Cognizant™
Passion for making a difference

Handout: SQL Server 2005

Version: SQLSERVER2005/Handout/0408/1.0
Date: 30-04-08

Cognizant
500 Glen Pointe Center West
Teaneck, NJ 07666
Ph: 201-801-0233
www.cognizant.com

TABLE OF CONTENTS

Introduction

About this Module

This module provides the developer with the SQL Server 2005 fundamentals and skills to code using TSQL.

Target Audience

This module is designed for entry level developer using SQL Server 2005.

Module Objectives

After completing this module, you will be able to understand:

- ❑ Create data types and tables
- ❑ Implement data integrity
- ❑ Perform programming SQL Server
- ❑ Create databases
- ❑ Manage databases
- ❑ Implement Cursors, Stored Procedures, Functions, Indexes, and Triggers
- ❑ Managing Transactions and Locks
- ❑ XML Support in SQL Server

Pre-requisite

Basic knowledge of general RDBMS concepts

Session 02: SQL Server 2005 Overview

Learning Objectives

After completing this session, you will be able to:

- ❑ Describe the architecture of SQL Server 2005
- ❑ Implement various SQL Server 2005 tools
- ❑ Apply SQL Server 2005 security mechanisms

SQL Server Overview

Microsoft SQL Server 2005 is a database and data analysis platform for large-scale online transaction processing (OLTP), data warehousing, and e-commerce applications.

The Database Engine is the core service for storing, processing, and securing data. The Database Engine provides controlled access and rapid transaction processing to meet the requirements of the most demanding data consuming applications within your enterprise. The Database Engine also provides rich support for sustaining high availability.

SQL Server is a RDBMS (Relational Database Management System) that:

- ❑ Manages data storage for transactions and analysis
- ❑ Stores data in a wide range of data types, including text, numeric, XML and large objects
- ❑ Responds to request form client applications
- ❑ Applies T-SQL (Transact-SQL), XML (eXtensible Markup Language) or other SQL Server commands to send requests between a client application and SQL Server

The RDBMS component of SQL Server is responsible for:

- ❑ Maintaining the relationships among data in a database
- ❑ Ensuring that data is stored correctly and that the rules defining the relationships among data are not violated
- ❑ Recovering all data to a point of known consistency, in the event of a system failure

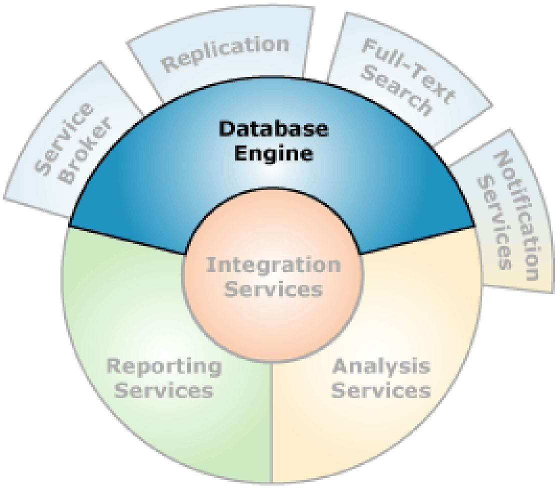
SQL Server 2005 Architecture and Components

SQL Server 2005 is more than just a database management system. It includes multiple components and services that make it a comprehensive platform for enterprise applications.

SQL Server 2005 is made up of the following components:

- ❑ Relational database engine
- ❑ Analysis Services
- ❑ SQL Server Integration Services (SSIS)
- ❑ Notification Services
- ❑ Reporting Services

- ❑ Service Broker
- ❑ Replication
- ❑ Full-text search



Relational database engine: The SQL Server relational database engine is at the heart of SQL Server 2005 and provides a high-performance, scalable, secure environment for storing, retrieving, and modifying data in relational or eXtensible Markup Language (XML) format

Analysis Services: Provides the basis of a powerful business intelligence solution that supports Online Analytical Processing (OLAP) applications and data mining

SQL Server Integration Services (SSIS): An engine for building data import and export solutions and performing transformations on data as it is transferred

Notification Services: A framework for solutions in which subscribers are sent notifications when specific events occur and notifications can be generated efficiently and sent to multiple device types

Reporting Services: Applied to extract data from SQL Server and generate reports

Service Broker: A queuing mechanism for reliable, transactional message based communication between software services

.NET Common Language Run time (CLR): Hosted in SQL Server, making it possible to implement database solutions applying managed code written in a .NET language such as Microsoft Visual C#® .NET or Microsoft Visual Basic® .NET.

Native HTTP Support: Enables client applications to connect to HTTP endpoints within SQL Server without requiring Internet Information Services (IIS).

Replication: A set of technologies for copying and distributing data and database objects from one database or server to another and then synchronizing between databases to ensure consistency

Full-text search: Enables fast and flexible indexing for queries based on keyword of text data stored in a SQL Server database.

SQL Server 2005 Tools

Microsoft SQL Server 2005 provides the tools to design, develop, deploy, and administer relational databases, Analysis Services cubes, data transformation packages, replication topologies, reporting servers, and notification servers.

SQL Server 2005 tools are as follows:

- ❑ SQL Server Management Studio
- ❑ SQL Server Business Intelligence Development Studio
- ❑ SQL Server Profiler
- ❑ SQL Server Configuration Manager
- ❑ SQL Server Surface Area Configuration
- ❑ Database Engine Tuning Advisor
- ❑ Command Prompt Utilities

SQL Server Management Studio

SQL Server Management Studio is an integrated environment for accessing, configuring, managing, and administering all components of SQL Server. Management studio replaces Enterprise Manager and Query Analyzer of previous SQL Server releases.

Microsoft SQL Server 2005 provides the SQL Server Management Studio for administering and designing SQL Server databases within the context of a script project. SQL Server Management Studio includes designers, editors, guides and wizards to assist users in developing, deploying and maintaining databases.

SQL Server Management Studio is a suite of administrative tools for managing the components belonging to SQL Server. This integrated environment allows users to perform a variety of tasks, such as backing up data, editing queries, and automating common functions within a single interface.

Tools that are now incorporated into the SQL Server Management Studio include:

- ❑ Code Editor is a rich script editor for writing and editing scripts. The Code Editor replaces the Query Analyzer included in previous releases of SQL Server.
- ❑ SQL Server Management Studio provides four versions of the Code Editor, namely the SQL Query Editor, MDX (Multidimensional Expressions)Query Editor, XML Query Editor, and SQL Mobile Query Editor.
- ❑ Object Explorer for locating, modifying, scripting or running objects belonging to instances of SQL Server and also to browse different servers and for viewing logs.
- ❑ Template explorer is applied to locate an existing template to be implemented in the code editor or to add a custom template.

Business Intelligence Development Studio

Business Intelligence Development Studio is Microsoft Visual Studio 2005 with additional project types that are specific to SQL Server 2005 business intelligence.

Business Intelligence Development Studio is the primary environment that you will apply to develop business solutions that include Analysis Services, Integration Services, and Reporting Services projects.

SQL Server Profiler

SQL Server Profiler is a tool that captures SQL Server 2005 events from a server. The events are saved in a trace file that can later be analyzed or applied to replay a specific series of steps when trying to diagnose a problem.

