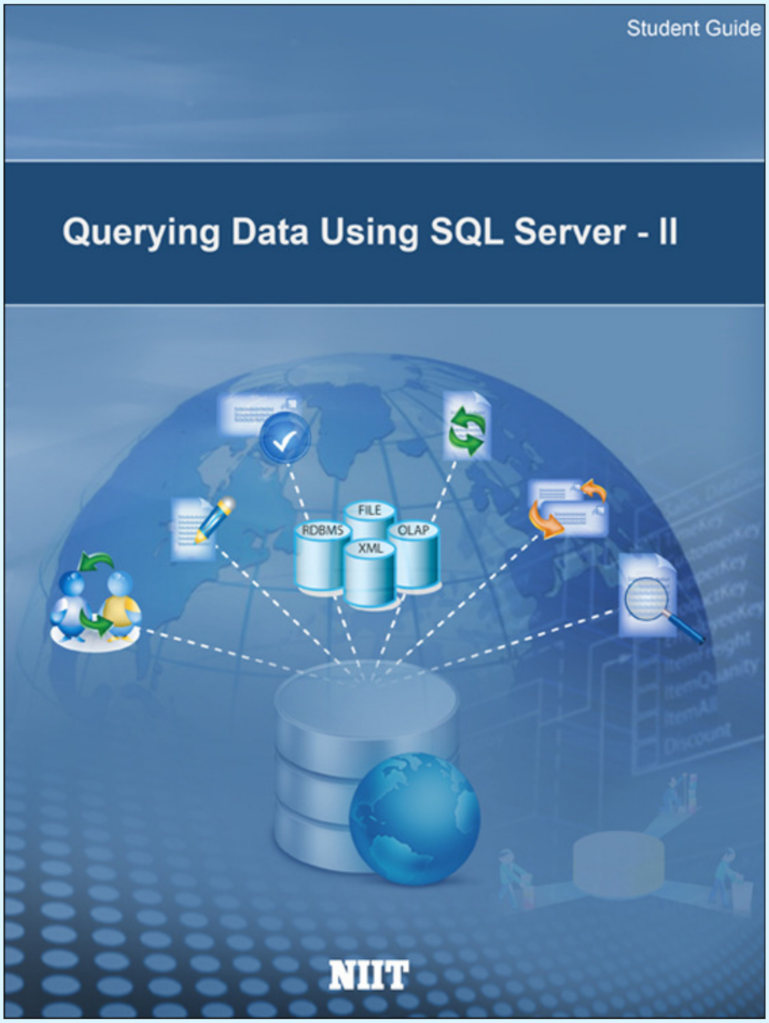
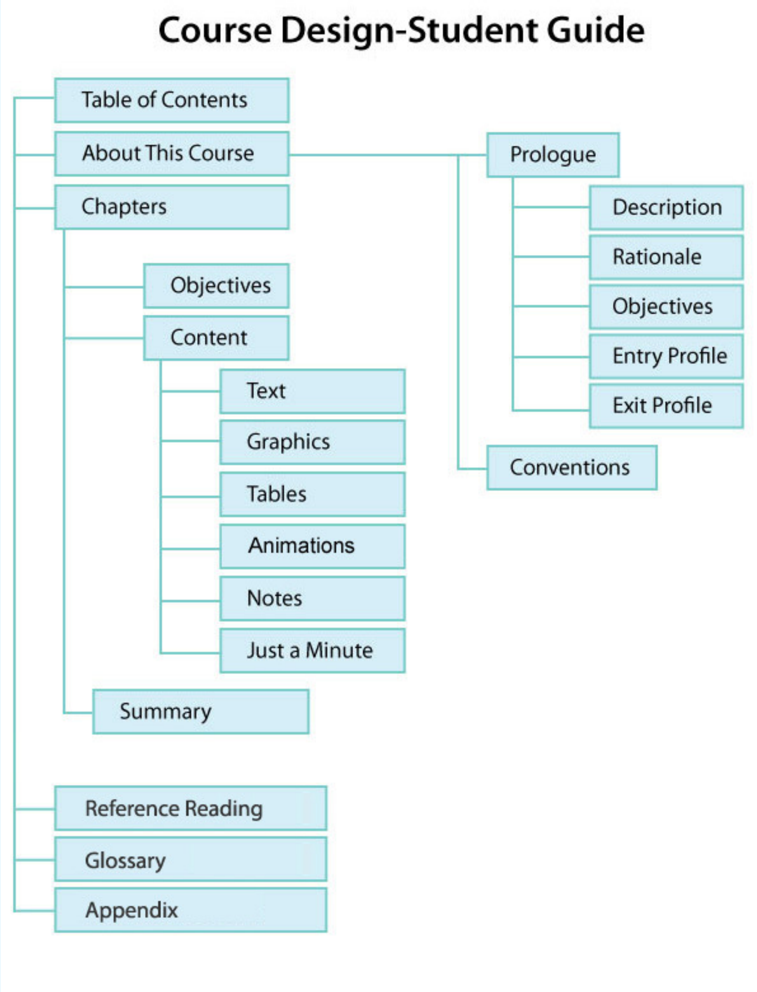


Student Guide

Querying Data Using SQL Server - II

NIIT


Course Design-Student Guide
Table of Contents
About This Course
Prologue
Description
Rationale
Objectives
Entry Profile
Exit Profile
Conventions
Chapters
Objectives
Content
Text
Graphics
Tables
Animations
Notes
Just a Minute
Summary
Reference Reading
Glossary
Appendix

QDUS - II_SG /12-M11-V01

About This Course

Prologue

Description

The Querying Data Using SQL Server - II course is designed to provide students the necessary skills to query and manage databases using SQL Server. The course starts with an introduction to client-server architecture and an overview of SQL Server. Next, it familiarizes the students with Structured Query Language (SQL) and builds on the complexity of query handling in SQL Server with every progressing session. It also introduces the students to some advanced concepts in SQL Server such as implementing managed code and implementing services for message-based communication.

Rationale

Databases are an integral part of an organization. Aspiring database developers should be able to efficiently implement and maintain databases. This knowledge will enable them to build robust database solutions.

Objectives

After completing this course, the student should be able to:

- ❑ Identify SQL Server tools
- ❑ Query data from a single table
- ❑ Query data from multiple tables
- ❑ Manage databases and tables
- ❑ Manipulate data in tables
- ❑ Implement indexes, views, and full-text search
- ❑ Implement stored procedures and functions
- ❑ Implement triggers and transactions
- ❑ Implement managed code
- ❑ Implement services for message-based communication

Entry Profile

The students who want to take this course should have:

- ❑ Knowledge of RDBMS concepts.
- ❑ Basic knowledge of computers.

Exit Profile

After completing this course, the students should be able to:

- ❑ Query and manage databases using SQL Server.

Conventions

Convention	***Indicates...***
NOTE	*Note*
🎞	*Animation*
📖	*Exercise in Activity Book*
⏰	*Just a Minute*

Chapter 5

Manipulating Data in Tables

After creating a database and tables, the next step is to store data in the database. As a database developer, you will be required to update or delete data. You can perform these data manipulations by using the DML statements of T-SQL.

The data stored in the database can be used by different types of client applications, such as mobile devices or Web applications. Therefore, data should be stored in a format that can be interpreted by any application. For this, SQL Server allows you to store data in the XML format that can be read by any application.

This chapter discusses how to use the DML statements to manipulate data in the tables. In addition, it explains how to manipulate the XML data in the database tables.

Objectives

In this chapter, you will learn to:

- ❑ Manipulate data by using DML statements
- ❑ Manipulate XML data

Manipulating Data by Using DML Statements

As a database developer, you need to regularly insert, update, or delete data. These operations ensure that the data is up-to-date. For example, you need to insert new records in the Employee table whenever a new employee joins the organization.

Similarly, if the details of an employee change, you need to update the existing records. For example, if the salary of any employee increases, you need to update the existing records to reflect this change.

You can use the DML statements to manipulate the data in any table.

Storing Data in a Table

The smallest unit of data that you can add in a table is a row. You can add a row by using the INSERT statement. The syntax of the INSERT statement is:

```
INSERT [INTO]{table_name}
[(column_list)]
VALUES {DEFAULT | values_list |
select_statement}
```

where,

`table_name` specifies the name of the table into which the data is to be inserted. The INTO keyword is optional.

`column_list` specifies an optional parameter. You can use it when partial data is to be inserted in a table or when the columns to be inserted are defined in a different order.

`DEFAULT` specifies the clause that you can use to insert the default value specified for the column. If a default value is not specified for a column and the column property is specified as NULL, NULL is inserted in the column. If the column does not have any default constraint attached to it and does not allow NULL as the column value, SQL Server returns an error message and the insert operation is rejected.

`values_list` specifies the list of values for the table columns that have to be inserted as a row in the table. If a column has to contain a default value, you can use the DEFAULT keyword instead of a column value. The column value can also be an expression.

`select_statement` specifies a nested SELECT statement that you can use to insert rows into the table.

Guidelines for Inserting Rows

While inserting rows into a table, you need to consider the following guidelines:

- ❑ The number of data values must be the same as the number of attributes in the table or column list.
- ❑ The order of inserting the information must be the same as the order in which attributes are listed for insertion.
- ❑ The values clause need not contain the column with the IDENTITY property.
- ❑ The data types of the information must match the data types of the columns of the table.

Consider an example of the EmpData table that is used to store the details of the employees.

The following table describes the structure of the EmpData table.

Column Name	***Data Type***	***Checks***
EmpName	*varchar(20)*	*NULL*
EmpNo	*Int*	*NOT NULL*
EmpAddress	*varchar(60)*	*NULL*
Salary	*Int*	*NULL*

The Structure of the EmpData Table

To insert a row into the EmpData table with all the column values, you can use any one of the following statements:

```
INSERT EmpData
VALUES ('Yang Kan', 101, '123 Nanjing
Lu', 2500)
```

Or

```
INSERT EmpData (EmpName, EmpAddress,
EmpNo, Salary)
```

```
VALUES ('Yang Kan', '123 Nanjing Lu',
101, 2500)
```

Or

```
INSERT EmpData (EmpName, EmpAddress,
Salary, EmpNo)
VALUES ('Yang Kan', '123 Nanjing Lu',
2500, 101)
```

You need to execute an INSERT statement for each row being inserted into a table. However, you can insert multiple rows into a table by using the T-SQL row constructor. The row constructor allows multiple rows of data to be specified in a single INSERT statement.

For example, the following statement inserts three rows in the EmpData table by using the row constructor:

```
INSERT EmpData (EmpName, EmpAddress,
Salary, EmpNo)
VALUES ('John', '123 Nanjing Lu',
3500, 102),
('Michel', '124 Nanjing Lu', 2200,
103),
('Shelda', '693 Park Street', 3500,
104)
```

Inserting Partial Data

Depending on the constraints applied to the columns of the tables, you can insert partial data into the database tables. This means that while performing an insert operation it is not necessary to insert values for all the columns in the table.

SQL server allows you to insert data in selected columns, leaving those columns that allow NULL or have a default constraint assigned to them. While inserting data, you need to specify the column names for which data is to be inserted by using the INSERT statement. The VALUES clause provides values for the specified columns only.

In EmpData table, the EmpAddress column allows you to enter a NULL value in a row. Therefore, you can use the following statements to insert partial data into the table:

```
INSERT EmpData
VALUES ('Yang Kan', 101, NULL, 2500)
```

Or

```
INSERT EmpData (EmpName, EmpNo,
Salary)
VALUES ('Yang Kan', 101, 2500)
```

Inserting Data in Related Tables

Data related to an entity can be stored in more than one table. Therefore, while adding information for a new entity, you need to insert new rows in all the related tables. In such a case, you need to first insert a row in the table that contains the primary key. Next, you can insert a row in the related table containing the foreign key.

For example, in the AdventureWorks database, the employee details are stored in Person.Contact, HumanResources.Employee, HumanResources.EmployeeDepartmentHistory, and HumanResources.EmployeePayHistory tables. To save the details for a new employee, you need to insert data in all these tables.

The following statements insert data of a new employee into the database:

```
-- inserting records in the
Person.Contact table
INSERT INTO Person.Contact VALUES (0,
'Mr.', 'Steven', NULL, 'Fleming',
NULL, 'stevenfleming@adventure-
works.com', 1,
'951-667-2401',
'B4802B37F8F077A6C1F2C3F50F6CD6C5379E9
C79', '3sa+edf=', NULL, DEFAULT,
DEFAULT)
-- inserting records in the
HumanResources.Employee table
INSERT INTO HumanResources.Employee
VALUES ('45879632', 19978,
'adventure-works\steven', 185, 'Tool
Designer', '1967-06-03
00:00:00.000','M', 'M', '2006-08-01
00:00:00.000', 1, 0, 0, 1, DEFAULT,
DEFAULT)
-- inserting records in the
HumanResources.EmployeeDepartmentHisto
ry table
INSERT INTO
HumanResources.EmployeeDepartmentHisto
ry VALUES (291, 2, 1, '2006-08-01
00:00:00.000', NULL, DEFAULT)
-- inserting records in the
HumanResources.EmployeePayHistory
table
INSERT INTO
HumanResources.EmployeePayHistory
VALUES (291,
'2006-08-01 00:00:00.000', 23.0769, 2,
DEFAULT)
```

In the preceding statements, the Person.Contact table contains the ContactID column, which is a foreign key in the HumanResources.Employee table. Once you have inserted a record in the Person.Contact table, you can insert a record in the HumanResources.Employee table to generate the employee ID for that person. This employee ID will be used to insert rows in the HumanResources.EmployeeDepartmentHistory and HumanResources.EmployeePayHistory tables.

Copying Data from an Existing Table into a New Table

While inserting data in a table, you might need to copy rows from an existing table into a new table. You can do this by using the SELECT statement.

For example, in the AdventureWorks database, data for the employees with a payment rate of 35 or above is to be copied into a new table called PreferredEmployee from the EmployeePayHistory table.

The following query copies the values from the EmployeePayHistory table into the PreferredEmployee table:

```
SELECT * INTO PreferredEmployee
FROM HumanResources.EmployeePayHistory
WHERE Rate >= 35
```

The preceding query will create a new table named PreferredEmployee. The table will have the same structure as HumanResources.EmployeePayHistory.

You can also copy the values from an existing table to another existing table. For example, you want to copy data for the employees with a payment rate of 10 or less from the EmployeePayHistory table into the PreferredEmployee table. The PreferredEmployee table exists in the database. You can perform this task by using the following statement:

```
INSERT INTO PreferredEmployee
SELECT * FROM
HumanResources.EmployeePayHistory
WHERE Rate <=10
```

> NOTE *Whenever you copy data to another existing table, the structure of the source and the target tables must be same. In addition, the SELECT INTO statement only copies the data of the source table into the target table. However, it does not copy the constraints, indexes, triggers, or keys of the source table in the target table.*

Inserting Data in an Identity Column into a Table

A column in a table for which the values are automatically generated by the database is called an identity column. You are not required to specify the values for the identity column while inserting data. For example, you can use the following statement to insert data in the Emp table:

```
INSERT INTO Emp VALUES ('Steve', 10)
```

In the preceding statement, the value of the ID column is not specified as it is set as an identity column. However, if you specify a value for the identity column while inserting data into the table, an error occurs. For example, try to execute the following statement to insert the value in the identity column:

```
INSERT INTO Emp(EmpCode, EmpName,
DeptNo) VALUES (3, 'Nancy', 20)
```

The preceding statement displays an error on execution because you cannot explicitly insert data in an identity column, EmpCode. To insert a value explicitly in an identity column, you need to use the SET_IDENTITY_INSERT statement.

The SET_IDENTITY_INSERT statement sets the identity insert property of a database object. You need to set this property to ON to insert a value in an identity column. By default, this property is set to OFF. For example, to execute the preceding INSERT statement without any error, you need to first execute the following statement:

```
SET IDENTITY_INSERT Emp ON
```

The preceding statement sets the identity insert property of the Emp table to ON. Now, you can execute the following statement to insert a value in the identity column:

```
INSERT INTO Emp(EmpCode, EmpName,
DeptNo) VALUES (3, 'Nancy', 20)
```

The preceding statement executes successfully and inserts a row in the Emp table.

Inserting the FILESTREAM Data

You can insert the FILESTREAM data into a table as a value of varbinary (max) data type. For example, you can use the following statement to insert data into the EmpDetails table:

```
USE TekSoft
INSERT INTO EmpDetails(EmployeeID,
EmployeeName,
EmployeeAddress,EmployeeDept,EmployeeP
hoto)
VALUES(NEWID(),'Mark', 'California',
'Testing',cast('D:\Emp_Photo
\Emp101.jpg'as varbinary(max)))
```

In the preceding statement, the NEWID() function is used to generate the unique values for the EmployeeID column. In addition, the cast() function is used to convert the string literal that represents the path of the employee photo to the varbinary (max) data.

> NOTE *The NEWID() function is used to generate unique values for the columns that are declared as the UNIQUEIDENTIFIER data type.*

Inserting Globally Unique Values

You can use the NEWSEQUENTIALID() function to generate a unique value or GUID for a column. This function creates GUID in a sequential manner. Therefore, GUID created by the NEWSEQUENTIALID() function is always greater than the values that were previously created by this function on the same computer since Windows was started. However, when Windows is restarted, GUID created by the NEWSEQUENTIALID() function again starts from a lower range, but still the generated GUID value will be globally unique.

The NEWSEQUENTIALID() function returns a value of the type, uniqueidentifier. The only restriction for the NEWSEQUENTIALID() function is that the function can be used only to generate a unique identifier value of a column with the DEFAULT constraint. For example, consider the following statement to create a table:

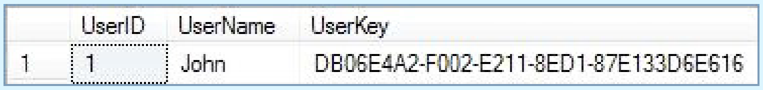
```
CREATE TABLE UserDetails (
UserID INT NOT NULL IDENTITY(1, 1),
UserName VARCHAR(20) NOT NULL,
UserKey UNIQUEIDENTIFIER DEFAULT
( NEWSEQUENTIALID()))
```

In the preceding statement, a table named UserDetails is created. This table contains a column named UserKey whose value will be generated by the NEWSEQUENTIALID() function.

You can use the following statement to insert values in the UserDetails table:

```
INSERT INTO UserDetails (UserName)
VALUES ('John')
```

Once executed, the preceding statement inserts a row in the UserDetails table, as shown in the following figure.



	UserID	UserName	UserKey
1	1	John	DB06E4A2-F002-E211-8ED1-87E133D6E616

The Ouput Derived After Using the NEWSEQUENTIALID() Function

The NEWSEQUENTIALID() function cannot be used to generate a value in a query statement. If the NEWSEQUENTIALID() function is used in a query, an error message is dispalyed. For example, consider the following query:

```
SELECT NEWSEQUENTIALID();
```

When the preceding query is executed, the following error message is displayed:

```
Msg 302, Level 16, State 0, Line 1
The newsequentialid() built-in
function can only be used in a DEFAULT
expression for a column of type
'uniqueidentifier' in a CREATE TABLE
or ALTER TABLE statement. It cannot be
combined with other operators to form
a complex scalar expression.
```

Inserting the Spatial Data

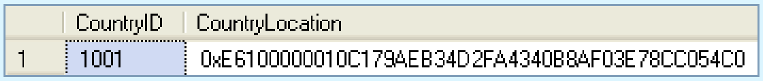
Geographic locations such as bank branches, restaurants, and company locations are usually defined in terms of latitude and longitude. SQL server provides you with the GEOGRAPHY data type to store geographic locations as points. A point is represented in terms of X and Y coordinates. For geographic locations, X-coordinate is the longitude, while the Y-coordinate is the latitude. You can use the Geography::Parse static method to convert the latitude and longitude into a geographic point.

The Geography::Parse method takes a point type value as a parameter. The X and Y coordinates in the point type value are separated by a space. For example, you can use the following statement to insert a row in the Country_Location table:

```
USE AdventureWorks
INSERT INTO Country_Location
(CountryID, CountryLocation)
VALUES(1001, Geography::Parse('POINT
(-83.0086 39.95954)'))
```

In the preceding statement, -83.0086 represents the longitude and 39.95954 represents the latitude. Therefore, the location of a country is specified as a point by using the Geography data type. When the preceding statement is executed, the data in the CountryLocation column will be in the hexadecimal format.

The following figure shows the data inserted in the Country_Location table.



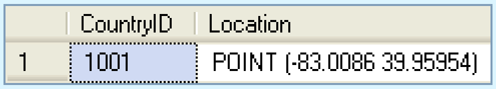
	CountryID	CountryLocation
1	1001	0xE6100000010C179AEB34D2FA4340B8AF03E78CC054C0

The Data Inserted in the Country_Location Table

To retrieve the data in the form of a point value, you need to use the ToString() function, as shown in the following query:

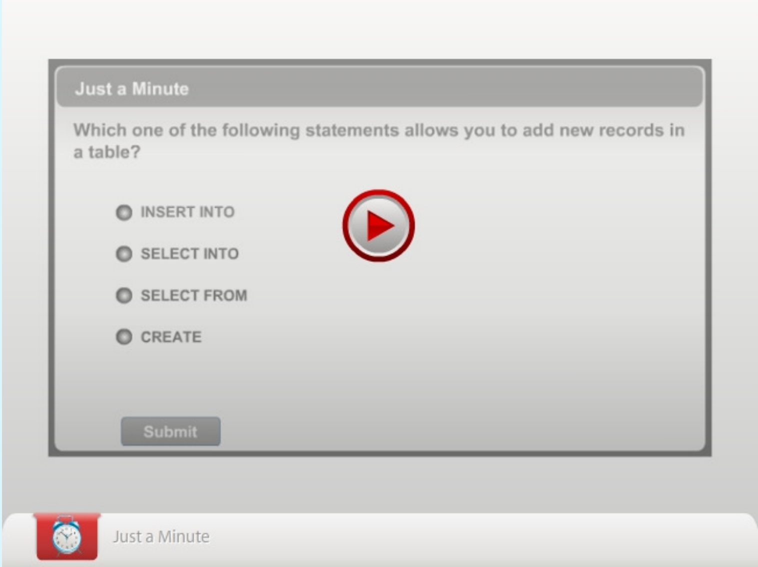
```
SELECT CountryID,
CountryLocation.ToString() AS Location
FROM Country_Location
```

The preceding query displays the output, as shown in the following figure.



	CountryID	Location
1	1001	POINT (-83.0086 39.95954)

The Ouput Derived After Using the ToString() Function



Just a Minute

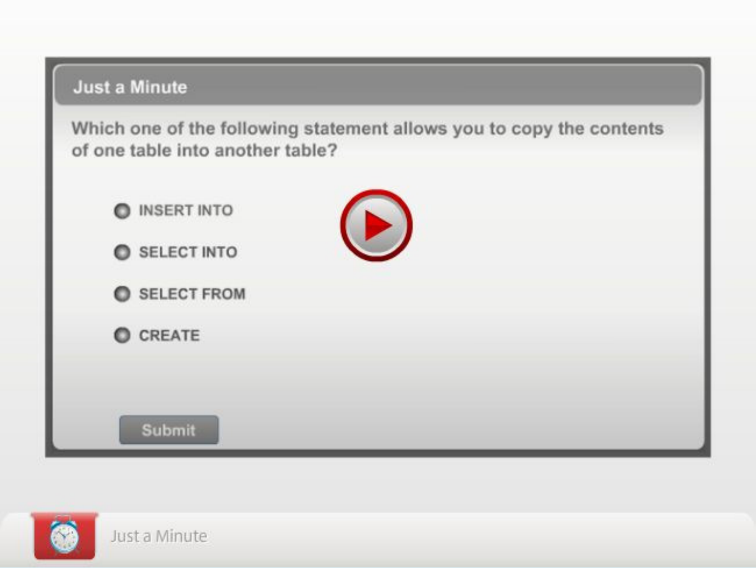
Which one of the following statements allows you to add new records in a table?

○ INSERT INTO

○ SELECT INTO

○ SELECT FROM

○ CREATE



Which one of the following statement allows you to copy the contents of one table into another table?

○ INSERT INTO

○ SELECT INTO

○ SELECT FROM

○ CREATE

Updating Data in a Table

You need to modify the data in the database when the specifications of a customer, a client, a transaction, or any other data maintained by the organization undergo a change.

For example, if a client changes his address or if the quantity of a product ordered is changed, the required changes need to be made to the respective rows in the tables. You can use the UPDATE statement to make the changes. Updating ensures that the latest and correct information is available at any point of time. You can update one or more than one column of a row.

You can update data in a table by using the UPDATE DML statement. The syntax of the UPDATE statement is:

```
UPDATE table_name
SET column_name = value [, column_name
= value]
[FROM table_name]
[WHERE condition]
```

where,

`table_name` specifies the name of the table you have to modify.

`column_name` specifies the columns you have to modify in the specified table.

`value` specifies the value(s) with which you have to update the column(s) of the table. Some valid values include an expression, a column name, and a variable name. The DEFAULT and NULL keywords can also be supplied.

`FROM table_name` specifies the table(s) that is used in the UPDATE statement.

`condition` specifies the rows that you have to update.

Guidelines for Updating Data

You need to consider the following guidelines while updating data:

- ❑ An update can be done on only one table at a time.
- ❑ If an update violates integrity constraints, then the entire update is rolled back.
- ❑ If you do not specify a condition by using the WHERE clause, all the rows will be updated with the value specified for the column.

For example, the following table displays the sample data from the EmpData table.

EmpName	EmpNo	EmpAddress	Salary
Yang Kan	101	123 Nanjing Lu	2500
John	102	123 Nanjing Lu	3500
Michel	103	124 Nanjing Lu	2200
Shelda	104	693 Park Street	3500

The Sample Data of the EmpData Table

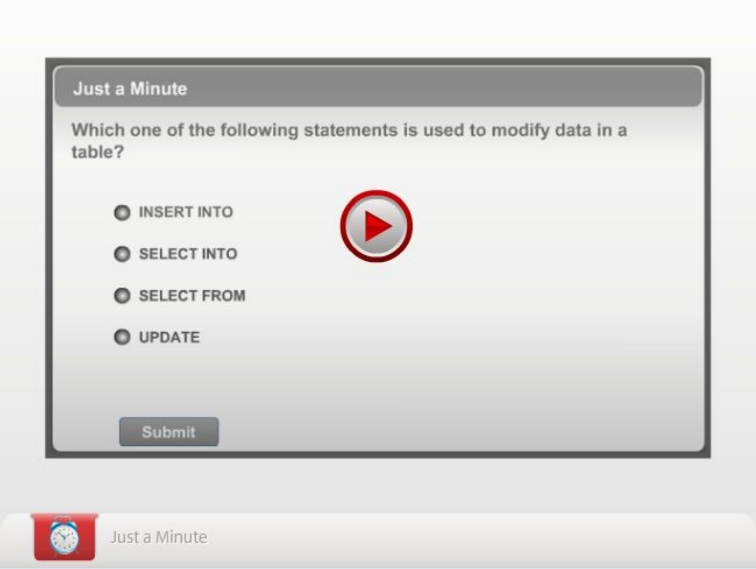
The following statement updates the address of the employee having EmpNo, 101:

```
UPDATE EmpData
SET EmpAddress = '123 Shi Lu'
WHERE EmpNo = 101
```

Consider another example, where you need to update the title of an employee named Lynn Tsoflias to Sales Executive in the Employee table. To perform this task, you need to first refer to the Contact table to obtain the Contact ID.

You can then use this ContactID to update the employee details by using the following statement:

```
UPDATE HumanResources.Employee SET
Title = 'Sales Executive'
FROM HumanResources.Employee e,
Person.Contact c
WHERE e.contactID = c.ContactID
AND c.FirstName = 'Lynn' and
c.LastName = 'Tsoflias'
```



Just a Minute

Which one of the following statements is used to modify data in a table?

○ INSERT INTO

○ SELECT INTO

○ SELECT FROM

○ UPDATE

Deleting Data from a Table

You need to delete data from a database when it is no longer required. The smallest unit that can be deleted from a database is a row.

You can delete a row from a table by using the DELETE DML statement. The syntax of the DELETE statement is:

```
DELETE [FROM] table_name
[WHERE condition]
```

where,

`table_name` specifies the name of the table from which you have to delete rows.

`condition` specifies the condition that identifies the row(s) to be deleted.

For example, the following statement deletes the address details of AddressID 104 from the Address table:

```
DELETE PreferredEmployee
WHERE EmployeeID = 4
```

Deleting Data from Related Tables

While deleting records from related tables, you first need to delete the records from the table that contains the foreign key. After that, you can delete the record from the table that contains the primary key.

Consider the example of the AdventureWorks database. The Employee table contains data of those employees who have retired from the company. This data is not required anymore. This increases the size of the database. You are required to ensure that this old data is removed from the Employee table.

You can delete this data by using the following statement:

```
DELETE FROM HumanResources.Employee
EmployeeID = 9
```

The database contains tables related to the Employee table. The related tables are HumanResources.EmployeeAddress, HumanResources.EmployeeDepartmentHistory, HumanResources.EmployeePayHistory, and HumanResources.JobCandidate. The EmployeeID attribute in these tables is a foreign key to the EmployeeID attribute of the Employee table. Therefore, the query results in an error. You need to delete data from the related tables before executing the preceding DELETE statement.

Deleting all the Records from a Table

As a database developer, you might need to delete all the records from a table. You can do this by using the following DELETE statement:

```
DELETE table_name
```

You can also use the TRUNCATE DML statement. The syntax of the TRUNCATE statement is:

```
TRUNCATE TABLE table_name
```

where,

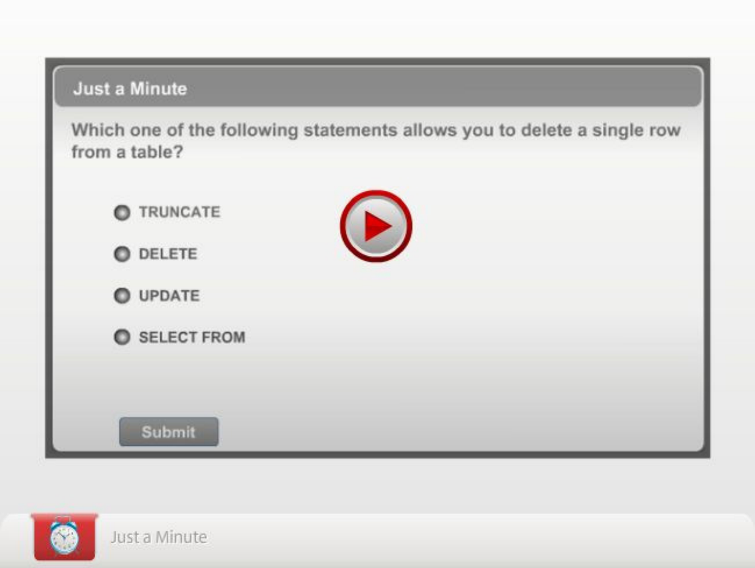
`table_name` specifies the name of the table from which you have to delete rows.

However, the TRUNCATE DML statement is executed faster.

NOTE *The TRUNCATE DML statement does not support the WHERE clause. In addition, the TRUNCATE TABLE statement does not fire a trigger. When truncate is used, the deleted rows are not entered in the transaction log.*

For example, the following statement deletes all the records from the EmpData table:

```
TRUNCATE TABLE EmpData
```



Just a Minute

Which one of the following statements allows you to delete a single row from a table?

○ TRUNCATE

○ DELETE

○ UPDATE

○ SELECT FROM

Retrieving the Modified Data

While performing various operations, such as insert, update, and delete on the tables in a database, SQL Server displays only the number of rows affected as a result of these operations. For example, if you delete a row from a table based on specific criteria, the following statement will be displayed as a result:

```
(1 row(s) affected)
```

Sometimes, you also need to retrieve the data affected by the DML statements. For example, consider one of the employees has been shifted from Production to the Quality Control department. For this, you would update the department ID of the employee in the Employee table. However, you also need to retain the historical data of all the employees. Therefore, you first need to archive the old employee data, and then update the employee data in the Employee table. SQL Server allows you to perform these two tasks in a single statement by using the OUTPUT clause.

The OUTPUT clause is used to retrieve the set of rows affected by the DML statements. You need to attach the OUTPUT clause to an INSERT, UPDATE, or DELETE statement to retrieve the modified data. The OUTPUT clause is used along with the INSERTED or DELETED statement to return the data of the affected rows.

When a row is inserted in a table, a virtual table named INSERTED is created that contains the inserted data. Similarly, when a row is deleted from a table, a virtual table named DELETED is created that contains the deleted data. You can reference the inserted or deleted data in a table by using the INSERTED or DELETED statements along with the OUPUT clause, respectively. Similarly, when a row is updated in a table, both the virtual tables, INSERTED and DELETED, are created.

For example, the rate of an employee having ID, 4 is changed from 8.62 to 9.50. Now, you need to update the Rate column in the PreferredEmployee table and insert the old rate in the PreferredEmployeeArchive table. Execute the following statement to create the PreferredEmployeeArchive table:

```
CREATE TABLE PreferredEmployeeArchive
(
  EmployeeID int NOT NULL,
  RateChangeDate datetime NOT NULL,
  Rate money NOT NULL,
  PayFrequency tinyint NOT NULL,
  ModifiedDate datetime NOT NULL
)
```

Now, execute the following statement to update the PreferredEmployee table and insert the old data in the PreferredEmployeeArchive table:

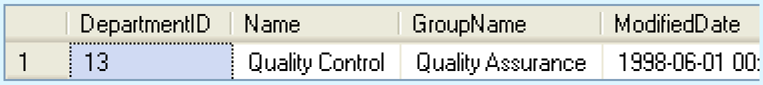
```
UPDATE PreferredEmployee
SET Rate = 9.62
OUTPUT DELETED.*
INTO PreferredEmployeeArchive
WHERE EmployeeID = 4
```

The preceding statement on execution inserts the old data of the employee having ID, 4 in the PreferredEmployeeArchive table.

The OUTPUT clause is also useful where you need to return the modified data to the client applications for further processing. For example, consider the following UPDATE statement, which updates the rows in the HumanResources.Department table where the name of the department is Quality Assurance:

```
UPDATE HumanResources.Department
SET Name='Quality Control'
OUTPUT INSERTED.*
WHERE Name='Quality Assurance'
```

In the preceding statement, the OUTPUT clause displays the rows updated in the Department table, as shown in the following figure.



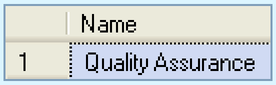
	DepartmentID	Name	GroupName	ModifiedDate
1	13	Quality Control	Quality Assurance	1998-06-01 00:

The Ouput Derived After Using the OUTPUT clause

However, if you want to display only the selected values affected by the DML statements, you can specify the column names to be displayed with the OUTPUT clause. For example, the following statement displays only the values of the Name column of the updated rows:

```
UPDATE HumanResources.Department
SET Name='Quality Assurance'
OUTPUT INSERTED.Name
WHERE Name='Quality Control'
```

The preceding statement displays the result, as shown in the following figure.



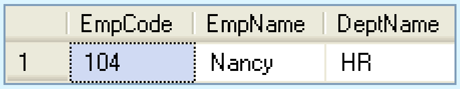
	Name
1	Quality Assurance

The Ouput Derived After Using the OUTPUT clause

Similar to the UPDATE statement, you can also use the OUTPUT clause to return the affected rows while executing the INSERT and DELETE statements. For example, consider the following statement:

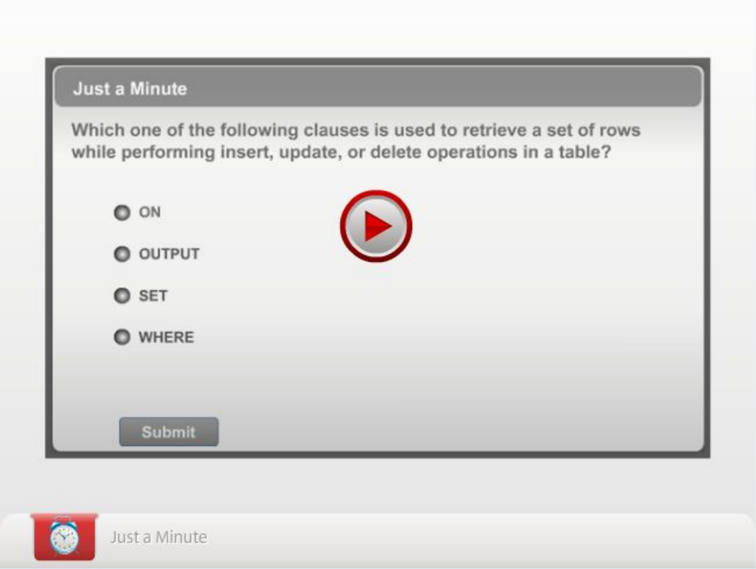
```
INSERT Emp
OUTPUT INSERTED.*
VALUES('Nancy', 'HR')
```

The preceding statement displays the inserted row in the Emp table, as shown in the following figure.



	EmpCode	EmpName	DeptName
1	104	Nancy	HR

The Ouput Derived After Using the OUTPUT clause



Just a Minute

Which one of the following clauses is used to retrieve a set of rows while performing insert, update, or delete operations in a table?

○ ON

○ OUTPUT

○ SET

○ WHERE

Comparing and Updating Data

Many business organizations keep a backup of their important data. When the data in the source table is changed, the changes must reflect in the backup table to synchronize both the tables. For example, the employee records are maintained in an Employee table. The backup of this table is taken in the Employee_Backup table. After taking the backup, one of the employees left the organization and some employees got married. You have to make these changes in the Employee table as well as the Employee_Backup table. For this, you need to first make the changes in the Employee table, and then reflect these changes in the Employee_Backup table. SQL Server allows you to perform this task by using a single MERGE statement.

The MERGE statement enables you to perform insert, update, and delete operations in a single statement. The insert, update, or delete operations are performed in the target table based on certain conditions in the source table.

The MERGE statement consists of the following clauses:

- ❑ **USING clause**: Specifies the source table to be matched with the target table.
- ❑ **ON clause**: Specifies the join conditions that determine whether the target and the source tables match.
- ❑ **WHEN clause**: Specifies the actions to be taken based on the results of the ON clause and any additional search criteria. It can be specified in the following forms:
  - **WHEN MATCHED clause**: Specifies a matched pair consisting of one row from the target and one row from the source table.
  - **WHEN NOT MATCHED BY TARGET clause**: Specifies a row in the source table that has no corresponding row in the target table.
  - **WHEN NOT MATCHED BY SOURCE clause**: Specifies a row in the target table that has no corresponding row in the source table.
- ❑ **OUTPUT clause**: Is used to return a row for each inserted, updated, or deleted row in the target table.

The syntax of the MERGE statement is:

```
MERGE
   [ TOP ( expression ) [ PERCENT ] ]
   [ INTO ] <target_table> [ WITH ( <merge_hint> ) ] [ [ AS ] table_alias ]
  USING <table_source>
  ON <merge_search_condition>
  [ WHEN MATCHED [ AND <clause_search_condition> ]
      THEN <merge_matched> ] [ ...n ]
  [ WHEN NOT MATCHED [ BY TARGET ] [ AND <clause_search_condition> ]
       THEN <merge_not_matched> ]
  [ WHEN NOT MATCHED BY SOURCE [ AND <clause_search_condition> ]
       THEN <merge_matched> ] [ ...n ]
  [ <output_clause> ]
```

where,

`TOP ( expression ) [ PERCENT ]` specifies the number or percentage of rows that are affected.

`<target_table>` specifies the table or views where the rows returned by the WHEN clause is inserted, updated, or deleted.

`<table_source>` specifies the table, view, or expression from where the rows to be matched with the target table are retrieved.

`<merge_search_condition>` specifies the condition used to decide whether the rows in the source and destination tables match.

`WHEN MATCHED THEN <merge_matched>` specifies the action to be performed on the rows in the target table when a match exists between the source and the target rows based on the specified conditions.

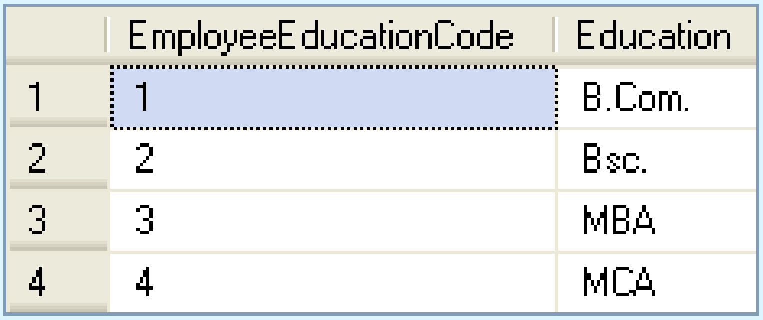
`WHEN NOT MATCHED [BY TARGET] THEN <merge_not_matched>` specifies a row to be inserted into the target table for every row that does not match a row in the source table based on the specified conditions.

`WHEN NOT MATCHED BY SOURCE THEN <merge_matched>` specifies an update or delete action on rows that exist in the target table but not in the source table.

`<clause_search_condition>` is used to specify any valid search condition.

`<output_clause>` is used to return a row for every updated, inserted, or deleted row in the target table.

For example, you have been assigned the task to maintain the backup of the Education table. Therefore, you need to regularly compare the Education table with the backup table and perform the insert, update, or delete operations on the backup table, as required. The following figure shows the contents of the Education table.



	EmployeeEducationCode	Education
1	1	B.Com.
2	2	Bsc.
3	3	MBA
4	4	MCA

The Contents of the Education Table

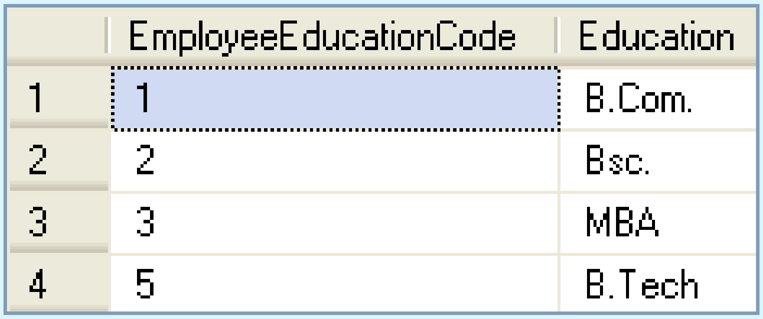
You take the backup of the Education table in the Education_Backup table by using the following statement:

```
SELECT * INTO Education_Backup FROM
Education
```

After taking the backup, the Education table is modified by using the following statements:

```
INSERT Education VALUES (5, 'B.Tech')
DELETE Education WHERE
EmployeeEducationCode = 4
```

Now, the Education table contains different records, as shown in the following figure.



	EmployeeEducationCode	Education
1	1	B.Com.
2	2	Bsc.
3	3	MBA
4	5	B.Tech

The Modified Content of the Education Table

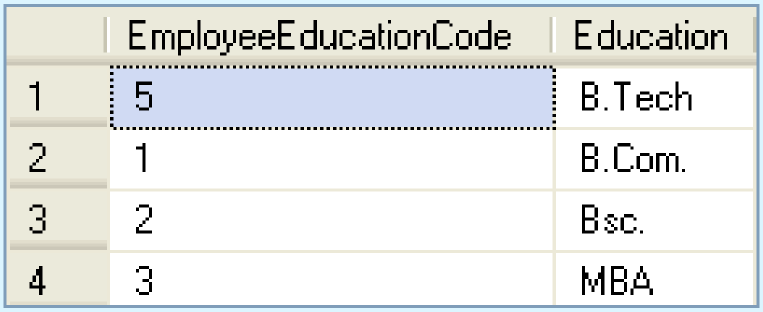
Therefore, you need to reflect the changes made in the Education table to the Education_Backup table. Here, the Education is the source table and the Education_Backup is the target table. You can use the following MERGE statement to reflect the changes in the Education_Backup table:

```
MERGE Education_Backup AS TARGET
USING Education AS SOURCE
ON (TARGET.EmployeeEducationCode =
SOURCE.EmployeeEducationCode)
WHEN MATCHED AND TARGET.Education <>
SOURCE.Education
THEN UPDATE SET TARGET.Education =
SOURCE.Education
WHEN NOT MATCHED THEN
INSERT VALUES
(SOURCE.EmployeeEducationCode,
SOURCE.Education)
WHEN NOT MATCHED BY SOURCE THEN
DELETE;
```

When the preceding statement is executed, the source table, Education and the target table, Education_Backup are matched and the Education_Backup table is updated accordingly. You can use the following query to verify the result of the preceding statement:

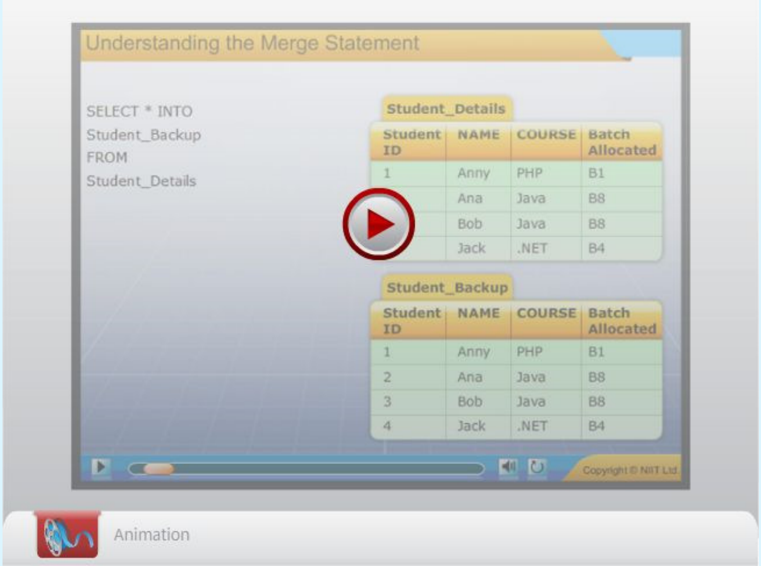
```
SELECT * FROM Education_Backup
```

The following figure shows the output of the preceding query.



	EmployeeEducationCode	Education
1	5	B.Tech
2	1	B.Com.
3	2	Bsc.
4	3	MBA

The Modified Content of the Education_Backup Table



Understanding the Merge Statement

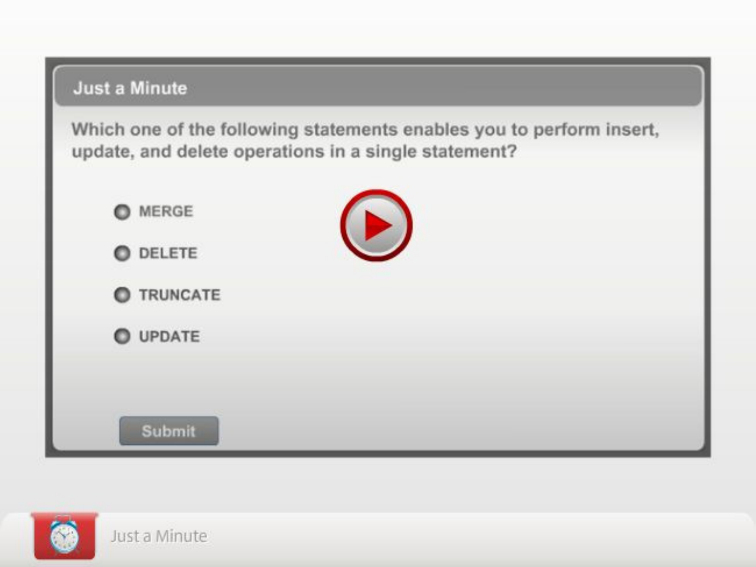
SELECT * INTO
Student_Backup
FROM
Student_Details

Student_Details

Student ID	NAME	COURSE	Batch Allocated
1	Anny	PHP	B1
[illegible]	Ana	Java	B8
[illegible]	Bob	Java	B8
[illegible]	Jack	.NET	B4

Student_Backup

Student ID	NAME	COURSE	Batch Allocated
1	Anny	PHP	B1
2	Ana	Java	B8
3	Bob	Java	B8
4	Jack	.NET	B4



Just a Minute

Which one of the following statements enables you to perform insert, update, and delete operations in a single statement?

- ○ MERGE
- ○ DELETE
- ○ TRUNCATE
- ○ UPDATE

Submit

Just a Minute

Managing Hierarchical Data in Tables

Consider a scenario where you need to store the information about the employees along with their reporting authorities in a table. In addition, you want to store the level of each employee in the organization. For example, you want the top-level employees who do not report to any other employee to be kept at level 0. However, employees who report to the employees at level 0 should be kept at level 1. Similarly, other employees should be kept at the next level of the employees whom they report to.

In the preceding scenario, the data that represents a level in a hierarchy is referred to as the hierarchical data. Another example of the hierarchical data may be the file system of an operating system. This file system may contain a root directory, which contains other subdirectories. Further, the subdirectories may contain other subdirectories.

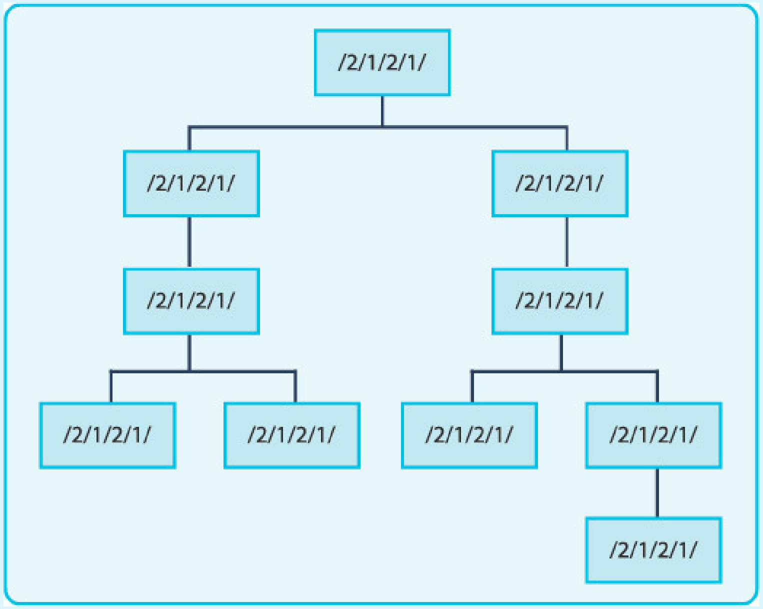
To manage the hierarchical data, SQL Server provides the hierarchyid data type. The hierarchyid data type is a variable length system-defined data type that can be used to represent a position in a hierarchy. You can create tables by using the hierarchyid data type that can store hierarchical data.

Using the Hierarchyid Data Type

A hierarchy in general is represented as a tree structure in which the data is represented in the form of parent and child nodes. In a tree structure, the nodes at a higher level are referred to as the parents of the nodes at the next level in the hierarchy. The nodes at the next level of a parent node are referred to as child nodes. A node that does not have a parent node and is placed at the top of the hierarchy is referred to as the root node. A node in a hierarchy can contain any number of child nodes. However, a node in a hierarchy must have only one parent node.

Similarly, the hierarchical data is also stored in a logical tree structure in a table. The values in the columns of the hierarchyid data type in a table are referred to as hierarchyid. Hierarchyid is the information about the position of a node in a hierarchy tree. This information is in the hexadecimal format and is generated by encoding the path from the root of the tree to that node. The information stored in hierarchyid can be represented in the form of a string that logically represents a sequence of nodes visited after the root.

The following figure represents the position of nodes in a hierarchy tree.



The Position of Nodes in a Hierarchy Tree

In the preceding figure, '/' represents the root node. The first child of the root node is '/1/'. The second child of the root node is '/2/'. This number gets incremented by one with the increase in the number of children at a particular level. For two given hierarchyid values 'a' and 'b', a<b means that 'a' comes before 'b' in the hierarchy tree. Therefore, the second child of a node is said to be greater than its first child or first child is less than the second child. Similarly, the third child of a node is said to be greater than its second child.

The node at '/1/' has the first child represented by '/1/1/' and second child represented by '/1/2/'. Similarly, if there will be third child, it will be represented as '/1/3/'. For the node at '/1/1/', the first child is represented as '/1/1/1/' and second child as '/1/1/2/'. Therefore, a node at a particular level is represented logically by a string.

The following table lists the string representation of the values stored in some of the hierarchyid nodes in a hierarchy.

String Representation	***Description***
/	*Represents the top level root node of the hierarchy.*
/1/	*Represents the first child*

	node one level below the root node.
/2/	*Represents the second child node of the root node.*
/2/1/	*Represents the first child node of the second child of the root node.*
/1/2/	*Represents the second child node of the first child of the root node.*
/n/	*Represents the nth child node of the root node.*
/n/r/	*Represents the rth child node of the nth child of the root node.*

The String Representation of the Values Stored in the Hierarchyid Nodes

To store the values stored in a column of the hierarchyid data type, you need to generate the hierarchy ids for the various nodes. These hierarchy ids represent the logical placement of a node in the hierarchy tree.

