



# Data Management Using Microsoft SQL Server

Session: 8

Accessing Data



# Objectives

- Describe the SELECT statement, its syntax, and use
- Explain the various clauses used with SELECT
- State the use of ORDER BY clause
- Describe working with typed and untyped XML
- Explain the procedure to create, use, and view XML schemas
- Explain the use of Xquery to access XML data



# Introduction

- The `SELECT` statement is a core command used to access data in SQL Server 2012.
- XML allows developers to develop their own set of tags and makes it possible for other programs to understand these tags.
- XML is the preferred means for developers to store, format, and manage data on the Web.



# SELECT Statement 1-2

A table with its data can be viewed using the SELECT statement.

The SELECT statement retrieves rows and columns from one or more tables.

The output of the SELECT statement is another table called resultset.

The SELECT statement also joins two tables or retrieves a subset of columns from one or more tables.

The SELECT statement defines the columns to be used for a query.

# SELECT Statement 2-2

The syntax of `SELECT` statement can consist of a series of expressions separated by commas.

Each expression in the statement is a column in the resultset.

The columns appear in the same sequence as the order of the expression in the `SELECT` statement.

➤ The syntax for the `SELECT` statement is as follows:

## Syntax:

```
SELECT <column_name1>...<column_nameN> FROM <table_name>
```

where,

`table_name`: is the table from which the data will be displayed.

`<column_name1>...<column_nameN>`: are the columns that are to be displayed.

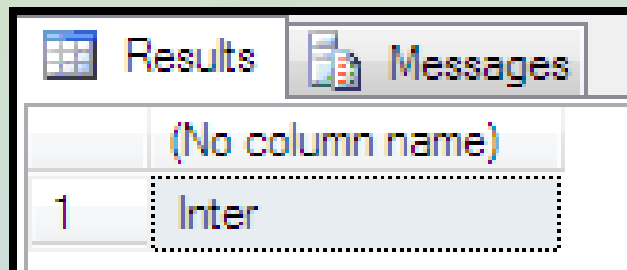
# SELECT Without FROM

Many SQL versions use FROM in their query, but in all the versions from SQL Server 2005, including SQL Server 2012, one can use SELECT statements without using the FROM clause.

Following code snippet demonstrates the use of SELECT statement without using the FROM clause:

```
SELECT LEFT('International',5)
```

- The code will display only the first five characters from the extreme left of the word 'International'.
- The output is shown in the following figure:



| (No column name) |       |
|------------------|-------|
| 1                | Inter |



# Displaying All Columns 1-2

The asterisk (\*) is used in the `SELECT` statement to retrieve all the columns from the table.

It is used as a shorthand to list all the column names in the tables named in the `FROM` clause.

➤ The syntax for selecting all columns is as follows:

## Syntax:

```
SELECT * FROM <table_name>
```

where,

\*: specifies all columns of the named tables in the `FROM` clause.

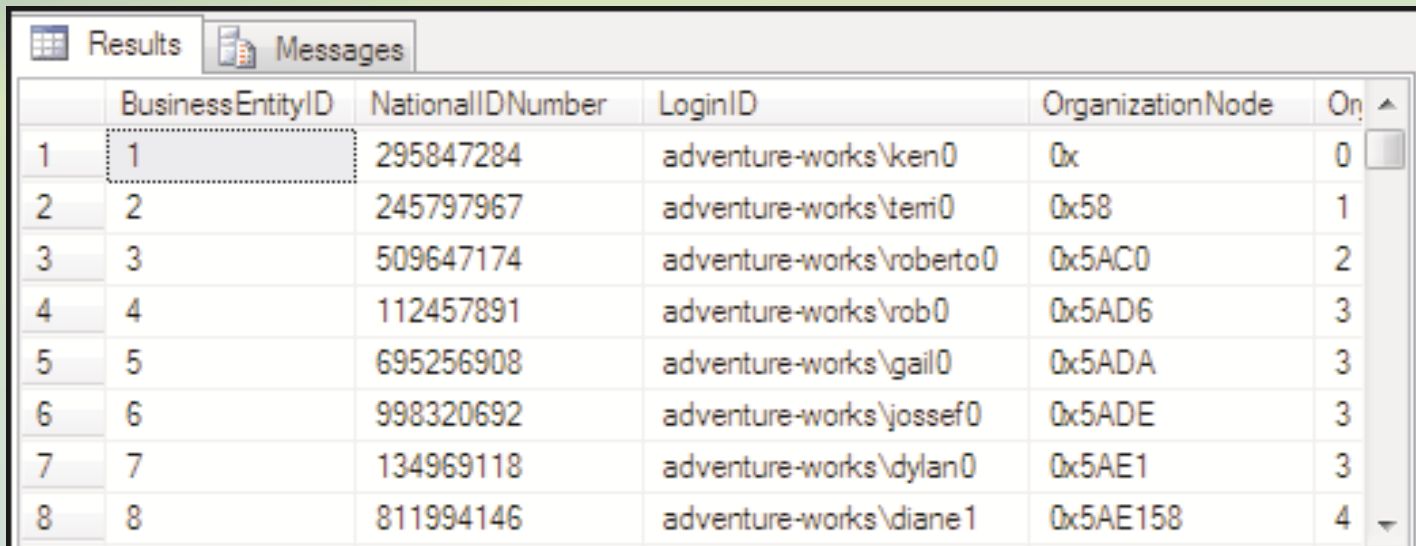
<table\_name>: is the name of the table from which the information is to be retrieved. It is possible to include any number of tables. When two or more tables are used, the row of each table is mapped with the row of others. This activity takes a lot of time if the data in the tables are huge. Hence, it is recommended to use this syntax with a condition.

# Displaying All Columns 2-2

- Following code snippet demonstrates the use of ' \* ' in the SELECT statement:

```
USE AdventureWorks2012
SELECT * FROM HumanResources.Employee
GO
```

- The partial output with some columns of HumanResources.Employee table is shown in the following figure:



|   | BusinessEntityID | NationalIDNumber | LoginID                  | OrganizationNode | On |
|---|------------------|------------------|--------------------------|------------------|----|
| 1 | 1                | 295847284        | adventure-works\ken0     | 0x               | 0  |
| 2 | 2                | 245797967        | adventure-works\temi0    | 0x58             | 1  |
| 3 | 3                | 509647174        | adventure-works\roberto0 | 0x5AC0           | 2  |
| 4 | 4                | 112457891        | adventure-works\rob0     | 0x5AD6           | 3  |
| 5 | 5                | 695256908        | adventure-works\gail0    | 0x5ADA           | 3  |
| 6 | 6                | 998320692        | adventure-works\jossef0  | 0x5ADE           | 3  |
| 7 | 7                | 134969118        | adventure-works\dylan0   | 0x5AE1           | 3  |
| 8 | 8                | 811994146        | adventure-works\diane1   | 0x5AE158         | 4  |



# Displaying Selected Columns 1-2

The `SELECT` statement displays or returns certain relevant columns that are chosen by the user or mentioned in the statement.

To display specific columns, the knowledge of the relevant column names in the table is needed.