SQL Server Configuration Manager

SQL Server Configuration Manager is a tool:

- To manage the services associated with SQL Server
- To configure the network protocols applied by SQL Server
- To manage the network connectivity configuration from SQL Server client computers

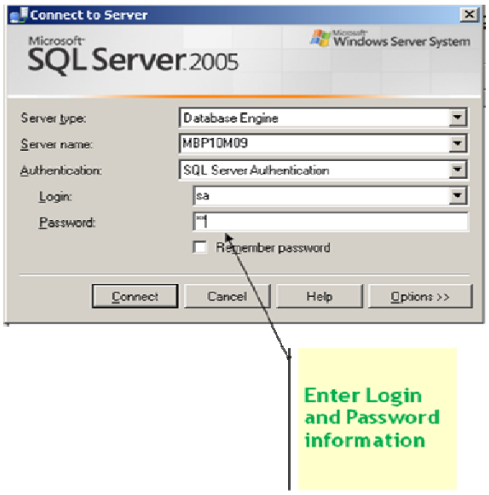
Command Prompt Utilities tools

Command Prompt Utilities are tools for moving bulk data, running scripts, managing Notification Services, and so on.

Some of Command Prompt Utilities tools as follows:

- **bcp Utility:** Applied to copy data between an instance of Microsoft SQL Server and a data file in a user-specified format.
- **dta Utility:** Applied to analyze a workload and recommend physical design structures to optimize server performance for that workload.
- **osql Utility:** Allows you to enter Transact-SQL statements, system procedures, and script files at the command prompt.
- **sqlcmd Utility:** Allows you to enter Transact-SQL statements, system procedures, and script files at the command prompt.

SQL Server Security



Server type: Database Engine
Server name: MBP10M09
Authentication: SQL Server Authentication
Login: sa
Password: *
☐ Remember password

Enter Login and Password information

SQL Server supports two forms of authentication:

- Windows Authentication
- SQL Server Authentication

Windows Authentication

Windows Authentication is used to connect to SQL Server, Microsoft Windows is completely responsible for authenticating the client. In this case, the client is identified by its Windows user account.

SQL Server Authentication

SQL Server Authentication is used when SQL Server authenticates the client by comparing the client-supplied user name and password to the list of valid user names and passwords maintained within SQL Server.

Database

SQL Server service running on a machine is called an “Instance”. Multiple instances can run on the same machine and each instance manages multiple databases to store data.

A database can span multiple files across multiple disks.

System Databases

System database always stores information about SQL Server as a whole. SQL Server applies system database to operate and manage the system.

SQL Server system databases are as follows:

- Master
- Model
- MSDB
- Tempdb
- Resource

Master: Records all the system-level information for an instance of SQL Server. This includes instance-wide metadata such as logon accounts, endpoints, linked servers, and system configuration settings. Also, master is the database that records the existence of all other databases and the location of those database files. Additionally, master records the initialization information for SQL Server. Therefore, SQL Server cannot start if the master database is unavailable.

Model: Applied as the template for all databases created on the instance of SQL Server. Modifications made to the model database, such as database size, collation, recovery model, and other database options, are applied to any databases created afterward.

MSDB: Applied by SQL Server Agent for scheduling alerts and jobs

Tempdb: Workspace for holding temporary objects or intermediate result sets.

Resource: A read-only database that contains copies of all system objects that ship with Microsoft SQL Server 2005

Transact SQL

The features of Transact SQL are:

- Powerful and unique superset of the SQL standard
- Follows ANSI SQL- 92 specification
- Specific to SQL Server databases only

The Transact SQL statements classified as:

- Data Definition Language Statements (DDL)
- Data Control Language Statements (DCL)
- Data Manipulation Language Statements (DML)

DDL statements: applied to build and modify the structure of your tables and other objects in the database

- Define the database objects with the following statements:
  - CREATE object_name
  - ALTER object_name
  - DROP object_name
- User Must have the appropriate permissions to execute DDL Statements

```
CREATE TABLE MyCustomer
(
cust_id int, company varchar(40),
contact varchar(30),
phone char(12)
 )
GO
```

DCL statements: applied to control access to data in a database

- Set or change permissions with following statements:
  - GRANT
  - DENY
  - REVOKE
- User Must have the appropriate permissions to execute DCL Statements

```
GRANT SELECT ON Mycustomer TO public
GO
```

DML statements: Applied to change data or retrieve information

- Manipulate the data using following statements:
  - SELECT
  - INSERT
  - UPDATE
  - DELETE
- User Must have the appropriate permissions to execute DML Statements

```
SELECT categoryid, productname, productid, unitprice
FROM products
GO
```

Data types

In SQL Server 2005, each column, local variable, expression, and parameter has a related data type. A data type is an attribute that specifies the type of data that the object can hold: integer data, character data, monetary data, data and time data, binary strings, and so on.

SQL Server supplies a set of system data types that define all the types of data that can be used with SQL Server. You can also define your own data types in Transact-SQL.

System Data Types

Data types in SQL Server 2005 are organized into the following categories:

Exact numerics	Unicode character strings
Approximate numerics	Binary strings
Date and time	Other data types
Character strings	

Exact Numerics

bigint	decimal
int	numeric
smallint	money
tinyint	smallmoney
bit	

Approximate Numerics

float	real

Date and Time

datetime	smalldatetime

Character Strings

char	text
varchar	

Unicode Character Strings

nchar	ntext
nvarchar	

Binary Strings

binary	image
varbinary	

Other Data Types

cursor	timestamp
sql_variant	uniqueidentifier
table	xml

In SQL Server 2005, based on their storage characteristics, some data types are designated as belonging to the following groups:

Large value data types: varchar(max), nvarchar(max), and varbinary(max)

- ❑ varchar (max):
  - o Character data types of variable length
  - o Applied when the sizes of the column data entries vary considerably and the size might exceed 8,000 bytes

- nvarchar (max):
  - Unicode character data types of variable length
  - Applied when the sizes of the column data entries vary considerably and the size might exceed 8,000 bytes
- varbinary (max):
  - Binary data types of variable length
  - Applied when the column data entries exceed 8,000 bytes

Large object data types: text, ntext, image, varchar(max), nvarchar(max), varbinary(max), and xml

- xml:
  - Used to store XML data.

User Defined Data Types

User defined or alias data type is a custom data type based on a data type supplied by system applied for common data elements with a specific format

Created by applying the **CREATE TYPE** statement:

```
Use AdventureWorks
CREATE TYPE dbo.EmailAddress
FROM nvarchar(50)
NULL
GO
```

Summary

- SQL Server is a RDBMS product from Microsoft.
- SQL Server supports Windows and SQL authentication mechanisms.
- Multiple SQL Server Instances can be deployed on the same machine.
- The Enterprise Manager and Query Analyzer are the most important SQL Server tools replaced with Management Studio in SQL Server 2005.
- SQL Server uses Command Line Utility tools such as **bcp, dta, sqlcmd, osql**

Test Your Understanding

1. Command Prompt Utilities Allows you to execute Transact-SQL statements, system procedures, and script files

   a) sqlcmd

   b) bcp

   c) dta

   d) Management Studio

2. Which tool replaces Enterprise Manager and Query Analyzer of previous SQL Server releases?
    a) SQL Server Profiler
    b) SQL Server Configuration Manager
    c) SQL Server Surface Area Configuration
    d) SQL Server Management Studio

Exercises

1. Login to **SQL Server Management Studio** either using SQL Authentication or Windows Authentication and try to execute SQL Queries on Sample **Adventureworks** database.
2. Login to SQL Server using **sqlcmd** utility command and try to execute SQL Queries on Sample **AdventureWorks** database.