SQL Server provides the following built-in methods to generate and manage the hierarchy ids for the nodes in the hierarchy tree:

- **GetRoot()**: Returns the hierarchyid for the root node in the hierarchy tree. Therefore, this method is called first to insert the root node in the table containing the hierarchyid column. The syntax for using this method is:

  ```
  hierarchyid::GetRoot()
  ```
- **GetAncestor (n)**: Returns the hierarchyid of the nth parent of the current node. Here, n is an integer value that represents the number of levels to go up in the hierarchy. If you specify the value of n as 1, then the hierarchyid of the immediate parent of the current node will be returned. For example, if you invoke the GetAncestor(1) method on a node one level below the root, the hierarchyid of the root node will be returned. The syntax for using this method is:

  ```
  child.GetAncestor(n)
  ```

  where,

  `child` represents the hierarchyid of the node for which the nth parent is to be determined.
- **GetDescendant (Child1, Child2)**: Returns a child node of the current node based on the values specified for the Child1 and Child2 parameters. The values of these parameters are used to place the child node at a certain position among the other child nodes. The syntax for using this method is:

  ```
  parent.GetDescendant(Child1,
  Child2)
  ```

  where,

  `parent` represents the hierarchyid of the parent node.

  `Child1` and `Child2` represent the hierarchyids of the child nodes of the parent node.

  The following table describes the position of the child node to be inserted based on the values of the Child1 and Child2 parameters.

Child1	***Child2***	***Description***
NULL	*NULL*	*Returns the first child of the current node. This should be used when the first child of a node is to be inserted into the hierarchical structure.*
NOT NULL	*NULL*	*Returns a child of the current node greater than that of Child1. For example, if Child1 is /1/ in the hierarchical structure, the returned value will be /2/.*
NULL	*NOT NULL*	*Returns a child of the current node less than Child2. For example, if Child2 is /1/, then the inserted child node will be /0/.*
NOT NULL	*NOT NULL*	*Returns a child of the current node greater than Child1 but less than Child2.The child node will be inserted between Child1 and Child2. For example, if Child1 is /1/*

	and Child2 is /2/, then the inserted child node will be /1.1/.

The Position of the Child Node to be Inserted

For example, if you need to create the first child node of a node, you should pass NULL as the values of the Child1 and Child2 parameters of the GetDescendant() method. However, if you need to create a child node after an already inserted child node, you should pass the hierarchyid of the existing child node as the value of Child1 and NULL as the value of Child2 in the GetDescendant() method.

- ❑ **GetLevel()**: Returns an integer that represents the depth of the current node in the hierarchy. The syntax for using this method is:

  ```
  node.GetLevel()
  ```

  where,

  `node` represents the hierarchyid of the node for which the level is to be determined.

  For the root node, this method returns 0. This number is incremented by one for each level as you traverse down the hierarchy tree. All the child nodes of a parent node are said to be at the same level. Therefore, all the child nodes of the root node will be at level 1.
- ❑ **IsDescendantOf (node)**: Returns true if the specified hierarchyid node is the parent of the current node. Otherwise, returns false. The syntax for using this method is:

  ```
  child.IsDescendantOf(parent)
  ```

  where,

  `child` and `parent` are the hierarchyids of the nodes in the hierarchy tree.
- ❑ **Parse(string)**: Converts the specified hierarchy represented as string into a hierarchyid value. The syntax for using this method is:

  ```
  hierarchyid::Parse(string)
  ```
- ❑ **ToString()**: Returns a string that contains the logical string representation of the current hierarchyid node. The syntax for using this method is:

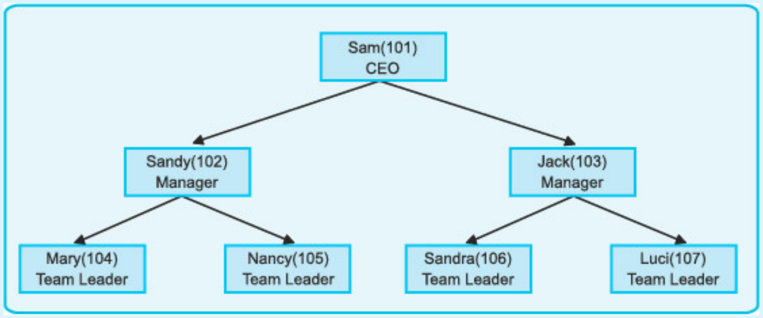
  ```
  node.ToString()
  ```

  where,

  `node` represents the hierarchyid of the node for which the equivalent string representation is to be determined.

Storing the Hierarchical Data

To store the hierarchical data, you first need to create a table with a column of the hierarchyid data type. For example, you can create a table that stores the employee data along with the reporting hierarchy of the employees. The following figure shows the reporting hierarchy of the employees.



The Reporting Hierarchy of the Employees

In the preceding figure, the name, employee ID, and designation of the employees are displayed to represent their respective position in the hierarchy.

You can use the following statement to create a table named EmployeeHierarchy in the AdventureWorks database:

```
CREATE TABLE EmployeeHierarchy
(
  HierarchyNode   hierarchyid   PRIMARY
KEY,
  HierarchyLevel                    AS
HierarchyNode.GetLevel(),
  EmployeeID int UNIQUE NOT NULL,
  EmpName varchar(20) NOT NULL,
  Designation varchar(20) NULL
)
```

In the preceding statement, the HierarchyNode column will be used to store the hierarchyids of the different nodes in the hierarchy. HierarchyLevel is a computed column that stores the hierarchy levels of the hierarchy nodes. The GetLevel() method retrieves the level of nodes and stores it in the HierarchyLevel column. This column will automatically insert the value based on the value of hierarchyid in the HierarchyNode column.

After creating the table with a column of the hierarchyid data type, you can insert the hierarchical data in this table by using the INSERT statement. For example, you can use the following INSERT statement to insert the root node (Sam, 101, CEO) in the hierarchy:

```
INSERT      INTO      EmployeeHierarchy
(HierarchyNode,  EmployeeID,  EmpName,
Designation)
VALUES  (hierarchyid::GetRoot(),  101,
'Sam', 'CEO') ;
```

The preceding statement inserts Sam as the root node in the table.

After inserting the root node, you can use the GetDescendant() method to create and insert the child nodes in the hierarchy. To insert the first child of the root node, you need to pass the NULL values as the Child1 and

Child2 parameters of the GetDescendant() method.

Before you insert the child of a node, you need to reach that node. To traverse the nodes in the hierarchy, you can use the logical string representation of the nodes. For example, the root node is represented as / in string representation. Therefore, you can use the following statements to insert the first child node (Sandy, 102, Manager) of the root node:

```
DECLARE @CEO hierarchyid
SELECT @CEO = CAST('/' AS hierarchyid)
FROM EmployeeHierarchy;
INSERT      INTO      EmployeeHierarchy
(HierarchyNode,   EmployeeID,   EmpName,
Designation)
VALUES
(@CEO.GetDescendant(NULL,  NULL),  102,
'Sandy', 'Manager')
```

In the preceding statement, the CAST() method is used to get the hierarchyid of the root node from the equivalent string representation. Further, the GetDescendant() method is used to create the first child node of the root node.

Similarly, you can create and insert a child of a node after an existing child by using the GetDescendant() method. However, to create a child node after an existing child of a node, you need to pass the hierarchyid of the existing child node and NULL as the values of the Child1 and Child2 parameters of the GetDescendant() method, respectively. For example, to insert a child node after the first child of the root node, you can use the following statements:

```
DECLARE @CEO hierarchyid
SELECT @CEO = CAST('/' AS hierarchyid)
FROM EmployeeHierarchy
DECLARE @FirstChild hierarchyid
SELECT  @FirstChild  =  CAST('/1/'  AS
hierarchyid)
FROM EmployeeHierarchy
INSERT      INTO      EmployeeHierarchy
(HierarchyNode,   EmployeeID,   EmpName,
Designation)
VALUES
(@CEO.GetDescendant(@FirstChild,
NULL), 103, 'Jack', 'Manager')
```

In the preceding statements, hierarchyid of the root and the first child node is retrieved from their string representation by using the CAST() method. Further, the GetDescendant () method is used to create the second child node of the root node.

Now, to insert Mary as a child of Sandy, you need to use the following statements:

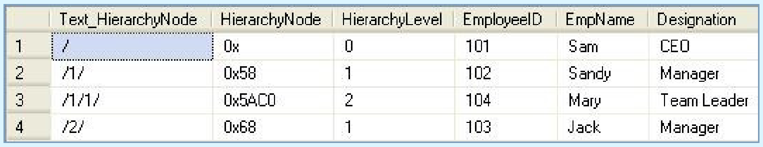
```
DECLARE @parent hierarchyid
SELECT    @parent    =    CAST('/1/'    AS
hierarchyid)
FROM EmployeeHierarchy;
INSERT      INTO      EmployeeHierarchy
(HierarchyNode,   EmployeeID,   EmpName,
Designation)
VALUES
(@parent.GetDescendant(NULL,     NULL),
104, 'Mary', 'Team Leader')
```

In the preceding statements, '/1/' represents the string representation of the position of Sandy in the hierarchy.

Similarly, you can insert other nodes in the EmployeeHierarchy table. The HierarchyNode column of this table stores the hierarchyids of each node in a hexadecimal format. However, you can use the ToString() method to convert the hierarchyid into a string. For example, you can use the following query to view the records of the EmployeeHierarchy table:

```
SELECT    HierarchyNode.ToString()    AS
Text_HierarchyNode,
HierarchyNode,          HierarchyLevel,
EmployeeID, EmpName, Designation
FROM EmployeeHierarchy
```

The following figure displays the output of the preceding query.



	Text_HierarchyNode	HierarchyNode	HierarchyLevel	EmployeeID	EmpName	Designation
1	/	0x	0	101	Sam	CEO
2	/1/	0x58	1	102	Sandy	Manager
3	/1/1/	0x5AC0	2	104	Mary	Team Leader
4	/2/	0x68	1	103	Jack	Manager

The Output Showing the Records of the EmployeeHierarchy Table

Activity 5.1: Manipulating Data in Tables

Activity 5.2: Using the MERGE Statement

Manipulating XML Data

With a growth in clients accessing data through heterogeneous hardware and software platforms, we needed a language that could be interpreted by any environment. This resulted in the evolution of a language called eXtensible Markup Language (XML). XML is a markup language that is used to describe the structure of data in a standard hierarchical manner. The structure of the documents containing the data is described with the help of tags contained in the document. Therefore, various business applications store their data in XML documents.

SQL Server allows you to save or retrieve data in the XML format. This enables SQL Server to provide database support to various applications. As a database developer, it is important for you to learn to manipulate XML data by using SQL Server.

> *The structure of XML data can be defined in the form of a supporting Document Type Definition (DTD) or schema. You can read more about XML in the Appendix.*

Storing XML Data in a Table

XML data is available in the form of XML fragments or complete XML documents. An XML fragment contains XML data without a top-level element.

The following example displays a complete XML document:

```
<customerdata>
<customer ID=C001>
  <custname>John</custname>
</customer>
<customer ID=C002>
  <custname>Peter</custname>
</customer>
</customerdata>
```

The following example displays XML data in the form of XML fragment:

```
<customer      ID=C001><custname>John</
custname></customer>
```

SQL Server uses XML as a data type to save XML data in its original state. You can create tables or variables by using this data type to store XML data. However, you can also shred XML data and store the values in different columns in a rowset. This process of transforming XML data into a rowset is called *shredding*.

In SQL Server, you can store XML data in the following ways:

- ❑ A rowset
- ❑ An XML column

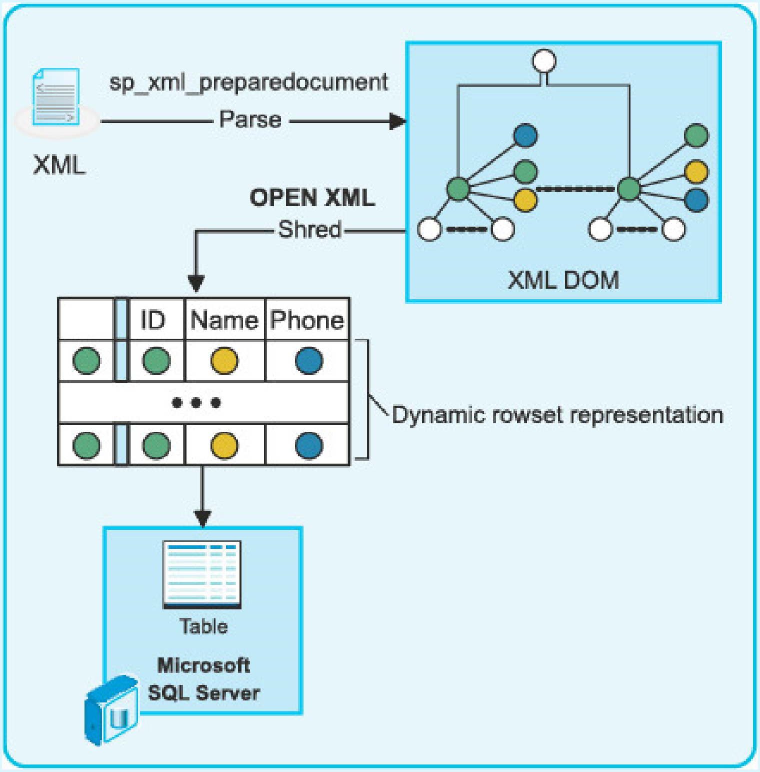
Storing XML Data in a Rowset

Consider an example. You have received order details from a vendor. The order details are generated by the application used by the vendor in an XML document. You need to store this data in a database table. For this, you need to shred the XML data. SQL Server allows you to shred the XML data by using the OPENXML function and its related *stored procedures*.

Shredding an XML document involves the following tasks:

1. Parse the XML document.
2. Retrieve a rowset from the tree.
3. Store the data from the rowset.
4. Clear the memory.

The following figure represents the shredding of an XML document.



Shredding an XML Document

In the preceding figure, you must first call the sp_xml_preparedocument procedure to parse the XML document. This procedure returns a handle to the parsed document that is ready for consumption. The parsed document is a Document Object Model (DOM) tree representation of various nodes in the XML document. The document handle is passed to OPENXML. OPENXML then provides a rowset view of the document based on the parameters passed to it. The internal representation of an XML document must be removed from memory by calling the sp_xml_removedocument system stored procedure.

Parsing the XML Document

SQL Server provides the sp_xml_preparedocument stored procedure to parse the XML document. This stored procedure reads the XML document and parses it with the

MSXML parser. *Parsing* an XML document involves validating the XML data with the structure defined in the DTD or schema. The parsed document is an internal tree representation of various nodes in the XML document, such as elements, attributes, text, and comments.

sp_xml_preparedocument returns a handle or pointer that can be used to access the newly created internal representation of the XML document. This handle is valid for the duration of the session or until the handle is destroyed by executing the sp_xml_removedocument stored procedure.

Retrieving a Rowset from the Tree

After verifying the accuracy of the structure and completeness of data, you need to extract the data from the available XML data. For this, you can use the OPENXML function to generate an in-memory rowset from the parsed data.

The syntax of the OPENXML function is:

```
OPENXML( idoc int [ in], rowpattern
nvarchar [ in ], [ flags byte
[ in ] ] )
[ WITH ( SchemaDeclaration |
TableName ) ]
```

where,

`idoc` specifies the document handle of the internal tree representation of an XML document.

`rowpattern` specifies the XPath pattern used to identify the nodes (in the XML document whose handle is passed in the idoc parameter) to be processed as rows. The row pattern parameter of OPENXML allows you to traverse the XML hierarchy.

`flags` indicate the mapping that should be used between the XML data and the relational rowset. It is an optional parameter and can have the following values:

- ❑ `0`: To use the default mapping (attributes).
- ❑ `1`: To retrieve attribute values.
- ❑ `2`: To retrieve element values.
- ❑ `3`: To retrieve both attribute and element values.

`SchemaDeclaration` specifies the rowset schema declaration for the columns to be returned by using a combination of column names, data types, and patterns.

`TableName` specifies the table name that can be given instead of SchemaDeclaration if a table with the desired schema already exists and no column patterns are required.

Storing Data from the Rowset

You can use the rowset created by OPENXML to store data in the same way that you would use any other rowset. You can insert the rowset data into permanent tables in a database. For example, you can insert the data received by a supplier in the XML format into the SalesOrderHeader and SalesOrderDetail tables.

Clearing the Memory

After saving the data permanently in the database, you need to release the memory where you stored the rowset. For this, you can use the sp_xml_removedocument stored procedure.

Consider an example, where customers shop online and the orders given by the customers are transferred to the supplier in the form of an XML document. The following data is available in the XML document:

```
DECLARE @Doc int
DECLARE @XMLDoc nvarchar(1000)
SET @XMLDoc = N'<ROOT>
<Customer CustomerID="JH01"
ContactName="John Henriot">
  <Order OrderID="1001"
CustomerID="JH01">
      <OrderDate="2006-07-04T00:00:00"
>
    <OrderDetail ProductID="11"
Quantity="12"/>
    <OrderDetail ProductID="22"
Quantity="10"/>
  </Order>
</Customer>
<Customer CustomerID="SG01"
ContactName="Steve Gonzlez">
    <Order OrderID="1002"
CustomerID="SG01">
      OrderDate="2006-08-16T00:00:00">
    <OrderDetail ProductID="32"
Quantity="3"/>
  </Order>
 </Customer>
</ROOT>'
```

To view the preceding XML data in a rowset, you need to perform the following steps:

1. Create an internal representation of the XML document by executing the following statement:
   ```
   EXEC sp_xml_preparedocument @Doc
   OUTPUT, @XMLDoc
   ```
2. Execute the following statement to store the data in a table, CustomerDetails by using the OPENXML function:
   ```
   SELECT * INTO CustomerDetails
   FROM OPENXML (@Doc, '/ROOT/
   Customer', 1)
       WITH (CustomerID varchar(10),
         ContactName varchar(20))
   ```
   In the preceding statement, the OPENXML function takes three parameters. The first parameter, @Doc is the document handle, which stores the internal representation of the XML document. The second parameter specifies that the Customer element is to be processed. The third parameter, 1 allows you to retrieve the attribute values of the Customer element. The WITH clause specifies the column names along

with the data types where attribute values will be stored. When the preceding query is executed, the details of the customers are stored in the CustomerDetails table, as shown in the following table.

CustomerID	*ContactName*
JH01	*John Henriot*
SG01	*Steve Gonzlez*

Representation of the XML Data in a Tabular Format

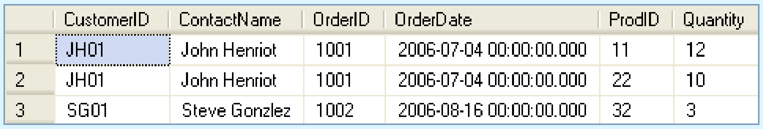
3. Remove the internal tree from the memory by executing the following statement:
   ```
   EXEC sp_xml_removedocument @Doc
   ```

You can also specify the column pattern to map the rowset columns and the XML attributes and elements. For example, consider the following statements:

```
DECLARE @Doc int
DECLARE @XMLDoc nvarchar(1000)
SET @XMLDoc = N'<ROOT>
<Customer             CustomerID="JH01"
ContactName="John Henriot">
<Order                   OrderID="1001"
CustomerID="JH01">
     <OrderDate="2006-07-04T00:00:00">
     <OrderDetail        ProductID="11"
Quantity="12"/>
     <OrderDetail        ProductID="22"
Quantity="10"/>
   </Order>
</Customer>
<Customer             CustomerID="SG01"
ContactName="Steve Gonzlez">
<Order                   OrderID="1002"
CustomerID="SG01">
     OrderDate="2006-08-16T00:00:00">
     <OrderDetail        ProductID="32"
Quantity="3"/>
</Order>
</Customer>
</ROOT>'
EXEC     sp_xml_preparedocument     @Doc
OUTPUT, @XMLDoc
SELECT *
FROM  OPENXML  (@Doc,  '/ROOT/Customer/
Order/OrderDetail',1)
  WITH (CustomerID varchar(10) '../../
@CustomerID',
    ContactName   varchar(20)   '../../
@ContactName',
      OrderID int '../@OrderID',
      OrderDate      datetime      '../
@OrderDate',
      ProdID int '@ProductID',
      Quantity int)
EXEC sp_xml_removedocument @Doc
```

In the preceding statements, the second parameter (/ROOT/Customer/Order/OrderDetail) of the OPENXML function specifies that the current node is OrderDetail, which needs to be processed. The WITH clause specifies the column names, data types, and patterns. Here, the '@' symbol specifies the attribute, and the double dot (..) symbol represents the parent node of the current node. Therefore, the expression, ../../@CustomerID, in the WITH clause represents the CustomerID attribute of the Customer element.

The output of the preceding statements is displayed in the following figure.



	CustomerID	ContactName	OrderID	OrderDate	ProdID	Quantity
1	JH01	John Henriot	1001	2006-07-04 00:00:00.000	11	12
2	JH01	John Henriot	1001	2006-07-04 00:00:00.000	22	10
3	SG01	Steve Gonzlez	1002	2006-08-16 00:00:00.000	32	3

The Ouput Derived After Shredding an XML Document

Storing XML Data in a Rowset

What is Shredding?

Consider the following XML file:

```
<Studentdata>
tudent ID=C001>
dentName>John</StudentName>
hNumber>AW0001</BatchNumber>
udent>
<Student ID=C002>
<StudentName>Peter</StudentName>
<BatchNumber>AW0002</BatchNumber>
</Student>
<Student ID=C003>
<StudentName>Martin</StudentName>
<BatchNumber>AW0003</BatchNumber>
</Student>
</Studentdata>
```

Importing XML data into SQL Server

At times, you need to import the entire data contained in an XML file into a table in SQL Server. For example, the details of the students are stored in a file named Students.xml. The OPENROWSET() function can be used to connect with and access remote data stored in the XML document. It can be used in the FROM clause of the SELECT statement.

The following syntax is used for the OPENROWSET() function to perform a bulk import operation:

```
OPENROWSET( BULK,'file-name',Encoding-Scheme)
```

where,

`BULK` is a keyword to be used for bulk import.

`file-name` is the name of the XML file to be imported.

`Encoding-Scheme` is the scheme to be used. SINGLE_BLOB, if provided as the encoding scheme, ensures that the XML parser in SQL Server imports the data according to the encoding scheme specified in the XML declaration.

Further, XQuery can be used to parse XML into a tabular format.

Consider the following Students.xml file, which is stored on the C drive of your local machine:

```
<Students>
<Student>
   <SID>1</SID>
   <Marks>70</Marks>
   <Grade>B</Grade>
</Student>
<Student>
   <SID>2</SID>
   <Marks>80</Marks>
   <Grade>A</Grade>
  </Student>
</Students>
```

You need to import this XML file into a table named STUDENTS, which has been created by using the following statement:

```
CREATE  TABLE  STUDENTS(SID  int  PRIMARY
KEY, Marks int, Grade varchar(5))
```

Use the following statement to insert the data into the table, STUDENTS, from the document, STUDENTS.xml:

```
INSERT INTO STUDENTS(SID,Marks,Grade)
SELECT   x.Stud.QUERY('SID').VALUE('.',
'INT'),
x.Stud.QUERY('Marks').VALUE('.',
'INT'),
x.Stud.QUERY('Grade').VALUE('.',
'VARCHAR(5)')
FROM
(SELECT  CAST(x  AS  XML)FROM  OPENROWSET
(BULK  'd:\Students.xml',SINGLE_BLOB)AS
T(x))
AS  T(x)CROSS  APPLY  x.nodes('Students/
Student') AS X(Stud);
```

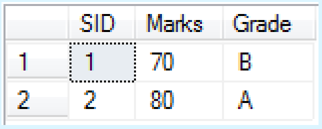
In the preceding statement, the QUERY() method is used to retrieve XML data from an XML instance.

The `VALUE()` method is used to retrieve the value of the XML type into the SQL type that has been specified. The single dot (.) used inside the `VALUE()` method represents the current element.

The following statement is used to display the data from the table Students:

```
SELECT * FROM Students
```

The output of the preceding statement is displayed in the following figure.



	SID	Marks	Grade
1	1	70	B
2	2	80	A

The Output Derived After Using the Select Statement

Storing XML Data in XML Columns

At times, you need to store XML data in its original state in a column of a database table. For example, you need to save the details of customers in the database. The details of individual customers are maintained by a website. The website saves the details of each customer in an XML file. As a database developer, you need to save this data in SQL Server. For this, you can create the CustDetails table to store the customer details, as shown in the following statement:

```
CREATE TABLE CustDetails
(
CUST_ID int,
CUST_DETAILS XML
)
```

You can save the following types of data in the columns with the XML data types:

- ❑ **Untyped XML data**: Is also a well-formed data, but is not associated with a schema. SQL Server does not validate this data but ensures that the data being saved with the XML data type is well-formed.
- ❑ **Typed XML data**: Is a well-formed data that is associated with a schema defining the elements and their attributes. It also specifies a namespace for the data. When you save the typed XML data in a table, SQL Server validates the data against the schema. Then, SQL Server assigns the appropriate data type to the data based on the data types defined in the schema. This helps in saving the storage space.

As a database developer, you should know how to store both types of data on SQL Server.

Storing Untyped XML Data

To store untyped XML data, you can use columns or variables with the XML data type. For example, to store customer data in the CustDetails table, you can use the following INSERT INTO statement:

```
INSERT   INTO   CustDetails   VALUES   (2,
'<Customer     Name="Abrahim     Jones"
City="Selina" />')
```

In the preceding statement, the string value that contains an XML fragment is implicitly converted to XML. However, you can also convert a string value to XML by using the CONVERT or CAST functions. In this example, you can use the following statement to convert the data type of the string value to XML before inserting the record into the table:

```
INSERT   INTO   CustDetails   VALUES(2,
convert(XML,'<Customer   Name="Abrahim
Jones" City="Selina" />'))
```

Similarly, you can also use the CAST function, as shown in the following statement:

```
INSERT INTO CustDetails VALUES(4, cast
('<Customer Name="Abrahim Jones"
City="Selina" />' as XML))
```

Storing Typed XML Data

To store typed XML data, you need to first register an XML schema in the XML schema collection objects in the database. The XML schema collection is an object on SQL Server that is used to save one or more XML schemas. You can create an XML schema collection object by using the following statement:

```
CREATE XML SCHEMA COLLECTION <Name> as
Expression
```

where,

`Name` specifies an identifier name with which SQL Server will identify the schema collection.

`Expression` specifies an XML value that contains one or more XML schema documents.

For example, the customer details are associated with the following schema:

```
<schema xmlns="http://www.w3.org/2001/
XMLSchema">
   <element name="CustomerName"
type="string"/>
   <element name="City" type="string"/
>
</schema>'
```

You can use the following statement to register the preceding schema with the database:

```
CREATE XML SCHEMA COLLECTION
CustomerSchemaCollection AS
'<schema xmlns="http://
www.w3.org/2001/XMLSchema">
   <element name="CustomerName"
type="string"/>
   <element name="City" type="string"/
>
</schema>'
```

The preceding statement creates a XML schema collection object named CustomerSchemaCollection.

> *You can view information about the registered schemas in a database by querying the sys.XML_schema_collections catalog view, as shown in the following query: `SELECT * FROM sys.XML_schema_collections`*

> *To drop an XML Schema Collection from the database, you can use the following statement: `DROP XML SCHEMA COLLECTION CustomerSchemaCollection`*

After registering the XML schema, you can use the schemas to validate typed XML values while inserting records into the tables. You need to specify this while creating a table that will store the XML data. In the preceding example, if you need to validate the customer details with the CustomerSchemaCollection schema, you need to create the Customer_Details table by using the following statement:

```
CREATE TABLE Customer_Details
(
CustID int,
CustDetail XML
(CustomerSchemaCollection)
)
```

You can insert the data in the Customer_Details table by using the following statement:

```
INSERT INTO Customer_Details
VALUES(2,'<CustomerName>Abrahim
Jones</CustomerName><City>Selina</
City>')
```

While executing the preceding statement, SQL Server will validate the values for the CustDetails column against the CustomerSchemaCollection schema.

Insert another record using the following statement:

```
INSERT INTO Customer_Details
VALUES(2,'<Name>John</Name><City>New
York</City>')
```

In the preceding statement, the value of CustDetail column is not following the schema definition. The execution of the preceding statement will produce the following error:

```
Msg 6913, Level 16, State 1, Line 1
XML Validation: Declaration not found
for element 'Name'. Location: /*:Name
[1]
```

Retrieving Table Data into XML Format

At times, you need to retrieve the relational data from a table into the XML format for reporting purposes or to share the data across different applications. This involves extracting data from a table in the form of well-formed XML fragments. You can retrieve the data in XML format by using:

- ❑ The FOR XML clause in the SELECT statement.
- ❑ XQuery.

Using the FOR XML Clause in the SELECT Statement

SQL Server allows you to extract data from relational tables into an XML format by using the SELECT statement with the FOR XML clause. You can use the FOR XML clause to retrieve the XML data by using the following modes:

- ❑ RAW
- ❑ AUTO
- ❑ PATH

❑ EXPLICIT

Using the RAW Mode

The RAW mode is used to return an XML file with each row representing an XML element. The RAW mode transforms each row in the query result set into an XML element with the element name, row. Each column value that is not NULL is mapped to an attribute with the same name as the column name.

The following query displays the details of employees with employee ID as 1 or 2:

```
SELECT EmployeeID, ContactID, LoginID,
Title
FROM HumanResources.Employee
WHERE EmployeeID=1 OR EmployeeID=2
FOR XML RAW
```

The preceding query displays the employee details in the following format:

```
<row  EmployeeID="1"  ContactID="1209"
Loginid="adventure-works\guy1"
Title="Production Technician - WC60" /
>
<row  EmployeeID="2"  ContactID="1030"
Loginid="adventure-works\kevin0"
Title="Marketing Assistant" />
```

If the ELEMENTS directive is specified with the FOR XML clause, each column value is mapped to a subelement of the <row> element, as shown in the following query:

```
SELECT EmployeeID, ContactID, LoginID,
Title
FROM HumanResources.Employee
WHERE EmployeeID=1 OR EmployeeID=2
FOR XML RAW, ELEMENTS
```

The preceding query displays the employee details in the following format:

```
<row>
  <EmployeeID>1</EmployeeID>
  <ContactID>1209</ContactID>
  <LoginID>adventure-works\guy1</
LoginID>
  <Title>Production    Technician    -
WC60</Title>
</row>
<row>
  <EmployeeID>2</EmployeeID>
  <ContactID>1030</ContactID>
  <LoginID>adventure-works\kevin0</
LoginID>
  <Title>Marketing Assistant</Title>
</row>
```

When element-centric XML is returned, null columns are omitted in the results. However, you can specify that null columns should yield empty elements with the xsi:nil attribute instead of being omitted. For this, you can use the XSINIL option with the ELEMENTS directive in AUTO, RAW, and PATH mode queries. For example, the following query displays the product details:

```
SELECT ProductID, Name, Color
FROM Production.Product Product
WHERE ProductID = 1 OR ProductID = 317
FOR XML RAW, ELEMENTS XSINIL
```

In the preceding query, the value of Color column for ProductID 1 is NULL. When you execute the preceding query, the output will be displayed in the following format:

```
<row                 xmlns:xsi="http://
www.w3.org/2001/XMLSchema-instance">
  <ProductID>1</ProductID>
  <Name>Adjustable Race</Name>
  <Color xsi:nil="true" />
</row>
<row                 xmlns:xsi="http://
www.w3.org/2001/XMLSchema-instance">
  <ProductID>317</ProductID>
  <Name>LL Crankarm</Name>
  <Color>Black</Color>
</row>
```

In the preceding format, for ProductID 1, the xsi:nil attribute is added for the NULL value in the Color column.

In EXPLICIT mode queries, you can use the elementxsinil column mode to yield empty elements for the columns having NULL.

Using the AUTO Mode

The AUTO mode is used to return query results as nested XML elements. Similar to the RAW mode, each column value that is not NULL is mapped to an attribute that is named after either the column name or the column alias. The element that these attributes belong to is named to the table that they belong to or to the table alias that is used in the SELECT statement, as shown in the following query:

```
SELECT EmployeeID, ContactID, LoginID,
Title
FROM HumanResources.Employee Employee
WHERE EmployeeID=1 OR EmployeeID=2
FOR XML AUTO
```

The preceding query displays employee details in the following format:

```
<Employee                EmployeeID="1"
ContactID="1209"     Loginid="adventure
works\guy1"            Title="Production
Technician - WC60" />
<Employee                EmployeeID="2"
ContactID="1030"    Loginid="adventure-
works\kevin0"            Title="Marketing
Assistant" />
```

If the optional ELEMENTS directive is specified in the FOR XML clause, the columns listed in the SELECT clause are mapped to subelements, as shown in the following query:

```
SELECT EmployeeID, ContactID, LoginID,
Title
FROM HumanResources.Employee Employee
WHERE EmployeeID=1 OR EmployeeID=2
FOR XML AUTO, ELEMENTS
```

The output of the preceding query is displayed in the following format:

```
<Employee>
  <EmployeeID>1</EmployeeID>
  <ContactID>1209</ContactID>
  <LoginID>adventure-works\guy1</
LoginID>
  <Title>Production Technician -
WC60</Title>
</Employee>
<Employee>
  <EmployeeID>2</EmployeeID>
  <ContactID>1030</ContactID>
  <LoginID>adventure-works\kevin0</
LoginID>
  <Title>Marketing Assistant</Title>
</Employee>
```

Using the PATH Mode

The PATH mode is used to return specific values by indicating the column names for which you need to retrieve the data, as shown in the following query:

```
SELECT EmployeeID "@EmpID",
  FirstName "EmpName/First",
  MiddleName "EmpName/Middle",
  LastName "EmpName/Last"
FROM HumanResources.Employee e JOIN
Person.Contact c
ON e.ContactID = c.ContactID
AND e.EmployeeID=1
FOR XML PATH
```

The preceding query displays the output in the following format:

```
<row EmpID="1">
 <EmpName>
  <First>Guy</First>
  <Middle>R</Middle>
  <Last>Gilbert</Last>
 </EmpName>
</row>
```

In the preceding format, the EmployeeID column is mapped to the EmpID attribute, and FirstName, MiddleName, and LastName columns are mapped as subelements of the EmpName element. A node preceded by '@' symbol represents an attribute. Subelements are preceded by the parent element name followed by the /.

You can also use the optional ElementName argument with the PATH mode query to modify the name of the default row element, as shown in the following query:

```
SELECT EmployeeID "@EmpID",
  FirstName "EmpName/First",
  MiddleName "EmpName/Middle",
  LastName "EmpName/Last"
FROM HumanResources.Employee e JOIN
Person.Contact c
ON e.ContactID = c.ContactID
AND e.EmployeeID=1
FOR XML PATH('Employee')
```

The preceding query displays the output in the following format:

```
<Employee EmpID="1">
 <EmpName>
  <First>Guy</First>
  <Middle>R</Middle>
  <Last>Gilbert</Last>
 </EmpName>
</Employee>
```

In the preceding format, the Employee element becomes the root element.

Using the EXPLICIT Mode

The EXPLICIT mode is used to return an XML file that obtains the format as specified in the SELECT statement. Separate SELECT statements can be combined with the UNION ALL statement to generate each level/element in the resulting XML output. Each of these SELECT statements requires the first two tags to be called Tag and Parent. The Parent element is used to control the nesting of elements. It contains the tag number of the parent element of the current element. The top-level element in the document should have the Parent value set to 0 or NULL.

While writing EXPLICIT mode queries, column names in the resulting rowset must be specified in the following format:

```
ElementName!TagNumber!AttributeName!
Directive
```

where,

`ElementName` specifies the name of the element.

`TagNumber` specifies the unique tag value assigned to an element. It, along with the value in the Parent tag, determines the nesting of the elements in the resulting XML.

`AttributeName` specifies the name of the attribute. This attribute will be created in the element specified by the ElementName, if the directive is not specified.

`Directive` specifies the type of AttributeName. It is used to provide additional information for construction of the XML. It is optional and can have values, such as xml, cdata, or element. If you specify the element, it will generate a subelement instead of an attribute.

For example, the managers of AdventureWorks want to access the information regarding products through their mobile devices. These devices cannot directly connect to SQL Server but can read the data provided in the XML format. Therefore, you need to convert the details of the products from the Product table into the XML document.

To perform this task, you need to create an XML document with <Product> as the parent tag. The <Product> tag will contain ProductID as an attribute, and <ProductName> and <Color> as child elements.

To perform this task, you can create the following query:

```
SELECT 1 AS Tag,
  NULL AS Parent,
  ProductID AS [Product!1!ProductID],
  Name AS [Product!1!ProductName!
element],
  Color AS [Product!1!Color!
elementxsinil]
FROM Production.Product
Where ProductID = 1 OR ProductID = 317
FOR XML EXPLICIT
```

The preceding query assigns 1 as Tag value for the <Product> element and NULL as Parent because <Product> is the top-level element.

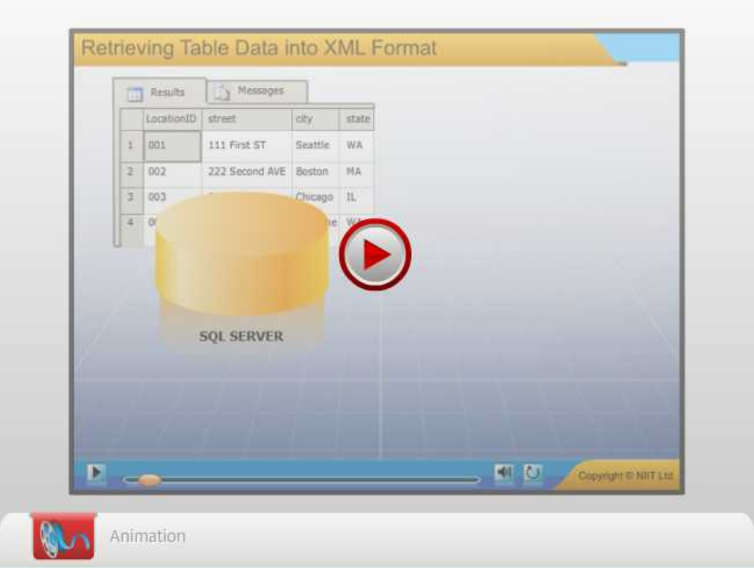
The Product!1!ProductID column specifies that the ProductId will be the attribute of the Product element. If the type of the node is not specified, it will be an attribute by default.

The Product!1!ProductName!element column specifies that the ProductName will be the child element of Product as the type is specified as element.

The Product!1!Color!elementxsinil column specifies that the Color element will be the child element of Product. It also generates the element for null values as the type is specified as elementxsinil.

The execution of the preceding query generates the output in the following format:

```
<Product xmlns:xsi="http://
www.w3.org/2001/XMLSchema-instance"
ProductID="1">
  <ProductName>Adjustable Race</
ProductName>
  <Color xsi:nil="true" />
</Product>
<Product xmlns:xsi="http://
www.w3.org/2001/XMLSchema-instance"
ProductID="317">
  <ProductName>LL Crankarm</
ProductName>
  <Color>Black</Color>
</Product>
```



	LocationID	street	city	state
1	001	111 First ST	Seattle	WA
2	002	222 Second AVE	Boston	MA
3	003	[illegible]	Chicago	IL
4	0	[illegible]	e	W

Including XML Namespaces

An XML namespace is a collection of elements and attributes. It is used to resolve naming conflicts among the elements. The elements belonging to a particular namespace cannot have similar names. The name of the XML namespace is in the form of Uniform Resource Identifier (URI). If needed, it can also be given an alias name.

You can add a namespace to the XML data retrived from the database tables. The WITH NAMESPACES clause can be used with the FOR XML clause to specify the namespace and the alias.

Consider the following statement:

```
WITH XMLNAMESPACES ('http://
www.adventureWorks.com' AS ad)
SELECT EmployeeID AS 'ad:EMPID',
  ContactID AS 'ad:ContactID',
  Title AS 'ad:Title'
FROM HumanResources.Employee Employee
FOR XML RAW ('ad:Employee'), ELEMENTS
```

In the preceding statement, the data is retrieved in an XML format from the table, Employee. The WITH XMLNAMESPACES clause accepts two arguments. The first argument is the name of the namespace in URI format, and the second argument is the alias given to the namespace.

Once executed, the preceding statement gives the following output:

```
ad:Employee xmlns:ad="http://
www.adventureWorks.com">
  <ad:EMPID>1</ad:EMPID>
  <ad:ContactID>1209</ad:ContactID>
  <ad:Title>Production Technician -
WC60</ad:Title>
</ad:Employee>
<ad:Employee xmlns:ad="http://
www.adventureWorks.com">
  <ad:EMPID>2</ad:EMPID>
  <ad:ContactID>1030</ad:ContactID>
  <ad:Title>Marketing      Assistant</
ad:Title>
</ad:Employee>
```

Using XQuery

In addition to FOR XML, SQL Server allows you to extract data stored in variables or columns with the XML data type by using XQuery. XQuery is a language that uses a set of statements and functions provided by the XML data type to extract data. As compared to the FOR XML clause of the SELECT statement, the XQuery statements allow you to extract specific parts of the XML data.

Each XQuery statement consists of two parts, prolog and body. In the prolog section, you declare the namespaces. In addition, schemas can be imported in the prolog. The body part specifies the XML nodes to be retrieved. It contains query expressions that define the result of the query. The XQuery language supports the following expressions:

- **for**: Used to iterate through a set of nodes at the same level as in an XML document.
- **let**: Used to declare variables and assign values.
- **order by**: Used to specify a sequence.
- **where**: Used to specify criteria for the data to be extracted.
- **return**: Used to specify the XML returned from a statement.

The preceding expressions are defined by the acronym FLOWR (pronounced "flower"). The XQuery statements also use the following functions provided by the XML data type:

- **Query**: Used to extract XML from an XML data type. The XML to be retrieved is specified by the XQuery expression passed as a parameter.

  For example, the following DeliverySchedule table in the Sales schema stores the delivery details of the sales orders.

Column Name	*Data Type*
ScheduleID	*int IDENTITY PRIMARY KEY*
ScheduleDate	*DateTime*
DeliveryRoute	*int*
DeliveryDriver	*nvarchar(20)*
DeliveryList	*Xml*

  The DeliverySchedule Table

  A record is inserted into the DeliverySchedule table by using the following statement:

  ```
  INSERT INTO
  Sales.DeliverySchedule
  VALUES
  (GetDate(), 3, 'Bill',
  '<?xml version="1.0" ?>
  <DeliveryList xmlns="http://
  schemas.adventure works.com/
  DeliverySchedule">
    <Delivery SalesOrderID="43659">
      <CustomerName>Steve Schmidt</
  CustomerName>
      <Address>6126 North Sixth
  Street, Rockhampton</Address>
    </Delivery>
    <Delivery SalesOrderID="43660">
      <CustomerName>Tony Lopez</
  CustomerName>
      <Address>6445 Cashew Street,
  Rockhampton</Address>
    </Delivery>
  </DeliveryList>')
  ```

  The following query retrieves the delivery driver and the name of the customers from the DeliverySchedule table:

  ```
  SELECT DeliveryDriver,
  DeliveryList.query
    ('declare namespace ns="http://
  schemas.adventure works.com/
  DeliverySchedule";
  ns:DeliveryList/ns:Delivery/
  ns:CustomerName') as 'Customer
  Names'
  FROM Sales.DeliverySchedule
  ```

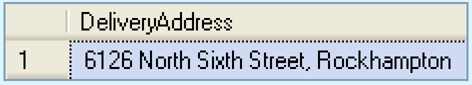
  In the preceding query, the query() function is used to retrieve the customer names. This function is taking the path of the CustomerName element as a parameter. Here, ns is the prefix of the namespace used in the XML document.

- **Value**: Used to return a single value from an XML document. To extract a single value, you need to specify an XQuery expression that identifies a single node and a data type of the value to be retrieved.

  For example, the following query retrieves the address of the first delivery:

  ```
  SELECT DeliveryList.value
  ('declare namespace ns="http://
  schemas.adventure works.com/
  DeliverySchedule";
    (ns:DeliveryList/ns:Delivery/
  ns:Address)[1]', 'nvarchar(100)')
  DeliveryAddress
  FROM Sales.DeliverySchedule
  ```

  In the preceding query, the value() function takes two parameters, the path of Address element and its data type. Here, '(ns:DeliveryList/ns:Delivery/ns:Address)[1]' represents the index of the first address. The preceding query will display the result, as shown in the following figure.



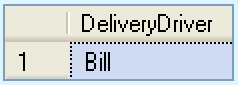
	DeliveryAddress
1	6126 North Sixth Street, Rockhampton

The Ouput Derived After Using the value() function

- **Exist**: Used to check the existence of a node in an XML data. The function returns 1 if the specified node exists, else it returns 0. The following query finds the driver for a specific order:

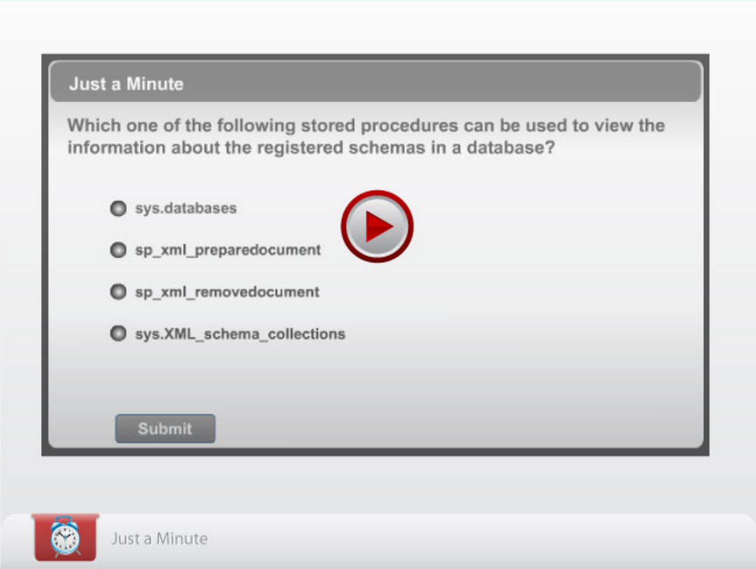
```
SELECT DeliveryDriver
FROM Sales.DeliverySchedule
WHERE DeliveryList.exist
  ('declare namespace ns="http://
schemas.adventure works.com/
DeliverySchedule";
  /ns:DeliveryList/ns:Delivery
[@SalesOrderID=43659]') = 1
```

The preceding query returns the driver for the order whose sales order ID is 1, as shown in the following figure.



	DeliveryDriver
1	Bill

The Ouput Derived After Using the SELECT statement



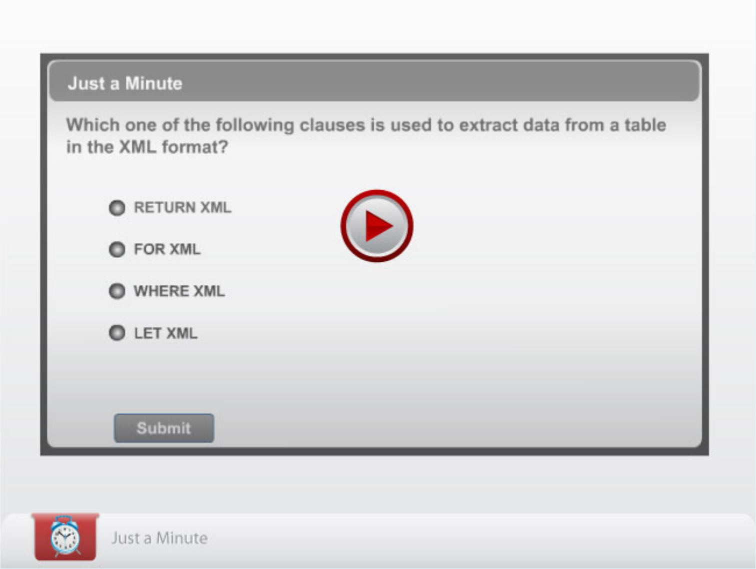
Which one of the following stored procedures can be used to view the information about the registered schemas in a database?

○ sys.databases

○ sp_xml_preparedocument

○ sp_xml_removedocument

○ sys.XML_schema_collections



Which one of the following clauses is used to extract data from a table in the XML format?

○ RETURN XML

○ FOR XML

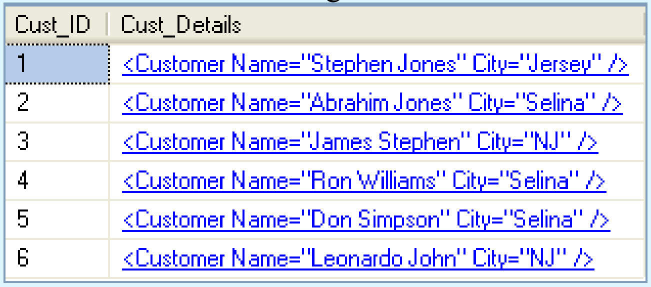
○ WHERE XML

○ LET XML

Modifying XML Data

Similar to any other type of data, you might also need to modify XML data. For this, you can use the modify function provided by the XML data type of SQL Server. The modify function specifies an XQuery expression and a statement that specifies the kind of modification that needs to be done. This function allows you to perform the following modifications:

- **Insert**: Used to add nodes to XML in an XML column or variable. For example, in the Adventureworks database, the customer details are stored in the following CustomDetails table.



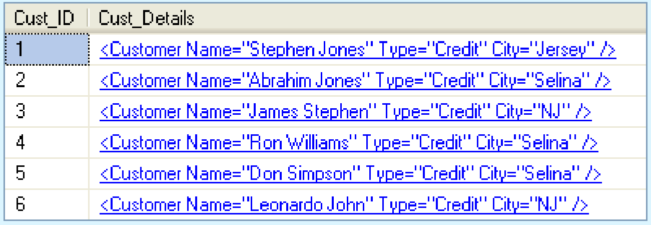
Cust_ID	Cust_Details
1	<Customer Name="Stephen Jones" City="Jersey" />
2	<Customer Name="Abrahim Jones" City="Selina" />
3	<Customer Name="James Stephen" City="NJ" />
4	<Customer Name="Ron Williams" City="Selina" />
5	<Customer Name="Don Simpson" City="Selina" />
6	<Customer Name="Leonardo John" City="NJ" />

The CustomDetails Table

In the preceding table, the customer details are stored in an XML format in the Cust_Details column. The management of AdventureWorks wants to store the type of the customer in the CustomDetails table. This can be done by adding an attribute, Type in the XML data of the Cust_Details column. The default value of the Type attribute should be 'Credit'. To perform the required task, you can create the following statement:

```
UPDATE      CustomDetails      SET
Cust_Details.modify('
insert  attribute  Type{"Credit"}
as first
into (/Customer)[1]')
```

The execution of the preceding statement adds the Type attribute with a default value, Credit. The 'as first' clause specifies that the attribute will be inserted in the beginning of all attributes. After the addition of the attribute, the CustomDetails table will contain the data, as shown in the following figure.



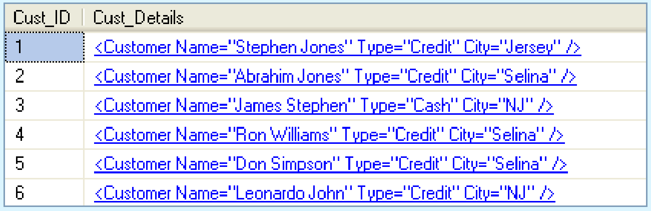
Cust_ID	Cust_Details
1	<Customer Name="Stephen Jones" Type="Credit" City="Jersey" />
2	<Customer Name="Abrahim Jones" Type="Credit" City="Selina" />
3	<Customer Name="James Stephen" Type="Credit" City="NJ" />
4	<Customer Name="Ron Williams" Type="Credit" City="Selina" />
5	<Customer Name="Don Simpson" Type="Credit" City="Selina" />
6	<Customer Name="Leonardo John" Type="Credit" City="NJ" />

The CustomDetails Table After Adding a New Attribute

- ❑ **Replace**: Used to update XML data. For example, James Stephen, one of the customers of AdventureWorks, has decided to change his customer type from Credit to Cash. As a database developer, you can create the following statement to reflect this change:

```
UPDATE      CustomDetails      SET
Cust_Details.modify ('
replace  value  of(Customer/@Type)
[1] with "Cash"') WHERE Cust_ID =
3
```

In the preceding statement, the value of the Type attribute will be replaced as cash for the customer whose customer ID is 3. After replacing the attribute, the CustomDetails table will contain the data, as shown in the following figure.



Cust_ID	Cust_Details
1	<Customer Name="Stephen Jones" Type="Credit" City="Jersey" />
2	<Customer Name="Abrahim Jones" Type="Credit" City="Selina" />
3	<Customer Name="James Stephen" Type="Cash" City="NJ" />
4	<Customer Name="Ron Williams" Type="Credit" City="Selina" />
5	<Customer Name="Don Simpson" Type="Credit" City="Selina" />
6	<Customer Name="Leonardo John" Type="Credit" City="NJ" />

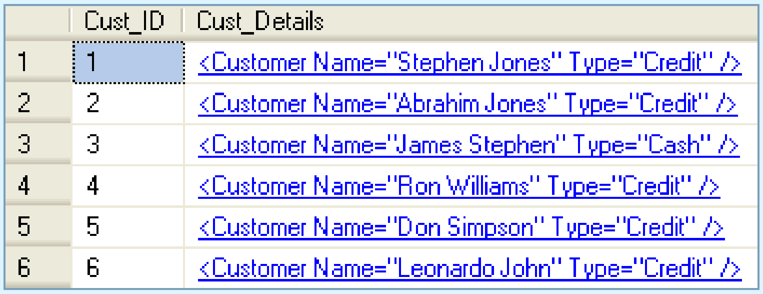
The CustomDetails Table After Replacing the Attribute Value

- ❑ **Delete**: Used to remove a node from XML data. For example, the management of AdventureWorks has decided to remove the details of the city from the customer details. You can write the following statement to remove the City attribute:

```
UPDATE      CustomDetails      SET
Cust_Details.modify (
'delete(/Customer/@City)[1]')
```

The preceding statement deletes the City attribute from the Cust_Details column in the CustomDetails table.