- The syntax for selecting specific columns is as follows:

## Syntax:

```
SELECT <column_name1>..<column_nameN> FROM <table_name>
```

where,

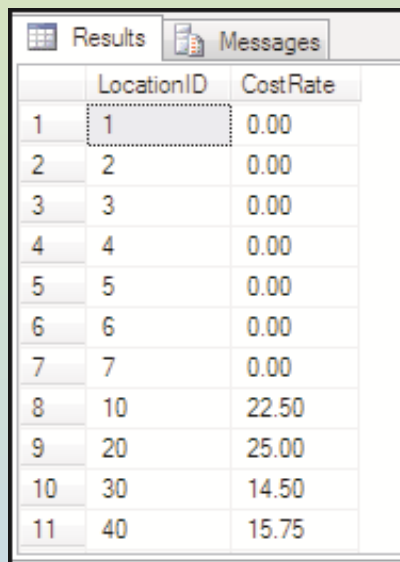
`<column_name1>..<column_nameN>`: are the columns that are to be displayed.

## Displaying Selected Columns 2-2

- For example, to display the cost rates in various locations from `Production.Location` table in AdventureWorks2012 database, the `SELECT` statement is as shown in the following code snippet:

```
USE AdventureWorks2012
SELECT LocationID, CostRate FROM Production.Location
GO
```

- Following figure shows `LocationID` and `CostRate` columns from AdventureWorks2012 database:



|    | LocationID | CostRate |
|----|------------|----------|
| 1  | 1          | 0.00     |
| 2  | 2          | 0.00     |
| 3  | 3          | 0.00     |
| 4  | 4          | 0.00     |
| 5  | 5          | 0.00     |
| 6  | 6          | 0.00     |
| 7  | 7          | 0.00     |
| 8  | 10         | 22.50    |
| 9  | 20         | 25.00    |
| 10 | 30         | 14.50    |
| 11 | 40         | 15.75    |



# Using Constants in Result Sets 1-2

Character string constants are used when character columns are joined.

They help in proper formatting or readability.

These constants are not specified as a separate column in the resultset.

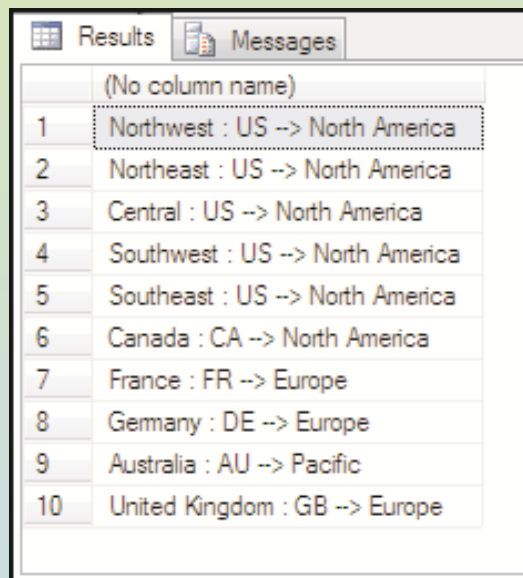
It is usually more efficient for an application to build the constant values into the results when they are displayed, rather than making use of the server to incorporate the constant values.

## Using Constants in Result Sets 2-2

- For example, to include ' : ' and '→' in the resultset so as to display the country name, country region code, and its corresponding group, the `SELECT` statement is shown in the following code snippet:

```
USE AdventureWorks2012
SELECT [Name] + ': ' + CountryRegionCode + '→' + [Group] FROM
    Sales.SalesTerritory
GO
```

- Following figure displays the country name, country region code, and corresponding group from `Sales.SalesTerritory` of AdventureWorks2012 database:



|    | (No column name)               |
|----|--------------------------------|
| 1  | Northwest : US → North America |
| 2  | Northeast : US → North America |
| 3  | Central : US → North America   |
| 4  | Southwest : US → North America |
| 5  | Southeast : US → North America |
| 6  | Canada : CA → North America    |
| 7  | France : FR → Europe           |
| 8  | Germany : DE → Europe          |
| 9  | Australia : AU → Pacific       |
| 10 | United Kingdom : GB → Europe   |



# Renaming ResultSet Column Names 1-2

When columns are displayed in the resultset they come with corresponding headings specified in the table.

These headings can be changed, renamed, or can be assigned a new name by using AS clause.

Therefore, by customizing the headings, they become more understandable and meaningful.

# Renaming ResultSet Column Names 2-2

- Following code snippet demonstrates how to display '**ChangedDate**' as the heading for **ModifiedDate** column in the **dbo.Individual** table, the **SELECT** statement:

```
USE CUST_DB  
SELECT ModifiedDate as 'ChangedDate' FROM dbo.Individual  
GO
```

- The output displays '**ChangedDate**' as the heading for **ModifiedDate** column in the **dbo.Individual** table.
- Following figure shows the original heading and the changed heading:

The figure displays two side-by-side screenshots of SQL Server query results. Both screenshots show a single row with the value '1981-02-02'. The left screenshot shows the column heading as 'ModifiedDate', while the right screenshot shows it as 'ChangedDate'. Both screenshots have 'Results' and 'Messages' tabs at the top.

|   | ModifiedDate |
|---|--------------|
| 1 | 1981-02-02   |

|   | ChangedDate |
|---|-------------|
| 1 | 1981-02-02  |



# Computing Values in ResultSet 1-2

A `SELECT` statement can contain mathematical expressions by applying operators to one or more columns.

It allows a resultset to contain values that do not exist in the base table, but which are calculated from the values stored in the base table.

For example, consider the table `Production.ProductCostHistory` from `AdventureWorks2012` database.

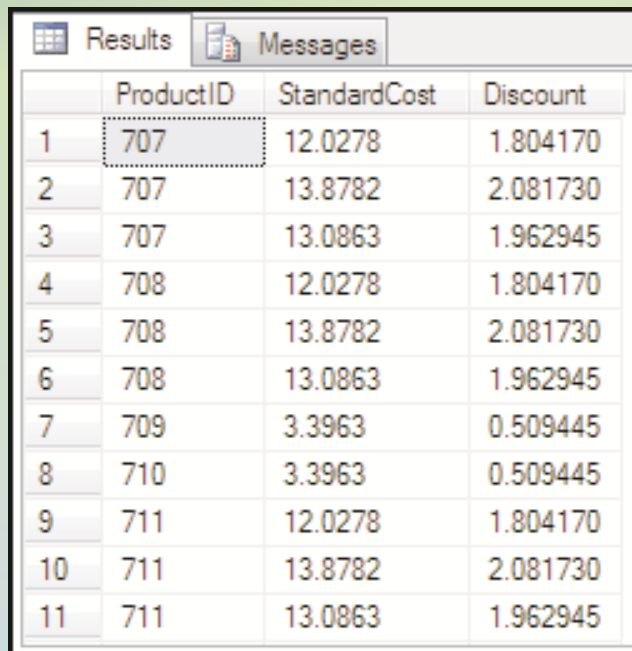
Consider the example where the production people decide to give 15% discount on the standard cost of all the products.

# Computing Values in ResultSet 2-2

- The discount amount does not exist, but can be calculated by executing the `SELECT` statement shown in the following code snippet:

```
USE AdventureWorks2012
SELECT ProductID,StandardCost,StandardCost * 0.15 as Discount FROM
Production.ProductCostHistory
GO
```

- Following figure shows the output where discount amount is calculated using `SELECT` statement:



|    | ProductID | StandardCost | Discount |
|----|-----------|--------------|----------|
| 1  | 707       | 12.0278      | 1.804170 |
| 2  | 707       | 13.8782      | 2.081730 |
| 3  | 707       | 13.0863      | 1.962945 |
| 4  | 708       | 12.0278      | 1.804170 |
| 5  | 708       | 13.8782      | 2.081730 |
| 6  | 708       | 13.0863      | 1.962945 |
| 7  | 709       | 3.3963       | 0.509445 |
| 8  | 710       | 3.3963       | 0.509445 |
| 9  | 711       | 12.0278      | 1.804170 |
| 10 | 711       | 13.8782      | 2.081730 |
| 11 | 711       | 13.0863      | 1.962945 |





# Using DISTINCT

The keyword `DISTINCT` prevents the retrieval of duplicate records.