Session 07: Implementing Data Integrity

Learning Objectives

After completing this session, you will be able to:

- ❑ Explain the Data Integrity Overview
- ❑ Implement Constraints
- ❑ Compute columns and persisted columns

Data Integrity Overview

An Important step in database planning is deciding the best way to enforce the integrity of the data. Data integrity refers to the consistency and accuracy of data that is stored in a database. In this chapter you will learn about the different types of data integrity in a relational database and the options provide by SQL Server 2005 to enforce data integrity.

Enforcing data integrity guarantees the quality of the data in the database. The various types of data integrity as follows:

- ❑ Entity integrity
- ❑ Domain integrity
- ❑ Referential integrity
- ❑ User-defined integrity

Entity Integrity: Entity integrity defines a row as a unique entity for a particular table. Entity integrity enforces the integrity of the identifier columns or the primary key of a table, through indexes, UNIQUE constraints, PRIMARY KEY constraints, or IDENTITY properties.

Domain Integrity: Domain integrity is the validity of entries for a specific column. You can enforce domain integrity to restrict the type by using data types, restrict the format by using CHECK constraints and rules, or restrict the range of possible values by using FOREIGN KEY constraints, CHECK constraints, DEFAULT definitions, NOT NULL definitions, and rules.

Referential Integrity: Referential integrity preserves the defined relationships between tables when records are entered or deleted. In SQL Server 2005, referential integrity is based on relationships between foreign keys and primary keys or between foreign keys and unique keys, through FOREIGN KEY and CHECK constraints. Referential integrity makes sure that key values are consistent across tables. This kind of consistency requires that there are no references to nonexistent values and that if a key value changes, all references to it change consistently throughout the database.

When you enforce referential integrity, SQL Server prevents users from doing the following:

- ❑ Adding or changing records to a related table if there is no associated record in the primary table.
- ❑ Changing values in a primary table that causes orphaned records in a related table.
- ❑ Deleting records from a primary table if there are matching related records.

User-Defined Integrity: User-defined integrity lets you define specific business rules that do not fall into one of the other integrity categories. All the integrity categories support user-defined integrity. This includes all column-level and table-level constraints in CREATE TABLE, stored procedures, and triggers.

Enforcing Data Integrity

Planning and creating tables requires identifying valid values for the columns and deciding how to enforce the integrity of the data in the columns. SQL Server 2005 provides the following mechanisms to enforce the integrity of the data in a column:

- PRIMARY KEY Constraints
- FOREIGN KEY Constraints
- UNIQUE Constraints
- CHECK Constraints
- DEFAULT Definitions
- Allowing Null Values

PRIMARY KEY Constraints

A PRIMARY KEY constraint defines one or more columns in a table that constitute a primary key. The Primary key uniquely identifies a row in a table and enforces entity integrity of the table. You can create a primary key by defining a PRIMARY KEY constraint when you create or modify a table.

Consider the following facts before you implement a PRIMARY KEY constraint:

- A table can have only one PRIMARY KEY constraint,
- Column that participates in the PRIMARY KEY constraint cannot accept null values. Because PRIMARY KEY constraints guarantee unique data, they are frequently defined on an identity column.
- A PRIMARY KEY constraint creates a unique index for the primary key columns.

Consider using a PRIMARY KEY constraint when:

- One or more columns in a table should uniquely identify each row (entity) in the table.
- One of the columns in the table is identity column.

Syntax:

```
CONSTRAINT {constraint_name}
{ PRIMARY KEY | UNIQUE }
{ ( column_name [ ASC | DESC ] [ ,...n ] ) }
```

Example:

```
CREATE TABLE customer (
cust_id int NOT NULL,
cust_name varchar(30) NOT NULL,
CONSTRAINT PK_Customer_CustID PRIMARY KEY (cust_id) )
```

FOREIGN KEY Constraints

A FOREIGN KEY (FK) is a column or combination of columns that is used to establish and enforce a link between the data in two tables. A FOREIGN KEY constraint enforces referential integrity. The FOREIGN KEY constraint defines a reference to a column with a PRIMARY KEY or UNIQUE constraint in the same, or another, table. The values in the foreign key column must appear in the primary key column, while references exist to the primary key values, they cannot be changed or deleted.

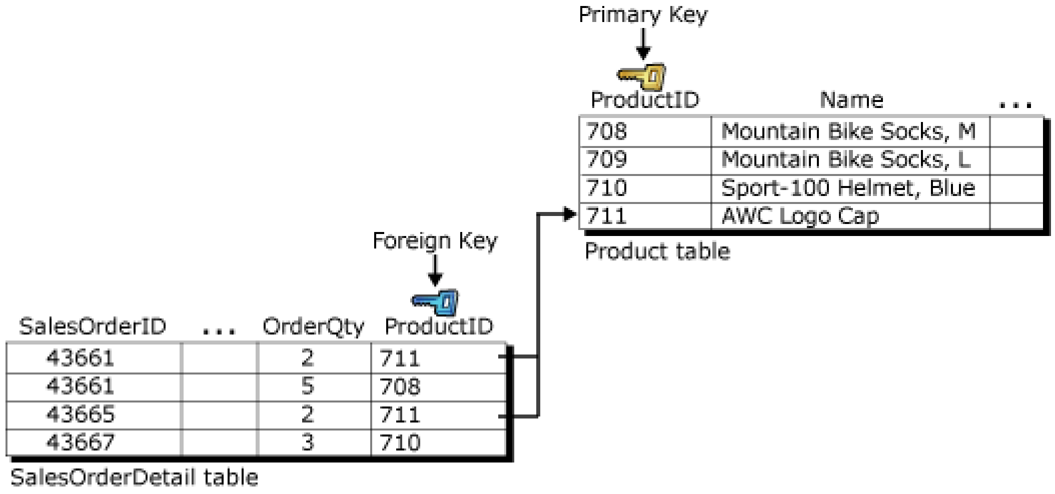
Consider the following facts before you implement a FOREIGN KEY constraint:

- ❑ A FOREIGN KEY constraint provides single-column or multicolumn referential integrity. The number of columns and data types that are specified in the FOREIGN KEY statement must match the number of columns and data types in the REFERENCES clause.
- ❑ Unlike PRIMARY KEY or UNIQUE constraints, FOREIGN KEY constraints do not create indexes automatically.

Consider using a FOREIGN KEY constraint when:

- ❑ The data in one or more columns can hold only values contained in certain columns in the same or another table.
- ❑ The rows in a table should not be deleted while rows in another table depend on them.

