distinct values

```
SELECT distinct city from Employees
```

Without the keyword DISTINCT in the above command, a city will be displayed FROM every record in the table. Since a city may be found in more than one employee record, there will be a duplication of cities. The keyword DISTINCT ensures that a value is displayed only once.

String Operator

Wildcard	Description
%	Any string of 0 or more characters
_	Single character
[ ]	Any single character within specified range
[^]	Any single character not within specified range

Example

```
SELECT ProductName FROM Products
WHERE ProductName LIKE '%tea%'
or  ProductName LIKE '%coffee%'

SELECT distinct shipcountry  FROM orders
        where shipcountry like '_ _ _ _ _'
```

Pragati Software Pvt. Ltd., 207, Lok Center, Marol-Maroshi Road, Marol, Andheri (East), Mumbai 400 059. www.pragatisoftware.com 15

You can use the LIKE search condition in combination with wildcard characters to select rows by comparing character strings. When you use the LIKE search condition, consider the following facts:

All characters in the pattern string are significant, including leading and trailing blank spaces.

LIKE can be used only with data of the **char**, **nchar**, **varchar**, **nvarchar**, or **datetime** data types.

Use the following four wildcard characters to form your character string search criteria.

% Any string of zero or more characters

_ Any single character

[] Any single character within the specified range or set

[^] Any single character *not* within the specified range or set

The following table lists examples of the use of wildcards with the LIKE search condition.

LIKE 'BR%' Every name beginning with the letters BR

LIKE 'Br%' Every name beginning with the letters Br

LIKE '%een' Every name ending with the letters een

LIKE '%en%' Every name containing the letters en

LIKE '_en' Every three-letter name ending in the letters en

LIKE '[CK]%' Every name beginning with the letter C or K

LIKE '[S-V]ing' Every four-letter name ending in the letters ing and beginning with any single letter from S to V

LIKE 'M[^c]%' Every name beginning with the letter M that does not have the letter c as the second letter

This example retrieves companies from the customers table that have the word restaurant in their company names.

```
SELECT companyname FROM customers
WHERE companyname LIKE '%Restaurant%'
```

Formatting Result Sets

Sorting data

- ORDER BY clause:
  - Used to display results in specified order
  - Default sort order is Ascending

Example

```
SELECT productid, productname, categoryid, unitprice
FROM products  ORDER BY categoryid, unitprice DESC
```

Changing column names

```
SELECT   ProductName,   UnitPrice*UnitsInStock   AS   ValueOfStockInHand
      FROM Products
```

Using literals

```
SELECT ProductName ,' costs : ' , UnitPrice FROM Products
```


You can improve the readability of a result set by sorting the order in which the result set is listed, eliminating any duplicate rows, changing column names to column aliases, or using literals to replace result set values. These formatting options do not change the data, only the presentation of it.

Module 3: Grouping and Summarizing Data

- Overview
  - Listing the TOP n Values
  - Using Aggregate Functions
  - Using the GROUP BY clause
  - GROUP BY clause with the HAVING clause
  - Using the COMPUTE and COMPUTE BY Clauses

Listing the TOP *n* Values

- Lists only the first *n* rows of a result set
- Specifies the range of values in the ORDER BY clause
- Returns ties if WITH TIES is used

Examples:

```
SELECT TOP 1 orderid, productid, quantity
FROM [order details] ORDER BY quantity DESC

SELECT TOP 1 WITH TIES orderid, productid, quantity
FROM [order details] ORDER BY quantity DESC
```


Use the TOP *n* keyword to list only the first *n* rows or *n* percent of a result set. Although the TOP *n* keyword is not ANSI-standard, it is useful, for example, to list a company's top selling products.

When you use the TOP *n* or TOP *n* PERCENT keyword, consider the following facts and guidelines:

•Specify the range of values in the ORDER BY clause. If you do not use an ORDER BY clause, SQL Server returns rows that satisfy the WHERE clause in no particular order.

•Use an unsigned integer following the TOP keyword.

•If the TOP *n* PERCENT keyword yields a fractional row, SQL Server rounds to the next integer value.

•Use the WITH TIES clause to include ties in your result set. Ties result when two or more values are the same as the last row that is returned in the ORDER BY clause. Your result set may therefore include any number of rows.

Note

You can use the WITH TIES clause only when an ORDER BY clause exists.

Using Aggregate Functions

Aggregate function	Description
AVG	Average of values in a numeric expression
COUNT	Number of values in an expression
COUNT(*)	Number of selected rows
MAX	Highest value in the expression
MIN	Lowest value in the expression
SUM	Total values in a numeric expression

Functions that calculate averages and sums are called *aggregate functions*.

When an aggregate function is executed, SQL Server summarizes values for an entire table or for groups of columns within the table, producing a single value for each set of rows for the specified columns:

•You can use aggregate functions with the SELECT statement or in combination with the GROUP BY clause.

•With the exception of the COUNT(*) function, all aggregate functions return a NULL if no rows satisfy the WHERE clause. The COUNT(*) function returns a value of zero if no rows satisfy the WHERE clause.

Using Aggregate Functions with Null Values

- Most aggregate functions ignore null values
- COUNT(*) function includes records with null values

Examples

```
SELECT COUNT(*)  FROM orders
SELECT COUNT(shippeddate) FROM orders
SELECT SUM(quantity)  FROM [order details]
SELECT AVG(unitprice) FROM products
SELECT  MAX (unitprice) FROM products
SELECT  MIN (unitprice) FROM products
```


Function data type

COUNT

COUNT is the only aggregate function that can be used on columns with **text**, **ntext**, or **image** data types.

MIN and MAX

You cannot use the MIN and MAX functions on columns with **bit** data types.

SUM and AVG

You can use only the SUM and AVG aggregate functions on columns with **int**, **smallint**, **tinyint**, **decimal**, **numeric**, **float**, **real**, **money**, and **smallmoney** data types.

Using the GROUP BY Clause

Use the GROUP BY clause…

- On columns or expressions
- To organize rows into groups and to summarize those groups

Examples

```
SELECT productid, orderid, quantity
FROM [order details]

SELECT productid ,SUM(quantity) AS total_quantity
FROM [order details] GROUP BY productid ORDER BY productid
```


Use the GROUP BY clause on columns or expressions to organize rows into groups and to summarize those groups. For example, use the GROUP BY clause to determine the quantity of each product that was ordered for all orders.

When you use the GROUP BY clause, consider the following facts and guidelines:

•SQL Server produces a column of values for each defined group.

•SQL Server returns only single rows for each group that you specify; it does not return detail information.

•All columns that are specified in the GROUP BY clause must be included in the select list.

•If you include a WHERE clause, SQL Server groups only the rows that satisfy the WHERE clause conditions.

•You can have up to 8,060 bytes in the column list of the GROUP BY clause.

•Do not use the GROUP BY clause on columns that contain multiple null values because the null values are processed as a group.

•Use the ALL keyword with the GROUP BY clause to display all rows with null values in the aggregate columns, regardless of whether the rows satisfy the WHERE clause.

GROUP BY Clause with the HAVING Clause

Examples

- SELECT productid ,SUM(quantity) AS total_quantity
  FROM [order details] **GROUP BY** productid ORDER BY 2

- SELECT productid, SUM(quantity) AS total_quantity
  FROM [order details]
  GROUP BY productid **HAVING** SUM(quantity)>=400

Use the HAVING clause on columns or expressions to set conditions on the groups included in a result set. The HAVING clause sets conditions on the GROUP BY clause in much the same way that the WHERE clause interacts with the SELECT statement.

Using the COMPUTE Clauses

COMPUTE clause with Select statement is used to generate summary rows using aggregate functions in query results

Example

```
SELECT productid, orderid ,quantity FROM [order details]
COMPUTE SUM(quantity)
COMPUTE AVG(quantity)
```


The COMPUTE clause produces detailed rows and a single aggregate value for a column. When you use the COMPUTE clause, consider the following facts and guidelines:

•You can use multiple COMPUTE clauses with the COMPUTE BY clause in a single statement.

•SQL Server requires that you specify the same columns in the COMPUTE clause that are listed in the select list.

Using the COMPUTE BY Clauses

COMPUTE BY clause is used to calculate summary values of resultset **on a group** of data

Used to generate a group summary report with individual data rows from table (control-break reports).

Example

```
SELECT productid, orderid, quantity  FROM [order details]
ORDER BY productid, orderid
COMPUTE SUM(quantity) BY productid
COMPUTE SUM(quantity)
```

Limitation

The ORDER BY clause must be used with COMPUTE BY

The COMPUTE BY clause generates detail rows and multiple summary values. Summary values are generated when column values change. Use COMPUTE BY for data that is easily categorized. When you use the COMPUTE BY clause, consider the following facts and guidelines:

•You should use an ORDER BY clause with the COMPUTE BY clause so that rows are grouped together.

•Specify the column names after the COMPUTE BY clause to determine which summary values that SQL Server generates.

•The columns listed after the COMPUTE BY clause must be identical to or a subset of those that are listed after the ORDER BY clause. They must be listed in the same order (left-to-right), start with the same expression, and not skip any expressions.

Module 4: Creating Table with Constraints

- Overview
  - System-supplied Data Types
  - Creating a Table
  - Creating Constraints
  - Types of Constraints
  - Disabling Constraints

System-supplied Data Types

Data type	Range	Stores
int	-2^31 (-2,147,483,648) to 2^31-1 (2,147,483,647)	Integer Data
smallint	-2^15 (-32,768) to 2^15-1 (32,767)	Integer Data
tinyint	0 to 255	Integer Data
bigint	-2^63 (-9,223,372,036,854,775,808) to 2^63-1(9,223,372,036,854,775,807)	Integer Data
money	-922,337,203,685,477.5808 to 922,337,203,685,477.5807	Monetory Data
smallmoney	-214,748.3648 to 214,748.3647	Monetory Data
float	- 1.79E+308 to 1.79E+308	Floating precision data

Data Types

The given data types in the above slide are available for columns in a table.

System-supplied Data Types (Contd…)

Data type	Range	Stores
numeric (p,s)	10^38 +1 through 10^38 – 1 Default Precision is 18	Fixed-Precision data
char (n)	n characters where n can be 1 to 8000	Fixed-length character data.
varchar (n)	n characters where n can be 1 to 8000	Variable-length character data
text	Maximum Length of 2^31 – 1(2,417,483,647) characters	Character String
image	Maximum Length of 2^31 – 1(2,417,483,647) bytes	Variable length binary data for images
datetime	01/01/1753 to 12/31/9999	Date and Time data
rowversion	Max. storage of 8 bytes	Unique no. that gets updated every time a row is inserted or updated

Data Types

The given data types in the above slide are available for columns in a table.

System-supplied Data Types

- Guidelines for specifying data types
  - If column length varies, use a variable data type
  - Use 'tinyint' appropriately
  - For numeric data types, commonly use decimal
  - If storage is greater than 8000 bytes, use text or image
  - Do not use float or real as primary keys

Consider the following guidelines for selecting data types and balancing storage size with requirements:

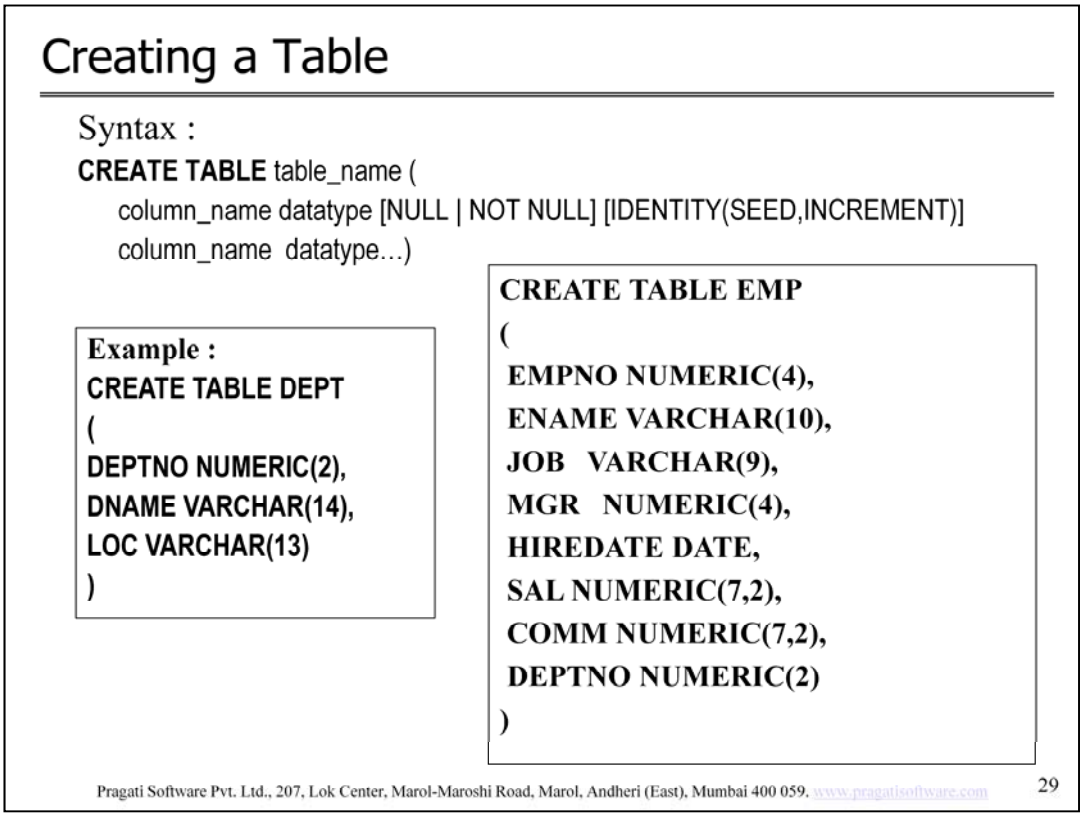
•If column length varies, use one of the variable data types. For example, if you have a list of names, you can set it to **varchar** instead of **char** (fixed).

•If you own a growing bookselling business with many locations, and you have specified the **tinyint** data type for the store identifier in the database, you will have problems when you decide to open store number 256.

•For numeric data types, the size and required level of precision helps to determine your choice. In general, use **decimal**.

•If the storage is greater than 8000 bytes, use **text** or **image**. If it is less than 8000, use **binary** or **char**. When possible, it is best to use **varchar** because it has more functionality than **text** and **image**.

•Do not use the approximate data types **float** and **real** as primary keys. Because the values of these data types are not precise, it is not appropriate to use them in comparisons.



Naming Tables and Columns

It must consist of a combination of 1 through 128 letters, digits, or the symbols #, $, @, or _.

Syntax :

CREATE TABLE table_name (

column_name datatype [NULL | NOT NULL] [IDENTITY(SEED,INCREMENT)],

column_name datatype

)

Where,

table_name is the name of the table

column_name name of column in table

datatype is system-defined or userdefined datatype

[NULL | NOT NULL] keywords allowing/disallowing null values for column

IDENTITY used to generate sequential numbers

SEED starting or the initial value for IDENTITY column

INCREMENT step value to generate next value (can be –ve).

Creating Constraints

- Use CREATE TABLE or ALTER TABLE
- Can add constraints to a table with existing data
- Can place constraints on single or multiple columns
  - Single column, called column-level constraint
  - Multiple columns, called table-level constraint
- Considerations for using constraints
  - Can be changed without recreating a table
  - Require error-checking in applications and transactions
  - Verify existing data

You create constraints by using the CREATE TABLE or ALTER TABLE statement. You can add constraints to a table with existing data, and you can place constraints on single or multiple columns:

•If the constraint applies to a single column, it is called a *column-level* constraint.

•If a constraint references multiple columns, it is called a *table-level* constraint, even if it does not reference all columns in the table.

Syntax :

```
CREATE TABLE table_name
( { < column_definition >   | < table_constraint > } [ ,...n ])
< column_definition > ::= { column_name data_type }
[ [ DEFAULT constant_expression ]  [ < column_constraint > ] [ ,..n]
< column_constraint > ::=  [ CONSTRAINT constraint_name ]   |
[ { PRIMARY KEY | UNIQUE }
[ CLUSTERED | NONCLUSTERED ] ]  |
[ [ FOREIGN KEY ]   REFERENCES ref_table [ ( ref_column ) ]
[ ON DELETE { CASCADE | NO ACTION } ]
[ ON UPDATE { CASCADE | NO ACTION } ]]  |
CHECK ( logical_expression ) }
< table_constraint > ::= [ CONSTRAINT constraint_name ]
{ [ { PRIMARY KEY | UNIQUE } [ CLUSTERED | NONCLUSTERED ]
{ ( column [ ASC | DESC ] [ ,...n ] ) } ]  |
FOREIGN KEY   [ ( column [ ,...n ] ) ]
REFERENCES ref_table [ ( ref_column [ ,...n ] ) ]
[ ON DELETE { CASCADE | NO ACTION } ]
[ ON UPDATE { CASCADE | NO ACTION } ]    |
CHECK ( search_conditions ) }
```

Types of Constraints

- DEFAULT Constraints
- CHECK Constraints
- PRIMARY KEY Constraints
- UNIQUE Constraints
- FOREIGN KEY Constraints
- Cascading Referential Integrity

This section describes the types of constraints. Syntax, examples, and considerations for use define each constraint.

DEFAULT Constraints

- Apply only to INSERT statements
- Only one DEFAULT constraint per column
- Allow some system-supplied values

```
ALTER TABLE Emp
ADD CONSTRAINT DF_Emp_Job
DEFAULT 'Executive' FOR Job
```


A DEFAULT constraint enters a value in a column when one is not specified in an INSERT statement. DEFAULT constraints enforce domain integrity.

Syntax :

```
[CONSTRAINT constraint_name]
DEFAULT constant_expression
```

This example adds a DEFAULT constraint that inserts the UNKNOWN value in the **Customers** table if a contact name is not provided.

Example

```
ALTER TABLE Customers
ADD CONSTRAINT DF_contactname DEFAULT 'UNKNOWN' FOR ContactName
```

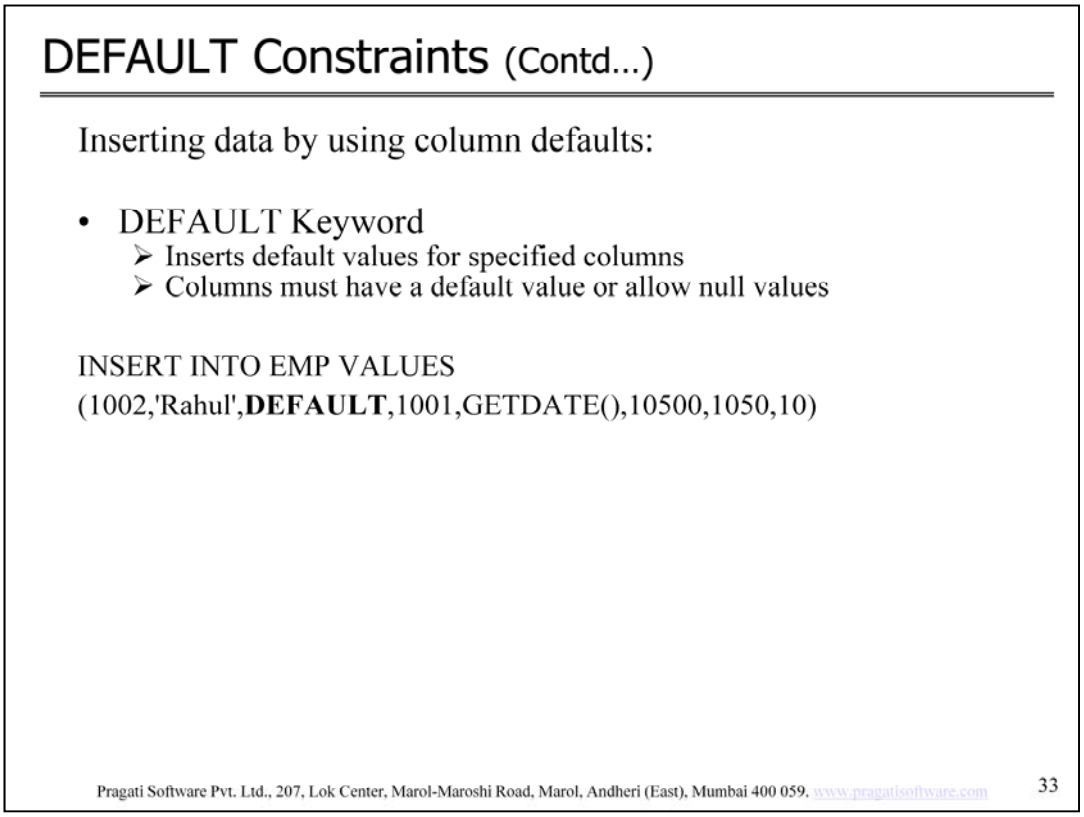
Consider the following facts when you apply a DEFAULT constraint:

•It verifies existing data in the table.

•It applies only to INSERT statements.

•Only one DEFAULT constraint can be defined per column.

•It cannot be placed on columns with the **Identity** property or on columns with the **rowversion** data type.



When you insert rows into a table, you can save time when entering values by using the DEFAULT or DEFAULT VALUES keywords with the INSERT statement.

DEFAULT Keyword

When a table has default constraints, or when a column has a default value, use the DEFAULT keyword in the INSERT statement to have SQL Server supply the default value for you. When you use the DEFAULT keyword, consider the following facts and guidelines:

•SQL Server inserts a null value for columns that allow null values and do not have default values.

•If you use the DEFAULT keyword, and the columns do not have default values or allow null values, the INSERT statement fails.

•You cannot use the DEFAULT keyword with a column that has the IDENTITY property (an automatically assigned, incremented value).

Therefore, do not list columns with an IDENTITY property in the *column_list* or VALUES clause.

•SQL Server inserts the next appropriate value for columns that are defined with the **rowversion** data type.

CHECK Constraints

- Are used with INSERT and UPDATE statements
- Can reference other columns in the same table
- Cannot:
  - Be used with the **rowversion** data type
  - Contain subqueries

Example:

```
ALTER TABLE Emp
ADD  CONSTRAINT CK_HireDate
CHECK (HireDate > '01-01-1900' AND HireDate < getdate())
```


A CHECK constraint restricts the data that users can enter into a particular column to specific values. CHECK constraints are similar to WHERE clauses in that you can specify the conditions under which data will be accepted.

Syntax :

```
[CONSTRAINT constraint_name]
CHECK (logical_expression)
```

This example adds a CHECK constraint to ensure that a birth date conforms to an acceptable range of dates.

Example :

```
ALTER TABLE Emp
ADD  CONSTRAINT CK_DateOfBirth
CHECK (DateOfBirth > '01-01-1900' AND DateOfBirth < getdate())
```

Consider the following facts when you apply a CHECK constraint:

•It verifies data every time that you execute an INSERT or UPDATE statement.

•It can reference other columns in the same table.

For example, a **salary** column could reference a value in a **job_grade** column.

•It cannot be placed on columns with the **rowversion** data type.

•It cannot contain subqueries.

•If any data violates the CHECK constraint, you can execute the DBCC CHECKCONSTRAINTS statement to return the violating rows.

PRIMARY KEY Constraints

- Only one PRIMARY KEY constraint per table
- Values must be unique
- Null values are not allowed

Example:

```
ALTER TABLE Emp
ADD CONSTRAINT PK_EmpNo
PRIMARY KEY (Empno)
```


A PRIMARY KEY constraint defines a primary key on a table that uniquely identifies a row. It enforces entity integrity.

Syntax :

```
[CONSTRAINT constraint_name]
PRIMARY KEY [CLUSTERED | NONCLUSTERED] z
{ ( column[,...n] ) }
```

Example :

```
ALTER TABLE Emp
ADD CONSTRAINT PK_EmpNo
PRIMARY KEY NONCLUSTERED (Empno)
```

Consider the following facts when you apply a PRIMARY KEY constraint:

•Only one PRIMARY KEY constraint can be defined per table.

•The values entered must be unique.

•Null values are not allowed.

•It creates a unique index on the specified columns. You can specify a clustered or nonclustered index (clustered is the default if it does not already exist).

Note:

The index created for a PRIMARY KEY constraint cannot be dropped directly. It is dropped when you drop the constraint.

UNIQUE Constraints

- Allow one null value
- Allow multiple UNIQUE constraints on a table
- Defined with one or more columns

Example:

```
ALTER TABLE Dept
ADD CONSTRAINT U_DeptName
UNIQUE (DeptName)
```


A UNIQUE constraint specifies that two rows in a column cannot have the same value. This constraint enforces entity integrity with a unique index. A UNIQUE constraint is helpful when you already have a primary key, such as an employee number, but you want to guarantee that other identifiers, such as an employee's driver's license number, are also unique.

Syntax :

```
CONSTRAINT constraint_name]
UNIQUE [CLUSTERED | NONCLUSTERED]
{ ( column[,...n] ) }
```

Example :

```
ALTER TABLE Dept
        ADD CONSTRAINT U_DeptName
        UNIQUE NONCLUSTERED (DeptName)
```

Consider the following facts when you apply a UNIQUE constraint:

•It can allow one null value.

•You can place multiple UNIQUE constraints on a table.

•You can apply the UNIQUE constraint to one or more columns that must have unique values, but are not the primary key of a table.

•The UNIQUE constraint is enforced through the creation of a unique index on the specified column or columns.

FOREIGN KEY Constraints

- Must reference a PRIMARY KEY or UNIQUE constraint
- Provide single or multicolumn referential integrity
- Users must have SELECT or REFERENCES permissions on referenced tables
- Use only REFERENCES clause within same table

Example:

```
ALTER TABLE Dept ALTER Column DeptNo int NOT NULL

ALTER TABLE Dept add constraint pk_dept Primary Key (DeptNo)

ALTER TABLE dbo.Emp
ADD CONSTRAINT FK_DeptNo
FOREIGN KEY (DeptNo)
REFERENCES Dept(Deptno)
```


A FOREIGN KEY constraint enforces referential integrity. The FOREIGN KEY constraint defines a reference to a column with a PRIMARY KEY or UNIQUE constraint in the same, or another table.

Syntax :

```
[CONSTRAINT constraint_name]
[FOREIGN KEY] [(column[,.n])]
REFERENCES ref_table [(ref_column [,.n])].
```

This example uses a FOREIGN KEY constraint to ensure that customer identification in the **dbo.Orders** table is associated with a valid identification in the **dbo.Customers** table.

```
USE Northwind
ALTER TABLE dbo.Orders
ADD CONSTRAINT FK_Orders_Customers
FOREIGN KEY (CustomerID) REFERENCES dbo.Customers(CustomerID)
```

Example :

Consider the following facts and guidelines when you apply a FOREIGN KEY constraint:

•It provides single or multicolumn referential integrity. The number of columns and data types that are specified in the FOREIGN KEY statement must match the number of columns and data types in the REFERENCES clause.