It eliminates rows that are repeating from the resultset of a `SELECT` statement.

For example, if the `StandardCost` column is selected without using the `DISTINCT` keyword, it will display all the standard costs present in the table.

On using the `DISTINCT` keyword in the query, SQL Server will display every record of `StandardCost` only once as shown in the following code snippet:

```
USE AdventureWorks2012
SELECT DISTINCT StandardCost FROM Production.ProductCostHistory
GO
```

# Using TOP and PERCENT

The TOP keyword will display only the first few set of rows as a resultset.

The set of rows is either limited to a number or a percent of rows.

The TOP expression can also be used with other statements such as INSERT, UPDATE, and DELETE.

➤ The syntax for the TOP keyword is as follows:

## Syntax:

```
SELECT [ALL|DISTINCT] [TOP expression [PERCENT] [WITH TIES]]
```

where,

expression: is the number or the percentage of rows to be returned as the result.

PERCENT: returns the number of rows limited by percentage.

WITH TIES: is the additional number of rows that is to be displayed.

# SELECT with INTO 1-3

The INTO clause creates a new table and inserts rows and columns listed in the SELECT statement into it.

INTO clause also inserts existing rows into the new table.

In order to execute this clause with the SELECT statement, the user must have the permission to CREATE TABLE in the destination database.

➤ The syntax for the SELECT statement is as follows:

## Syntax:

```
SELECT <column_name1>..<column_nameN> [INTO new_table] FROM table_list
```

where,

new\_table: is the name of the new table that is to be created.



## SELECT with INTO 2-3

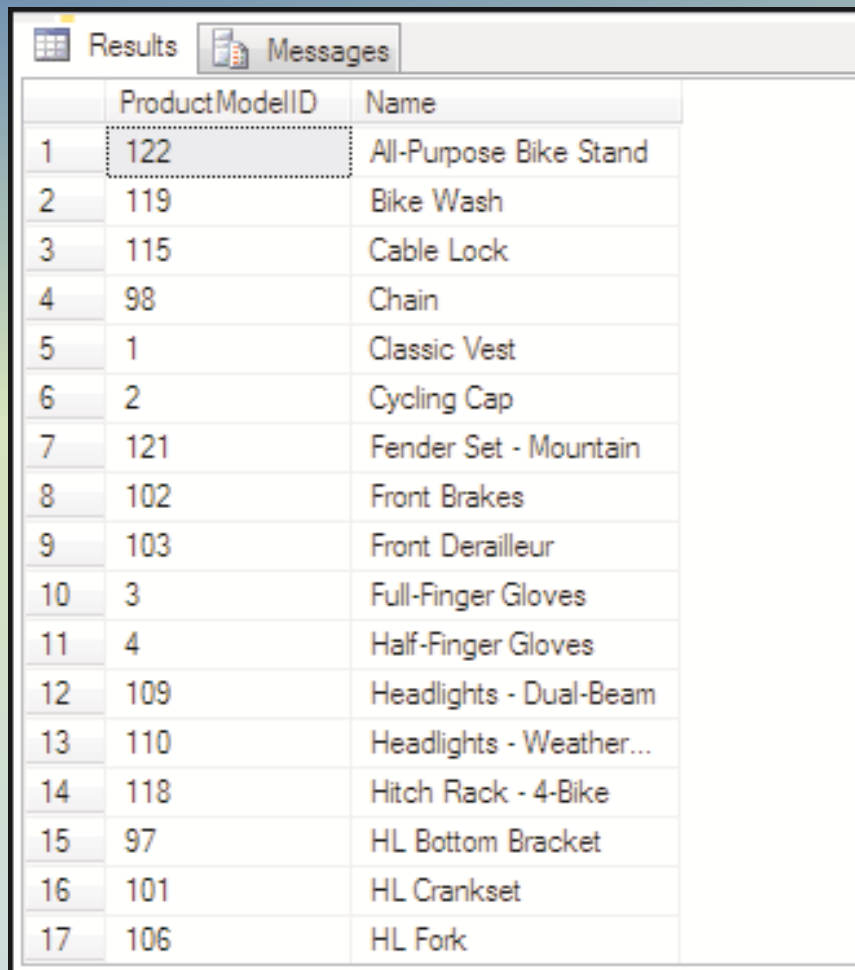
- Following code snippet uses an INTO clause which creates a new table `Production.ProductName` with details such as the product's ID and its name from the table `Production.ProductModel`:

```
USE AdventureWorks2012
SELECT ProductModelID,Name INTO Production.ProductName FROM
Production.ProductModel
GO
```

- After executing the code, a message stating '(128 row(s) affected)' is displayed.

## SELECT with INTO 3-3

- If a query is written to display the rows of the new table, the output will be as shown in the following figure:



The screenshot shows a SQL Server Results window with a table containing 17 rows of data. The table has two columns: ProductModelID and Name. The first row is highlighted with a dashed border, indicating it is the current row. The table is titled 'Results' and 'Messages' are visible in the top right corner.

|    | ProductModelID | Name                    |
|----|----------------|-------------------------|
| 1  | 122            | All-Purpose Bike Stand  |
| 2  | 119            | Bike Wash               |
| 3  | 115            | Cable Lock              |
| 4  | 98             | Chain                   |
| 5  | 1              | Classic Vest            |
| 6  | 2              | Cycling Cap             |
| 7  | 121            | Fender Set - Mountain   |
| 8  | 102            | Front Brakes            |
| 9  | 103            | Front Derailleur        |
| 10 | 3              | Full-Finger Gloves      |
| 11 | 4              | Half-Finger Gloves      |
| 12 | 109            | Headlights - Dual-Beam  |
| 13 | 110            | Headlights - Weather... |
| 14 | 118            | Hitch Rack - 4-Bike     |
| 15 | 97             | HL Bottom Bracket       |
| 16 | 101            | HL Crankset             |
| 17 | 106            | HL Fork                 |

# SELECT with WHERE 1-8

The **WHERE** clause with **SELECT** statement is used to conditionally select or limit the records retrieved by the query.

A **WHERE** clause specifies a **Boolean** expression to test the rows returned by the query.

The row is returned if the expression is true and is discarded if it is false.

➤ The syntax for the **SELECT** statement is as follows:

## Syntax:

```
SELECT <column_name1>...<column_nameN> FROM <table_name> WHERE <
search_condition>]
```

where,

**search\_condition**: is the condition to be met by the rows.

## SELECT with WHERE 2-8

- Following table shows the different operators that can be used with the `WHERE` clause:

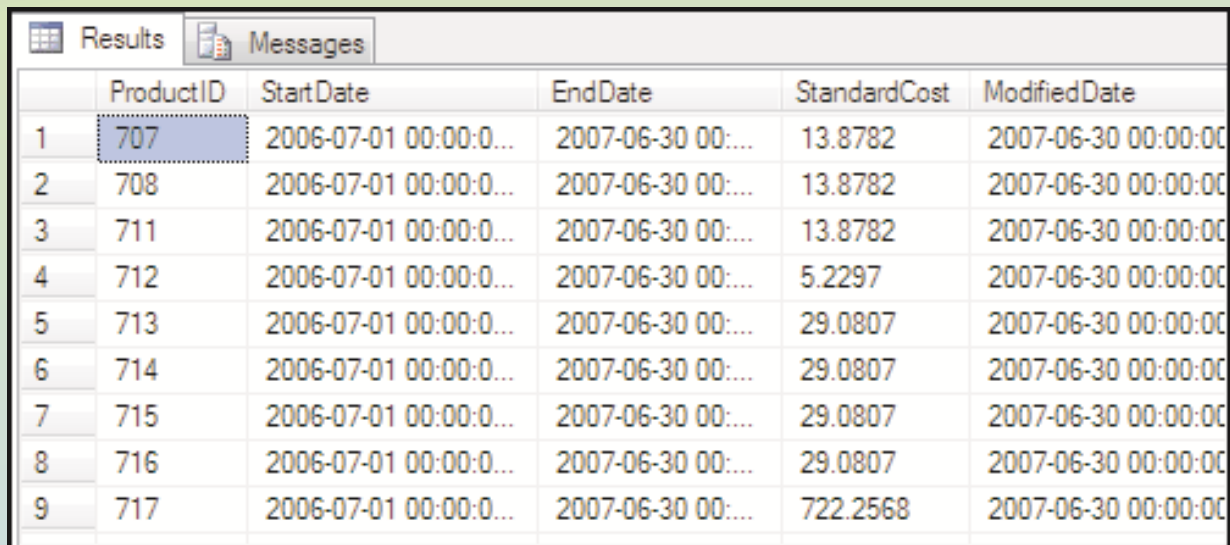
| Operator | Description                   |
|----------|-------------------------------|
| =        | Equal to                      |
| < >      | Not equal to                  |
| >        | Greater than                  |
| <        | Less than                     |
| > =      | Greater than or equal to      |
| < =      | Less than or equal to         |
| !        | Not                           |
| BETWEEN  | Between a range               |
| LIKE     | Search for an ordered pattern |
| IN       | Within a range                |

# SELECT with WHERE 3-8

Following code snippet demonstrates the equal to operator with WHERE clause to display data with EndDate 6/30/2007 12:00:00 AM:

```
USE AdventureWorks2012
SELECT * FROM Production.ProductCostHistory WHERE EndDate = '6/30/2007
12:00:00 AM'
GO
```

➤ The output SELECT with WHERE clause is shown in the following figure:



|   | ProductID | StartDate             | EndDate           | StandardCost | ModifiedDate        |
|---|-----------|-----------------------|-------------------|--------------|---------------------|
| 1 | 707       | 2006-07-01 00:00:0... | 2007-06-30 00:... | 13.8782      | 2007-06-30 00:00:00 |
| 2 | 708       | 2006-07-01 00:00:0... | 2007-06-30 00:... | 13.8782      | 2007-06-30 00:00:00 |
| 3 | 711       | 2006-07-01 00:00:0... | 2007-06-30 00:... | 13.8782      | 2007-06-30 00:00:00 |
| 4 | 712       | 2006-07-01 00:00:0... | 2007-06-30 00:... | 5.2297       | 2007-06-30 00:00:00 |
| 5 | 713       | 2006-07-01 00:00:0... | 2007-06-30 00:... | 29.0807      | 2007-06-30 00:00:00 |
| 6 | 714       | 2006-07-01 00:00:0... | 2007-06-30 00:... | 29.0807      | 2007-06-30 00:00:00 |
| 7 | 715       | 2006-07-01 00:00:0... | 2007-06-30 00:... | 29.0807      | 2007-06-30 00:00:00 |
| 8 | 716       | 2006-07-01 00:00:0... | 2007-06-30 00:... | 29.0807      | 2007-06-30 00:00:00 |
| 9 | 717       | 2006-07-01 00:00:0... | 2007-06-30 00:... | 722.2568     | 2007-06-30 00:00:00 |



# SELECT with WHERE 4-8

- All queries in SQL use single quotes to enclose the text values.
- For example, consider the following query, which retrieves all the records from `Person.Address` table having `Bothell` as city.
- Following code snippet demonstrates the equal to operator with `WHERE` clause to display data with address having `Bothell` city.

```
USE AdventureWorks2012
SELECT DISTINCT StandardCost FROM Production.ProductCostHistory
GO
```

- The output of the query is shown in the following figure:

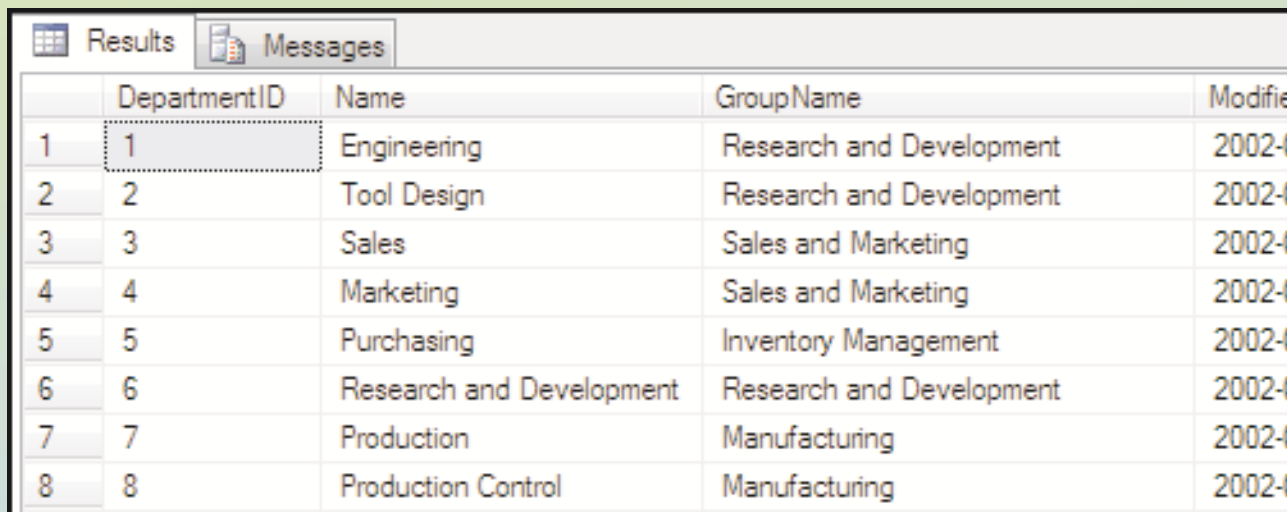
|   | AddressID | AddressLine1         | AddressLine2 | City    | StateProvinceID | PostalCode |
|---|-----------|----------------------|--------------|---------|-----------------|------------|
| 1 | 5         | 1226 Shoe St.        | NULL         | Bothell | 79              | 98011      |
| 2 | 11        | 1318 Lasalle Street  | NULL         | Bothell | 79              | 98011      |
| 3 | 6         | 1399 Firestone Drive | NULL         | Bothell | 79              | 98011      |
| 4 | 18        | 1873 Lion Circle     | NULL         | Bothell | 79              | 98011      |
| 5 | 40        | 1902 Santa Cruz      | NULL         | Bothell | 79              | 98011      |
| 6 | 1         | 1970 Napa Ct.        | NULL         | Bothell | 79              | 98011      |
| 7 | 10        | 250 Race Court       | NULL         | Bothell | 79              | 98011      |
| 8 | 868       | 25111 228th St Sw    | NULL         | Bothell | 79              | 98011      |
| 9 | 19        | 3148 Rose Street     | NULL         | Bothell | 79              | 98011      |

# SELECT with WHERE 5-8

- Numeric values are not enclosed within any quotes as shown in the following code snippet:

```
USE AdventureWorks2012
SELECT * FROM HumanResources.Department WHERE DepartmentID < 10
GO
```