For example, with the **Sales.SalesOrderDetail** and **Production.Product** tables in the **AdventureWorks** database, referential integrity is based on the relationship between the foreign key (**ProductID**) in the **Sales.SalesOrderDetail** table and the primary key (**ProductID**) in the **Production.Product** table. This relationship makes sure that a sales order can never reference a product that does not exist in the **Production.Product** table.



ProductID	Name	...
708	Mountain Bike Socks, M	
709	Mountain Bike Socks, L	
710	Sport-100 Helmet, Blue	
711	AWC Logo Cap	

Product table

SalesOrderID	...	OrderQty	ProductID
43661		2	711
43661		5	708
43665		2	711
43667		3	710

SalesOrderDetail table

Example:

```
CONSTRAINT (FK_SaleOrderDetails_Product_ProductID) FOREIGN KEY
(ProductID) REFERENCES Production.Product (ProductID)
```

UNIQUE Constraints

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

Consider the following facts before you implement a UNIQUE constraint:

- ❑ Only one null value can appear in a column with a UNIQUE constraint.
- ❑ A table can have multiple UNIQUE constraints, whereas it can have only a single primary key.
- ❑ A UNIQUE constraint is enforced through the creation of unique index on the specified column or columns. This index cannot allow the table to exceed the 249 limit on nonclustered indexes.
- ❑ The database engine will return an error if you create a UNIQUE constraint on the column that contains data in which duplicate values are found.

Consider using a UNIQUE constraint when:

- ❑ The table contains columns that are not part of the primary key but that, individually or as a unit, must contain unique values.
- ❑ Business logic dictates that the data stored in a column must be unique.
- ❑ The data stored in a column has natural uniqueness on the values it can contain, such as tax numbers or passport numbers.

Example:

```
TaxNumber nvarchar(30) NOT NULL
UNIQUE NONCLUSTERED
```

CHECK Constraints

A CHECK constraint restricts the data values that users can enter into a particular column during INSERT and UPDATE statements. Check constraint can be applied at column and table level. Column-level CHECK constraints restrict the values that can be stored in that column. Table-level CHECK constraints can reference multiple columns in the same table to allow for cross-referencing and comparison of column values.

Consider the following facts before you implement a CHECK constraint:

- ❑ A CHECK constraint verifies data every time you execute an INSERT and UPDATE statement.
- ❑ A CHECK constraint can be any logical (Boolean) expression that returns TRUE OR FALSE.
- ❑ A CHECK constraint cannot contain subqueries.
- ❑ A single column can have multiple CHECK constraints.
- ❑ A CHECK constraint cannot be placed on columns with the **rowversion, text, ntext,** or **image** data types.
- ❑ The Database Consistency Checker (DBCC) CHECKCONSTRAINTS statement will return any rows containing data that violates a CHECK constraint.

Consider using a CHECK constraint when:

- Business logic dictates that the data stored in a column must be a member of a specific set or range of values.
- The data stored in a column has natural limits on the values it can contain.
- Relationships exist between the columns of a table that restrict that values a column can contain.

Syntax:

```
CONSTRAINT {constraint_name} CHECK {Check_expression}
```

Example:

```
CONSTRAINT DepartmentConstraint
CHECK (department_no < 0 AND department_no > 100)
```

DEFAULT Constraints

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

Consider the following facts before you implement a DEFAULT constraint:

- A DEFAULT constraint applies only to insert statements.
- Each column can have only one DEFAULT constraint.
- Columns with the IDENTITY property or the **rowversion** data type cannot have a DEFAULT constraint placed on them.

Consider using a DEFAULT constraint when:

- The data stored in a column has an obvious default value.
- The column does not accept null values
- The column done not enforce unique values.

Example:

```
CREATE TABLE job(… , job_desc VARCHAR (50) DEFAULT ‘N/A’, …)
CREATE TABLE job(… , job_desc VARCHAR (50) CONSTRAINT DF_job_desc
DEFAULT ‘N/A’,
…)
```

Identity property

Identity is a column property that auto generates unique ids for newly inserted rows. Identity, like a primary key, identifies a row uniquely. SQL Server implements row identification using a numeric value. As rows are inserted, SQL Server generates the row value for an identity column by adding an increment to the existing maximum value.

Considerations for using the IDENTITY property:

- A table can have only one column defined with the IDENTITY property, and that column must be defined using the decimal, int, numeric, smallint, bigint, or tinyint data type.

- The seed and increment can be specified. The default value for both is 1.
- The identifier column must not allow null values and must not contain a DEFAULT definition or object.
- The column can be referenced in a select list by using the IDENTITYCOL keyword after the IDENTITY property has been set.
- The OBJECTPROPERTY function can be used to determine if a table has an IDENTITY column, and the COLUMNPROPERTY function can be used to determine the name of the IDENTITY column.
- @@IDENTITY returns the last-inserted identity value.
- SET IDENTITY_INSERT { table } ON is used to temporarily disallow autogeneration. It allows explicit values to be inserted into the identity column of a table. If the value inserted is larger than the current identity value for the table, SQL Server automatically uses the new inserted value as the current identity value

Syntax:

```
IDENTITY [(seed, increment)]
```

Arguments

- **Seed:** Is the value that is used for the very first row loaded into the table.
- **Increment:** Is the incremental value that is added to the identity value of the previous row that was loaded.

Example:

```
CREATE TABLE customer (
cust_id int IDENTITY NOT NULL,
cust_name varchar(30) NOT NULL,
CONSTRAINT PK_Customer_CustID PRIMARY KEY (cust_id) )
```

Computed Columns and Persisted Columns

- A computed column is computed from an expression that can apply other columns in the same table.
- The expression can be a noncomputed column name, constant, function, variable, and any combination of these connected by one or more operators. The expression cannot be a subquery.
- Computed columns are virtual columns that are not physically stored in the table. Their values are recalculated every time they are referenced in a query.
- The SQL Server 2005 Database Engine applies the PERSISTED keyword in the CREATE TABLE and ALTER TABLE statements to physically store computed columns in the table.

Example:

```
CREATE TABLE Price
(Qty int,
Unitprice int,
Total as Qty+Unitprice PERSISTED ) ;
```

Summary

- ❑ Enforcing data integrity guarantees the quality of the data in the database.
- ❑ Entity integrity defines a row as a unique entity for a particular table.
- ❑ Domain integrity is the validity of entries for a specific column
- ❑ Referential integrity preserves the defined relationships between tables
- ❑ SQL Server 2005 provides the following mechanisms to enforce the integrity of the data in a column:
  - o PRIMARY KEY Constraints
  - o FOREIGN KEY Constraints
  - o UNIQUE Constraints
  - o CHECK Constraints
  - o DEFAULT Definitions
  - o Allowing Null Values

Test your Understanding

1. What are the different types of integrity?
2. What are the two ways of enforcing domain integrity? Which one of them is preferable?
3. Can Primary and Foreign keys have null values?
4. How would you auto generate primary keys?

Session 09 and 12: Subqueries, Joins and T-SQL Enhancements

Learning Objectives

After completing this session, you will be able to:

- ❑ Explain TSQL Fundamentals
- ❑ Execute DML Statements like Select, Insert, Update and Delete
- ❑ Write Queries, SubQueries, and Joins
- ❑ Execute DDL Statements like Create table
- ❑ Describe TSQL Enhancements (CTE, PIVOT, UNPIVOT, APPLY).