After deleting the attribute, the CustomDetails table will contain the data, as shown in the following figure.



	Cust_ID	Cust_Details
1	1	<Customer Name="Stephen Jones" Type="Credit" />
2	2	<Customer Name="Abrahim Jones" Type="Credit" />
3	3	<Customer Name="James Stephen" Type="Cash" />
4	4	<Customer Name="Ron Williams" Type="Credit" />
5	5	<Customer Name="Don Simpson" Type="Credit" />
6	6	<Customer Name="Leonardo John" Type="Credit" />

The CustomDetails Table After Deleting the Attribute

Activity 5.3: Working with XML Data

Summary

In this chapter, you learned that:

- ❑ The INSERT statement is used to insert data into a table.
- ❑ While inserting data into a table, the data type of the information must match the data types of the columns of the table.
- ❑ If a default value is not specified for a column, and the column property is specified as NULL, NULL is inserted into the column.
- ❑ A column in a table for which the values are automatically generated by the database is called an identity column.
- ❑ You can copy contents from one table into another table by using the SELECT INTO statement.
- ❑ You can insert the FILESTREAM data into a table as a value of the varbinary (max) data type.
- ❑ SQL server provides you with the GEOGRAPHY data type to store geographic locations as points.
- ❑ You can use the UPDATE statement to make changes in a table.
- ❑ You can delete a row from a table by using the DELETE DML statement.
- ❑ You use the TRUNCATE TABLE statement to remove all the rows from a table.
- ❑ The OUTPUT clause is used to retrieve the set of rows affected by the DML statements.
- ❑ The MERGE statement enables you to perform insert, update, and delete operations in a single statement.
- ❑ SQL Server uses XML as a data type to save XML data in its original state.
- ❑ SQL Server allows you to shred XML data by using the OPENXML function and its related stored procedures.
- ❑ Untyped XML data is a well-formed data, but is not associated with a schema.

- Typed XML data is a well-formed data that is associated with a schema.
- You can create an XML schema collection object by using the CREATE XML SCHEMA COLLECTION statement.
- You can use the FOR XML clause of the SELECT statement to retrieve the XML data in different ways by using the RAW, AUTO, PATH, and EXPLICIT modes.
- You can use the XQuery functions to extract XML data stored in a column with the XML data type.
- The XQuery statement uses the Query, Value, and Exist functions to query XML data in a table.
- You can modify XML data by using the modify function provided by the XML data type.
- Using the modify function, you can insert, update, or remove a node from XML data.

Reference Reading

Manipulating Data by Using DML Statements

Reference Reading: Books	***Reference Reading: URLs***
Microsoft SQL Server 2012 Bible by Adam Jorgensen, Jorge Segarra, and Patrick LeBlanc	*http://msdn.microsoft.com/en-us/library/ff848766.aspx*

Manipulating XML Data

Reference Reading: Books	***Reference Reading: URLs***
SQL Server 2012 T-SQL Recipes: A Problem-Solution Approach by Jason Brimhall, David Dye, and Andy Roberts	*http://www.pearsonitcertification.com/articles/article.aspx?p=101369&#38;seqNum=5*

Chapter 6

Implementing Indexes, Views, and Full-Text Search

A database developer is often required to improve the performance of queries. SQL Server allows you to reduce the execution time of queries by implementing indexes. In addition, you can restrict the view of data to different users by implementing *views*.

SQL Server also provides an in-built full-text search capability to allow fast searching of data.

This chapter discusses how to create and manage indexes and views. In addition, it discusses about implementing the full-text search capability.

Objectives

In this chapter, you will learn to:

- Create and manage indexes
- Create and manage views
- Implement a full-text search

Creating and Managing Indexes

When a user queries data from a table based on conditions, the server scans all the data stored in the database table. With an increasing volume of data, the execution time for queries also increases. As a database developer, you need to ensure that the users are able to access data in the least possible time. SQL Server allows you to create *indexes* on tables to enable quick access to data. In addition, SQL Server allows you to create *XML indexes* for columns that store XML data.

At times, the table that you need to search contains huge amount of data. In such cases, it is advisable to create partitioned indexes. A partitioned index makes the index more manageable and scaleable as they store data of a particular partition only.

As a database developer, you need to create and manage indexes. Before creating an index, it is important to identify different types of indexes.

Sometimes, you need to view and control the execution plan of a query. SQL Server provides the facility to view the execution plan of a query. In addition, you can control the execution plan of an query by using various types of hints.

Identifying the Types of Indexes

Before identifying the types of indexes, it is important to understand the need to implement an index.

The data in the database tables is stored in the form of data pages. Each data page is 8 KB in size. Therefore, the entire data of the table is stored in multiple data pages. When a user queries a data value from the table, the query processor searches for the data value in all the data pages. When it finds the value, it returns the result set. As the data in the table increases, this process of querying data takes time.

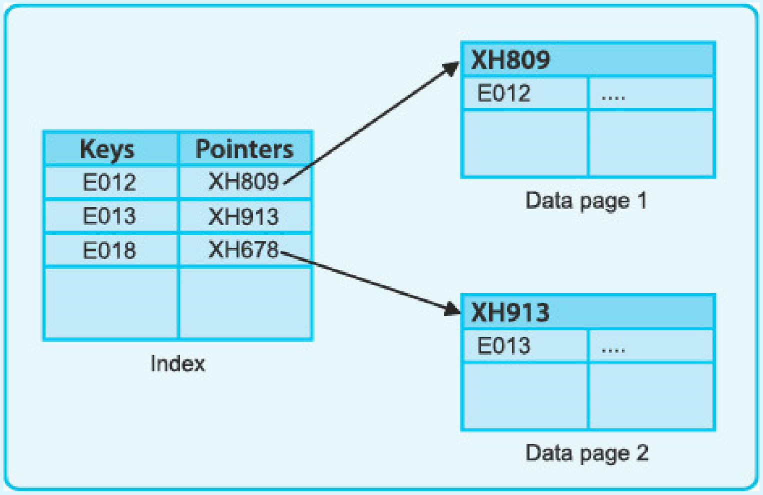
To reduce the data query time, SQL Server allows you to implement indexes on tables. An index is a data structure associated with a table that enables fast searching of data. Indexes in SQL Server are like the indexes at the back of a book that you can use to locate text in the book.

Indexes provide the following benefits:

- Accelerate queries that join tables, and perform sorting and grouping
- Enforce uniqueness of rows, (if configured for that)

An index contains a collection of keys and pointers. Keys are the values built from one or more columns in the table. The column on which the key is built is the one on which the data is frequently searched. Pointers store the address of the location where a data page is stored in the memory.

The following figure depicts the structure of an index.



The Structure of an Index

When the users query data with conditions based on the key columns, the query processor scans the indexes, retrieves the address of the data page where the required data is stored, and accesses the information. The query processor does not need to search for data in all the data pages. Therefore, the query execution time is reduced.

The keys in the indexes are stored in a B-Tree structure in the memory. A B-Tree is a data-indexing method that organizes the index into a multilevel set of nodes. Each page in an index B-Tree is called an index node. Each index contains a single root page at the top of the tree. This root page, or root node, branches out into n number of pages at each intermediate level until it reaches the bottom, or leaf level of the index. The index tree is traversed by following pointers from the upper-level pages down through the lower-level pages.

The key values in the root page and the intermediate pages are sorted in the ascending order. Therefore, in the B-Tree structure, the set of nodes on which the server will search for data values is reduced. This enables SQL Server to find the records associated with the key values quickly and efficiently. When you modify the data of an indexed column, the associated indexes are updated automatically.

SQL Server allows you to create the following types of indexes:

- ❑ Clustered index
- ❑ Nonclustered index

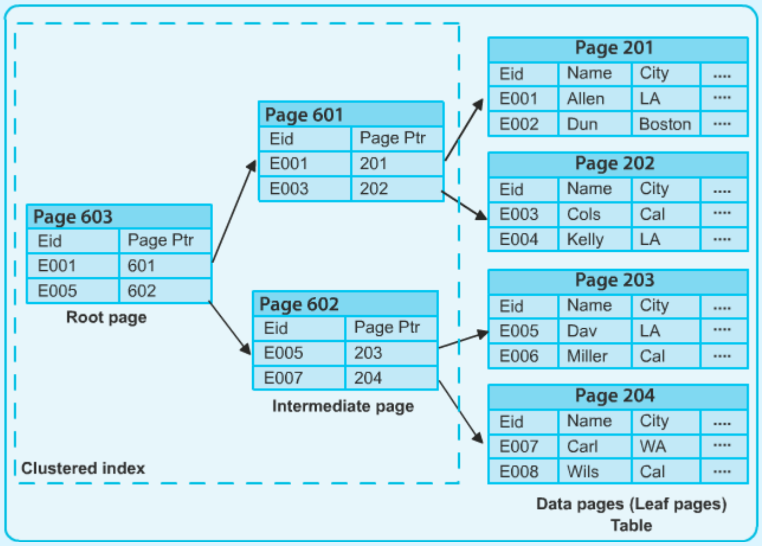
Clustered Index

A *clustered index* is an index that sorts and stores the data rows in the table based on their key values. Therefore, the data is physically sorted in the table when a clustered index is defined on it. Only one clustered index can be created per table. Therefore, you should build the clustered index on attributes that have a high percentage of unique values and are not modified often.

In a clustered index, data is stored at the leaf level of the B-Tree. SQL Server performs the following steps when it uses a clustered index to search for a value:

1. SQL Server obtains the address of the root page from the sysindexes table, which is a system table containing the details of all the indexes in the database.
2. The search value is compared with the key values on the root page.
3. The page with the highest key value less than or equal to the search value is found.
4. The page pointer is followed to the next lower level in the index.
5. Steps 3 and 4 are repeated until the data page is reached.
6. The rows of data are searched on the data page until the search value is found. If the search value is not found on the data page, no rows are returned by the query.

For example, the rows of the Employee table are sorted according to the Eid attribute. The following figure shows the working of a clustered index for the Employee table.



The Working of a Clustered Index

The preceding figure displays a clustered index on the Employee table. To search for any record, SQL Server would start at the root page. It would then move down the B-Tree and the data values would be found on the leaf pages of the B-Tree. For example, if the row containing Eid E006 was to be searched by using a clustered index (refer to the preceding figure), SQL Server performs the following steps:

1. SQL Server starts from page 603, the root page.
2. SQL Server searches for the highest key value on the page, which is less than or equal to the search value. The result of this search is the page containing the pointer to Eid, E005.
3. The search continues from page 602. There, Eid E005 is found and the search continues to page 203.
4. Page 203 is searched to find the required row.

NOTE *A clustered index determines the order in which the rows are actually stored. Therefore, you can define only one clustered index on a table.*

Nonclustered Index

Similar to the clustered index, a *nonclustered index* also contains the index key values and the row locators that point to the storage location of the data in a table. However, in a nonclustered index, the physical order of the rows is not the same as the index order.

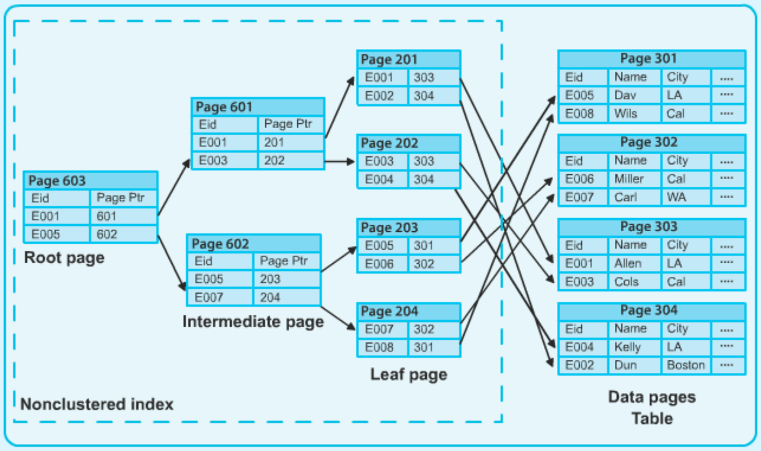
Nonclustered indexes are typically created on columns used in joins and the WHERE clause. These indexes can also be created on columns where the values are modified frequently. SQL Server creates nonclustered indexes by default when the CREATE INDEX command is given. There can be as many as 999 nonclustered indexes per table.

The data in a nonclustered index is present in a random order, but the logical ordering is specified by the index. The data rows may be randomly spread throughout the table. The nonclustered index tree contains the index keys in a sorted order, with the leaf level of the index containing a pointer to the data page.

SQL Server performs the following steps when it uses a nonclustered index to search for a value:

1. SQL Server obtains the address of the root page from the sysindexes table.
2. The search value is compared with the key values on the root page.
3. The page with the highest key value less than or equal to the search value is found.
4. The page pointer is followed to the next lower level in the index.
5. Steps 3 and 4 are repeated until the data page is reached.
6. The rows are searched on the leaf page for the specified value. If a match is not found, the table contains no matching rows. If a match is found, the pointer is followed to the data page and the requested row is retrieved.

The following figure represents the working of a nonclustered index.



The Working of a Nonclustered Index

The preceding figure displays a nonclustered index present on the Eid attribute of the Employee table. To search for any value, SQL Server would start from the root page and move down the B-Tree until it reaches a leaf page that contains a pointer to the required record. It would then use this pointer to access the record in the table. For example, to search for the record containing Eid, E006 by using the nonclustered index, SQL Server performs the following steps:

1. SQL Server starts from page 603, which is the root page.
2. It searches for the highest key value less than or equal to the search value, that is, the page containing the pointer to Eid, E005.
3. The search continues from page 602.
4. Eid, E005 is found and the search continues to page 203.
5. Page 203 is searched to find a pointer to the actual row. Page 203 is the last page, or the leaf page of the index.
6. The search then moves to page 302 of the table to find the actual row.

NOTE *In an index, more than one row can contain duplicate values. However, if you configure an index to contain unique values, the index will also contain unique values. Such an index is called a unique index. You can create a unique index on columns that contain unique values, such as the primary key columns.*

A unique index can be clustered or nonclustered depending on the nature of the column on which it is built.

Creating Indexes

You should create indexes on the most frequently queried columns in a table. However, at times, you might need to create an index based on the combination of one or more columns. An index based on one or more columns is called a *composite index*. A composite index can be based on a maximum of 16 columns. The maximum allowable size of the combined index values is 900 bytes. Columns that are of the Large Object (LOB) data types, such as ntext, text, varchar(max), nvarchar(max), varbinary(max), xml, or image cannot be specified as key columns for an index. However, indexes with less number of columns use less disk space and involve fewer resources when compared to indexes based on more columns.

To create an index, you can use the CREATE INDEX statement. The syntax of the CREATE INDEX statement is:

```
CREATE       [UNIQUE][CLUSTERED          |
NONCLUSTERED] INDEX index_name
ON   [{database_name.[schema_name].    |
schema_name.}]
{table_or_view_name}(column    [ASC    |
DESC][,...n])
[INCLUDE (column_name [,...n])]
[WITH(<relational_index_option>
[,...n])]
[ON              {partition_scheme_name
(column_name)
| filegroup_name | DEFAULT}]
<relation_index_option>::=
{PAD_INDEX = {ON | OFF}
| FILLFACTOR = fillfactor
| SORT_IN_TEMPDB = {ON | OFF}
| IGNORE_DUP_KEY = {ON | OFF}
| STATISTICS_NO_RECOMPUTE = {ON | OFF}
| DROP_EXISTING = {ON | OFF}
| ONLINE = {ON | OFF}
```

where,

UNIQUE creates an index where each row should contain a different index value.

CLUSTERED specifies a clustered index where data is sorted on the index attribute.

NONCLUSTERED specifies a nonclustered index that organizes data logically. The data is not sorted physically.

index_name specifies the name of the index.

table_or_view_name specifies the name of the table or view that contains the attributes on which the index is to be created.

column specifies the column or columns on which the index will be created.

INCLUDE (column_name [,...n])] specifies the columns as non-key members of a - index to covermore queries. It improves the query performance as all the columns in the query are included in the index either as key or non-key columns.

ON partition_scheme_name (column_name) specifies the partition scheme along with the column name. The partition scheme defines the filegroups onto which the partitions of the partitioned index will be mapped. The column name specifies the column against which a partitioned index will be partitioned.

ON filegroup_name specifies the filegroup on which index is created.

ON DEFAULT specifies that the specified index will be created on the default filegroup.

PAD_INDEX = { ON | OFF } specifies the index padding, which is OFF, by default.

FILLFACTOR = fillfactor specifies a percentage that indicates how full the leaf level of each index page should become during index creation or rebuild. The default value is 0.

SORT_IN_TEMPDB = { ON | OFF } specifies about storing temporary sort results in tempdb.

IGNORE_DUP_KEY = { ON | OFF } specifies whether a duplicate key value can be inserted.

STATISTICS_NO_RECOMPUTE = { ON | OFF } specifies about recomputing the distribution statistics.

DROP_EXISTING = { ON | OFF } specifies that the preexisting clustered, nonclustered, or XML index is dropped and rebuilt.

ONLINE = { ON | OFF } checks whether the underlying tables and associated indexes are available to query and modify the data during the index operation.

> NOTE *You can create online indexes only in the SQL Server Enterprise Edition.*

Consider an example of an organization that maintains employee details in the Employee table. You can create a clustered index on the EmployeeID attribute of the Employee table by using the following code:

```
CREATE CLUSTERED INDEX IX_EmployeeID
ON Employee (EmployeeID)
WITH FILLFACTOR = 10
```

In the preceding code, the FILLFACTOR value of 10 has been specified to reserve a percentage of free space on each data page of the index to accommodate future expansion.

The following example creates a nonclustered index on the ManagerID attribute of the Employee table:

```
CREATE       NONCLUSTERED       INDEX
IDX_Employee_ManagerID
ON Employee (ManagerID)
```

> NOTE *When a PRIMARY KEY or UNIQUE constraint is created on a table, an index is created automatically with the same name as the constraint.*

Consider another example, where users frequently query the Document table that contains the following three columns:

- Title nvarchar(50)
- FileName nvarchar(400)
- FileExtension nvarchar(8)

To enhance the performance of such queries, you need to create a nonclustered index that includes all the three columns. However, while creating an index, you need to consider that the size of the index key columns should not exceed 900 bytes. In the preceding example, size of all the three columns would be 916 bytes (458 * 2) as both the nchar and nvarchar data types require two bytes for each character. To avoid exceeding this limitation, you can create an index that contains the Title and FileExtension columns, and include FileName as a non-key column. In this way, the index size would be 114 bytes (58 * 2). You can use the following statement to create such an index:

```
CREATE INDEX IX_Doc_Title
ON     Production.Document     (Title,
FileExtension)
INCLUDE (FileName);
```

The preceding statement creates a nonclustered index on the Document table. This index contains two key columns, Title and FileExtension, and one non-key column, FileName. Therefore, this index covers all the required columns with index key size still under the limit. This way, you can create nonclustered indexes that cover all the columns in the query. This enhances the performance of the query as all the columns included in the query are covered within the index either as key or non-key columns.

Guidelines for Creating Indexes

You need to consider the following guidelines while creating indexes on a table:

- Create clustered indexes on columns that have unique or not null values.
- Do not create an index that is not used frequently. You require time and resources to maintain indexes.

- Create a clustered index before creating a nonclustered index. A clustered index changes the order of rows. A nonclustered index would need to be rebuilt if it is built before a clustered index.
- Create nonclustered indexes on all columns that are frequently used in predicates and join conditions in queries.

Creating Filtered Indexes

Consider a scenario where you have created an index on a particular column of a table to retrieve data. The data in the table is huge. Therefore, the indexing process took a long time to search for an item in the table. SQL Server allows you to solve this problem by creating filtered indexes. A filtered index allows you to create an index on the specific rows of a column rather than the entire column. Therefore, it creates a filter to index a subset of rows within a table. This helps in reducing the index storage space and maintainance costs as compared to full-table indexes.

You can create a filtered index based on a condition specified by the WHERE clause. Filtered indexes do not allow the IGNORE_DUP_KEY option. Only nonclustered filtered indexes can be created on a table. For example, you need to create a filtered index for those records in the Employee table, where the value in the Title column is Tool Manager. Execute the following statement to create the specified index:

```
CREATE NONCLUSTERED INDEX
FX_EmployeeID
ON HumanResources.Employee(EmployeeID)
WHERE Title= 'Tool Manager'
```

In the preceding statement, the FX_EmployeeID index is created on the EmployeeID column of the HumanResources.Employee database based on a value in the Title column.

Creating XML Indexes

When a query is based on an XML column, the query processor needs to parse the XML data each time the query is executed. In SQL Server, an XML data value can be of a maximum of 2 GB. Therefore, the XML values can be very large and the server might take time to generate the result set. To speed up the execution of the query based on the XML data type, SQL Server allows you to create an index that is based on columns storing XML data values. Such indexes are called *XML indexes*.

XML indexes are of the following types:

- Primary XML index
- Secondary XML index

Primary XML Index

This is a clustered B-Tree representation of the nodes in the XML data. When an index is created on a column with the XML data type, an entry will be created for all the nodes in the XML data. Therefore, the index creates several rows of data for each XML value in the column. To create a primary XML index, the table must have a clustered index on the primary key column.

You can create XML indexes on XML columns by using the CREATE PRIMARY XML INDEX and CREATE XML INDEX T-SQL commands. For example, the ProductModel table contains the CatalogDescription column that stores XML values. You can create a primary XML index on this column by using the following statement:

```
CREATE PRIMARY XML INDEX
PXML_ProductModel_CatalogDescription
ON Production.ProductModel
(CatalogDescription)
```

The preceding statement will create an index for all the nodes in the XML data stored in the CatalogDescription column.

NOTE *The first index on an XML type column must be the primary XML index.*

Secondary XML Index

This is a nonclustered index of the primary XML index. A primary XML index must exist before any secondary index can be created on a table. After you have created the primary XML index, an additional three kinds of secondary XML indexes can be defined on the table. The secondary XML indexes assist in the XQuery processing.

The three types of secondary XML indexes are:

- Path indexes
- Value indexes
- Property indexes

Path Indexes

The path index is built on the path id and value columns of the primary XML indexes. This index improves the performance of queries that use paths and values to select data.

For example, if you execute a query that checks for the existence of a product model ID by using an XQuery expression as /PD:ProductDescription/@ProductModelID [.="19"], you can create a path secondary index on the CatalogDescription column of the ProductModel table. In this path index, you can use the primary index created previously.

The following statement creates a Path index on the CatalogDescription column:

```
CREATE XML INDEX
PIdx_ProductModel_CatalogDescription_P
ATH ON Production.ProductModel
(CatalogDescription)
USING XML INDEX
PXML_ProductModel_CatalogDescription
FOR PATH
```

The preceding statement creates a path index, PIdx_ProductModel_CatalogDescription_PATH, on the CatalogDescription column of the table.

Value Indexes

The value indexes contain the same items as path indexes but in the reverse order. It contains the value of the column first and then the path id. This index improves the performance of queries that use paths to select data.

For example, if you execute a query that checks the existence of a node in an XQuery expression such as // Item[@ProductID="1"], you can create a value secondary index by using the primary index created previously.

The following statement creates a value index on the CatalogDescription column:

```
CREATE XML INDEX
PIdx_ProductModel_CatalogDescription_V
ALUE ON Production.ProductModel
(CatalogDescription)
USING XML INDEX
PXML_ProductModel_CatalogDescription
FOR VALUE
```

The preceding statement creates a value index, PIdx_ProductModel_CatalogDescription_VALUE, on the CatalogDescription column of the table.

Property Indexes

The property index contains the primary key of the base table, path id, and the clause columns of primary XML indexes. This index improves the performance of queries that use paths to select data.

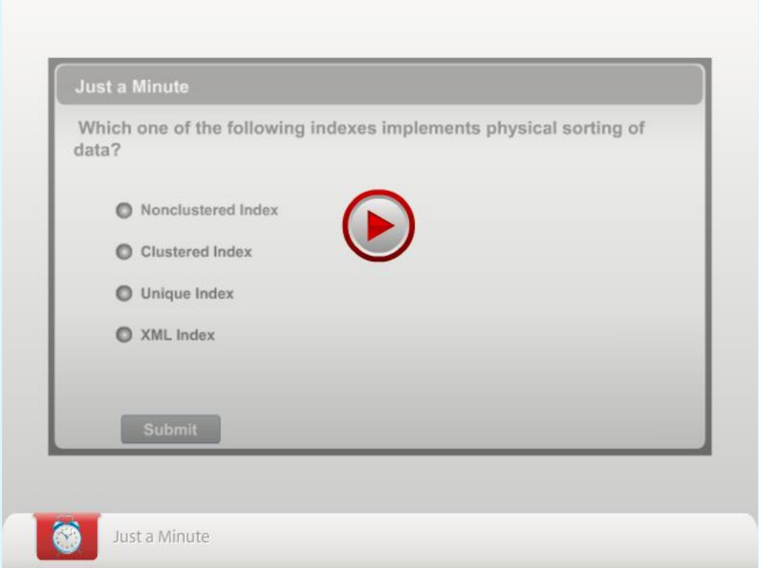
For example, if you execute a query that returns a value of the node in an XQuery expression, such as /ItemList/Item/ @ProductID)[1], you can create a property secondary index on the CatalogDescription column of the ProductModel table by using the following statement:

```
CREATE XML INDEX
PIdx_ProductModel_CatalogDescription_P
ROPERTY ON Production.ProductModel
(CatalogDescription)
USING XML INDEX
PXML_ProductModel_CatalogDescription
FOR PROPERTY
```

The preceding statement creates a property index, PIdx_ProductModel_CatalogDescription_PROPERTY, on the CatalogDescription column of the table.

You need to consider the following guidelines while creating an XML index:

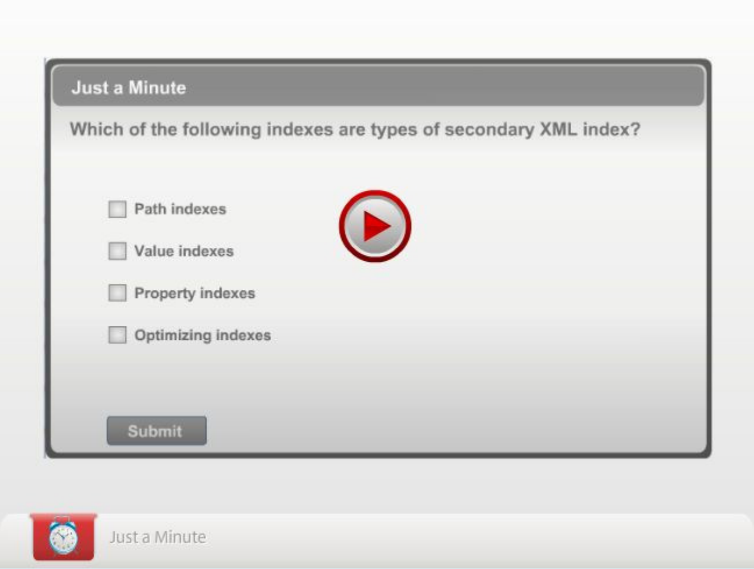
- ❑ XML indexes can be created only on XML columns.
- ❑ XML indexes only support indexing a single XML column.
- ❑ XML indexes can only be added to tables, views, table-valued variables with XML columns or XML variables.
- ❑ XML indexes created on a table do not allow you to modify the primary key. To do so, you first need to drop all the XML indexes on the table.



Just a Minute

Which one of the following indexes implements physical sorting of data?

○ Nonclustered Index
○ Clustered Index
○ Unique Index
○ XML Index



Just a Minute

Which of the following indexes are types of secondary XML index?

□ Path indexes
□ Value indexes
□ Property indexes
□ Optimizing indexes

Creating Partitioned Indexes

In SQL Server, indexes can also be partitioned based on the value ranges. Similar to the partitioned tables, the partitioned indexes also improve query performance. Partitioning enables you to manage and access subsets of data quickly and efficiently. When indexes become very large, you can partition the data into smaller, more manageable sections as the data is spread across filegroups in a database.

Consider an example. In the AdventureWorks database, the SalesOrderHeader table contains the details about the orders received by AdventureWorks, Inc. As the data in this table is large, the query takes a long time to execute. To solve this problem, you can create a partitioned index on the table. Partitioning an index will distribute the data in the table into multiple filegroups, thereby partitioning the table. This will enable the database engine to read or write data quickly. This also helps in maintaining the data efficiently.

To create a partitioned index, you need to perform the following tasks:

1. Create a partition function.
2. Create a partition scheme.
3. Create a clustered index.

Creating a Partition Function

Similar to creating a partitioned table, you need to create a partition function to create a partitioned index. The partition function will determine the boundary values for creating partitions.

Consider an example. The queries on the SalesOrderHeader table are mostly based on the OrderDate column. The sales manager of AdventureWorks requires the details of the orders received, on a yearly basis. The table contains the details of orders for the last five years beginning from 2001. Based on this information, you can create a partition function:

```
CREATE PARTITION FUNCTION PFOrderDate
(datetime)
AS RANGE RIGHT FOR VALUES
('2002-01-01', '2003-01-01',
'2004-01-01', '2005-01-01')
```

The preceding statement creates a partition function, PFOrderDate, by using the datetime data type. It specifies four boundary values. Therefore, there will be five partitions. As range right is specified for partitioning, the first partition will contain data less than the first boundary value, 2002-01-01. The second partition will contain data greater than or equal to 2002-01-01 and less than or equal to 2003-01-01. Similarly, other partitions will store data.

Creating a Partition Scheme

After creating the partition function, you need to create a partition scheme to associate it with the partition function. Based on the boundary values defined in the partition function, there will be five partitions. The data of each partition is stored in a filegroup. You should have the same number of filegroups as partitions. If there are five partitions, you need to create five filegroups: fg1, fg2, fg3, fg4, and fg5. Then, create at least one file in each of these filegroups.

The following statement create the PSOrderDate partition scheme, associating it with the PFOrderDate partition function:

```
CREATE PARTITION SCHEME PSOrderDate
AS PARTITION PFOrderDate
TO (fg1, fg2, fg3, fg4, fg5)
```

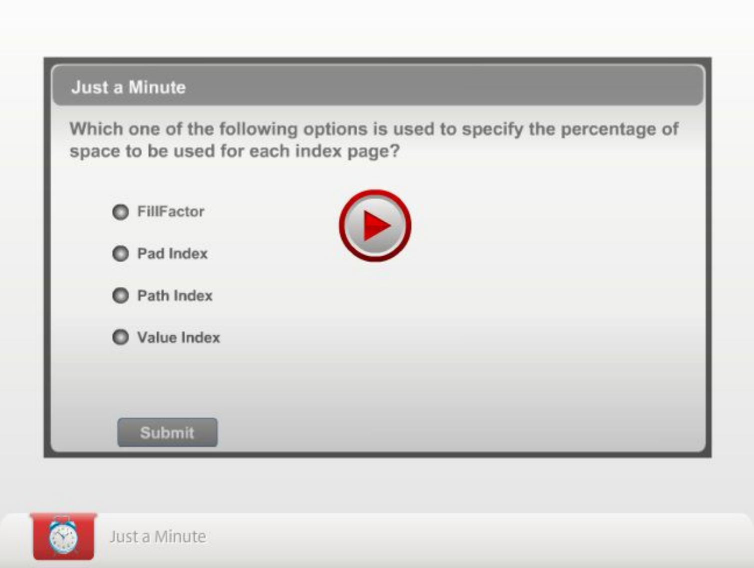
The partition scheme, PSOrderDate, created in the preceding statement directs each partition to a separate filegroup.

Creating a Clustered Index

After creating the partition scheme, you need to associate it with a clustered index. As the clustered index is created on the attribute having unique and non-null values, you can create the index on the SalesOrderID column of the SalesOrderHeader table. To create the partitioned index, you need to associate the clustered index with the partition scheme:

```
CREATE CLUSTERED INDEX ix_SalesOrderID
ON              Sales.MySalesOrderHeader
(SalesOrderID)
ON PSOrderDate (OrderDate)
```

The preceding statement will distribute the table data into five filegroups based on the yearly data of orders stored in the OrderDate column.


Just a Minute

Which one of the following options is used to specify the percentage of space to be used for each index page?

○ FillFactor

○ Pad Index

○ Path Index

○ Value Index

Managing Indexes

In addition to creating indexes, you also need to maintain them to ensure their continued optimal performance. The common index maintenance tasks include disabling, enabling, renaming, and dropping an index. As a database developer, you need to regularly monitor the performance of the index and optimize it.

Disabling Indexes

When an index is disabled, the user is not able to access the index. If a clustered index is disabled then the table data is not accessible to the user. However, the data still remains in the table, but is unavailable for Data Modification Language (DML) operations until the index is dropped or rebuilt.

To rebuild and enable a disabled index, use the ALTER INDEX REBUILD statement or the CREATE INDEX WITH DROP_EXISTING statement.

The following query disables a nonclustered index, IX_EmployeeID, on the Employee table:

```
ALTER INDEX IX_EmployeeID
ON Employee DISABLE
```

Enabling Indexes

After an index is disabled, it remains in the disabled state until it is rebuilt or dropped. You can enable a disabled index by rebuilding it through one of the following methods:

- ❑ Using the ALTER INDEX statement with the REBUILD clause
- ❑ Using the CREATE INDEX statement with the DROP_EXISTING clause

By using one of the preceding statements, the index is rebuilt and the index status is set to enable. You can rebuild a disabled clustered index, when the ONLINE option is set to ON. The DROP_EXISTING clause is used to rebuild the index after dropping the existing clustered and nonclustered indexes. The new index must have the same name as that of an existing index. However, you can modify the index definition by specifying different columns, sorting order, or partitioning scheme while creating a new index.The default value of the DROP_EXISTING index is set to OFF.

Renaming Indexes

You can rename the current index with the help of the sp_rename system stored procedure.

The following statement renames the IX_JobCandidate_EmployeeID index on the JobCandidate table to IX_EmployeeID:

```
EXEC                              sp_rename
'HumanResources.JobCandidate.IX_JobCan
didate_EmployeeID',
'IX_EmployeeID','index'
```

Dropping Indexes

When you no longer need an index, you can remove it from a database. You cannot drop an index used by either a PRIMARY KEY or UNIQUE constraint, except by dropping the constraint.

The following statement drops the IDX_Employee_ManagerID index on the Employee table:

```
DROP INDEX IDX_Employee_ManagerID
ON Employee
```

Optimizing Indexes

SQL Server automatically maintains indexes whenever insert, update, or delete operations are made to the underlying data. These modifications cause the information within the index to become scattered in the database. Fragmentation exists when indexes have pages where the logical ordering does not match the physical ordering within the data file. Heavily fragmented indexes can degrade the query performance and cause your application to respond slowly.

Fragmentation normally occurs when a large number of insert and update operations are performed on the table. Fragmented data can cause SQL Server to perform unnecessary data reads. This affects the performance of the query. Therefore, index defragmentation is necessary to speed up the query performance.

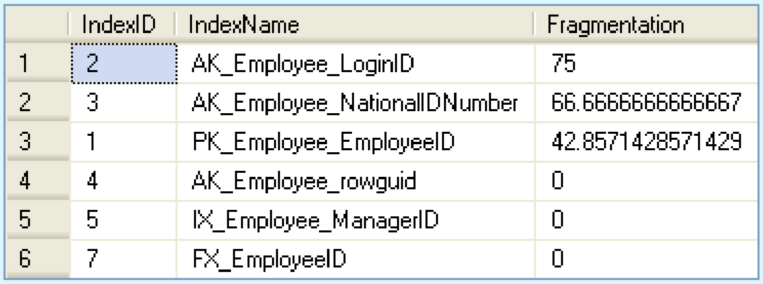
The first step in deciding which defragmentation method to use is to determine the degree of index fragmentation. You can detect index fragmentation by using the sys.dm_db_index_physical_stats system function.

In SQL Server, index defragmentation can be done by either reorganizing or rebuilding an index.

The following statement displays a list of all the indexes on the HumanResources.Employee table with their fragmentation level:

```
SELECT a.index_id AS IndexID, name AS
IndexName,
avg_fragmentation_in_percent        AS
Fragmentation
FROM     sys.dm_db_index_physical_stats
(DB_ID (N'AdventureWorks'),
OBJECT_ID  ('HumanResources.Employee'),
NULL, NULL, NULL) AS a
JOIN sys.indexes AS b ON a.object_id =
b.object_id     AND     a.index_id     =
b.index_id ORDER BY Fragmentation desc
```

The following figure displays the output of the preceding statement.



	IndexID	IndexName	Fragmentation
1	2	AK_Employee_LoginID	75
2	3	AK_Employee_NationalIDNumber	66.6666666666667
3	1	PK_Employee_EmployeeID	42.8571428571429
4	4	AK_Employee_rowguid	0
5	5	IX_Employee_ManagerID	0
6	7	FX_EmployeeID	0

The Output Derived by Using Fragmentation

The fragmentation shown in the preceding figure may differ based on the DML operations performed on the HumanResources.Employee table in the AdventureWorks *database of your SQL Server instance.*

In the preceding output, you can notice that the AK_Employee_LoginID index shows a high level of fragmentation.

After the degree of fragmentation is known, the fragmentation needs to be corrected. The following table lists the actions to be taken at different fragmentation levels to defragment the index.

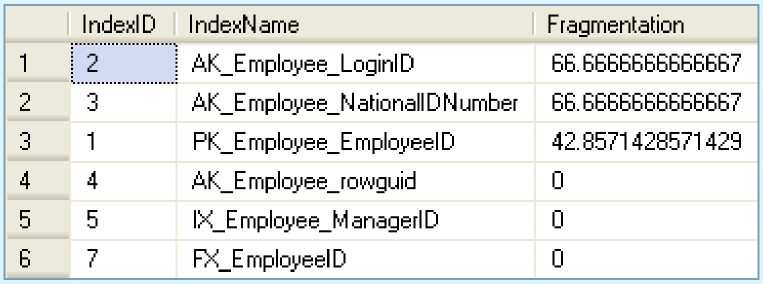
Fragmentation Level	*Action to be Taken*
> 5% and < = 30%	*ALTER INDEX REORGANIZE*
> 30%	*ALTER INDEX REBUILD WITH (ONLINE = ON)*

Actions to be Taken at Different Fragmentation Levels

You can execute the following statement to defragment the AK_Employee_LoginID index:

```
ALTER INDEX AK_Employee_LoginID ON
HumanResources.Employee
REBUILD
```

After executing the preceding statement, the degree of fragmentation is reduced. The following figure displays the output of the preceding statement.



	IndexID	IndexName	Fragmentation
1	2	AK_Employee_LoginID	66.6666666666667
2	3	AK_Employee_NationalIDNumber	66.6666666666667
3	1	PK_Employee_EmployeeID	42.8571428571429
4	4	AK_Employee_rowguid	0
5	5	IX_Employee_ManagerID	0
6	7	FX_EmployeeID	0

The Output

Displaying Execution Plan

In the real-time environment, the faster excution of a query along with timely display of data is essential. If a query is taking too much time to execute, you need to analyse its performance. SQL Server provides the execution plan to view the details of execution of a SQL query. The execution plan of a query enables you to identify how the SQL Server query optimizer executes that query. In other words, the execution plan provides the troubleshooting method to analyze a slow executing query.

When a query is submitted to the database server, it is parsed to generate a parse tree that represents the logical steps to execute the query. The query optimizer processes the parse tree to generate the best possible execution plan for the query. This execution plan includes various factors, such as the cost of the execution plan and the required CPU time. In addition, it also decides the use of the index for executing a particular query. Once, the optimized execution plan is generated, the query is executed and results are generated.

SQL Server provides the following types of formats to view the execution plan:

- ❑ Graphical plan
- ❑ Text plan
- ❑ XML plan

Graphical Plan

Sometimes, as a database developer, you need to have detailed analysis of execution plans. You also need to have query execution plan with information, such as data retrival method applied by the query optimizer, estimated row size, and estimated I/O cost. To accomplish this task, you need to use the graphical execution plan.The graphical execution plan provides the execution details of a query in the form of a tree structure. Each node in the tree structure is represented as an icon. Each icon specifies the logical and physical operator being used to execute that part of the query or statement.

You can display either the estimated or the actual execution plan. The estimated execution plan is displayed before the execution of the query while the actual execution plan is displayed after the execution of the query.

You can view the estimated execution plan by selecting the query in Query Editor. window and clicking the Display Estimated Execution Plan button on the SQL Editor toolbar.

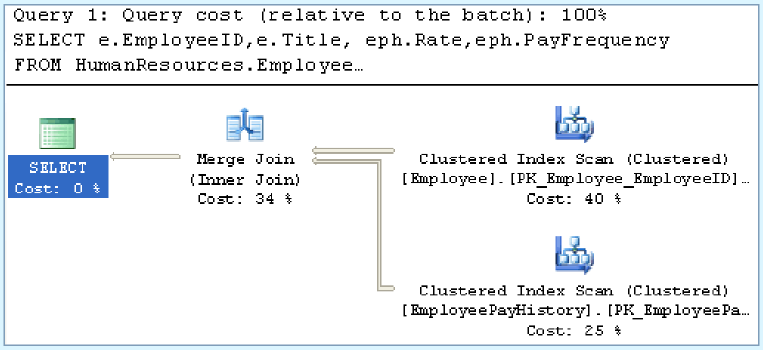
You can view the execution plan by either of the following ways:

- ❑ Clicking the Include Action Execution Plan button on the SQL Editor toolbar
- ❑ Right-clicking the query and selecting the Include Action Execution Plan option
- ❑ Selecting the Include Action Execution Plan option from the query menu

For example, consider the following query:

```
SELECT              e.EmployeeID,e.Title,
eph.Rate,eph.PayFrequency
FROM HumanResources.Employee e
JOIN  HumanResources.EmployeePayHistory
eph
ON e.EmployeeID = eph.EmployeeID
```

The following figure displays the estimated execution plan of the preceding query.



```
Query 1: Query cost (relative to the batch): 100%
SELECT e.EmployeeID,e.Title, eph.Rate,eph.PayFrequency
FROM HumanResources.Employee…
```

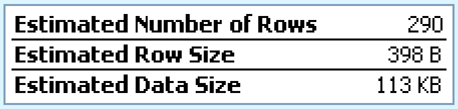
SELECT
Cost: 0 %
Merge Join
(Inner Join)
Cost: 34 %
Clustered Index Scan (Clustered)
[Employee].[PK_Employee_EmployeeID]…
Cost: 40 %
Clustered Index Scan (Clustered)
[EmployeePayHistory].[PK_EmployeePa…
Cost: 25 %

Displaying the Estimated Execution Plan

In the preceding figure, the two clustered index scan icons on the right side represents that the clustered indexes of the HumanResources.Employee and HumanResources.EmployeePayHistory tables are scanned to execute the query. The result set produced after scanning the index is merged by the Merge Join operator. This merged result set is passed to the SELECT icon that is on the left side of the figure. The arrows in the figure show that the data is passed between the operators, as represented by icons, such as SELECT, merge join, and clustered index scan.

A number in the form of a percentage is displayed below each icon. This number represents the relative cost to the query for that operator. This cost represents the estimated execution time for that operation. The graphical execution plan is read from right to left and top to bottom. A pop-up window called ToolTip is associated with each of the icons and the arrow. You can access this TootTip by hovering the mouse pointer over the icon.

If you hover the mouse over the arrow between the SELECT and merge join operators, a ToolTip displaying data, such as estimated number of rows, row size, and data size is displayed, as shown in the following figure.

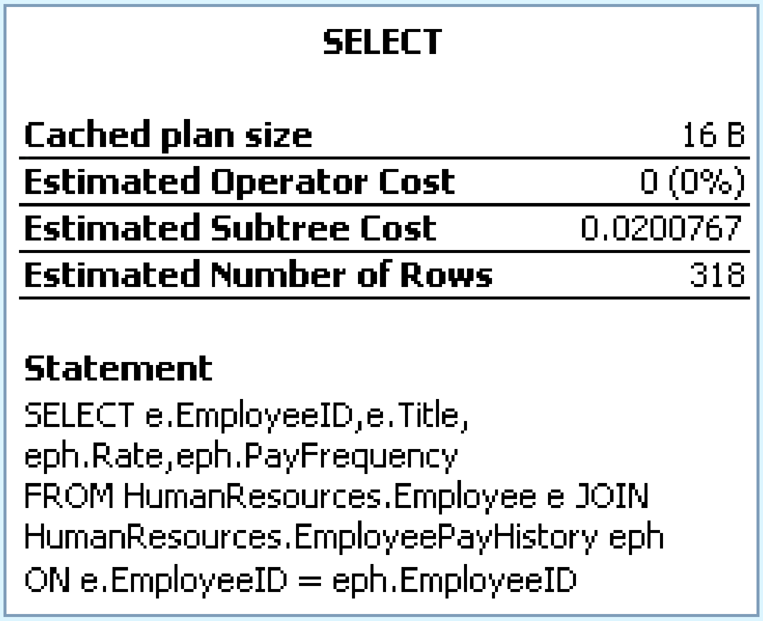


Estimated Number of Rows	290
Estimated Row Size	398 B
Estimated Data Size	113 KB

A ToolTip Displaying Data

NOTE *The details shown in the preceding figure may differ based on the DML operations performed on the HumanResources.Employee and HumanResources.EmployeePayHistory tables in the AdventureWorks database of your SQL Server instance.*

When you hover the mouse over the SELECT icon, the SELECT page is displayed, as shown in the following figure.



SELECT

Cached plan size	16 B
Estimated Operator Cost	0 (0%)
Estimated Subtree Cost	0.0200767
Estimated Number of Rows	318

Statement

SELECT e.EmployeeID,e.Title,
eph.Rate,eph.PayFrequency
FROM HumanResources.Employee e JOIN
HumanResources.EmployeePayHistory eph
ON e.EmployeeID = eph.EmployeeID

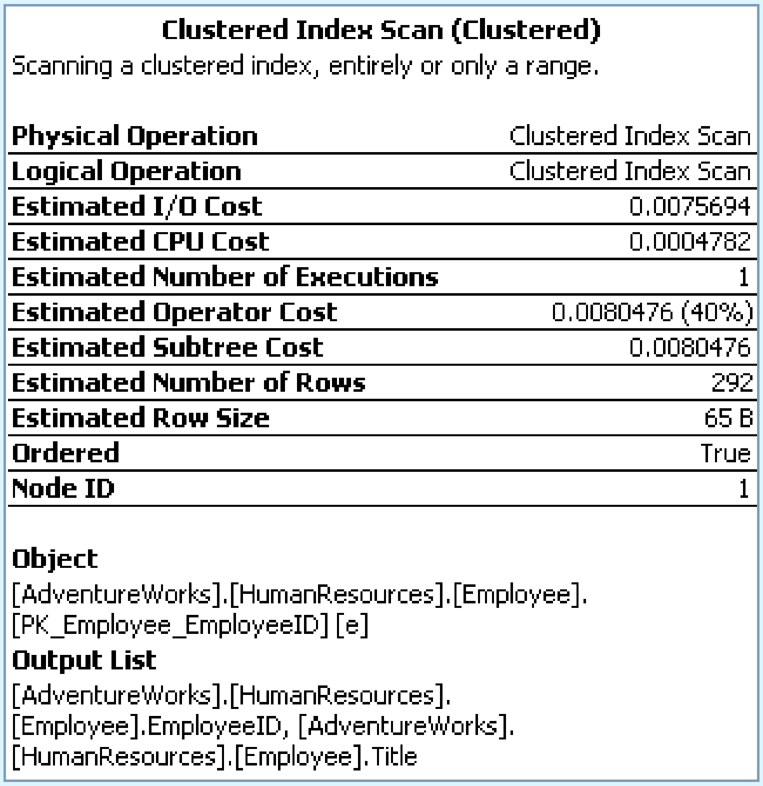
The SELECT Page

In the preceding figure, the cached plan size represents the amount of memory the plan generated by this query will take up in cache. The estimated operator cost represents the estimated cost incurred in executing that operation. Here, it is represented as 0% because the data is just passing to the SELECT operator. Therefore, there is approximately no cost associated with this operator. Though, a 0% cost still amounts to a small cost associated with the operator.

The estimated subtree cost is used by the query optimizer to determine the total cost incurred in executing the current operation as well as the operations preceding it in the query. Here, before the SELECT operation, two clustered index scans and a join operation is performed. Therefore, it is the sum of the estimated operator costs of the merge join and two clustered index scan operators. You can check the costs of each of these operators by hovering the mouse over them. The estimated number of rows represents the number of rows expected to be returned in the result set.

Similarly, you can view the details of the clustered index scan performed on the HumanResources.Employee table by hovering the mouse on the clustered index scan icon.

The details of the clustered index scan are displayed, as shown in the following figure.



Clustered Index Scan (Clustered)

Scanning a clustered index, entirely or only a range.

Physical Operation	Clustered Index Scan
Logical Operation	Clustered Index Scan
Estimated I/O Cost	0.0075694
Estimated CPU Cost	0.0004782
Estimated Number of Executions	1
Estimated Operator Cost	0.0080476 (40%)
Estimated Subtree Cost	0.0080476
Estimated Number of Rows	292
Estimated Row Size	65 B
Ordered	True
Node ID	1

Object

[AdventureWorks].[HumanResources].[Employee].
[PK_Employee_EmployeeID] [e]

Output List

[AdventureWorks].[HumanResources].
[Employee].EmployeeID, [AdventureWorks].
[HumanResources].[Employee].Title

The Details of the Clustered Index Scan

In the preceding figure, the details of the clustered index scan is represented in the form of following options:

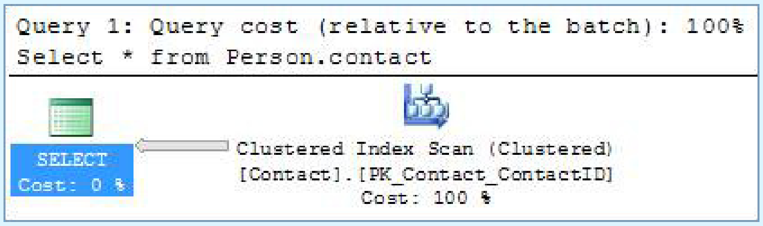
- ❑ **Physical Operation**: Represents the operator used in the query execution.
- ❑ **Logical Operation**: Matches the physical operator included in the execution plan.
- ❑ **Estimated I/O Cost**: Specifies the estimated cost of all I/O activities for the operation. This value should be as low as possible.
- ❑ **Estimated CPU Cost**: Specifies the estimated cost of all CPU activities for the operation.
- ❑ **Estimated Number of Executions**: Specifies the estimated number of times the operator will be executed while running the current query.
- ❑ **Estimated Operator Cost**: Specifies the cost incurred in executing the operations included in the query. The cost of the operation is displayed as a percentage of the total cost of the query inside the parentheses. Since there is only one operation involved in the execution of the query, the cost of the operation is displayed as 100%.
- ❑ **Estimated Subtree Cost**: Represents the cost incurred in executing the specified operation and all the operations preceding it.
- ❑ **Estimated Number of Rows**: Represents the number of rows produced by the operator specified in the query.
- ❑ **Estimated Row Size**: Represents the estimated size of the row produced by the operator in bytes.
- ❑ **Ordered**: Specifies whether the data on which query is executed is ordered.
- ❑ **Node ID**: Represents the number assigned to that node. Nodes in the execution plan are numbered from left to right.

The ToolTip window displays limited information. You can view the detailed information by right-click any icon in the graphical execution plan and selecting the Properties option from the pop-up menu. This will display the detailed list of information about that operation.