- The query displays all those records where the value in `DepartmentID` is less than 10.
- The output of the query is shown in the following figure:



The screenshot shows a SQL Server Enterprise Manager window with two tabs: 'Results' and 'Messages'. The 'Results' tab is active, displaying a table with 5 columns: 'DepartmentID', 'Name', 'GroupName', and 'ModifiedDate'. The table contains 8 rows of data. The first row is highlighted with a dashed border.

|   | DepartmentID | Name                     | GroupName                | ModifiedDate |
|---|--------------|--------------------------|--------------------------|--------------|
| 1 | 1            | Engineering              | Research and Development | 2002-1       |
| 2 | 2            | Tool Design              | Research and Development | 2002-1       |
| 3 | 3            | Sales                    | Sales and Marketing      | 2002-1       |
| 4 | 4            | Marketing                | Sales and Marketing      | 2002-1       |
| 5 | 5            | Purchasing               | Inventory Management     | 2002-1       |
| 6 | 6            | Research and Development | Research and Development | 2002-1       |
| 7 | 7            | Production               | Manufacturing            | 2002-1       |
| 8 | 8            | Production Control       | Manufacturing            | 2002-1       |

# SELECT with WHERE 6-8

- WHERE clause can also be used with wildcard characters as shown in the following table:

| Wildcard | Description  | Example  |
|----------|--|--|
| _        | It will display a single character   | SELECT * FROM Person.<br>Contact WHERE Suffix LIKE<br>'Jr_ '                       |
| %        | It will display a string of any length   | SELECT * FROM Person.<br>Contact WHERE LastName<br>LIKE 'B% '                      |
| [ ]      | It will display a single character within the range enclosed in the brackets       | SELECT * FROM Sales.<br>CurrencyRate WHERE<br>ToCurrencyCode LIKE<br>'C[AN] [DY] ' |
| [ ^ ]    | It will display any single character not within the range enclosed in the brackets | SELECT * FROM Sales.<br>CurrencyRate WHERE<br>ToCurrencyCode LIKE<br>'A[^R] [^S] ' |

- All wildcard characters are used along with LIKE keyword to make the query accurate and specific.



# SELECT with WHERE 7-8

WHERE clause also uses logical operators such as AND, OR, and NOT. These operators are used with search conditions in WHERE clauses.

AND operator joins two or more conditions and returns TRUE only when both the conditions are TRUE.

So, it returns all the rows from the tables where both the conditions that are listed are true. Following code snippet demonstrates the use of AND operator:

```
USE AdventureWorks2012
SELECT * FROM Sales.CustomerAddress WHERE AddressID > 900 AND
AddressTypeID = 5
GO
```

# SELECT with WHERE 8-8

- OR operator returns TRUE and displays all the rows if it satisfies any one of the conditions. Following code snippet demonstrates the use of OR operator:

```
USE AdventureWorks2012
SELECT * FROM Sales.CustomerAddress WHERE AddressID < 900 OR
AddressTypeID = 5
GO
```

- The query will display all the rows whose AddressID is less than 900 or whose AddressTypeID is equal to five.
- The NOT operator negates the search condition.
- Following code snippet demonstrates the use of NOT operator:

```
USE AdventureWorks2012
SELECT * FROM Sales.CustomerAddress WHERE NOT AddressTypeID = 5
GO
```

- The code will display all the records whose AddressTypeID is not equal to 5.
- Multiple logical operators in a single SELECT statement can be used.
- When more than one logical operator is used, NOT is evaluated first, then AND, and finally OR.

# GROUP BY Clause 1-2

The GROUP BY clause partitions the resultset into one or more subsets. Each subset has values and expressions in common.

If an aggregate function is used in the GROUP BY clause, the resultset produces single value per aggregate.

Every grouped column restricts the number of rows of the resultset. For every grouped column, there is only one row.

The GROUP BY clause can have more than one grouped column. The syntax for GROUP BY clause is as follows:

## Syntax:

```
SELECT <column_name1>..<column_nameN> FROM <table_name> GROUP BY <column_name>
```

where,

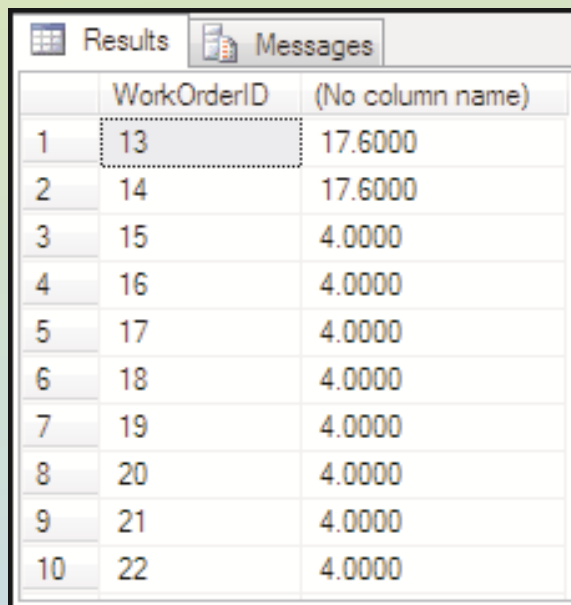
column\_name1: is the name of the column according to which the resultset should be grouped.

# GROUP BY Clause 2-2

- For example, consider that if the total number of resource hours has to be found for each work order, the query in the following code snippet would retrieve the resultset:

```
USE AdventureWorks2012
SELECT WorkOrderID, SUM(ActualResourceHrs) FROM
Production.WorkOrderRouting GROUP BY WorkOrderID
GO
```

- The output is shown in the following figure:



The screenshot shows a SQL Server Results window with two tabs: 'Results' and 'Messages'. The 'Results' tab is active, displaying a table with three columns: 'WorkOrderID' and '(No column name)'. The table contains 10 rows of data, with the first row highlighted. The 'WorkOrderID' column contains values from 13 to 22, and the '(No column name)' column contains the sum of 'ActualResourceHrs' for each work order.

|    | WorkOrderID | (No column name) |
|----|-------------|------------------|
| 1  | 13          | 17.6000          |
| 2  | 14          | 17.6000          |
| 3  | 15          | 4.0000           |
| 4  | 16          | 4.0000           |
| 5  | 17          | 4.0000           |
| 6  | 18          | 4.0000           |
| 7  | 19          | 4.0000           |
| 8  | 20          | 4.0000           |
| 9  | 21          | 4.0000           |
| 10 | 22          | 4.0000           |

# Clauses and Statements 1-7

- Microsoft SQL Server 2012 provides enhanced query syntax elements for more powerful data accessing and processing.