•Unlike PRIMARY KEY or UNIQUE constraints, FOREIGN KEY constraints do not create indexes automatically. However, if you will be using many joins in your database, you should create an index for the FOREIGN KEY to improve join performance.

•To modify data, users must have SELECT or REFERENCES permissions on other tables that are referenced with a FOREIGN KEY constraint.

•You can use only the REFERENCES clause without the FOREIGN KEY clause when you reference a column in the same table.

Cascading Referential Integrity

- [CONSTRAINT *constraint_name*]
- [FOREIGN KEY] [(*column*[,*.n*])]
- REFERENCES *ref_table* [(*ref_column* [,*.n*])].
- [ON DELETE { CASCADE | NO ACTION }]
- [ON UPDATE { CASCADE | NO ACTION }]

Example:

```
ALTER TABLE EMP DROP Constraint FK_DeptNo

ALTER TABLE Emp
 ADD CONSTRAINT FK_DeptNo
 FOREIGN KEY (DeptNo)
 REFERENCES Dept(DeptNo)
 ON DELETE CASCADE
```

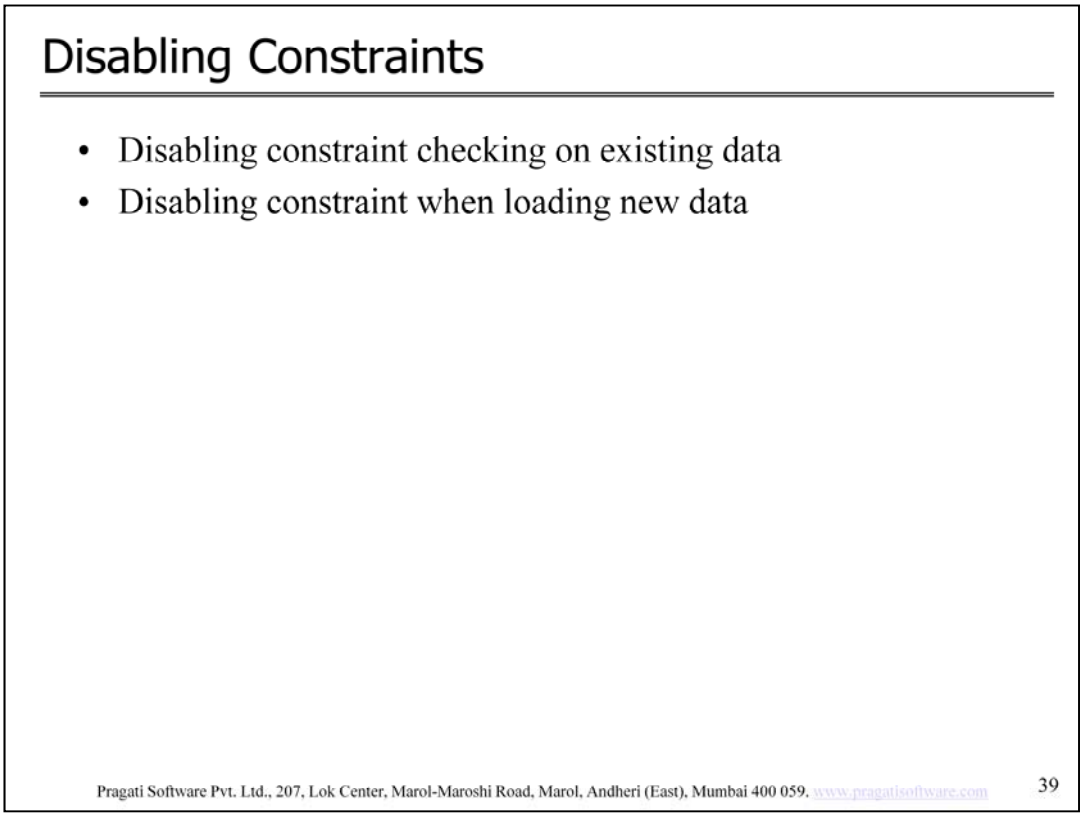
Pragati Software Pvt. Ltd., 207, Lok Center, Marol-Maroshi Road, Marol, Andheri (East), Mumbai 400 059. www.pragatisoftware.com 38

The FOREIGN KEY constraint includes a CASCADE option that allows any change to a column value that defines a UNIQUE or PRIMARY KEY constraint to automatically propagate the change to the foreign key value. This action is referred to as *cascading referential integrity*. The REFERENCES clauses of the CREATE TABLE and ALTER TABLE statements support ON DELETE and ON UPDATE clauses. These clauses allow you to specify the CASCADE or NO ACTION option.

Syntax :

```
[CONSTRAINT constraint_name]
[FOREIGN KEY] [(column[,.n])]
REFERENCES ref_table [(ref_column [,.n])].
[ ON DELETE { CASCADE | NO ACTION } ]
[ ON UPDATE { CASCADE | NO ACTION } ]
```

NO ACTION specifies that any attempt to delete or update a key referenced by foreign keys in other tables raises an error and the change is rolled back. NO ACTION is the default. If CASCADE is defined and a row is changed in the parent table, the corresponding row is then changed in the referencing table.



For reasons of performance, it is sometimes advisable to disable constraints. For example, it is more efficient to allow large batch operations to process before enabling constraints. This section describes how to disable constraint checking when you are creating a new constraint or disabling an existing one.

Disabling Constraint Checking on Existing Data

- Use WITH NOCHECK option to disable constraint checking on existing data when you add a constraint to the table
- Applies to CHECK and FOREIGN KEY constraints
- Data must conform to constraints if the data is updated.

Example:

```
ALTER TABLE EMP DROP Constraint FK_DeptNo

ALTER TABLE Emp
 WITH NOCHECK
 ADD CONSTRAINT FK_DeptNo
 FOREIGN KEY (DeptNo)
 REFERENCES Emp(EmpNo)
```


When you define a constraint on a table that already contains data, SQL Server checks the data automatically to verify that it meets the constraint requirements. However, you can disable constraint checking on existing data when you add a constraint to the table. Consider the following guidelines for disabling constraint checking on existing data:

•You can disable only CHECK and FOREIGN KEY constraints. Other constraints must be dropped and then added again.

•To disable constraint checking when you add a CHECK or FOREIGN KEY constraint to a table with existing data, include the WITH NOCHECK option in the ALTER TABLE statement.

•Use the WITH NOCHECK option if existing data will not change. Data must conform to CHECK constraints if the data is updated.

•Be certain that it is appropriate to disable constraint checking. You can execute a query to change existing data before you decide to add a constraint.

Disabling Constraint When Loading New Data

- Applies to CHECK and FOREIGN KEY Constraints
- Use When:
  - Data conforms to constraints
  - You load new data that does not conform to constraints

Example:

```
ALTER TABLE Emp
NOCHECK
CONSTRAINT FK_DeptNo
```


You can disable constraint checking on existing CHECK and FOREIGN KEY constraints so that any data that you modify or add to the table is not checked against the constraint.

To avoid the costs of constraint checking, you might want to disable constraints when:

•You already have ensured that the data conforms to the constraints.

•You want to load data that does not conform to the constraints. Later, you can execute queries to change the data and then re-enable the constraints.

Module 5: Modifying Data and Table Structures

- Overview
  - Inserting Data
  - Deleting Data
  - Updating Data
  - Using Transactions
  - Adding, Modifying and dropping a column
  - Altering and dropping a table

Inserting Data

- Inserting a row of data by values
- Inserting partial data
- Using the INSERT…SELECT statement
- Creating a table using the SELECT INTO statement

Inserting a Row of Data by Values

- Must adhere to destination constraints or the INSERT transaction fails
- Use a column list to specify destination columns
- Specify a corresponding list of values

Example:

```
INSERT Dept ( DeptNo,DeptName )
VALUES (1,'Import')
```


The INSERT statement adds rows to a table.

Syntax :

```
INSERT [INTO]
{ table_name | view_name}
{ [(column_list)] }
VALUES ( { DEFAULT | NULL| expression}[,.n])
```

Use the INSERT statement with the VALUES clause to add rows to a table. When you insert rows, consider the following facts and guidelines:

•Must adhere to destination constraints or the INSERT transaction fails.

•Use the *column_list* to specify columns that will store each incoming value. You must enclose the *column_list* in parentheses and delimit it by commas. If you are supplying values for all columns, using the *column_list* is optional.

•Specify the data that you want to insert by using the VALUES clause. The VALUES clause is required for each column in the table or *column_list.*

The column order and data type of new data must correspond to the table column order and data type. Many data types have an associated entry format. For example, character data and dates must be enclosed in single quotation marks.

Inserting Partial Data

- Adding new data

```
INSERT dept (deptno)VALUES (30)
```

- Verifying new data

```
SELECT *
FROM dept
WHERE deptno= 30
```

Takes 'null' value as implicit for column for which value is not given

DeptNo	DeptName
30	Null

Pragati Software Pvt. Ltd., 207, Lok Center, Marol-Maroshi Road, Marol, Andheri (East), Mumbai 400 059. www.pragatisoftware.com 45

If a column has a default value or accepts null values, you can omit the column from an INSERT statement. SQL Server automatically inserts the values.

When you insert partial data, consider the following facts and guidelines:

•List only the column names for the data that you are supplying in the INSERT statement.

•Specify the columns for which you are providing a value in the *column_list*. The data in the VALUES clause corresponds to the specified columns. Unnamed columns are filled in as if they had been named and a default value had been supplied.

•Do not specify columns in the *column_list* that have an IDENTITY property or that allow default or null values.

•Enter a null value explicitly by typing Null without single quotation marks.

Using the INSERT…SELECT Statement

- All rows that satisfy the SELECT statement are inserted
- Verify that the table that receives new row exists
- Ensure that data types are compatible

Example:

```
CREATE TABLE ShippedOrders
   (OrderID int,
    OrderDate datetime,
    ShippedDate datetime,
    ShipCountry varchar(20))

INSERT ShippedOrders
   SELECT OrderID , OrderDate , ShippedDate , ShipCountry FROM orders
   WHERE ShippedDate is not null
```


The INSERT.SELECT statement adds rows to a table by inserting the result set of a SELECT statement.

Using the INSERT.SELECT statement is more efficient than writing multiple, single-row INSERT statements. When you use the INSERT.SELECT statement, consider the following facts and guidelines:

•All rows that satisfy the SELECT statement are inserted into the outermost table of the query.

•You must verify that the table that receives the new rows exists in the database.

•You must ensure that the columns of the table that receives the new values have data types compatible with the columns of the table source.

•You must determine whether a default value exists or whether a null value is allowed for any columns that are omitted. If null values are not allowed, you must provide values for these columns.

Syntax :

```
INSERT table_name
            SELECT column_list
            FROM table_list
            WHERE search_conditions
```

Creating Table Using the SELECT INTO Statement

- Use to create a table and insert rows into the table in a single operation
- Create column alias or specify column names in the select list for new table

Example:

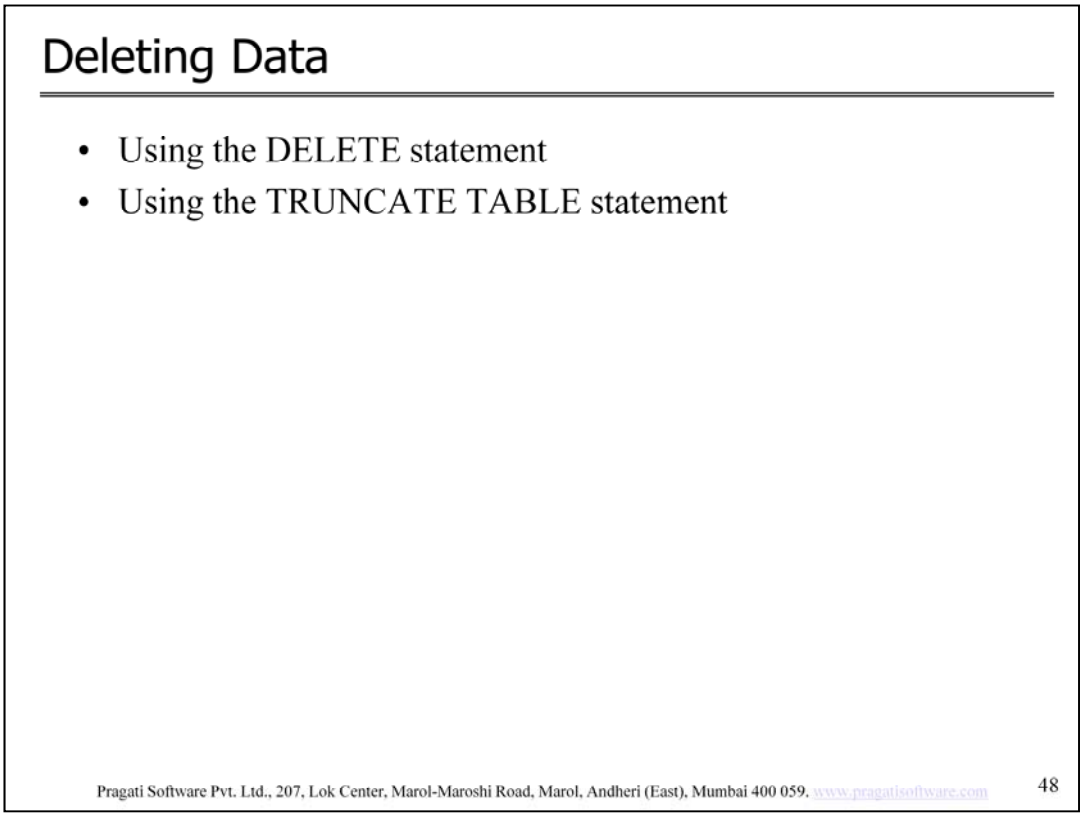
```
SELECT productname AS products,
       unitprice AS price,
       (unitprice * 1.1) AS tax
INTO pricetable
FROM products
```


You can place the result set of any query into a new table by using the SELECT INTO statement. Use the SELECT INTO statement to populate new tables in a database with imported data. You also can use the SELECT INTO statement to break down complex problems that require a data set from various sources. If you first create a temporary table, the queries that you execute on it are simpler than those you would execute on multiple tables or databases.

When you use the SELECT INTO statement, consider the following facts and guidelines:

•You can use the SELECT INTO statement to create a table and to insert rows into the table in a single operation. Ensure that the table name that is specified in the SELECT INTO statement is unique. If a table exists with the same name, the SELECT INTO statement fails.

•You must create column aliases or specify the column names of the new table in the select list.



You can specify the data that you want to delete.

The DELETE statement removes one or more rows from a table or view by using a transaction. You can specify which rows SQL Server deletes by filtering on the targeted table, or by using a JOIN clause or a subquery. The TRUNCATE TABLE statement is used to remove all rows from a table without using a transaction.

Using the DELETE Statement

- The DELETE statement removes one or more rows in a table according to the WHERE clause condition, if specified
- Each deleted row is logged in the transaction log

Example:

```
DELETE ShippedOrders
WHERE ShipCountry = 'USA'
```

```
DELETE ShippedOrders
```

Pragati Software Pvt. Ltd., 207, Lok Center, Marol-Maroshi Road, Marol, Andheri (East), Mumbai 400 059. www.pragatisoftware.com 49

The DELETE statement removes rows from tables. Use the DELETE statement to remove one or more rows from a table.

Syntax :

```
DELETE [from] {table_name|view_name}
WHERE search_conditions
```

When you use the DELETE statement, consider the following facts:

•SQL Server deletes all rows from a table unless you include a WHERE clause in the DELETE statement.

•Each deleted row is logged in the transaction log.

Using the TRUNCATE TABLE Statement

- The TRUNCATE TABLE statement deletes all rows in a table
- SQL server retains table structure and associated objects
- Only de-allocation of data pages is logged in the transaction log

Example:

TRUNCATE TABLE pricetable

The TRUNCATE TABLE removes all data from a table. Use the TRUNCATE TABLE statement to perform a non logged deletion of all rows.

Syntax :

```
TRUNCATE TABLE [[database.]owner.]table_name
```

When you use the TRUNCATE TABLE statement, consider the following facts:

•SQL Server deletes all rows but retains the table structure and its associated objects.

•The TRUNCATE TABLE statement executes more quickly than the DELETE statement because SQL Server logs only the de-allocation of data pages.

•If a table has an IDENTITY column, the TRUNCATE TABLE statement resets the seed value.

Updating Data

- Updating rows based on data in the table
  - WHERE clause specifies rows to change
  - SET keyword specifies the new data
  - Input values must have compatible data types with the columns
  - Updates do not occur in rows that violate any integrity constraints

Examples:

UPDATE dept SET dname='Services'
WHERE deptno = 30

UPDATE products **SET** unitprice = (unitprice * 1.1)

The UPDATE statement can change data values in single rows, groups of rows, or all rows in a table or view. You can update a table based on data in the table or on data in other tables.

The UPDATE statement modifies existing data.

Syntax :

```
UPDATE {table_name | view_name}
SET {  column_name = {expression | DEFAULT | NULL}  |
@variable=expression}[,.n]
WHERE {search_conditions}
```

Use the UPDATE statement to change single rows, groups of rows, or all of the rows in a table. When you update rows, consider the following facts and guidelines:

•Specify the rows to update with the WHERE clause.

•Specify the new values with the SET clause.

•Verify that the input values are the same as the data types that are defined for the columns.

•SQL Server does not update rows that violate any integrity constraints. The changes do not occur, and the statement is rolled back.

•You can change the data in only one table at a time.

•You can set one or more columns or variables to an expression. For example, an expression can be a calculation (like **price** * 2) or the addition of two columns.

Adding, Modifying and Dropping a Column

- Adding a column:
  ALTER TABLE Emp **ADD** Commission money
- Modifying a column:
  ALTER TABLE Emp **ALTER COLUMN** empAddress varchar(50)
- Dropping a column
  ALTER TABLE Emp **DROP COLUMN** Commission

Adding and dropping columns are two ways to modify tables.

Syntax :

```
ALTER TABLE table
  {| [ALTER COLUMN column_name ] |{  ADD
      {  <column_definition> ::=
            column_name data_type
            {  [NULL | NOT NULL]
      | DROP column column_name} [,.n]
```

Adding a Column

The type of information that you specify when you add a column is similar to that which you supply when you create a table. This example adds a column that allows null values.

Example

```
ALTER TABLE CategoriesNew
ADD Commission money null
```

Dropping a Column

Dropped columns are unrecoverable. Therefore, be certain that you want to remove a column before doing so. This example drops a column from a table.

Example

```
ALTER TABLE CategoriesNew
DROP COLUMN Sales_date
```

All indexes and constraints that are based on a column must be removed before you drop the column.

Altering and dropping a table

- The ALTER TABLE command is used to modify the structure of an existing table

```
ALTER TABLE Emp
  ADD DateOfBirth datetime
```

- Dropping a table removes the table definition and all data, as well as the permission specifications

```
DROP TABLE pricetable
```

Pragati Software Pvt. Ltd., 207, Lok Center, Marol-Maroshi Road, Marol, Andheri (East), Mumbai 400 059. www.pragatisoftware.com 53

Modifying a Table Structure

The ALTER TABLE command is used to modify the structure of an existing table:

•modifying the data type of an existing column.

Dropping a Table

Dropping a table removes the table definition and all data, as well as the permission specifications for that table. Before you can drop a table, you should remove any dependencies between the table and other objects. To view existing dependencies, execute the **sp_depends** system stored procedure.

Syntax :

```
DROP TABLE table_name
```

Module 6: Built-in Functions

- Overview:
  - Numeric functions
  - Character functions
  - Date functions
  - Conversion functions

Functions on Numeric data types

Function	Returns	Example	Result
floor (n)	Largest integer equal to or less than n.	SELECT floor (9.86)	9
power (m, n)	Number m raised to the power of n.	SELECT power (5, 2)	25
round (n, m)	Number n rounded off to m decimal places.	SELECT round (9.86, 1)	9.90
sign (n)	If n = 0, returns 0. If n > 0, returns 1. If n < 0, returns -1.	SELECT sign (9.86)	1.00
sqrt (n)	Square root of n.	SELECT sqrt (25)	5
abs(n)	Returns an absolute value	SELECT abs(-1)	1
rand([seed])	Returns a random float no. between 0 and 1	SELECT rand()	0.90831884935795

Common functions on numeric data types

Some common functions on numeric data types are given in the above slide with explanation and example.

Functions on Character data type

Function	Returns	Example	Result
ascii	ASCII code of leftmost character	SELECT ascii('ABC')	65
lower (char)	Converts the entire string to lowercase.	SELECT lower ('Inder Kumar Gujral')	inder kumar gujral
upper (char)	Converts the entire string to uppercase.	SELECT upper ('Inder Kumar Gujral')	INDER KUMAR GUJRAL
ltrim(char)	Removes leading spaces	SELECT ltrim('RICHARD')	RICHARD without leading spaces
rtrim(char)	Removes trailing spaces	SELECT rtrim('RICHARD ')	RICHARD without trailing spaces
left(char,m)	Part of char starting from leftmost character upto m characters	SELECT left('RICHARD',4)	RICH
right(char,m)	Part of char starting from rightmost character upto m characters	SELECT right('RICHARD',4)	HARD
substring (char, m, n)	Part of char, starting FROM position m and taking characters upto n.	SELECT substring ('Computer', 1, 4)	Comp
len (char)	Length of char.	SELECT len('RICHARD')	7

Common functions on character data types

Some common functions on character data types are given in the above slide with explanation and example.

Functions on Date data types

Function	Returns	Example	Result
getdate()	Current date and time	SELECT getdate()	2008-11-05 11:41:25.377
dateadd(datepart, number,date)	Adds no. of dateparts to the date	SELECT dateadd(mm, 1, getDate())	2008-12-05 11:46:11.523
datediff(datepart, date1,date2)	Calculates the no. of dateparts between the two dates	SELECT datediff(yy, '11/18/2000',getDate())	8
datepart(datepart, date)	Returns datepart as numeric value	SELECT datepart(dd, getdate())	5
datename(datepart, date)	Returns datepart as character value	SELECT datename(dw, getdate())	Wednesday

Common functions on date data types

Some common functions on date data types are given in the above slide with explanation and example.

Functions on Date data types

Where date part can be:

Datepart	Abbreviation	Values
year	yy, yyyy	1753-9999
quarter	qq, q	1-4
month	mm,m	1-12
day of year	dy,y	1-366
day	dd,d	1-31
week	wk ,ww	0-51
weekday	dw	1-7(1 is Sunday)
hour	hh	(0-23)
minute	mi, n	(0-59)
second	ss, s	0-59
millisecond	ms	0-999

The date component called date part is used in conjunction with the date functions to specify an element of a date value for parsing or date arithmetic

Conversion Functions

- CAST and CONVERT:
  - Explicitly converts an expression of one data type to another.
- Syntax using CAST:
  - CAST(expression AS data_type)
- Syntax using CONVERT:
  - CONVERT (data_type[(length)], expression[,style])
    - Some style values are 0,1,2,3,4,5,100,101,102,103,104,105

Examples:

- SELECT **CAST**('02/07/09' **AS** datetime)
- SELECT **CONVERT**(char(10), getdate(),2)

Common functions on datatype conversion

CAST and CONVERT:

Explicitly converts an expression of one data type to another.

Style values used with convert:

Value	Output
0 or 100	mon dd yyyy hh:mm (AM or PM)
1	mm/dd/yy
2	yy.mm.dd
3	dd/mm/yy
4	dd.mm.yy
5	dd-mm-yy
101	mm/dd/yyyy
102	yyyy.mm.dd
103	dd/mm/yyyy
104	dd.mm.yyyy
105	dd-mm-yyyy

Module 7: Additional Create Table Features

- Overview:
  - Using Identity to generate sequential numbers
  - Creating user defined data types
  - Using Defaults and Rules
  - Dropping user defined data types
  - Temporary Tables
    - Private Temporary Tables
    - Global Temporary Tables

Using Identity to generate sequential numbers

- Used to generate unique numeric values.
- A *column property* for whole-number datatype only.
- A table can have only one column with the Identity property.
- Specify the starting number (seed) default 1 and increment/decrement value default +1 with identity.

```
CREATE TABLE customer(
cust_id     int IDENTITY  NOT NULL,
cust_name   varchar(50)    NOT NULL)
```

- To refer to a column that has Identity property set –

  IDENTITYCOL

```
SELECT IDENTITYCOL FROM customer
SELECT cust_id FROM customer

INSERT customer VALUES ('ACME Widgets')
INSERT customer (cust_name) VALUES ('Pragati Software')
```


It is common to provide simple counter-type values for tables that don't have a natural or efficient primarykey. Columns such as*customer_number* are usually simple counter fields. SQL Server provides the Identity property that makes generating unique numeric values easy. Identity isn't a datatype; it's a *columnproperty* that you can declare on a whole-number datatype such as*tinyint*, *smallint*, *int*, and *numeric/decimal* (having a scale of zero). Each table can have only one column with the Identity property.

The table's creator can specify the starting number (seed) and the amount that value increments or decrements. If not otherwise specified, the seed value starts at 1 and increments by 1, as shown in this example:

```
CREATE TABLE customer( cust_id smallint, IDENTITY  NOT NULL, cust_name varchar(50)  NOT NULL,)
```

To find out which seed and increment values were defined for a table, you can use the IDENT_SEED(*tablename*) and IDENT_INCR(*tablename*) functions. The statement SELECT IDENT_SEED('CUSTOMER'), IDENT_INCR('CUSTOMER') for the *customer* table because values weren't explicitly declared and the default values were used. This next example explicitly starts the numbering at 100 (seed) and increments the value by 20.

```
CREATE TABLE customer( cust_id       smallint, IDENTITY(100,20)  NULL,
cust_name    varchar(50)     NOT NULL)
```

The value automatically produced with the Identity property will normally be unique, but it isn't guaranteed by the Identity property itself, nor is it guaranteed to be consecutive. For efficiency, a value is considered used as soon as it is presented to a client doing an INSERT operation. If that client doesn't ultimately commit the INSERT, the value will never appear, so a break will occur in the consecutive numbers. An unacceptable level of serialization would exist if the next number couldn't be parceled out until the previous one was actually committed or rolled back. (And even then, as soon as a row was deleted, the values would no longer be consecutive. Gaps are inevitable.)