TSQL Fundamentals

Transact-SQL is Microsoft's version of the Structured Query Language (SQL). It is the core of the decision-support capabilities of the SQL server engine. Transact SQL (T-SQL) is a powerful and unique superset of the SQL standard.

The Transact-SQL language is compliant with the American National Standards Institute (ANSI) SQL-92 standard at the entry level. It offers considerably more power because of its unique extensions to the standard. It is specific to SQL Server databases only.

Transact-SQL provides a number of capabilities that extend beyond typical implementations of SQL. These capabilities allow easily and efficiently writing queries that are difficult to write in standard SQL.

For example, it is possible to embed additional SELECT statements in the SELECT list, and drill into a result set by further selecting data directly from a SELECT statement, a feature known as a derived table.

Transact-SQL provides many system functions for dealing with strings (for concatenating strings, finding substrings and so on), for converting datatypes, and for manipulating and formatting date information. It provides a large collection of operators for doing arithmetic, comparing values, doing bit operations, and concatenating strings.

Transact-SQL also provides mathematical operations such as returning the square root of a number. In addition, special operators such as CUBE and ROLLUP can be used to efficiently perform multidimensional analysis at the database server, where the analysis can be optimized as part of the execution plan of a query. The CASE expression can be used to easily make complex conditional substitutions in the SELECT statement.

Transact-SQL provides programming constructs—such as variables, conditional operations (IFTHEN-ELSE), and looping, cursors, and error checking—that can simplify application development by allowing the user to use a simple SQL script rather than a third-generation programming language (3GL).

Stored procedures and functions are collections of SQL statements stored in a SQL Server database. Complex queries and transactions can be written as stored procedures and then invoked directly from the front-end application. Stored procedures are basic building blocks in database application development. Stored procedures and functions differ from each other in how they're used in the code. Stored procedures perform actions and usually contain all the error checking associated with that action. For example, a stored procedure might be written to insert a new order into a table, and the data must be checked for validity before the actual insert occurs.

A function, on the other hand, returns a value and that value can be used anywhere that any SQL statement can use a single value.

Triggers are a special class of stored procedure defined to execute automatically when an UPDATE, INSERT, or DELETE statement is issued against a table or view.

The SELECT Statement

The SELECT statement is the most frequently used SQL command and is the fundamental way to query data. It retrieves rows from the database and allows the selection of one or many rows or columns from one or many tables. The T-SQL Select is similar to the SQL statement, but more powerful. The Select can be used to retrieve data, join tables, assign local variables, and even create other tables.

The full syntax of the SELECT statement is complex, but the main clauses can be summarized as:

```
WITH <common_table_expression>]
SELECT select_list [ INTO new_table ]
[ FROM table_source ] [ WHERE search_condition ]
[ GROUP BY group_by_expression ]
[ HAVING search_condition ]
[ ORDER BY order_expression [ ASC | DESC ] ]
```

The UNION, EXCEPT and INTERSECT operators can be used between queries to combine or compare their results into one result set.

Processing Order of WHERE, GROUP BY, and HAVING Clauses

The following steps show the processing order for a SELECT statement with a WHERE clause, a GROUP BY clause, and a HAVING clause:

1. The FROM clause returns an initial result set.
2. The WHERE clause excludes rows not meeting its search condition.
3. The GROUP BY clause collects the selected rows into one group for each unique value in the GROUP BY clause.
4. Aggregate functions specified in the select list calculate summary values for each group.
5. The HAVING clause additionally excludes rows not meeting its search condition.

Example:

```
SELECT GetDate()
```

SELECT Clause

Syntax

```
SELECT [ ALL | DISTINCT ]
[ TOP expression [ PERCENT ] [ WITH TIES ] ]
<select_list>
<select_list> ::=
    {
                *
      | { table_name | view_name | table_alias }.*
      | { column_name | [ ] expression | $IDENTITY | $ROWGUID }
      | udt_column_name [ { . | :: } { { property_name | field_name } |
method_name(argument [,...n] ) } ]
      [ [ AS ] column_alias ]
      | column_alias = expression
    } [ ,...n ]
```

Arguments:

ALL

Specifies that duplicate rows can appear in the result set. ALL is the default.

DISTINCT

Specifies that only unique rows can appear in the result set. Null values are considered equal for the purposes of the DISTINCT keyword.

TOP expression [PERCENT] [WITH TIES]

Indicates that only a specified first set or percent of rows will be returned from the query result set. *expression* can be either a number or a percent of the rows.

The TOP clause can be used in SELECT, INSERT, UPDATE, and DELETE statements. Parentheses delimiting *expression* in TOP is required in INSERT, UPDATE, and DELETE statements.

< select_list >

The columns to be selected for the result set. The select list is a series of expressions separated by commas. The maximum number of expressions that can be specified in the select list is 4096.

*****

Specifies that all columns from all tables and views in the FROM clause should be returned. The columns are returned by table or view, as specified in the FROM clause, and in the order in which they exist in the table or view.

table_ name | view_ name | table_ alias.*

Limits the scope of the ***** to the specified table or view.

column_ name

Is the name of a column to return. Qualify *column_name* to prevent an ambiguous reference, such as occurs when two tables in the FROM clause have columns with duplicate names. For example, the **SalesOrderHeader** and **SalesOrderDetail** tables in the **AdventureWorks** database both have a column named **ModifiedDate**. If the two tables are joined in a query, the modified date of the **SalesOrderDetail** entries can be specified in the select list as **SalesOrderDetail.ModifiedDate**.

expression

Is a column name, constant, function, any combination of column names, constants, and functions connected by an operator or operators, or a subquery.

$IDENTITY

Returns the identity column.

$ROWGUID

Returns the row GUID column.

column_ alias

Is an alternative name to replace the column name in the query result set. For example, an alias such as **Quantity**, or **Quantity to Date**, or **Qty** can be specified for a column named **quantity**.

Aliases are used also to specify names for the results of expressions, for example:

```
USE AdventureWorks;
GO
SELECT AVG(UnitPrice) AS 'Average Price'
FROM Sales.SalesOrderDetail;
```

column_alias can be used in an ORDER BY clause. However, it cannot be used in a WHERE, GROUP BY, or HAVING clause. If the query expression is part of a DECLARE CURSOR statement, *column_alias* cannot be used in the FOR UPDATE clause.

FROM Clause

Specifies the tables, views, derived tables, and joined tables used in DELETE, SELECT, and UPDATE statements. In the SELECT statement, the FROM clause is required except when the select list contains only constants, variables, and arithmetic expressions (no column names).