## Common Table Expression (CTE) in SELECT and INSERT statement

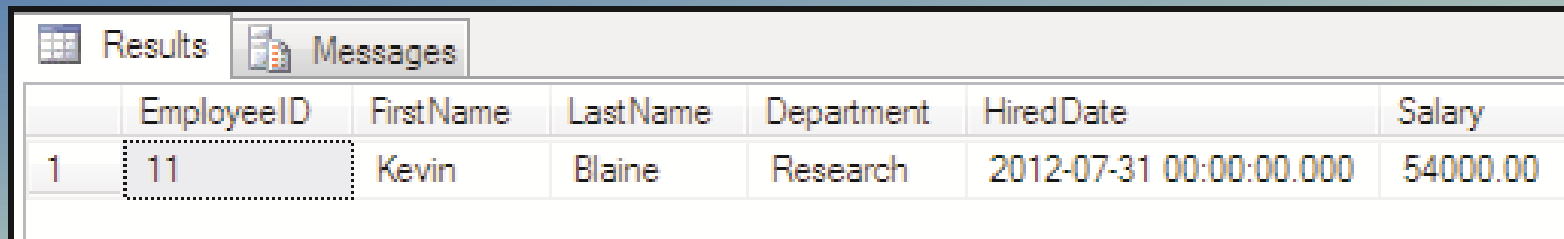
- A CTE is a named temporary resultset based on the regular SELECT and INSERT query.
- Following code snippet demonstrates the use of CTE in INSERT statement:

```
USE CUST_DB
CREATE TABLE NewEmployees (EmployeeID smallint, FirstName char(10),
LastName char(10), Department varchar(50), HiredDate datetime, Salary
money );
INSERT INTO NewEmployees
VALUES(11,'Kevin','Blaine', 'Research', '2012-07-31', 54000);
WITH EmployeeTemp (EmployeeID,FirstName,LastName,Department,
HiredDate,Salary)
AS
(
SELECT * FROM NewEmployees
)
SELECT * FROM EmployeeTemp
```



# Clauses and Statements 2-7

- The query inserts a new row for the `NewEmployees` table and transfers the temporary resultset to `EmployeeTemp` as shown in the following figure:



The screenshot shows the 'Results' tab in SQL Server Enterprise Manager. It displays a single row of data from the `EmployeeTemp` table. The columns are `EmployeeID`, `FirstName`, `LastName`, `Department`, `HiredDate`, and `Salary`. The row contains the values: 11, Kevin, Blaine, Research, 2012-07-31 00:00:00.000, and 54000.00. The `EmployeeID` cell is highlighted with a dashed border.

|   | EmployeeID | FirstName | LastName | Department | HiredDate               | Salary   |
|---|------------|-----------|----------|------------|-------------------------|----------|
| 1 | 11         | Kevin     | Blaine   | Research   | 2012-07-31 00:00:00.000 | 54000.00 |

## OUTPUT clause in INSERT and UPDATE statements

- The `OUTPUT` clause returns information about rows affected by an `INSERT` statement and an `UPDATE` statement.
- Following code snippet demonstrates how to use `UPDATE` statement with an `INSERT` statement:

```
USE CUST_DB;  
GO
```

# Clauses and Statements 3-7

```
CREATE TABLE dbo.table_3
(
id INT,
employee VARCHAR(32)
)
go
INSERT INTO dbo.table_3 VALUES
(1, 'Matt')
, (2, 'Joseph')
, (3, 'Renny')
, (4, 'Daisy');
GO
DECLARE @updatedTable TABLE
(
id INT, olddata_employee VARCHAR(32), newdata_employee VARCHAR(32)
);
UPDATE dbo.table_3
Set employee= UPPER(employee)
```

# Clauses and Statements 4-7

OUTPUT

```
inserted.id,  
deleted.employee,  
inserted.employee  
INTO @updatedTable  
SELECT * FROM @updatedTable
```

- The output where rows are affected by an INSERT statement and an UPDATE statement is shown in the following figure:

| Results |    | Messages         |                  |
|---------|----|------------------|------------------|
|         | id | olddata_employee | newdata_employee |
| 1       | 1  | Matt             | MATT             |
| 2       | 2  | Joseph           | JOSEPH           |
| 3       | 3  | Renny            | RENNY            |
| 4       | 4  | Daisy            | DAISY            |

# Clauses and Statements 5-7

## .WRITE clause

- .WRITE clause is used in an UPDATE statement to replace a value in a column having large value data type.
- The syntax for the .WRITE clause is as follows:

### Syntax:

```
.WRITE(expression, @offset, @Length)
```

where,

`expression`: is the character string which is to be placed into the large value data type column.

`@offset`: is the starting value (units) where the replacement is to be done.

`@Length`: is the length of the portion in the column, starting from `@offset` that is replaced by `expression`.

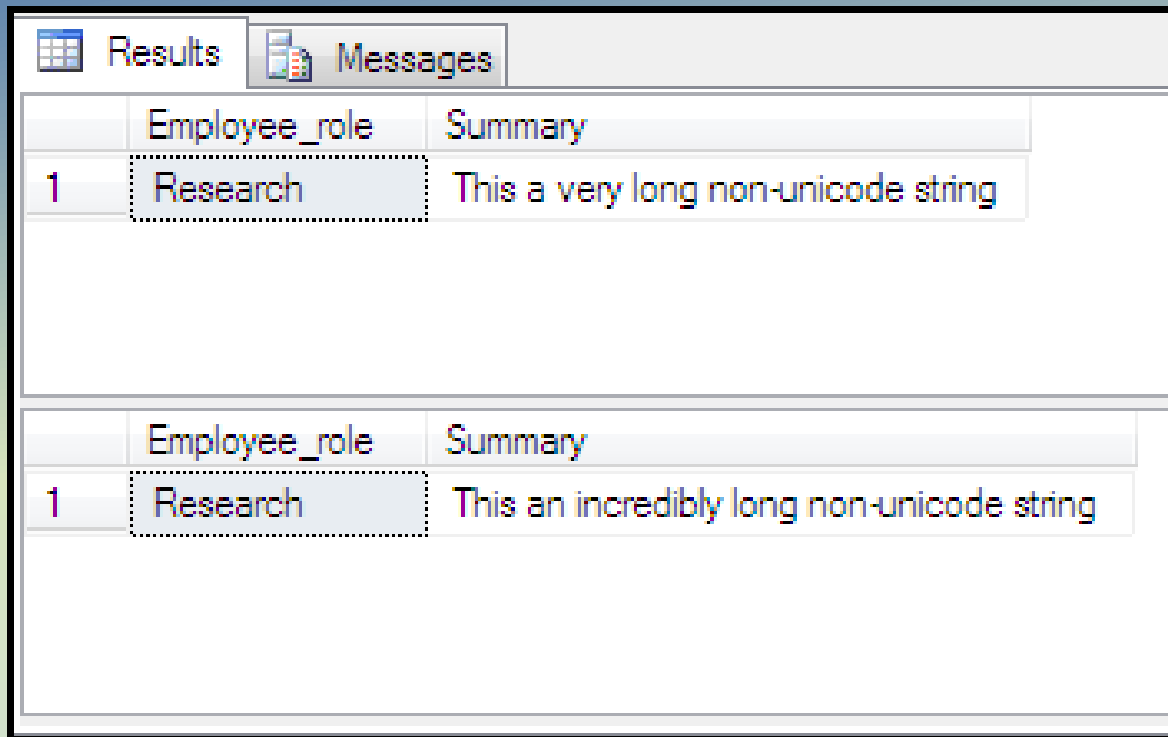
# Clauses and Statements 6-7

- Following code snippet demonstrates how .WRITE clause is used in UPDATE statement:

```
USE CUST_DB;
GO
CREATE TABLE dbo.table_5
(
Employee_role VARCHAR(max),
Summary VARCHAR(max)
)
INSERT INTO dbo.table_5(Employee_role, Summary) VALUES ('Research', 'This
a very long non-unicode string')
SELECT *FROM dbo.table_5
UPDATE dbo.table_5 SET Summary .WRITE('n incredibly', 6,5)
WHERE Employee_role LIKE 'Research'
SELECT *FROM dbo.table_5
```

# Clauses and Statements 7-7

- Following figure displays the output of .WRITE clause query:



|   | Employee_role | Summary                             |
|---|---------------|-------------------------------------|
| 1 | Research      | This a very long non-unicode string |

|   | Employee_role | Summary                                    |
|---|---------------|--|
| 1 | Research      | This an incredibly long non-unicode string |

# ORDER BY Clause 1-2

It specifies the order in which the columns should be sorted in a resultset.