Using Identity to generate sequential numbers

- To find out seed value for a table
  - IDENT_SEED(tablename)
- To find out increment value for a table
  - IDENT_INCR(tablename).

```
SELECT IDENT_SEED('CUSTOMER'), IDENT_INCR('CUSTOMER')
```

- To get the last identity value used by connection –

```
SELECT @@IDENTITY
```

- To temporarily disable the generation of values in an *identity*column-

```
SET IDENTITY_INSERT tablename ON.

SET IDENTITY_INSERT CUSTOMER ON

insert into customer (cust_id,cust_name) values (100,'New Foods')

SET IDENTITY_INSERT CUSTOMER OFF
```


To temporarily disable the automatic generation of values in an *identity* column, use the SET IDENTITY_INSERT *tablename* ON option. In addition to filling in gaps in the identity sequence, this option is useful for tasks such as bulk loading data in which the previous values already exist. For example, perhaps you're loading a new database with customer data from your previous system. You might want to preserve the previous customer numbers but have new ones automatically assigned using Identity. The SET option was created exactly for cases like this. Because of the SET option's ability to override values, the Identity property alone doesn't enforce uniqueness of a value within the table. Although Identity will generate a unique number, it can be overridden with the SET option. To enforce uniqueness (which you'll almost always want to do when using Identity), you should also declare a UNIQUE or PRIMARY KEY constraint on the column. If you insert your own values for an *identity* column (using SET IDENTITY_INSERT), when automatic generation resumes, the next value will be the next incremented value (or decremented value) of the highest value that exists in the table, whether it was generated previously or explicitly inserted.

The keyword IDENTITYCOL automatically refers to the specific column in a table, whatever its name, that has the Identity property. If *cust_id*is that column, you can refer to the column as IDENTITYCOL without knowing or using the column name, or you can refer to it explicitly as *cust_id*. For example, the following two statements work identically and return the same data:

SELECT IDENTITYCOL FROM customer

SELECT cust_id FROM customer

The column name returned to the caller is *cust_id*, not IDENTITYCOL, in both of these cases. When inserting rows, you must omit an *identity* column from the column list and VALUES section. (The only exception is when the IDENTITY_INSERT option is on.) If you do supply a column list, you must omit the column for which the value will be automatically supplied. Here are two valid INSERT statements for the *customer* table shown earlier:

INSERT customer VALUES ('ACME Widgets')

INSERT customer (cust_name) VALUES ('AAA Gadgets')

Selecting these two rows produces this output:

cust_id	cust_name
1	ACME Widgets
2	AAA Gadgets

Creating user defined data types

- To remove the inconsistency in table structures which arises when two attributes that should have the same datatype use different system datatypes.
- Is specific to the database in which it is created. If created in MODEL database, included in all newly created databases.
- Are constructed using system datatype

Syntax:

sp_addtype name, system_data_type[,'null_type']

Example:

exec sp_addtype phone_number,'varchar(20)','not null'

For example, your database stores various phone numbers in many tables. Although no single, definitive way exists to store phone numbers, in this database consistency is important. You can create a *phone_number* UDDT and use it consistently for any column in any table that keeps track of phone numbers to ensure that they all use the same datatype.

To create a user-defined datatype, we make use of the sp_addtype system stored procedure. This procedure creates a user-defined datatype by adding a descriptive record to the systypes system table. Here's how to create this UDDT:

```
exec spaddtype  phone_number,'varchar(20)','not null'
```

And here's how to use the new UDDT when creating a table:

```
CREATE TABLE customer2(
cust_id       smallint       NOT NULL,
cust_name     varchar(50)    NOT NULL,
cust_addr1    varchar(50)    NOT NULL,
cust_addr2    varchar(50)    NOT NULL,
cust_city     varchar(50)    NOT NULL,
cust_state    char(2)        NOT NULL,
cust_zip      varchar(10)    NOT NULL,
cust_phone    phone_number,
cust_fax      varchar(20)    NOT NULL,
cust_email    varchar(30)    NOT NULL,
cust_web_url  varchar(20)    NOT NULL)
```

When the table is created, internally the datatype of*cust_phone*is known to be*varchar(20)*. Notice that both *cust_phone* and *cust_fax* are*varchar(20)*, although *cust_phone* has that declaration through its definition as a UDDT.

Using Defaults and Rules

- As Independent Objects They:
  - Are defined once
  - Can be bound to one or more columns or user-defined data types
- Rules specify the acceptable values to insert into a column
  - A rule definition can contain any expression that is valid in a WHERE clause.
  - A column or user-defined data type can have only one rule that is bound to it.

```
CREATE RULE regioncode_rule AS @regioncode IN ('IA', 'IL', 'KS', 'MO')

EXEC sp_bindrule regioncode_rule, 'Customers.Region'
```

- To detach a rule
  - Execute the **sp_unbindrule** system stored procedure.

Defaults and rules are objects that can be bound to one or more columns or user-defined data types, making it possible to define them once and use them repeatedly. A disadvantage to using defaults and rules is that they are not ANSI-compliant.

Creating a Rule

Rules specify the acceptable values that you can insert into a column. They ensure that data falls within a specified range of values, matches a particular pattern, or matches entries in a specified list. Consider these facts about rules:

•A rule definition can contain any expression that is valid in a WHERE clause.

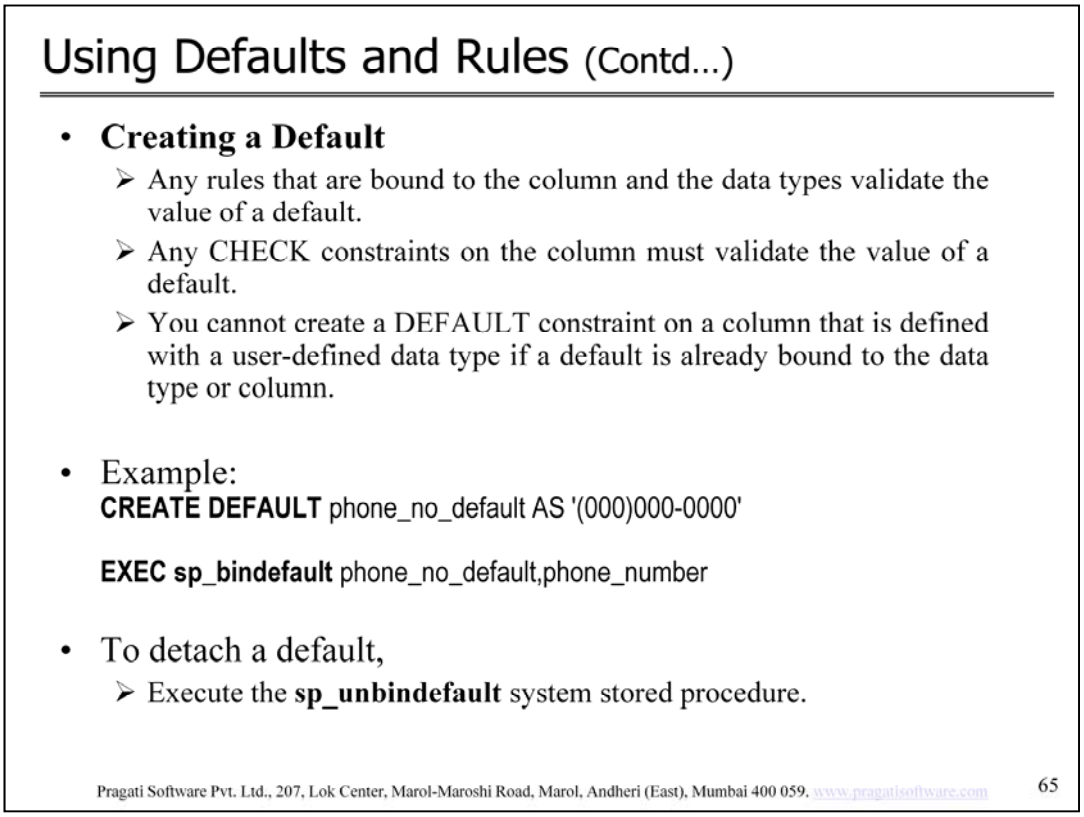
•A column or user-defined data type can have only one rule that is bound to it.

Syntax :

```
CREATE RULE rule
AS condition_expression
```

Binding a Rule

After you create a rule, you must bind it to a column or user-defined data type by executing the **sp_bindrule** system stored procedure. To detach a rule, execute the **sp_unbindrule** system stored procedure.



Creating a Default

If a value is not specified when you insert data, a default specifies one for the column to which the object is bound. Consider these facts before you create defaults:

•Any rules that are bound to the column and the data types validate the value of a default.

•Any CHECK constraints on the column must validate the value of a default.

•You cannot create a DEFAULT constraint on a column that is defined with a user-defined data type if a default is already bound to the data type or column.

Syntax :

```
CREATE DEFAULT default
AS constant_expression
```

Binding a Default

After you create a default, you must bind it to a column or user-defined data type by executing the **sp_bindefault** system stored procedure. To detach a default, execute the **sp_unbindefault** system stored procedure.

Dropping user defined data types

Can be dropped with the following syntax:-

sp_droptype type

Example

sp_droptype phone_number

Cannot be dropped if other database object reference it.

Dropping a user-defined datatpe is an even easier command than the one that creates it. Just follow this

Syntax:

```
sp_droptype user_datatype_name
```

The only restriction on using sp_droptype is that a user-defined datatype cannot be dropped if a table or other database object references it. You must first drop any database objects that reference the user-defined datatype before dropping the user datatype itself. You can see that both the cust_phone and cust_fax columns have the same datatype, although the cust_phone column shows that the datatype is a UDDT. The type is resolved when the table is created, and the UDDT can't be dropped or changed as long as one or more tables are currently using it. Once declared, a UDDT is static and immutable, so no inherent performance penalty occurs in using a UDDT instead of the native datatype.

Temporary Tables

- Temporary workspace for intermediate data processing or to share work-in-progress with other connections.
- Can be created from any database, but exist only in *tempdb* database, which is cleared when the server is restarted
- All constraints except FOREIGN KEY constraints work
- Ways to use temporary tables:
  - private
  - global

Temporary Tables (Contd...)

Private Temporary Tables (#)

- Created by prefixing table name with a single hash sign (#)
- Only the connection that created the table can access the table
- Privileges can't be granted to another connection.
- Exists only for the life of creator connection
- Creator connection can drop the table using DROP TABLE
- Won't encounter a name collision if you choose a table name that's used in another connection.

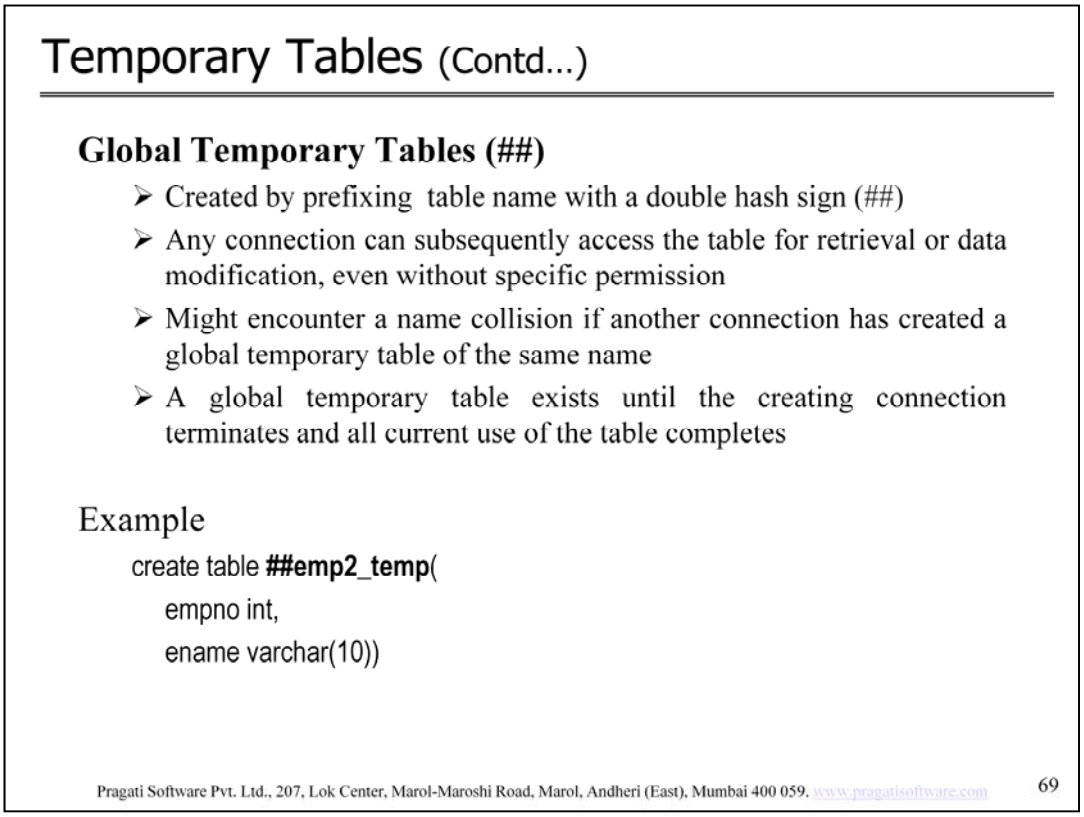
Example

```
create table #emp_temp(
    empno int,
    ename varchar(10))
```


Private Temporary Tables (#)

By prefixing a table name with a single hash sign (#)—as in CREATE TABLE #my_table—you can create it (from within any database) as a private temporary table. Only the connection that created the table can access the table, making it truly private. Privileges can't be granted to another connection. As a temporary table, it exists for the life of that connection only; that connection can drop the table using DROP TABLE. Because the scoping of a private temporary table is specific to the connection that created it, you won't encounter a name collision should you choose a table name that's used in another connection

```
Create table #emp_temp(
        empno int,
        ename varchar(10)
)
```



Global Temporary Tables (##)

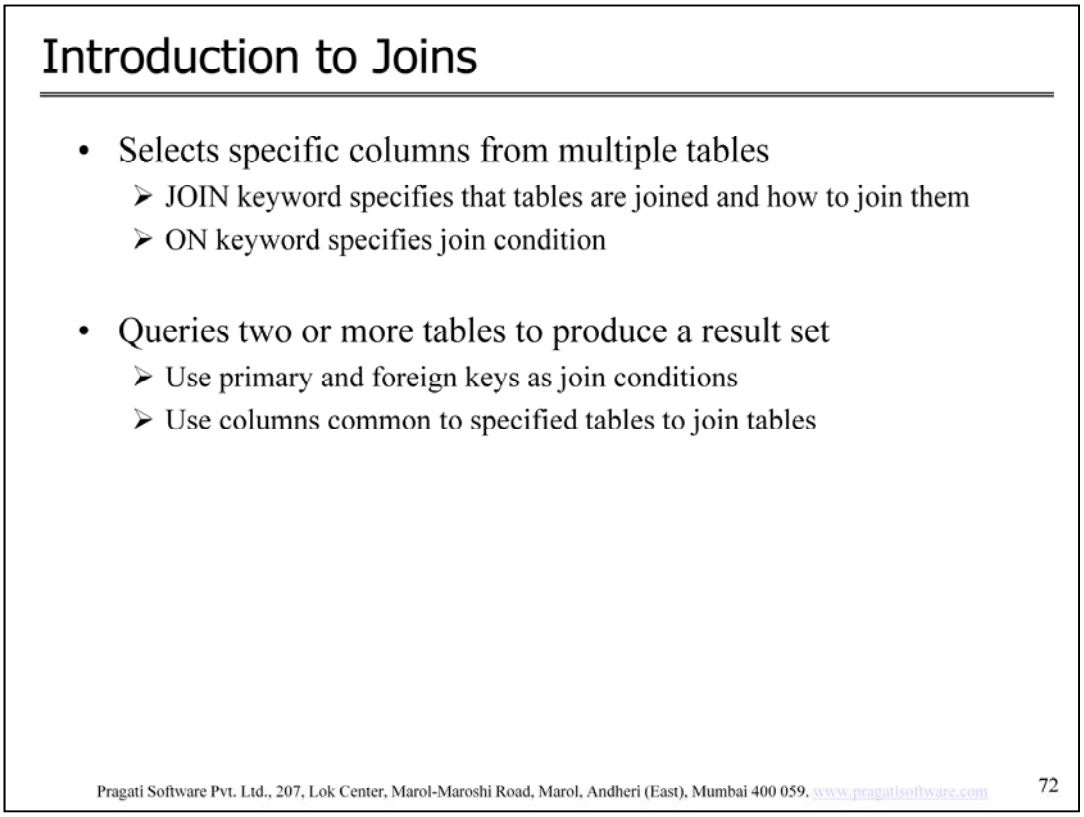
By prefixing a table name with double hash signs (##)—as in CREATE TABLE*## our_table*—you can create a global temporary table (from within any database and any connection). Any connection can subsequently access the table for retrieval or data modification, even without specific permission. Unlike with private temporary tables, all connections can use the single copy of a global temporary table. Therefore, you might encounter a name collision if another connection has created a global temporary table of the same name, and the CREATE TABLE statement will fail. A global temporary table exists until the creating connection terminates and all current use of the table completes. After the creating connection terminates, however, only those connections already accessing it are allowed to finish, and no further use of the table is allowed. If you want a global temporary table to exist permanently, you can create the table in a stored procedure that's marked to auto start whenever SQL Server is started.

Module 8: Joining Multiple Tables

- Overview
  - Using Aliases for Table Names
  - Combining Data from Multiple Tables
  - Combining Multiple Result Sets

Combining Data from Multiple Tables

- Introduction to Joins
- Using Inner Joins
- Using Outer Joins
- Using Cross Joins
- Joining More Than Two Tables
- Joining a Table to Itself



Selects Specific Columns from Multiple Tables

A join allows you to select columns from multiple tables by expanding on the FROM clause of the SELECT statement. Two additional keywords are included in the FROM clause.JOIN and ON:

•The JOIN keyword specifies which tables are to be joined and how to join them.

•The ON keyword specifies which columns the tables have in common.

Queries Two or More Tables to Produce a Result Set. A join allows you to query two or more tables to produce a single result set. When you implement joins, consider the following facts and guidelines:

•Specify the join condition based on the primary and foreign keys.

•If a table has a composite primary key, you must reference the entire key in the ON clause when you join tables.

•Use columns common to the specified tables to join the tables. These columns should have the same or similar data types.

•Reference a table name if the column names of the joined tables are the same. Qualify each column name by using the table_name. column_name format.

•Limit the number of tables in a join because the more tables that you join, the longer SQL Server takes to process your query.

•You can include a series of joins within a SELECT statement.

Using Inner Joins

- Inner joins combine tables by comparing values in columns that are common to both tables.
- Returns only rows that match the join conditions.

Examples:

```
SELECT DISTINCT companyname, orderdate
FROM orders INNER JOIN customers
ON orders.customerid = customers.customerid
WHERE orderdate > '1/1/98'
```


Why to Use Inner Joins

Use inner joins to obtain information from two separate tables and combine that information in one result set. When you use inner joins, consider the following

facts and guidelines:

•Inner joins are the SQL Server default. You can abbreviate the INNER JOIN clause to JOIN.

•Specify the columns that you want to display in your result set by including the qualified column names in the select list.

•Include a WHERE clause to restrict the rows that are returned in the result set.

•Do not use a null value as a join condition because null values do not evaluate equally with one another.

Using Outer Joins

- Combine rows from two tables that match the join condition, plus any unmatched rows of either the left or right table as specified in the JOIN clause.
- Rows that do not match the join condition display NULL in the result set.

```
insert into orders (orderdate) values (getdate())
```

- Left outer join displays all rows from the first-named table (the table on the left of the expression).

```
SELECT companyname, customers.customerid, orderdate
FROM customers  LEFT OUTER JOIN orders
ON customers.customerid = orders.customerid
```


Use left or right outer joins when you require a complete list of data that is stored in one of the joined tables in addition to the information that matches the join condition. When you use left or right outer joins, consider the following facts and guidelines:

! SQL Server returns only unique rows when you use left or right outer joins.

! Use a left outer join to display all rows from the first-named table (the table on the left of the expression). If you reverse the order in which the tables are listed in the FROM clause, the statement yields the same result as a right outer join.

Using Outer Joins

- Right outer join displays all rows from the second-named table (the table on the left of the expression).

```
SELECT companyname, customers.customerid, orderdate
FROM customers  RIGHT OUTER JOIN orders
ON customers.customerid = orders.customerid
```

- You also can use full outer joins to display all rows in the joined tables, regardless of whether the tables have any matching values.

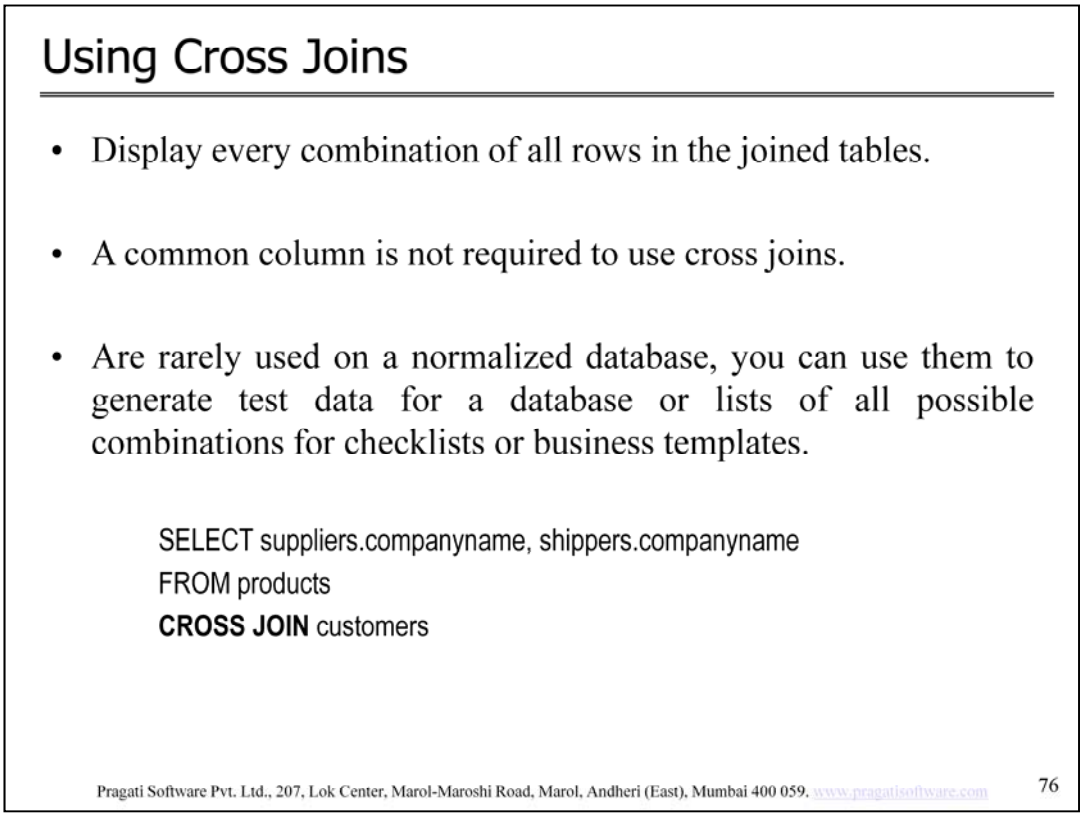
```
SELECT companyname, customers.customerid, orderdate
FROM customers  FULL OUTER JOIN orders
ON customers.customerid = orders.customerid
```

- You can abbreviate LEFT OUTER JOIN or RIGHT OUTER JOIN or FULL OUTER JOIN as LEFT JOIN or RIGHT JOIN or FULL JOIN

! Use a right outer join to display all rows from the second-named table (the table on the right of the expression). If you reverse the order in which the tables are listed in the FROM clause, the statement yields the same result as a left outer join.

! You can abbreviate the LEFT OUTER JOIN or RIGHT OUTER JOIN clause as LEFT JOIN or RIGHT JOIN.

! You can use outer joins between two tables only.



Why to Use Cross Joins?

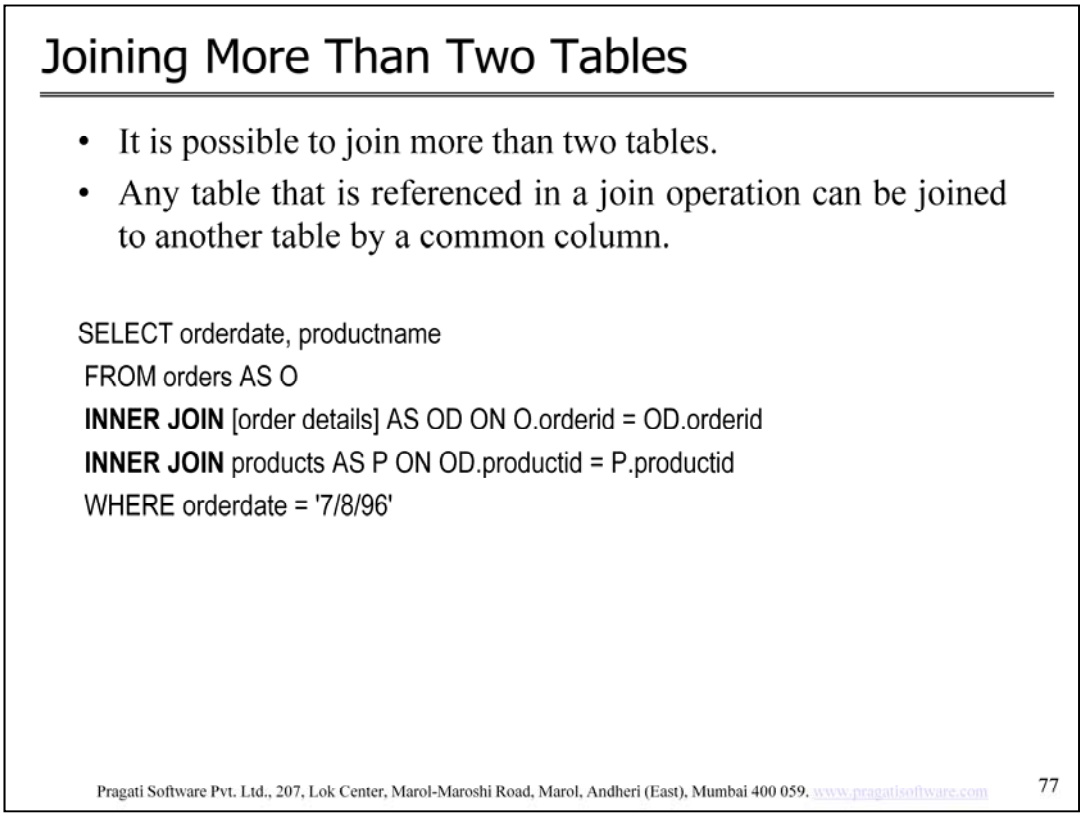
While cross joins are rarely used on a normalized database, you can use them to generate test data for a database or lists of all possible combinations for checklists or business templates. When you use cross joins, SQL Server produces a Cartesian product in which the number of rows in the result set is equal to the number of rows in the first table, multiplied by the number of rows in the second table.