Syntax

```
[ FROM { <table_source> } [ ,...n ] ]
<table_source> ::=
{
        table_or_view_name [ [ AS ] table_alias ] [ <tablesample_clause>
]
        [ WITH ( < table_hint > [ [ , ]...n ] ) ]
    | rowset_function [ [ AS ] table_alias ]
        [ ( bulk_column_alias [ ,...n ] ) ]
    | user_defined_function [ [ AS ] table_alias ]
    | OPENXML <openxml_clause>
    | derived_table [ AS ] table_alias [ ( column_alias [ ,...n ] ) ]
    | <joined_table>
    | <pivoted_table>
    | <unpivoted_table>
}
<tablesample_clause> ::=
    TABLESAMPLE [SYSTEM] ( sample_number [ PERCENT | ROWS ] )
```

```
        [ REPEATABLE ( repeat_seed ) ]

<joined_table> ::=
{
    <table_source> <join_type> <table_source> ON <search_condition>
    | <table_source> CROSS JOIN <table_source>
    | left_table_source { CROSS | OUTER } APPLY right_table_source
    | [ ( ] <joined_table> [ ) ]
}
<join_type> ::=
    [ { INNER | { { LEFT | RIGHT | FULL } [ OUTER ] } } [ <join_hint> ]
]
    JOIN

<pivoted_table> ::=
        table_source PIVOT <pivot_clause> table_alias

<pivot_clause> ::=
        ( aggregate_function ( value_column )
        FOR pivot_column
        IN ( <column_list> )
    )

<unpivoted_table> ::=
        table_source UNPIVOT <unpivot_clause> table_alias

<unpivot_clause> ::=
        ( value_column FOR pivot_column IN ( <column_list> ) )

<column_list> ::=
          column_name [ , ... ]
```

Arguments

<table_source>

Specifies a table, view, or derived table source, with or without an alias, to use in the Transact-SQL statement. Up to 256 table sources can be used in a statement, although the limit varies depending on available memory and the complexity of other expressions in the query. Individual queries may not support up to 256 table sources. A **table** variable can be specified as a table source.

The order of table sources after the FROM keyword does not affect the result set that is returned. SQL Server 2005 returns errors when duplicate names appear in the FROM clause.

table_or_view_name

Is the name of a table or view.

If the table or view exists in another database on the same computer that is running an instance of SQL Server, use a fully qualified name in the form *database.schema.object_name*. If the table or view exists outside the local server on a linked server, use a four-part name in the form *linked_server*.*catalog*.*schema*.*object*. A four-part table or view name that is constructed by using the OPENDATASOURCE function as the server part of the name can also be used to specify the table source.

[AS] table_alias

Is an alias for *table_source* that can be used either for convenience or to distinguish a table or view in a self-join or subquery. An alias is frequently a shortened table name used to refer to specific columns of the tables in a join. If the same column name exists in more than one table in the join, SQL Server requires that the column name be qualified by a table name, view name, or alias. The table name cannot be used if an alias is defined.

When a derived table, rowset or table-valued function, or operator clause (such as PIVOT or UNPIVOT) is used, the required *table_alias* at the end of the clause is the associated table name for all columns, including grouping columns, returned.

WITH **(**<table_hint> **)**

Specifies that the query optimizer use an optimization or locking strategy with this table and for this statement.

rowset_function

Specifies one of the rowset functions, such as OPENROWSET, that returns an object that can be used instead of a table reference.

bulk_column_alias

Is an optional alias to replace a column name in the result set. Column aliases are allowed only in SELECT statements that use the OPENROWSET function with the BULK option. When you use *bulk_column_alias*, specify an alias for every table column in the same order as the columns in the file.

user_defined_function

Specifies a table-valued function.

OPENXML <openxml_clause>

Provides a rowset view over an XML document.

derived_table

Is a subquery that retrieves rows from the database. *derived_table* is used as input to the outer query.

column_alias

Is an optional alias to replace a column name in the result set of the derived table. Include one column alias for each column in the select list, and enclose the complete list of column aliases in parentheses.

ROWS

Specifies that approximately *sample_number* of rows will be retrieved. When ROWS is specified, SQL Server returns an approximation of the number of rows specified. When ROWS is specified, the *sample_number* expression must evaluate to an integer value greater than zero.

REPEATABLE

Indicates that the selected sample can be returned again. When specified with the same *repeat_seed* value, SQL Server will return the same subset of rows as long as no changes have been made to any rows in the table. When specified with a different *repeat_seed* value, SQL Server will likely return some different sample of the rows in the table. The following actions to the table are considered changes: insert, update, delete, index rebuild or defragmentation, and database restore or attach.

repeat_seed

Is a constant integer expression used by SQL Server to generate a random number. *repeat_seed* is **bigint**. If *repeat_seed* is not specified, SQL Server assigns a value at random. For a specific *repeat_seed* value, the sampling result is always the same if no changes have been applied to the table. The *repeat_seed* expression must evaluate to an integer greater than zero.

<joined_table>

Is a result set that is the product of two or more tables. For multiple joins, use parentheses to change the natural order of the joins.

<join_type>

Specifies the type of join operation.

INNER

Specifies all matching pairs of rows are returned. Discards unmatched rows from both tables. When no join type is specified, this is the default.

FULL [OUTER]

Specifies that a row from either the left or right table that does not meet the join condition is included in the result set, and output columns that correspond to the other table are set to NULL. This is in addition to all rows typically returned by the INNER JOIN.

LEFT [OUTER]

Specifies that all rows from the left table not meeting the join condition are included in the result set, and output columns from the other table are set to NULL in addition to all rows returned by the inner join.

RIGHT [OUTER]

Specifies all rows from the right table not meeting the join condition are included in the result set, and output columns that correspond to the other table are set to NULL, in addition to all rows returned by the inner join.

<join_hint>

Specifies that the SQL Server query optimizer use one join hint, or execution algorithm, per join specified in the query FROM clause.

JOIN

Indicates that the specified join operation should occur between the specified table sources or views.

ON <search_condition>

Specifies the condition on which the join is based. The condition can specify any predicate, although columns and comparison operators are frequently used, for example:

```
SELECT p.ProductID, v.VendorID
FROM Production.Product p
JOIN Purchasing.ProductVendor v
ON (p.ProductID = v.ProductID)
```

When the condition specifies columns, the columns do not have to have the same name or same data type; however, if the data types are not the same, they must be either compatible or types that SQL Server 2005 can implicitly convert. If the data types cannot be implicitly converted, the condition must explicitly convert the data type by using the CONVERT function.

There can be predicates that involve only one of the joined tables in the ON clause. Such predicates also can be in the WHERE clause in the query. Although the placement of such predicates does not make a difference for INNER joins, they might cause a different result when OUTER joins are involved. This is because the predicates in the ON clause are applied to the table before the join, whereas the WHERE clause is semantically applied to the result of the join.

CROSS JOIN

Specifies the cross-product of two tables. Returns the same rows as if no WHERE clause was specified in an old-style, non-SQL-92-style join.

left_table_source { CROSS | OUTER } APPLY right_table_source

Specifies that the *right_table_source* of the APPLY operator is evaluated against every row of the *left_table_source*. This functionality is useful when the *right_table_source* contains a table-valued function that takes column values from the *left_table_source* as one of its arguments.

Either CROSS or OUTER must be specified with APPLY. When CROSS is specified, no rows are produced when the *right_table_source* is evaluated against a specified row of the *left_table_source* and returns an empty result set.

When OUTER is specified, one row is produced for each row of the *left_table_source* even when the *right_table_source* evaluates against that row and returns an empty result set.