It sorts query results by one or more columns. A sort can be in either ascending (ASC) or descending (DESC) order.

By default, records are sorted in an ASC order. To switch to the descending mode, use the optional keyword DESC.

When multiple fields are used, SQL Server considers the leftmost field as the primary level of sort and others as lower levels of sort.

## Syntax:

```
SELECT <column_name> FROM <table_name> ORDER BY column_name> {ASC|DESC}
```

# ORDER BY Clause 2-2

- The SELECT statement in the following code snippet sorts the query results on the SalesLastYear column of the Sales.SalesTerritory table:

```
USE AdventureWorks2012
SELECT * FROM Sales.SalesTerritory ORDER BY SalesLastYear
GO
```

- The output is shown in the following figure:



The screenshot shows the SQL Server Enterprise Manager interface with the 'Results' tab selected. The query results are displayed in a table with 8 columns: TerritoryID, Name, Country..., Group, Sales..., SalesLastYear, and Cos. The results are sorted by SalesLastYear in descending order. The first row (TerritoryID 8) is highlighted with a blue selection box.

|   | TerritoryID | Name           | Country... | Group         | Sales... | SalesLastYear | Cos |
|---|-------------|----------------|------------|---------------|----------|---------------|-----|
| 1 | 8           | Germany        | DE         | Europe        | 3805...  | 1307949.7917  | 0.0 |
| 2 | 10          | United Kingdom | GB         | Europe        | 5012...  | 1635823.3967  | 0.0 |
| 3 | 9           | Australia      | AU         | Pacific       | 5977...  | 2278548.9776  | 0.0 |
| 4 | 7           | France         | FR         | Europe        | 4772...  | 2396539.7601  | 0.0 |
| 5 | 3           | Central        | US         | North America | 3072...  | 3205014.0767  | 0.0 |
| 6 | 1           | Northwest      | US         | North America | 7887...  | 3298694.4938  | 0.0 |
| 7 | 2           | Northeast      | US         | North America | 2402...  | 3607148.9371  | 0.0 |
| 8 | 5           | Southeast      | US         | North America | 2538...  | 3925071.4318  | 0.0 |
| 9 | 4           | Southwest      | US         | North America | 1051...  | 5366575.7098  | 0.0 |





# Working with XML 1-2

Extensible Markup Language (XML) allows developers to develop their own set of tags and makes it possible for other programs to understand these tags.

XML is the preferred means for developers to store, format, and manage data on the Web.

Applications of today have a mix of technologies such as ASP, Microsoft .NET technologies, XML, and SQL Server 2012 working in tandem.

In such a scenario, it is better to store XML data within SQL Server 2012.

# Working with XML 2-2

- Native XML databases in SQL Server 2012 have a number of advantages. Some of them are listed as follows:

## Easy Data Search and Management

- All the XML data is stored locally in one place, thus making it easier to search and manage.

## Better Performance

- Queries from a well-implemented XML database are faster than queries over documents stored in a file system.
- Also, the database essentially parses each document when storing it.

## Easy data processing

- Large documents can be processed easily.

- SQL Server 2012 supports native storage of XML data by using the xml data type.

# XML Data Type 1-3

In addition to regular commonly used data types, SQL Server 2012 provides a brand new data type in the form of xml data type.

The xml data type is used to store XML documents and fragments in an SQL Server database.

An XML fragment is an XML instance with the top-level element missing from its structure.

- The syntax to create a table with columns of type xml is as follows:

## Syntax:

```
CREATE TABLE <table_name> ( [ column_list,] <column_name> xml [,  
column_list])
```

## XML Data Type 2-3

- Following code snippet creates a new table named PhoneBilling with one of the columns belonging to xml data type:

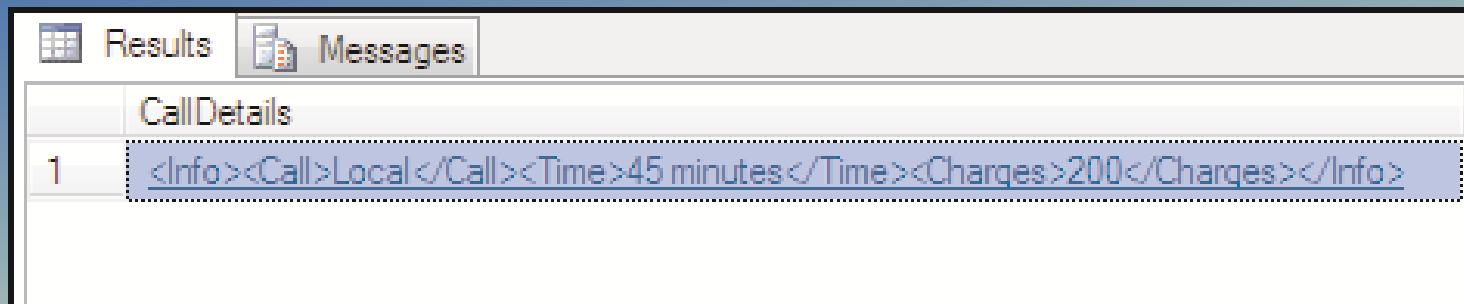
```
USE AdventureWorks2012
CREATE TABLE Person.PhoneBilling (Bill_ID int PRIMARY KEY, MobileNumber
bigint UNIQUE, CallDetails xml)
GO
```

- A column of type xml can be added to a table at the time of creation or after its creation.
- The xml data type columns support DEFAULT values as well as the NOT NULL constraint.
- Data can be inserted into the xml column in the Person.PhoneBilling table as shown in the following code snippet:

```
USE AdventureWorks2012
INSERT INTO Person.PhoneBilling VALUES (100,9833276605,
'<Info> <Call>Local</Call> <Time>45 minutes </Time> <Charges> 200
</Charges> </Info>')
SELECT CallDetails FROM Person.PhoneBilling
GO
```

# XML Data Type 3-3

- The output is shown in the following figure:



The screenshot shows a SQL Server Enterprise Manager window with two tabs: 'Results' and 'Messages'. The 'Results' tab is active, displaying a table named 'CallDetails'. The table has one row with the ID '1'. The data in this row is an XML string: `<Info><Call>Local</Call><Time>45 minutes</Time><Charges>200</Charges></Info>`. The XML string is highlighted with a blue selection box.

|   | CallDetails   |
|---|---|
| 1 | <code>&lt;Info&gt;&lt;Call&gt;Local&lt;/Call&gt;&lt;Time&gt;45 minutes&lt;/Time&gt;&lt;Charges&gt;200&lt;/Charges&gt;&lt;/Info&gt;</code> |

- The DECLARE statement is used to create variables of type xml.
- Following code snippet shows how to create a variable of type xml:

```
DECLARE @xmlvar xml
SELECT @xmlvar='<Employee name="Joan" />'
```

- The xml data type columns cannot be used as a primary key, foreign key, or as a unique constraint.



# Typed and Untyped XML 1-4

There are two ways of storing XML documents in the xml data type columns, namely, typed and untyped XML.

An XML instance which has a schema associated with it is called typed XML instance. A schema is a header for an XML instance or document.

It describes the structure and limits the contents of XML documents by associating xml data types with XML element types and attributes.

Associating XML schemas with the XML instances or documents is recommended because data can be validated while it is being stored into the xml data type column.

SQL Server does not perform any validation for data entered in the xml column. However, it ensures that the data that is stored is well-formed.

Untyped XML data can be created and stored in either table columns or variables depending upon the need and scope of the data.