Sometimes, index scan is not beneficial in terms of performance when the table contains a large number of records and the result set contains comparatively lesser number of records. For example, consider the following query:

```
SELECT * FROM Person.contact
```

When you execute the preceding query, the execution plan is generated, as shown in the following figure.



```
Query 1: Query cost (relative to the batch): 100%
Select * from Person.contact
```

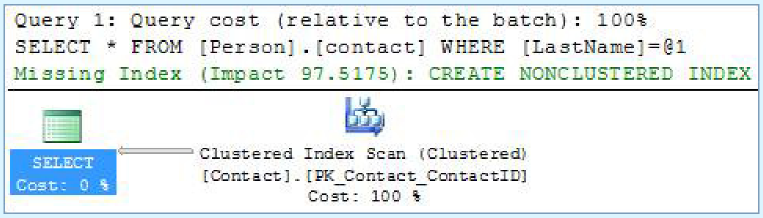
SELECT
Cost: 0 %
Clustered Index Scan (Clustered)
[Contact].[PK_Contact_ContactID]
Cost: 100 %

The Execution Plan Showing Index Scan

In the preceding figure, a complete index scan is performed as all the data from the table is required. Now, execute the following query:

```
SELECT   FirstName,   LastName   FROM
Person.Contact   WHERE   LastName   =
'Russell'
```

The following figure shows the execution plan generated by executing the preceding query.



```
Query 1: Query cost (relative to the batch): 100%
SELECT * FROM [Person].[contact] WHERE [LastName]=@1
Missing Index (Impact 97.5175): CREATE NONCLUSTERED INDEX
```

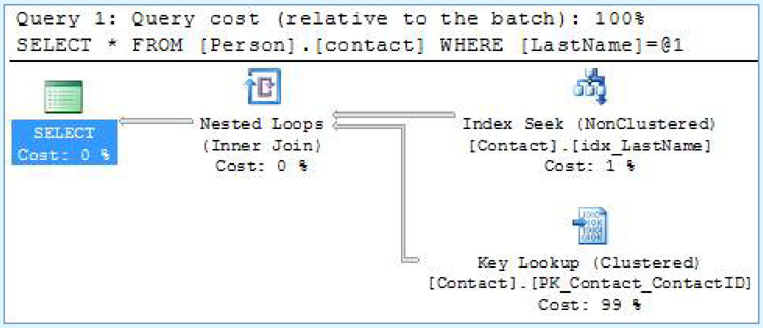
SELECT
Cost: 0 %
Clustered Index Scan (Clustered)
[Contact].[PK_Contact_ContactID]
Cost: 100 %

The Execution Plan Showing Missing Index Details

In the preceding figure, the complete index scan is performed. However, the execution plan informs you that there is a missing index that can enhance the performance of the query. Right-click on the query execution plan, and then select the Missing Index Details options. A new query window opens containing the script to create a nonclustered index on the LastName column of the Person table. Execute that script to create the missing nonclustered index by the name, idx_LastName, as shown in the following statement:

```
CREATE NONCLUSTERED INDEX idx_LastName
ON Person.Contact (LastName)
```

After creating the nonclustered index, execute the preceding query again. This generates the execution plan, as shown in the following figure.



```
Query 1: Query cost (relative to the batch): 100%
SELECT * FROM [Person].[contact] WHERE [LastName]=@1
```

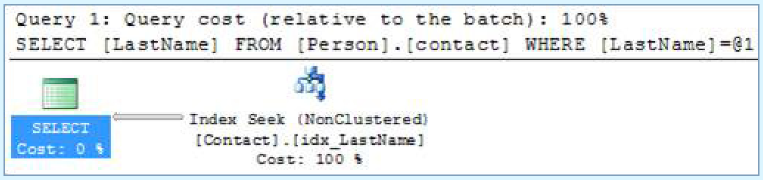
The Execution Plan Showing Key Lookup

In the preceding figure, an index seek operation is performed, instead of index scan. The index seek operation is faster if the query is trying to search for specfic records as specified in the WHERE clause. The execution plan is performing the index seek operation as the database engine finds the nonclustered index created on the LastName column. Therefore, instead of performing index scan by using the clustered index, it directly seeks the records by searching the nonclustered index. The execution plan is also performing key lookup operation on the clustered index. The reason for this key look up operation is that the the nonclustered index only contains the LastName column. However, the SELECT query also requires records from the FirstName column on which there is no index. Therefore, to retrieve the records from the FirstName column, the clustered index is used.

Now, execute the following query:

```
SELECT LastName FROM Person.contact
WHERE LastName = 'Russell'
```

The following figure shows the execution plan of the preceding query.



```
Query 1: Query cost (relative to the batch): 100%
SELECT [LastName] FROM [Person].[contact] WHERE [LastName]=@1
```

The Execution Plan Showing Index Seek

In the preceding figure, only the index seek operation is performed. This is because that query needs to search for the records only in the LastName column and the nonclustered index exists on this column. Therefore, instead of performing index scan, only an index seek operation is performed and the query performance is improved.

The index seek operation affects those rows that satisfy the query condition. Therefore, it searches for those pages that contain the required rows. This is highly beneficial in terms of performance, especially when the table contains a large amount of data. Index seek is faster as compared to index scan because index seek has to search a less number of records as compared to index scan, which performs complete scan of the index data.

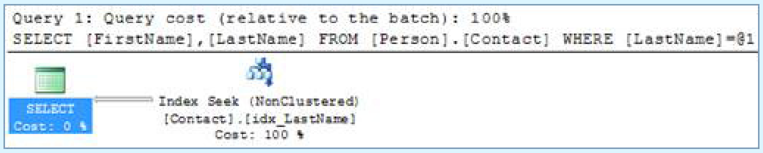
The index seek operation is also beneficial for those tables, which contain indexes with included columns. For example, modify the idx_LastName index, as shown in the following statement:

```
DROP INDEX idx_LastName ON
Person.Contact
GO
CREATE NONCLUSTERED INDEX idx_LastName
ON Person.Contact (LastName)
INCLUDE (FirstName)
GO
```

Now, execute the following query:

```
SELECT FirstName, LastName FROM
Person.Contact WHERE LastName =
'Russell'
```

The execution of the preceding query generates the execution plan, as shown in the following figure.



```
Query 1: Query cost (relative to the batch): 100%
SELECT [FirstName],[LastName] FROM [Person].[Contact] WHERE [LastName]=@1
```

The Execution Plan Showing Index Seek on Included Columns in the Index

The preceding execution plan is only performing index seek operation on the nonclustered index as this index contains all the columns required by the query.

Text Plan

The graphical execution plan displays the execution plan in diffrent tool tip windows for each icon. However, the entire information regarding the execution plan is not visible at once. In this situation, you need to display the query execution plan in a textual format. It represents the query execution plan as a hierarchical tree containing the text of the T-SQL statement. The text execution plan displays all the information regarding the execution plan in the form of a single representation. To display the text plan, you first need to activate the text execution plan by using the following command:

```
SET SHOWPLAN_ALL ON
```

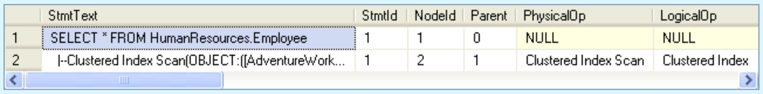
When SHOWPLAN_ALL is set to ON, the execution information is collected for all subsequent T-SQL statements. Howeve, these statements are not actually executed. You need to set SHOWPLAN_ALL as OFF after viewing the relevant information. Otherwise, the CREATE, UPDATE, or DELETE statements will not be executed. To turn-off the SHOWPLAN_ALL option, execute the following command:

```
SET SHOWPLAN_ALL OFF
```

After setting the SHOWPLAN_ALL option as ON, you can display the text execution plan by executing the following SQL query:

```
SELECT * FROM HumanResources.Employee
```

The following figure shows the text execution plan of the preceding query.



	StmtText	StmtId	NodeId	Parent	PhysicalOp	LogicalOp
1	SELECT * FROM HumanResources.Employee	1	1	0	NULL	NULL
2	|--Clustered Index Scan(OBJECT:([AdventureWork...	1	2	1	Clustered Index Scan	Clustered Index

The Text Execution Plan

The preceding figure shows the execution plan in the text format. The first row displays the SQL statement that was submitted to the database server. The subsequent rows represent the operations occurring within the execution plan. The PhysicalOp and LogicalOp columns denote the physical and logical operations performed on the query. In the preceding query, a clustered index scan is performed.

XML Plan

Sometimes, you need to transfer the query execution plans to different computers where SQL Server is not installed. If you use graphical or text execution plans, then it would be required to execute the queries and then only the execution plan may be displayed. To accomplish this task, you need to use the XML execution plan that ensures better portability of query execution plan across multiple computers. Moreover, it also exempts the user from installing SQL Server on the destination machines.

An XML plan displays the execution plan in an XML format. Compared to the graphical and text execution plans, the XML execution plan can be easily moved across the computers. In addition, they can be programmed by using XML technologies, such XPath and Xquery.

You need to activate the XML execution plan by using the following statement:

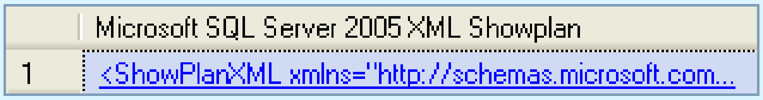
```
SET SHOWPLAN_XML ON
```

NOTE *Before executing the preceding statement, ensure that the Display Estimated Execution Plan and Include Actual Execution Plan buttons are not enabled on the SQL Editor toolbar.*

Now, after setting the SHOWPLAN_XML option as ON, you can display the text execution plan by executing the following query:

```
SELECT * FROM HumanResources.Employee
```

The following figure displays the result of the execution of the preceding query.



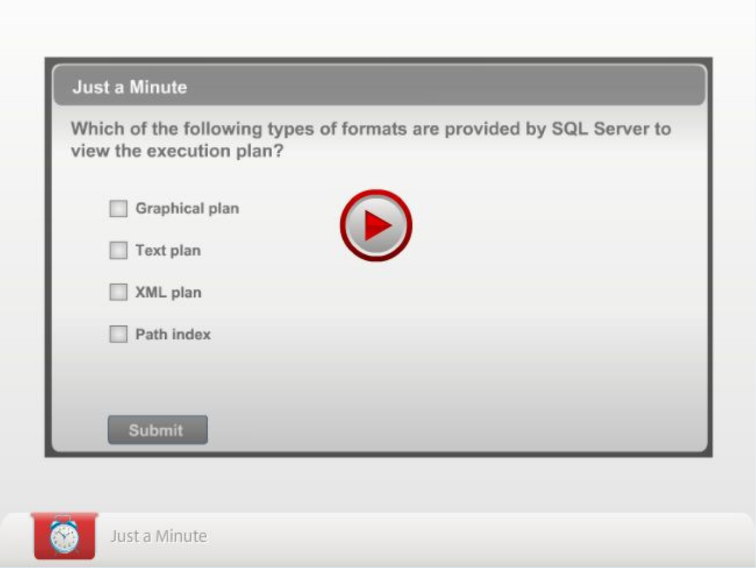
	Microsoft SQL Server 2005 XML Showplan
1	<ShowPlanXML xmlns="http://schemas.microsoft.com...

The Output Derived by Using the Preceding Statement

You can view the XML plan in the XML editor by clicking on the link shown in the preceding figure.

To turn-off the SHOWPLAN_XML option, execute the following statement:

```
SET SHOWPLAN_XML OFF
```



Just a Minute

Which of the following types of formats are provided by SQL Server to view the execution plan?

□ Graphical plan

□ Text plan

□ XML plan

□ Path index

Identifying the Logical Order of Operations in SELECT Statement

In most of the programming languages, the order of execution of the statements is predefined. The statements are executed in the order in which they are written from top to bottom. The order of execution of statements in T-SQL is little different from other programming languages. In T-SQL, it is not necessary that the statements will be executed in the order in which you have written your query. The query that you write is first optimized by the database engine. When you submit your query to the database server, it is parsed to generate a parse tree that represents the logical steps to execute the query. Further, the query optimizer processes the parse tree to generate the best possible execution order or plan for the query. Finally, the query is executed according to the execution plan and then the result is displayed.

A SELECT statement is constructed with one or more clauses, such as FROM and WHERE. These clauses are executed in different phases. Each phase executes a clause and generates a virtual table that is feed as an input to the next phase. The phases and their orders are:

1. **FROM**: If more than one table is listed in the FROM clause, a cartesian product of the first two tables listed in the FROM clause is generated. This result is stored in a virtual table, for example VT1. If there is only one table listed in the FROM clause, all the records are extracted from the table and the result is stored in a virtual table.
2. **ON**: The ON condition is applied on the virtual table, VT1. This generates a new virtual table, VT2, containing the rows which satisfies the join condition.
3. **JOIN**: The rows are retrieved from the virtual table, VT2, according to the join type and stored

in a new virtual table, VT3.

4. If more than two tables are specified in the FROM clause, the step 1 to 3 is repeated between the VT3 and the next table specified in the FROM clause.
5. **WHERE**: This condition is applied on VT3. The rows that satisfy the WHERE condition is stored in a new virtual table, VT4.
6. **GROUP BY**: The rows of the virtual table, VT4, are arranged according to the column listed in the GROUP BY clause and stored in a new virtual table, VT5.
7. **WITH CUBE or WITH ROLLUP**: The ROLLUP and CUBE operators are used to apply multiple levels of aggregation on the result retrieved in the virtual table, VT5, using the GROUP BY clause. The result is stored in a new virtual table, VT6.
8. **HAVING**: This clause retrieves the groups that match the condition. The result is stored in a new virtual table, VT7.
9. **SELECT**: The columns specified in the SELECT list in the query is retrieved from the virtual table, VT7, and stored in a new virtual table, VT8.
10. **DISTINCT**: The duplicate rows from the virtual table, VT8 are removed and a new virtual table, VT9 is generated.
11. **ORDER BY**: The rows of the virtual table, VT9, is arranged in the order specified in the ORDER BY clause and a new virtual table, VT10, is generated.
12. **TOP**: The first set of rows from the top of the virtual table, VT10, is retrieved and stored in a new virtual table, VT11. The rows of this table are returned as a result to the caller. A caller may be a client application or the outer query.

The actual physical execution of the clauses in the SELECT statement may differ from the specified order.

Virtual tables are temporary tables used by SQL Server while executing the query and not available to the user.

The logical order of operation in SELECT statement can be specified using the syntax of SELECT statement as:

```
(8)  SELECT
(9)  DISTINCT
(11)<TOP specification> <select list>
(1)  FROM <left table>
(3)  <join type> JOIN <right table>
(2)  ON <join condition>
(4)  WHERE <search condition>
(5)  GROUP BY <group by list>
(6)  WITH {CUBE | ROLLUP}
(7)  HAVING <search condition>
(10)ORDER BY <order by list>
```

To understand the logical order of operation of SELECT statement, consider the following query:

```
SELECT      C.CustomerID,        COUNT
(O.SalesOrderID) AS Numorders
FROM  AdventureWorks.Sales.Customer  AS
C
LEFT             OUTER             JOIN
AdventureWorks.Sales.SalesOrderHeader
AS O
ON C.CustomerID = O.CustomerID
WHERE C.CustomerType = 'S'
GROUP BY C.CustomerID
HAVING COUNT(O.SalesOrderID) < 3
ORDER BY Numorders;
```

In the preceding query, the cartesian product of the Customer and the SalesOrderHeader tables is generated and stored in a virtual table, VT1. Further, the rows whose C.CustomerID is equal to the O.CustomerID is returned and stored in a new virtual table, VT2. Further, the rows from VT2 and rows from the Customer table for which no match was found in previous step is returned and stored in a new virtual table, VT3. The rows from VT3 whose CustomerType is S is returned and stored in a new virtual table, VT4. The rows of VT4 are arranged in group based on the C.CustomerID column and stored in a new virtual table, VT5. Further, those groups whose number of rows of O.SalesOrderID column is less than three is returned and stored in a new virtual table, VT6. Further, the CustomerID column and the SalesOrderID column as Numorders is retrieved and stored in a new virtual table, VT7. Further, the rows from VT7 are sorted in ascending order based on the Numorders column and stored in a new virtual table, VT8. The virtual table, VT8, is returned as a result to the user.

The Impact of Logical Processing Order on Alias

The alias created with the SELECT statement cannot be used with the expression which is executed before the SELECT statement in the logical order of operation. It means you cannot use an alias created with the SELECT statement with the clauses, such as WHERE or HAVING. However, you can use the aliases created with the SELECT statement with the ORDER BY clause, because the ORDER BY clause is executed after the execution of the SELECT clause. For example, the following query will run without any error:

```
SELECT  OrderQty,  UnitPrice,  OrderQty
AS Order_Quantity
FROM Sales.SalesOrderDetail
ORDER BY Order_Quantity;
```

However, the following query will not execute successfully, because the WHERE clause is executed before the SELECT clause:

```
SELECT  OrderQty,  UnitPrice,  OrderQty
AS Order_Quantity
FROM Sales.SalesOrderDetail
WHERE Order_Quantity>5;
```

When you execute the preceding statement, the SQL Server returns the following error message:

```
Msg 102, Level 15, State 1, Line 3
Incorrect          column          name
'Order_Quantity'.
```

Therefore, you need to repeat the expression in the WHERE and HAVING clause for which you have created alias in the SELECT clause. Consider the following query:

```
SELECT  OrderQty,  UnitPrice,  OrderQty
AS Order_Quantity
FROM Sales.SalesOrderDetail
WHERE OrderQty > 5;
```

In the preceding query, the OrderQty column is written in the WHERE clause instead of writing the Order_Quantity alias created in the SELECT statement.

In addition, the expression aliases cannot even be used by other expressions within the same SELECT list. For example, consider the following query:

```
SELECT  OrderQty,  UnitPrice  as  price,
price * OrderQty AS Total
FROM Sales.SalesOrderDetail;
```

The preceding query when executed generates the following error message:

```
Msg 207, Level 16, State 1, Line 1
Invalid column name 'price'.
```

Therefore, you need to write the names of the columns in the next SELECT list instead of using aliases. Consider the following query:

```
SELECT  OrderQty,  UnitPrice  as  price,
UnitPrice * OrderQty AS Total
FROM Sales.SalesOrderDetail;
```

The FROM Clause and Virtual Table

When a SELECT statement is executed, the FROM clause is the first clause to be processed by the database engine. A FROM clause can have multiple tables including views, derived tables, and joined tables. When a FROM clause is processed, a virtual table is created by the database engine. This virtual table is used by the other clauses, such as WHERE and GROUP BY clause of the SELECT statement. The virtual table produced by the FROM clause can be given aliases and used in the other clauses of the SELECT statement as the table name. However, there is an exception of this concept is selfjoin. For example, consider the following queries:

```
SELECT
Sales.SalesOrderDetail.SalesOrderID,
Sales.SalesOrderDetail.ProductID,
Sales.SalesOrderDetail.OrderQty,
Sales.SalesOrderHeader.OrderDate,
Sales.SalesOrderHeader.DueDate
FROM Sales.SalesOrderDetail
JOIN      Sales.SalesOrderHeader      ON
SalesOrderDetail.SalesOrderID          =
Sales.SalesOrderHeader.SalesOrderID;

SELECT                 SOD.SalesOrderID,
SOD.ProductID, SOD.OrderQty,
SOH.OrderDate, SOH.duedate
FROM Sales.SalesOrderDetail SOD
JOIN  Sales.SalesOrderHeader  AS  SOH  ON
SOD.SalesOrderID = SOH.SalesOrderID;
```

Both the preceding queries returns the same result set. However, there is a big difference in writing both the queries. The first query uses the fully qualified name of the tables, but the second query uses aliases. Using aliases with join is very convenient to create queries. In addition, the table alias improves the readability of queries.

If you uses alias to refer a table in a query, it is the best practice to use that alias everywhere in the query instead of the table name.

Controlling Execution Plan

The query optimizer processes the parse tree to generate the best possible execution plan for the query. Sometimes, the query optimizer does not provide the best execution plan when the performance of a query is not optimum and it consumes more CPU time to execute. In such situations, you need to control the query execution by providing hints to the query optimizer. These hints override any execution plan that the query optimizer might select for a query. They instruct the database engine on how to execute the query. For example, you can use a hint to instruct the database engine to use as little memory as possible, or to use or not use an index in the execution of a query.

SQL Server allows you to provide the following types of hints:

- ❑ **Query hints**: Is used when you need to apply a certain kind of logic to the whole query. For example, you can instruct the database engine to discard the execution plan generated for the query after it executes and recompiles the execution plan the next time the same query is executed.
- ❑ **Join hints**: Is used to instruct the query optimizer to enforce a join strategy between the tables. Consider an example. The student data of Tebisco Ltd. is stored in the Student table and the course details are stored in the Course table.

CourseId	***CourseName***
121	*Java*

140	*.Net*

The Course Table

StudentId	***CourseId***
1	*121*
2	*121*
3	*140*
4	*121*
5	*140*

The Student Table

You want to view the StudentID and the name of the course the student is enrolled for. Therefore, you need to perform an inner join on the Student and the Course tables. You can execute the following query:

```
SELECT StudentId,CourseName
FROM Course INNER JOIN Student ON
Course.CourseId                  =
Student.CourseId
```

Though the preceding statement is absolutely correct and gives you the desired output, the query optimizer can also be instructed to adopt a specific joining strategy to optimize the execution plan and performance.

You can use the following types of join hints:

- ❑ **Loop join**:The loop join compares each row of the table on the left side (first table) of the JOIN operator with each row of the table on the right side (second table). If a match is found, the rows from both the tables are joined. After this the scanning of the second table continues,for the next matching record. Following algorithm explains the working of the loop join:

  ```
  for each row R1 in table1
    for each row R2 in table2
      if R1 joins with R2
        return (R1, R2)
  ```

  To perform an inner loop join you can execute the following query:

  ```
  SELECT StudentId,CourseName
  FROM Course INNER LOOP JOIN
  Student ON Course.CourseId =
  Student.CourseId
  ```

  In case of loop joins, the total cost of join is proportional to the size of the table1 multiplied by the size of the table2.

- ❑ **Merge join**: The merge join compares the rows of the table on the left side of the JOIN operator with each row of the table on the right side. If a match is found, the rows from both the tables are joined. The scanning of the records in the second table continues for the next matching record. This continues until the end of table is reached. Thereafter, the scanning control goes back to the next record in the first table and the the scanning of the second table starts again in search of a matching record For merge join, both the tables must be sorted on the common column.

  Following algorithm explains the working of the merge join:

  ```
  get first row R1 from table 1
  get first row R2 from table 2
  while not at the end of either table
  begin
    if R1 joins with R2
    begin
      return (R1, R2)
      get next row R2 from table 2
    end
    else
      get next row R1 from table 1
  ```

  To perform the merge join, you can execute the following query:

  ```
  SELECT StudentId,CourseName
  FROM Course INNER MERGE JOIN
  Student ON Course.CourseId =
  Student.CourseId
  ```

  In case of merge join, the total cost of join is proportional to the sum of the number of rows in both the tables.

- ❑ **Hash join**: It is used for joining very large data sets, specifically, if one of the tables is small and the other is very large. This is the best case for a hash join to be used and dramatically improves the query performance. The query optimizer uses the smaller table to build a temporary table in the memory called as the hash table. Then the larger table is scanned, searching for a matching record in the hash table. Use of a hash table saves memory space as well as makes the query execution fast.

  To perform the hash join, you can execute the following query:

  ```
  SELECT StudentId,CourseName
  FROM Course INNER HASH JOIN
  Student ON Course.CourseId =
  Student.CourseId
  ```

> NOTE *Nested loop join can not be specified with RIGHT or FULL OUTER join.*

> NOTE *Hash Join is based on hash function . The understanding of the algorithm for the hash join is beyond the scope of this module.*

- ❑ **Table hints**: Are used when you need to control the locking mechanism of tables. When you execute the T-SQL statements, such as SELECT, UPDATE, DELETE, INSERT, or MERGE, SQL Server applies the appropriate lock on tables. However, if you want to overwrite this default behavior of locking, you can use table hints.

Activity 6.1: Creating Index

Creating and Managing Views

At times, the database administrator might want to restrict access of data to different users. They might want some users to be able to access all the columns of a table whereas other users to be able to access only selected columns. SQL Server allows you to create views to restrict user access to the data. Views also help in simplifying query execution when the query involves retrieving data from multiple tables by applying joins.

A *view* is a virtual table, which provides access to a subset of columns from one or more tables. It is a query stored as an object in the database, which does not have its own data. A view can derive its data from one or more tables, called the base tables or underlying tables. Depending on the volume of data, you can create a view with or without an index. As a database developer, it is important for you to learn to create and manage views.

Creating Views

A view is a database object that is used to view data from the tables in the database. A view has a structure similar to a table. It does not contain any data, but derives its data from the underlying tables.

Views ensure security of data by restricting access to:

- ❑ Specific rows of a table.
- ❑ Specific columns of a table.
- ❑ Specific rows and columns of a table.
- ❑ Rows fetched by using joins.
- ❑ Statistical summary of data in a given table.
- ❑ Subsets of another view or a subset of views and tables.

Apart from restricting access, views can also be used to create and save queries based on multiple tables. To view data from multiple tables, you can create a query that includes various joins. If you need to frequently execute this query, you can create a view that executes this query. You can access data from this view every time you need to execute the query.

You can create a view by using the CREATE VIEW statement. The syntax of the CREATE VIEW statement is:

```
CREATE VIEW view_name
[(column_name [, column_name]...)]
[WITH ENCRYPTION [, SCHEMABINDING]]
AS select_statement [WITH CHECK OPTION]
```

where,

`view_name` specifies the name of the view.

`column_name` specifies the name of the column(s) to be used in a view.

`WITH ENCRYPTION` specifies that the the text of the view will be encrypted in the syscomments view.

`SCHEMABINDING` binds the view to the schema of the underlying table or tables.

`AS` specifies the actions to be performed by the view.

`select_statement` specifies the SELECT statement that defines a view. The view may use the data contained in other views and tables. When the SCHEMABINDING option is used, you need to specify the two-part names (schema.object) of tables, views, or *user-defined functions* that are being referenced in the view.

`WITH CHECK OPTION` forces the data modification statements to meet the criteria given in the SELECT statement defining the view. The data is visible through the view after the modifications have been made permanent.

NOTE *Syscomments view is a system-defined view stored in the database. It contains entries for each view, rule, default, CHECK constraint, DEFAULT constraint, and stored procedure within the database. This view contains a text column that stores the original SQL definition statements.*

Guidelines for Creating Views

While creating views, you should consider the following guidelines:

- ❑ The name of a view must follow the rules for identifiers and must not be the same as that of the table on which it is based.
- ❑ A view can be created only if there is a SELECT permission on its base table.
- ❑ A view cannot derive its data from temporary tables.
- ❑ In a view, ORDER BY cannot be used in the SELECT statement.

For example, to provide access only to the employee ID, marital status, and department ID for all the employees you can create the following code:

```
CREATE VIEW
HumanResources.vwEmployeeDepData
AS
SELECT e.EmployeeID, MaritalStatus,
DepartmentID
FROM HumanResources.Employee e JOIN
HumanResources.EmployeeDepartmentHistory d
```

```
ON e.EmployeeID = d.EmployeeID
```

The preceding code creates the vwEmployeeDepData view containing selected columns from the Employee and EmployeeDepartmentHistory tables.

Restrictions at the Time of Modifying Data Through Views

Views do not maintain a separate copy of the data, but only display the data present in the base tables. Therefore, you can modify the base tables by modifying the data in the view. However, the following restrictions exist while inserting, updating, or deleting data through views:

- ❑ You cannot modify data in a view if the modification affects more than one underlying table. However, you can modify data in a view if the modification affects only one table at a time.
- ❑ You cannot change a column that is the result of a calculation, such as a computed column or an aggregate function.

For example, a view displaying the employee id, manager id, and rate of the employees has been defined by using the following statement:

```
CREATE VIEW vwSal AS
SELECT    i.EmployeeID,    i.ManagerID,
j.Rate FROM HumanResources.Employee AS
i
JOIN  HumanResources.EmployeePayHistory
AS j ON
i.EmployeeID = j.EmployeeID
```

After creating the view, try to execute the following update statement:

```
UPDATE vwSal
SET ManagerID = 2, Rate = 12.45
WHERE EmployeeID = 1
```

The execution of the preceding statement generates an error. This is because the data is being modified in two tables through a single update statement. Therefore, instead of a single UPDATE statement, you need to execute two UPDATE statements for each table.

The following statement would update the EmployeeID attribute in the Employee base table:

```
UPDATE vwSal
SET ManagerID = 2
WHERE EmployeeID = 1
```

The following statement would update the Rate attribute in the EmployeePayHistory table:

```
UPDATE vwSal
SET Rate = 12.45
WHERE EmployeeID = 1
```

Therefore, to modify the data in two or more underlying tables through a view, you need to execute separate UPDATE statements for each table.

You can create an INSTEAD OF trigger on the view to modify data in a view if the modification affects more than one underlying table. You will learn about triggers in Chapter 8.

Managing Views

In addition to creating views, you also need to manage them. Management of a view includes altering, dropping, or renaming views.

Altering Views

If you define a view with a SELECT * statement and then alter the structure of the underlying tables by adding columns, the new columns do not appear in the view. Similarly, when you select all the columns in a CREATE VIEW statement, the columns list is interpreted only when you first create the view. To add new columns in the view, you must alter the view.

You can modify a view without dropping it. This ensures that permissions on the view are not lost. You can modify a view without affecting its dependent objects.

To modify a view, you need to use the ALTER VIEW statement. The syntax of the ALTER VIEW statement is:

```
ALTER VIEW view_name [(column_name)]
[WITH [ENCRYPTION][SCHEMABINDING]
AS select_statement
[WITH CHECK OPTION]
```

where,

`view_name` specifies the view to be altered.

`column_name` specifies the name of the column(s) to be used in a view.

`ENCRYPTION` option encrypts the text of the view in the syscomments view.

`SCHEMABINDING` option binds the view to the schema of the underlying table or tables.

`AS` specifies the actions to be performed by the view.

`select_statement` specifies the SELECT statement that defines a view.

WITH CHECK OPTION forces the data modification statements to follow the criteria given in the SELECT statement.

For example, you created a view to retrieve selected data from the Employee and EmployeeDepartmentHistory tables. You need to alter the view definition by including the LoginID attribute from the Employee table.

To modify the definition, you can write the following statement:

```
ALTER VIEW
HumanResources.vwEmployeeDepData
AS
SELECT e.EmployeeID, LoginID,
MaritalStatus, DepartmentID
FROM HumanResources.Employee e JOIN
HumanResources.EmployeeDepartmentHisto
ry d ON e.EmployeeID = d.EmployeeID
```

The preceding statement alters the view definition by including the LoginID attribute from the Employee table.

Renaming Views

At times, you might need to change the name of a view. You can rename a view without dropping it. This ensures that permissions on the view are not lost. A view can be renamed by using the sp_rename system stored procedure.

The syntax of the sp_rename procedure is:

```
sp_rename old_viewname, new_viewname
```

where,

`old_viewname` is the view that needs to be renamed.

`new_viewname` is the new name of the view.

For example, you can use the following statement to rename the vwSal view:

```
sp_rename vwSal, vwSalary
```

The preceding statement renames the vwSal view as vwSalary.

While renaming views, you must ensure the following points:

- The view must be in the current database.
- The new name for the view must follow the rules for identifiers.
- The view can only be renamed by its owner.
- The owner of the database can also rename the view.

Dropping Views

You need to drop a view when it is no longer required. You can drop a view from a database by using the DROP VIEW statement. When a view is dropped, it has no effect on the underlying table(s). Dropping a view removes its definition and all the permissions assigned to it.

Further, if you query any view that references a dropped table, you receive an error message. Dropping a table that references a view does not drop the view automatically. You have to use the DROP VIEW statement explicitly.

The syntax of the DROP VIEW statement is:

```
DROP VIEW view_name
```

where,

`view_name` is the name of the view to be dropped.

For example, you can use the following statement to remove the vwSalary view:

```
DROP VIEW vwSalary
```

The preceding statement will drop the vwSalary view from the database.

You can drop multiple views with a single DROP VIEW statement. The names of the views that need to be dropped are separated by commas in the DROP VIEW statement.

Indexing Views

Similar to tables, you can create indexes on views. By default, the views created on a table are nonindexed. However, you can index the views when the volume of data in the underlying tables is large and not frequently updated. Indexing a view helps in improving the query performance.

Another benefit of creating an indexed view is that the optimizer starts using the view index in queries that do not directly name the view in the FROM clause. If the query contains references to columns that are also present in the indexed view, and the query optimizer estimates that using the indexed view offers the lowest cost access mechanism, the query optimizer selects the indexed view.

When indexing a view, you need to first create a unique clustered index on a view. After you have defined a unique clustered index on a view, you can create additional nonclustered indexes. When a view is indexed, the rows of the view are stored in the database in the same format as a table.

Guidelines for Creating an Indexed View

You should consider the following guidelines while creating an indexed view:

- A unique clustered index must be the first index to be created on a view.
- The view must not reference any other views, it can reference only base tables.
- All base tables referenced by the view must be in the same database and have the same owner as the view.
- The view must be created with the SCHEMABINDING option. Schema binding binds the view to the schema of the underlying base tables.
- The view must not contain the COUNT (*), MIN, MAX, TOP, UNION, and DISTINCT keywords. In addition, you cannot create an index on a view that contains self join or subquery.
- The view must not contain FULLTEXT

predicates, such as CONTAINS or FREETEXT.

Creating an Indexed View by Using the CREATE INDEX Statement

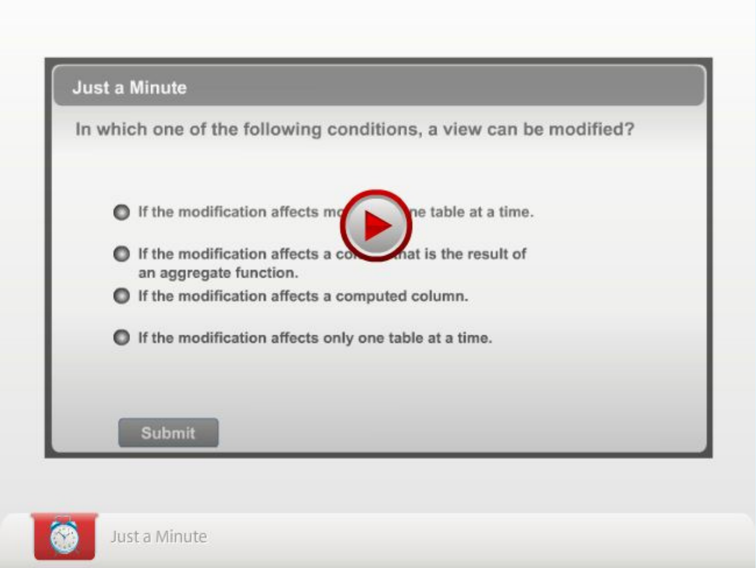
You can create indexes on views by using the CREATE INDEX statement. For example, you can use the following statement for creating a unique clustered index on the vwEmployeeDepData view:

```
CREATE UNIQUE CLUSTERED INDEX
idx_vwEmployeeDepData
ON HumanResources.vwEmployeeDepData
(EmployeeID, DepartmentID)
```

The execution of the preceding statement generates an error. This is because the vwEmployeeDepData view was not bound to the schema at the time of creation. Therefore, before executing the preceding statement, you need to bind the vwEmployeeDepData view to the schema by using the following statement:

```
ALTER VIEW
HumanResources.vwEmployeeDepData WITH
SCHEMABINDING
AS
SELECT e.EmployeeID, MaritalStatus,
DepartmentID
FROM HumanResources.Employee e JOIN
HumanResources.EmployeeDepartmentHisto
ry d
ON e.EmployeeID = d.EmployeeID
```

The preceding statement alters the existing view, vwEmployeeDepData, and binds it with the schema of the underlying tables. You can then create a unique clustered index on the view.



Just a Minute

In which one of the following conditions, a view can be modified?

○ If the modification affects mo [illegible] ne table at a time.

○ If the modification affects a co [illegible] hat is the result of an aggregate function.

○ If the modification affects a computed column.

○ If the modification affects only one table at a time.

Creating Distributed Partitioned Views

Over time, the data in databases becomes huge. This leads to performance problems while retrieving data from the database. Therefore, you can distribute your data over several servers to optimize query performance. Distribution of data over several servers requires partitioning of the tables. You can distribute the data of a table over several servers. In this way, a table is divided into smaller tables containing only a few rows. The data is partitioned between these smaller tables based on a range of data values in one of the columns. To retrieve data from such a table, you need to connect with all the servers where the data is distributed. SQL Server allows you to perform this task by using distributed partitioned views.

SQL Server provides the distributed partitioned view to retrieve data distributed across different servers. Before you implement a partitioned view, you must first partition a table based on a range of key values. For example, the Sales.SalesOrderHeader table contains the details of sales orders. This table contains huge volume of data. Therefore, you decided to partition the Sales.SalesOrderHeader table on the basis of SalesOrderID. The records containing SalesOrderID<50000 are stored in a table named SalesOrderHeader1 on server01 and the records containing SalesOrderID>=50000 are stored in a table named SalesOrderHeader2 on server02.

Execute the following statement on server01 to create the Sales.SalesOrderHeader1 table:

```
SELECT * INTO Sales.SalesOrderHeader1
FROM Sales.SalesOrderHeader where
SalesOrderID < 50000
```

Execute the following statement on server02 to create the Sales.SalesOrderHeader2 table:

```
SELECT * INTO Sales.SalesOrderHeader2
FROM Sales.SalesOrderHeader where
SalesOrderID >= 50000
```

Now, after partitioning the SalesOrderHeader table, you need to retrieve the data after combining the records in the SalesOrderHeader1 and SalesOrderHeader2 tables located on different servers.

Since, the partitioned tables are created on isolated servers, thay are not able to exchange or combine data from each other. To accomplish this task, you need to perform the following tasks:

1. Create linked servers.
2. Create the view.

Creating Linked Servers

You need to accesss data distributed across various servers by creating linked servers. A linked server enables you to connect to another instance of SQL Server and access its data. For example, if you need to access data stored in server01 from server02, you need to create a linked server on server02 for server01. Similarly, if you need to access data stored in server02 from server01, you need to create a linked server on server01 for server02.

Therefore, to access the SalesOrderHeader1 and SalesOrderHeader2 tables, you need to create a linked server on server01 for server02 and vice-versa. You can create a linked server by using the sp_addlinkedserver system stored procedure.

The syntax of the sp_addlinkedserver system stored procedure is:

```
sp_addlinkedserver [ @server= ]
'server' [ , [ @srvproduct= ]
'product_name' ]
 [ , [ @provider= ] 'provider_name' ]
 [ , [ @datasrc= ] 'data_source' ]
```

where,

`[ @server= ] 'server'` is the name of the linked server.

`[ @srvproduct= ] 'product_name'` is the product name of the OLE DB data source to add as a linked server. For SQL Server, its value isSQLServer OLEDB Provider.

`[ @provider= ] 'provider_name'` is a unique programmatic identifier of the OLE DB provider that corresponds to this data source. For SQL Server, its value is SQLOLEDB.

`[ @datasrc= ] 'data_source'` is the name of the server from which the data needs to be accessed.

For example, execute the following statement on server02 to create a linked server named server1 for accessing the data of server01:

```
USE master
GO
EXEC sp_addlinkedserver
 @server = 'server1',
 @srvproduct = 'SQLServer OLEDB
 Provider',
 @provider = 'SQLOLEDB',
 @datasrc = 'server01
```

Similarly, execute the following code on server01 to create a linked server named server2 for accessing the data of server02:

```
USE master
GO
EXEC sp_addlinkedserver
 @server = 'server2',
 @srvproduct = 'SQLServer OLEDB
 Provider',
 @provider = 'SQLOLEDB',
 @datasrc = 'server02'
```

Creating the View

Now,you need to create the distributed partioned views to retrieve data from the tables across different servers. To accomplish this task, execute the following code to create a distributed partitioned view on server01:

```
USE AdventureWorks
GO
CREATE VIEW DistPart_View
AS
SELECT * FROM Sales. SalesOrderHeader1
UNION ALL
SELECT * FROM
server2.AdventureWorks.Sales.SalesOrde
rHeader2
```

Similarly, execute the following code to create a distributed partitioned view on server02:

```
USE AdventureWorks
GO
CREATE VIEW DistPart_View
AS
SELECT * FROM Sales. SalesOrderHeader2
UNION ALL
SELECT * FROM
server1.AdventureWorks.Sales.SalesOrde
rHeader1
```

Now, execute the following query from either of the servers to retrieve the data from both the tables, SalesOrderHeader1 and SalesOrderHeader2:

```
SELECT * FROM DistPart_View
```

The preceding query will display all the records from both the tables, SalesOrderHeader1 and SalesOrderHeader2.

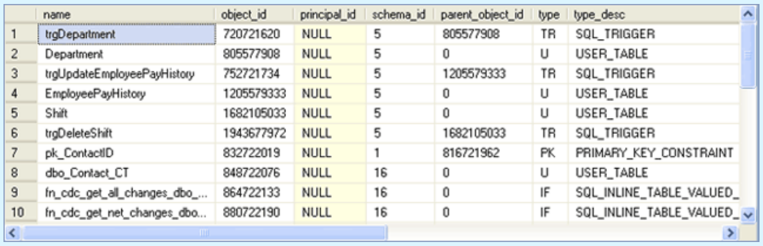
Understanding Catalog Views

The system table stores information about SQL Server configuration, objects, databases, and constraints. The access to these system tables is restricted due to security reasons. Sometimes, you need the information stored in these system tables to manage the database objects. For example, you need to view the database objects that have been modified during the last three days. For this, you need to access the system tables to retrieve the required information. However, you might not have permissions to access the system tables. Therefore, SQL Server provides you with the catalog views to view the data stored in the system tables.

The catalog views represent the information stored in the system tables. For example, you can use the sys.objects catalog view to view the database objects that have been modified during the last three days, as shown in the following query:

```
SELECT * FROM sys.objects
WHERE modify_date > GETDATE() - 3
ORDER BY modify_date;
```

The following figure displays the output of the preceding query.



	name	object_id	principal_id	schema_id	parent_object_id	type	type_desc
1	trgDepartment	720721620	NULL	5	805577908	TR	SQL_TRIGGER
2	Department	805577908	NULL	5	0	U	USER_TABLE
3	trgUpdateEmployeePayHistory	752721734	NULL	5	1205579333	TR	SQL_TRIGGER
4	EmployeePayHistory	1205579333	NULL	5	0	U	USER_TABLE
5	Shift	1682105033	NULL	5	0	U	USER_TABLE
6	trgDeleteShift	1943677972	NULL	5	1682105033	TR	SQL_TRIGGER
7	pk_ContactID	832722019	NULL	1	816721962	PK	PRIMARY_KEY_CONSTRAINT
8	dbo_Contact_CT	848722076	NULL	16	0	U	USER_TABLE
9	fn_cdc_get_all_changes_dbo_...	864722133	NULL	16	0	IF	SQL_INLINE_TABLE_VALUED_
10	fn_cdc_get_net_changes_dbo...	880722190	NULL	16	0	IF	SQL_INLINE_TABLE_VALUED_

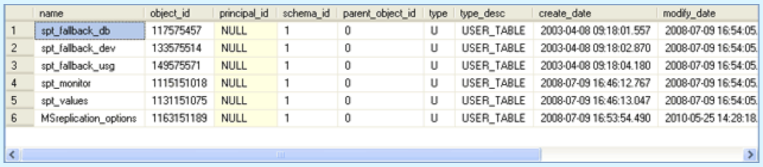
The Output Derived by Using the sys.objects Catalog View

> *The data in the preceding figure may differ depending upon the operations performed in your database.*

The sys.objects catalog view stores information about all the objects, such as tables, views, indexes, stored procedures, and triggers, in the database. For example, you can display all the user tables contained in the master database by executing the following query:

```
SELECT * FROM sys.objects
WHERE type_desc = 'USER_TABLE'
```

The following figure displays the output of the preceding query.



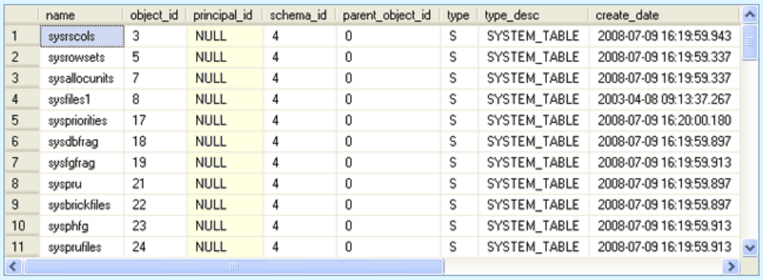
	name	object_id	principal_id	schema_id	parent_object_id	type	type_desc	create_date	modify_date
1	spt_fallback_db	117575457	NULL	1	0	U	USER_TABLE	2003-04-08 09:18:01.557	2008-07-09 16:54:05.
2	spt_fallback_dev	133575514	NULL	1	0	U	USER_TABLE	2003-04-08 09:18:02.870	2008-07-09 16:54:05.
3	spt_fallback_usg	149575571	NULL	1	0	U	USER_TABLE	2003-04-08 09:18:04.180	2008-07-09 16:54:05.
4	spt_monitor	1115151018	NULL	1	0	U	USER_TABLE	2008-07-09 16:46:12.767	2008-07-09 16:54:05.
5	spt_values	1131151075	NULL	1	0	U	USER_TABLE	2008-07-09 16:46:13.047	2008-07-09 16:54:05.
6	MSreplication_options	1163151189	NULL	1	0	U	USER_TABLE	2008-07-09 16:53:54.490	2010-05-25 14:28:18.

The Output Derived by Using the sys.objects Catalog View to Display the User Tables

Similarly, you can display all the system tables in the master database by executing the following query:

```
SELECT * FROM sys.objects
WHERE type_desc = 'SYSTEM_TABLE'
```

The following figure displays the output of the preceding query.



	name	object_id	principal_id	schema_id	parent_object_id	type	type_desc	create_date
1	sysrscols	3	NULL	4	0	S	SYSTEM_TABLE	2008-07-09 16:19:59.943
2	sysrowsets	5	NULL	4	0	S	SYSTEM_TABLE	2008-07-09 16:19:59.337
3	sysallocunits	7	NULL	4	0	S	SYSTEM_TABLE	2008-07-09 16:19:59.337
4	sysfiles1	8	NULL	4	0	S	SYSTEM_TABLE	2003-04-08 09:13:37.267
5	syspriorities	17	NULL	4	0	S	SYSTEM_TABLE	2008-07-09 16:20:00.180
6	sysdbfrag	18	NULL	4	0	S	SYSTEM_TABLE	2008-07-09 16:19:59.897
7	sysfgfrag	19	NULL	4	0	S	SYSTEM_TABLE	2008-07-09 16:19:59.913
8	syspru	21	NULL	4	0	S	SYSTEM_TABLE	2008-07-09 16:19:59.897
9	sysbrickfiles	22	NULL	4	0	S	SYSTEM_TABLE	2008-07-09 16:19:59.897
10	sysphfg	23	NULL	4	0	S	SYSTEM_TABLE	2008-07-09 16:19:59.913
11	sysprufiles	24	NULL	4	0	S	SYSTEM_TABLE	2008-07-09 16:19:59.913

The Output Derived by Using the sys.objects Catalog View to Display the System Tables

To display all the views in the database, you need to execute the following query:

```
SELECT * FROM sys.views
```

Activity 6.2: Creating Views

Implementing a Full-Text Search

While querying data, you can use the LIKE operator to search for a text value in a column. However, at times, you might need to perform a complex search on the data. For example, you need to search for synonyms or antonyms of a particular word. SQL Server allows you to improve the data search by configuring the full-text search feature. The full-text search feature helps you to search for complex strings in the database.

In SQL Server, the full-text search is enabled by default. As a database developer, you should know how to configure the full-text search and how to search data by using full-text search.

Configuring Full-Text Search

The full-text query feature in SQL Server enables users to search for a wide range of text in the SQL tables. Consider an example. The sales management team of AdventureWorks, Inc. makes frequent searches on the ProductDescription table to develop marketing strategies. The search is based on the data stored in the Description column of the table.

A bike racing competition is scheduled to begin in Texas. The sales manager of AdventureWorks wants to see the details of all the bikes that are related to racing, so that a marketing strategy can be designed to increase the sale of these bikes. Specifically, he wants a list of all the bikes that have the keyword 'race winners' in the description.

As the data is large, the search query takes a long time to retrieve data from the table. In this scenario, you can apply a full-text index on the Description column of the ProductDescription table to improve the speed of searching.

To retrieve the required details by using full-text search, you need to configure full-text search on the database. For this, you need to perform the following tasks:

1. Create a full-text catalog.
2. Create a unique index.
3. Create a full-text index.
4. Populate the full-text index.

> *You need to be a member of the sysadmin role to create a full-text catalog and full-text index.*

Creating a Full-Text Catalog

A full-text catalog serves as a container to store full-text indexes. After enabling the full-text search, you need to create a full-text catalog. A full-text catalog is a container that contains full-text indexes. A full-text catalog may have multiple full-text indexes. You can create a full-text catalog by using the following statement:

```
CREATE    FULLTEXT    CATALOG    Cat1    AS
DEFAULT
```

Creating a Unique Index

After creating the full-text catalog, you need to identify a unique index on the table. This unique index will be mapped to the values in the full-text index. You can use an existing unique index defined on the table or create a new one. For example, you can create a unique index on the Production.ProductDescription table, as shown in the following statement:

```
CREATE UNIQUE INDEX Ix_Desc ON
Production.ProductDescription
(ProductDescriptionID)
```

Creating a Full-Text Index

After you have created the full-text catalog and a unique index, you can create a full-text index on the table. A full-text index stores information about significant words and their location within a given column. You can use this information to compute full-text queries that search for rows with particular words or combinations of words. Full-text indexes can be created on the base tables but not on the views or the system tables.

There are certain words that are used often and may hinder a query. These words are called *stopwords* and are excluded from the search string. A database object is used to manage these stopwords. This object is known as stoplist. A stoplist is a list of stopwords. Some of the stopwords are a, an, the, and are.The stoplist, when associated with a full-text index, is applied to full-text queries on that index You can use either the system-defined stoplist or create your own.

For example, if your search string is 'Who is the governor of California', a full-text search will not look for words, such as 'is' and 'the'.

You can create a full-text index by using the CREATE FULLTEXT INDEX statement. The syntax of using the CREATE FULLTEXT INDEX statement is:

```
CREATE FULLTEXT INDEX ON Table_name
( Column_name )
KEY INDEX index_name
[ON fulltext_catalog_name]
[WITH [CHANGE_TRACKING = { MANUAL |
AUTO | OFF [, NO POPULATION ]
[STOPLIST = { OFF | SYSTEM |
stoplist_name }] ]
```

where,

Table_name is the name of the table on which the FULLTEXT index is created.

Column_name is the name of the column included in the full-text index. Only columns of type char, varchar, nchar, nvarchar, text, ntext, image, xml, varbinary, and varbinary (max) can be indexed for full-text search.

KEY INDEX index_name is the name of the unique key index on the table.

fulltext_catalog_name specifies the full-text catalog used for the full-text index. It is optional. If it is not specified, a default catalog is used. If there is no default catalog, SQL Server returns an error.

CHANGE_TRACKING specifies whether changes made to table columns are covered by the full-text index. It can contain the following values:

- ❑ MANUAL: Specifies that the tracked changes must be propagated.
- ❑ AUTO: Specifies that the tracked changes will be propagated automatically as data is modified in the base table.
- ❑ OFF [, NO POPULATION]: Specifies that SQL Server does not keep a list of changes to the indexed data.

STOPLIST associates a full-text stoplist with the index. It can contain the following values:

- ❑ OFF: Specifies that no stoplist be associated with the full-text index.
- ❑ SYSTEM: Specifies that the default full-text system stoplist should be used for this full-text index.
- ❑ stoplist_name: Specifies the user-defined stoplist should be used for this full-text index.

Based on the preceding scenario, you can create a full-text index on the Description column, as shown in the following statement:

```
CREATE FULLTEXT INDEX ON
Production.ProductDescription
(Description) KEY INDEX Ix_Desc
```

The preceding statement will create a full-text index on the ProductionDescription table. This index is based on the Ix_Desc index created earlier on the Description column of the table. If you do not define the stoplist while creating the full-text index, the system-defined full-text stoplist is associated with the index.

NOTE *You can also create a full-text index in the Object Explorer window by right-click the table on which you need to create the full-text index, and select Full-Text index→Define Full-Text Index.*

Populating the Full-Text Index

After creating the full-text index, you need to populate it with the data in the columns enabled for full-text support. SQL Server full-text search engine populates the full-text index through a process called *population*. Population involves filling the index with words and their location in the data page. When a full-text index is created, it is populated by default. In addition, SQL Server automatically updates the full-text index as the data is modified in the associated tables.

However, SQL Server does not keep a list of changes made to the indexed data when the CHANGE_TRACKING option is OFF. This option is specified while creating the full-text index by using the CREATE FULLTEXT INDEX statement.

If you do not want the full-text index to be populated when it is created by using the CREATE FULLTEXT INDEX statement, then you must specify NO POPULATION along with the CHANGE TRACKING OFF option. To populate the index, you need to execute the ALTER FULLTEXT INDEX command along with the START FULL, INCREMENTAL, or UPDATE

POPULATION clause.

For example, to create an empty full-text index on the ProductDescription table, you can execute the following statement:

```
CREATE FULLTEXT INDEX ON
Production.ProductDescription
(Description)
KEY INDEX
PK_ProductDescription_ProductDescripti
onID
WITH CHANGE_TRACKING OFF, NO
POPULATION
```

To populate the index, you need to execute the following statement:

```
ALTER FULLTEXT INDEX ON
Production.ProductDescription START
FULL POPULATION
```

The preceding statement will populate the full-text index created on the ProductDescription table.

Similar to regular SQL indexes, full-text indexes can also be updated automatically as the data is modified in the associated tables. This repopulation can be time-consuming and adversely affect the usage of resources of the database server during periods of high database activity. Therefore, it is better to schedule repopulation of full-text indexes during periods of low database activity.