This example displays a cross join between the **products** and **customers** tables that is useful for listing all of the possible combinations of customers and products. The use of a cross join displays all possible row combinations between these two tables. The **products** table has 77 rows, and the **customers** table has 91 rows. The result set contains 1077 rows.

SELECT suppliers.companyname, shippers.companyname

FROM products

CROSS JOIN customers



It is possible to join any number of tables. Any table that is referenced in a join operation can be joined to another table by a common column.

Why to JOIN more than two tables?

Use multiple joins to obtain related information from multiple tables. When you join more than two tables, consider the following facts and guidelines:

•You must have one or more tables with foreign key relationships to each of the tables that you want to join.

•You must have a JOIN clause for each column that is part of a composite key.

•Include a WHERE clause to limit the number of rows that are returned.

This example displays information from the **orders** and **products** tables by using the **order details** table as a link. For example, if you want a list of products that are ordered each day, you need information from both the **orders** and **products** tables. An order can consist of many products, and a product can have many orders. To retrieve information from both the **orders** and **products** tables, you can use an inner join through the **order details** table. Even though you are not retrieving any columns from the **order details** table, you must include this table as part of the inner join in order to relate the **orders** table to the **products** table. In this example, the **orderid** column is common to both the **orders** and **order details** tables, and the **productid** column is common to both the **order details** and **products** tables.

```
SELECT orderdate, productname
FROM orders AS O
INNER JOIN [order details] AS OD ON O.orderid = OD.orderid
INNER JOIN products AS P ON OD.productid = P.productid
WHERE orderdate = '7/8/96'
```

Joining a Table to Itself

- If you want to find rows that have values in common with other rows in the same table, you can use a self-join to join a table to another instance of itself.

```
SELECT a.employeeid, a.lastname AS name, a.title AS title, b.employeeid,
b.lastname AS name, b.title AS title  FROM
employees AS a LEFT JOIN employees AS b
ON a.reportsto = b.employeeid
```


Why to use SELF JOINs?

While self-joins rarely are used on a normalized database, you can use them to reduce the number of queries that you execute when you compare values of different columns of the same table. When you use self-joins, consider the following guidelines:

•You must specify table aliases to reference two copies of the table. Remember that table aliases are different from column aliases. Table aliases are designated as the table name followed by the alias.

•When you create self-joins, each row matches itself and pairs are repeated, resulting in duplicate rows. Use a WHERE clause to eliminate these duplicate rows.

Combining Multiple Result Sets

- Each Query Must Have:
  - Similar data types
  - Same number of columns
  - Same column order in select list
- To define new column headings for the result set, you must create the column aliases in the first SELECT statement.
- Use the UNION operator to create a single result set from multiple queries. Removes duplicate rows in the result set.

```
SELECT (firstname + ' ' + lastname) AS name,city, postalcode FROM employees
UNION SELECT companyname, city, postalcode FROM customers
```

- UNION ALL includes duplicates in the result set.

```
SELECT city FROM employees
UNION ALL SELECT city FROM customers
```


To combine multiple resultsets,

•SQL Server requires that the referenced tables have similar data types, the same number of columns, and the same column order in the select list of each query.

•You must specify the column names in the first SELECT statement. Therefore, if you want to define new column headings for the result set, you must create the column aliases in the first SELECT statement.

The UNION operator combines the result of two or more SELECT statements into a single result set. SQL Server removes duplicate rows in the result set. However, if you use the ALL option, all rows (including duplicates) are included in the result set.

Combining Multiple Result Sets

- INTERSECT operator returns common values of the resultsets of both the queries on the left and right sides of the INTERSECT operand

```
SELECT city FROM employees
INTERSECT SELECT city FROM customers
```

- EXCEPT operator returns any distinct values from the left query that are not also found on the right query.

```
SELECT city FROM employees
EXCEPT SELECT city FROM customers

SELECT city FROM customers
EXCEPT SELECT city FROM employees
```


INTERSECT operator returns common values of the resultsets of both the queries on the left and right sides of the INTERSECT operand

EXCEPT operator returns any distinct values from the left query that are not also found on the right query.

Module 9: Working with Subqueries

- Overview
  - Introduction to Subqueries
  - Using a Subquery as a Derived Table
  - Using a Subquery as an Expression
  - Correlated Subquery
  - Using the EXISTS and NOT EXISTS Clauses
  - Deleting rows based on other tables
  - Updating rows based on other tables

Introduction to Subqueries

- Use Subqueries
  - To break down a complex query into a series of logical steps
  - To answer a query that relies on the results of an other query
- Why to Use Joins Rather Than Subqueries
  - Joins are executed faster than subqueries
- How to Use Subqueries
  - Enclose subqueries in parentheses.
  - Use a subquery in place of an expression as long as a single value or list of values is returned.
  - Use a subquery that returns a multi-column record set in place of a table or to perform the same function as a join.

A *subquery* is a SELECT statement nested inside a SELECT, INSERT, UPDATE, or DELETE statement or inside another subquery. Often you can rewrite subqueries as joins and use subqueries in place of an expression. An *expression* is a combination of identifiers, values, and operators that SQL Server evaluates to obtain a result.

Why to use subqueries?

You use subqueries to break down a complex query into a series of logical steps and, as a result, to solve a problem with a single statement. Subqueries are useful when your query relies on the results of another query.

Why to use joins rather than subqueries?

Often, a query that contains subqueries can be written as a join. Query performance may be similar with a join and a subquery. The query optimizer usually optimizes subqueries so that it uses the sample execution plan that a semantically equivalent join would use. The difference is that a subquery may require the query optimizer to perform additional steps, such as sorting, which may influence the processing strategy.

Using joins typically allows the query optimizer to retrieve data in the most efficient way. If a query does not require multiple steps, it may not be necessary to use a subquery.

How to use subqueries?

When you decide to use subqueries, consider the following facts and guidelines:

•You must enclose subqueries in parentheses.

•You can use a subquery in place of an expression as long as a single value or list of values is returned. You can use a subquery that returns a multi-column record set in place of a table or to perform the same function as a join.

Using a Subquery as a Derived Table

- Is a recordset within a query that functions as a table
- Takes the place of a table in the FROM clause
- Is optimized with the rest of the query

Example:

```
SELECT T.orderdate, T.productname ,contactname from
(SELECT orderdate, productname ,customerid
 FROM orders AS O
 INNER JOIN [order details] AS OD ON O.orderid = OD.orderid
 INNER JOIN products AS P ON OD.productid = P.productid
 WHERE orderdate = '7/8/96' ) AS T
 JOIN customers c ON T.customerid=c.customerid
```


You create a derived table by using a subquery in place of a table in a FROM clause. A derived table is a special use of a subquery in a FROM clause to which an alias or user-specified name refers. The result set of the subquery in the FROM clause forms a table that the outer SELECT statement uses.

This example uses a subquery to create a derived table in the inner part of the query that the outer part queries. The derived table itself is functionally equivalent to the whole query, but it is separated for illustrative purposes.

```
SELECT T.orderdate, T.productname ,contactname from
(SELECT orderdate, productname ,customerid
 FROM orders AS O
 INNER JOIN [order details] AS OD ON O.orderid = OD.orderid
 INNER JOIN products AS P ON OD.productid = P.productid
 WHERE orderdate = '7/8/96' ) AS T
 JOIN customers c ON T.customerid=c.customerid
```

When used as a derived table, consider that a subquery:

•Is a recordset within a query that functions as a table.

•Takes the place of a table in the FROM clause.

•Is optimized with the rest of the query.

Using a Subquery as an Expression

- Is evaluated and treated as an expression
- Is executed once for the query

Example:

```
SELECT productName,UnitPrice,
(
        SELECT avg(unitprice)
        FROM products
) Average,
UnitPrice - (
        SELECT avg(unitprice)
        FROM products
) Difference
FROM products
```


Transact-SQL, you can substitute a subquery wherever you use an expression. The subquery must evaluate to a scalar value, or to a single column list of values. Subqueries that return a list of values replace an expression in a WHERE clause that contains the IN keyword.

When used as an expression, consider that a subquery:

•Is evaluated and treated as an expression. The query optimizer often evaluates an expression as equivalent to a join connecting to a table that has one row.

•Is executed once for the entire statement.

Correlated Subquery

To list Orders where product ordered is less then 10% of the average order:

```
select distinct OrderId
from [Order Details] OD
where Quantity < (select avg(Quantity) * .1
                  from [Order Details]
                  where OD.ProductID = ProductID)
```

Steps of evaluating a correlated subquery

1. Outer query passes column values to the inner query
2. Inner query uses that value to satisfy the inner query
3. Inner query returns a value back to the outer query
4. The process is repeated for the next row of the outer query
5. Back to Step 1

When you create a correlated subquery, the inner subqueries are evaluated repeatedly, once for each row of the outer query:

•SQL Server executes the inner query for each row that the outer query selects.

•SQL Server compares the results of the subquery to the results outside the subquery.

This example returns a list of customers who ordered more than 20 pieces of product number 23.

```
SELECT orderid, customerid  FROM orders AS or1
 WHERE 20 < (SELECT quantity
             FROM [order details] AS od
             WHERE or1.orderid = od.orderid AND od.productid = 23)
```

Correlated subqueries return a single value or a list of values for each row specified by the FROM clause of the outer query. The following steps describe how the correlated subquery is evaluated in example 1:

1. The outer query passes a column value to the inner query. The column value that the outer query passes to the inner query is the **orderid**. The outer query passes the first **orderid** in the **orders** table to the inner query.

2. The inner query uses the values that the outer query passes. Each **orderid** in the **orders** table is evaluated to determine whether an identical **orderid** is found in the **order details** table. If the first **orderid** matches an **orderid** in the **order details** table and that **orderid** purchased product number 23, then the inner query returns that **orderid** to the outer query.

3. The inner query returns a value back to the outer query. The WHERE clause of the outer query further evaluates the **orderid** that purchased product number 23 to determine whether the quantity ordered exceeds 20.

4. The process is repeated for the next row of the outer query. The outer query passes the second **orderid** in the **orders** table to the inner query, and SQL Server repeats the evaluation process for that row.

Using the EXISTS and NOT EXISTS Clauses

- Use with correlated subqueries
- Determine whether data exists in a list of values
- SQL server process
  - Outer query tests for the existence of rows
  - Inner query returns TRUE or FALSE
  - No data is produced

Examples:

```
SELECT lastname, employeeid FROM employees AS e
WHERE EXISTS (
        SELECT *
        FROM orders AS o
        WHERE e.employeeid = o.employeeid  AND  o.orderdate = '9/5/97'
)
```

Pragati Software Pvt. Ltd., 207, Lok Center, Marol-Maroshi Road, Marol, Andheri (East), Mumbai 400 059. www.pragatisoftware.com 86

You can use the EXISTS and NOT EXISTS operators to determine whether data exists in a list of values.

Use with correlated subqueries

Use the EXISTS and NOT EXISTS operators with correlated subqueries to restrict the result set of an outer query to rows that satisfy the subquery. The EXISTS and NOT EXISTS operators return TRUE or FALSE, based on whether rows are returned for subqueries.

Determine whether data exists in a list of values

When a subquery is introduced with the EXISTS operator, SQL Server tests whether data that matches the subquery exists. No rows are actually retrieved. SQL Server terminates the retrieval of rows when it knows that at least one row satisfies the WHERE condition in the subquery.

SQL server process

When SQL Server processes subqueries that use the EXISTS or NOT EXISTS operator:

•The outer query tests for the existence of rows that the subquery returns.

•The subquery returns either a TRUE or FALSE value based on the given condition in the query.

•The subquery does not produce any data.

Deleting Rows Based on Other Tables

- Specifying Conditions in the WHERE Clause (Subqueries determine which rows to delete)

```
DELETE FROM [order details]
WHERE orderid IN (
    SELECT orderid
    FROM orders
    WHERE orderdate = '4/14/1998')
```

- Using an additional FROM clause

```
DELETE FROM [order details]
    FROM orders AS o
    INNER JOIN [order details] AS od   ON o.orderid = od.orderid
    WHERE orderdate = '4/14/1998'
```


Using an additional FROM clause

In a DELETE statement, the WHERE clause references values in the table itself and is used to decide which rows to delete. If you use an additional FROM clause, you can reference other tables to make this decision. When you use the DELETE statement with an additional FROM clause, consider the following facts:

•The first FROM clause indicates the table from which the rows are deleted.

•The second FROM clause may introduce a join and acts as the restricting criteria for the DELETE statement.

Syntax :

```
DELETE [FROM] {table_name | view_name}
[FROM {<table_source>} [,.n]]
[WHERE search_conditions ]
```

In example shown in slide, uses a join operation with the DELETE statement to remove rows from the **order details** table for orders taken on 4/14/1998.

Specifying conditions in the WHERE clause

You also can use subqueries to determine which rows to delete from a table based on rows of another table. You can specify the conditions in the WHERE clause rather than using an additional FROM clause. Use a nested or correlated subquery in the WHERE clause to determine which rows to delete.

Updating Rows Based on Other Tables

- How the UPDATE statement works
  - Never updates the same row twice
  - Requires table prefixes on ambiguous column names
- Specifying Rows to Update Using Subqueries (Correlates the subquery with the updated table)

```
UPDATE products
SET unitprice = unitprice + 2
WHERE supplierid in (
   SELECT supplierid
   FROM suppliers
   WHERE country = 'USA'
)
```

Pragati Software Pvt. Ltd., 207, Lok Center, Marol-Maroshi Road, Marol, Andheri (East), Mumbai 400 059. www.pragatisoftware.com 88

Use the UPDATE statement with a FROM clause to modify a table based on values from other tables.

Using the UPDATE statement

When you use joins and subqueries with the UPDATE statement, consider the following facts and guidelines:

•SQL Server never updates the same row twice in a single UPDATE statement. This is a built-in restriction that minimizes the amount of logging that occurs during updates.

•Use the SET keyword to introduce the list of columns or variable names to be updated. Columns referenced by the SET keyword must be unambiguous. For example, you can use a table prefix to eliminate ambiguity.

Syntax :

```
UPDATE {table_name | view_name}
SET
{ column_name={expression | DEFAULT | NULL}
  |@variable=expression}[,.n]
[FROM {  <table_source> ]
[WHERE search_conditions]
```

Specifying rows to update using subqueries

When you use subqueries to update rows, consider the following facts and guidelines:

•If the subquery does not return a single value, you must introduce the subquery with the IN, EXISTS, ANY or ALL keyword.

•Consider using aggregate functions with correlated subqueries, because SQL Server never updates the same row twice in a single UPDATE statement.

Updating Rows Based on Other Tables (Contd…)

- Specifying rows to update using joins
  - Uses the FROM clause
  - This example uses a join to update the **products** table by adding $2.00 to the **unitprice** column for all products supplied by suppliers in the United States (USA).

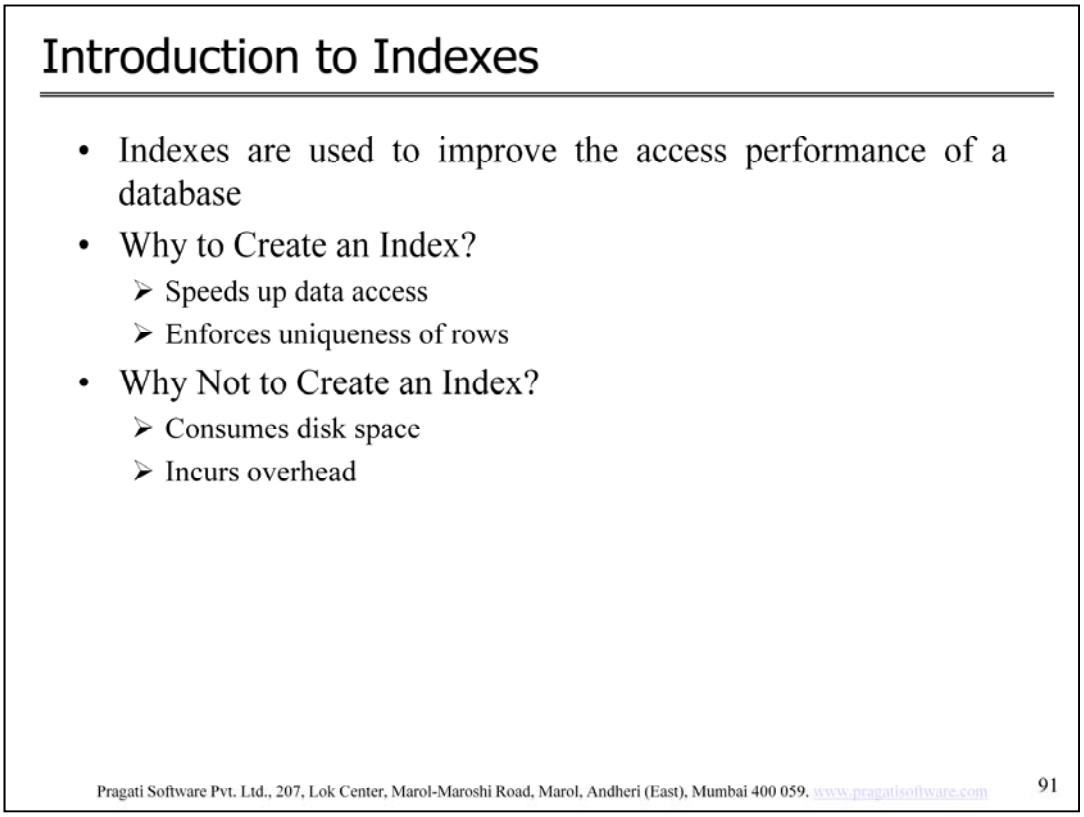
```
UPDATE products
SET unitprice = unitprice + 2
FROM products INNER JOIN suppliers
ON products.supplierid = suppliers.supplierid
WHERE suppliers.country = 'USA'
```


Specifying rows to update using JOINs

When you use joins to update rows, use the FROM clause to specify joins in the UPDATE statement.

Module 10: Planning Indexes

- Overview
  - Introduction to Indexes
  - Deciding Which Columns to Index
  - Creating indexes
  - Maintaining indexes



Indexes are the other significant user-defined, on-disk data structure (in addition to tables). An Index provides fast access to data when the data can be searched by the value that is the Index key. Think of Indexes in your everyday life. You're reading a SQL Server book, and you want to find entries for the word *SELECT*. You have two basic choices for doing this: you can open the book and scan through it page by page, or you can look in the Index in the back, find the word *SELECT*, and then turn to the page numbers listed. That is exactly how an Index works in SQL Server. Understanding how data is stored is the basis for understanding how SQL Server accesses data.

Why to Create an Index?

• Indexes generally accelerate queries that join tables and perform sorting or grouping operations.

• Indexes enforce the uniqueness of rows if uniqueness is defined when you create the index.

• Indexes are created and maintained in ascending or descending sorted order.

• Indexes are best created on columns with a high degree of selectivity, that is, columns or combinations of columns in which the majority of the data is unique.

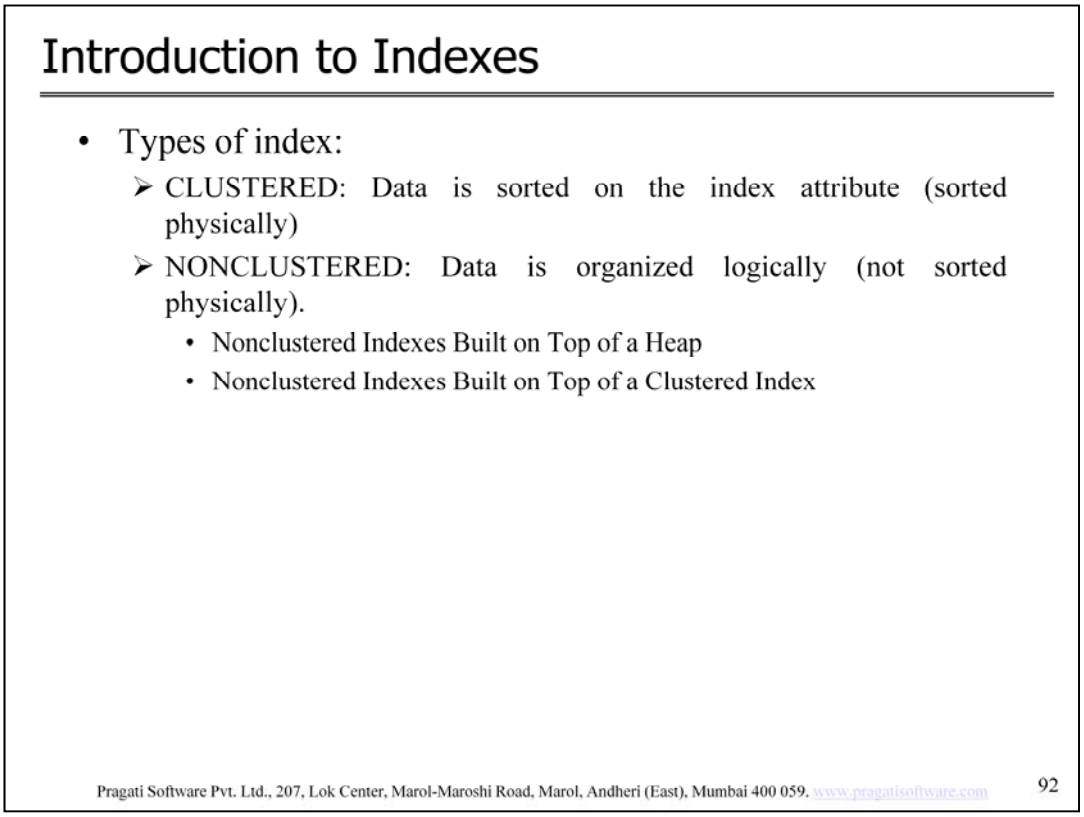
Why Not to Create an Index?

Indexes are useful, but they consume disk space and incur overhead and maintenance costs. Consider the following facts and guidelines about indexes:

•When you modify data on an indexed column, SQL Server updates the associated indexes.

•Maintaining indexes requires time and resources. Therefore, do not create an index that you will not use frequently.

•Indexes on columns containing a large amount of duplicate data may have few benefits.



There are two types of index:

CLUSTERED: Data is sorted on the index attribute

NONCLUSTERED: Data is organized logically (not sorted physically).

A table without a clustered index is known as a heap.

A table can have maximum of one clustered index

Clustered Indexes

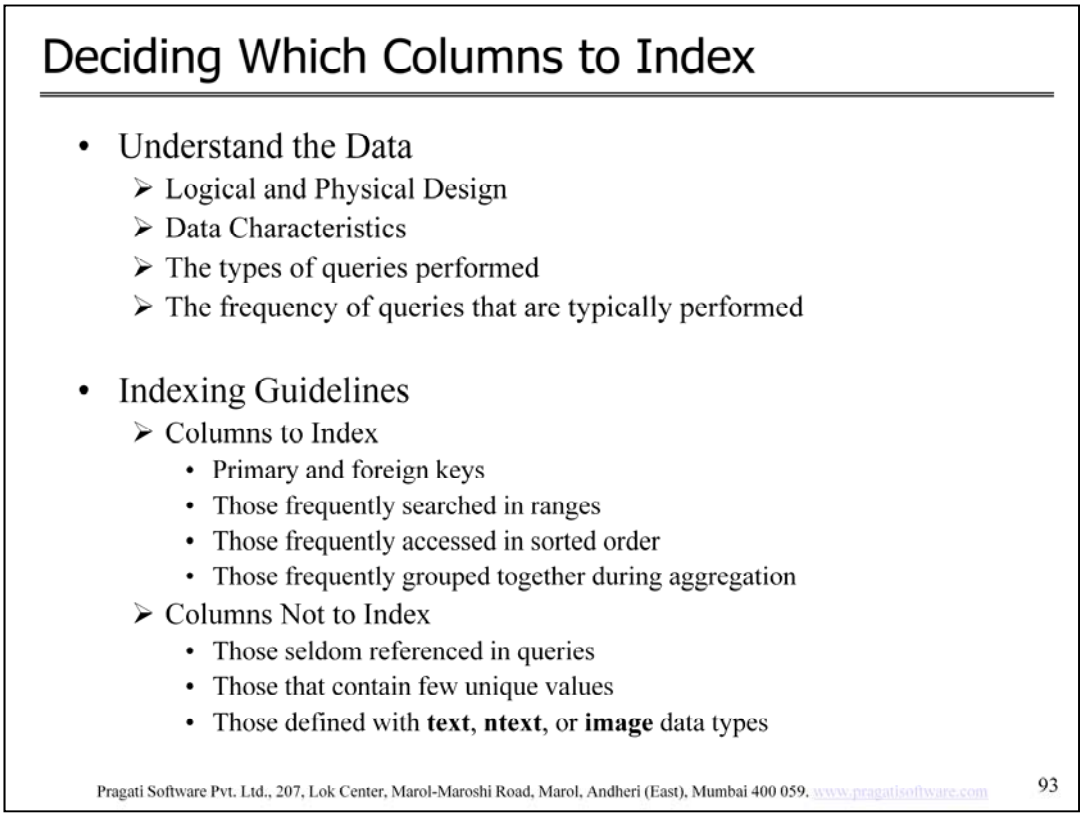
In a clustered index, the leaf level is the actual data page. Data is physically stored on a data page in ascending order. The order of the values in the index pages is also ascending.

Nonclustered Indexes Built on Top of a Heap

When a nonclustered index is built on top of a heap, SQL Server uses row identifiers in the index pages that point to rows in the data pages. The row identifiers store data location information.

Nonclustered Indexes Built on Top of a Clustered Index

When a nonclustered index is built on top of a table with a clustered index, SQL Server uses a clustering key in the index pages that point to the clustered index. The clustering key stores data location information.



Before you create an index, you should have a thorough understanding of the data, including:

•Logical and physical design.

•Data characteristics.

•How data is used.

To design useful and effective indexes, you must rely on the analysis of queries that users send. A poor analysis of how users access data becomes apparent in the form of slow query response or even unnecessary table locks. You should be aware of how users access data by observing:

•The types of queries performed.

•The frequency of queries that are typically performed.

Having a thorough understanding of the user's data requirements helps to determine which columns to index and what types of indexes to create. You might have to sacrifice some speed on one query to gain better performance on another.

Indexing Guidelines

Your business environment, data characteristics, and use of the data determine the columns that you specify to build an index. The usefulness of an index is directly related to the percentage of rows returned from a query. Low percentages or high selectivity are more efficient. When you create an index on a column, the column is referred to as the index column. A value within an index column is called a key value.

Columns to Index

Create indexes on frequently searched columns, such as:

•Primary keys.