# Typed and Untyped XML 2-4

- The first step in using typed XML is registering a schema.
- This is done by using the `CREATE XML SCHEMA COLLECTION` statement as shown in the following code snippet:

```
USE SampleDB
CREATE XML SCHEMA COLLECTION CricketSchemaCollection
AS N'<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema" >
<xsd:element name="MatchDetails">
<xsd:complexType>
<xsd:complexContent>
<xsd:restriction base="xsd:anyType">
<xsd:sequence>
<xsd:element name="Team" minOccurs="0" maxOccurs="unbounded">
<xsd:complexType>
<xsd:complexContent>
<xsd:restriction base="xsd:anyType">
<xsd:sequence />
<xsd:attribute name="country" type="xsd:string" />
<xsd:attribute name="score" type="xsd:string" />

```

# Typed and Untyped XML 3-4

```
</xsd:restriction>
</xsd:complexContent>
</xsd:complexType>
</xsd:element>
</xsd:sequence>
</xsd:restriction>
</xsd:complexContent>
</xsd:complexType>
</xsd:element>
</xsd:schema>'
GO
```

- The `CREATE XML SCHEMA COLLECTION` statement creates a collection of schemas, any of which can be used to validate typed XML data with the name of the collection.
- This example shows a new schema called `CricketSchemaCollection` being added to the `SampleDB` database.
- Once a schema is registered, the schema can be used in new instances of the `xml` data type.



# Typed and Untyped XML 4-4

- Following code snippet creates a table with an xml type column and specifies a schema for the column:

```
USE SampleDB

CREATE TABLE CricketTeam ( TeamID int identity not null, TeamInfo
xml(CricketSchemaCollection) )

GO
```

- To create new rows with the typed XML data, the INSERT statement can be used as shown in the following code snippet:

```
USE SampleDB

INSERT INTO CricketTeam (TeamInfo) VALUES ('<MatchDetails><Team
country="Australia" score="355"></Team><Team country="Zimbabwe"
score="200"></Team><Team country="England"
score="475"></Team></MatchDetails>')

GO
```

- A typed XML variable can also be created by specifying the schema collection name as shown in the following code snippet:

```
USE SampleDB

DECLARE @team xml(CricketSchemaCollection)

SET @team = '<MatchDetails><Team
country="Australia"></Team></MatchDetails>'

SELECT @team

GO
```

# XQuery 1-4

After XML data has been stored using the `xml` data type, it can be queried and retrieved using a language named XQuery.

XML Query or XQuery is a new query language, which combines syntax of relational database and XPath language.

XQuery can be query structured or semi-structured XML data.

To query an XML instance stored in a variable or column of `xml` type, `xml` data type methods are used.

Developers need to query XML documents, and this involves transforming XML documents in the required format.

XQuery makes it possible to perform complex queries against an XML data source over the Web.

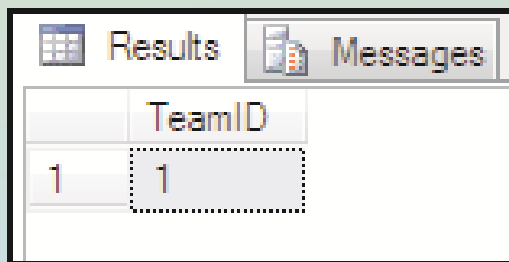
- Some of the xml data type methods used with XQuery are described as follows:

### exist()

- This method is used to determine if one or more specified nodes are present in the XML document.
- It returns 1 if the XQuery expression returned at least one node, 0 if the Xquery expression evaluated to an empty result, and NULL if the xml data type instance against which the query was executed is NULL.
- Following code snippet demonstrates the use of `exist()` method:

```
USE SampleDB
SELECT TeamID FROM CricketTeam WHERE
TeamInfo.exist('(/MatchDetails/Team)') = 1
GO
```

- This will return only those TeamID values where the Team element has been specified in the TeamInfo. The output is shown in the following figure:

A screenshot of the SQL Server Enterprise Manager interface. At the top, there are two tabs: 'Results' and 'Messages'. The 'Results' tab is active, displaying a table with two columns. The first column is labeled '1' and the second column is labeled 'TeamID'. The first row of data shows the value '1' in the first column and '1' in the second column. The 'TeamID' column header is highlighted with a dashed border.

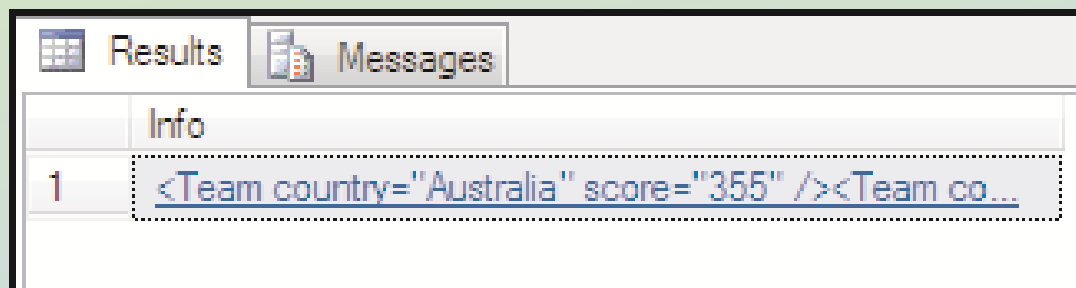
| 1 | TeamID |
|---|--------|
| 1 | 1      |

## query()

- The `query()` method can be used to retrieve either the entire contents of an XML document or a selected section of the XML document.
- Following code snippet shows the use of `query()` method:

```
USE SampleDB
SELECT TeamInfo.query('/MatchDetails/Team') AS Info FROM CricketTeam
GO
```

- The output is shown in the following figure:



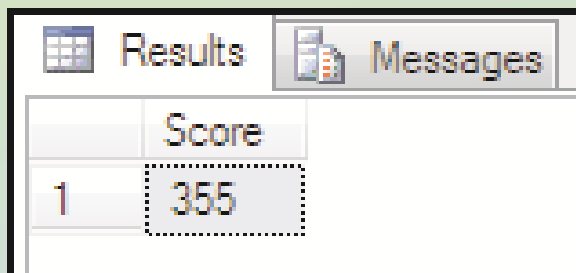
| Results |   | Messages |  |
|---------|---|----------|--|
|         | Info  |          |  |
| 1       | <Team country="Australia" score="355" /><Team co... |          |  |

## value()

- The `value()` method can be used to extract scalar values from an xml data type.
- Following code snippet demonstrates the use of this method:

```
USE SampleDB  
SELECT TeamInfo.value('(/MatchDetails/Team/@score)[1]', 'varchar(20)') AS  
Score FROM CricketTeam where TeamID=1  
GO
```

- The output is shown in the following figure:



The screenshot shows a SQL Server query results window. The 'Results' tab is active, displaying a table with two columns: 'TeamID' and 'Score'. The first row contains the values '1' and '355'. The 'Score' cell is highlighted with a dashed border.

| Results |       | Messages |  |
|---------|-------|----------|--|
|         | Score |          |  |
| 1       | 355   |          |  |



# Summary

- The SELECT statement retrieves rows and columns from tables.
- SELECT statement allows the users to specify different expressions in order to view the resultset in an ordered manner.
- A SELECT statement can contain mathematical expressions by applying operators to one or more columns.
- The keyword DISTINCT prevents the retrieval of duplicate records.
- XML allows developers to develop their own set of tags and makes it possible for other programs to understand these tags.
- A typed XML instance is an XML instance which has a schema associated with it.
- XML data can be queried and retrieved using XQuery language.