You can specify the following types of full-text index population methods to repopulate the index:

- ❑ Full population
- ❑ Change tracking-based population
- ❑ Incremental timestamp-based population

Full Population

You can use this method when you need to populate the full-text catalog or the full-text index for the first time. After that, you can the maintain the indexes by using change tracking or incremental populations.

During a full population of a full-text catalog, index entries are built for all the rows in all the tables covered by the catalog. If a full population is requested for a table, index entries are built for all the rows in that table.

Change Tracking-Based Population

SQL Server maintains a record of the rows that have been modified in a table set up for full-text indexing. These changes are propagated to the full-text index.

Incremental Timestamp-Based Population

The incremental population method updates the full-text index with the data that has been changed since the last time the index was refreshed. For an incremental population refresh to work, the indexed table must have a column of the timestamp data type. If a table does not have a column of the timestamp data type, then only a full population refresh can be done.

Searching Data by Using a Full-Text Search

After the full-text index has been created on a table, you can query the table by using the full-text predicates. The full-text predicates are used to specify how the search string should be searched in the table. The following predicates can be used while performing the full-text search:

- ❑ **FREETEXT**: When the search criteria are given, FREETEXT searches for any variation of a word or a group of words given in the search column. FREETEXT is used for the prefix searches.

  Considering the previous scenario of the bike racing competition, you can use the FREETEXT predicate to obtain the desired output, as shown in the following query:

  ```
  SELECT Description FROM
  Production.ProductDescription
  WHERE FREETEXT (Description,
  'race winners')
  ```

- ❑ **FREETEXTTABLE**:It searches the character type columns and returns a table of rows that contain the words nearer to searched data in meaning. However, it does not return the exact searched word. It expands the searched terms to include plurals, gender, and verb forms. For example, while searching the word 'fog', the search is expanded to include fog, fogging, fogged, and so on. In addition, it returns the ranking value and full-text key for each row.

  It is referenced in the FROM clause of the SELECT statement similar to the regular table. It returns a table that contains a column named KEY. This column contains full-text key values. Each full-text indexed table has a KEY column that contains unique values. The values returned in the KEY column are the full-text key values that match the selection criteria specified in the search condition.

  The returned table also contains a column named RANK. This column contains values from 0 through 1000 for each row indicating how well a row matches the searched condition. You can use the ORDER BY clause to return the highest-ranking rows as the first rows in the result set. The syntax of using the FREETEXTTABLE predicate is:

  ```
  FREETEXTTABLE (table ,
  { column_name | (column_list) |
  * } ,'search_string')
  ```

  where,

  `table` is the name of the table to be used for full-text querying.

`column_name` is the name of one or more full-text indexed columns of the table specified in the FROM clause.

`column_list` indicates that you can specify several columns, separated by a comma.

`search_string` is the text to be searched in the column.

For example, you can use the following query to search the term 'Chromony steel' in the Production.ProductDescription table:

```
SELECT
f.RANK,ProductDescriptionID,Descr
iption
From
Production.ProductDescription d
INNER JOIN FREETEXTTABLE
(Production.ProductDescription,
Description,'Chromony steel')f
ON d.ProductDescriptionID = f.
[KEY]
ORDER BY f.RANK DESC
```

In the preceding query, f.[KEY] represents the full-text key values in the table returned by FULLTEXTTABLE. This key is matched with the key of the primary table to get the desired matching data.

❑ **CONTAINS**: This predicate is used in queries when you want to search for a specific phrase or for the exact match. It also searches for the proximity of words within a text. For example, you can use the following query to search for the word 'Ride' near the word 'Bike' in the ProductDescription table:

```
SELECT Description FROM
Production.ProductDescription
WHERE CONTAINS (Description,
'ride NEAR bike')
```

If you want to search for those products who contain the different forms of the word, ride, then you can use the following query:

```
SELECT Description FROM
Production.ProductDescription
WHERE CONTAINS (Description,
'FORMSOF (INFLECTIONAL, ride)')
```

In the preceding query, INFLECTIONAL keyword is used to search for different tenses of word, ride.

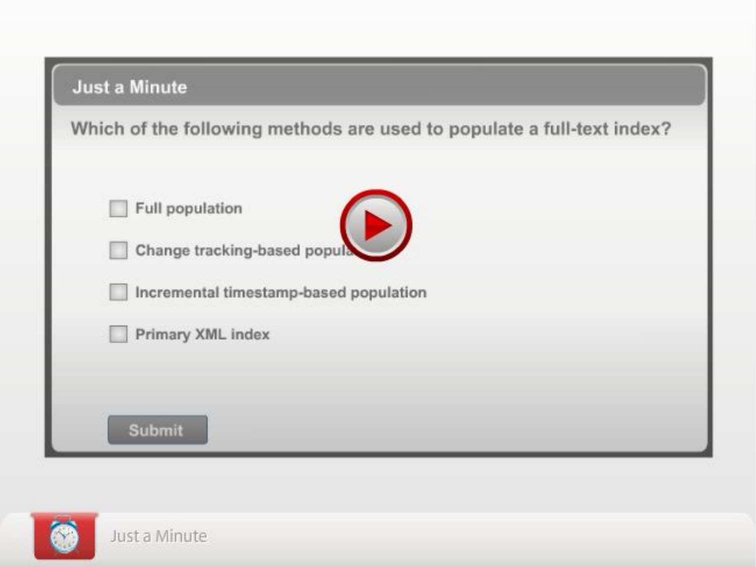
You can also use logical operators to specify different search conditions. For example, consider the following query:

```
SELECT Description FROM
Production.ProductDescription
WHERE CONTAINS (Description,
'"Advanced" AND "Seat"')
```

In the preceding query, the AND logical operator is used to specify the CONTAINS predicate should search for those descriptions which contain both the words, Advanced and Seat.

❑ **CONTAINSTABLE**: It returns the search result in the form of a table. The resulting table consists of two columns, a key column named KEY and a rank column named RANK. The actual rows returned by these functions are joined with the source table. The CONTAINSTABLE predicate is referenced in the FROM clause of the SELECT statement. Its syntax is similar to the FREETEXTABLE predicate. For example, you can use the following query to return the matching word for 'Entry' in the Production.ProductDescription table:

```
SELECT
f.RANK,ProductDescriptionID,Descr
iption
From
Production.ProductDescription d
INNER JOIN CONTAINSTABLE
(Production.ProductDescription,
Description,'Entry ')f
ON d.ProductDescriptionID = f.
[KEY]
ORDER BY f.RANK DESC
```



Just a Minute

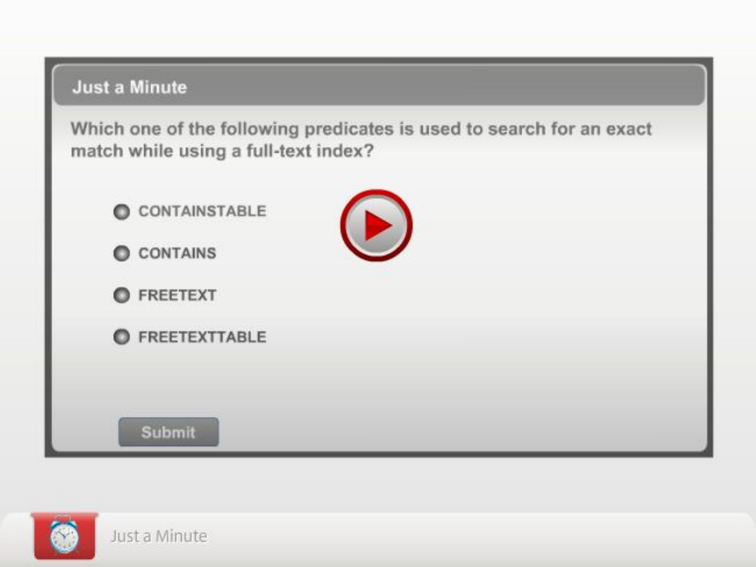
Which of the following methods are used to populate a full-text index?

□ Full population

□ Change tracking-based popula

□ Incremental timestamp-based population

□ Primary XML index


Just a Minute

Which one of the following predicates is used to search for an exact match while using a full-text index?

○ CONTAINSTABLE

○ CONTAINS

○ FREETEXT

○ FREETEXTTABLE

Activity 6.3: Implementing a Full-Text Search

Summary

In this chapter, you learned that:

- Indexes are created to enhance the performance of queries.
- There are two types of indexes, clustered and nonclustered.
- Indexes are created by using the CREATE INDEX statement.
- Clustered indexes should be built on an attribute whose values are unique and do not change often. Data is physically sorted in a clustered index.
- In a nonclustered index, the physical order of rows is not the same as that of the index order.
- A nonclustered index is the default index that is created with the CREATE INDEX command.
- An XML index is built on columns with the XML data type.
- Indexes can also be partitioned based on the value ranges.
- The common index maintenance tasks include disabling, enabling, renaming, and dropping an index.
- SQL Server provides the execution plan to view the details of execution of a SQL query.
- The execution plan of a query enables you to identify how the SQL Server query optimizer executes that query.
- SQL Server provides the following types of formats to view the execution plan:
    - Graphical plan
    - Text plan
    - XML plan
- The graphical execution plan provides the execution details of a query in the form of a tree structure.
- The text execution plan displays all the information regarding the execution plan in the form of a single representation.
- An XML plan displays the execution plan in an XML format.
- Hints instruct the database engine on how to execute the query.
- SQL Server allows you to provide the following types of hints:
    - Query hints
    - Join hints
        - Loop join
        - Merge join
        - Hash join
    - Table hints
- A viewis a virtual table, which derives its data from one or more tables known as the base or underlying tables.
- Views serve as security mechanisms, thereby protecting data in the base tables.
- SQL Server allows data to be modified only in one of the underlying tables when using views, even if the view is derived from multiple underlying tables.
- SQL Server provides the distributed partitioned view to retrieve data across different servers.
- SQL Server provides catalog views that represent the information stored in the system tables.
- SQL Server enables users to search for a wide range of text in the SQL tables through the full-text query feature.
- A full-text catalog can be created by using the CREATE FULLTEXT CATALOG statement.
- A full-text index can be created by using the CREATE FULLTEXT INDEX statement.
- The types of full-text population methods are full population, change tracking-based population, and incremental timestamp-based population.
- Full-text predicates that can be used to perform full-text search are CONTAINS, CONTAINSTABLE, FREETEXT, and FREETEXTTABLE.
- A FREETEXT predicate searches for the word or the words given in the search column.
- The CONTAINS predicate searches for a specific phrase or for the exact match.

Reference Reading

Creating and Managing Indexes

Reference Reading: Books	***Reference Reading: URLs***
Beginning SQL Server 2012 for Developers by Robin Dewson	*http://rusanu.com/2012/05/29/inside-the-sql-server-2012-columnstore-indexes/*

Creating and Managing Views

Reference Reading: Books	***Reference Reading: URLs***
Beginning Microsoft SQL Server 2012 Programming by Paul Atkinson and Robert Vieira	*http://msdn.microsoft.com/en-us/library/ms187956.aspx*

Implementing a Full-Text Search

Reference Reading: Books	***Reference Reading: URLs***
Professional Microsoft SQL Server 2012 Administration by Adam Jorgensen, Steven Wort, and Ross LoForte	*http://msdn.microsoft.com/en-us/library/ms142571.aspx*

Chapter 7

Implementing Stored Procedures and Functions

As a database developer, you might need to execute a set of SQL statements together. SQL Server allows you to create batches with multiple statements that can be executed together. These batches can also contain programming constructs that include conditional logic to examine conditions before executing the statements.

At times, it might be required to execute a batch repeatedly. In such a case, a batch can be saved as database objects called stored procedures or functions. These database objects contain a precompiled batch that can be executed many times without recompilation.

This chapter explains how to create batches to execute multiple SQL statements. Further, it explains how to implement stored procedures and functions in SQL Server.

Objectives

In this chapter, you will learn to:

- ❑ Implement batches
- ❑ Implement stored procedures
- ❑ Implement functions

Implementing Batches

As a database developer, you might need to execute more than one SQL statement to perform a task. For example, when a new employee joins AdventureWorks, Inc., you need to insert the employee details in the database. The details of the employees are stored in more than one table. Therefore, you need to execute multiple insert statements to store the details in each table. In such a case, you can send all the SQL statements together to SQL Server to be executed as a unit. This helps in reducing the network traffic.

At times, you might also need to check conditions before executing the SQL statements. For example, in a manufacturing unit, the InventoryIssue table stores the details of an item issued for the manufacturing process. When you insert a record in this table, you need to check that the quantity on hand is more than or equal to the quantity issued. In such a case, you can create conditional constructs that check for a condition before executing a statement.

Creating Batches

A*batch* is a group of SQL statements submitted together to SQL Server for execution. While executing batches, SQL Server compiles the statements of a batch into a single executable unit called an *execution plan*. This helps in saving execution time.

For example, you have to execute 10 statements and you are executing them one by one by sending 10 requests. This process takes time if your queries are in a queue. All the statements might not get executed together. Instead, if you execute all the 10 statements together in a batch, then the execution process becomes faster as all the statements are sent to the server together.

To create a batch, you can write multiple SQL statements followed by the keyword GO at the end. The syntax of creating a batch is:

```
<T-SQL Statement1>
<T-SQL Statement2>
<T-SQL Statement3>
…
GO
```

GO is a command that specifies the end of the batch and sends the SQL statements for execution.

For example, if you want to store the details of new employees in the AdventureWorks database, you can write multiple INSERT statements in a batch, as shown in the following statements:

```
INSERT INTO [AdventureWorks].[Person].
[Contact]
VALUES (0, null, 'Robert', 'J',
'Langdon', NULL ,'rbl@adventure-
works.com', 0, '1 (11) 500
555-0172' ,'9E685955-ACD0-4218-
AD7F-60DDF224C452', '2a31OEw=', NULL
, newid(), getdate())

INSERT INTO [AdventureWorks].
[HumanResources].[Employee]
VALUES ('AS01AS25R2E365W', 19978,
'robertl', 16, 'Tool Designer',
'1972-05-15', 'S', 'M', '1996-07-31',
0, 16, 20, 1, newid(), getdate())
GO
```

When a batch is submitted to SQL Server, it is compiled to create an execution plan. If any compilation error occurs, such as a syntax error, the execution plan is not created. Therefore, none of the statements in the batch is executed. However, if a run-time error occurs after the execution plan is created, the execution of the batch stops. In such a case, the statements executed before the statement that encountered the run-time error are not affected.

Using Variables

While creating batches, you might need to store some values temporarily during the execution time. For example, you might need to store some intermediate values while performing calculations. To store the intermediate values, you can declare variables and assign values to them. You can declare a variable by using the DECLARE statement. A variable name is always preceded by the "@" symbol. The syntax of the DECLARE statement is:

```
DECLARE @variable_name data_type
```

Variables that are declared in a batch and can be used in any statement inside the batch are called *local variables*.

The following statements declare a variable, @Rate, and assigns the maximum value of the Rate column from the EmployeePayHistory table to the variable:

```
DECLARE @Rate int
SELECT @Rate = max(Rate)
FROM HumanResources.EmployeePayHistory
GO
```

In the preceding statements, the max aggregate function is used to retrieve the maximum pay rate from the EmployeePayHistory table.

Displaying User-Defined Messages

At times, you need to display values of variables or user-defined messages when the batch is executed. For this, you can use the PRINT statement.

The following statements display the value of the @Rate variable by using the PRINT statement:

```
DECLARE @Rate int
SELECT @Rate = max(Rate)
FROM HumanResources.EmployeePayHistory
PRINT @Rate
GO
```

You can also use comment entries in batches to write a description of the code. This will help understand the purpose of the code. A comment entry can be written in the following ways:

- Multiple line comment entries enclosed within /* and */
- Single line comment entry starting with -- (double hyphens)

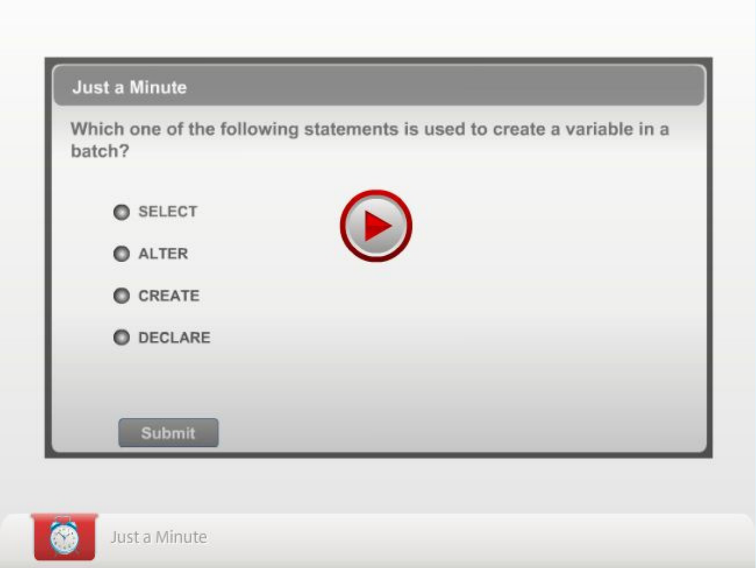
Guidelines to Create Batches

While creating batches, you need to consider the following guidelines:

- You cannot combine statements such as CREATE DEFAULT, CREATE FUNCTION, CREATE PROCEDURE, CREATE RULE, CREATE TRIGGER, and CREATE VIEW with other statements while creating a batch. Any statement that follows the create statement is interpreted as part of the definition.
- You can use the EXECUTE statement in a batch when it is not the first statement of the batch, otherwise the EXECUTE statement works implicitly.

In addition, you need to consider the following restrictions:

- You cannot bind rules and defaults to columns and use them in the same batch.
- You cannot define and use the CHECK constraint in the same batch.
- You cannot drop objects and recreate them in the same batch.
- You cannot alter a table by adding a column and then referring to the new column in the batch created earlier.



Just a Minute

Which one of the following statements is used to create a variable in a batch?

○ SELECT

○ ALTER

○ CREATE

○ DECLARE

Using Constructs

SQL Server allows you to use programming constructs in batches for conditional execution of statements. For example, you need to retrieve data based on a condition. If the condition is not satisfied, a message should be displayed.

SQL Server allows you to use the following constructs to control the flow of statements:

- IF…ELSE statement
- CASE statement
- WHILE statement

Using the IF…ELSE Statement

You can use the IF…ELSE statement for conditional execution of SQL statements. A particular action is performed when the given condition evaluates to TRUE, and another action is performed when the given condition evaluates to FALSE.

The syntax of the IF…ELSE statement is:

```
IF boolean_expression
 {sql_statement | statement_block}
ELSE
 {sql_statement | statement_block}]
```

where,

`boolean_expression` specifies the condition that evaluates to either TRUE or FALSE.

`sql_statement` specifies a T-SQL statement.

statement_block is a collection of T-SQL statements. You need to use the BEGIN and END keywords to define multiple T-SQL statements.

For example, you can retrieve the pay rate of an employee from the EmployeePayHistory table to a variable, @Rate. The value of the @Rate variable is compared with the value 15 by using the < (less than) comparison operator. Based on the condition, different messages are displayed, as shown in the following statements:

```
DECLARE @Rate money
SELECT    @Rate    =    Rate    FROM
HumanResources.EmployeePayHistory
WHERE EmployeeID = 23
IF @Rate < 15
 PRINT 'Review of the rate is required'
ELSE
BEGIN
PRINT  'Review  of  the  rate  is  not
required'
 PRINT 'Rate ='
 PRINT @Rate
END
GO
```

In the preceding statements, the IF statement checks whether the @Rate variable is storing a value less than 15. If the result is true, the PRINT statement displays "Review of the rate is required", else it displays "Review of the rate is not required". Further, the next PRINT statement displays the value of the rate.

Consider another example, where a check is performed to see the existence of the Sales department, as shown in the following statement:

```
IF    EXISTS    (SELECT    *    FROM
HumanResources.Department WHERE Name =
'Sales')
BEGIN
 SELECT             *             FROM
 HumanResources.Department WHERE Name =
 'Sales'
END
ELSE
 PRINT   'Department   details   not
 available'
GO
```

In the preceding statement, if the Sales department exists, all the details are displayed; otherwise, a user-defined message is displayed.

Using the CASE Statement

You can use the CASE statement in situations where several conditions need to be evaluated. The CASE statement evaluates a list of conditions and returns one of the possible results. You can use the IF statement to do the same task. However, you can use a CASE statement when there are more than two conditions that check a common variable for different values. The syntax of the CASE statement is:

```
CASE input_expression
 WHEN       when_expression       THEN
 result_expression
 [[WHEN      when_expression      THEN
 result_expression] [...]]
 [ELSE else_result_expression]
END
```

where,

input_expression specifies the input expression that is evaluated. input_expression is any valid expression.

when_expression is the expression that is compared with input_expression.

result_expression is the expression returned when the comparison of input_expression with when_expression evaluates to TRUE. This can be a constant, a column name, a function, a query, or any combination of arithmetic, bit-wise, and string operators.

else_result_expression is the expression returned if no comparison operation evaluates to TRUE. If this argument is omitted and no comparison operation evaluates to TRUE, the result will be NULL.

In a simple CASE construct, a variable or an expression is compared with the expression in each WHEN clause. If any of these expressions evaluate to TRUE, then the expression specified with the THEN clause is executed. If the expression does not evaluate to TRUE, the expression with the ELSE statement is executed.

Consider the following statements, where a CASE construct is included in the SELECT statement to display the marital status as "Married" or "Single":

```
SELECT EmployeeID, 'Marital Status' =
CASE MaritalStatus
 WHEN 'M' THEN 'Married'
 WHEN 'S' THEN 'Single'
 ELSE 'Not specified'
END
FROM HumanResources.Employee
GO
```

Using the WHILE Statement

You can use the WHILE statement in a batch to allow a set of T SQL statements to execute repeatedly as long as the given condition holds true. The syntax of the WHILE statement is:

```
WHILE boolean_expression
 {sql_statement | statement_block}
 [BREAK]
 {sql_statement | statement_block}
 [CONTINUE]
```

where,

boolean_expressionis an expression that evaluates to TRUE or FALSE.

sql_statementis any SQL statement.

statement_block is a group of SQL statements.

BREAK causes the control to exit from the WHILE loop.

CONTINUE causes the WHILE loop to restart, skipping all the statements after the CONTINUE keyword.

SQL Server provides the BREAK and CONTINUE statements to control the statements within the WHILE loop. The BREAK statement causes an exit from the WHILE loop. Any statements that appear after the END keyword, which marks the end of the loop, are executed after the BREAK statement is executed. The CONTINUE statement causes the WHILE loop to restart, skipping any statements after this statement inside the loop.

For example, the HR department of AdventureWorks, Inc. has decided to review the salary of all the employees. As per the current HR policy, the average hourly salary rate of all the employees should be approximately $20. You need to increase the hourly salary of all the employees until the average hourly salary reaches near $20. In addition, you need to ensure that the maximum hourly salary should not exceed $127. To accomplish this task, you can use the following statement:

```
WHILE    (SELECT    AVG(Rate)+1    FROM
HumanResources.EmployeePayHistory)    <
20
BEGIN
 UPDATE
 HumanResources.EmployeePayHistory
 SET Rate = Rate + 1
 FROM HumanResources.EmployeePayHistory
 IF (SELECT max(Rate)+1 FROM
   HumanResources.EmployeePayHistory)
 >127
 BREAK
 ELSE
 CONTINUE
END
```

Handling Errors and Exceptions

When you execute a query, it is parsed for syntactical errors before execution. If the syntax is correct, it is compiled and executed. Sometimes, due to factors, such as incorrect data, an error can occur during execution even if the query is syntactically correct. The errors that occur at run time are known as *exceptions*.

For example, there is a primary key constraint applied on the EmployeeID attribute of the Employee table. When you try to insert an employee ID that already exists in the table, an error occurs while executing the INSERT statement.

Exceptions can be handled in the following ways:

- By using the TRY…CATCH construct
- By using the RAISERROR statement and handling the error in the application

Using TRY…CATCH

A TRY…CATCH construct includes a TRY block followed by a CATCH block. A TRY block is a group of SQL statements enclosed in a batch, stored procedure, trigger, or function. If an error occurs in any statement of the TRY block, the control is passed to another group of statements that are enclosed in a CATCH block.

A CATCH block contains SQL statements that perform some operations when an error occurs. Therefore, an associated CATCH block must immediately follow a TRY block, as shown in the following syntax:

```
TRY
<SQL statements>
…
CATCH
<SQL statements>
…
END CATCH
```

If there are no errors in the code that is enclosed in a TRY block, the control is passed to the statement immediately after the associated END CATCH statement. In this case, statements enclosed in the CATCH block are not executed.

The TRY…CATCH construct can be nested. Either a TRY block or a CATCH block can contain nested TRY… CATCH constructs. A CATCH block can contain an embedded TRY…CATCH construct to handle errors encountered by the CATCH block.

In the CATCH block, you can use the following system functions to determine the information about errors:

- **ERROR_LINE()**: Returns the line number at which the error occurred.
- **ERROR_MESSAGE()**: Specifies the text of the message that would be returned to the application.
- **ERROR_NUMBER()**: Returns the error number.
- **ERROR_PROCEDURE()**: Returns the name of the stored procedure or trigger in which the error occurred. This function returns NULL if the error did not occur within a stored procedure or trigger.
- **ERROR_SEVERITY()**: Returns the severity.
- **ERROR_STATE()**: Returns the state of the error.

For example, the EmployeeID attribute of the Employee table in the AdventureWorks database is an IDENTITY column and its value cannot be specified while inserting a new record. However, if you specify the value for EmployeeID in the INSERT statement, an error will occur.

To handle such run-time errors, you can include the insert statement in a TRY block and send the control to the CATCH block where the error information is displayed, as shown in the following statements:

```
BEGIN TRY
INSERT  INTO  [AdventureWorks].[Person].
[Contact]
VALUES   (0,   null,   'Robert',   'J',
'Langdon',     NULL     ,'rbl@adventure-
works.com',     0,     '1    (11)    500
555-0172'        ,'9E685955-ACD0-4218-
AD7F-60DDF224C452', '2a31OEw=', NULL,
newid(), getdate())
INSERT INTO [AdventureWorks].
[HumanResources].[Employee]
VALUES ('AS01AS25R2E365W', 19979,
'robertl', 16, 'Tool Designer',
'1972-05-15', 'S', 'M', '1996-07-31',
0, 16, 20, 1, newid(), getdate())
END TRY
BEGIN CATCH
SELECT 'There was an error! ' +
ERROR_MESSAGE() AS ErrorMessage,
 ERROR_LINE() AS ErrorLine,
 ERROR_NUMBER() AS ErrorNumber,
 ERROR_PROCEDURE() AS ErrorProcedure,
 ERROR_SEVERITY() AS ErrorSeverity,
 ERROR_STATE() AS ErrorState
END CATCH
GO
```

Using RAISERROR

A RAISERROR statement is used to return messages to the business applications that are executing the SQL statements. This statement uses the same format as a system error or warning message generated by the database engine. For example, consider an application that is executing a batch. If an error occurs while executing this batch, an error message will be raised and sent to the application. The application, in turn, will include the code to handle the error.

You can also return user-defined error messages by using the RAISERROR statement. The syntax of the RAISERROR statement is:

```
RAISERROR('Message', Severity, State)
```

where,

`Message` is the text that you want to display.

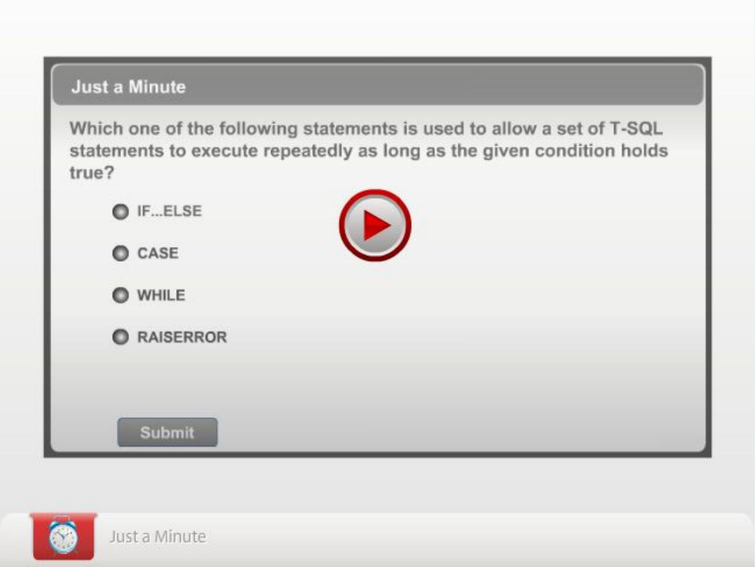
`Severity` is the user-defined severity level associated with the message. It represents how serious the error is. Severity level can range from 0 to 25. The levels from 0 through 18 can be specified by any user. The preferable value used by users is 10, which is for displaying informational error messages. Severity levels from 20 through 25 are considered fatal.

`State` is an integer value from 0 through 255. If the same user-defined error is raised at multiple locations, using a unique state number for each location, it can help find which section of the code is raising the errors.

A RAISERROR severity of 11 to 19 executed in the TRY block causes the control to be transferred to the associated CATCH block. For example, you have to update the Shift table to store the details of the shift in which the employees work. While updating the shift details, you need to ensure that the difference between the start time and the end time is eight hours, as shown in the following statements:

```
BEGIN TRY
DECLARE @Start datetime
DECLARE @End datetime
DECLARE @Date_diff int
SELECT @Start = '1900-01-01
23:00:00.000', @End = '1900-01-02
06:00:00.000'
SELECT @Date_diff = datediff(hh,
@Start, @End)
IF (@Date_diff != 8)
 RAISERROR('Error Raised', 16, 1)
ELSE
BEGIN
 UPDATE HumanResources.Shift
 SET StartTime = @Start, EndTime = @End
 WHERE ShiftID = 3
END
END TRY
BEGIN CATCH
 PRINT 'The difference between the
 Start and End time should be 8
 hours'
END CATCH
GO
```

In the preceding statements, if the difference between the start time and the end time is less than eight hours, an error is raised, and the update process is stopped.



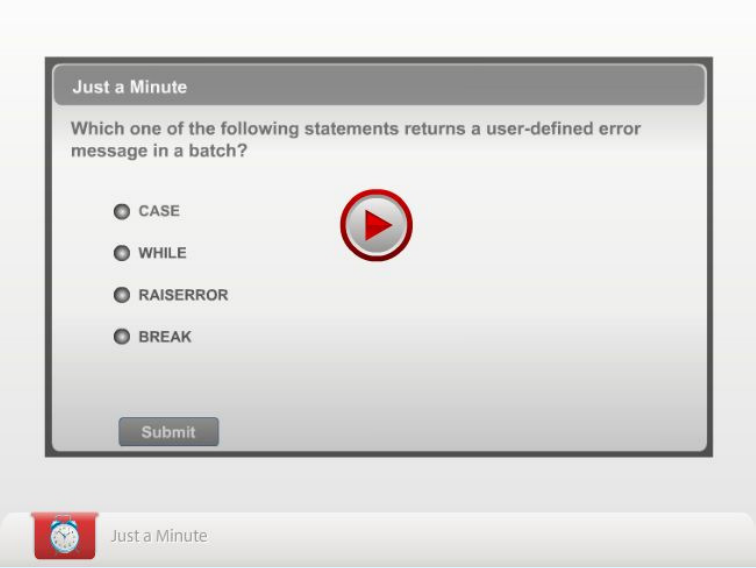
Which one of the following statements is used to allow a set of T-SQL statements to execute repeatedly as long as the given condition holds true?

○ IF...ELSE

○ CASE

○ WHILE

○ RAISERROR



Just a Minute

Which one of the following statements returns a user-defined error message in a batch?

○ CASE

○ WHILE

○ RAISERROR

○ BREAK

Implementing Stored Procedures

Batches are temporary in nature. To execute a batch more than once, you need to recreate SQL statements and submit them to the server. This leads to an increase in the overhead, as the server needs to compile and create the execution plan for these statements again. Therefore, if you need to execute a batch multiple times, you can save it within a stored procedure. A *stored procedure* is a precompiled object stored in the database.

Stored procedures can invoke the Data Definition Language (DDL) and Data Manipulation Language (DML) statements and can return values. If you need to assign values to the variables declared in the procedures at run time, you can pass parameters while executing them. You can also execute a procedure from another procedure. This helps in using the functionality of the called procedure within the calling procedure.

An application is designed in three layers. These layers are the presentation, business, and data access layers. An application is well maintained if the stored procedures are used. These procedures are indeed a part of the data access layer. They are mapped to certain methods of the business layer. The application can access the required data from the database by invoking the specific methods of the business layer. These methods, in turn, invoke the statements or the procedures in the data access layer. Therefore, the entire process is abstracted.

As a database developer, it is important for you to learn how to implement procedures.

Creating Stored Procedures

You can create a stored procedure by using the CREATE PROCEDURE statement.

The syntax of the CREATE PROCEDURE statement is:

```
CREATE PROCEDURE proc_name
WITH    [ENCRYPTION][RECOMPILE][EXECUTE
AS]
AS
BEGIN
 sql_statement1
 sql_statement2
END
```

where,

proc_name specifies the name of the stored procedure.

ENCRYPTION is used to encrypt the original text of the CREATE PROCEDURE statement.

RECOMPILE specifies that the stored procedure is recompiled every time it executes.

EXECUTE AS specifies the security context under which the stored procedure is executed. It can take any one of the following values:

- ❑ CALLER: Specifies that the stored procedure is executed under the execution context of the user executing it.
- ❑ OWNER: Specifies that the stored procedure is executed under the execution context of its owner.
- ❑ user_name: Specifies that the stored procedure is executed under the execution context of the specified user.
- ❑ SELF: Specifies that the stored procedure is executed under the execution context of the user creating or modifying it.

The following statement creates a stored procedure to view the department names from the Department table:

```
CREATE PROCEDURE prcDept
AS
BEGIN
 SELECT              Name              FROM
 HumanResources.Department
END
```

When the CREATE PROCEDURE statement is executed, the server compiles the procedure and saves it as a database object. The procedure is then available for various applications to execute.

The process of compiling a stored procedure involves the following steps:

1. The procedure is compiled and its components are broken into various pieces. This process is known as *parsing*.
2. The existence of the referred objects, such as tables and views, are checked. This process is known as *resolving*.
3. The name of the procedure is stored in the sysobjects table and the code that creates the stored procedure is stored in the syscomments table.
4. The procedure is compiled and a blueprint for how the query will run is created. This blueprint

is specified as an execution plan. The execution plan is saved in the procedure cache.

5. When the procedure is executed for the first time, the execution plan will be read, fully optimized, and then run. When the procedure is executed again in the same session, it will be read directly from the cache. This increases performance, as there is no repeated compilation.

> *After creating the stored procedure, you can view the code of the procedure by using the sp_helptext statement.*

Guidelines to Create a Stored Procedure

The following points need to be considered before creating a stored procedure:

- ❑ You cannot combine the CREATE PROCEDURE statement with other SQL statements in a single batch.
- ❑ You must have the CREATE PROCEDURE permission to create a procedure in the database and the ALTER permission on the schema, where the procedure is being created.
- ❑ You can create a stored procedure only in the current database.

After creating a stored procedure, you can execute the procedure. You can also alter the procedure definition or drop it, if the existing procedure is not required.

Executing a Stored Procedure

A procedure can be executed by using the EXECUTE and EXEC statement. The syntax of the EXECUTE statement is:

```
EXEC | EXECUTE proc_name
```

where,

proc_name is the name of the procedure that you need to execute.

You can execute the stored procedure, prcDept, as shown in the following statement:

```
EXEC prcDept
```

Altering a Stored Procedure

A stored procedure can be modified by using the ALTER PROCEDURE statement. The syntax of the ALTER PROCEDURE statement is:

```
ALTER PROCEDURE proc_name
```

You can alter the stored procedure by using the following statement:

```
ALTER PROCEDURE prcDept
AS
BEGIN
SELECT   DepartmentID,   Name   FROM
HumanResources.Department
END
```

In the preceding statement, the prcDept stored procedure will be modified to display department Ids along with the department name.

When you execute a stored procedure, the number of rows affected by it is also returned. If the stored procedure is getting executed by a Web application, then the database engine returns the result set along with the number of rows affected. This creates an overhead on the network and reduces the performance of the application. To avoid this, you can turn off the message that contains the number of rows affected by the SQL statement. You can perform this task by using the SET NOCOUNT statement. The syntax of this statement is:

```
SET NOCOUNT { ON | OFF }
```

where,

`ON` prevents the message containing the count of the number of rows affected by the SQL statement or stored procedure from being returned.

`OFF` allows the message containing the count of the number of rows affected by the SQL statement or stored procedure to be returned.

For example, consider the following code snippet:

```
ALTER PROCEDURE prcDept
AS
BEGIN
SET NOCOUNT ON
SELECT    DepartmentID,    Name    FROM
HumanResources.Department
END
```

When you execute the prcDept stored procedure, the result set does not display the number of rows affected by the procedure.

Dropping a Stored Procedure

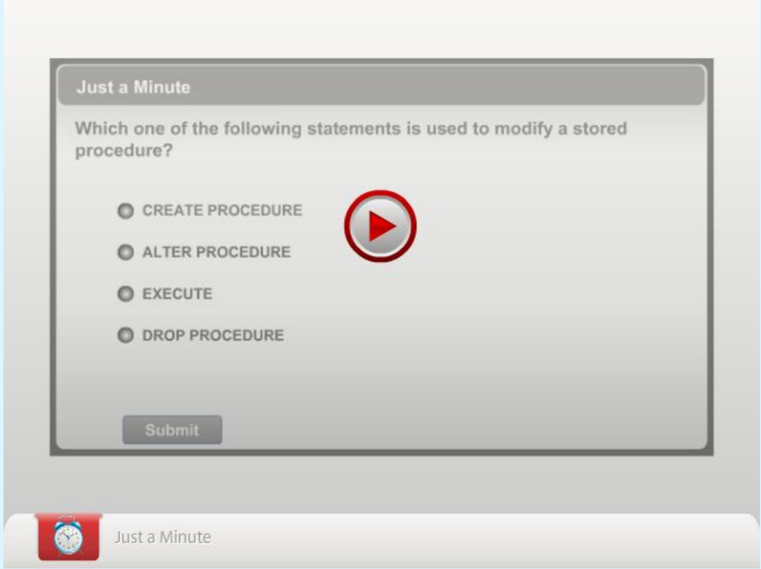
You can drop a stored procedure from the database by using the DROP PROCEDURE statement. The syntax of the DROP PROCEDURE statement is:

```
DROP PROCEDURE proc_name
```

You cannot retrieve a procedure once it is dropped.

You can drop the prcDept stored procedure by using the following statement:

```
DROP PROCEDURE prcDept
```



Just a Minute

Which one of the following statements is used to modify a stored procedure?

○ CREATE PROCEDURE

○ ALTER PROCEDURE

○ EXECUTE

○ DROP PROCEDURE

Creating Parameterized Stored Procedures

At times, you need to execute a procedure for different values of a variable that are provided at run time. For this, you can create a parameterized stored procedure. Parameters are used to pass values to the stored procedure during run time. These values can be passed by using standard variables. The parameter that passes the value to the stored procedure is defined as an input parameter. A stored procedure has the capability of using a maximum of 2100 parameters. Each parameter has a name, data type, direction, and a default value.

> NOTE
>
> *Direction represents whether a parameter is an input parameter or an output parameter. By default, the direction of a parameter is input.*

The following statement creates a stored procedure displaying the employee ID, the login ID, and the title of employees that have the same title provided as an input during execution:

```
CREATE PROC prcListEmployee @title
char(50)
AS
BEGIN
 PRINT 'List of Employees'
 SELECT EmployeeID, LoginID, Title
 FROM HumanResources.Employee
 WHERE Title = @title
END
```

You can execute the stored procedure, prcListEmployee, by using the following statement:

```
EXECUTE prcListEmployee 'Tool
Designer'
```

While executing stored procedures, you can also provide the values for the parameters by explicitly specifying the name and value of the parameter. In the previous example, you can also pass the parameter value by using the name of variable, as shown in the following statement:

```
EXECUTE prcListEmployee @title = 'Tool
Designer'
```

Apart from variables, you can pass a table-valued parameter to a stored procedure. The table-valued parameter enables you to send multiple rows of data to the stored procedure without creating a temporary table or many parameters. It must be declared with the READONLY option while creating a stored procedure. This is because you cannot modify the data in the rows of the table-valued parameter.

For example, create a table named Branch having columns as Branch_ID and Branch_Name by using the following code:

```
CREATE TABLE Branch(
Branch_ID int Primary Key,
Branch_Name Varchar (20)
)
```

Now, you need to insert data into the Branch table by using a stored procedure. For this, you need to create a stored procedure that accepts multiple values to be inserted in the Branch table. You can perform this task easily by using a table-valued parameter. Execute the following code to create a table type named NType:

```
CREATE TYPE NType AS TABLE (Br_ID int,
Br_Name Varchar(20))
```

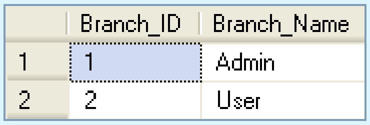
In the preceding code, the NType table type consists of two columns, Br_ID and Br_Name. Now, you need to create a stored procedure that takes a parameter of table type by using the following code:

```
CREATE procedure Table_Proc
@ParamTable NType READONLY
AS
INSERT INTO Branch
(Branch_ID,Branch_Name)
SELECT * FROM @ParamTable
```

You can execute the Table_Proc stored procedure by using the following code:

```
DECLARE @TableTest AS NType
INSERT INTO @TableTest (Br_ID,
Br_Name) VALUES(1,'Admin'), (2,'User')
EXECUTE Table_Proc @TableTest
SELECT * FROM BRANCH
```

When the preceding statements are executed, two rows are inserted in the Branch table, as shown in the following figure.



	Branch_ID	Branch_Name
1	1	Admin
2	2	User

The Output Derived After Executing the Procedure

Returning Values from Stored Procedures

Similar to providing input values to the procedures at run time, you can also return values as an output from the procedures. The values can be returned to the calling application through output parameters. To specify a parameter as the output parameter, you can use the OUTPUTkeyword.

The OUTPUT keyword has to be specified in both the CREATE PROCEDURE and the EXECUTE statements. If the OUTPUT keyword is omitted, the procedure will be executed but will not return any value.

The syntax of declaring an output parameter using the OUTPUT keyword is:

```
CREATE PROCEDURE procedure_name
[
{@parameter data_type} [OUTPUT]
]
AS
sql_statement [...n]
```

@parameter data_type [OUTPUT] allows the stored procedure to pass a data value to the calling procedure. If the OUTPUT keyword is not used, then the parameter is treated as an input parameter.

You can also return values from the stored procedure by using the RETURN statement. The RETURN statement allows the stored procedure to return only an integer value to the calling application. The syntax of the RETURN statement is:

```
RETURN value
```

where,

`value` is any integer.

If a value is not specified, then the stored procedure returns a default value of 0 to specify success and a nonzero value to specify failure.

For example, you need to display the details of an employee whose employee ID has been provided as an input. For this, you need to create a procedure prcGetEmployeeDetail that will accept employee ID as an input and return the department name and ID of the shift in which the employee works. You can create the procedure, as shown in the following statement:

```
CREATE PROCEDURE prcGetEmployeeDetail
@EmpId int, @DepName char(50) OUTPUT,
@ShiftId int OUTPUT
AS
BEGIN
IF EXISTS(SELECT * FROM
HumanResources.Employee WHERE
EmployeeID = @EmpId)
BEGIN
  SELECT @DepName = d.Name, @ShiftId =
h.ShiftID
  FROM HumanResources.Department d
JOIN
  HumanResources.EmployeeDepartmentHis
tory h
  ON d.DepartmentID = h.DepartmentID
  WHERE EmployeeID = @EmpId AND
h.Enddate IS NULL
  RETURN 0
END
ELSE
  RETURN 1
END
```

In the preceding statement, the prcGetEmployeeDetail procedure accepts the employee ID as an input parameter and returns the department name and shift ID as the output parameters. The procedure first checks the existence of the given employee ID. If it exists, the procedure returns an integer value 0 along with the required details.

Calling a Procedure from Another Procedure

At times, you might need to use the values returned by a procedure in another procedure. For this, you can execute or call one procedure from another procedure. A procedure that calls or executes another procedure is known as the calling procedure, and the procedure that is called or executed by the calling procedure is termed as the called procedure. You can also execute a procedure from another procedure if you need to use the functionality provided by one into another.

Consider the previous example where the prcGetEmployeeDetail procedure returns the employee details for a given employee ID. You can create the prcDisplayEmployeeStatus procedure, which accepts the employee ID of an employee as an input and displays the department name and shift ID where the employee is working along with the manager ID and the title of the employee. To perform this task, you need to call the prcGetEmployeeDetail procedure from the prcDisplayEmployeeStatus procedure, as shown in the following statement:

```
CREATE PROCEDURE
prcDisplayEmployeeStatus @EmpId int
AS
BEGIN
DECLARE @DepName char(50)
DECLARE @ShiftId int
DECLARE @ReturnValue int
EXEC @ReturnValue =
prcGetEmployeeDetail @EmpId,
    @DepName OUTPUT,
    @ShiftId OUTPUT
IF (@ReturnValue = 0)
```

```
BEGIN
  PRINT 'The details of an employee
 with ID: ' + convert(char(10),
  @EmpId)
  PRINT 'Department Name: ' + @DepName
  PRINT 'Shift ID: ' + convert( char
 (1), @ShiftId)
  SELECT ManagerID, Title FROM
 HumanResources.Employee
  WHERE EmployeeID = @EmpID
END
ELSE
  PRINT 'No records found for the given
 employee'
END
```

To execute the prcDisplayEmployeeStatus procedure, you need to execute the following statement:

```
EXEC prcDisplayEmployeeStatus 2
```

> NOTE
>
> *SQL Server provides a function, @@ROWCOUNT, which returns the number of rows affected by the last statement. You can use this statement in the IF construct to check the result of the last executed statement.*

What is a stored procedure?

A stored procedure is a named, precompiled batch of DDL or DML statements that is permanently stored in the database.

Using System Stored Procedures

System stored procedures provide functionalities to manage system resources. By using the system stored procedure, database administrators can manage the database server effectively. SQL Server provides a set of system-defined stored procedures used for general maintenance of a particular database or the database server as a whole. The following table lists some of the system-defined stored procedures in SQL Server.

System-defined Stored Procedure	***Description***	***Example***
sp_addsrvrolemember	*Is used to add a login as a member of a fixed server role.*	*The following statement adds George to the sysadmin fixed server role:* `sp_addsrvrolemember 'George', 'sysadmin'`
sp_addrolemember	*Is used to add a database user, database role, Windows login, or Windows group to a database role in the current database.*	*The following statement adds the database user George to the db_securityadmin role in the current database:* `sp_addrolemember 'db_securityadmin', 'George'`
sp_helpdb	*Is used to see the details of the database.*	*The following statement displays the details of the AdventureWorks database:* `sp_helpdb AdventureWorks`
sp_renamedb	*Is used to rename a database.*	*The following statement renames the ImportExport database to Export:* `sp_renamedb ImportExport , Export`
sp_bindrule	*Is used to bind a rule with a column or an alias data type.*	*The following statement binds the ruleType rule to the LeaveType column of the EmployeeLeave table in the HumanResources schema:* `sp_bindrule 'ruleType','HumanResources.EmployeeLeave.LeaveType'`
sp_unbindrule	*Is used to unbind a rule from a column or alias data type.*	*The following statement unbinds the rules bound to*

		the LeaveType column of the EmployeeLeave table in the HumanResources schema: `sp_unbindrule 'HumanResources.EmployeeLeave.LeaveType'`
sp_bindefault	Is used to bind a default to a column or an alias data type.	The following statement binds the default value CL to the LeaveType column of the EmployeeLeave table in the HumanResources schema: `sp_bindefault 'CL' 'HumanResources.EmployeeLeave.LeaveType'`
sp_unbindefault	Is used to unbind or remove a default from a column or an alias data type in the current database.	The following statement unbinds the default bound to the LeaveType column of the EmployeeLeave table in the HumanResources schema: `sp_unbindefault 'HumanResources.EmployeeLeave.LeaveType'`
sp_rename	Is used to rename a table, index, or view.	The following statement renames the EmployeeVacation table in the HumanResources schema to EmployeeLeave: `sp_rename [HumanResources.EmployeeVacation], [HumanResources.EmployeeLeave]`
sp_help	Is used to see the details of a table, built-in data types, or user-defined data types.	The following statement displays the details of the Department table in the HumanResources schema: `sp_help 'HumanResources.Department'`
sp_helpindex	Is used to display information about the indexes on a table or a view.	The following statement displays the details of the indexes on the Department table in the HumanResources schema: `sp_helpindex 'HumanResources.Department'`

sp_xml_preparedocument	Is used to parse the XML document. This stored procedure reads the XML document and parses it with the MSXML parser.	The following statements parse the XML document stored in the @XMLDoc variable and stores the output in the @Doc variable:
		`DECLARE @Doc int`
		`DECLARE @XMLDoc nvarchar(1000)`
		`SET @XMLDoc = N'<ROOT><Customer ID="JH01"></Customer></ROOT>'`
		`EXEC sp_xml_preparedocument @Doc OUTPUT, @XMLDoc`

sp_xml_removedocument	*Is used to release the memory by removing the internal tree representation of the parsed XML document created by using the sp_xml_preparedocument stored procedure.*	*The following statement releases the memory by removing the internal tree representation of the parsed XML document stored in the @Doc variable:* `EXEC sp_xml_removedocument @Doc`
sp_fulltext_catalog	*Is used to create and drop a full-text catalog. In addition, it is used to start and stop the indexing action for a catalog.*	*The following statement creates a full-text catalog named F_Cat:* `EXEC sp_fulltext_catalog 'F_Cat', 'create'`
sp_helptext	*Is used to display the definition for database objects, such as a user-defined rule, a default unencrypted T-SQL stored procedure, a user defined T-SQL function, a trigger, a CHECK constraint, or a view.*	*The following statement displays the definition of the dEmployee trigger in the HumanResources schema:* `EXEC sp_helptext 'HumanResources.dEmployee'`
sp_settriggerorder	*Is used to change the sequence of execution of the AFTER triggers.*	*The following statement specifies that the trgDeleteShift1trigger is the first trigger to be fired after a DELETE operation occurs on the table:* `sp_settriggerorder 'HumanResources.trgDeleteShift1', 'FIRST', 'DELETE'`
sp_configure	*Is used to display or change global configuration settings for the current server.*	*The following statement changes the global configuration setting of the current server to 1 (enabled):* `sp_configure CLR_ENABLED, 1`
sp_lock	*Is used to display information about locks.*	*The following statement displays the details of the locks available on the current instance of SQL Server:* `sp_lock`
sp_recompile	*Is used to recompile the stored procedures and triggers, the next time they are run.*	*The following statement marks the dEmployee trigger in the HumanResources schema for recompilation:* `sp_recompile 'HumanResources.dEmployee'`
sp_dbremove	*Is used to remove a database and all files associated with that database.*	*The following statement removes the Export database and its associated files:* `sp_dbremove 'Export'`
sp_helptrigger	*Is used to display the type or types of DML triggers defined on the specified table for the current database.*	*The following statement displays the details of the DML triggers on the Employee table in the HumanResources schema:* `sp_helptrigger 'HumanResources.Employee'`
sp_addlinkedsrvlogin	*Is used to create or update a mapping between a login on the local instance of SQL Server and a security account on a remote server.*	*The following statement creates a mapping to ensure that the logins to the local server connect to the SQLSERVER02*

		linked server by using their own user credentials: *sp_addlinkedsrvlogin 'SQLSERVER02'*

sp_executesql	*Is used to execute a T-SQL statement or batch.*	*The following statements execute the T-SQL statement stored in the @SQLString variable:*
		DECLARE @SQLString nvarchar (500);
		SET @SQLString = 'SELECT EmployeeID, NationalIDNumber FROM AdventureWorks.HumanResources.Employee WHERE EmployeeID = 2';
		EXEC sp_executesql @SQLString

The System-defined Stored Procedures

Implementing Dynamic SQL

SQL statements can be broadly classified into two categories, static and dynamic. Static SQL statements do not change at the time of execution and can be hard-coded in the application. The complete text of static SQL statements is known at the compile time. A stored procedure that does not expect any parameter at the time of execution is the simplest example of static SQL. At times, your query depends on certain values or parameters, which are not fixed and are provided at the time of execution. Therefore, the complete text of the SQL statement is not known until execution. Such statements constitute dynamic SQL.

For example, you write a procedure, which calculates the total amount payable, on the basis of the principle amount, loan duration, and rate of interest. The execution of this procedure depends on these values provided at runtime.

SQL Server provides the following ways to implement dynamic SQL:

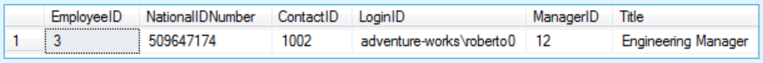
- ❑ Write a parameterized query.
- ❑ Use the EXEC() function.
- ❑ Use SP_EXECUTESQL.