•Foreign keys or columns that are used frequently in joining tables.

•Columns that are searched for ranges of key values.

•Columns that are accessed in sorted order.

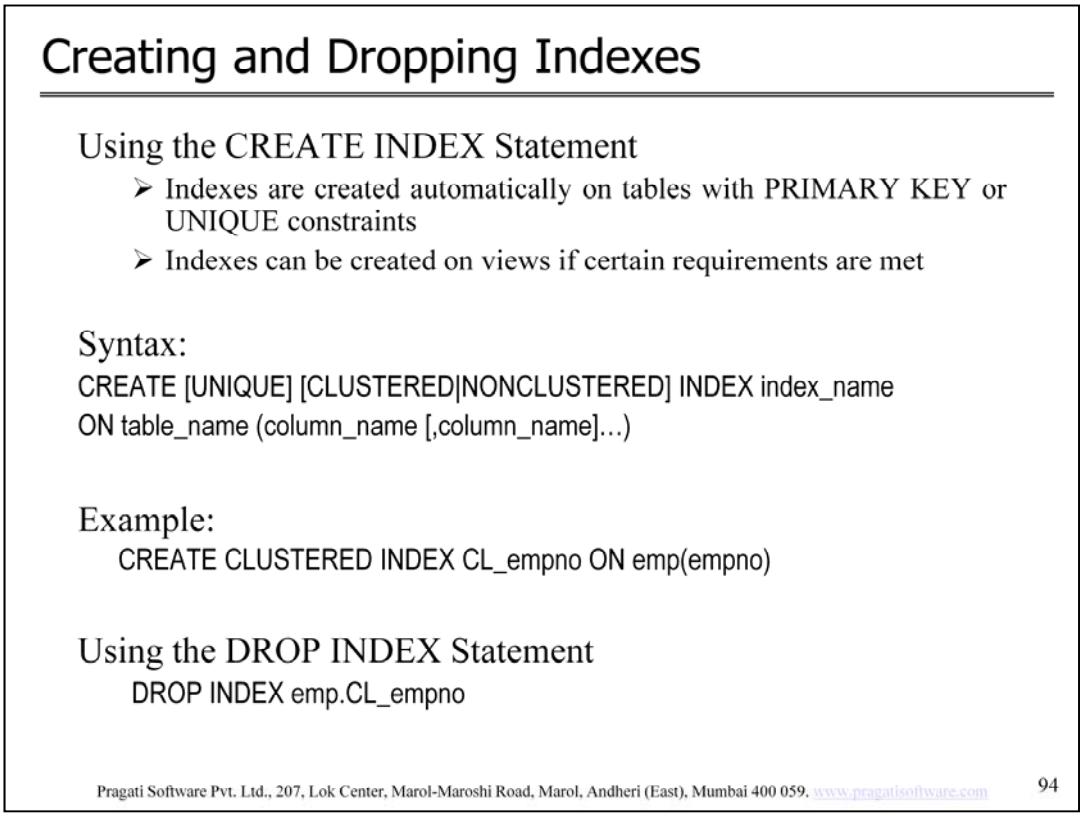
•Columns that are grouped together during aggregation.

Columns Not to Index

•You seldom reference in a query.

•Contain few unique values. For example, an index on a column with two values, male and female, returns a high percentage of rows.

•Are defined with **text**, **ntext**, and **image** data types. Columns with these data types cannot be indexed.



Now that you are familiar with the different index architectures, we will discuss creating and dropping indexes and obtaining information on existing indexes.

You create indexes by using the CREATE INDEX statement and can remove them by using the DROP INDEX statement. You must be the table owner to execute either statement in a database.

Using the CREATE INDEX Statement

Use the CREATE INDEX statement to create indexes. This example creates a clustered index on the **LastName** column in the **Employees** table.

```
CREATE CLUSTERED INDEX CL_lastname
ON employees (lastname)
```

Using the DROP INDEX Statement

Use the DROP INDEX statement to remove an index on a table. This example drops the **cl_lastname** index from the **Member** table.

```
USE Northwind
DROP INDEX employees.CL_lastname
```

Creating Unique & Composite Indexes

Creating Unique Indexes:

- Duplicate key values are not allowed when a new row is added to the table

Example:

```
CREATE UNIQUE NONCLUSTERED INDEX U_DeptNo
ON Dept(DeptNo)
```

Creating Composite Indexes:

- Composite indexes specify more than one column as the key value.

Example:

```
CREATE UNIQUE NONCLUSTERED INDEX U_OrdID_ProdID
ON [Order Details] (OrderID, ProductID)
```


A *unique* index ensures that all data in an indexed column is unique and does not contain duplicate values. Unique indexes ensure that data in indexed columns is unique. If the table has a PRIMARY KEY or UNIQUE constraint, SQL Server automatically creates a unique index when you execute the CREATE TABLE or ALTER TABLE statement. This example creates a unique, nonclustered index named **U_CustID** on the **Customers** table. The index is built on the **CustomerID** column. The value in the **CustomerID** column must be a unique value for each row of the table.

```
CREATE UNIQUE NONCLUSTERED INDEX U_CustID
ON customers(CustomerID)
```

Finding All Duplicate Values in a Column

If duplicate key values exist when you create a unique index, the CREATE INDEX statement fails. SQL Server returns an error message with the first duplicate, but other duplicate values may exist, as well. Use the following sample script on any table to find all duplicate values in a column. Replace the italicized text with information specific to your query.

```
SELECT index_col, COUNT (index_col)
FROM tablename
GROUP BY index_col
HAVING COUNT(index_col)>1 ORDER BY index_col
```

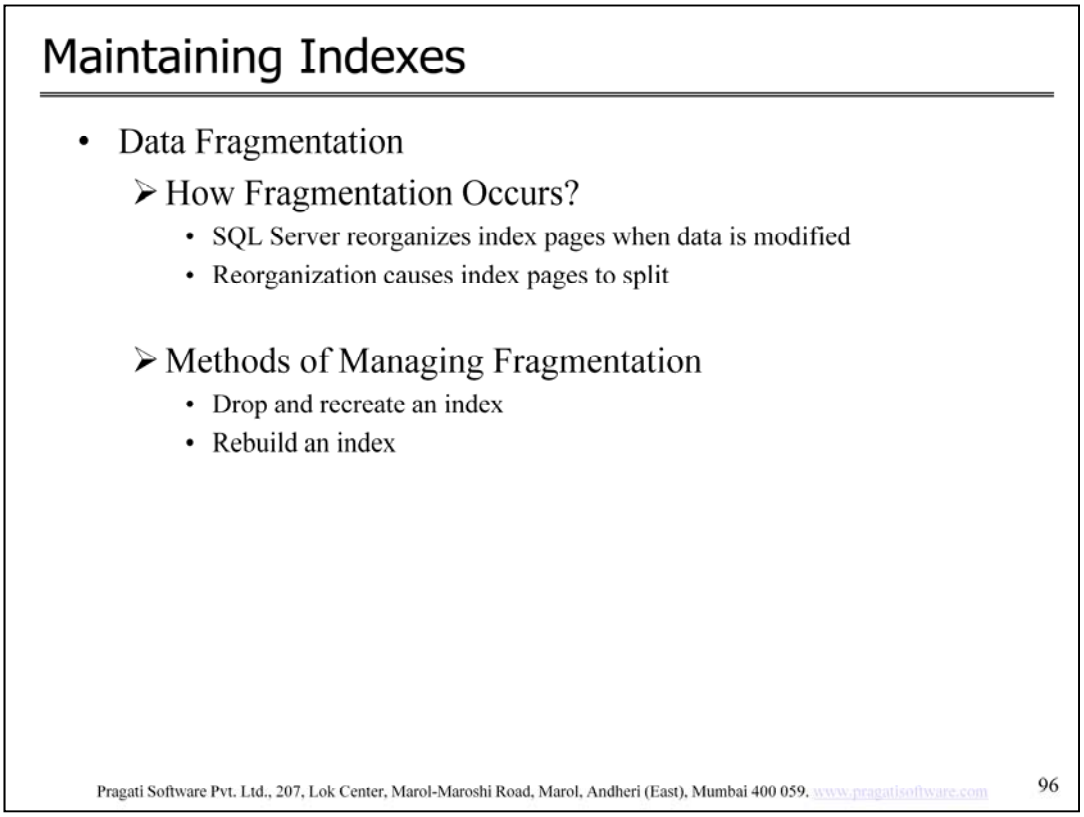
***Composite* indexes specify more than one column as the key value. You create composite indexes:**

•When two or more columns are best searched as a key.

•If queries reference only the columns in the index.

This example creates a nonclustered, composite index on the **Order Details** table. The **OrderID** and the **ProductID** columns are the composite key values. Notice that the **OrderID** column is listed first because it is more selective than the **ProductID** column.

```
CREATE UNIQUE NONCLUSTERED INDEX U_OrdID_ProdID
ON [Order Details] (OrderID, ProductID)
```

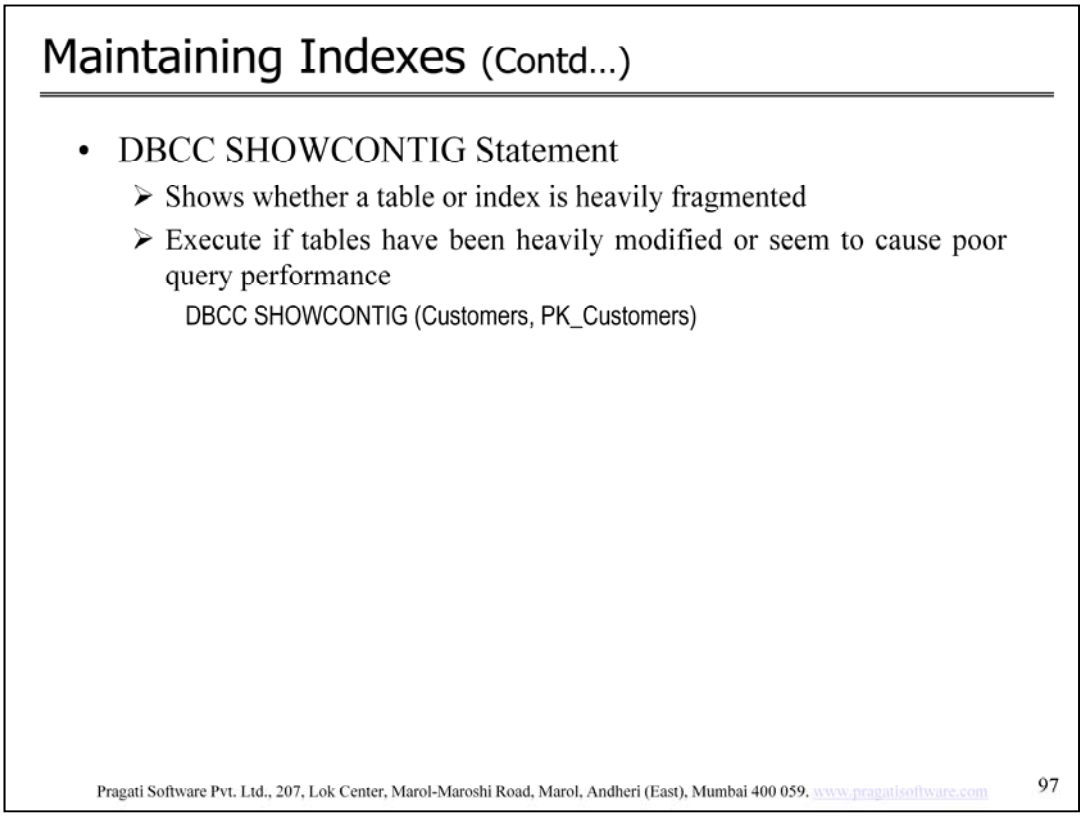


How Fragmentation Occurs?

Fragmentation occurs when data is modified. For example, when rows of data are added to or deleted from a table, or when values in the indexed columns are changed, SQL Server adjusts the index pages to accommodate the changes and to maintain the storage of the indexed data. The adjustment of the index pages is known as a *page split*. The splitting process increases the size of a table and the time that is needed to process queries.

Methods of Managing Fragmentation

There are two methods of managing fragmentation in SQL Server. The first method is to drop and recreate a clustered index The second method is to rebuild an index.



What DBCC SHOWCONTIG Statement Determines?

When you execute the DBCC SHOWCONTIG statement, SQL Server goes across the index pages at the leaf level to determine whether a table or specified index is heavily fragmented. The DBCC SHOWCONTIG statement also determines whether the data and index pages are full. This example executes a statement that accesses the **Customers** table.

```
DBCC SHOWCONTIG (Customers, PK_Customers)
```

Where,

Pages Scanned, defines the total number of pages in the table or index

Extents Scanned, defines the total number of extents in the table or index

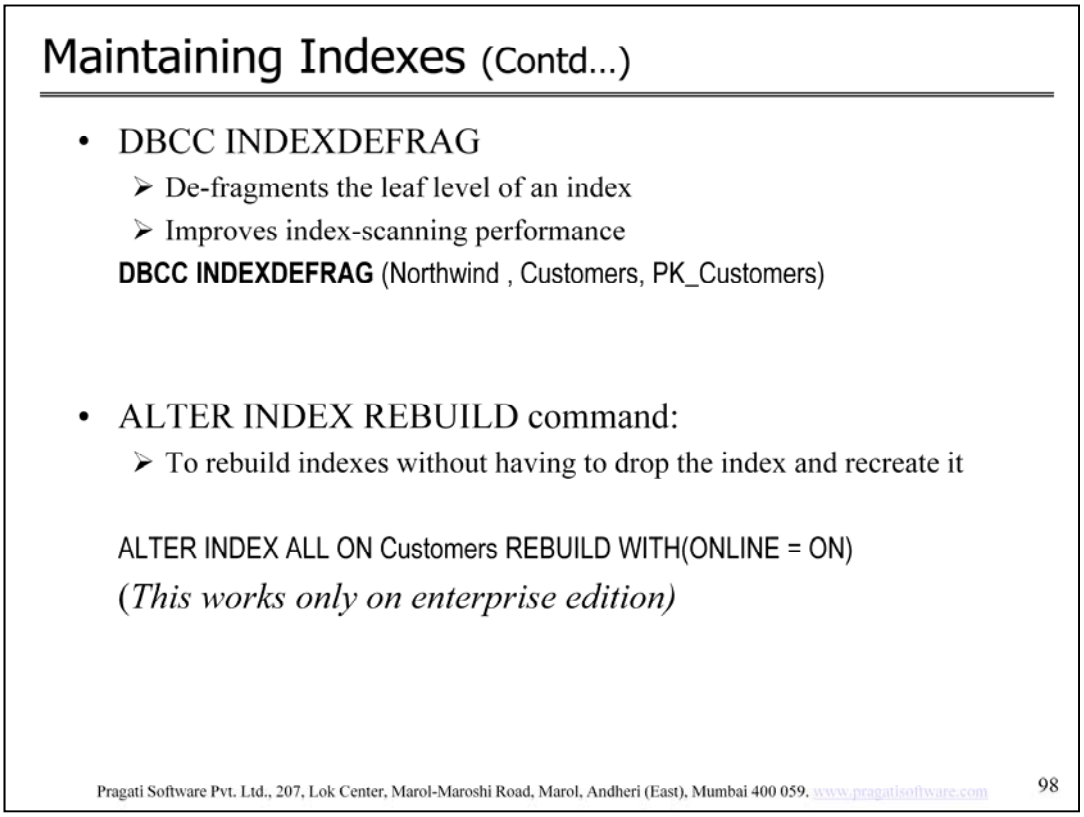
Extents Switches, defines the number of times DBCC moved from one Extent to another while traversing the page chain

Avg. Pages per Extent, defines the no. of pages in each extent that are related

Scan Density [Best Count: Actual Count], defines the percentage of fragmentation that exists. If the percentage is less than 100 it indicates fragmentation. Best Count: Actual Count is ratio of the ideal no. to actual no. of extent changes

Logical Scan Fragmentation, defines the percentage of page that are physically not next to each other

Extent Scan Fragmentation, defines the no. of extents that are physically not next to each other



DBCC INDEXDEFRAG

As data in a table changes, the indexes on the table sometimes become *fragmented*. The DBCC INDEXDEFRAG statement can *de-fragment* the leaf level of clustered and nonclustered indexes on tables and views. De-fragmenting arranges the pages so that the physical order of the pages matches the left-to-right logical order of the leaf nodes. This rearrangement improves index-scanning performance.

Example shown next executes the DBCC INDEXDEFRAG statement on the PK_Customers index of the Customers table in the Northwind database.

```
DBCC INDEXDEFRAG (Northwind , Customers, PK_Customers)
```

Use the DROP_EXISTING option

Use the DROP_EXISTING option, to change the characteristics of an index or to rebuild indexes without having to drop the index and recreate it. The benefit of using the DROP_EXISTING option is that you can modify indexes created with PRIMARY KEY or UNIQUE constraints.

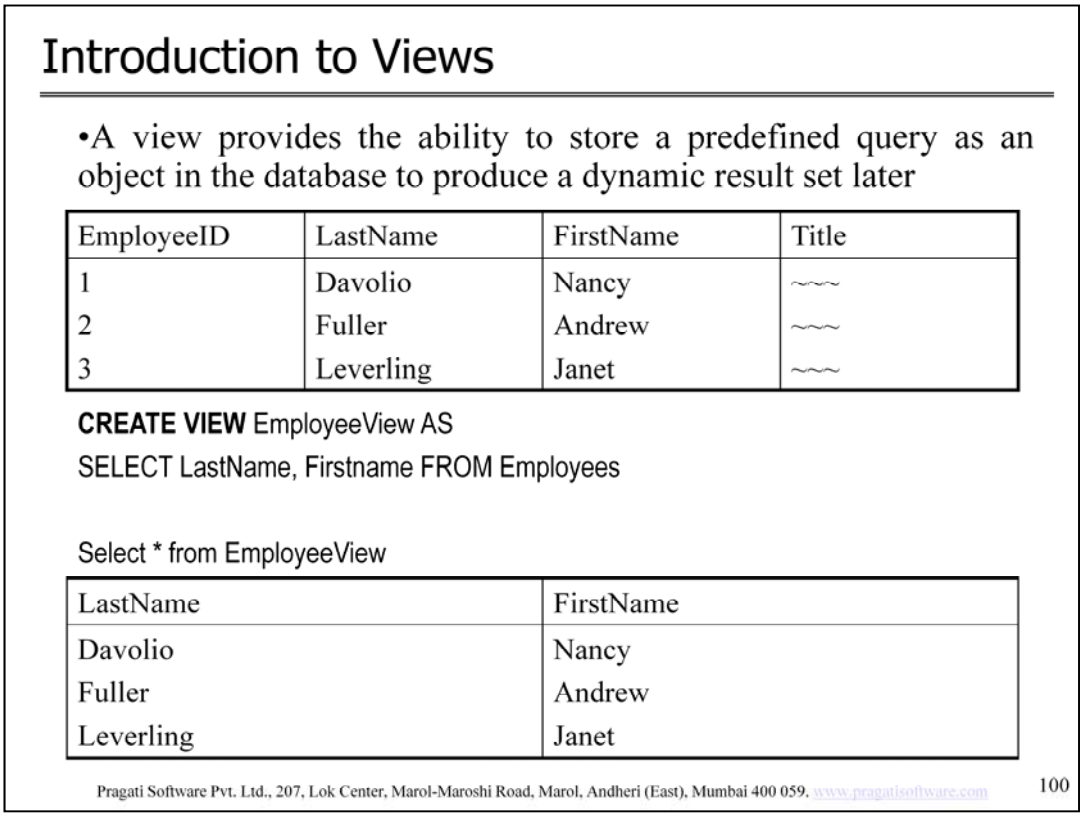
ALTER INDEX REBUILD command:

Use ALTER INDEX REBUILD command to rebuild indexes without having to drop the index and recreate it.

*This works only on enterprise edition

Module 11: Implementing Views

- Overview
  - Introduction to Views
  - Advantages of Views
  - Defining Views
  - Modifying Data Through Views



Introduction to Views

A view provides the ability to store a predefined query as an object in the database for later use. The tables queried in a view are called *base tables*. With a few exceptions, you can name and store any SELECT statement as a view.

Common examples of views are:

•A subset of rows or columns of a base table.

•A union of two or more base tables.

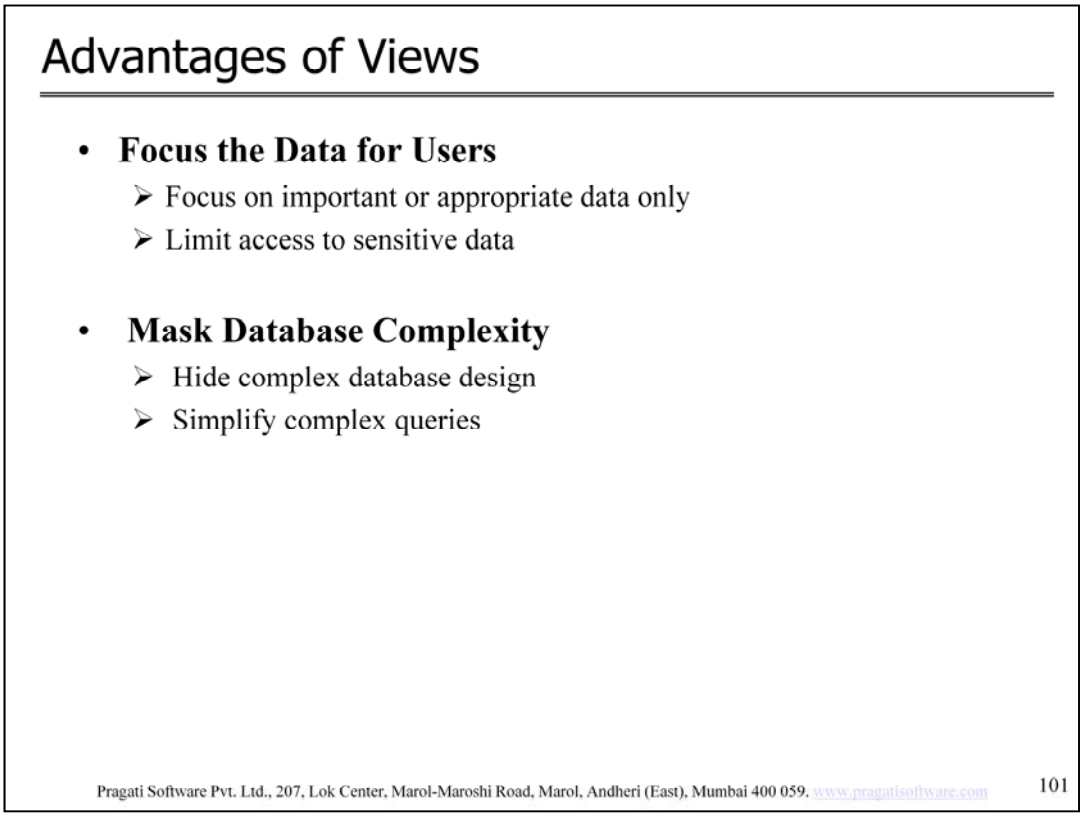
•A join of two or more base tables.

•A statistical summary of a base table.

•A subset of another view, or some combination of views and base tables.

This example creates the **dbo.EmployeeView** view in the **Northwind** database. The view displays two columns in the **Employees** table.

```
CREATE VIEW dbo.EmployeeView
AS
SELECT LastName, Firstname
FROM Employees
```



Advantages of Views

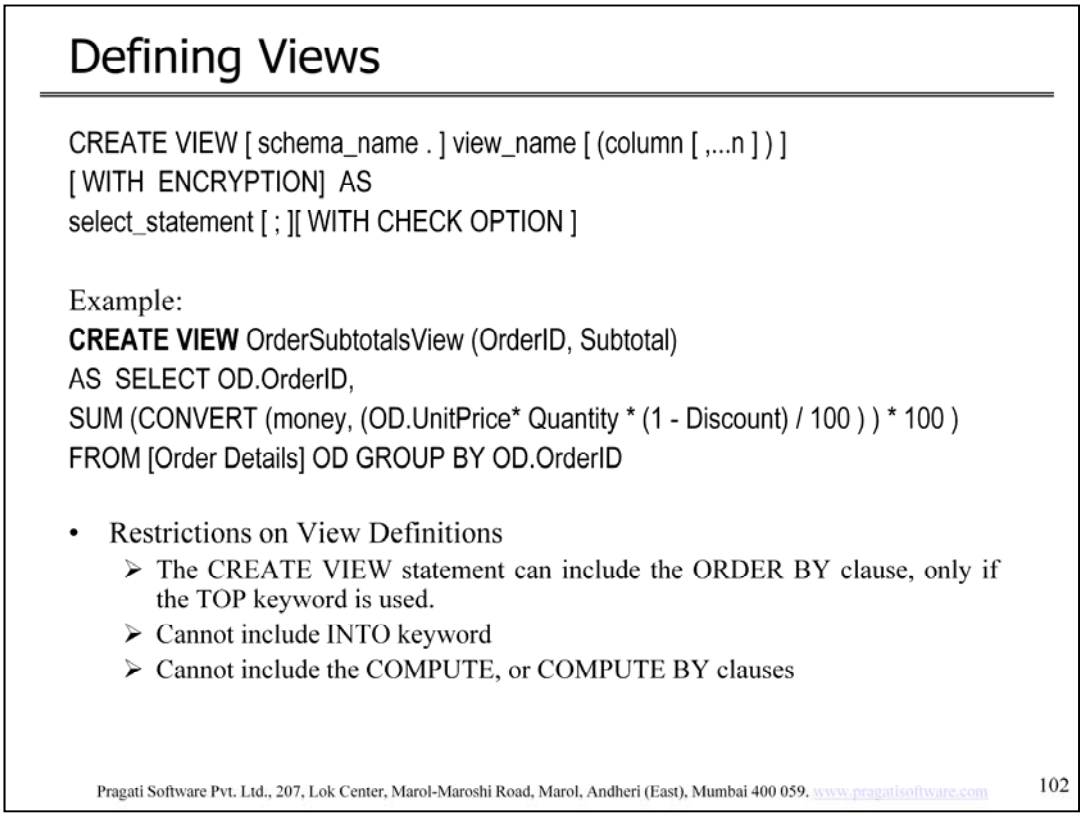
Views offer several advantages, including focusing data for users, masking data complexity

Focus the Data for Users

Views create a controlled environment that allows access to specific data while other data is concealed. Data that is unnecessary, sensitive, or inappropriate can be left out of a view. Users can manipulate the display of data in a view, as is possible in a table. In addition, with the proper permissions and a few restrictions, users can modify the data that a view produces.