For more information, see the Remarks section and Using APPLY.

left_table_source

Is a table source as defined in the previous argument. For more information, see the Remarks section.

right_table_source

Is a table source as defined in the previous argument. For more information, see the Remarks section.

table_source PIVOT <pivot_clause>

Specifies that the *table_source* is pivoted based on the *pivot_column*. *table_source* is a table or table expression. The output is a table that contains all columns of the *table_source* except the *pivot_column* and *value_column*. The columns of the *table_source*, except the *pivot_column* and *value_column*, are called the grouping columns of the pivot operator.

PIVOT performs a grouping operation on the input table with regard to the grouping columns and returns one row for each group. Additionally, the output contains one column for each value specified in the *column_list* that appears in the *pivot_column* of the *input_table*.

aggregate_function

Is a system or user-defined aggregate function. The aggregate function should be invariant to null values. An aggregate function invariant to null values does not consider null values in the group while it is evaluating the aggregate value.

The COUNT(*) system aggregate function is not allowed.

value_column

Is the value column of the PIVOT operator. When used with UNPIVOT, *value_column* cannot be the name of an existing column in the input *table_source*.

FOR pivot_column

Is the pivot column of the PIVOT operator. *pivot_column* must be of a type implicitly or explicitly convertible to **nvarchar()**. This column cannot be **image** or **rowversion**.

When UNPIVOT is used, *pivot_column* is the name of the output column that becomes narrowed from the *table_source*. There cannot be an existing column in *table_source* with that name.

IN **(** column_list **)**

In the PIVOT clause, lists the values in the *pivot_column* that will become the column names of the output table. The list cannot specify any column names that already exist in the input *table_source* that is being pivoted.

In the UNPIVOT clause, lists the columns in *table_source* that will be narrowed into a single *pivot_column*.

table_alias

Is the alias name of the output table. *pivot_table_alias* must be specified.

UNPIVOT < unpivot_clause >

Specifies that the input table is narrowed from multiple columns in *column_list* into a single column called *pivot_column*.

Using APPLY

Both the left and right operands of the APPLY operator are table expressions. The main difference between these operands is that the *right_table_source* can use a table-valued function that takes a column from the *left_table_source* as one of the arguments of the function. The *left_table_source* can include table-valued functions, but it cannot contain arguments that are columns from the *right_table_source*.

The APPLY operator works in the following way to produce the table source for the FROM clause:

1. Evaluates *right_table_source* against each row of the *left_table_source* to produce rowsets.

   The values in the *right_table_source* depend on *left_table_source*. *right_table_source* can be represented approximately this way: TVF(left_table_source.row), where TVF is a table-valued function.
2. Combines the result sets that are produced for each row in the evaluation of right_table_source with the left_table_source by performing a UNION ALL operation.

   The list of columns produced by the result of the APPLY operator is the set of columns from the *left_table_source* that is combined with the list of columns from the *right_table_source*.

Using PIVOT and UNPIVOT

The *pivot_column* and *value_column* are grouping columns that are used by the PIVOT operator. PIVOT follows the following process to obtain the output result set:

1. Performs a GROUP BY on its *input_table* against the grouping columns and produces one output row for each group.

   The grouping columns in the output row obtain the corresponding column values for that group in the *input_table*.
2. Generates values for the columns in the column list for each output row by performing the following:
   a) Grouping additionally the rows generated in the GROUP BY in the previous step against the *pivot_column*.

b) For each output column in the *column_list*, selecting a subgroup that satisfies the condition:

```
pivot_column = CONVERT(<data type of pivot_column>,
'output_column')
```

c) *aggregate_function* is evaluated against the *value_column* on this subgroup and its result is returned as the value of the corresponding *output_column*. If the subgroup is empty, SQL Server generates a null value for that *output_column*. If the aggregate function is COUNT and the subgroup is empty, zero (0) is returned.

Examples

Using a simple FROM clause

The following example retrieves the TerritoryID and Name columns from the SalesTerritory table in the AdventureWorks sample database.

```
USE AdventureWorks ;
GO
SELECT TerritoryID, Name
FROM Sales.SalesTerritory
ORDER BY TerritoryID ;
```

Using the TABLOCK and HOLDLOCK optimizer hints

The following partial transaction shows how to place an explicit shared table lock on Employee and how to read the index. The lock is held throughout the whole transaction.

```
USE AdventureWorks ;
GO
BEGIN TRAN
SELECT COUNT(*)
FROM HumanResources.Employee WITH (TABLOCK, HOLDLOCK) ;
```

Using the SQL-92 CROSS JOIN syntax

The following example returns the cross product of the two tables Employee and Department. A list of all possible combinations of EmployeeID rows and all Department name rows are returned.

```
USE AdventureWorks ;
GO
SELECT e.EmployeeID, d.Name AS Department
FROM HumanResources.Employee e
CROSS JOIN HumanResources.Department d
ORDER BY e.EmployeeID, d.Name ;
```

Using the SQL-92 FULL OUTER JOIN syntax

The following example returns the product name and any corresponding sales orders in the SalesOrderDetail table. It also returns any sales orders that have no product listed in the Product table, and any products with a sales order other than the one listed in the Product table.

```
USE AdventureWorks ;
GO
-- The OUTER keyword following the FULL keyword is optional.
```

```
SELECT p.Name, sod.SalesOrderID
FROM Production.Product p
FULL OUTER JOIN Sales.SalesOrderDetail sod
ON p.ProductID = sod.ProductID
WHERE p.ProductID IS NULL
OR sod.ProductID IS NULL
ORDER BY p.Name ;
```

Using the SQL-92 LEFT OUTER JOIN syntax

The following example joins two tables on ProductID and preserves the unmatched rows from the left table. The Product table is matched with the SalesOrderDetail table on the ProductID columns in each table. All products, ordered and not ordered, appear in the result set.

```
USE AdventureWorks ;
GO
SELECT p.Name, sod.SalesOrderID
FROM Production.Product p
LEFT OUTER JOIN Sales.SalesOrderDetail sod
ON p.ProductID = sod.ProductID
ORDER BY p.Name ;
```

The following example returns all product names and sales order IDs.

```
USE AdventureWorks ;
GO
-- By default, SQL Server performs an INNER JOIN if only the JOIN
-- keyword is specified.
SELECT p.Name, sod.SalesOrderID
FROM Production.Product p
INNER JOIN Sales.SalesOrderDetail sod
ON p.ProductID = sod.ProductID
ORDER BY p.Name ;
```

Using the SQL-92 RIGHT OUTER JOIN syntax

The following example joins two tables on TerritoryID and preserves the unmatched rows from the right table. The SalesTerritory table is matched with the SalesPerson table on the TerritoryID column in each table. All salespersons appear in the result set, whether or not they are assigned a territory.

```
USE AdventureWorks ;
GO
SELECT st.Name AS Territory, sp.SalesPersonID
FROM Sales.SalesTerritory st
RIGHT OUTER JOIN Sales.SalesPerson sp
ON st.TerritoryID = sp.TerritoryID ;
```

Using HASH and MERGE join hints

The following example performs a three-table join among the Product, ProductVendor, and Vendor tables to produce a list of products and their vendors. The query optimizer joins Product and