Writing a Parameterized Query

You can pass parameters to the WHERE clause of the SQL statements. Consider a scenario, where you need to find out the details of the employee whose EmployeeID is provided at runtime. For this, you can execute the following statements:

```
DECLARE @empid int
SET @empid = 3
SELECT * FROM HumanResources.Employee
WHERE
HumanResources.Employee.EmployeeID =
@empid
```

In the preceding statements, an integer type parameter, @empid, is declared and assigned the value, 3. You can change the value of this parameter to display the details of different employees. The preceding statements display the details of the employee whose ID is 3, as shown in the following figure.



	EmployeeID	NationalIDNumber	ContactID	LoginID	ManagerID	Title
1	3	509647174	1002	adventure-works\roberto0	12	Engineering Manager

The Output Derived After Executing the Statements

Using the EXEC() Function

The EXEC() function can be used to execute the procedures as well as the SQL statements. This function accepts the SQL statement to be executed as a parameter.

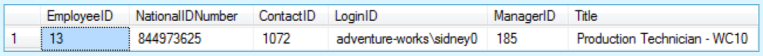
Consider the following statements that show the execution of a static SQL statement:

```
EXEC('SELECT EmployeeID,
NationalIDNumber,ContactID,LoginID,Man
agerID,Title FROM
HumanResources.Employee WHERE
HumanResources.Employee.EmployeeID =
13')
```

The EXEC() function can also be used to execute dynamic SQL statements. For example, the following code shows the use of EXEC() to execute dynamic SQL statements:

```
DECLARE @sqlCommand varchar(1000)
DECLARE @columnList varchar(75)
DECLARE @empid varchar(10)
SET @columnList =
'EmployeeID,NationalIDNumber,ContactID
,LoginID,ManagerID,Title'
SET @empid = '13'
SET @sqlCommand = 'SELECT ' +
@columnList +
' FROM HumanResources.Employee WHERE
HumanResources.Employee.EmployeeID = '
+ @empid
EXEC (@sqlCommand)
```

The preceding statements display the details of the employee whose ID is 13, as shown in the following figure.



	EmployeeID	NationalIDNumber	ContactID	LoginID	ManagerID	Title
1	13	844973625	1072	adventure-works\sidney0	185	Production Technician - WC10

The Output Derived After Executing the EXEC() Function

Using SP_EXECUTESQL

The SP_EXECUTESQL stored procedure is a better option to execute SQL statements. Unlike the EXEC() function, it supports parameter substitution. The use of SP_EXECUTESQL generates an execution plan which can be reused by SQL Server. Consider the preceding example given for the EXEC() function, which expects empid as a parameter. If we execute the above statement, once with empid as 13 and again with empid as 14, SQL Server will generate two different execution plans.

However, in the case of SP_EXECUTESQL, the same execution plan will be reused by the SQL Server query optimizer for different parameters passed at the time of execution. Therefore, there is no overhead of compiling a new execution plan for every new parameter passed. This, in turn, contributes to query performance. The following syntax is used for SP_EXECUTESQL:

```
sp_executesql [ @stmt = ] statement
[
{ , [ @params = ] N'@parameter_name
data_type [ OUT | OUTPUT ][ ,...n ]' }
  { , [ @param1 = ]
'value1' [ ,...n ] }
]
```

where,

[@stmt=] statement specifies the Unicode string that contains a Transact-SQL statement or batch to be executed.

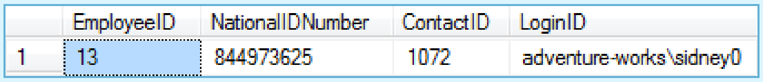
[@params=] N'@parameter_namedata_type [,... n] ' specifies the string that contains the definitions of all parameters defined in @stmt.

[@param1=] 'value1' specifies the value of the first parameter defined in @params. Similarly, you need to specify the value of all the parameters in sequence as defined in @params.

For example, consider the following statements:

```
DECLARE @sqlCommand nvarchar(1000)
DECLARE @columnList varchar(75)
DECLARE @ID varchar(75)
SET @columnList = 'EmployeeID
,NationalIDNumber
,ContactID
,LoginID'
SET @ID = '13'
SET @sqlCommand = 'SELECT ' +
@columnList +
' FROM HumanResources.Employee WHERE
HumanResources.Employee.EmployeeID =
@empid'
EXECUTE sp_executesql @sqlCommand,
N'@empid nvarchar(75)', @empid = @ID
```

The preceding statements display the details of the employee whose ID is 13, as shown in the following figure.



	EmployeeID	NationalIDNumber	ContactID	LoginID
1	13	844973625	1072	adventure-works\sidney0

The Output Derived by Using the SP_EXECUTESQL Statement

Activity 7.1: Creating Stored Procedures

Implementing Functions

Similar to stored procedures, you can also create functions to store a set of T-SQL statements permanently. These functions are also referred to as User-Defined Functions (UDFs). A *UDF* is a database object that contains a set of T-SQL statements, accepts parameters, performs an action, and returns the result of that action as a value. The return value can be either a single scalar value or a result set.

UDFs have a limited scope as compared to stored procedures. You can create functions in situations when you need to implement a programming logic that does not involve any permanent changes to the database objects outside the function. For example, you cannot modify a database table from a function.

UDFs are of different types, scalar functions and table-valued functions. As a database developer, you must learn to create and manage different types of UDFs.

Creating UDFs

A UDF contains the following components:

- ❑ Function name with optional schema/owner name
- ❑ Input parameter name and data type
- ❑ Options applicable to the input parameter
- ❑ Return parameter data type and optional name
- ❑ Options applicable to the return parameter
- ❑ One or more T-SQL statements

To create a function, you can use the CREATE FUNCTION statement. The syntax of the CREATE FUNCTION statement is:

```
CREATE FUNCTION [ schema_name. ]
function_name
( [ { @parameter_name [ AS ]
[ type_schema_name. ]
parameter_data_type
```

```
 [ = default ] }
 [ ,...n ]
   ]
)
RETURNS return_data_type
 [    WITH    [SCHEMABINDING][ENCRYPION]
 [EXECUTE AS] [,...n ] ]
 [ AS ]
 BEGIN
   function_body
   RETURN expression
 END
 [ ; ]
```

where,

schema_name is the name of the schema to which the UDF belongs.

function_name is the name of the UDF. Function names must comply with the rules for identifiers and must be unique within the database and to its schema.

@parameter_name is a parameter in the UDF. One or more parameters can be declared.

[type_schema_name.] parameter_data_type is the data type of the parameter, and optionally the schema to which it belongs.

[= default] is a default value for the parameter. While executing a UDF with the default value, you must specify the DEFAULT keyword for the parameter having the default value.

return_data_type is the return value of a scalar user-defined function.

function_body specifies a series of T-SQL statements.

ENCRYPTION option is used to convert the text of the function body into the encrypted format.

SCHEMABINDING option is used to bind the function to the schema of the data objects it references. As a result, you cannot execute the ALTER or DROP statement to modify or to remove the tables, columns, views that are referenced by the UDF.

EXECUTE AS clause is used to allow a user to grant all the necessary rights to execute the functions. The access rights included in the EXECUTE AS clause are similar to those specified with stored procedures.

Creating Scalar Functions

Scalar functions accept a single parameter and return a single data value of the type specified in the RETURNS clause. A scalar function can return any data type except text, ntext, image, cursor, and timestamp. Some scalar functions, such as current_timestamp, do not require any arguments.

A function contains a series of T-SQL statements defined in a BEGIN...END block of the function body that returns a single value.

For example, consider a scalar function that calculates the monthly salary of employees. This function accepts the pay rate and returns a single value after multiplying the rate with the number of hours and number of days. You can create this function by using the following statement:

```
CREATE                              FUNCTION
HumanResources.MonthlySal     (@PayRate
float)
RETURNS float
AS
BEGIN
RETURN (@PayRate * 8 * 30)
END
```

You can execute the preceding function by using the following statements:

```
DECLARE @PayRate float
SET            @PayRate                =
HumanResources.MonthlySal(12.25)
PRINT @PayRate
```

In the preceding statements, @PayRate is a variable that will store a value returned by the MonthlySal function.

Creating Table-Valued Functions

A table-valued function returns a table as an output, which can be derived as part of a SELECT statement. Table-valued functions return the output as a table data type. The table data type is a special data type used to store a set of rows. Table-valued functions are of the following types:

❑ Inline table-valued function

❑ Multistatement table-valued function

Inline Table-Valued Function

An inline table-valued function returns a variable of a table data type from the result set of a single SELECT statement. An inline function does not contain a function body within the BEGIN and END statements.

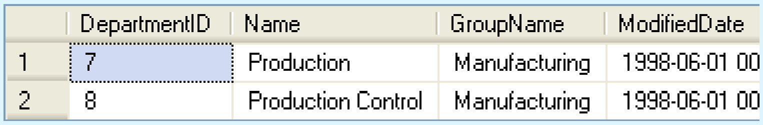
For example, the inline table-valued function, fx_Department_GName, accepts a group name as a parameter and returns the details of the departments that belong to the group from the Department table. You can create the function by using the following statement:

```
CREATE   FUNCTION   fx_Department_GName
( @GrName nvarchar(20) )
RETURNS table
AS
RETURN (
 SELECT *
 FROM HumanResources.Department
 WHERE GroupName=@GrName
 )
GO
```

You can use the following query to execute the fx_Department_GName function with a specified argument:

```
SELECT   *   FROM   fx_Department_GName
('Manufacturing')
```

The preceding query will return a result set, as shown in the following figure.



	DepartmentID	Name	GroupName	ModifiedDate
1	7	Production	Manufacturing	1998-06-01 00
2	8	Production Control	Manufacturing	1998-06-01 00

The Output Derived After Executing the Query

Consider another example of an inline function that accepts rate as a parameter and returns all the records that have a rate greater than the parameter value. You can create this function, as shown in the following statement:

```
CREATE FUNCTION HumanResources.Emp_Pay
(@Rate int)
RETURNS table
AS
RETURN (
 SELECT e.EmployeeID, e.Title, er.Rate
 FROM HumanResources.Employee AS e
 JOIN HumanResources.EmployeePayHistory
 AS er
 ON e.EmployeeID=er.EmployeeID WHERE
 er.Rate>@Rate
)
GO
```

In the preceding statement, the Emp_Pay function will return a result set that displays all the records of the employees who have the pay rate greater that the parameter. You can execute the preceding function by using the following query:

```
SELECT * FROM HumanResources.Emp_Pay
(50)
```

Multistatement Table-Valued Function

A multistatement table-valued function uses multiple statements to build the table that is returned to the calling statement. The function body contains a BEGIN...END block, which holds a series of T-SQL statements to build and insert rows into a temporary table. The temporary table is returned in the result set and is created based on the specification mentioned in the function.

For example, the multistatement table-valued function, PayRate, is created to return a set of records from the EmployeePayHistory table. You can create this function by using the following statement:

```
CREATE FUNCTION PayRate (@rate money)
RETURNS @table TABLE
(EmployeeID int NOT NULL,
RateChangeDate datetime NOT NULL,
Rate money NOT NULL,
PayFrequency tinyint NOT NULL,
ModifiedDate datetime NOT NULL)
AS
BEGIN
 INSERT @table
 SELECT *
 FROM HumanResources.EmployeePayHistory
 WHERE Rate > @rate
 RETURN
END
```

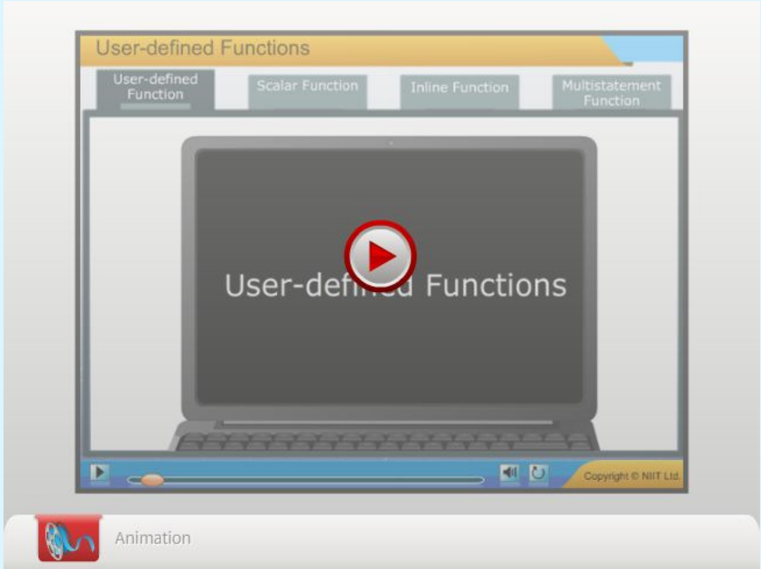
In the preceding statement, the function returns a result set in the form of a temporary table, @table, created within the function. You can execute the function by using the following query:

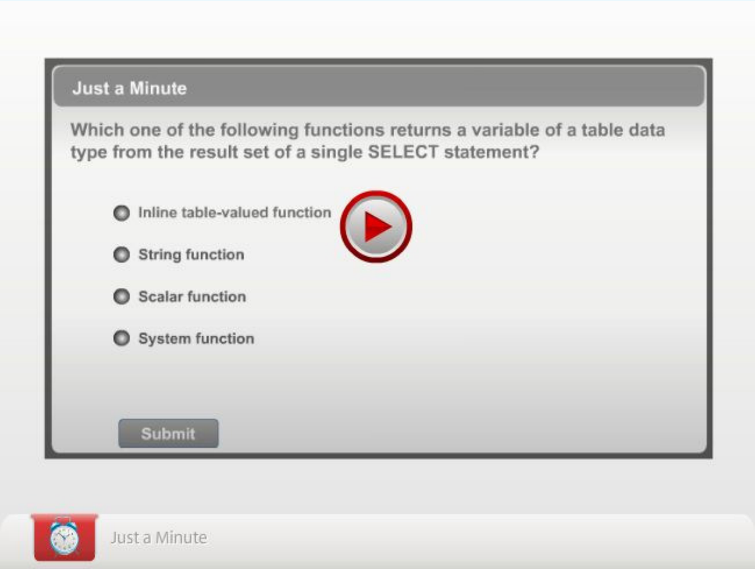
```
SELECT * FROM PayRate(45)
```

NOTE

Depending on the result set returned by a function, a function can be categorized as deterministic or nondeterministic. Deterministic functions always return the same result whenever they are called with a specific set of input values. However, nondeterministic functions may return different results each time they are called with a specific set of input values.

An example of a deterministic function is dateadd(), which returns the same result for any given set of argument values for its three parameters. getdate() is a nondeterministic function because it is always invoked without any argument, but the return value changes on every execution.





Just a Minute

Which one of the following functions returns a variable of a table data type from the result set of a single SELECT statement?

○ Inline table-valued function

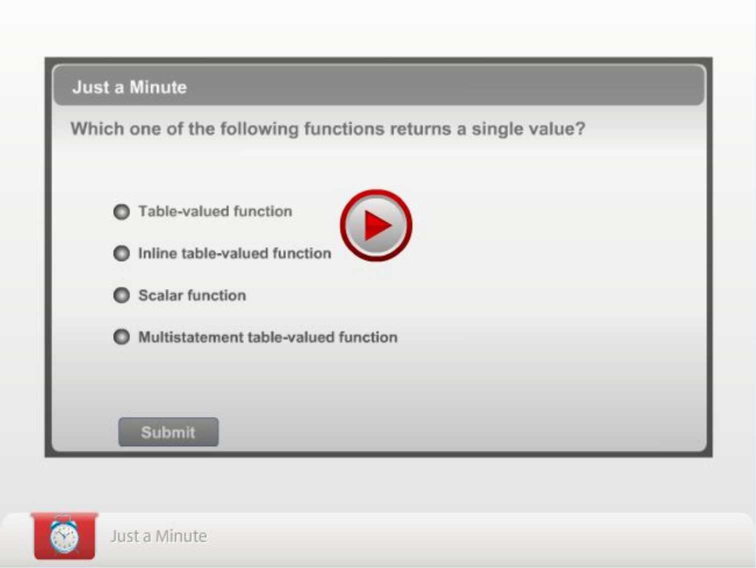
○ String function

○ Scalar function

○ System function

Submit

Just a Minute



Just a Minute

Which one of the following functions returns a single value?

○ Table-valued function

○ Inline table-valued function

○ Scalar function

○ Multistatement table-valued function

Activity 7.2: Creating Functions

Implementing Cursors

You have been following a set-based approach to work with data in SQL Server. This approach allows working on multiple rows together as a single unit. Being a relational database system, SQL Server is designed to process the sets, which are an unordered collection of rows, also known as a result set.

It means that multiple rows are handled together in an unordered manner during query execution. However, cursors follow an iterative approach. This means that the database engine handles a single row at a time in a specific order.

The following list helps you to understand how the set-based approach is different from an iterative approach:

- ❑ In case of the set-based approach, you need to instruct the database engine what is to be done without specifying the how part. But in the iterative approach, you also need to specify how you want the engine to do a particular job.
- ❑ The Iterative approach involves the use of the SELECT statement, loop management, navigation, and state-checking. Therefore, it involves more of code writing in comparison to the set- based approach.
- ❑ Since the iterative approach allows accessing one row at a time, it is slower than the set-based approach.

A cursor in SQL Server allows you to work with individual rows of a result set. Associating a cursor with the result set enables you to process the result set one row at a time. A cursor allows you to perform the following operations:

- ❑ Navigate to a specific row of the result set.
- ❑ Retrieve a row from a particular position.
- ❑ Modify a row at a particular position.

Implementing a cursor includes the following tasks:

- ❑ Declare a cursor.
- ❑ Open the cursor.
- ❑ Fetch rows from the cursor.
- ❑ Close the cursor.
- ❑ Deallocate the cursor.

Declaring a Cursor

A cursor is declared by defining the SQL statement or the query used to build the result set on which the cursor operates. Additional attributes, such as the scrolling behavior or the scope of the cursor, are also defined at the time of declaration. The following syntax is used to declare a cursor:

```
DECLARE cursor_name CURSOR [ LOCAL |
GLOBAL ]
   [ FORWARD_ONLY | SCROLL ]
 [ STATIC | KEYSET | DYNAMIC |
 FAST_FORWARD ]
 [ READ_ONLY | SCROLL_LOCKS |
 OPTIMISTIC ]
 FOR select_statement
 [ FOR UPDATE [ OF column_name
 [ ,...n ] ] ]
 [;]
```

where,

Cursor_name is the name of the cursor.

LOCAL specifies that the scope of the cursor is local. It means that the cursor can be referenced only the batch, stored procedure, or trigger in which it has been created.

GLOBAL specifies that the scope of the cursor is global. It means that the cursor can be referenced in any batch, stored procedure, or trigger executed by the current connection.

FORWARD_ONLY specifies that the cursor can be scrolled in a forward direction only.

SCROLL specifies that all the fetch options are available and the cursor can fetch data in all directions.

STATIC specifies that the cursor makes a temporary copy of the data and all the requests to the cursor are answered from this temporary table. A static cursor can move in both the directions forward and backward. However, data cannot be updated or deleted by using static cursors.

DYNAMIC specifies that the cursor reflects all data changes made to the rows in its result set as you scroll around the cursor.

FAST_FORWARD specifies that the cursor is FORWARD ONLY and READ ONLY. Such cursors produce the least amount of overhead on SQL Server.

READ_ONLY specifies that the cursor is read only and cannot be updated.

SCROLL_LOCKS specifies that the rows will be locked as they are read into the cursor. This is done to ensure that the updates or deletes made through the cursor succeeds.

OPTIMISTIC specifies that the cursor does not lock rows as they are read into the cursor. Therefore, if any user changes the data in the rows being read by the cursor, the updates or deletes made through the cursor will not succeed.

select_statement specifies the SQL statements to define the result set of the cursor.

FOR UPDATE [OF column_name [,...n] specifies the columns that can be updated within the cursor. If OF column_name [,...n] is specified, then only the listed columns can be updated. If OF column_name [,...n] is not specified, then all the columns can be updated.

Opening the Cursor

The OPEN statement is used to open and populate the cursor by executing the T-SQL statements defined in the DECLARE CURSOR statement. The following syntax is used to open a cursor:

```
OPEN [ LOCAL|GLOBAL ] cursor_name
```

where,

cursor_name is the name of the cursor you want to open.

Fetching Rows from the Cursor

The FETCH statement is used to retrieve a row from the cursor, one by one. A row can be fetched from a cursor by using the following syntax:

```
FETCH
 [ [ NEXT | PRIOR | FIRST | LAST
     | ABSOLUTE n | @nvar
     | RELATIVE n | @nvar
   ]
   FROM
 ]
{ { [ GLOBAL ] cursor_name } |
@cursor_variable_name }
[ INTO @variable_name [ ,...n ] ]
```

where,

NEXT is used to return the row immediately following the current row. If FETCH NEXT is the first fetch against a cursor, it returns the first row in the result set. NEXT is the default cursor fetch option.

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

FIRST is used to return the first row in the cursor.

LAST is used to return the last row in the cursor.

ABSOLUTE { n| @nvar} is used to return the nth row in the cursor. If n or @nvar is positive, the rows are counted from top of the result set. If n or @nvar is negative, the rows are counted from bottom of the result set. If n or @nvar is 0, no rows are returned. n must be an integer constant and @nvar must be smallint, tinyint, or int.

RELATIVE { n| @nvar} is used to return the nth row related to the current row. If n or @nvar is positive, the nth row after the current row is returned. If n or @nvar is negative, nth row prior to the current row is returned. If n or @nvar is 0, the current row is returned.

GLOBAL is used to specify the name of the cursor that refers to a global cursor.

cursor_name specifies the name of the open cursor from which the fetch should be made. @cursor_variable_name is used as the name of a cursor variable referencing the open cursor from which the fetch should be made.

INTO @variable_name[,...n] specifies the local variable names used to hold the data returned from the cursor. The number of these variables must be equal to the number of columns specified in the cursor select list. Each variable in the list, from left to right, is associated with the corresponding column in the cursor result set.

Closing the Cursor

A cursor can be closed by using the CLOSE statement. When the cursor is closed, the current result set is released. A cursor once closed can be reopened if required. The following syntax is used to close a cursor:

```
CLOSE cursor_name
```

Deallocating the cursor

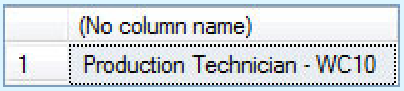
Deallocating removes a cursor reference from the memory. Once deallocated, a cursor cannot be reopened. A cursor is deallocated by using the following syntax:

```
DEALLOCATE cursor_name
```

Consider an example. You want to retrieve the title of an employee on the basis of EmployeeID. You have written the following statements for the same:

```
CREATE PROCEDURE cursor_demo @id int
AS
BEGIN
 DECLARE @Ctitle varchar(50);
 DECLARE cur1 CURSOR FOR SELECT
 HumanResources.Employee.Title FROM
 HumanResources.Employee
 WHERE
 HumanResources.Employee.EmployeeID =
 @id;
 OPEN cur1;
 FETCH cur1 INTO @Ctitle;
 SELECT @Ctitle;
 CLOSE cur1;
 END
 EXEC cursor_demo 10
```

The preceding statements displays the title of the employee whose ID is 10, as shown in the following figure.



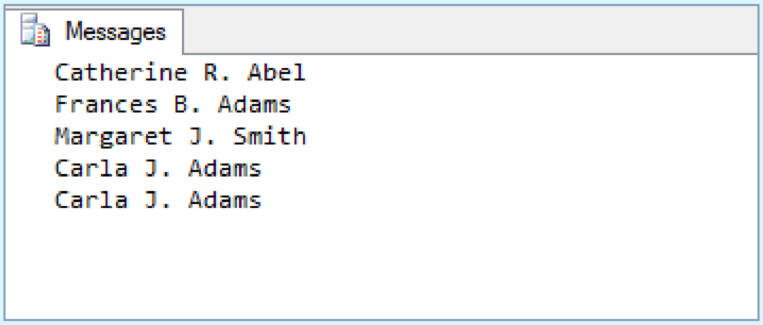
```
   (No column name)
1  Production Technician - WC10
```

The Output Derived After Executing the Statements

Consider another example, where you need to display the first name, middle name, and the last name of a person on the basis of the contact ID. For this, you have written the following statements:

```
DECLARE        @firstname        varchar
(15),@middlename           varchar(15),
@lastname varchar(15)
DECLARE EmpDetail CURSOR
LOCAL SCROLL STATIC
FOR
SELECT firstName, MiddleName, LastName
FROM Person.Contact WHERE ContactID <
10 AND MiddleName IS NOT NULL
OPEN EmpDetail
FETCH NEXT FROM EmpDetail
 INTO    @firstname,    @middlename    ,
 @lastname
 PRINT @firstname + ' ' + @middlename
 +' ' + @lastname
WHILE @@FETCH_STATUS = 0
BEGIN
 FETCH NEXT FROM EmpDetail
 INTO @firstname,@middlename, @lastName
 PRINT @firstname + ' '+@middlename+' '
 + @lastname
END
CLOSE EmpDetail
```

The preceding statements display the names of those employees whose contact ID is less than 10 and the middle name is not null, as shown in the following figure.



```
Catherine R. Abel
Frances B. Adams
Margaret J. Smith
Carla J. Adams
Carla J. Adams
```

The Output Derived After Executing the Statements

Summary

In this chapter, you learned that:

- A batch is a set of SQL statements submitted together to the server for execution.
- You can use a variable to store a temporary value.
- You can use the PRINT statement to display a user-defined message or the content of a variable on the screen.
- You can use the comment entries in batches to write a description of the code.
- You can use the IF…ELSE statement for conditional execution of SQL statements.
- The CASE statement evaluates a list of conditions and returns one of the various possible results.
- You can use the WHILE statement in a batch to allow a set of T SQL statements to execute repeatedly as long as the given condition holds true.
- The BREAK statement causes an exit from the WHILE loop.
- The CONTINUE statement causes the WHILE loop to restart, skipping any statements after the CONTINUE statement within the loop.
- Two ways of handling errors in a batch are:
  - TRY…CATCH
  - RAISERROR
- A stored procedure is a collection of various T-SQL statements that are stored under one name and are executed as a single unit.
- A stored procedure can be created by using the CREATE PROCEDURE statement.
- A stored procedure allows you to declare parameters, variables, and use T-SQL statements and programming logic.
- A stored procedure provides better performance, security, accuracy, and reduces network congestion.
- A stored procedure accepts data through input parameters.
- A stored procedure returns data through output parameters or return statements.
- A stored procedure is executed by using the EXECUTE statement.
- A stored procedure can be altered by using the ALTER PROCEDURE statement.
- A UDF is a database object that contains a set of T-SQL statements.
- UDFs can return either a single scalar value or a result set.
- UDFs are of two types, scalar functions and table-valued functions.
- A scalar function accepts a single value and returns a single value.
- A table-valued function returns a table as an output, which can be derived as part of a SELECT statement.
- At times, your query depends on certain values or parameters, which are not fixed and are provided at the time of execution. Therefore, the complete

text of the SQL statement is not known until execution. Such statements constitute dynamic SQL.

- ❑ Cursor allows working on multiple rows together as a single unit.

Reference Reading

Implementing Batches

Reference Reading: Books	***Reference Reading: URLs***
Microsoft SQL Server 2012 Bible by Adam Jorgensen, Jorge Segarra, and Patrick LeBlanc	*http://msdn.microsoft.com/en-us/library/ms378070.aspx*

Implementing Stored Procedures

Reference Reading: Books	***Reference Reading: URLs***
Beginning SQL Server 2012 for Developers by Robin Dewson	*http://www.mssqltips.com/sqlservertutorial/160/sql-server-stored-procedure/*

Implementing Functions

Reference Reading: Books	***Reference Reading: URLs***
Microsoft SQL Server 2012 Bible by Adam Jorgensen, Jorge Segarra, and Patrick LeBlanc	*http://msdn.microsoft.com/en-us/library/ms174318.aspx*

Chapter 8

Working with Triggers and Transactions

In a relational database, data in a table is related to other tables. Therefore, while manipulating data in one table, you need to verify and validate its effect on the data in the related tables. In addition, after inserting or updating data in a table, you might need to manipulate data in another table. You also need to ensure that if an error occurs while updating the data in a table, the changes are reverted. This helps in maintaining data integrity. SQL Server allows you to implement triggers and transactions to maintain data integrity.

This chapter explains the various types of triggers that can be created in SQL Server. In addition, it discusses how to implement triggers to enforce data integrity. Further, it explains how to implement transactions.

Objectives

In this chapter, you will learn to:

- ❑ Implement triggers
- ❑ Implement transactions

Implementing Triggers

At times, while performing data manipulation on a database object, you might also need to perform manipulation on another object. For example, in an organization, the employees use the Online Leave Approval system to apply for leaves. When an employee applies for a leave, the leave details are stored in the LeaveDetails table. In addition, a new record is added to the LeavesForApproval table. When the supervisors log on to the system, all the leaves pending for their approval are retrieved from the LeavesForApproval table and displayed to them.

To perform such operations, SQL Server allows you to implement triggers. A trigger consists of a set of T-SQL statements activated in response to certain actions, such as insert or delete. Triggers are used to ensure data integrity before or after performing data manipulations. Therefore, a trigger is a special kind of stored procedure that executes in response to specific events.

Triggers are also used to enhance an application's performance. Consider a scenario where a client needs to send 10 SQL statements to SQL Server. Suppose out of these 10 SQL statements, if five SQL statements are fired by a trigger, the client saves at least 10 communication messages between the client and the server. One is to send the SQL statement to the server. The other is to reply to the message from the server after executing the SQL statement. This reduction in the number of messages, which are passing between the client and the server, minimizes the load on the network traffic. This improves the performance of the server when the network traffic is slow and there are more SQL statements to be executed.

Before you implement a trigger, it is important to learn about the various types of triggers that can be created by using SQL Server.

Identifying Types of Triggers

In SQL Server, various kinds of triggers can be used for different types of data manipulation operations. Triggers can be nested and fired recursively. SQL Server supports the following types of triggers:

- ❑ Data Modification Language (DML) triggers
- ❑ Data Definition Language (DDL) triggers

DML Triggers

A DML trigger is fired when data in the underlying table is affected by DML statements, such as INSERT, UPDATE, or DELETE. These triggers help in maintaining consistent, reliable, and correct data in tables. They enable the process of reflecting the changes made in a table to other related tables.

The DML triggers have the following characteristics:

- ❑ Fired automatically by SQL Server whenever any data modification statement is issued.
- ❑ Cannot be explicitly invoked or executed, as in the case of the stored procedures.
- ❑ Prevent incorrect, unauthorized, and inconsistent changes in data.
- ❑ Do not return data to the user.
- ❑ Can be nested up to 32 levels. The nesting of triggers occurs when a trigger performs an action that initiates another trigger.

Whenever a trigger is fired in response to the INSERT, DELETE, or UPDATE statement, SQL Server creates two temporary tables, called *magic tables*. The magic tables are called *Inserted* and *Deleted*. These are logical tables and are similar in structure to the table on which the trigger is defined.

The *Inserted table* contains a copy of all the records that are inserted in the trigger table. A trigger table refers to the table on which the trigger is defined. The Deletedtable contains all the records that have been deleted from the trigger table. Whenever you update data in a table, the trigger uses both theInserted table and the Deletedtable.

Depending on the operation performed, DML triggers can be further categorized as:

- ❑ **INSERT trigger**: Is fired whenever an attempt is made to insert a row in the trigger table. When an

INSERT statement is executed, the newly inserted rows are added to the Inserted table.

- **DELETE trigger**: Is fired whenever an attempt is made to delete a row from the trigger table. When a DELETE statement is executed, the deleted rows are added to the Deleted table. The Deleted table and the trigger table do not have any rows in common, as in the case of the Insertedtable and the trigger table.

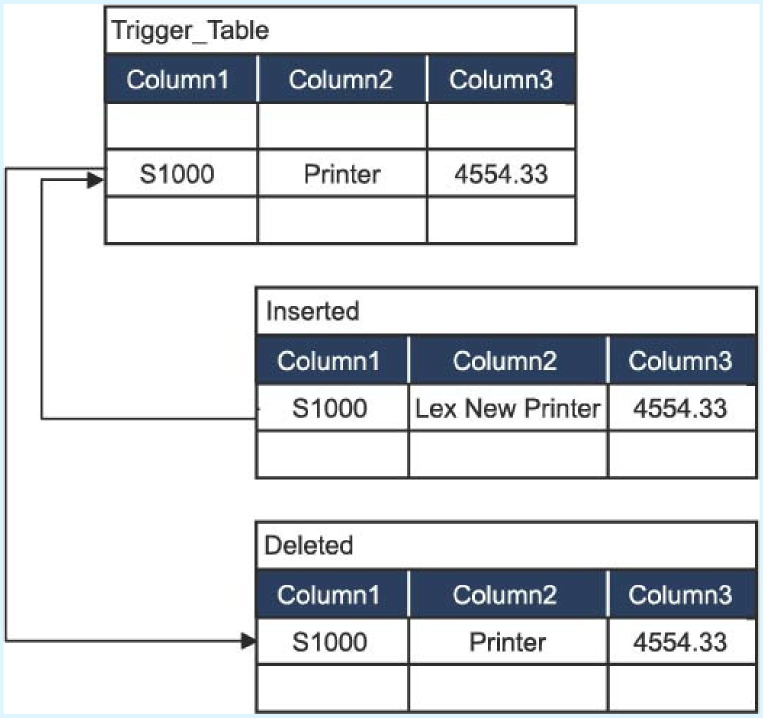
  There are three ways of implementing referential integrity by using a DELETE trigger:

  - **The cascade method**: Deletes the corresponding records from the dependent tables whenever a record is deleted from the master table.
  - **The restrict method**: Restricts the deletion of records from the master table if the related records are present in the dependent tables.
  - **The nullify method**: Nullifies the values in the specified columns of the dependent tables whenever a record is deleted from the master table.

- **UPDATE trigger**: Is fired when an UPDATE statement is executed in the trigger table. It uses the two logical tables for its operations. The Deleted table contains the original rows (the rows with the values before updating) and the Inserted table stores the new rows (the modified rows). After all the rows are updated, the Deleted table and the Inserted table are populated and the trigger is fired.

  For example, you have a table with three columns. It stores the details of hardware devices. You updated a value in Column2 from 'Printer' to 'Lex New Printer'. During the update process, the Deleted table holds the original row (the row with the values before updating) with the value 'Printer' in Column2. The Insertedtable stores the new row (the modified row) with the value 'Lex New Printer' in Column2.

The following figure illustrates the functioning of the UPDATE trigger.



Trigger_Table

Column1	Column2	Column3
S1000	Printer	4554.33

Inserted

Column1	Column2	Column3
S1000	Lex New Printer	4554.33

Deleted

Column1	Column2	Column3
S1000	Printer	4554.33

The Functioning of the UPDATE Trigger

When a DML trigger is fired, one or more rows of data can be affected. This is a common behavior of the UPDATE and DELETE triggers because these statements are frequently used for affecting multiple rows. This behavior is less common for the INSERT trigger because usually, only one record is inserted in a table. However, if the INSERT INTO (table_name) SELECT statement is used, multiple rows can be inserted by using a single trigger.

Depending on the way the triggers are fired, they can be further categorized as:

- AFTER triggers
- INSTEAD OF triggers

AFTER Triggers

The AFTER trigger can be created on any table for the insert, update or delete operation just like other triggers. The main difference in the functionality of an AFTER trigger is that it is fired after the execution of the DML operation for which it has been defined. The AFTER trigger is executed when all the constraints and triggers defined on the table are successfully executed.

If more than one AFTER trigger is created on a table, they are executed in the order of creation.

For example, the EmpSalary table stores the salary and tax details for all the employees in an organization. You need to ensure that after the salary details of an employee are updated in the EmpSalary table, the tax details should also be recalculated and updated. In such a scenario, you can implement an AFTER trigger to update the tax details when the salary details are updated.

You can have multiple AFTER triggers for a single DML operation.

INSTEAD OF Triggers

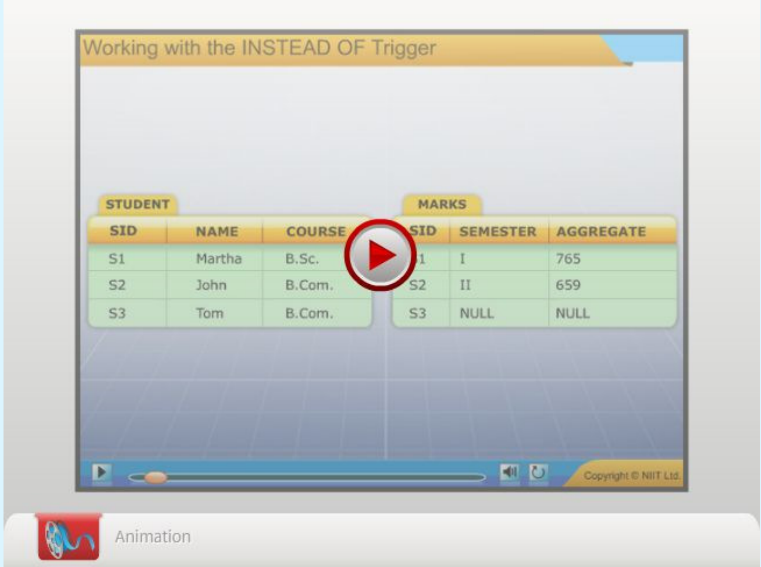
The INSTEAD OF triggers can be primarily used to perform an action, such as a DML operation on another table or view. This type of trigger can be created on both a table as well as a view. For example, if a view is created with multiple columns from two or more tables, then an insert operation on the view is only possible if the primary key fields from all the base tables are used in the query. Alternatively, if you use an INSTEAD OF trigger, you can insert data in the base tables individually. This makes the view logically updateable.

An INSTEAD OF trigger can be used for the following actions:

- ❑ Ignoring parts of a batch
- ❑ Not processing a part of a batch and logging the problem rows
- ❑ Taking an alternative action when an error condition is encountered

You can even create an INSTEAD OF trigger to restrict deletion in a master table. For example, you can display a message "Master records cannot be deleted" if a delete statement is executed on the Employee table of the AdventureWorks database.

Unlike AFTER triggers, you cannot create more than one INSTEAD OF trigger for a DML operation on the same table or view.



Working with the INSTEAD OF Trigger

STUDENT

SID	NAME	COURSE
S1	Martha	B.Sc.
S2	John	B.Com.
S3	Tom	B.Com.

MARKS

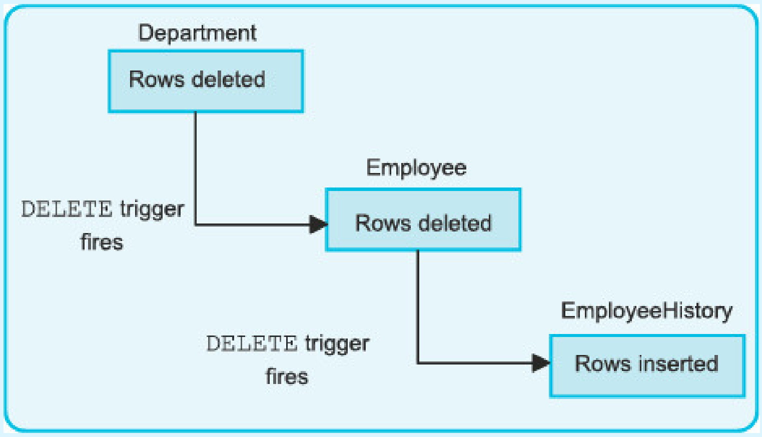
SID	SEMESTER	AGGREGATE
1	I	765
S2	II	659
S3	NULL	NULL

DDL Triggers

DDL triggers are fired in response to DDL statements, such as CREATE TABLE or ALTER TABLE. They can be used to perform administrative tasks, such as database auditing. Database auditing helps in monitoring DDL operations on a database. DDL operations can include operations such as creation of a table or view, or modification of a table or procedure. Consider an example, where you want the database administrator to be notified whenever a table is created in the master database. For this purpose, you can create a DDL trigger.

Nested Triggers

Nested triggers are fired due to actions of other triggers. Both DML and DDL triggers can be nested when a trigger performs an action that initiates another trigger. For example, a DELETE trigger on the Department table deletes the corresponding employee records in the Employee table and a DELETE trigger on the Employee table inserts these deleted employee records in the EmployeeHistory table. Therefore, the DELETE trigger on the Department table initiates the DELETE trigger on the Employee table, as shown in the following figure.



A Nested Trigger

Recursive Triggers

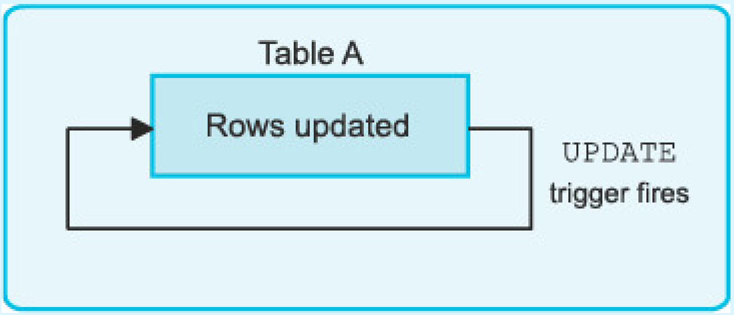
Recursive triggers are a special case of nested triggers. Unlike nested triggers, support for recursive triggers is at the database level. As the name implies, a recursive trigger eventually calls itself. The types of recursive triggers are:

- ❑ Direct
- ❑ Indirect

Direct Recursive Triggers

When a trigger fires and performs an action that causes the same trigger to fire again, the trigger is called a direct recursive trigger. For example, an application updates the Table A, which causes trigger T1 to fire. T1 updates the Table A again, which causes the trigger T1 to fire again.

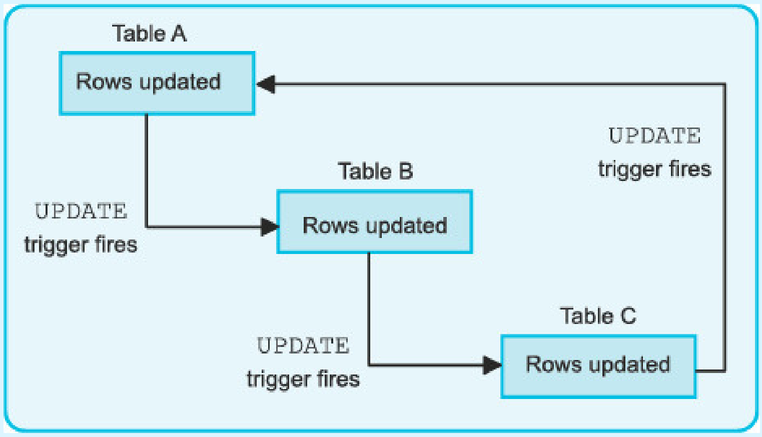
The following figure depicts the execution of a direct recursive trigger.



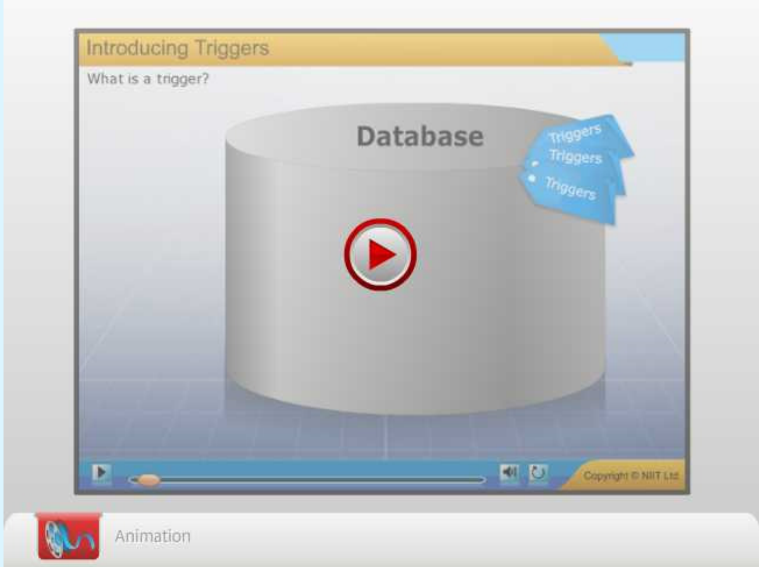
A Direct Recursive Trigger

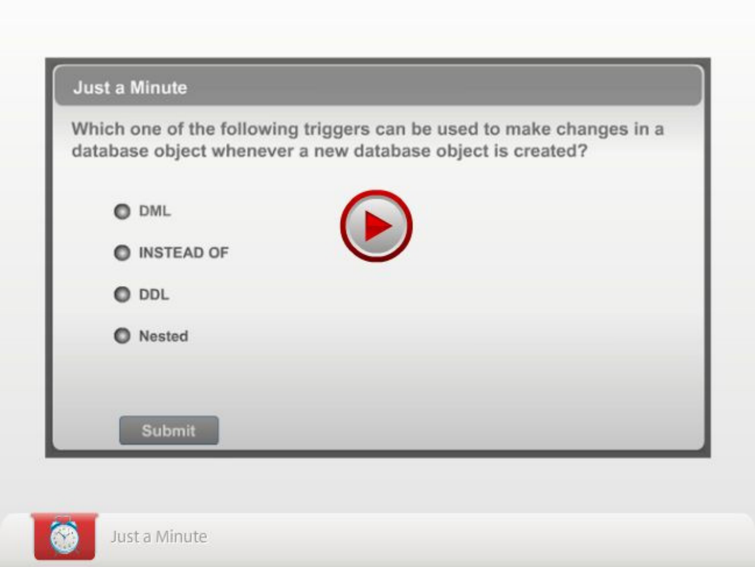
Indirect Recursive Triggers

An indirect recursive trigger fires a trigger on another table and eventually the nested trigger ends up firing the first trigger again. For instance, an UPDATE statement on Table A fires a trigger that, in turn, fires an update on Table B. The update on Table B fires another trigger that performs an update on Table C. Table C has a trigger that causes an update on Table A again. As a result, the UPDATE trigger of Table A is fired again. The following figure depicts the execution of an indirect recursive trigger.



An Indirect Recursive Trigger





Just a Minute

Which one of the following triggers can be used to make changes in a database object whenever a new database object is created?

○ DML

○ INSTEAD OF

○ DDL

○ Nested

Creating Triggers

You can use the CREATE TRIGGER statement to create triggers. The syntax of the CREATE TRIGGER statement is:

```
CREATE TRIGGER trigger_name
ON { table | view }
{ FOR | AFTER | INSTEAD OF }
[WITH [ENCRYPTION] [EXECUTE AS] ]
{ AS
{ sql_statement [ ...n ] }
}
```

where,

`trigger_name` specifies the name of the trigger to be created.

`table | view` specifies the name of the table or view on which the trigger is to be created.

`FOR | AFTER | INSTEAD OF` specifies the precedence and execution context of a trigger.

`WITH ENCRYPTION` encrypts the text of the CREATE TRIGGER statement.

`EXECUTE AS` specifies the security context under which the trigger is executed.

`AS sql_statement` specifies the trigger conditions and actions. A trigger can contain any number of T-SQL statements, provided these are enclosed within the BEGIN and END keywords.

For example, the following statement creates a trigger on the HumanResources.Department table of the AdventureWorks database:

```
CREATE TRIGGER [HumanResources].
[trgDepartment] ON [HumanResources].
[Department]
AFTER UPDATE AS
BEGIN
 UPDATE [HumanResources].[Department]
 SET [HumanResources].[Department].
 [ModifiedDate] = GETDATE()
 FROM Inserted
 WHERE Inserted.[DepartmentID] =
 [HumanResources].[Department].
 [DepartmentID];
END;
```

The preceding statement creates a trigger named trgDepartment. This trigger is fired on every successfully executed UPDATE statement on the HumanResources.Department table. The trigger updates the ModifiedDate column of every updated value with the current date.

The following statement creates a trigger to display the data that is inserted in the magic tables:

```
 CREATE TRIGGER trgMagic ON
 EmpDeptHistory
```

```
AFTER UPDATE AS
BEGIN
SELECT * FROM Deleted
SELECT * FROM Inserted
END;
```

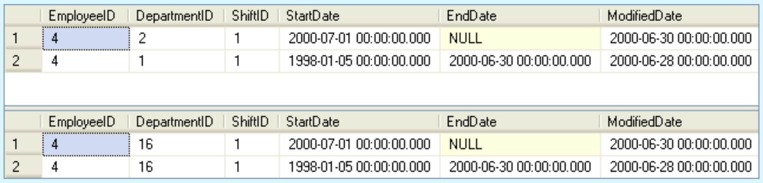
The preceding statement creates an AFTER UPDATE trigger on the EmpDeptHistory table. Whenever an UPDATE statement is fired on the EmpDeptHistory table, the trgMagic trigger is executed and displays the previous value in the table as well as the updated value.

*Use the following statement to create the EmpDeptHistory table: SELECT * INTO EmpDeptHistory FROM HumanResources.EmployeeDepartmentHistory*

For example, you can execute the following UPDATE statement on the EmpDeptHistory table:

```
UPDATE EmpDeptHistory SET DepartmentID
= 16
WHERE EmployeeID = 4
```

When the preceding statement is executed, the trgMagic trigger is fired displaying the output, as shown in the following figure.



	EmployeeID	DepartmentID	ShiftID	StartDate	EndDate	ModifiedDate
1	4	2	1	2000-07-01 00:00:00.000	NULL	2000-06-30 00:00:00.000
2	4	1	1	1998-01-05 00:00:00.000	2000-06-30 00:00:00.000	2000-06-28 00:00:00.000

	EmployeeID	DepartmentID	ShiftID	StartDate	EndDate	ModifiedDate
1	4	16	1	2000-07-01 00:00:00.000	NULL	2000-06-30 00:00:00.000
2	4	16	1	1998-01-05 00:00:00.000	2000-06-30 00:00:00.000	2000-06-28 00:00:00.000

The Output Derived by Using the UPDATE Statement

In the preceeding figure, the result set on the top shows the values before the execution of the UPDATE statement and the result set at the bottom shows the updated values.

Creating an INSERT Trigger

An INSERT trigger gets fired at the time of adding new rows in the trigger table. For example, users at AdventureWorks, Inc. want the modified date to be set to the current date whenever a new record is entered in the Shift table. To perform this task, you can use the following statement:

```
CREATE TRIGGER
HumanResources.trgInsertShift
ON HumanResources.Shift
FOR INSERT
AS
 DECLARE @ModifiedDate datetime
 SELECT @ModifiedDate = ModifiedDate
 FROM Inserted
 IF (@ModifiedDate != getdate())
 BEGIN
   PRINT 'The modified date should be
 the current date.
   Hence, cannot insert.'
   ROLLBACK TRANSACTION
 END
RETURN
```

The ROLLBACK TRANSACTION statement is used to roll back transactions. The ROLLBACK TRANSACTION statement in the trgInsertShift trigger is used to undo the insert operation.

Creating a DELETE Trigger

A DELETE trigger gets fired at the time of deleting rows from the trigger table. For example, the following statement creates a trigger to disable the deletion of rows from the Department table:

```
CREATE TRIGGER trgDeleteDepartment
ON HumanResources.Department
FOR DELETE
AS
 PRINT 'Deletion of Department is not
 allowed'
 ROLLBACK TRANSACTION
RETURN
```

Creating an UPDATE Trigger

An UPDATE trigger gets fired at the time of updating records in the trigger table. For example, you need to create a trigger to ensure that the average of the values in the Rate column of the EmployeePayHistory table should not be more than 20 when the value of Rate is increased. To perform this task, you can use the following statement:

```
CREATE TRIGGER
trgUpdateEmployeePayHistory
ON HumanResources.EmployeePayHistory
FOR UPDATE
AS
IF UPDATE (Rate)
BEGIN
DECLARE @AvgRate float
SELECT @AvgRate = AVG(Rate)
FROM HumanResources.EmployeePayHistory
IF(@AvgRate > 20)
BEGIN
 PRINT 'The average value of rate
 cannot be more than 20'
   ROLLBACK TRANSACTION
 END
END
```

Creating an AFTER Trigger

An AFTER trigger gets fired after the execution of a DML statement. For example, you need to display a message after a record is deleted from the Employee table. To perform this task, you can write the following statement:

```
CREATE TRIGGER trgDeleteShift ON
HumanResources.Shift
AFTER DELETE
```

```
AS
PRINT 'Deletion successful'
```

If there are multiple AFTER triggers for a single DML operation, you can change the sequence of execution of these triggers by using the sp_settriggerorder system stored procedure. The syntax of the sp_settriggerorder stored procedure is:

```
sp_settriggerorder <triggername>,
<order-value>, <DML-operation>
```

where,

`triggername` specifies the name of the trigger whose order of execution needs to be changed.

`order-value` specifies the order in which the trigger needs to be executed. The values that can be entered are FIRST, LAST, and NONE. If FIRST is mentioned, then the trigger is the first trigger to be executed. If LAST is mentioned, then the trigger will be the last trigger to be executed. If NONE is specified, then the trigger is executed on a random basis.

`DML-operation` specifies the DML operation for which the trigger was created. This should match the DML operation associated with the trigger. For example, if UPDATE is specified for a trigger that is created for the INSERT operation, the sp_settriggerorder stored procedure will generate an error.