Mask Database Complexity

Views shield the complexity of the database design from the user. This provides developers with the ability to change the design without affecting user interaction with the database. In addition, users can see a friendlier version of the data by using names that are easier to understand than the cryptic names that are often used in databases. Complex queries can also be masked through views. The user queries the view instead of writing the query or executing a script.



Creating a View

When you create a view, SQL Server verifies the existence of objects that are referenced in the view definition. Your view name must follow the rules for identifiers. Specifying a view owner name is optional. You should develop a consistent naming convention to distinguish views from tables.

For example, you could add the word view as a suffix to each view object that you create. This allows similar objects (tables and views) to be easily distinguished when you query the **INFORMATION_SCHEMA.TABLES** view.

```
CREATE VIEW [ schema_name . ] view_name [ (column [ ,...n ] ) ]
[ WITH  ENCRYPTION]  AS
select_statement [ ; ][ WITH CHECK OPTION ]
```

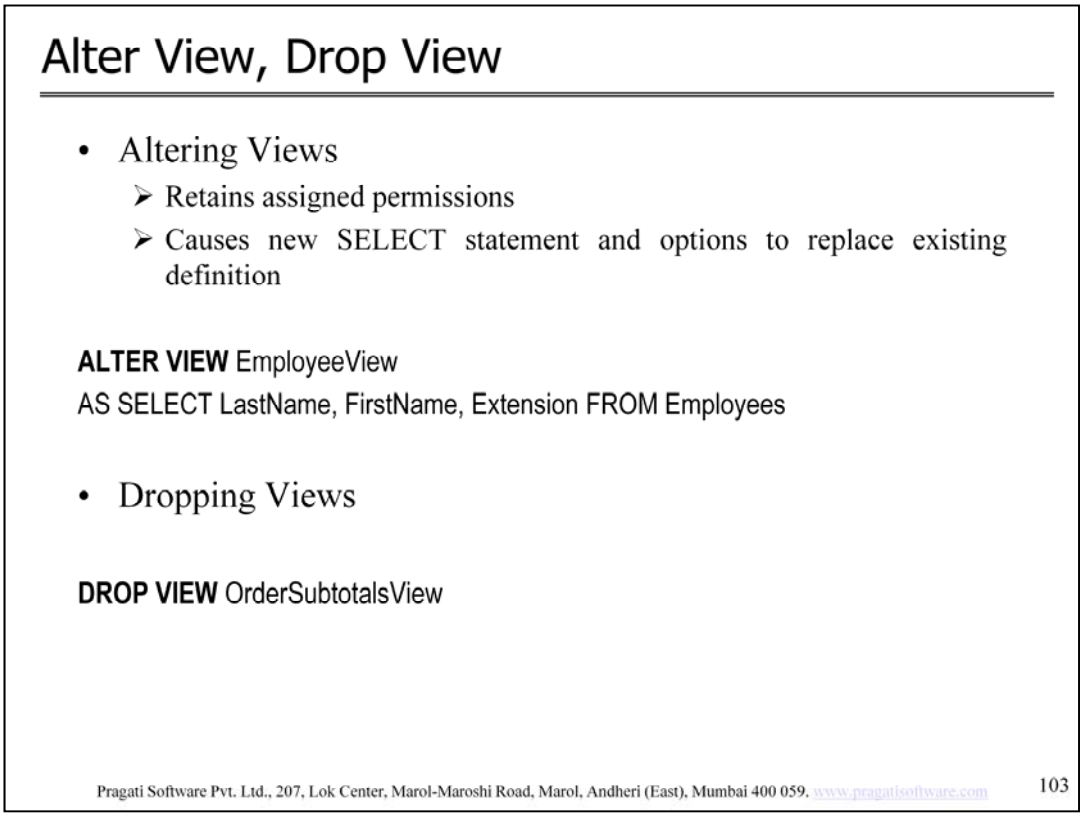
Restrictions on View Definitions

When you create views, consider the following restrictions:

•The CREATE VIEW statement cannot include the COMPUTE, or COMPUTE BY clauses.

•The CREATE VIEW statement cannot include the INTO keyword.

•The CREATE VIEW statement can include the ORDER BY clause, only if the TOP keyword is used.



You often alter views in response to requests from users for additional information or to changes in the underlying table definition. You can alter a view by dropping and recreating it or by executing the ALTER VIEW statement.

Altering Views

The ALTER VIEW statement changes the definition of a view, including indexed views, without affecting dependent stored procedures or triggers. This allows you to retain permissions for the view. This statement is subject to the same restrictions as the CREATE VIEW statement. If you drop a view and then recreate it, you must reassign permissions to it.

Dropping Views

If you no longer need a view, you can remove its definition from the database by executing the DROP VIEW statement. Dropping a view removes its definition and all permissions assigned to it. Furthermore, if users query any views that reference the dropped view, they receive an error message. However, dropping a table that references a view does not drop the view automatically. You must drop it explicitly.

Hiding View Definition

Because users may display the definition of a view by using SQL Server Enterprise Manager, by querying **INFORMATION_SCHEMA.VIEWS**, or by querying the **syscomments** system table, you might want to hide certain view definitions.

Hiding View Definition

- It is possible to see view definition using system procedure sp_helptext:

  sp_helptext EmployeeView

- To hide view definition of sensitive data use the WITH ENCRYPTION Option

```
CREATE VIEW OrderSubtotalsView (OrderID, Subtotal)
WITH ENCRYPTION
AS  SELECT OD.OrderID,
SUM (CONVERT (money, (OD.UnitPrice* Quantity * (1 - Discount) / 100 ) ) * 100 )
FROM [Order Details] OD GROUP BY OD.OrderID
```


Use the WITH ENCRYPTION Option

You can encrypt the **syscomments** table entries that contain the text of the CREATE VIEW statement by specifying the WITH ENCRYPTION option in the view definition.

Before you encrypt a view, ensure that the view definition (script) is saved to a file. To decrypt the text of a view, you must drop the view and recreate it, or alter the view and use the original syntax.

In this example, OrderSubtotalsView is created by using the WITH ENCRYPTION option so that the view definition is hidden.

```
CREATE VIEW OrderSubtotalsView (OrderID, Subtotal)
WITH ENCRYPTION
AS  SELECT OD.OrderID,
SUM (CONVERT (money, (OD.UnitPrice* Quantity * (1 - Discount) / 100 ) ) * 100 )
FROM [Order Details] OD GROUP BY OD.OrderID
```

Modifying Data Through Views

- In general, the view must not include aggregate functions, built-in functions, UNION, a GROUP BY clause, or DISTINCT. clauses in the SELECT statement to be updateable

- The Modification of data through views:-
  - Cannot Affect More Than One Underlying Table
  - Cannot Be Made to Certain Columns (Calculated columns)
  - Can Cause Errors if they Affect Columns That Are Not Referenced in the View
  - Are Verified If the WITH CHECK OPTION Has Been Specified

Views do not maintain a separate copy of data. Instead, they show the result set of a query on one or more base tables. Therefore, whenever you modify data in a view, you are actually modifying the base table.

With some restrictions, you can insert, update, or delete table data freely through a view. In general, the view must be defined on a single table and must not include aggregate functions or GROUP BY clauses in the SELECT statement.

Specifically, modifications that are made by using views:

•Cannot affect more than one underlying table. You can modify views that are derived from two or more tables, but each update or modification can affect only one table.

•Cannot be made on certain columns. SQL Server does not allow you to change a column that is the result of a calculation, such as columns that contain computed values, built-in functions, or row aggregate functions.

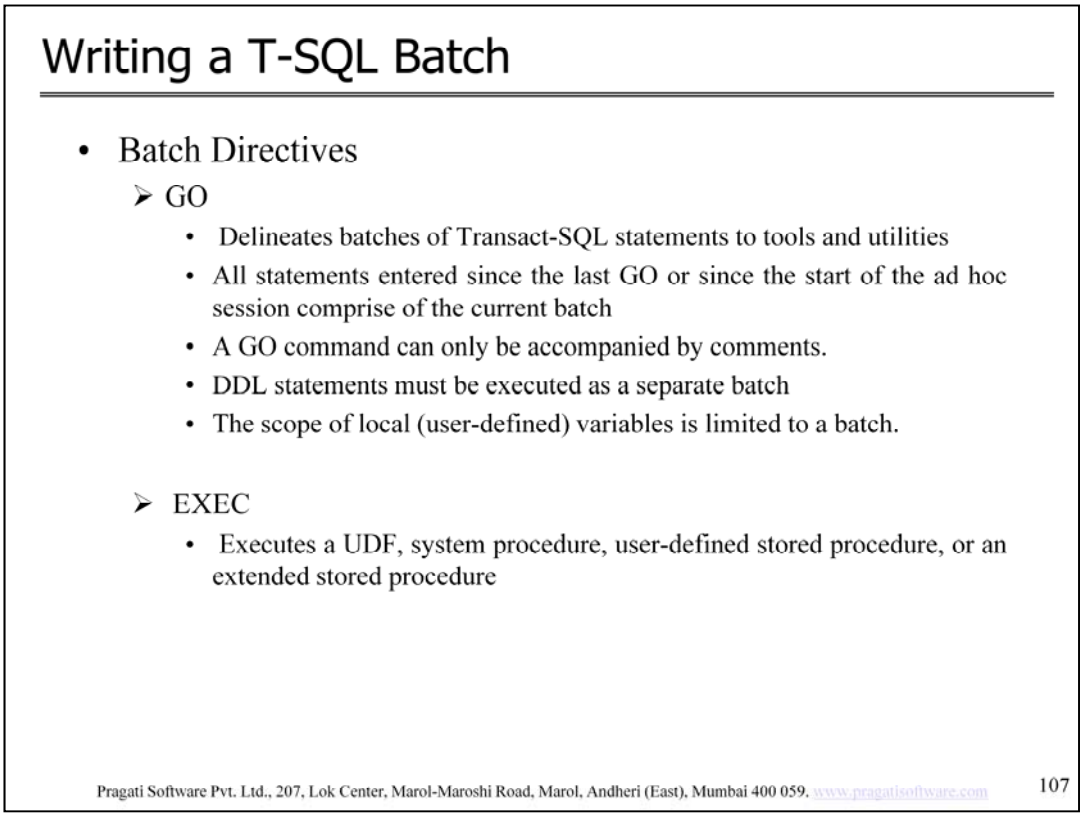
•Can cause errors if modifications affect columns that are not referenced in the view.

For example, you will receive an error message if you insert a row into a view that is defined on a table that contains columns that are not referenced in the view and that do not allow NULLs or contain default values.

•Are verified if the WITH CHECK OPTION has been specified in the view definition. The WITH CHECK OPTION forces all data modification statements that are executed against the view to adhere to certain criteria. These criteria are specified within the SELECT statement that defines the view. If the changed values are out of the range of the view definition, SQL Server rejects the modifications.

Module 12:Introduction to Programming Objects

- Overview
  - Writing a T-SQL Batch
  - Using variables
  - Including Comments
  - The IF ELSE construct
  - The BEGIN END Block
  - The CASE Expression
  - The PRINT Statement



SQL Server processes single or multiple Transact-SQL statements in batches. A batch directive instructs SQL Server to parse and execute all of the instructions within the batch. There are two basic methods for handing off batches to SQL Server.

GO

SQL Server utilities interpret GO as a signal to send the current batch of Transact-SQL statements to SQL Server. A GO command delineates batches of Transact-SQL statements to tools and utilities. A GO command is not an actual Transact-SQL statement.

When using GO, consider these facts:

•The current batch of statements is composed of all statements entered since the last GO, or since the start of the ad hoc session (or script, if this is the first GO).

•A Transact-SQL statement cannot occupy the same line as a GO command, although the line can contain comments.

•Users must follow the rules for batches.

For example, some Data Definition Language statements must be executed separately from other Transact-SQL statements by separating the statements with a GO command. The scope of local (user-defined) variables is limited to a batch, and cannot be referenced after a GO command.

EXEC

The EXEC directive is used to execute a user-defined function, system procedure, user-defined stored procedure, or an extended stored procedure; it can also control the execution of a character string within a Transact-SQL batch. Parameters can be passed as arguments, and a return status can be assigned.

Using variables

- Stores results temporarily, for later use
- Declare user-defined local variable with DECLARE @ statement
- Assign values with SET or SELECT @ Statement
- Variables Have Local or Global Scope
  - A local variable is shown with one @ symbol preceding its name
  - A global variable is shown with two @@ symbols preceding its name.

Example

```
DECLARE @EmpID varchar(11) ,@vlName char(20)
SET @vlname = 'Dodsworth'
SELECT @EmpID = employeeid FROM employees WHERE LastName = @vlname
SELECT @EmpID AS EmployeeID
GO
```


Variables are language elements with assigned values. You can use local variables in Transact-SQL. A local variable is user-defined in a DECLARE @ statement, assigned an initial value in a SET or SELECT statement, and then used within the statement, batch, or procedure in which it was declared. A local variable is shown with one @ symbol preceding its name; a global variable is shown with two @ symbols preceding its name. Local variables last only for the duration of a batch, whereas global variables last for the duration of a session.

Syntax:

```
DECLARE {@local_variable data_type} [,...n]
SET @local_variable_name = expression
```

Example

```
DECLARE  @EmpID  varchar(11), @vlName char(20)
SET @vlname = 'Dodsworth'
SELECT @EmpID = employeeid FROM employees WHERE LastName = @vlname
SELECT @EmpID AS EmployeeID
GO
```

This example creates the **@EmpID** and **@vlname** local variables, assigns a value to **@vlname**, and then assigns a value to **@EmpID** by querying the **Northwind** database to select the record containing the value of the **@vlname** slocal variable.

Including Comments

- T-SQL supports two forms of comments in source code
- In-line Comments: Double-hyphen (--) is used to include single-line comments

```
SELECT productname, (unitsinstock - unitsonorder) -- Calculates inventory
, supplierID FROM products
```

- Block Comments : C-style pair of /*…*/ can be used for marking a block comment

```
/* This code retrieves all rows of the products table and displays the unit price, the
   unit price increased by 10 percent, and the name of the product. */
SELECT unitprice, (unitprice * 1.1), productname FROM products
```


Comments are non-executing strings of text placed in statements to describe the action that the statement is performing or to disable one or more lines of the statement. Comments can be used in one of two ways. in line with a statement or as a block.

In-line Comments

You can create in-line comments using two hyphens (--) to set a comment apart from a statement. Transact-SQL ignores text to the right of the comment characters. This commenting character can also be used to disable lines of a statement.

This example uses an in-line comment to explain what a calculation is doing.

```
SELECT productname, (unitsinstock - unitsonorder) -- Calculates inventory
 , supplierid
FROM products
```

The IF ELSE construct

- These elements specify that SQL Server should execute the first alternative if the certain condition is true.
- Otherwise, SQL Server should execute the second alternative.

```
DECLARE @minreorderlevel smallint
SELECT @minreorderlevel = min(reorderlevel) FROM Products
IF @minreorderlevel <= 0
   UPDATE Products SET reorderlevel=5 WHERE reorderlevel<=0
ELSE
   UPDATE Products SET reorderlevel=reorderlevel+reorderlevel*0.5
GO
```


The IF…ELSE Construct

The IF…ELSE construct is used to specify conditional execution of code, with an optional ELSE part specifying an alternative code to be executed.

Example:

```
DECLARE @minreorderlevel smallint
SELECT @minreorderlevel = min(reorderlevel) FROM Products
IF @minreorderlevel <= 0
    UPDATE Products SET reorderlevel=5 WHERE reorderlevel<=0
ELSE
    UPDATE Products SET reorderlevel=reorderlevel+reorderlevel*0.5
GO
```

In above example if minimum reorder level is less than 0 then update reorder level to 5 else increase reorder level by 0.5.

The BEGIN END Block

- The BEGIN…END statements are used to mark a block of statements to be executed.
- Typically, BEGIN immediately follows IF, ELSE, or WHILE statements.

```
CREATE TABLE reorderhistory (comments varchar(75),change_date datetime)

DECLARE @minreorderlevel smallint
SELECT @minreorderlevel = min(reorderlevel) FROM Products
IF @minreorderlevel <= 0
    BEGIN
        UPDATE Products SET reorderlevel=5 WHERE reorderlevel<=0
        INSERT reorderhistory VALUES ('Updated to 5 where <= 0',getdate())
    END
ELSE
    BEGIN
        UPDATE Products SET reorderlevel=reorderlevel+reorderlevel*0.5
        INSERT reorderhistory VALUES ('Incremented by 0.5 times',getdate())
    END
```


The BEGIN…END Block:

The BEGIN…END statements are used to mark a block of statements to be executed. Typically, BEGIN immediately follows IF, ELSE, or WHILE statements.

In above example, we are creating table which will get updated when reorderlevel column from product gets updated. So in if & else part whenever data of reorderlevel gets updated, it inserts data in reorderhistory table. Updating recoedlevel & inserting data into reorderhistory table these are two statements, so we will use BEGIN & END block.

The CASE Expression

- An alternative to writing code with nested IF…ELSE IF…ELSE statements
- Is not a control-of-flow keyword. It can only be used within SELECT or UPDATE statements
- Example illustrates the use of CASE expression in an in a SELECT statement:

```
SELECT Employeeid, Firstname, Lastname ,
   CASE
     WHEN Region = 'WA' THEN 'Washington'
     WHEN Region = 'NY' THEN 'New York'
     ELSE 'Unassigned'
   END 'Region'
FROM Employees
```


T-SQL provides for the CASE expression, which can be used as an alternative to writing code with nested IF…ELSE IF…ELSE statements.

However, it is to be noted that CASE is not a control-of-flow keyword, it can only be used within SELECT or UPDATE statements.

In example above, when case is matching with ‘WA’ then it is replaced with ‘'Washington’ and when case is matching with ‘NY” then it is replaced with ‘New York’, else region is ‘Unassigned’.

The PRINT Statement

- PRINT statement to display maximum 255 characters.
  print 'Hello, world'
- Cannot use string functions or concatenation operators in the PRINT statement. Statement given below is invalid:
  PRINT 'today is ' + getdate ()
- You can accomplish the same by using a variable:
  DECLARE @output char (40)
  SELECT @output = 'Today is ' +convert (varchar (11), getdate (), 100)
  PRINT @output

The PRINT Statement:

T-SQL provides the PRINT statement to display a character string of maximum 255 characters. You can display a literal character string, or a variable of type char or varchar.

For example:

```
PRINT "Hello, world"
```

However, you need to remember that the PRINT statement takes a single parameter, you cannot use string functions or concatenation operators in the PRINT statement.

For example, the following is invalid:-- invalid, concatenation not allowed.

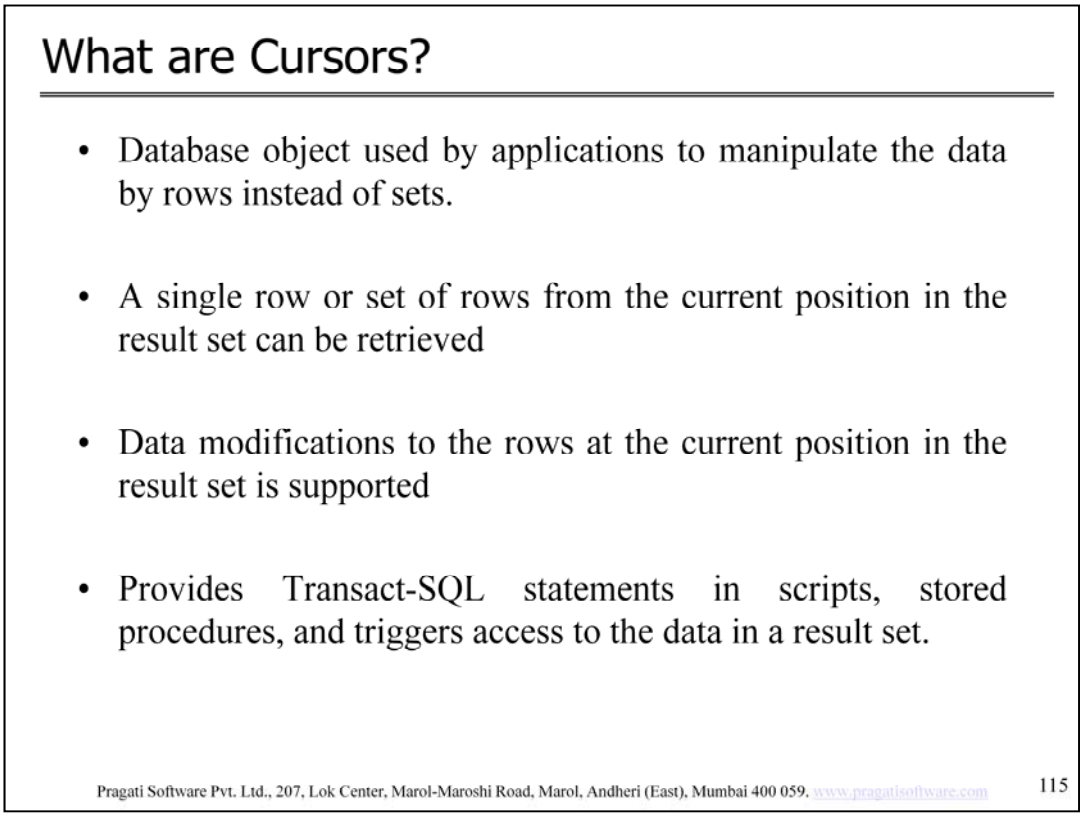
```
PRINT 'today is ' + getdate() -- invalid, this is not C.
```

However, you can accomplish the same by using a variable:

```
DECLARE @output char (40)
SELECT @output = 'Today is ' +convert (varchar (11), getdate (), 100)
PRINT @output
```

Module 13: Cursors

- Overview
  - What are Cursors?
  - How Does the Cursor Process?
  - How to Create a Cursor?
  - How does Fetching and Scrolling work?
  - Examples on cursors



What are Cursors?

Cursors is a database object used by applications to manipulate the data by rows instead of sets. Using cursors, multiple operations can be performed row by row against the result set with or without returning to the original table. In other words, cursors conceptually return a result set based on tables within the database

What is Possible Using Cursors?

•Allow positioning at specific rows of the result set.

•Retrieving a single row or set of rows from the current position in the result set.

•Supporting data modifications to the rows at the current position in the result set.

•Supporting different levels of visibility to changes made by other users to the database data that is presented in the result set.

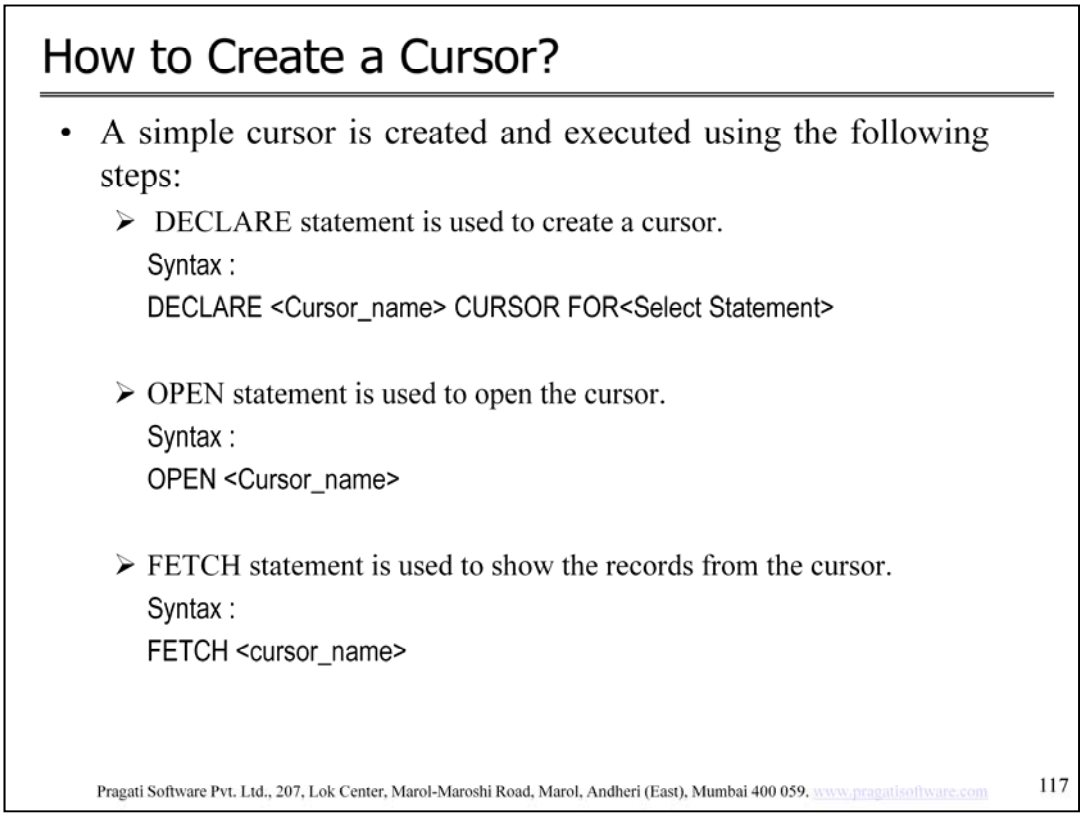
•Providing Transact-SQL statements in scripts, stored procedures, and triggers access to the data in a result set.

How Does the Cursor Process?

- Cursor Processing involves following steps:-
  - Associate a cursor with the result set of a T-SQL statement and define characteristics of a cursor such as how rows are going to be retrieved.
  - Execute the Transact-SQL statement to populate the cursor.
  - Retrieve the rows in a cursor. The operation to retrieve one row or a set of rows from a cursor called a fetch. Performing series of fetch to retrieve the rows backward or forward is called scrolling.
  - Optionally, perform the modifications (update or delete) on the row at the current cursor position.
  - Close the cursor

How Does the Cursor Process?

1. Associate a cursor with the result set of a Transact –SQL statement and define characteristics of a cursor such as how rows are going to be retrieved.
2. Execute the Transact-SQL statement to populate the cursor.
3. Retrieve the rows in a cursor. The operation to retrieve one row or a set of rows from a cursor called a fetch. Performing series of fetch to retrieve the rows backward or forward is called scrolling.
4. Optionally, perform the modifications (update or delete) on the row at the current cursor position.
5. Close the cursor.



How to Create a Cursor?

A cursor is said to exist in many states. The different states of a cursor are the different stages in which it dwells when it is created and used. A simple cursor is created and executed using the following steps. The syntax is:

1. DECLARE statement is used to create a cursor. It contains the SELECT statement to include the records from the table.

Syntax:

```
DECLARE <Cursor_name> CURSORFOR<Select Statement>
```

2. After a cursor is created and before you fetch the records from the cursor, you need to open the cursor. OPEN statement is used to open the cursor.

Syntax:

```
OPEN <Cursor_name>
```

3.Once the cursor is opened, records are fetched from the cursor to display them on the screen. FETCH statement is used to show the records from the cursor.

Syntax:

```
FETCH <cursor_name>
```

How to Create a Cursor?

- CLOSE statement is used to close the cursor when it is not required temporarily. It closes an open cursor by releasing the current result set , the rows can be fetched only after it is reopened.

  Syntax : CLOSE <Cursor_name>

- When the cursor is not required any more, its reference is removed using DEALLOCATE statement

  Syntax : DEALLOCATE <Cursor_name>

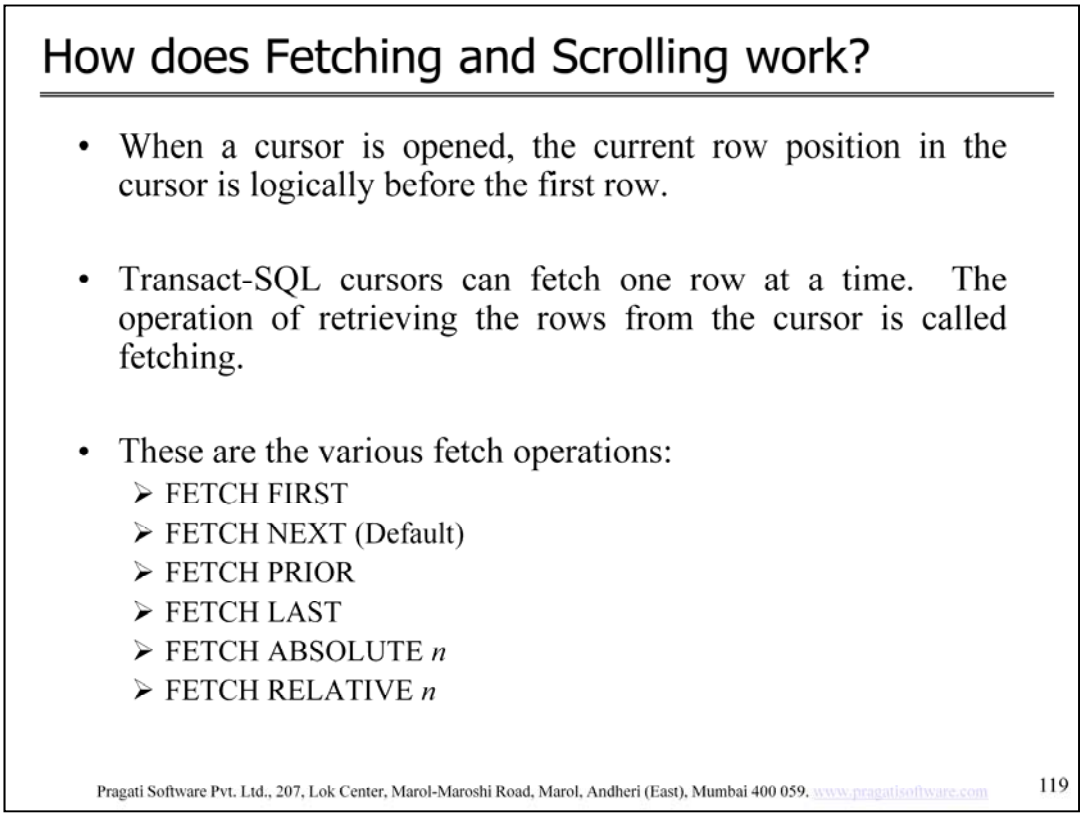
4.Optionally a cursor can be closed when it is not required temporarily. A cursor is closed using the CLOSE TSQL statement. It closes an open cursor by releasing the current result set. Once a cursor is closed, the rows can be fetched only after it is reopened.

Syntax:

```
CLOSE : <Cursor_name>
```

5.When the cursor is not required any more, its reference is removed. DEALLOCATE statement is used to remove the reference of a cursor. Once the cursor is created and opened, rows are fetched from the cursor.

We will see in detail about fetching and scrolling.



How does Fetching and Scrolling work?

When a cursor is opened, the current row position in the cursor is logically before the first row. Transact-SQL cursors can fetch one row at a time. The operation of retrieving the rows from the cursor is called fetching. These are the various fetch operations:

FETCH FIRST Fetches the first row in the cursor.

FETCH NEXT Fetches the row after the previously fetched row.

FETCH PRIOR Fetches the row before the previously fetched row.

FETCH LAST Fetches the Last row in the cursor.

FETCH ABSOLUTE *n* If *n* is a positive integer, it fetches the *n* row from the first row in the cursor. If n is a negative integer, then row before the last row in a cursor is fetched. If *n* is 0, no rows are fetched .For example, FETCH Absolute 2 will display the second record from a table.

FETCH RELATIVE *n* Fetch the nth row from the previously fetched row, if n is positive. If n is negative, the nth row before previously fetched row is fetched. If n is 0, the same row is fetched again.

Note: By default only the FETCH NEXT option works. If you want to use some other options with the FETCH statement, you should include some more options in the DECLARE statement while creating the cursors

How does Fetching and Scrolling work?

- Some more attributes can be added to the DECLARE statement to enhance the scroll ability of the cursors.

Syntax:

```
DECLARE <Cursor_Name> CURSOR
[LOCAL |GLOBAL]
[FORWARD ONLY | SCROLL]
FOR <Select Statements>
```


Some more attributes can be added to the DECLARE statement to enhance the scroll ability of the cursors. As shown in above syntax, each attribute is explained here:

LOCAL: Specifies the scope of the cursor to the stored procedure or trigger in which it is created. In other words the name of the cursor is valid within the scope. The cursor is implicitly de-allocated when the stored procedure or the trigger terminates.

GLOBAL: Specifies the scope of a cursor is global. The cursor name can be referenced in any stored procedure.

FORWARD_ONLY: Specifies that the cursor can be scrolled from the first to the last row. FETCH NEXT is the only supported fetch option. By default a cursor is FORWARD ONLY.

SCROLL: Specifies that all fetch options (FIRST, LAST, PRIOR, NEXT, RELATIVE, ABSOLUTE) are available. If SCROLL is not specified in DECLARE CURSOR, NEXT is the only fetch option supported.

STATIC: Defines a cursor that makes a temporary copy of the data to be used by the cursor. All requests to the cursor are answered from this temporary table in tempdb. Therefore, modifications made to the base tables are not reflected by fetches made to this cursor, and this cursor does not allow modifications.

KEYSET: Specifies the order of the rows in the cursor is fixed when the cursor is opened.

DYNAMIC: Defines a cursor that reflects all the changes made to the rows in its result set as one scrolls around the cursor. It does not support the ABSOLUTE fetch options

FAST_FORWARD: Specifies a FORWARD_ONLY and READ_ONLY cursor. FAST_FORWARD cannot be specified with SCROLL or FOR_UPDATE options. FORWARD ONLY and FAST_FORWARD cursors are mutually exclusive, if one is specified the other one cannot be specified. READ_ONLY Prevents updates made through this cursor. It cannot be referenced in a WHERE CURRENT OF clause in an UPDATE or DELETE statement. A DELETE statement can have a reference of a cursor using WHERE CURRENT OF clause.

SCROLL_LOCKS: Specifies that positioned updates or deletes made through the cursor are guaranteed to succeed. SQL Server locks the rows as they are read into the cursor to ensure their availability for later modification.

OPTIMISTIC: Specifies the positioned updates or deletes made through the cursor do not succeed, if the row is updated since it was read into the cursor.

UPDATE [OF Column_name [,..n]]: Defines updatable columns within the cursor. If OF Column_name [,…n], only the listed columns are allowed for modification.

Examples on cursors

- Example to fetch the details of the orders of the employees who stay in London:

```
SET NOCOUNT ON
DECLARE @EmployeeID int, @FirstName varchar(10),
@LastName varchar(20), @message varchar(180), @orderid int, @companyname
nvarchar(40),@orderdate datetime
PRINT '-------- Order report for Employees from London --------'
DECLARE employees_cursor CURSOR FOR
SELECT EmployeeID, firstname, lastname FROM employees
WHERE city = 'London' ORDER BY employeeid
OPEN employees_cursor
FETCH NEXT FROM employees_cursor INTO
@EmployeeId, @FirstName, @LastName
WHILE @@FETCH_STATUS = 0
BEGIN
        SELECT @message='----- Orders by Employee: ' +@FirstName + '
        ' +          @LastName
        PRINT @message
        DECLARE orders_cursor CURSOR FOR
        SELECT o.orderid,c.companyname,o.orderdate FROM orders o, employees
        e,customers c WHERE e.employeeid=o.employeeid and o.customerid =
        c.customerid AND e.employeeid = @EmployeeID --Variable from the outer cursor
```


Now we see how to implement cursors.

This example is using nested cursors.

First we use a cursor to fetch the details of those employees who stay in London.

Then we open another cursor within this cursor to fetch the details of the orders of the current employee

Examples on cursors (Contd…)

```
OPEN orders_cursor
FETCH NEXT FROM orders_cursor INTO
  @orderid,@companyname,@orderdate
IF @@FETCH_STATUS <> 0
PRINT '       <<No Orders>>'
WHILE @@FETCH_STATUS = 0
BEGIN
        SELECT @message = '       '+convert(varchar(40),@orderid)+'
                    '+CONVERT(CHAR(40),@companyname)+'
          '+convert(VARCHAR(20),@orderdate)
  PRINT @message
  FETCH NEXT FROM orders_cursor INTO
  @orderid,@companyname,@orderdate
END
CLOSE orders_cursor
DEALLOCATE orders_cursor-- Get the next author.
FETCH NEXT FROM employees_cursor INTO @EmployeeId, @FirstName, @LastName
END
CLOSE employees_cursor
DEALLOCATE employees_cursor
GO
```


Then we display all the relevant details using PRINT function and relevant conversion functions as and when required.

And we CLOSE and DEALLOCATE the cursors when not required

Example on SCROLL cursor

- Example to create a SCROLL cursor to allow full scrolling capabilities through the LAST, PRIOR,RELATIVE, and ABSOLUTE options.

```
PRINT 'Printing data for employee class.'
SELECT LastName, FirstName FROM Employees ORDER BY LastName,
    FirstName
-- Declare the cursor.
DECLARE employees_cursor SCROLL CURSOR
FOR
    SELECT LastName, FirstName
    FROM Employees
    ORDER BY LastName, FirstName--Declare variables to fetch the values
OPEN employees_cursor
PRINT 'Fetching the last row in the cursor.'
FETCH LAST FROM employees_cursor
```


This example illustrates the scrolling capabilities of a SCROLL CURSOR through the LAST, PRIOR,RELATIVE, and ABSOLUTE options.

Example on SCROLL cursor (Contd…)

```
PRINT 'Fetch the row immediately prior to the current row in the cursor.'
FETCH PRIOR FROM employees_cursor

PRINT 'Fetch the second row in the cursor.'
FETCH ABSOLUTE 2 FROM employees_cursor

PRINT 'Fetch the row that is three rows after the current row.'
FETCH RELATIVE 3 FROM employees_cursor

PRINT 'Fetch the row that is two rows prior to the current row.'
FETCH RELATIVE -2 FROM employees_cursor

CLOSE employees_cursor
DEALLOCATE employees_cursor
GO
```

Module 14: Implementing Stored Procedures

- Overview
  - Introduction to Stored Procedures
  - Creating & Executing Stored Procedures
  - Passing parameters
  - Output parameters
  - Modifying and Dropping Stored Procedures
  - Handling Errors

Introduction to Stored Procedures

- Stored procedure is a named collection of Transact-SQL statements that is stored on the server.
- It gets stored within the database including its execution plan
- It is a method of encapsulating repetitive tasks.
- It support user-declared variables, conditional execution and other powerful programming features.

A stored procedure is a named collection of Transact-SQL statements that is stored on the server. Stored procedures are a method of encapsulating repetitive

tasks. Stored procedures support user-declared variables, conditional execution, and other powerful programming features.

Stored procedures in SQL Server can:

•Contain statements that perform operations in the database, including the ability to call other stored procedures.

•Accept input parameters.

•Return a status value to a calling stored procedure or batch to indicate success or failure (and the reason for failure).

•Return multiple values to the calling stored procedure or batch in the form of output parameters.

Creating & Executing Stored Procedures

- Syntax to create stored procedure:

  CREATE PROC [EDURE] *procedure_name* [{ *@parameter data_type* }[= *default*] [OUTPUT]] [,...n]

  [WITH { RECOMPILE | ENCRYPTION | RECOMPILE , ENCRYPTION }]

  AS *sql_statement* [...*n*]

- The following statements create a stored procedure that lists all overdue orders in the Northwind database.

  CREATE PROC OverdueOrders AS
  SELECT * FROM dbo.Orders WHERE
  RequiredDate < GETDATE() AND ShippedDate IS Null

- The following executes the procedure OverdueOrders:

  EXEC OverdueOrders

You can create a stored procedure in the current database only. except for temporary stored procedures, which are always created in the **tempdb** database. Creating a stored procedure is similar to creating a view. First, write and test the Transact-SQL statements that you want to include in the stored procedure. Then, if you receive the results that you expect, create the stored procedure.

Syntax:

```
CREATE PROC [ EDURE ] procedure_name [ { @parameter data_type }[ = default ] [ OUTPUT ] ] [ ,...n ]
[ WITH { RECOMPILE | ENCRYPTION | RECOMPILE , ENCRYPTION } ]
AS sql_statement [ ...n ]
```

Example:

```
CREATE PROC OverdueOrders AS
SELECT * FROM Orders
WHERE RequiredDate < GETDATE() AND ShippedDate IS Null
```

In above example we are creating stored procedure OverdueOrders, which finds RequiredDate value is less than GETDATE() and ShippedDate value is Null from Orders table.

You can execute stored procedure using EXEC followed by stored procedure name. The following executes the procedure OverdueOrders:

```
EXEC OverdueOrders
```

Passing parameters

- Syntax to create stored procedure passing parameters:

```
CREATE PROCEDURE [Year to Year Sales]
 @BeginningDate DateTime, @EndingDate DateTime AS
SELECT O.ShippedDate, O.OrderID, OS.Subtotal,
    DATENAME(yy,ShippedDate) AS Year
FROM ORDERS O INNER JOIN [Order Subtotals] OS
  ON O.OrderID = OS.OrderID
WHERE O.ShippedDate BETWEEN @BeginningDate AND @EndingDate
```

- The following executes the procedure [Year to Year Sales] passing parameters:

```
EXEC [Year to Year Sales] '1997-01-01','1998-01-01'
```


Input parameters allow information to be passed into a stored procedure. To define a stored procedure that accepts input parameters, you declare one or more variables as parameters in the CREATE PROCEDURE statement.

Syntax:

```
@parameter data_type [= default]
```

Consider the following facts and guidelines when you specify parameters:

•All incoming parameter values should be checked at the beginning of a stored procedure to trap missing and invalid values early.

•You should provide appropriate default values for a parameter.

If a default is defined, a user can execute the stored procedure without specifying a value for that parameter. You can set the value of a parameter by either passing the value to the stored procedure, by parameter name, or by position. You should not mix the different formats when you supply values.

Passing Values by Parameter Name

Specifying a parameter in an EXECUTE statement in the format *@parameter = value* is referred to as *passing by parameter name*. When you pass values by parameter name, the parameter values can be specified in any order, and you can omit parameters that allow null values or that have a default. The default value of a parameter, if defined for the parameter in the stored procedure, is used when:

•No value for the parameter is specified when the stored procedure is executed.

•The DEFAULT keyword is specified as the value for the parameter.

```
 [ [ EXEC [ UTE ] ]
{ [ @return_status = ] { procedure_name [ ;number ] | @procedure_name_var }
 [ [ @parameter = ] { value | @variable [ OUTPUT ] | [ DEFAULT ] ] [ ,...n ]
```

Output parameters

- This example creates a MathTutor stored procedure that calculates the product of two numbers and stores it in a OUTPUT parameter:

```
CREATE PROCEDURE MathTutor
  @m1 smallint, @m2 smallint, @result smallint OUTPUT AS
  SET @result = @m1* @m2
```

- The following executes the procedure MathTutor passing parameters and storing the value in answer variable passed as an OUTPUT parameter:

```
DECLARE @answer smallint
EXECUTE MathTutor 5,6, @answer OUTPUT
SELECT 'The result is: ', @answer
```


Stored procedures can return information to the calling stored procedure or client with output parameters (variables designated with the OUTPUT keyword). By using output parameters, any changes to the parameter that result from the execution of the stored procedure can be retained, even after the stored procedure completes execution.

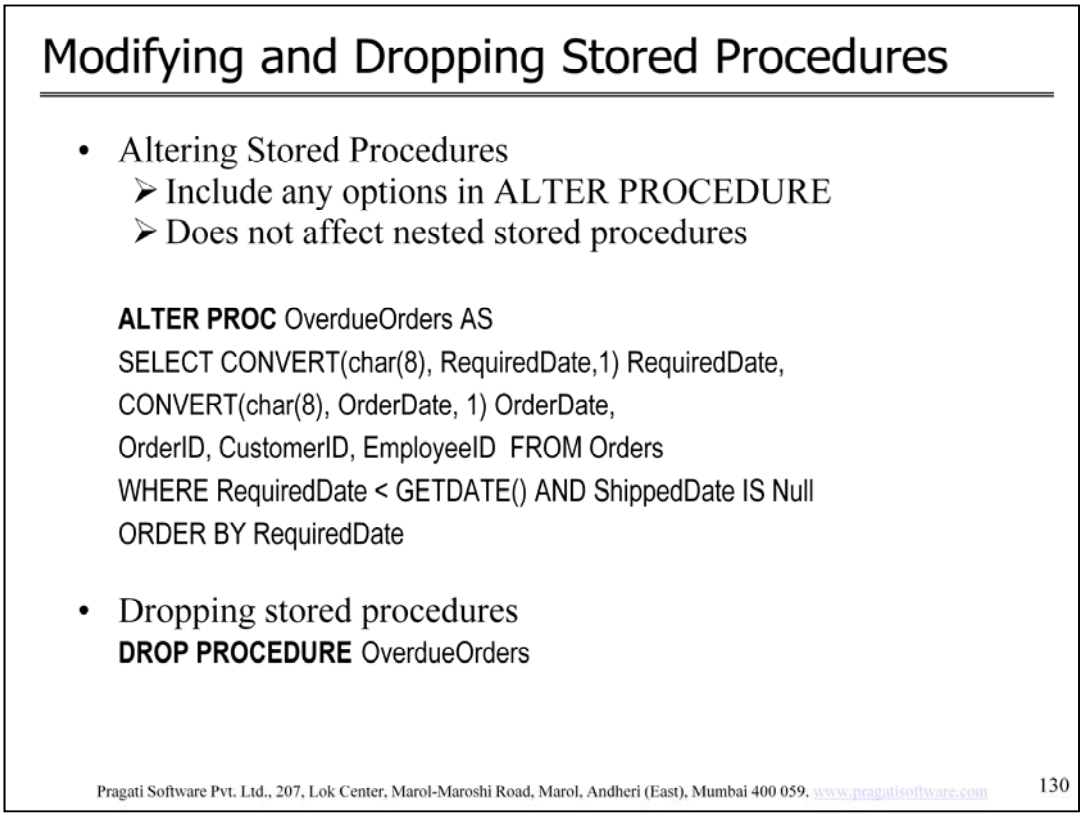
To use an output parameter, you must specify the OUTPUT keyword in both the CREATE PROCEDURE and EXECUTE statements. If the keyword OUTPUT is omitted when the stored procedure is executed, the stored procedure still executes but does not return a value. Output parameters have the following characteristics:

•The calling statement must contain a variable name to receive the return value.

•You can use the variable subsequently in additional Transact-SQL statements in the batch or the calling stored procedure.

•The parameter can be of any data type, except **text** or **image**.

•They can be cursor placeholders.



Stored procedures are often modified in response to requests from users or to changes in the underlying table definitions.

Altering Stored Procedures

To modify an existing stored procedure and retain permission assignments, use the ALTER PROCEDURE statement. SQL Server replaces the previous definition of the stored procedure when it is altered with ALTER PROCEDURE.

The following example modifies the **OverdueOrders** stored procedure to select only specific column names rather than all columns from the **Orders** table, as well as to sort the result set.

```
ALTER PROC OverdueOrders AS
SELECT CONVERT(char(8), RequiredDate, 1) RequiredDate,
  CONVERT(char(8), OrderDate, 1) OrderDate,
  OrderID, CustomerID, EmployeeID
  FROM Orders
WHERE RequiredDate < GETDATE() AND ShippedDate IS Null
ORDER BY RequiredDate
```

The following statement executes the **OverdueOrders** stored procedure.

```
EXEC OverdueOrders
```

Dropping Stored Procedures

Use the DROP PROCEDURE statement to remove user-defined stored procedures from the current database. Before you drop a stored procedure, execute the **sp_depends** stored procedure to determine whether objects depend on the stored procedure.

```
DROP PROCEDURE { procedure } [ ,...n ]
```

This example drops the **OverdueOrders** stored procedure.

```
DROP PROC OverdueOrders
```

Handling Errors

- RETURN Statement Exits Query or Procedure unconditionally

```
CREATE PROCEDURE dbo.GetOrders @CustomerID nchar (10) AS
SELECT OrderID, CustomerID, EmployeeID FROM [Orders Qry]
WHERE CustomerID = @CustomerID
RETURN (@@ROWCOUNT)
GO
declare @a int
exec @a = GetOrders 'VINET'
Select @a
```


To enhance the effectiveness of stored procedures, you should include error messages that communicate transaction status (success or failure) to the user. You should perform the task logic, business logic, and error checking *before* you begin transactions, and you should keep your transactions short. You can check the following in your error handling logic: return codes, SQL Server errors, and custom error messages.

RETURN Statement

The RETURN statement exits from a query or stored procedure unconditionally. It also can return an integer status value (return code). A return value of 0 indicates success. Return values 0 through -14 are currently in use, and return values from -15 through -99 are reserved for future use. If a user-defined return value is not provided, the SQL Server value is used. User-defined return values always take precedence over those that SQL Server supplies.

Handling Errors

- @@error Contains Error Number for Last Executed Statement
  - It is cleared and reset with each statement that is executed.
  - A value of 0 is returned if the statement executes successfully.
  - You can use the **@@error** to detect a specific error number or to exit a stored procedure conditionally.

```
DECLARE @Error int
INSERT INTO [Order Details] (OrderID, ProductID, UnitPrice, Quantity, Discount) VALUES
    (999999,11,10.00,10, 0) - - Bogus INSERT
SELECT @Error = @@ERROR /* Note that, after this statement,
                           @@Error will be reset to whatever error
                           number applies to this statement*/
PRINT '' - - Print out a blank separator line
PRINT 'The Value of @Error is ' + CONVERT(varchar, @Error)
- - The value of our holding variable is just what we would expect
PRINT 'The Value of @@ERROR is ' + CONVERT(varchar, @@ERROR)
- -The value of @@ERROR has been reset - it's back to zero
```


@@error

This contains the error number for the most recently executed Transact-SQL statement. It is cleared and reset with each statement that is executed. A value of 0 is returned if the statement executes successfully. You can use the **@@error** to detect a specific error number or to exit a stored procedure conditionally.

Handling Errors

- Create user-defined error messages with sp_addmessage stored procedure

  SP_ADDMESSAGE number, severity, "message"
  - The first parameter number is the ID of message (50001 onwards)
  - The second parameter, severity, is error severity (1-25)
  - The third parameter is the text of the error message

- Exec sp_addmessage 50001,10, 'Null values are not allowed'

Handling Errors

- RAISERROR Statement
  - Returns user-defined or system error message
  - Error code is the first parameter
  - Severity level is the second parameter
    - 0-18 can be used by any user. 19-25 should be used by sysadmin role members
  - Message state is the third parameter (pass any number as it is not used by SQL server)

```
ALTER PROCEDURE [Year to Year Sales]
@BeginningDate DateTime=NULL, @EndingDate DateTime=NULL AS
IF @BeginningDate IS NULL OR @EndingDate IS NULL
    BEGIN
      RAISERROR(50001,10,1)
      RETURN
    END
SELECT O.ShippedDate, O.OrderID, OS.Subtotal, DATENAME(yy,ShippedDate) AS Year
FROM ORDERS O INNER JOIN [Order Subtotals] OS  ON O.OrderID = OS.OrderID
WHERE O.ShippedDate BETWEEN @BeginningDate AND @EndingDate
GO

EXEC [Year to Year Sales]
```

Pragati Software Pvt. Ltd., 207, Lok Center, Marol-Maroshi Road, Marol, Andheri (East), Mumbai 400 059. www.pragatisoftware.com 134

RAISERROR Statement

The RAISERROR statement returns a user-defined error message and sets a system flag to record that an error has occurred. You must specify an error severity level and message state when using the RAISERROR statement.

Module 15: Implementing User-defined Functions

- Overview
  - What Is a User-defined Function?
  - Creating a User-defined Function
  - Altering and Dropping User-defined Functions
  - Using a Scalar User-defined Function
  - Using a Multi-Statement Table-valued Function
  - Using an In-Line Table-valued Function

What Is a User-defined Function?

- A UDF takes zero, or more, input parameters and returns either a *scalar* value or a table.
- Input parameters can be any data type except **timestamp**, **cursor**, or **table**.
- UDF's do not support output parameters.
- There are 3 types of UDFs:-
  - Scalar Functions
  - Multi-Statement Table-valued Functions
  - In-Line Table-valued Functions

A user-defined function takes zero, or more, input parameters and returns either a *scalar* value or a table. Input parameters can be any data type except **timestamp**, **cursor**, or **table**. User-defined functions do not support output parameters.

SQL Server supports three types of user-defined functions:

•**Scalar Functions**

A scalar function is similar to a built-in function.

•**Multi-Statement Table-valued Functions**

A multi-statement table-valued function returns a table built by one or more Transact-SQL statements and is similar to a stored procedure. Unlike a stored procedure, a multi-statement table-valued function can be referenced in the FROM clause of a SELECT statement as if it were a view.

•**In-Line Table-valued Functions**

An in-line table-valued function returns a table that is the result of a single SELECT statement. It is similar to a view but offers more flexibility than views in the use of parameters, and extends the features of indexed views.

Creating a User-defined Function

- You create user-defined functions by using the CREATE FUNCTION statement.

```
CREATE FUNCTION [ owner_name. ] function_name
( [ { @parameter_name scalar_parameter_data_type [ = default ] } [ ,...n
  ] ] )
RETURNS scalar_return_data_type [ AS ]
BEGIN
function_body
RETURN scalar_expression
END
```

- UDF's should have unique name
- It specifies the input parameters with their data types, the processing instructions, and the value returned with each data type.