For example, you have created another AFTER trigger, trgDeleteShift1 on the Shift table, as shown in the following statement:

```
CREATE TRIGGER trgDeleteShift1 ON
HumanResources.Shift
AFTER DELETE
AS
PRINT 'Attempting to Delete'
```

By default, triggers are executed in the sequence of their creation. However, if you need to execute the trigger named trgDeleteShift1 before the trgDeleteShift trigger, you can execute the following statement:

```
sp_settriggerorder
'HumanResources.trgDeleteShift1',
'FIRST', 'DELETE'
RETURN
```

Creating an INSTEAD OF Trigger

INSTEAD OF triggers are executed in place of the events that cause the trigger to fire. For example, if you create an INSTEAD OF UPDATE trigger on a table, the statements specified in the trigger will be executed instead of the UPDATE statement that caused the trigger to fire.

These triggers are executed after the inserted table and the deleted table reflecting the changes to the base table are created, but before any other action is taken. They are executed before any constraint, and therefore, supplement the action performed by a constraint. You can create the following INSTEAD OF trigger to restrict the deletion of records in the Customer table:

```
CREATE TRIGGER trgDelete ON
Sales.Customer
INSTEAD OF DELETE
AS
PRINT 'Customer records cannot be
deleted'
```

Creating a DDL Trigger

DDL triggers are special kind of triggers that fire in response to the DDL statements. They can be used to perform administrative tasks in the database such as auditing and regulating database operations. The following statement creates a DDL trigger:

```
CREATE TRIGGER safety
ON DATABASE
FOR DROP_TABLE, ALTER_TABLE
AS
 PRINT 'You must disable Trigger
 "safety" to drop or alter tables!'
 ROLLBACK
```

In the preceding statement, the safety trigger will fire whenever a DROP_TABLE or ALTER_TABLE event occurs in the database.

While creating DDL triggers, you need to use DDL events that can be used to fire a DDL trigger. Some of the DDL events that can be used with DDL triggers are:

- CREATE_FUNCTION
- ALTER_FUNCTION
- DROP_FUNCTION
- CREATE_INDEX
- ALTER_INDEX
- DROP_INDEX
- CREATE_PROCEDURE
- ALTER_PROCEDURE
- DROP_PROCEDURE
- CREATE_SYNONYM
- DROP_SYNONYM
- CREATE_TABLE
- ALTER_TABLE
- DROP_TABLE
- CREATE_TRIGGER
- ALTER_TRIGGER
- DROP_TRIGGER

The events in the preceding list correspond to the SQL statement, the syntax of which is modified to include underscores ('_') between keywords.

Managing Triggers

While managing triggers, you can perform the following operations on a trigger:

- Alter a trigger
- Delete a trigger

Altering a Trigger

As a database developer, you might need to modify the logic or code behind a trigger. For example, a trigger is used to calculate a 10 percent discount on every item sold. With the new management policy, the discount rate has been increased to 15 percent. To reflect this change in the trigger, you need to change the code in the trigger. You can use the ALTER TRIGGER statement to modify the trigger. The syntax of the ALTER TRIGGER statement is:

```
ALTER TRIGGER trigger_name
{ FOR | AFTER | INSTEAD OF}
{ event_type [ ,...n ] |
DDL_DATABASE_LEVEL_EVENTS }
{ AS
 { sql_statement [ ...n ] }
}
```

For example, you can modify the trgDeleteShift trigger that was created earlier to restrict the deletion of records in the Shift table. To modify the trgDeleteShift trigger, you need to execute the following statement:

```
ALTER TRIGGER
HumanResources.trgDeleteShift ON
HumanResources.Shift
INSTEAD OF DELETE
AS
 PRINT 'Deletion of Shift details is
 not allowed'
 ROLLBACK TRANSACTION
RETURN
```

Apart from altering a DML trigger, you can modify a DDL trigger by executing the following ALTER statement:

```
ALTER TRIGGER safety
ON DATABASE
FOR DROP_TABLE
AS
PRINT 'YOU CAN ALTER BUT NOT DELETE
TABLE'
```

After executing the preceding statement, the safety trigger is modified for dropping a table. The safety trigger is fired whenever you try to drop any table in the database.

Deleting a Trigger

As the requirements change, you may need to delete some triggers. For example, you have a trigger that updates the salaries of the employees in their pay slips. Now, this function is performed by a front-end application in your organization. Therefore, you need to remove the trigger.

To delete a trigger, you can use the DROP TRIGGER statement. The syntax of the DROP TRIGGER statement is:

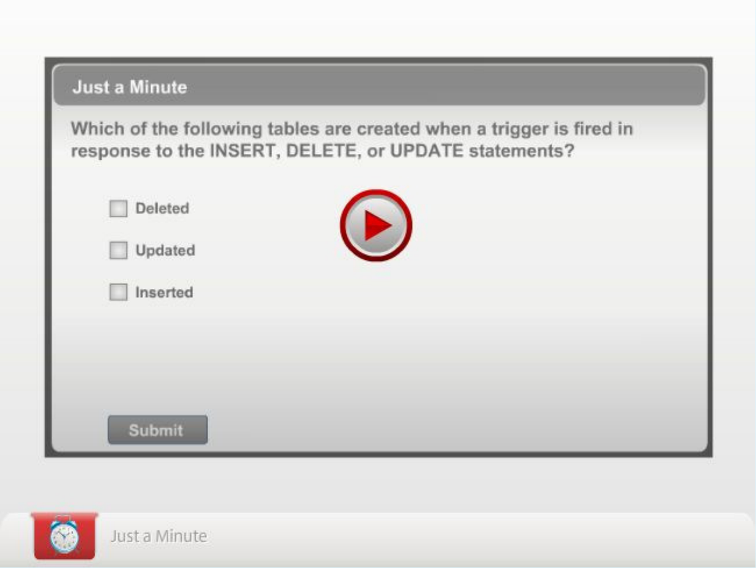
```
DROP TRIGGER { trigger }
```

where,

`trigger` is the name of the trigger you want to drop.

The following statement drops the trgMagic trigger:

```
DROP TRIGGER trgMagic
```



Just a Minute

Which of the following tables are created when a trigger is fired in response to the INSERT, DELETE, or UPDATE statements?

□ Deleted

□ Updated

□ Inserted

Disabling a Trigger

Sometimes, you need to prevent the execution of a trigger to perform a specific operation on a table. For example, a trigger is created on the Employee table that prevents the modification of the employee data. However, one of the employees has shifted to another department. This requires you to make the necessary changes in the Employee table. You can perform this task by disabling the trigger.

Once the trigger is disabled, it will not execute until it is enabled again. Disabling a trigger does not delete the trigger. It remains in the database, but does not execute in response to an event.

SQL Server provides the DISABLE TRIGGER statement to disable an existing trigger. The syntax of the DISABLE TRIGGER statement is:

```
DISABLE TRIGGER { [ schema_name . ]
trigger_name | ALL }
ON { object_name | DATABASE }
```

where,

`schema_name` specifies the name of schema to which the trigger belongs.

`trigger_name` specifies the name of the trigger to be disabled.

`All` specifies that all triggers defined at the scope of the ON clause are disabled.

`object_name` specifies the name of the table, or view on which the trigger was defined.

`DATABASE` is used for a DDL trigger.

For example, you can disable the safety trigger by executing the following statement:

```
DISABLE TRIGGER safety ON DATABASE
```

Enabling a Trigger

You disable the triggers when you do not want them to be executed. Later, if you want them to be executed in response to their respective events, you can again enable them.

SQL Server provides the ENABLE TRIGGER statement to enable a trigger. The syntax of the ENABLE TRIGGER statement is:

```
ENABLE TRIGGER { [ schema_name . ]
trigger_name | ALL }
ON { object_name | DATABASE }
```

For example, you can enable the safety trigger by executing the following statement:

```
ENABLE TRIGGER safety ON DATABASE
```

Displaying Events of a Trigger

Sometimes, you want to know the information about the events that caused the trigger to be executed. For example, a new payroll software is being installed in an organization. James, a database developer, wants to understand the behavior of the new software under the predefined data. He has created a trigger that is fired when the database is modified. While executing the trigger, he wants to get the detailed information, such as the events that caused the trigger to be fired and the SQL statement that was executed to fire the trigger.

SQL Server provides the EVENTDATA() function that can be embedded with the CREATE TRIGGER statement to provide the information, such as time and type of the event that caused the trigger to fire. The EVENTDATA() function returns the details of the trigger in the XML format. This XML data contains <EVENT_INSTANCE> as the parent tag. The sample format of the XML data returned by the EVENTDATA() function is:

```
<EVENT_INSTANCE>
 <EventType> </EventType>
 <PostTime> </PostTime>
 <SPID> </SPID>
 <ServerName> </ServerName>
 <LoginName> </LoginName>
 <UserName> </UserName>
 <DatabaseName> </DatabaseName>
 <SchemaName> </SchemaName>
 <ObjectName> </ObjectName>
 <TSQLCommand>
 <CommandText> </CommandText>
 </TSQLCommand>
</EVENT_INSTANCE>
```

To retrieve the data stored in the preceding XML format, you need to use the functions of the XQuery language.

For example, you can create a trigger on the AdventureWorks database with the EVENTDATA() function by executing the following statement:

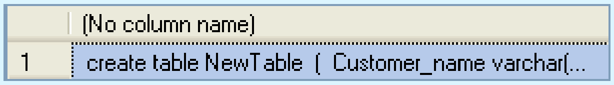
```
CREATE TRIGGER Info_Event
ON DATABASE
FOR CREATE_TABLE
AS
 PRINT 'CREATING TABLE'
 RAISERROR ('New tables cannot be
 created in this database.', 16, 1)
 SELECT EVENTDATA().value('(/
 EVENT_INSTANCE/TSQLCommand/
 CommandText)[1]','nvarchar(max)')
 ROLLBACK
```

In the preceding code, the value() function is used to display the data of the CommandText tag. This tag stores the actual query that caused the trigger to fire.

You can display the event information after executing the following CREATE TABLE statement:

```
CREATE TABLE NewTable
(
Customer_name varchar(30),
Salary int
)
```

After executing the preceding statement, the Info_Event trigger is fired and the output is displayed, as shown in the following figure.



	(No column name)
1	create table NewTable (Customer_name varchar(...

The Output Derived After Info_Event Trigger Fires

Using the UPDATE() Function

While performing the UPDATE operation on a database object, you may need to perform specific actions on another database object. To achieve this functionality, you can use the UPDATE trigger on a database object. The scope of the UPDATE trigger is tables. In other words, the UPDATE trigger is fired on updating any column of a table on which the trigger is defined. However, at times, you require that the UPDATE trigger should be fired when a specific column in a table is updated.

For example, in an organization, the employees use the Online Leave Approval system to apply for leaves. When an employee applies for a leave, the leave details are stored in the LeaveDetails table. In addition, a new record is added to the LeavesForApproval table. When the supervisors log on to the system, all the leaves pending for their approval are retrieved from the LeavesForApproval table and displayed to them. The supervisor approves or denies a leave by updating the LeaveStatus column of the LeavesForApproval table. Therefore, you want that after a supervisor updates the value of the LeaveStatus column in the LeavesForApproval table, the LeaveStatus column in the LeaveDetails table should also be updated.

To achieve the functionality required in the preceding scenario, you can use the UPDATE() function with an UPDATE trigger. SQL Server provides the UPDATE() function that can be used to create an UPDATE trigger applied to specific columns. The syntax of the UPDATE() function is:

```
UPDATE (column)
```

where,

`column` specifies the name of the column on which an

UPDATE trigger is applied.

For example, you want that whenever an attempt is made to update the VacationHours or Title columns of the Employee table, a message should be displayed and the updates should be rolled back. For this, you can create an AFTER UPDATE trigger with the UPDATE() function. To accomplish this task, you can use the following statement:

```
USE AdventureWorks
GO
CREATE TRIGGER reminder
ON HumanResources.Employee
AFTER UPDATE
AS
IF  (UPDATE  (VacationHours)  OR  UPDATE
(Title))
BEGIN
Print 'You cannot update the specified
columns'
ROLLBACK TRAN
END
GO
```

In the preceding statement, the VacationHours and Title columns are specified in the UPDATE() function to fire the trigger whenever an attempt is made to update these columns.

After creating the trigger, you can use the following statement to verify the trigger:

```
UPDATE HumanResources.Employee
SET VacationHours=25
WHERE Employee.EmployeeID=1
```

When the preceding statement is executed, the reminder trigger is fired to prevent the updation of the VacationHours column.

Activity 8.1: Implementing Triggers

Implementing Transactions

At times, you are required to execute a sequence of statements as a single logical unit of work. For example, whenever you transfer money from one bank account to another account, the amount is debited from one account and credited to another account. In such a situation, you need either all the statements to be executed successfully or none of them to be executed. This helps in ensuring data integrity.

In SQL Server, you can implement transactions to ensure data integrity. In a multi user environment, there can be multiple transactions accessing the same resource at the same time. To prevent errors that could occur due to transactions accessing the same resource, you can use locks. Locks provide a mechanism to secure a resource until one transaction is executed so that only one transaction can work on a database resource at a time.

Creating Transactions

A transaction can be defined as a sequence of operations performed together as a single logical unit of work. A single unit of work must possess the following properties called *ACID* (*Atomicity*, *Consistency*, *Isolation*, and *Durability*).

- **Atomicity**: This states that either all the data modifications are performed or none of them are performed.
- **Consistency**: This states that all data is in a consistent state after a transaction is completed successfully. All rules in a relational database must be applied to the modifications in a transaction to maintain complete data integrity.
- **Isolation**: This states that any data modification made by concurrent transactions must be isolated from the modifications made by other concurrent transactions. In simpler words, a transaction either accesses data in the state in which it was before a concurrent transaction modified it or accesses the data after the second transaction has been completed. There is no scope for the transaction to see an intermediate state.
- **Durability**: This states that any change in data by a completed transaction remains permanently in effect in the system. Therefore, any change in data due to a completed transaction persists even in the event of a system failure. This is ensured by the concept of backing up and restoring transaction logs.

It is important that a database system provides mechanisms to ensure the physical integrity of each transaction. To fulfill the requirements of the ACID properties, SQL Server provides the following features:

- **Transaction management**: Ensures the atomicity and consistency of all transactions. A transaction must be successfully completed after it has started or SQL Server undoes all the data modifications made since the start of the transaction.
- **Locking**: Preserves transaction durability and isolation.

SQL Server allows implementing transactions in the following ways:

- Autocommit transaction
- Implicit transaction
- Explicit transaction

Autocommit Transaction

The autocommit transaction is the default transaction management mode of SQL Server. Based on the completeness of every T-SQL statement, transactions are automatically committed or rolled back. A statement is committed if it is completed successfully, and it is rolled back if it encounters an error.

Implicit Transaction

An implicit transaction is the one where you do not need to define the start of the transaction. You are only required to commit or roll back the transaction. You also need to turn on the implicit transaction mode to specify the implicit transaction. After you have turned on the implicit transaction mode, SQL Server starts the transaction when it executes any of the statements listed in the following table.

ALTER TABLE	*INSERT*	*CREATE*
OPEN	*DELETE*	*DROP*
REVOKE	*SELECT*	*FETCH*
TRUNCATE TABLE	*GRANT*	*UPDATE*

The Implicit Transaction Statements

The transaction remains in effect until you issue a COMMIT or ROLLBACK statement. After the first transaction is committed or rolled back, SQL Server starts a new transaction the next time any of the preceding statements is executed. SQL Server keeps generating a chain of implicit transactions until you turn off the implicit transaction mode.

You can turn on the implicit transaction mode by using the following statement:

```
SET IMPLICIT_TRANSACTIONS ON;
```

For example, consider the following statements:

```
SET IMPLICIT_TRANSACTIONS ON;
Insert  into  Depositor  (Customer_name,
Acc_num)  values('Nora',101),  ('Robin',
4101),
('James', 501), ('Jennifer', 7101)

-- Commit first transaction
COMMIT TRANSACTION;
--  Second  implicit  transaction  started
by a SELECT statement
SELECT COUNT(*) FROM Depositor
INSERT  INTO  Depositor  VALUES  ('Peter',
9200)
SELECT * FROM Depositor;
-- Commit second transaction
COMMIT TRANSACTION;
```

In the preceding statements, the implicit transaction mode is turned on. After the first transaction is committed, the second implicit transaction is started as soon as the SELECT statement is executed.

You can turn off the implicit transaction mode by using the following statement:

```
SET IMPLICIT_TRANSACTIONS OFF;
```

Explicit Transaction

An explicit transaction is the one where both the start and the end of the transaction are defined explicitly. Explicit transactions were called user-defined or user-specified transactions in earlier versions of SQL Server. They are specified by using the following statements:

- □ **BEGIN TRANSACTION**: Is used to set the starting point of a transaction.
- □ **COMMIT TRANSACTION**: Is used to save the changes permanently in the database.
- □ **ROLLBACK TRANSACTION**: Is used to undo the changes.
- □ **SAVE TRANSACTION**: Is used to establish save points that allow partial rollback of a transaction.

> NOTE *You can use the BEGIN TRANSACTION statement to override the default autocommit mode. SQL Server returns to the autocommit mode when the explicit transaction is committed or rolled back.*

Beginning a Transaction

The BEGIN TRANSACTION statement marks the start of a transaction. The syntax of the BEGIN TRANSACTION statement is:

```
BEGIN  TRAN[SACTION]  [transaction_name
| @tran_name_variable]
```

where,

`transaction_name` is the name assigned to the transaction. This parameter must conform to the rules for identifiers and should not be more than 32 characters.

`@tran_name_variable` is the name of a user-defined variable that contains a valid transaction name. This variable must be declared with a char, varchar, nchar, or nvarchar data type.

Commiting a Transaction

The COMMIT TRANSACTION or COMMIT WORK statement marks the end of an explicit transaction. This statement is used to end a transaction for which no errors were encountered during the transaction. The syntax of the COMMIT TRANSACTION statement is:

```
COMMIT          [          TRAN[SACTION]
[transaction_name                       |
@tran_name_variable] ]
```

where,

`transaction_name` specifies a transaction name assigned by a previous BEGIN TRANSACTION statement. It can be used to indicate the nested BEGIN TRANSACTION statement that is associated with the COMMIT TRANSACTION statement.

`@tran_name_variable` is the name of a user-defined variable that contains a valid transaction name. This variable must be declared with a char, varchar, nchar, or nvarchar data type.

Consider a scenario where the salary of an employee named Robert is changed to $15000 and the salary of an employee named Jack is changed to $18000. To perform these transactions, you need to execute the following statements:

```
UPDATE EmployeeDetails
SET Salary = 15000
WHERE EmpName = 'Robert'
UPDATE EmployeeDetails
SET Salary = 18000
WHERE EmpName = 'Jack'
```

When the preceding statements are executed, either both should be executed successfully or none of them should be executed. If any of the statements fails to execute, the entire transaction should be rolled back. Therefore, you need to define the beginning and end of a transaction, as shown in the following statements:

```
BEGIN TRAN myTran
UPDATE EmployeeDetails
SET Salary = 10000
WHERE EmpName = 'Robert'
UPDATE EmployeeDetails
SET Salary = 12000
WHERE EmpName = 'Jack'
COMMIT TRAN myTran
```

The preceding statements create a transaction named myTran, which updates the salaries of the employees in the EmployeeDetails table.

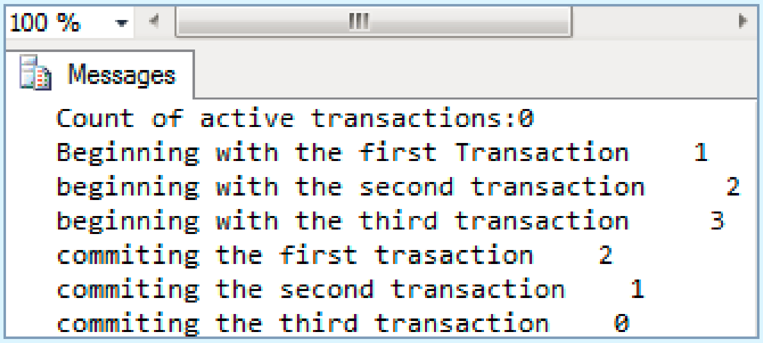
Determining the Count of Local Transactions

In SQL Server, the @@TRANCOUNT function is used to count the number of local transactions executing on the current connection. SQL Server allows transactions to be nested. The @@TRANCOUNT function returns the current nesting level of these transactions. The outermost BEGIN TRANSACTION statement sets the value of @@TRANCOUNT to one. Its value increases by one for every BEGIN TRANSACTION statement, which follows and decreases by one with every COMMIT TRANSACTION statement. However, the ROLLBACK TRANSACTION statement sets the value of @@TRANSACTION to zero.

Consider the following statements:

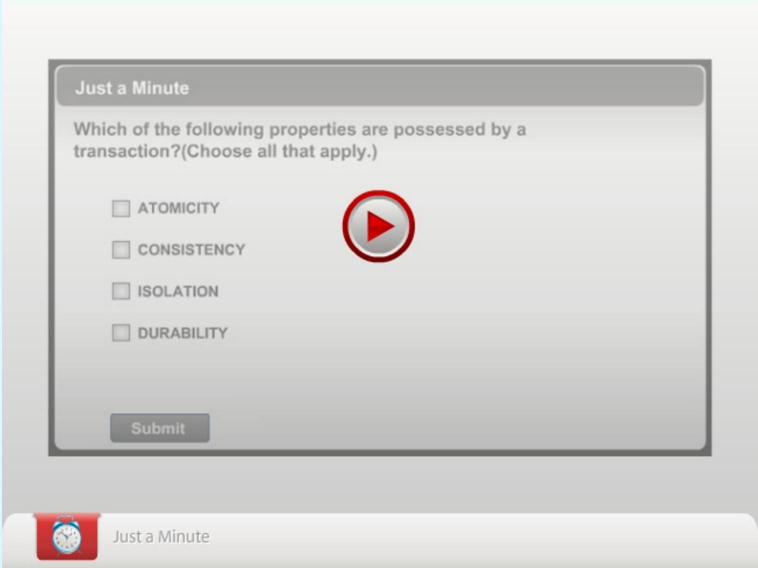
```
PRINT 'Count of active
transactions:'+convert(varchar
(3),@@TRANCOUNT)
BEGIN TRAN
 PRINT 'Beginning with the first
 Transaction '+ convert(varchar
 (3),@@TRANCOUNT)
 BEGIN TRAN
     PRINT'beginning with the second
transaction '+ convert(varchar
(3),@@TRANCOUNT)
      BEGIN TRAN
    PRINT'beginning with the third
transaction ' + convert(varchar
(3),@@TRANCOUNT)
COMMIT
PRINT 'commiting the first trasaction
'+ convert(varchar(3),@@TRANCOUNT)
COMMIT
PRINT 'commiting the second
transaction '+ convert(varchar
(3),@@TRANCOUNT)
commit
PRINT 'commiting the third transaction
'+ convert(varchar(3),@@TRANCOUNT)
```

The following figure displays the output of the preceding statements.



```
Count of active transactions:0
Beginning with the first Transaction    1
beginning with the second transaction    2
beginning with the third transaction    3
commiting the first trasaction    2
commiting the second transaction    1
commiting the third transaction    0
```

The Output Derived by Using the @@TRANCOUNT Function


Just a Minute

Which of the following properties are possessed by a transaction?(Choose all that apply.)

☐ ATOMICITY

☐ CONSISTENCY

☐ ISOLATION

☐ DURABILITY

Reverting Transactions

Sometimes, all the statements of a transaction do not execute successfully due to some problem. For example, in case of a power failure between two statements, one statement will be executed and the other will not. This leaves the transaction in an invalid state. In such a case, you need to revert to the statement that has been successfully executed to maintain consistency.

The ROLLBACK TRANSACTION statement rolls back an explicit or implicit transaction to the beginning of the transaction or to a savepoint within a transaction. A savepoint is used to divide a transaction into smaller parts. Savepoints allow you to discard parts of a long transaction instead of rolling back the entire transaction.

The syntax of the ROLLBACK TRANSACTION statement is:

```
ROLLBACK                  [TRAN[SACTION]
[transaction_name  |@tran_name_variable
|savepoint_name |
@savepoint_variable] ]
```

where,

`transaction_name` is the name assigned to the transaction. This parameter must conform to the rules for identifiers and should not be more than 32 characters.

`@tran_name_variable` is the name of a user-defined variable that contains a valid transaction name. This variable must be declared with a char, varchar, nchar, or nvarchar datatype.

`savepoint_name` is the name assigned to the savepoint.

`@savepoint_variable` is the name of a user-defined variable containing a valid savepoint name. This variable must be declared with a char, varchar, nchar, or nvarchar datatype.

For example, while updating the personal details of an employee, you want all the statements to be rolled back if any query fails. To ensure this, you can write the following statements:

```
BEGIN TRANSACTION TR1
BEGIN TRY
UPDATE Person.Contact
SET EmailAddress='jolyn@yahoo.com'
WHERE ContactID = 1070
--Statement 1
UPDATE   HumanResources.EmployeeAddress
SET AddressID = 32533
WHERE EmployeeID = 1
COMMIT TRANSACTION TR1
--Statement 2
SELECT 'Transaction Executed'
END TRY
BEGIN CATCH
ROLLBACK TRANSACTION TR1
SELECT 'Transaction Rollbacked'
END CATCH
```

In the preceding statements, the TR1 transaction updates the e-mail address of an employee. In addition, this transaction updates the address of the employee. In the query, the first statement is executed, but the second statement gives an error due to which the whole transaction is rolled back.

At times, you may want SQL Server to automatically roll back the current transaction if a T-SQL statement raises a runtime error. To accomplish this requirement, SQL Server provides the SET XACT_ABORT statement. The syntax of the SET XACT_ABORT statement is:

```
SET XACT_ABORT { ON | OFF }
```

If `SET  XACT_ABORT` is `ON` and a runtime error is encountered in the T-SQL statement, the entire current transaction is rolled back.

If `SET  XACT_ABORT` is `OFF` and a runtime error is encountered in the T-SQL statement, the current statement is rolled back and the transaction continues processing. However, depending on the severity of the transaction, the entire current transaction may be rolled back. `SET XACT_ABORT OFF` is the default setting.

Consider the following statements to create tables named table1 and table2:

```
CREATE TABLE table1
 (x INT NOT NULL PRIMARY KEY);
CREATE TABLE table2
 (x INT NOT NULL REFERENCES table1(x));
GO
```

Consider the following statements to insert records into table1:

```
INSERT INTO table1 VALUES (1);
INSERT INTO table1 VALUES (3);
INSERT INTO table1 VALUES (4);
INSERT INTO table1 VALUES (6);
GO
```

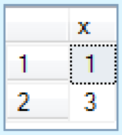
The following statement turns off the automatic rollback of the current transaction if any runtime error occurs:

```
SET XACT_ABORT OFF;
```

The following statements insert the values in table2:

```
BEGIN TRANSACTION;
INSERT INTO table2 VALUES (1);
INSERT  INTO  table2  VALUES  (2);  //This
statement   violates   referential   //
integrity and will be rolled back
INSERT INTO table2 VALUES (3);
COMMIT TRANSACTION;
GO
```

The preceding statements insert the values, 1 and 3, into table2. The value, 2, is not inserted into table2 as this value does not exist in table1. Therefore, it violates the referential integrity. The following figure displays the data stored in table2.



	x
1	1
2	3

The Data Stored Inside the Table2

Now, execute the following statement to turn on the automatic rollback of the current transaction if any runtime error occurs:

```
SET XACT_ABORT ON;
GO
```

Execute the following statements to insert more records in table2:

```
BEGIN TRANSACTION;
INSERT INTO table2 VALUES (4);
INSERT  INTO  table2  VALUES  (5);  //This
statement   violates   referential   //
integrity  and  the  entire  transaction
will be rolled back
INSERT INTO table2 VALUES (6);
COMMIT TRANSACTION;
GO
```

All the preceding statements are rolled back as the value, 5, does not exist in table1 and violates the referential integrity. Therefore, table2 still contains two records, 1 and 3.

Implementing Transactional Integrity

In a multi-user environment, multiple users can access the database server at a single point in time. Multiple users might also execute the UPDATE or SELECT statements on the same data. This may lead to data redundancy or incorrectness in the database.

Consider the example of AdventureWorks, Inc. A user sends a request to update all the records in a table. At the same time, another user might send a query to update data in only selected records in the same table. In such a case, there are chances of losing information that results inconsistency in the database. Such problems can be resolved by using locks.

SQL Server uses the concept of locking to ensure transactional integrity. In a multi-user environment, locking prevents different users from changing the same data at the same time. In SQL Server, locking is implemented automatically. However, you can also explicitly use locks. Locks are used to guarantee that the current user of a resource has a consistent view of that resource, from the beginning to the end of an operation. This ensures completeness of data modifications.

For a transaction-processing database, a DBMS resolves potential conflicts between two different processes that are attempting to change the same information at the same time.

NOTE *Transactional concurrency is the ability of multiple transactions to access or change the shared data at the same time. Transactional concurrency is impacted when a transaction trying to modify the data prevents other transactions from reading the data.*

Need for Locking

In the absence of locking, problems may occur if more than one transaction uses the same data from a database at the same time. These problems include:

- Lost updates
- Uncommitted dependency
- Inconsistent analysis
- Phantom reads

Lost Updates

A lost update occurs when two or more transactions try to modify the same row. In this case, each transaction is unaware of the other transaction. The last update in the transaction queue overwrites the updates made by the previous transactions. This leads to loss of data manipulation performed by the previous transactions.

For example, in a banking application, two transactions are simultaneously trying to update the balance details for a particular account. Both the transactions select data from the table simultaneously and get the same value for the current balance. When one transaction is committed, the balance is updated, but the second transaction does not get the updated value. Therefore, when the second transaction is committed, the changes done by the previous transaction will be lost. This results in the loss of an update.

Uncommitted Dependency

An uncommitted dependency is also known as a dirty read. This problem occurs when a transaction queries data from a table when the other transaction is in the process of modifying data.

For example, the details of all the products are stored in the Products table in a database. A user is executing a query to update the price for all the products. While the changes are being made, another user generates a report from the same table and distributes it to the intended audience. The update query finally commits and the table is updated now. In this case, the distributed report contains data that no longer exists and should be treated as redundant.

This problem is knows as the problem of dirty read. To avoid such a problem, you should not allow any user to read the table until the changes are finalized.

Inconsistent Analysis

An inconsistent analysis problem is also known as a non-repeatable problem. This problem arises when the data is changed between simultaneous read by one user.

For example, in a banking application, a user generates a report to display the balance of all the accounts. The user uses the result set to update data. Then, the user again retrieves the same result set to reflect the changes. Between the execution of these two queries, another user updates the original table. When the user queries the table for the second time, the data is changed. This leads to confusion.

Phantom Reads

A phantom read is also known as the phantom problem. This problem occurs when new records inserted by a user are identified by transactions that started prior to the INSERT statement. For example, in a ticket reservation application, a user, User 1, starts a transaction and queries the available seats. The query returns a value X. Then, the transaction tries to reserve the X seats. Simultaneously, another user, User 2, reserves the X-2 seats. When the transaction of User 1 tries to reserve X seats, the required number of seats is unavailable.

Locking in SQL Server

SQL Server implements multi-granular locking, which allows transactions to lock different types of resources at different levels. To minimize the effort on locking, SQL Server automatically locks resources at a level appropriate to the transaction, for example, row level or data page level. For transactions to access resources, SQL Server resolves a conflict between the concurrent transactions by using lock modes. SQL Server uses the following lock modes:

- ❑ Shared locks
- ❑ Exclusive locks
- ❑ Update locks
- ❑ Intent locks
- ❑ Schema locks
- ❑ Bulk update locks

Shared Locks

Shared (S) locks, by their functionality, allow concurrent transactions to read a resource. If there are any shared locks on a resource, no other transaction can modify the data on that resource. A shared lock releases the resource after the data has been read by the current transaction.

Exclusive Locks

Exclusive (X) locks, by their functionality, exclusively restrict concurrent transactions from accessing a resource. No other transaction can read or modify the data locked with an exclusive lock.

Update Locks

An update (U) lock falls in between a shared and an exclusive lock. For example, to update all the products with a price more that $10, you can run an UPDATE statement on the table. To determine the records that need to be updated, the query will acquire a shared lock on the table.

When physical updates occur, your query acquires an exclusive lock. In the time gap between the shared and the exclusive lock, any other transaction might change the data that you are going to update. Therefore, an update lock can be acquired. An update lock is applied to the table along with a shared lock, which prevents other transactions from updating the table until the update is complete.

Intent Locks

An intent (I) lock, by its functionality, indicates that SQL Server wants to acquire a shared or exclusive lock on some of the resources lower in the hierarchy. For example, when a shared intent lock is implemented at the table level, a transaction will place shared locks on pages or rows within that table.

Implementing an intent lock at the table level ensures that no other transaction can subsequently acquire an exclusive lock on the table containing that page. SQL Server examines the intent locks only at the table level to determine if a transaction can safely acquire a lock on that table. Therefore, you need not examine every row or page lock on the table to determine whether a transaction can lock the entire table.

Intent locks with their diversified features include Intent Shared (IS), Intent Exclusive (IX), and Shared with Intent Exclusive (SIX) locks.

Schema Locks

A schema lock (SL) is used when a schema dependent operation is performed on a table. SQL Server considers schema modification (Sch-M) locks when any DDL operation is performed on a table. However, SQL Server considers schema stability (Sch-S) locks while compiling queries. An Sch-S lock does not block other locks including the exclusive (X) locks. Therefore, other transactions including those with exclusive (X) locks on a table can run while a query is being compiled.

Bulk Update Locks

A bulk update lock (BU) secures your table from any other normal T-SQL statement, but multiple BULK INSERT statements or a bulk copy program can be performed at the same time.

Controlling Locks

Locks are implemented automatically in SQL Server. By default, SQL Server locks every row that you query. Sometimes, when you query a large record set, locks can grow from rows to data pages and further to table levels. If the query you are executing takes time to execute, it will prevent other users from accessing the database objects. This results in the lack of concurrency in the database. In addition, you might need to change the lock mode from a normal shared lock to an exclusive lock. To resolve such problems, you need to use *isolation levels*.

You can use isolation levels to specify the rights other transactions will have on the data being modified by a transaction. SQL Server supports the following types of isolation levels:

- ❑ READ UNCOMMITTED
- ❑ READ COMMITED
- ❑ REPEATABLE READ
- ❑ SNAPSHOT
- ❑ SERIALIZABLE

READ UNCOMMITTED

The READ UNCOMMITED isolation level specifies that a transaction can read the data modified by the current transaction but the modifications have not been committed yet. Transactions running with this isolation level do not perform a shared lock on the database object, enabling other transactions to modify the data being read by the current transaction. The database objects are also not blocked by the exclusive locks enabling other transaction to read the data being modified but not committed by the current transaction.

When this level is set, the transaction can read the uncommitted data, resulting in the dirty read problem. It is the least safe isolation level.

READ COMMITED

The READ COMMITED isolation level specifies that a transaction cannot read the data that is being modified by the current transaction. This prevents the problem of dirty read.

This isolation level places an exclusive lock on each UPDATE statement in the current transaction. When this isolation level is set, other transactions can update the data that has been read by the current transaction. This results in a problem of phantom read.

It is the default isolation level in SQL Server.

REPEATABLE READ

The REPEATABLE READ isolation level specifies that a transaction cannot read the data that is being modified by the current transaction. In addition, no other transaction can update the data that has been read by the current transaction until the current transaction completes.

This isolation level places an exclusive lock on each UPDATE statement within the current statement. In addition, it places a shared lock on each SELECT statement.

When this isolation level is set, other transactions can insert new rows, which result in a phantom read.

SNAPSHOT

The SNAPSHOT isolation provides every transaction with a snapshot of the current data. Every transaction works and makes changes on its own copy of the data. When a transaction is ready to update the changes, it checks whether the data has been modified since the time it started working on the data and decides whether to update the data.

The ALLOW_SNAPSHOT_ISOLATION database option must be set to ON before you can start a transaction that uses the SNAPSHOT isolation level. If a transaction using the SNAPSHOT isolation level accesses data in multiple databases, the ALLOW_SNAPSHOT_ISOLATION must be set to ON in each database.

SERIALIZABLE

The SERIALIZABLE isolation level specifies that no transaction can read, modify, or insert new data while the data is being read or updated by the current transaction.

This is the safest isolation level provided by SQL Server as it places a lock on each statement of the transaction. In this isolation level, the concurrency is at the lowest.

Implementing Isolation Levels

You can implement isolation levels in your transactions by using the SET TRANSACTION ISOLATION LEVEL statement before beginning a transaction.

The syntax of the SET TRANASCTION ISOLATION LEVEL statement is:

```
SET TRANSACTION ISOLATION LEVEL { READ
UNCOMMITTED | READ COMMITTED |
REPEATABLE READ | SNAPSHOT |
SERIALIZABLE } [ ; ]
BEGIN TRANSACTION
COMMIT TRANSACTION
```

For example, while updating the records of an employee, you do not want any other transaction to read the uncommitted records. For this, you can use the following statements:

```
SET TRANSACTION ISOLATION LEVEL
READ COMMITTED
BEGIN TRANSACTION TR
BEGIN TRY
 UPDATE Person.Contact
 SET EmailAddress='jolyn@yahoo.com'
 WHERE ContactID = 1070
 UPDATE HumanResources.EmployeeAddress
 SET AddressID = 32533
 WHERE EmployeeID = 1
 COMMIT TRANSACTION TR
 PRINT 'Transaction Executed'
END TRY
BEGIN CATCH
 ROLLBACK TRANSACTION TR
 PRINT 'Transaction Rollbacked'
END CATCH
```

The preceding statements sets the isolation level of transaction "TR" as READ COMMITTED. This prevents other transactions from reading the uncommitted updates in the tables.

Consider another example. While reading records from the EmployeePayHistory table, you do not want any other transaction to update the records until the current transaction completes its execution. For this, you can write the following statements:

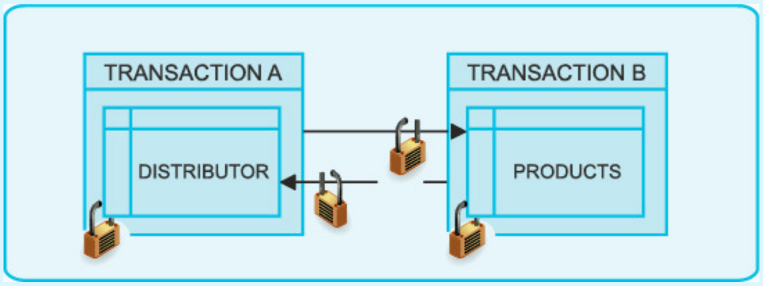
```
USE AdventureWorks
GO
SET TRANSACTION ISOLATION LEVEL
REPEATABLE READ;
GO
BEGIN TRANSACTION
GO
SELECT * FROM
HumanResources.EmployeePayHistory
GO
SELECT * FROM
```

```
HumanResources.Department;
GO
COMMIT TRANSACTION
GO
```

The preceding statements set the isolation level of the transaction as REPEATABLE READ. This prevents other transactions from updating the records untill the current transaction completes. However, new records can be inserted.

Resolving Deadlocks

A deadlock is a situation where two or more transactions have locks on separate objects, and each transaction waits for a lock on the other object to be released. Deadlocks are dangerous because they decrease the concurrency and availability of the database and the database objects. The following figure displays the deadlock between two objects.



The Deadlock Between Two Objects

In the preceding figure, Transaction A has locked the DISTRIBUTOR table and wants to lock the PRODUCTS table. Transaction B has locked the PRODUCTS table and wants to lock the DISTRIBUTOR table. This results in a deadlock as both the transactions wait for the other transaction to release the table. Since no lock will be released, this results in a deadlock.

Setting Deadlock Priority

SQL Server provides the SET DEADLOCK_PRIORITY statement to customize deadlocking. Setting DEADLOCK_PRIORITY as LOW for a session causes a particular session to be chosen as the deadlock victim. The DEADLOCK_PRIORITY option controls how the particular session reacts in a deadlock.

The syntax of the DEADLOCK_PRIORITY statement is:

```
SET DEADLOCK_PRIORITY {LOW | NORMAL |
@deadlock_var}
```

where,

`LOW`is used to specify that the current session is the preferred deadlock victim.

`NORMAL`is used to specify that the session returns to the default deadlock-handling method.

`@deadlock_var`is a character variable with a length of three charactersfor string of low priority (LOW) and six characters for string of normal priority (NORMAL). It also specifies the deadlock-handling method.

NOTE *The periodic detection mechanism of SQL Server reduces the overhead of deadlock detection because deadlocks affect only a small number of transactions.*

Customizing Lock Timeout

When a transaction attempts to lock a resource, which is already held by another transaction, SQL Server informs the first transaction of the current availability status of the resource. If the resource is locked, the first transaction is blocked, waiting for that resource. If there is a deadlock, SQL Server terminates one of the participating transactions. In case a deadlock does not occur, the requesting transaction is blocked until the other transaction releases the lock. By default, SQL Server does not enforce the *timeout* period.

The SET LOCK_TIMEOUT statement can be used to set the maximum time that a statement waits on a blocked resource. After LOCK_TIMEOUT is set, when a statement has waited longer than the LOCK_TIMEOUT setting, SQL Server automatically cancels the waiting transaction. The syntax of LOCK_TIMEOUT is:

```
SET LOCK_TIMEOUT [timeout_period]
```

where,

`timeout_period` is represented in milliseconds and is the time that will pass before SQL Server returns a locking error for a blocked transaction. You can specify a value of 1 to implement default timeout settings.

Detecting Deadlocks

In order to detect deadlock situations, SQL Server scans for sessions that are waiting for a lock request. SQL Server, during the initial scan, marks a flag for all the waiting sessions. When SQL Server scans the sessions for the second time, a recursive deadlock search begins. If any circular chain of lock requests is found, SQL Server cancels the least expensive transaction and marks that transaction as the deadlock victim.

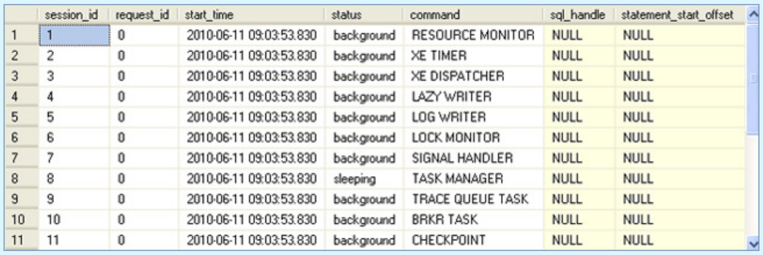
With the intelligent use of a deadlock scan for sessions, SQL Server ends a deadlock by automatically selecting the user who can break the deadlock as the deadlock victim. In addition, after rolling back the deadlock victim's transaction, SQL Server notifies the user's application through error message number 1205.

Using sys.dm_exec_requests

The sys.dm_exec_requests system view returns information about each transaction that is executing within SQL Server. This view can be used to detect the transaction that is causing the deadlock. You can query this view by using the following query:

```
SELECT * FROM sys.dm_exec_requests
```

The following figure shows the output of the preceding query.



	session_id	request_id	start_time	status	command	sql_handle	statement_start_offset
1	1	0	2010-06-11 09:03:53.830	background	RESOURCE MONITOR	NULL	NULL
2	2	0	2010-06-11 09:03:53.830	background	XE TIMER	NULL	NULL
3	3	0	2010-06-11 09:03:53.830	background	XE DISPATCHER	NULL	NULL
4	4	0	2010-06-11 09:03:53.830	background	LAZY WRITER	NULL	NULL
5	5	0	2010-06-11 09:03:53.830	background	LOG WRITER	NULL	NULL
6	6	0	2010-06-11 09:03:53.830	background	LOCK MONITOR	NULL	NULL
7	7	0	2010-06-11 09:03:53.830	background	SIGNAL HANDLER	NULL	NULL
8	8	0	2010-06-11 09:03:53.830	sleeping	TASK MANAGER	NULL	NULL
9	9	0	2010-06-11 09:03:53.830	background	TRACE QUEUE TASK	NULL	NULL
10	10	0	2010-06-11 09:03:53.830	background	BRKR TASK	NULL	NULL
11	11	0	2010-06-11 09:03:53.830	background	CHECKPOINT	NULL	NULL

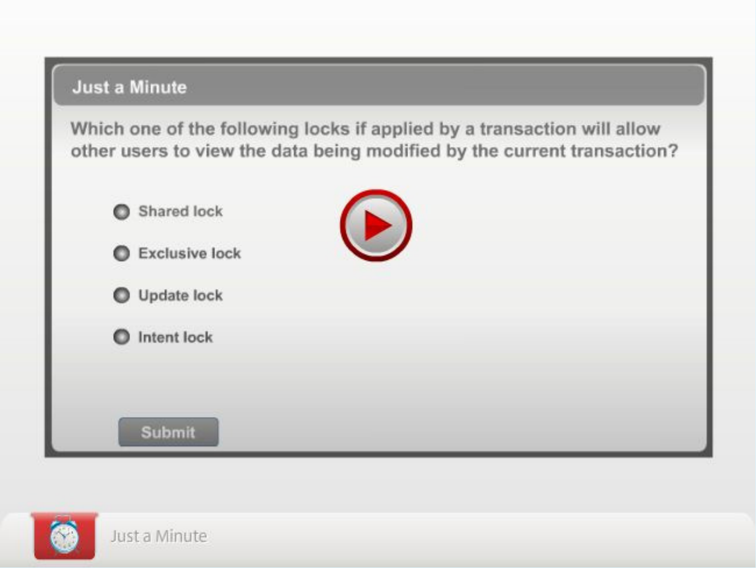
The Output Derived by Using sys.dm_exec_requests

The output shows the list of all the transactions currently running on the server. In addition, it shows the status of all the transactions, which includes the session_id, status, and the owner.

Avoiding Deadlocks by Using Update Locks

When two concurrent transactions acquire shared mode locks on a resource, and then attempt to update the data concurrently, one transaction attempts conversion of the lock to an exclusive lock. In such a scenario, the conversion from a shared mode to an exclusive lock must wait. This is because the exclusive lock for one transaction is not compatible with the shared mode lock of the other transaction. Therefore, a lock wait occurs.

The second transaction modifies the data and attempts to acquire an exclusive lock. Under these circumstances, when both the transactions are converting from shared to exclusive locks, a deadlock occurs because each transaction is waiting for the other transaction to release its shared mode lock. Therefore, update locks are used to avoid this potential deadlock problem. SQL Server allows only one transaction to obtain an update lock on a resource at a time. The update lock is converted to an exclusive lock if a transaction modifies a resource. Otherwise, the lock is converted to a shared mode lock.



Just a Minute

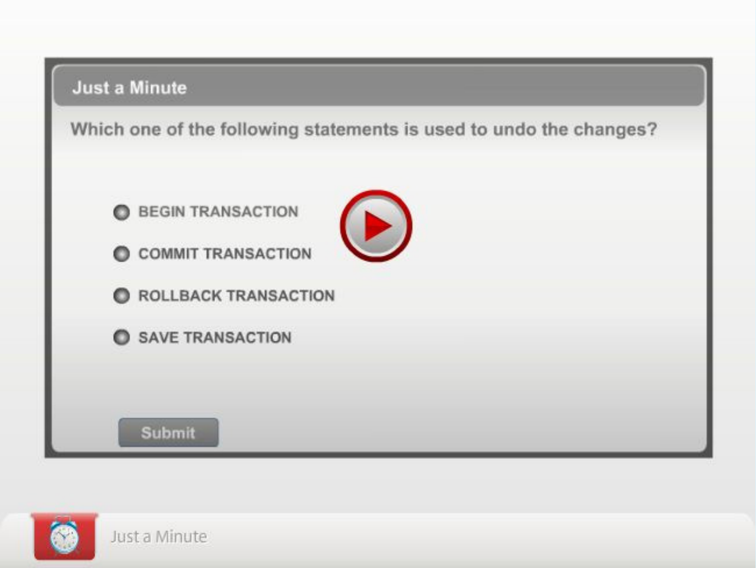
Which one of the following locks if applied by a transaction will allow other users to view the data being modified by the current transaction?

○ Shared lock

○ Exclusive lock

○ Update lock

○ Intent lock



Just a Minute

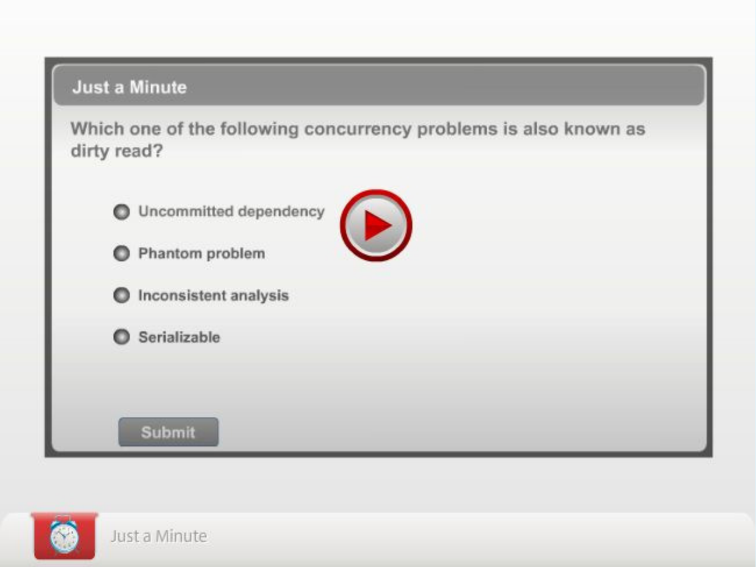
Which one of the following statements is used to undo the changes?

○ BEGIN TRANSACTION

○ COMMIT TRANSACTION

○ ROLLBACK TRANSACTION

○ SAVE TRANSACTION



Just a Minute

Which one of the following concurrency problems is also known as dirty read?

○ Uncommitted dependency

○ Phantom problem

○ Inconsistent analysis

○ Serializable

Activity 8.2: Implementing Transactions

Summary

In this chapter, you learned that:

- A trigger is a block of code that constitutes a set of T-SQL statements that are activated in response to certain actions.
- SQL Server supports the following triggers:
  - DML triggers
  - DDL triggers
- You can alter and delete a trigger.
- SQL Server provides the DISABLE TRIGGER statement to disable an existing trigger.
- SQL Server provides the ENABLE TRIGGER statement to enable an existing trigger.
- SQL Server provides the EVENTDATA() function that can be embedded with the CREATE TRIGGER statement to provide the information, such as time and type of the event that caused the trigger to fire.
- Transactions are used to execute a sequence of statements together as a single logical unit of work.
- Every transaction possesses the ACID property.
- SQL Server supports the following transactions:
  - Autocommit transaction
  - Implicit transaction
  - Explicit transaction
- Locks are used to maintain transactional integrity.
- In the absence of locking, the following problems may occur if multiple transactions use the same data from a database at the same time:
  - Lost updates
  - Uncommitted dependency
  - Inconsistent analysis
  - Phantom reads
- SQL Server supports the following lock modes:
  - Shared locks
  - Exclusive locks
  - Update locks
  - Intent locks
  - Schema locks
  - Bulk update locks
- A deadlock is a situation where two users (or transactions) have locks on separate objects, and each user wants to acquire a lock on the other user's object.

Reference Reading

Implementing Triggers

Reference Reading: Books	***Reference Reading: URLs***
Beginning Microsoft SQL Server 2012 Programming by Paul Atkinson and Robert Vieira	*http://msdn.microsoft.com/en-us/library/ms189799.aspx*

Implementing Transactions

Reference Reading: Books	***Reference Reading: URLs***
Professional Microsoft SQL Server 2012 Administration by Adam Jorgensen, Steven Wort, and Ross LoForte	*http://msdn.microsoft.com/en-us/library/ms174377.aspx*

Chapter 9

Monitoring and Optimizing Performance

With an increase in the size of a database or the number of users, the performance of the database server might get affected. An important task of a database administrator is to monitor the performance of the database server in terms of the time taken in responding to user queries or generating reports.

In addition to monitoring performance, a database administrator is responsible for the security of data in the database. Many users connect to the database server and make changes to the data stored in the database. The details regarding who accessed the database and what changes have been made are important for the security and integrity of data.

This chapter discusses the ways of monitoring and optimizing the performance of a database server by using the tools provided by SQL Server.

Objectives

In this chapter, you will learn to:

- ❑ Monitor performance
- ❑ Optimize performance

Monitoring Performance

One of the important activities of a database administrator is to monitor the performance of the database server. The performance of a database server is measured by the time and resources consumed in responding to queries of the users. An administrator should regularly monitor the performance of a database server and implement appropriate solutions to optimize it. SQL Server provides various tools to monitor and optimize performance.

Monitoring Performance by Using SQL Server Profiler

While monitoring performance of a server, an administrator needs to check for the events occurring on the server. These events include the execution of T-SQL statements or a change in the database properties.

SQL Server Profiler provides a graphical user interface to monitor the life cycle of the transactions being executed on a particular instance of the database engine. You can track the login details of users, or the execution of individual T-SQL statements or transactions. This tool is used by business application developers to identify the T-SQL statements or stored procedures that slow down the performance of the server.