You create a user-defined function in nearly the same way that you create a view or stored procedure.

Creating a Function:

You create user-defined functions by using the CREATE FUNCTION statement. Each fully qualified user-defined function name (database_name.owner_name.function_name) must be unique. The statement specifies the input parameters with their data types, the processing instructions, and the value returned with each data type.

Syntax:

```
CREATE FUNCTION [ owner_name. ] function_name
( [ { @parameter_name scalar_parameter_data_type [ = default ] } [
,...n ] ] )
RETURNS scalar_return_data_type
[ WITH < function_option> [,...n] ]
[ AS ]
BEGIN
function_body
RETURN scalar_expression
END
```

Creating a User-defined Function

- **CREATE FUNCTION** fn_NewRegion(@myinput nvarchar(30))
  RETURNS nvarchar(30)
  BEGIN
  IF @myinput IS NULL
  SET @myinput = 'Not Applicable'
  RETURN @myinput
  END

- When referencing a scalar user-defined function, specify the function owner and the function name in two-part syntax.
  SELECT LastName, City, **dbo.fn_NewRegion(**Region**)** AS Region, Country
  FROM dbo.Employees

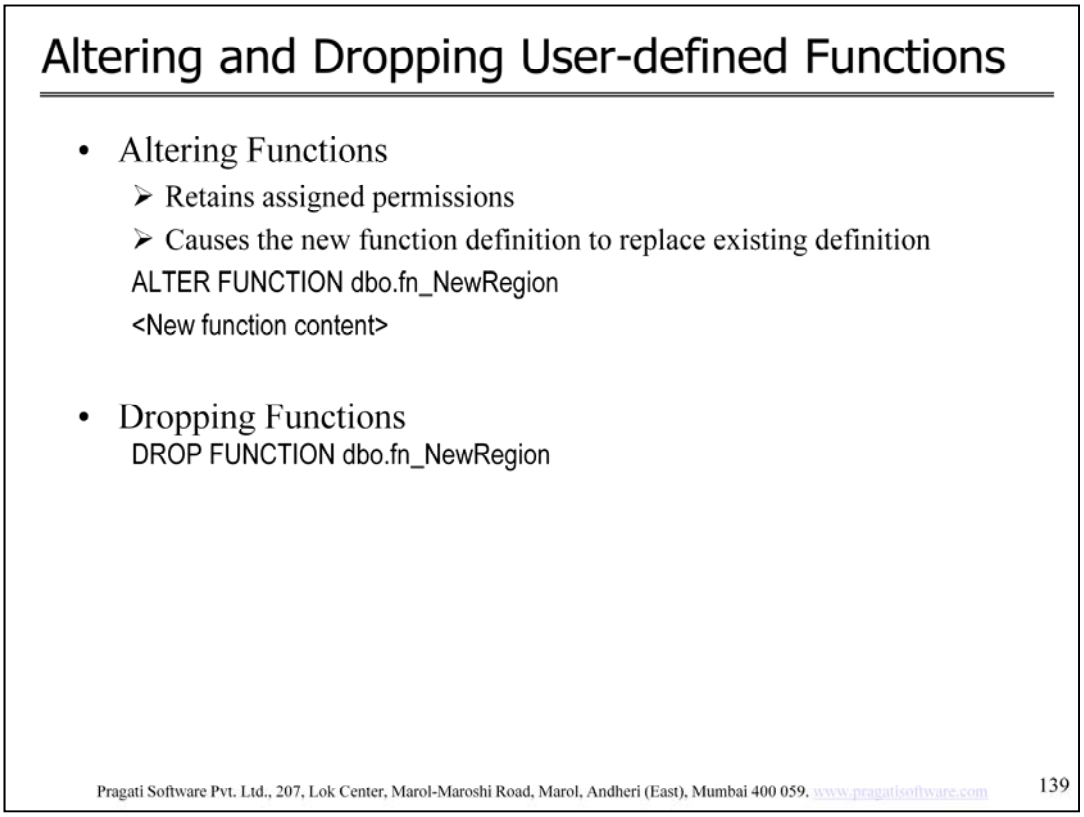
Pragati Software Pvt. Ltd., 207, Lok Center, Marol-Maroshi Road, Marol, Andheri (East), Mumbai 400 059. www.pragatisoftware.com 138

Above example creates a user-defined function to replace a NULL value with the words Not Applicable.

When referencing a scalar user-defined function, specify the function owner and the function name in two-part syntax.

```
SELECT LastName, City, dbo.fn_NewRegion(Region) AS Region,
Country FROM dbo.Employees
```

In example dbo is function owner and fn_NewRegion is function name.



You can alter and drop user-defined functions by using the ALTER FUNCTION statement. The benefit of altering a function instead of dropping and recreating it is the same as it is for views and procedures. The permissions on the function remain and immediately apply to the revised function.

Altering Functions

You modify a user-defined function by using the ALTER FUNCTION statement. This example shows how to alter a function.

```
ALTER FUNCTION dbo.fn_NewRegion
<New function content>
```

Dropping Functions

You drop a user-defined function by using the DROP FUNCTION statement. This example shows how to drop a function.

```
DROP FUNCTION dbo.fn_NewRegion
```

Using a Scalar User-defined Function

- RETURNS clause specifies data type
- Function is defined within a BEGIN and END block
- Return type is any data type except text, ntext, image, cursor, or timestamp
- Example

```
CREATE FUNCTION fn_DateFormat (@indate datetime, @separator char(1))
   RETURNS Nchar(20) AS
   BEGIN
        RETURN
        CONVERT(Nvarchar(20), datepart(mm,@indate)) + @separator
        + CONVERT(Nvarchar(20), datepart(dd, @indate)) + @separator
        + CONVERT(Nvarchar(20), datepart(yy, @indate))
    END
SELECT dbo.fn_DateFormat(GETDATE(), ':')
```


A scalar function returns a single data value of the type defined in a RETURNS clause. The body of the function, defined in a BEGIN…END block, contains the series of Transact-SQL statements that return the value. The return type can sbe any data type except **text**, **ntext**, **image**, **cursor**, or **timestamp**.

Using a Multi-Statement Table-valued Function

- BEGIN and END enclose multiple statements
- RETURNS clause names and defines the table
- Example:

```
CREATE FUNCTION fn_Employees    (@length nvarchar(9))
RETURNS  @fn_Employees  TABLE   (EmployeeID  int  PRIMARY  KEY  NOT  NULL,
   [Employee Name] Nvarchar(61) NOT NULL)  AS
BEGIN
  IF @length = 'ShortName'
    INSERT @fn_Employees SELECT EmployeeID, LastName FROM Employees
  ELSE IF @length = 'LongName'
    INSERT  @fn_Employees  SELECT  EmployeeID,   (FirstName  +  ' '  +  LastName)
    FROM Employees
RETURN
END
```

- You can call the function instead of a table or view:

```
SELECT * FROM dbo.fn_Employees('LongName')
```


A table-valued function (like a stored procedure) can use complex logic and multiple Transact-SQL statements to build a table. In the same way that you use a view, you can use a table-valued function in the FROM clause of a Transact-SQL statement.

When using a multi-statement table-valued function, consider the following facts:

•The BEGIN and END delimit the body of the function.

•The RETURNS clause specifies **table** as the data type returned.

•The RETURNS clause defines a name for the table and defines the format of the table. The scope of the return variable name is local to the function.

Using an In-Line Table-valued Function

- Content of the function is a SELECT statement
- BEGIN and END **do not** delimit the body of the function.
- RETURN specifies **table** as the data type returned.
- Format of the result set is set by the SELECT statement in the RETURN clause.

```
CREATE FUNCTION fn_CustomerNamesInRegion ( @RegionParameter nvarchar(30) ) RETURNS table AS
RETURN ( SELECT CustomerID, CompanyName FROM Northwind.dbo.Customers WHERE Region = @RegionParameter )
```

- Calling the Function Using a Parameter

```
SELECT * FROM fn_CustomerNamesInRegion('WA')
```

Pragati Software Pvt. Ltd., 207, Lok Center, Marol-Maroshi Road, Marol, Andheri (East), Mumbai 400 059. www.pragatisoftware.com 142

In-line user-defined functions return a table and are referenced in the FROM clause, just like a view. When using in-line user-defined functions, consider the following facts and guidelines:

•The RETURN clause contains a single SELECT statement in parentheses. The result set of the SELECT statement forms the table that the function returns. The SELECT statement used in an in-line function is subject to the same restrictions as SELECT statements used in views.

•BEGIN and END do not delimit the body of the function.

•RETURN specifies **table** as the data type returned.

•You do not have to define the format of a return variable, because it is set by the format of the result set of the SELECT statement in the RETURN clause.

Module 16: Implementing Triggers

- Overview
  - Introduction to Triggers
  - Defining Triggers
  - How Triggers Work
  - Examples of Triggers

Introduction to Triggers

- What is a Trigger?
- Uses of Triggers
- Considerations for Using Triggers

This section introduces triggers and describes when and how to use them.

What is a Trigger?

- A *trigger* is a stored procedure that executes when data in a specified table is modified.
- Users cannot circumvent triggers so you can use triggers to enforce complex business rules that maintain data integrity.
- Triggers cannot be called explicitly like stored procedures.

A trigger is a special kind of stored procedure that executes whenever an attempt is made to modify data in a table that the trigger protects. Triggers are tied to specific tables.

Associated with a Table Triggers are defined on a specific table, which is referred to as the trigger table.

Invoked Automatically When an attempt is made to insert, update, or delete data in a table, and a trigger for that particular action has been defined on the table, the trigger executes automatically. It cannot be circumvented.

Cannot Be Called Directly Unlike standard system-stored procedures, triggers cannot be called directly and do not pass or accept parameters.

Uses of Triggers

- Cascade changes through related tables in a database
- Enforce more complex data integrity than a CHECK constraint
  - Unlike CHECK constraints, triggers can reference columns in other tables.
- Define custom error messages
- Maintain de-normalized data
- Compare before and after states of data under modification

Triggers are best used to maintain low-level data integrity, *not* to return query results. The primary benefit of triggers is that they can contain complex processing logic. Triggers can cascade changes through related tables in a database, enforce more complex data integrity than a CHECK constraint, define custom error messages, maintain *de-normalized* data, and compare before and after states of data under modification.

Cascade Changes Through Related Tables in a Database: You can use a trigger to cascade updates and deletes through related tables in a database. For example, a delete trigger on the **Products** table in the **Northwind** database can delete matching rows in other tables that have rows that match the deleted **ProductID** values. A trigger does this by using the **ProductID** foreign key column as a way of locating rows in the **Order Details** table.

Enforce More Complex Data Integrity Than a CHECK Constraint: Unlike CHECK constraints, triggers can reference columns in other tables. For example, you could place an insert trigger on the **Order Details** table that checks the **UnitsInStock** column for that item in the **Products** table. The trigger could determine that when the **UnitsInStock** value is less than 10, that the maximum order amount is three items. This type of check references columns in other tables. Referencing columns in other tables is not permitted with a CHECK constraint.

Define Custom Error Messages: Occasionally, your implementation may benefit from custom error messages that indicate the status of an action. By using triggers, you can invoke predefined or dynamic custom error messages when certain conditions occur as a trigger executes.

Maintain De-normalized Data: Triggers can be used to maintain low-level data integrity in de-normalized database environments. Maintaining de-normalized data is different from cascading in that cascading typically refers to maintaining relationships between primary and foreign key values. De-normalized data is typically

contrived, derived, or redundant data values. You must use a trigger if:

•Referential integrity requires something that is not an exact match, such as maintaining derived data (year-to-date sales) or flagging columns (Y or N to indicate whether a product is available).

•You require customized messages and complex error messaging.

Compare Before and After States of Data Under Modification: Most triggers provide the ability to reference the changes that are made to the data by the INSERT, UPDATE, or DELETE statement. This allows you to reference the rows that are being affected by the modification statements inside the trigger.

Considerations for Using Triggers

- Triggers are reactive; constraints are proactive
- Constraints are checked first
- You must have permission to perform all statements that define triggers
- Cannot create triggers on temporary tables

Consider the following facts and guidelines when you work with triggers:

•Most triggers are reactive; constraints and the INSTEAD OF trigger are proactive. Triggers are executed after an INSERT, UPDATE, or DELETE statement is executed on the table in which the trigger is defined. For example, an UPDATE statement updates a row in a table, and then the trigger on that table executes automatically. Constraints are checked before an INSERT, UPDATE, or DELETE statement executes.

•Constraints are checked first. If constraints exist on the trigger table, they are checked prior to the trigger execution. If constraints are violated, the trigger does not execute.

•You must have permission to perform all trigger-defined statements. If permissions are denied to any portion of the Transact-SQL statements inside the trigger, the entire transaction is rolled back. Table owners cannot create AFTER triggers on views or temporary tables. Triggers can, however, reference views and temporary tables.

•Table owners can create INSTEAD OF triggers on views and tables, in which case INSTEAD OF triggers greatly extend the types of updates that a view can support. Triggers can handle multi-row actions.

An INSERT, UPDATE, or DELETE action that invokes a trigger can affect multiple rows. You can choose to:

•Process all of the rows together, in which case all affected rows must meet the trigger criteria for any action to occur.

•Allow conditional actions.

Defining Triggers

- Creating TRIGGER

```
CREATE TRIGGER [owner.] trigger_name  ON [owner.] table_name
[WITH ENCRYPTION] {FOR | AFTER | INSTEAD OF} {INSERT | UPDATE |
   DELETE} AS
[IF UPDATE (column_name).] [{AND | OR}UPDATE(column_name).]
 sql_statements}

CREATE TRIGGER Empl_Delete ON Employees FOR DELETE AS
IF (SELECT COUNT(*) FROM Deleted) > 1
BEGIN
    RAISERROR( 'You can delete only one employee at a time.', 16, 1)
     ROLLBACK TRANSACTION
END
```

- The following DELETE statement fires the trigger and prevents the transaction.

```
DELETE FROM Employees WHERE EmployeeID > 6
```

Pragati Software Pvt. Ltd., 207, Lok Center, Marol-Maroshi Road, Marol, Andheri (East), Mumbai 400 059. www.pragatisoftware.com 148

Create triggers by using the CREATE TRIGGER statement. The statement specifies the table on which a trigger is defined, the events for which the trigger executes, and the particular instructions for the trigger.

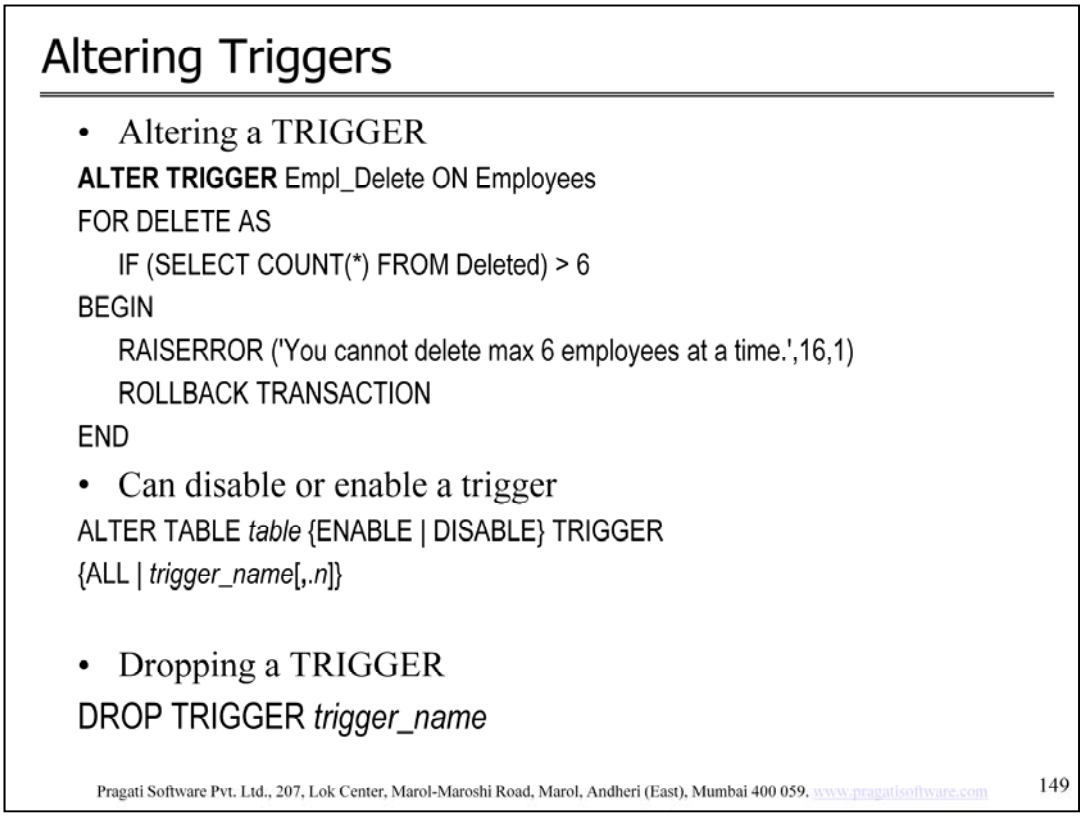
Syntax:

```
CREATE TRIGGER [owner.] trigger_name
ON [owner.] table_name [WITH ENCRYPTION]
{AFTER | INSTEAD OF} {INSERT | UPDATE | DELETE} AS
[IF UPDATE (column_name)...]  [{AND | OR} UPDATE (column_name)...]
  sql_statements}
```

When a FOR UPDATE action is specified, the IF UPDATE (*column_name*) clause can be used to focus action on a specific column that is updated. INSTEAD OF triggers cancel the triggering action and perform a new function instead.

To determine the tables with triggers, execute the **sp_depends** <*tablename*> system stored procedure. To view a trigger definition, execute the **sp_helptext**

<*triggername*> system stored procedure. To determine the triggers that exist on a specific table and their actions, execute the **sp_helptrigger** <*tablename*> system stored procedure.



Altering a Trigger

If you must change the definition of an existing trigger, you can alter it without having to drop it.

Changes the Definition Without Dropping the Trigger

The altered definition replaces the definition of the existing trigger with the new definition. Trigger action also can be altered. For example, if you create a trigger for INSERT and then change the action to UPDATE, the altered trigger executes whenever the table is updated.

With delayed name resolution, your trigger can reference tables and views that do not yet exist. If the object does not exist when a trigger is created, you receive a warning message and SQL Server updates the trigger definition immediately.

Disabling or Enabling a Trigger

You can disable or enable a specific trigger, or all triggers on a table. When a trigger is disabled, it is still defined for the table; however, when an INSERT, UPDATE, or DELETE statement is executed against the table, the actions in the trigger are not performed until the trigger is re-enabled. You can enable or disable triggers in the ALTER TABLE statement.

Syntax:

```
ALTER TABLE table {ENABLE | DISABLE} TRIGGER {ALL | trigger_name[,.n]}
```

Dropping a Trigger

You can remove a trigger by dropping it. Triggers are dropped automatically whenever their associated tables are dropped. Permission to drop a trigger defaults to the table owner and is non-transferable.

However, members of the system administrators (**sysadmin**) and database owner (**db_owner**) roles can drop any object by specifying the owner in the DROP TRIGGER statement.

Syntax:

```
DROP TRIGGER trigger_name
```

How an INSERT Trigger Works

- When an INSERT trigger is fired, new rows are added to both, the trigger table and the **inserted** table. The **inserted** table is a logical table that holds a copy of the rows that have been inserted.
- The trigger can examine the **inserted** table to determine whether, or how, the trigger actions should be carried out.

```
CREATE TRIGGER OrdDet_Insert ON [Order Details]
FOR INSERT AS
UPDATE P SET UnitsInStock = (P.UnitsInStock - I.Quantity)
FROM Products AS P INNER JOIN Inserted AS I
ON P.ProductID = I.ProductID
```


You can define a trigger to execute whenever an INSERT statement inserts data into a table.

When an INSERT trigger is fired, new rows are added to both the trigger table and the **inserted** table. The **inserted** table is a logical table that holds a copy of the rows that have been inserted. The **inserted** table contains the logged insert activity from the INSERT statement. The **inserted** table allows you to reference logged data from the initiating INSERT statement. The trigger can examine the **inserted** table to determine whether, or how, the trigger actions should be carried out. The rows in the **inserted** table are always duplicates of one or more rows in the trigger table.

All data modification activity (INSERT, UPDATE, and DELETE statements) is logged, but the information in the transaction log is unreadable. However, the **inserted** table allows you to reference the logged changes that the INSERT statement caused. Then you can compare the changes to the inserted data in order to verify them or take further action. You also can reference inserted data without having to store the information in variables

How a DELETE Trigger Works

- When a DELETE trigger is fired, deleted rows from the affected table are placed in a special **deleted** table. The **deleted** table is a logical table that holds a copy of the rows that have been deleted.
- Consider the following facts when you use the DELETE trigger:
  - When a row is appended to the **deleted** table, it no longer exists in the database table
  - A trigger that is defined for a DELETE action does not execute for the TRUNCATE TABLE statement

```
CREATE TRIGGER Category_Delete
ON Categories FOR DELETE AS
UPDATE P SET Discontinued = 1
FROM Products AS P INNER JOIN deleted AS d
ON P.CategoryID = d.CategoryID
```

Pragati Software Pvt. Ltd., 207, Lok Center, Marol-Maroshi Road, Marol, Andheri (East), Mumbai 400 059. www.pragatisoftware.com 151

When a DELETE trigger is fired, deleted rows from the affected table are placed in a special **deleted** table. The **deleted** table is a logical table that holds a copy of the rows that have been deleted. The **deleted** table allows you to reference logged data from the initiating DELETE statement.

Consider the following facts when you use the DELETE trigger:

•When a row is appended to the **deleted** table, it no longer exists in the database table; therefore, the **deleted** table and the database tables have no rows in common.

•Space is allocated from memory to create the **deleted** table. The **deleted** table is always in the cache.

•A trigger that is defined for a DELETE action does not execute for the TRUNCATE TABLE statement because TRUNCATE TABLE is not logged.

How an UPDATE Trigger Works

- An UPDATE statement can be thought of as two steps: the DELETE step that captures the *before image* of the data, and the INSERT step that captures the *after image* of the data.
- When an UPDATE statement is executed on a table that has a trigger defined on it, the original rows (before image) are moved into the **deleted** table, and the updated rows (after image) are inserted into the **inserted** table.
- Example

```
CREATE TRIGGER Emp_Update  ON Employees  FOR UPDATE  AS
IF UPDATE (EmployeeID)
BEGIN
 RAISERROR('Aborting Transaction...Employee ID cannot be changed.',10,1)
 ROLLBACK TRANSACTION
END
```


An UPDATE statement can be thought of as two steps: the DELETE step that captures the *before image* of the data, and the INSERT step that captures the *after image* of the data. When an UPDATE statement is executed on a table that has a trigger defined on it, the original rows (before image) are moved into the **deleted** table, and the updated rows (after image) are inserted into the **inserted** table.

The trigger can examine the **deleted** and **inserted** tables, as well as the updated table, to determine whether multiple rows have been updated and how the trigger actions should be carried out.

You can define a trigger to monitor data updates on a specific column by using the IF UPDATE statement. This allows the trigger to isolate activity easily for a specific column. When it detects that the specific column has been updated, it can take proper action, such as raising an error message that says that the column cannot be updated, or by processing a series of statements based on the newly updated column value.

How an INSTEAD OF Trigger Works

- You can specify an INSTEAD OF trigger on views.
- Each view is limited to one INSTEAD OF trigger for each triggering action (INSERT, UPDATE, or DELETE).
- You cannot create an INSTEAD OF trigger on views that have the WITH CHECK OPTION defined.

```
SELECT * INTO CustomersGer FROM Customers WHERE Customers.Country =
    'Germany'
SELECT * INTO CustomersMex FROM Customers WHERE Customers.Country =
    'Mexico'

CREATE VIEW CustomersView AS
SELECT * FROM CustomersGer UNION SELECT * FROM CustomersMex
```


You can specify an INSTEAD OF trigger on views. Each view is limited to one INSTEAD OF trigger for each triggering action (INSERT, UPDATE, or DELETE).

You cannot create an INSTEAD OF trigger on views that have the WITH CHECK OPTION defined.

How an INSTEAD OF Trigger Works (Contd..)

```
CREATE TRIGGER Customers_Upd2 ON CustomersView INSTEAD OF UPDATE AS
DECLARE @Country nvarchar(15)
SET @Country = (SELECT Country FROM Inserted)
IF @Country = 'Germany'
UPDATE CustomersGer SET CustomersGer.Phone = Inserted.Phone FROM
    CustomersGer JOIN Inserted ON CustomersGer.CustomerID = Inserted.CustomerID
ELSE
UPDATE CustomersMex SET CustomersMex.Phone = Inserted.Phone FROM
    CustomersMex JOIN Inserted ON CustomersMex.CustomerID = Inserted.CustomerID

--Test the trigger by updating the view
UPDATE CustomersView SET Phone = ' 030-007xxxx'  WHERE CustomerID = 'ALFKI'
SELECT CustomerID, Phone FROM CustomersView  WHERE CustomerID = 'ALFKI'
SELECT CustomerID, Phone FROM CustomersGer    WHERE CustomerID = 'ALFKI'
```

Examples of Triggers

- Enforcing data integrity

```
CREATE TRIGGER BackOrderList_Delete
 ON Products FOR UPDATE AS
IF (SELECT BO.ProductID FROM BackOrders AS BO JOIN
 Inserted AS I ON BO.ProductID = I.Product_ID  ) > 0
BEGIN
 DELETE BO FROM BackOrders AS BO INNER JOIN Inserted AS I
 ON BO.ProductID = I.ProductID
END
```

- Enforcing business rules

```
CREATE TRIGGER Product_Delete ON Products FOR DELETE AS
IF (Select Count (*)  FROM [Order Details] INNER JOIN deleted
   ON [Order Details].ProductID = Deleted.ProductID    ) > 0
BEGIN
  RAISERROR('Transaction Aborted....This product has order history.', 16, 1)
  ROLLBACK TRANSACTION
END
```


Enforcing Data Integrity:

You can use triggers to maintain data integrity by cascading changes to related tables throughout the database. The example shows how a trigger maintains data integrity on a **BackOrders** table. The **BackOrderList_delete** trigger maintains the list of products in the **BackOrders** table. When products are received, the UPDATE trigger on the **Products** table deletes records from a **BackOrders** table.

Enforcing Business Rules:

You can use triggers to enforce business rules that are too complex for the CHECK constraint. This includes checking the status of rows in other tables. For example, you may want to ensure that members' outstanding fines are paid before they are allowed to discontinue membership. The example creates a trigger that determines whether a product has order history. If it does, the DELETE is rolled back and the trigger returns a custom error message.