To enable server monitoring, SQL Server Profiler tracks the events that occur in the database engine. An *event* is an action generated within an instance of SQL Server, such as execution of T-SQL statements, start or end of stored procedures, or error returned by the database engine. An event can also include locks acquired or released by the database objects.

While monitoring the performance, you need to create a *trace* and specify the type of events you need to track. A trace is a type of template that contains the details of the events to be monitored. SQL Server Profiler uses a trace to store the details of the events you need to monitor.

Creating, Configuring, and Starting a Trace

While creating a trace, you can specify the events and the values for the events that you need to monitor. For example, you might need to monitor only those events for which the CPU time is greater than or equal to 100 milliseconds. Similarly, you can also monitor the events only for selected logins. To specify the values for which you want to monitor the events, you can apply filters.

If you need to monitor the same set of events regularly, you can store the trace settings in a *template*. SQL Server Profiler supports the following types of templates:

- ❑ Default template
- ❑ Custom template

Default Template

The default template is the template shipped with SQL Server Profiler. Under the default template, SQL Server captures typical values from a trace, such as start time, end time, and login name.

Custom Template

In addition to the default template, SQL Server Profiler allows the database administrators to create their own templates. You can select the columns and values to be included in the trace.

Consider an example. Robert, a database developer at Adventure Works, Inc. has notified the database administrator that the queries being executed from his computer are taking longer than usual. The database administrator has decided to trace the queries being executed from the computer used by Robert. In addition, the database administrator will need to trace the queries in future. Therefore, the database administrator needs to create a template so that it can be reused whenever required.

To monitor the activities of users, the database administrator needs to perform the following tasks:

1. Create a template.
2. Create and run a trace.

Task 9.1: Creating a Template

Task 9.2: Creating and Running a Trace

You can view the selected events in the trace file as they occur in the database server. In addition, the trace is stored in the file. This file can be used later to analyze the performance of the server. You can also correlate the event details stored in this trace file with the performance data logged by the System Monitor tool.

Monitoring Performance by Using DMVs

When monitoring the performance of a server, you might also need to check the state of the server at any given point in time. The state of the server can be in terms of the number of sessions running on the server or the amount of free space available.

SQL Server provides database administrators with Dynamic Management Views (DMVs), which return server state information. This information can be used to monitor the health of a server instance, diagnose problems, and tune performance. The DMVs are used as a tool to identify the activities of SQL Server.

The DMVs are a group of views and table-valued functions. Some DMVs are applied on the server as a whole and are stored in the master database. To view these DMVs, the user must have the VIEW SERVER STATE permission. The remaining DMVs are specific to a particular database, and to view them the user must have the VIEW DATABASE STATE permission. DMVs are defined in the sys schema and the names contain the dm prefix to distinguish them from catalog views. For example, the sys.dm_db_partition_stats DMV can be used to view the partition statistics.

In SQL Server, DMVs are organized in the various categories, as shown in following table.

Name of the category	***Used to***	***DMVs present in the category***
CLR-related DMVs	*Check the state of the server with regard to the CLR.*	*sys.dm_clr_appdomains* *sys.dm_clr_properties* *sys.dm_clr_tasks* *sys.dm_clr_loaded_assemblies*
I/O-related DMVs	*Check the details about the state of various I/O operations, such as backup and restore.*	*sys.dm_io_backup_tapes* *sys.dm_io_pending_io_requests* *sys.dm_io_cluster_shared_drives* *sys.dm_io_virtual_files_stats*
Database mirroring-related DMVs	*Check the details about the mirroring connections available on a server.*	*sys.dm_db_mirroring_connections* *sys.dm_db_mirroring_auto_page_repair*
Query notifications-related DMVs	*Check the details about active query notification subscriptions on the server.*	*sys.dm_qn_subscriptions*
Database-related DMVs	*Check the details about the database state present in SQL Server, such as the session space or the partition space of the current database.*	*sys.dm_db_file_space_usage* *sys.dm_db_session_space_usage* *sys.dm_db_partition_stats* *sys.dm_db_task_space_usage*
Replication-related DMVs	*Check the details about the state of various objects related to replication, such as the list of various database objects published in a replication.*	*sys.dm_repl_articles* *sys.dm_repl_tranhash* *sys.dm_repl_schemas* *sys.dm_repl_traninfo*
Execution-related DMVs	*Provide information related to the state of query execution on the server, such as the current sessions list and the query execution plans for queries.*	*sys.dm_exec_sessions* *sys.dm_exec_requests* *sys.dm_exec_sql_text* *sys.dm_exec_query_stats* *sys.dm_exec_query_optimizer_info* *sys.dm_exec_connections*
Service Broker-related DMVs	*Provide information related to the state of Service Broker objects on the server, such as the Service Broker*	*sys.dm_broker_activated_tasks* *sys.dm_broker_connections* *sys.dm_broker_forwarded_messages* *sys.dm_broker_qu*

	connections and the list of Service Broker queues.	*eue_monitors*
Full-text Search-related DMVs	*Provide information regarding various full-text related objects, such as currently active catalogs and available memory buffers.*	*sys.dm_fts_active_catalogs* *sys.dm_fts_crawl_angles* *sys.dm_fts_memory_pools* *sys.dm_fts_memory_buffers*
SQL Operating System-related DMVs	*Check the information about the operating system resources that are specific to SQL Server.*	*sys.dm_os_buffer_descriptors* *sys.dm_os_memory_pools* *sys.dm_os_child_instances* *sys.dm_os_sys_info* *sys.dm_os_loaded_modules* *sys.dm_os_tasks* *sys.dm_os_memory_clerks* *sys.dm_os_workers*
Index-related DMVs	*Provide details about the state of various indexes available on SQL Server.*	*sys.dm_db_index_operational_stats* *sys.dm_db_index_physical_stats* *sys.dm_db_index_usage_stats*
Transaction-related DMVs	*Provide details about the state of various transactions currently running on the server.*	*sys.dm_tran_active_snapshot_database_transactions* *sys.dm_tran_active_transactions* *sys.dm_tran_current_snapshot* *sys.dm_tran_current_transaction*
Resource Governor DMVs	*Display status of the resource governor.*	*sys.dm_resource_governor_configuration* *sys.dm_resource_governor_workload_groups* *sys.dm_resource_governor_resource_pools*
Security-related DMVs	*Display information about security measures.*	*sys.dm_audit_actions* *sys.dm_cryptographic_provider_properties* *sys.dm_audit_class_type_map* *sys.dm_cryptographic_provider_sessions* *sys.dm_cryptographic_provider_algorithms* *sys.dm_database_encryption_keys* *sys.dm_cryptographic_provider_keys* *sys.dm_server_audit_status*
Change Tracking-related DMVs	*Display transaction committed for a table that is tracked by SQL Server change tracking.*	*sys.dm_tran_commit_table*
Change Data Capture-related DMVs	*Return information about change data capture, such as row for each log scan session in the current database, error encountered during the change data capture log scan session.*	*sys.dm_cdc_log_scan_sessions* *sys.dm_cdc_errors* *sys.dm_repl_traninfo*
Extended Events DMVs	*Display information about extended events.*	*sys.dm_xe_map_values* *sys.dm_xe_session_events* *sys.dm_xe_object_columns* *sys.dm_xe_session_object_columns* *sys.dm_xe_objects* *sys.dm_xe_session_targets* *sys.dm_xe_packages* *sys.dm_xe_sessions* *sys.dm_xe_session_event_actions*

Object-related DMVs	*Return one row for each entity in the current database that references another user-defined entity by name.*	*sys.dm_sql_referenced_entities* *sys.dm_sql_referencing_entities*

The Categories of DMVs

The syntax to use a DMV is:

```
SELECT * FROM [DMV_NAME]
```

where,

`DMV_NAME` is the name of the DMV you want to view.

For example, you can use the sys.dm_tran_database_transactions DMV to view the information about transactions at the database level, as shown in the following query:

```
SELECT * FROM sys.dm_tran_database_transactions
```

DMVs are the most important tools that the database administrators use to monitor SQL Server. Some of the frequently used DMVs are:

- ❑ sys.dm_exec_sessions
- ❑ sys.dm_exec_connections
- ❑ sys.dm_exec_requests

The sys.dm_exec_sessions DMV

This DMV is used to view the authenticated SQL Server sessions, as shown in the following query:

```
SELECT * FROM sys.dm_exec_sessions
```

The preceding query provides a list of all the authenticated sessions running on a particular server instance.

> NOTE
>
> *The information returned from the sys.dm_exec_sessions DMV can be used to identify a session or transaction that is affecting the performance. For example, if a session is using lot of memory space, you can kill that session by using the KILL statement. The syntax of the KILL statement is:*
>
> ```
> KILL
> ```
>
> *where, <sp_id> is the session ID of an errant session.*

The sys.dm_exec_connections DMV

This DMV is used to view the connection information of all the sessions currently connected to SQL Server. The details include the IP address and the session ID of the client. You can use the following query to execute this DMV:

```
SELECT * FROM sys.dm_exec_connections
```

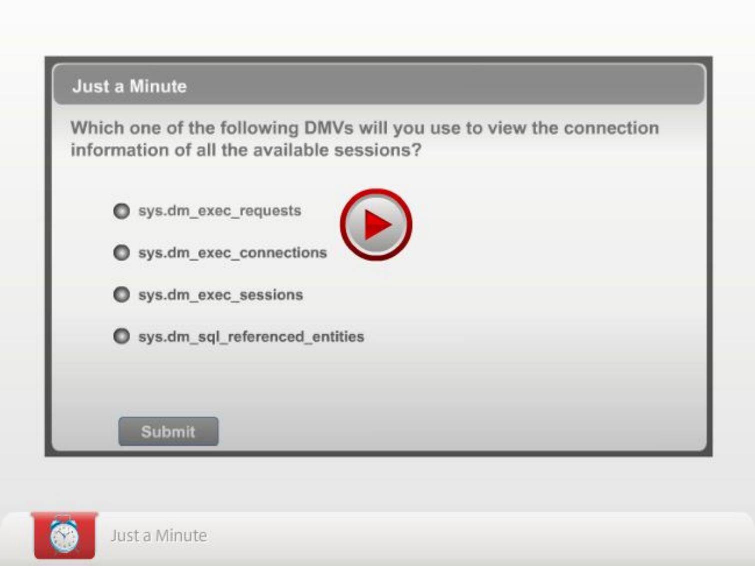
The preceding query provides the connection details of all the sessions currently available.

The sys.dm_exec_requests DMV

This DMV is used to view the requests being passed by various sessions on the server. This DMV can be used to observe each session on the server. You can use the following query to execute this DMV:

```
SELECT * FROM sys.dm_exec_requests
```

The preceding query displays a list of all the requests passed on the server by various sessions.



Just a Minute

Which one of the following DMVs will you use to view the connection information of all the available sessions?

○ sys.dm_exec_requests

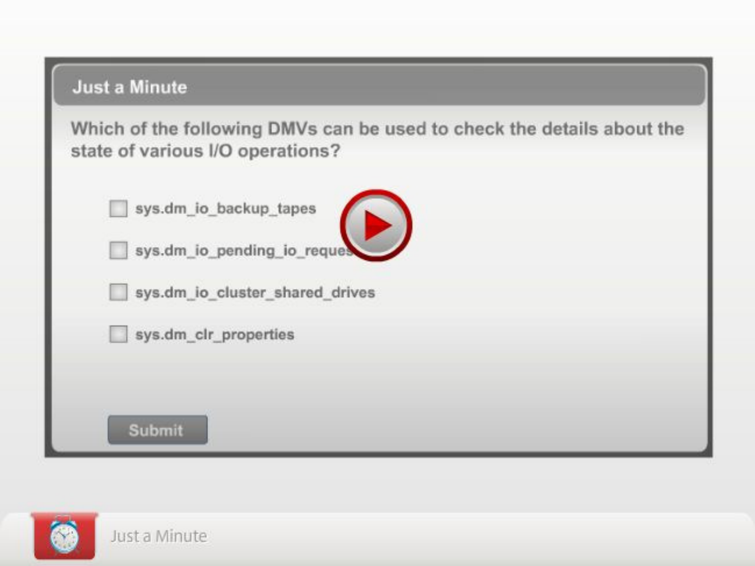
○ sys.dm_exec_connections

○ sys.dm_exec_sessions

○ sys.dm_sql_referenced_entities

Submit

Just a Minute



Just a Minute

Which of the following DMVs can be used to check the details about the state of various I/O operations?

□ sys.dm_io_backup_tapes

□ sys.dm_io_pending_io_reques

□ sys.dm_io_cluster_shared_drives

□ sys.dm_clr_properties

Submit

Just a Minute

Monitoring Performance by Using Database Engine Tuning Advisor

Database tuning is one of the most important tasks that a database administrator performs. It involves a set of activities used to optimize the performance of a database. It refers to configuring the database files, the Database Management System (DBMS), and the operating system and hardware the DBMS runs on.

The goal of database tuning is to maximize the use of system resources. Most systems are designed to manage work efficiently, but it is possible to improve performance by customizing settings and configuring the database and the DBMS. For example, if your query takes three seconds to execute, you can tune it to return the result in less than one second.

SQL Server provides Database Engine Tuning Advisor to help the database administrator tune the database. Database Engine Tuning Advisor tests the database in various conditions by putting a *workload* on it. Workload is a set of T-SQL statements that are executed on the database. The workload can be a trace file, a trace table, or a file consisting of T-SQL statements.

Based on the throughput of the queries given in the workload, a report is generated containing the current state of the database objects. In addition, the report includes suggestions to tune the database. The suggestions include partitioning the table, partitioning the indexes, dropping the indexes, or creating new indexes.

For example, if one of your database clients complains about the slow execution of queries, you can record the statements executed by the client in a trace and put them in the Database Engine Tuning Advisor to resolve the issues causing the slowing down of the query performance.

Task 9.3: Running the Database Engine Tuning Advisor

Optimizing Performance

When creating a database, a lot of indexes are created to improve the query performance. The performance of indexes decreases as the database grows in size. This results in a decrease in the query processing performance.

As a database administrator, it is your responsibility to ensure that the performance of queries does not suffer. For this, you should check and regulate the performance of indexes. You can do this by checking the *statistics* information. In addition, you can use the *Statistics IO* to get information regarding the amount of disk activity generated.

Implementing Statistics

Statistics in SQL Server refers to information that the server collects about the distribution of data in columns and indexes. This data, in turn, is used by the query optimizer to determine the execution plan for returning results when you run a query. Statistics is created automatically when you create indexes. In addition, SQL Server creates statistics for columns that do not have indexes defined on them.

SQL Server can automatically create and update statistics. This feature is set to ON by default. For majority of databases and applications, developers and administrators rely on the automatic creation and modification of statistics to provide accurate statistics about their data. By using SQL Server query optimizer, you can produce effective query plans consistently. This also helps in keeping the development and administration costs low. However, if you need to use statistics on columns that are not indexed in order to create more optimal execution plans, you can manually create and update statistics.

Creating Statistics

Statistics can be created on specific columns of a table or view by using the CREATE STATISTICS statement. The syntax of the CREATE STATISTICS is:

```
CREATE STATISTICS statistics_name
ON { table | view } ( column
[ ,...n ] )
[ WITH
   [ [ FULLSCAN
      | SAMPLE number { PERCENT |
ROWS } ]
   [ NORECOMPUTE ] ]
```

where,

`statistics_name` is the name of the statistics group to be created.

`table` is the name of the table on which the named statistics is to be created.

`view` is the name of the view on which the named statistics is to be created.

`column` is the column or set of columns on which statistics is to be created.

`FULLSCAN` specifies that all rows in the table or view should be read to gather the statistics. This option cannot be used with the SAMPLE option.

`SAMPLE number { PERCENT | ROWS }` specifies that a percentage or a specified number of rows of data should be read by using random sampling to gather the statistics number must be an integer. If PERCENT is specified, number should be from 0 through 100; if ROWS is specified, number can be from 0 to the total number of rows.

`NORECOMPUTE` specifies that the database engine should not automatically recompute statistics.

For example, you can create the ContactEMail statistics on the ContactID and EmailAddress columns of the Contact table of the AdventureWorks database by using the following statement:

```
USE AdventureWorks
GO
CREATE STATISTICS ContactEMail
ON      Person.Contact      (ContactID,
EmailAddress)
GO
```

You can view the list of all the statistics being maintained on a table by using the sp_helpstats system stored procedure. For example, you can view all the statistics on the Person.Contact table by using the following statement:

```
sp_helpstats 'Person.Contact', 'ALL'
```

You can also view the details of an individual statistics by using the DBCC SHOW_STATISTICS statement, as shown in the following statement:

```
DBCC                    SHOW_STATISTICS
('Person.Contact', AK_Contact_rowguid)
```

The preceding statement displays the information for the AK_Contact_rowguid statistics on the Person.Contact table.

By default, the AUTO_CREATE_STATISTICS database option is set as ON. As a result, the database engine automatically creates statistics for columns without indexes that are used in the WHERE clause.

Updating Statistics

The query optimizer determines when to update the statistics. In some cases, you need to update the statistics more frequently than automatically updated by the query optimizer. For example, the SalesDetail table contains the details of sales transactions. The data in this table is huge. Every day, lot of records are inserted or updated in this table. Due to huge data in the table, it takes too long to retrieve the data. In this case, you need to update the statistics manually to make the query optimizer use the updated statistics to enhance query performance.

Statistics can be updated by using the UPDATE STATISTICS statement. You can update the information about the distribution of key values for one or more statistics groups in the specified table or indexed view.

The syntax of the UPDATE STATISTICS statement is:

```
UPDATE STATISTICS table | view
 [
    {
        { index | statistics_name }
          |     (     {     index     |
 statistics_name } [ ,...n ] )
          }
        ]
        [WITH
        [
         [FULLSCAN ]
         |SAMPLE number { PERCENT |
ROWS } ]
         |RESAMPLE
       ]
        [ [ , ] [ ALL | COLUMNS |
INDEX ]
        [ [ , ] NORECOMPUTE ]
       ]
```

where,

`table | view` is the name of the table or indexed view for which the statistics is to be updated.

`index` is the index for which statistics are being updated.

`statistics_name` is the name of the statistics group (collection) to update.

`FULLSCAN` specifies that all rows in the table or view should be read to gather the statistics.

`SAMPLE number { PERCENT | ROWS }` specifies the percentage of the table or indexed view, or the number of rows to sample when collecting statistics for larger tables or views.

`RESAMPLE` specifies that statistics will be gathered by using an inherited sampling ratio for all existing statistics including indexes.

`ALL | COLUMNS | INDEX` specifies whether the UPDATE STATISTICS statement affects column statistics, index statistics, or all existing statistics.

`NORECOMPUTE` specifies that statistics that become out of date are not automatically recomputed.

For example, you can update the distribution statistics for all indexes on the Person.Contact table by using the following statements:

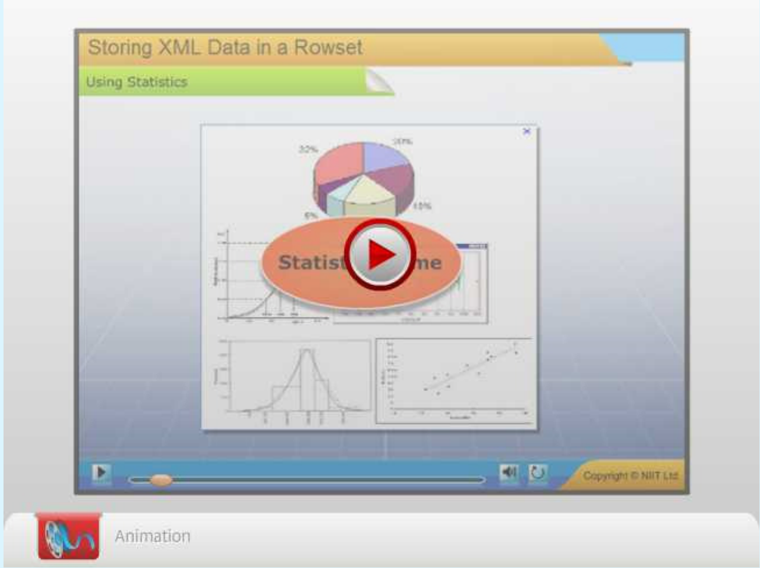
```
USE AdventureWorks
GO
UPDATE STATISTICS Person.Contact
GO
```

The following statement updates only the distribution information for the AK_Contact_rowguid index of the Person.Contact table:

```
UPDATE    STATISTICS    Person.Contact
AK_Contact_rowguid
```

Sampling the data, instead of analyzing all the data, minimizes the cost of automatic statistical update. You can control the amount of data sampled during manual statistics updates on a table-by-table basis by using the SAMPLE and FULLSCAN clauses of the UPDATE STATISTICS statement.

Statistics can also be created on all eligible columns in all user tables in the current database in a single statement by using the sp_createstats system stored procedure.



Statistics IO

In some cases, you need to know about the statistics information for the T-SQL statement. For example, you are retrieving the sales records from the SalesDetail table. You want to know about the amount of logical reads, physical reads and scan counts generated by the query.

In this case, you can use the Statistics IO to display information regarding the amount of disk activity generated by T-SQL statement. The syntax of statistics IO is:

```
SET STATISTICS IO { ON | OFF }
```

where,

`{ ON | OFF }` specifies that the statistics information is displayed when the STATISTICS is ON and the statistics information is not displayed when STATISTICS is OFF.

For example, you can get the disk information on the Person.Address table by using the following statements:

```
USE AdventureWorks
GO
SET statistics IO on
select * from person.address where
city='bothell'
SET STATISTICs IO off
GO
```

The preceding statements displays the disk information of the Person.Contact table.

Implementing Plan Guides

Consider a scenario where you are working as a database administrator at DBM Technologies. The company works with many third-party vendors. These vendors often execute some of their queries on the database application of the company. Some queries of third-party vendors are not performing as expected. Moreover, you do not have the permission to change the query text to optimize the performance of these queries.

In the preceding scenario, you can use the plan guides to optimize the performance of such queries. The plan guides allow you to attach query hints or a fixed query plan with the query to optimize its performance. You can attach the query hints by using the OPTION clause. Otherwise, you can attach a specific query plan with the query to optimize its performance. When the query executes, the OPTION clause of the plan guide that matches the query is attached to it for execution, or it uses the specified query plan.

The following types of plan guides can be created:

- ❑ **OBJECT**: This plan guide matches the query that executes in the context of stored-procedures, scalar user-defined functions, multi-statement table-valued user-defined functions, and DML triggers.
- ❑ **SQL**: This plan guide matches the query that executes in the context of a stand-alone T-SQL statement and batches that are not part of any database object like a procedure or a function.
- ❑ **TEMPLATE**: This plan guide matches the query that executes in the context of parameterized T-SQL statements or batches. A parameterized query allows you to substitute the literal values contained in a query with the parameters provided at the time of execution.

The sp_create_plan_guide statement is used to create the query plan. The syntax of the sp_create_plan_guide statement is:

```
sp_create_plan_guide [ @name = ]
N'plan_guide_name'
, [ @stmt = ] N'statement_text'
, [ @type = ] N'{ OBJECT | SQL |
TEMPLATE }'
, [ @module_or_batch = ]
{
        N'[ schema_name. ]
object_name'
      | N'batch_text'
      | NULL
}
, [ @params = ] { N'@parameter_name
data_type [ ,...n ]' | NULL }
, [ @hints = ] { N'OPTION ( query_hint
[ ,...n ] )'
          | N'XML_showplan'
          | NULL }
```

where,

`[ @name = ] N'plan_guide_name'` is the name of the plan guide.

`[ @stmt = ] N'statement_text'` is a T-SQL statement against which query plans need to be created.

`[ @type = ] N'{ OBJECT | SQL | TEMPLATE }'` is the context for matching the SQL statement to the plan_guide name.

`[ @module_or_batch = ]`

`{ N'[ schema_name. ] object_name' | N'batch_text' | NULL }` is the name of an object or a batch in which the query appears.

`[ @params = ] { N'@parameter_name data_type [ ,...n ]' | NULL }` specifies the definitions of all the parameters embedded in the query statement.

`[ @hints = ] { N'OPTION ( query_hint [ ,...n ] )' | N'XML_showplan' | NULL }` specifies the OPTION clause to attach to the query. The OPTION clause specifies the query hints to be attached.

Consider the following stored procedure:

```
CREATE PROCEDURE
HumanResources.GetEmployeeByCity
(@City varchar(20))
 AS
 BEGIN
 SELECT * FROM HumanResources.Employee
 E, HumanResources.EmployeeAddress EA,
 Person.Address PA
 Where PA.AddressID=EA.AddressID
 AND EA.EmployeeID=E.EmployeeID
 AND City = @City
 END
```

The preceding stored procedure takes the `@City` parameter and returns the records of those employees who live in the specified city.

Suppose, the preceding stored procedure is optimized for `@City = 'Portland'`. However, only a few employees live in Portland. The performance suffers when the query executes for other cities in which more employees live. As most of the employees live in the Bothell city, a query plan that is created for `@City = 'Bothell'` will optimize the query.

The `OPTIMIZE FOR` query hint can be used to instruct the query optimizer to use a specific value for a local variable when the query is compiled and optimized. Consider the following plan guide to optimize the stored procedure, `GetEmployeeByCity` for `@City='Bothell'`:

```
EXEC sp_create_plan_guide
 @name = N'Plan1',
@stmt = N'SELECT * FROM
HumanResources.Employee E,
    HumanResources.EmployeeAddress EA,
Person.Address PA
    Where PA.AddressID=EA.AddressID
    AND EA.EmployeeID=E.EmployeeID
    AND City = @City',
 @type = N'OBJECT',
 @module_or_batch =
 N'HumanResources.GetEmployeeByCity',
 @params = NULL,
 @hints = N'OPTION (OPTIMIZE FOR (@City
 = N''Bothell''))'
```

After creating the preceding plan guide, Plan1, when the query specified in the `sp_create_plan_guide` statement executes, the query is modified to include the `OPTIMIZE FOR (@City = N''Bothell'')` clause for optimizing the query.

Activity 9.1: Monitoring the Database

Tracking Data Changes

In an organization, tables are created to store the details of entities, such as employees, customer, and sales. The data in these tables is changed frequently by inserting, updating, or deleting the records. Sometimes, it is required to track the changes made on these tables to view the historical data. For example, an employee is shifted to another department. For this, you need to update the department details of that employee in the table. However, you also want to retain the previous department details of the employee as well in a separate table. For this, you need to track the changes made in the Employee table.

SQL Server provides functionality to track data changes by using Change Data Capture (CDC) and change tracking methods. CDC provides detailed information about the data that got changed in the table. It provides you with the details of the historical data as well as the updated data. However, change tracking only enables you to capture the information that the data in the table has changed.

Using CDC

Data in tables is changed when the DML statements are executed in the database for inserting, updating, or deleting records. SQL Server provides the CDC method to track the changes made by the DML statements. For example, if you perform an update operation on a table, CDC will provide you with the data before and after the update operation.

To use CDC to track changes, you need to perform the following steps:

- ❑ Enable CDC on a database.
- ❑ Enable CDC on a table.
- ❑ Track changes.

NOTE *If you enable CDC on an existing table, its database schema does not change. The table must have a primary key before applying CDC method.*

Enabling CDC on a Database

Before tracking the changes made by the DML statements, you need to enable CDC on the database by executing the sp_cdc_enable_db stored procedure. For example, to enable CDC on the AdventureWorks database, you need to execute the following statements:

```
USE AdventureWorks
EXEC sys.sp_cdc_enable_db
```

You can verify whether CDC is enabled by executing the following query:

```
SELECT        is_cdc_enabled        FROM
sys.databases
WHERE name = 'AdventureWorks'
```

If the value in the is_cd_enabled column is 1, then the AdventureWorks database is enabled for CDC. When CDC is enabled on a database, a new user named cdc, a schema named cdc, and the following tables are created in the CDC-enabled database:

- **cdc.captured_columns**: Contains the details of all the columns that are part of the CDC-enabled tables.
- **cdc.change_tables**: Contains the information about which tables have been enabled for CDC.
- **cdc.ddl_history**: Contains the details of the DDL statements executed on the CDC-enabled tables.
- **cdc.index_columns**: Contains the details of the indexes associated with the CDC-enabled tables.
- **cdc.lsn_time_mapping**: Contains the details of the transactions, such as begin and end time of the transaction, performed on the CDC-enabled tables.

All the preceding tables are created under the System Tables folder of the CDC-enabled database in the Object Explorer window.

Enabling CDC on a Table

After enabling CDC on the AdventureWorks database, you need to enable it on the table for which you want to capture data changes. The sp_cdc_enable_table stored procedure is used to enable CDC on a table. The syntax of sp_cdc_enable_table stored procedure is:

```
sys.sp_cdc_enable_table
 [ @source_schema= ] 'source_schema',
 [ @source_name= ] 'source_name',
 [ @role_name= ] 'role_name'
```

where,

`[ @source_schema= ] 'source_schema'` specifies the name of the schema to which the source table belongs.

`[ @source_name= ] 'source_name'` specifies the name of the source table on which CDC is to be enabled.

`[ @role_name= ] 'role_name'` specifies the name of the database role used to access the changed data. If you do not specify any role by passing a NULL value, access to the changed data will be available to everyone.

For example, to enable CDC on the HumanResources.Employee table, you need to execute the following statement:

```
EXEC sys.sp_cdc_enable_table
 @source_schema= 'HumanResources',
 @source_name= 'Employee',
 @role_name = 'NULL'
```

After the HumanResources.Employee table is enabled for CDC, a new table named cdc.HumanResources_Employee_CT is created in the system tables of the AdventureWorks database. This table will store the details of the DML operations performed on the HumanResources.Employee table.

Tracking Changes

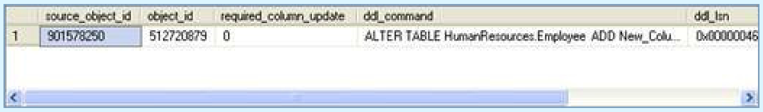
After enabling CDC on the HumanResources.Employee table, you can track the changes made in this table. To modify the HumanResources.Employee table, you can execute the following statement:

```
ALTER TABLE HumanResources.Employee
ADD New_Column int NULL;
```

The preceding statement adds a new column to the HumanResources.Employee table. The details of this statement are stored in the cdc.ddl_history table. You can view the details of the preceding statement by executing the following query:

```
SELECT * FROM cdc.ddl_history
```

The following figure displays the output of the preceding query.



	source_object_id	object_id	required_column_update	ddl_command	ddl_lsn
1	901578250	512720879	0	ALTER TABLE HumanResources.Employee ADD New_Colu...	0x00000046

The cdc.ddl_history Table

Using Change Tracking

You can also use the change tracking method to track the changes made in the tables. However, this method does not provide the information about the historical data. It only captures the fact that rows in a table were changed, but does not capture the data that was changed. This enables you to determine which rows have changed in the table. However, you need to query the table itself to retrieve the latest data. This method is useful for those applications that just require the information about the changed rows, and do not require historical data.

To use change tracking, you need to perform the following steps:

- Enable change tracking on a database.
- Enable change tracking on a table.
- Implement change tracking.

You must have dbcreator server role to implement *change tracking on a database.*

Enabling Change Tracking on a Database

Before tracking the changes using the change tracking method, you need to enable it on the database by using the ALTER DATABASE statement. The syntax of the ALTER DATABASE statement is:

```
ALTER DATABASE database_name
CHANGE_TRACKING = {ON |OFF}
[SET AUTO_CLEANUP = {ON | OFF}]
[SET         CHANGE_RETENTION         =
retention_period   {DAYS   |   HOURS   |
MINUTES}]
```

where,

`database_name` is the name of the database on which change tracking is enabled.

`CHANGE_TRACKING` option enables or disables change tracking on the database. If this option is set to ON, the database is enabled for change tracking. However, if set to OFF, the database is disabled for change tracking.

`SET AUTO_CLEANUP` option if set to ON automatically removes the change tracking information after the specified retention period. However, if this option is set to OFF, the change tracking data is not removed from the database.

`SET CHANGE_RETENTION = retention_period` is an integer value to specify the minimum time period for which the change tracking information is kept in the database. The default retention period is 2 days. The minimum retention period is 1 minute.

For example, to enable change tracking on the AdventureWorks database, you need to execute the following statement:

```
ALTER DATABASE AdventureWorks
SET CHANGE_TRACKING = ON
(CHANGE_RETENTION     =     3     DAYS,
AUTO_CLEANUP = ON)
```

In the preceding statement, the retention period is specified as 3 days. The AUTO_CLEANUP parameter is set as ON to remove the change tracking after 3 days. You can verify whether change tracking is enabled on the database by using the following query:

```
SELECT                *                FROM
sys.change_tracking_databases
```

The preceding code uses the sys.change_tracking_databases DMV that displays the database ID of the database on which the change tracking method is enabled. Further, you can view the name of the database by executing the following query:

```
SELECT  name  FROM  sys.databases  where
database_id = 17
```

Enabling Change Tracking on a Table

After enabling change tracking on the database, you need to enable it on tables. For example, you can enable change tracking on the HumanResources.Employee table by executing the following statement:

```
ALTER TABLE HumanResources.Employee
ENABLE CHANGE_TRACKING
```

You can disable change tracking on the HumanResources.Employee table by executing the following statement:

```
ALTER TABLE HumanResources.Employee
DISABLE CHANGE_TRACKING
```

Implementing Change Tracking

After enabling change tracking on a table, you can track the changes made in this table.

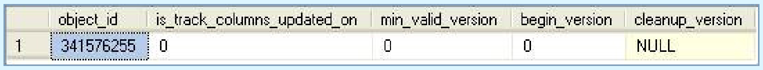
You can view the details of the tables on which track changes have been applied by executing the following query:

```
SELECT              *              from
sys.change_tracking_tables
```

The preceding query uses the sys.change_tracking_tables DMV which returns one row for each change tracking-enabled table in the current database. For example, you can enable track changing on the Person.Contact table by executing the following statement:

```
ALTER TABLE Person.Contact
ENABLE CHANGE_TRACKING
```

After this if you execute the preceding SELECT query again, the output will be displayed as shown in following figure.



	object_id	is_track_columns_updated_on	min_valid_version	begin_version	cleanup_version
1	341576255	0	0	0	NULL

The Output Derived by Enabling Track Changes on Contact Table

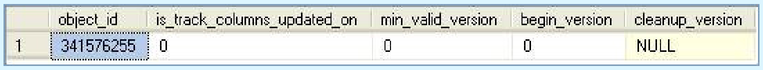
In the preceding figure, the min_valid_version column depicts the minimum valid version of change tracking information that is available for the table. The begin_version column depicts the version of the database when change tracking began for the table.

NOTE *The data in the preceding figure may differ depending upon the operations performed on the table in the database.*

You can verify the name of the table by executing the following query:

```
SELECT OBJECT_ID, NAME FROM SYS.TABLES
WHERE OBJECT_ID = 341576255
```

The following figure displays the object ID and the name of the table on which change tracking is enabled.



	object_id	is_track_columns_updated_on	min_valid_version	begin_version	cleanup_version
1	341576255	0	0	0	NULL

The Output Derived by Executing the Preceding Query

Then, you can use the CHANGETABLE function to view the changes made to the table. The syntax of the

CHANGETABLE function is:

```
CHANGETABLE (CHANGES table, version_number) [AS] table_alias
```

where,

`table` specifies the name of the table for which the changed data needs to be retrieved.

`version_number` specifies the version number. This number specifies the state of the table before performing a particular operation on the table. For a single operation performed on a change tracking-enabled table, the version number is incremented by 1. For example, after enabling change tracking on a table, the version number of the table is 0. You inserted a record in the table. This insert operation changes the version number of the table to 1. Now, if you perform an update operation, the version number will be incremented to 2. If you pass 0 as the version number, you will get all the changes that were done in the table after the 0 version. However, if you pass 1 as the version number, you will get all the changes that were done in the table after the 1 version.

`table_alias` is used to provide a name for the results that are returned.

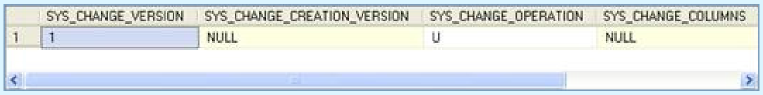
For example, you can execute the following statement to modify the Person.Contact table:

```
UPDATE Person.Contact
SET MiddleName = 'S.'
WHERE ContactID = 1
```

Then, execute the following query:

```
SELECT * FROM CHANGETABLE (CHANGES Person.Contact, 0) AS CT
```

The following figure displays the output of the preceding query.



	SYS_CHANGE_VERSION	SYS_CHANGE_CREATION_VERSION	SYS_CHANGE_OPERATION	SYS_CHANGE_COLUMNS
1	1	NULL	U	NULL

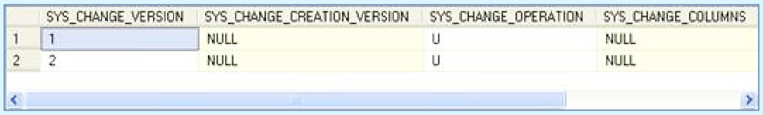
The Output Derived by Executing the Preceding Query

In the preceding figure, the SYS_CHANGE_VERSION depicts the current version of the table after performing the update operation. While retrieving changes from the table by using the CHANGETABLE function, the value of the version number must be greater than or equal to the version depicted in the min_valid_version column of the sys.change_tracking_tables table.

Modify the Person.Contact table again by executing the following statement:

```
UPDATE Person.Contact
SET MiddleName = 'S.'
WHERE ContactID = 2
```

Then, again execute the preceding SELECT query, the output will display all the changes that were done after the 0 version, as shown in the following figure.



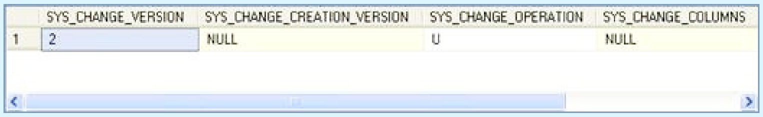
	SYS_CHANGE_VERSION	SYS_CHANGE_CREATION_VERSION	SYS_CHANGE_OPERATION	SYS_CHANGE_COLUMNS
1	1	NULL	U	NULL
2	2	NULL	U	NULL

The Output Derived by Executing the Preceding Query

However, if you want only the changes made after the version number 1, execute the following query:

```
SELECT * FROM CHANGETABLE (CHANGES Person.Contact, 1) AS CT
```

The following figure displays the output of the preceding query.



	SYS_CHANGE_VERSION	SYS_CHANGE_CREATION_VERSION	SYS_CHANGE_OPERATION	SYS_CHANGE_COLUMNS
1	2	NULL	U	NULL

The Output Derived by Executing the Preceding Query

Activity 9.2: Tracking Database Changes

Summary

In this chapter, you learned that:

- ❑ Performance of an SQL Server is measured by the time and resources consumed in responding to queries of the users.
- ❑ An event is any action generated within an instance of SQL Server, such as the log on status, execution of T-SQL statements, start or end of stored procedures, error returned by SQL Server, or lock acquired or released by an object.
- ❑ A trace captures and stores details of events you need to monitor for an instance of SQL Server.
- ❑ A template defines the standard set of events to be monitored in a trace.
- ❑ DMVs are a set of views and table-valued functions that return server state information to the administrator.
- ❑ The frequently used DMVs are:
  - sys.dm_exec_sessions
  - sys.dm_exec_connections
  - sys.dm_exec_requests
- ❑ Database Tuning Advisor is a tool that checks the performance of a workload and suggests how to improve the performance of the server.
- ❑ Statistics in SQL Server refers to information that the server collects about the distribution of data in columns and indexes.
- ❑ SQL Server can automatically create and update statistics. This feature is set to ON by default.
- ❑ Statistics can be created on specific columns of a table or view by using the CREATE STATISTICS

statement.

- ❑ Statistics can be updated by using the UPDATE STATISTICS statement.
- ❑ Statistics IO displays information regarding the amount of disk activity generated by T-SQL statement.
- ❑ SQL Server provides the CDC method to track the changes made by the DML statements.
- ❑ CDC provides the detailed information about the data that got changed.
- ❑ You can use CDC to track changes by performing the following steps:
  - Enabling CDC on a database
  - Enabling CDC on a table
  - Tracking changes
- ❑ Before tracking the changes made by the DML statements, you need to enable CDC on the database by executing the sp_cdc_enable_db stored procedure.
- ❑ The change tracking method only captures the fact that rows in a table were changed, but does not capture the data that was changed.

Reference Reading

Monitoring Performance

Reference Reading: Books	***Reference Reading: URLs***
Professional Microsoft SQL Server 2012 Administration by Adam Jorgensen, Steven Wort, and Ross LoForte	*http://msdn.microsoft.com/en-us/library/ms189081.aspx*

Optimizing Performance

Reference Reading: Books	***Reference Reading: URLs***
Microsoft SQL Server 2012 Performance Tuning Cookbook by R. Shah and B. Thaker	*http://blog.sqlauthority.com/2012/04/20/sql-server-performance-tuning-part-2-of-2-analysis-detection-tuning-and-optimizing/*

Tracking Data Changes

Reference Reading: Books	***Reference Reading: URLs***
Microsoft SQL Server 2012 Bible by Adam Jorgensen, Jorge Segarra, and Patrick LeBlanc	*http://msdn.microsoft.com/en-us/library/bb933994.aspx*

Glossary

A

Atomicity

A property of a transaction. It states that either all the data modifications are performed or none of them.

Atomicity, Consistency, Isolation, and Durability (ACID)

The four properties that a transaction must posses to work together as a single logical unit of work.

C

Clustered Index

An index that sorts and stores the data rows in the table based on their key values.

Column

Vertical slice of a table. Column is defined to be of a single data type.

Composite Index

An index based on one or more columns.

Consistency

A property of a transaction. It states that all the data should be in a consistent state after the transaction is completed.

Contract

An agreement between the participant applications defining the messages that will be exchanged between them.

Correlated Subquery

A subquery in which the information retrieved by the subquery is referenced by the outer, main query. A correlated subquery cannot be stand alone, it depends on the outer query.

D

Deadlock Victim

The transaction that is canceled by the SQL Server to resolve the deadlock.

Deleted Table

A table that contains all records that have been deleted from the trigger table.

Durability

A property of a transaction. It states that any change in data by a completed transaction remains permanently in effect in the system.

E

Exceptions

The errors that occur at run time.

Execution Plan

The SQL Server compiles the statements of a batch into a single executable unit called an execution plan while executing batches.

H

Hyper Text Transfer Protocol (HTTP)

HTTP is the protocol used for communication over the Internet.

I

Index

An internal table created to speed up queries and searches in a database.

Inserted Table

A table that contains a copy of all records that are inserted in the trigger table.

Isolation

A property of a transaction. It states that any data modification made by concurrent transactions must be isolated from the modifications made by other concurrent transactions.

Isolation Level

Plan used to specify the rights other transactions will have on the data being modified by a transaction.

M

Magic Tables

These are the conceptual tables and are similar in structure to the table on which the trigger is defined.

N

Nonclustered Index

An index in which the logical order of the index does not match the physical stored order of the rows in a table.

P

Parsing

The process of compiling and breaking the components of stored procedure into various parts.

Population

The process of storing values in the full-text index.

Q

Qualifier

A prefix used to identify a column of a particular table. For example, in "Student.sname" Student is the table qualifier.

R

Resolving

The process of checking the existence of the referred objects, such as tables and views.

S

Service Program

A program that provides logic to the service.

Shredding

The process of transforming XML data into a rowset.

Simple Object Access Protocol (SOAP)

A standard communication protocol to interchange information in a structured format within a distributed environment.

Stopwords

Words that are used in a search, but are excluded from the search string such as a, an, the, and are.

Stored Procedure

A precompiled object stored in the database.

T

Timeout

The time period for which a transaction waits for the lock to be released on a blocked resource.

U

User-Defined Functions (UDF)

A user-defined database object that contains a set of T-SQL statements, accepts parameters, performs an action, and returns the result of that action as a value.

V

View

A virtual table that provides access to a subset of columns from one or more tables.

X

XML Index

An index that is created on columns storing XML data values.

Appendix

Extensible Markup Language (XML)

XML is a markup language that is used to describe the structure of data in a standard hierarchical manner. The structure of the documents containing data is described with the help of tags contained in the document. An XML document is attached to a Document Type Definition (DTD) or XML Schema that describes its structure.

Components of an XML Document

An XML document is composed of a number of components that can be used to represent information in a hierarchical order. These components are:

- ❑ Processing Instruction
- ❑ Tags
- ❑ Elements
- ❑ Content
- ❑ Attributes
- ❑ Comments

Processing Instruction

An XML document usually begins with the XML declaration statement also called the Processing Instruction (PI). The PI provides information regarding the way in which the XML file should be processed. The PI statement can be written as shown in the following code snippet:

```
<?xml version="1.0" encoding="UTF-8"?>
```

In the preceding code snippet, the PI states that XML version 1.0 is used.

Tags

Tags are used to specify a name for a given piece of information. It is a means of identifying data. Data is marked-up by using tags. A tag consists of opening and closing angular bracket (<>). These brackets enclose the name of the tag. Tags usually occur in pairs. Each pair consists of a start tag and an end tag. The start tag only contains the name of the tag while the end tag includes a forward slash (/) before the name of the tag. Consider the following code snippet:

```
<P>Nick Shaw</P>
```

In the preceding code snippet, `<P>` is a predefined HTML tag or mark-up, which encloses the name of an employee. Here, `<P>` is the start tag and `</P>` is the end tag. As XML allows you to create your own tags, the same information can be stored as shown in the following code snippet:

```
<EMP_NAME>Nick Shaw</EMP_NAME>
```

In the preceding code snippet, `<EMP_NAME>` is a new tag created by using XML. This tag is used to store the name of the employee.

Elements

Elementsare the basic units that are used to identify and describe data in XML. They are the building blocks of an XML document. Elements are represented by using tags.

XML allows you to provide meaningful names to elements, which helps improve the readability of the code. Meaningful element names also enable easy identification of the element content. For example, an `Authorname` element can be created for an application. Consider the following code snippet:

```
<Authorname>John Smith</Authorname>
```

In the preceding code snippet, the Authorname element is used to store the names of authors. In this case, the element name provides a description of the content within the tags. The Authorname element is represented by using the `<Authorname>` and `</Authorname>` tags, and the author name is enclosed between these tags.

An XML document must always have a root element. A root element contains all other elements in the document. Consider the following code snippet:

```
<?xml version=“1.0”?>
<AUTHORS>
  <AUTHOR>
    <FIRSTNAME> John </FIRSTNAME>
    <LASTNAME> Norton </LASTNAME>
  </AUTHOR>
</AUTHORS>
```

In the preceding code snippet, the AUTHORS element contains all other elements in the XML document and is therefore called the root element. An XML document can contain only one root element and all other elements are embedded within the opening and closing tags of the root element.

Content

The information that is represented by the elements of an XML document is referred to as the content of that element. For example, consider the following code snippet:

```
<BOOKNAME>The Painted House</BOOKNAME>
```

In the preceding code snippet, the name of the book The Painted House is the content of the BOOKNAME element.

Elements can contain only textual information. Such elements are said to have character or data content. Consider the following code snippet:

```
<BOOKNAME>The Painted House</BOOKNAME>
```

In the preceding code snippet, the BOOKNAME element contains only textual information and is therefore said to have character or data content.

Attributes

Attributes provide additional information about the elements for which they are declared. An attribute consists of a name-value pair. Consider the following code snippet:

```
<PRODUCTNAME PRODID="P001">Barbie
Doll</PRODUCTNAME>
```

In the preceding code snippet, the element PRODUCTNAME has an attribute called PRODID. The value of this attribute is set to P001. The attribute name and the attribute value are specified within the opening tag of the PRODUCTNAME element.

Elements can have one or more attributes. Attributes or attribute values can be either mandatory or optional.

When you create the structure of an XML document, you must decide whether a specific piece of information is to be represented as an element or an attribute. In general, an element is used to represent a definable unit. An attribute is used to represent data that further qualifies the element. For example, an element called font could have an attribute called color that is used to specify the color of the font. In this case, the color attribute further qualifies the font element. Therefore, color is represented as an attribute and not as an element.

You may decide to represent information either by using elements or attributes. There are no rules governing this decision. However, you may consider the following guidelines while deciding whether to represent information by using an element or an attribute:

- ❑ If the data must be displayed, you can represent it as an element. In general, element attributes are used for intangible, abstract properties such as ID.
- ❑ If the data must be updated frequently, it is better represented as an element because it is easier to edit elements than attributes with XML editing tools. For example, in the case of products, the quantity on hand needs to be updated when a transaction takes place. Therefore, you can represent the quantity by using an element.
- ❑ If the value of a piece of information must be checked frequently, it may be represented as an attribute. This is because an XML processor can check the content of an attribute value more easily than it can check the content of an element. For example, you often retrieve details about products based on their IDs and categories. Therefore, you can represent product ID and category as attributes.

Comments

Comments are statements that are used to explain the code. They are also used to provide documentation information about an XML file or even the application to which the file belongs. When the code is executed, comment entries are ignored by the parser.

Comments are not essential in an XML file. However, it is good programming practice to include comments along with the code. This ensures that you can easily understand the code.

Comments are created by using an opening angular bracket followed by an exclamation mark and two hyphens (<!--). This is followed by the text that comprises the comment. The comment entry can be closed by using two hyphens followed by a closing angular bracket (-->). The following code snippet illustrates the use of a comment in an XML document:

```
<!--PRODUCTDATA is the root element-->
```

The text contained within a comment entry cannot have two hyphens that occur consecutively, as shown in the following code snippet:

```
<!--PRODUCTDATA is the --root
element-->
```

This is an invalid comment because the text enclosed by the opening and closing tags includes two consecutive hyphens.

Defining the Structure of an XML Document

To represent data in a consistent format, it needs to be given a meaningful structure. A well-formed document in XML may not necessarily have a meaningful structure. Anyone can create a well-formed structure, but this structure will be specific only to the XML document in which it is created. It cannot be applied consistently across multiple documents.

By defining the role of each element in a formal model, users of XML can ensure that each component of the document appears in a valid place in the XML document. DTD or XML Schemas can be used to define the structure of an XML document.

Document Type Definition

A DTD defines the structure of the content of an XML document, thereby allowing you to store data in a consistent format. It specifies the elements that can be present in the XML document, attributes of these elements, and their arrangement with relation to each other. A DTD also allows you to specify whether an element is optional or mandatory.

Creating a DTD is similar to creating a table in a database. In DTDs, you specify the structure of the XML document by declaring elements to denote the data. This is similar to creating columns in a table. You can also specify whether providing a value for the element is mandatory or optional. You can then store the data in an XML document that conforms to the DTD for that application. This is similar to adding records in a table.

XML allows you to create your own DTDs for applications. This gives you complete control over the process of checking the content and the structure of XML documents created for an application. The process of checking is called validation. XML documents that conform to a DTD are considered valid documents.

The following code is a sample XML DTD for the PRODUCTDATA data structure:

```
<!ELEMENT PRODUCTDATA (PRODUCT)+>
<!ELEMENT PRODUCT (PRODUCTNAME,
DESCRIPTION, PRICE, QUANTITY)>
<!ELEMENT PRODUCTNAME (#PCDATA)>
<!ELEMENT DESCRIPTION (#PCDATA)>
<!ELEMENT PRICE (#PCDATA)>
<!ELEMENT QUANTITY (#PCDATA)>
<!ATTLIST PRODUCT PRODUCTID ID
#REQUIRED
CATEGORY (BOOKS | TOYS) "TOYS">
```

XML Schema

An XML schema is used to define the structure of an XML document. A schema defines the list of elements and attributes that can be used in an XML document. In addition to the list of elements, an XML schema also specifies the order in which these elements appear in the XML document and their data types.

A sample XML schema for the PRODUCTDATA data structure must be given as shown in the following code:

```
<xsd:schema xmlns:xsd=“http://
www.w3.org/2001/XMLSchema”>
<xsd:element name="PRODUCTDATA"
type="prdata"/>
  <xsd:complexType name="prdata">
  <xsd:sequence>
      <xsd:element name="PRODUCT"
type="prdt"
      maxOccurs="unbounded"/>
  </xsd:sequence>
</xsd:complexType>
<xsd:complexType name="prdt">
      <xsd:sequence>
      <xsd:element name="PRODUCTNAME"
type="xsd:string"/>
      <xsd:element name="DESCRIPTION"
type="xsd:string"/>
      <xsd:element name="PRICE"
type="xsd:positiveInteger"/>
<xsd:element name="QUANTITY"
type="xsd:nonNegativeInteger"/>
    </xsd:sequence>
  </xsd:complexType>
</xsd:schema>
```

As shown in the preceding code, the declaration of an XML schema begins with the schema element. The schema element uses the xmlns attribute to specify the namespace associated with the document.

A namespace is a string that is used to refer to Unique Resource Identifiers (URI), such as "http://www.microsoft.com" and "http://www.w3.org/2001/XMLSchema", where the definition of the keywords and the data types used in the schema is stored.

For example, in the preceding code, the statement <xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"> specifies that xsd will be used as a prefix to identify the elements that are defined at the location http://www.w3.org/2001/XMLSchema. This is the location where the schema instructions are defined by the World Wide Web Consortium (W3C).