ProductVendor (p and pv) by using a MERGE join. Next, the results of the Product and ProductVendor MERGE join (p and pv) are HASH joined to the Vendor table to produce (p and pv) and v.

```
USE AdventureWorks ;
GO
SELECT p.Name AS ProductName, v.Name AS VendorName
FROM Production.Product p
INNER MERGE JOIN Purchasing.ProductVendor pv
ON p.ProductID = pv.ProductID
INNER HASH JOIN Purchasing.Vendor v
ON pv.VendorID = v.VendorID
ORDER BY p.Name, v.Name ;
```

Using a derived table

The following example uses a derived table, a SELECT statement after the FROM clause, to return the first and last names of all employees and the cities in which they live.

```
USE AdventureWorks ;
GO
SELECT RTRIM(c.FirstName) + ' ' + LTRIM(c.LastName) AS Name,
 d.City
FROM Person.Contact c
INNER JOIN HumanResources.Employee e ON c.ContactID = e.ContactID
INNER JOIN (SELECT AddressID, City FROM Person.Address) AS d
ON e.AddressID = d.AddressID
ORDER BY c.LastName, c.FirstName ;
```

Using TABLESAMPLE to read data from a sample of rows in a table

The following example uses TABLESAMPLE in the FROM clause to return approximately 10 percent of all the rows in the Customer table in the AdventureWorks database.

```
USE AdventureWorks ;
GO
SELECT *
FROM Sales.Customer TABLESAMPLE SYSTEM (10 PERCENT) ;
```

Using APPLY

The following example assumes that the following tables with the following schema exist in the database:

- Departments: DeptID, DivisionID, DeptName, DeptMgrID
- EmpMgr: MgrID, EmpID
- Employees: EmpID, EmpLastName, EmpFirstName, EmpSalary

There is also a table-valued function, GetReports(MgrID) that returns the list of all employees (EmpID, EmpLastName, EmpSalary) that report directly or indirectly to the specified MgrID.

The example uses APPLY to return all departments and all employees in that department. If a particular department does not have any employees, there will not be any rows returned for that department.

```
SELECT DeptID, DeptName, DeptMgrID, EmpID, EmpLastName, EmpSalary
FROM Departments d CROSS APPLY dbo.GetReports(d.DeptMgrID) ;
```

If you want the query to produce rows for those departments without employees, which will produce null values for the EmpID, EmpLastName and EmpSalary columns, use OUTER APPLY instead.

```
SELECT DeptID, DeptName, DeptMgrID, EmpID, EmpLastName, EmpSalary
FROM Departments d OUTER APPLY dbo.GetReports(d.DeptMgrID) ;
```

Using PIVOT and UNPIVOT

The following example returns the number of purchase orders placed by employee IDs 164, 198, 223, 231, and 233, categorized by vendor ID.

```
USE AdventureWorks
GO
SELECT VendorID, [164] AS Emp1, [198] AS Emp2, [223] AS Emp3, [231] AS
Emp4, [233] AS Emp5
FROM
(SELECT PurchaseOrderID, EmployeeID, VendorID
FROM Purchasing.PurchaseOrderHeader) p
PIVOT
(
COUNT (PurchaseOrderID)
FOR EmployeeID IN
( [164], [198], [223], [231], [233] )
) AS pvt
ORDER BY VendorID;
```

Here is a partial result set:

```
VendorID    Emp1        Emp2        Emp3        Emp4        Emp5
1           4           3           5           4           4
2           4           1           5           5           5
3           4           3           5           4           4
4           4           2           5           5           4
5           5           1           5           5           5
```

To unpivot the table, assume the result set produced in the previous example is stored as pvt. The query looks like the following.

```
--Create the table and insert values as portrayed in the previous
example.
CREATE TABLE pvt (VendorID int, Emp1 int, Emp2 int,
Emp3 int, Emp4 int, Emp5 int)
GO
```

```
INSERT INTO pvt VALUES (1,4,3,5,4,4)
INSERT INTO pvt VALUES (2,4,1,5,5,5)
INSERT INTO pvt VALUES (3,4,3,5,4,4)
INSERT INTO pvt VALUES (4,4,2,5,5,4)
INSERT INTO pvt VALUES (5,5,1,5,5,5)
GO
--Unpivot the table.
SELECT VendorID, Employee, Orders
FROM
    (SELECT VendorID, Emp1, Emp2, Emp3, Emp4, Emp5
    FROM pvt) p
UNPIVOT
    (Orders FOR Employee IN
        (Emp1, Emp2, Emp3, Emp4, Emp5)
)AS unpvt
GO
```

Here is a partial result set:

```
VendorID    Employee    Orders
1           Emp1        4
1           Emp2        3
1           Emp3        5
1           Emp4        4
1           Emp5        4
2           Emp1        4
2           Emp2        1
2           Emp3        5
2           Emp4        5
2           Emp5        5
```

WHERE clause

Specifies the search condition for the rows returned by the query.

Syntax

```
[ WHERE <search_condition> ]
```

Arguments

< search_condition >

Defines the condition to be met for the rows to be returned. There is no limit to the number of predicates that can be included in a search condition. For more

Is a combination of one or more predicates that use the logical operators AND, OR, and NOT.

Syntax

```
< search_condition > ::=
    { [ NOT ] <predicate> | ( <search_condition> ) }
    [ { AND | OR } [ NOT ] { <predicate> | ( <search_condition> ) } ]
[ ,...n ]
<predicate> ::=
    { expression { = | < > | ! = | > | > = | ! > | < | < = | ! < }
expression
    | string_expression [ NOT ] LIKE string_expression
  [ ESCAPE 'escape_character' ]
    | expression [ NOT ] BETWEEN expression AND expression
    | expression IS [ NOT ] NULL
    | CONTAINS
    ( { column | * } , '< contains_search_condition >' )
    | FREETEXT ( { column | * } , 'freetext_string' )
    | expression [ NOT ] IN ( subquery | expression [ ,...n ] )
    | expression { = | < > | ! = | > | > = | ! > | < | < = | ! < }
  { ALL | SOME | ANY} ( subquery )
    | EXISTS ( subquery )     }
```

Arguments

<search_condition>

Specifies the conditions for the rows returned in the result set for a SELECT statement, query expression, or subquery. For an UPDATE statement, specifies the rows to be updated. For a DELETE statement, specifies the rows to be deleted. There is no limit to the number of predicates that can be included in a Transact-SQL statement search condition.

NOT

Negates the Boolean expression specified by the predicate.

AND

Combines two conditions and evaluates to TRUE when both of the conditions are TRUE.

OR

Combines two conditions and evaluates to TRUE when either condition is TRUE.

< predicate >

Is an expression that returns TRUE, FALSE, or UNKNOWN.

expression

Is a column name, a constant, a function, a variable, a scalar subquery, or any combination of column names, constants, and functions connected by an operator or operators, or a subquery. The expression can also contain the CASE function.

=

Is the operator used to test the equality between two expressions.

<>

Is the operator used to test the condition of two expressions not being equal to each other.

!=

Is the operator used to test the condition of two expressions not being equal to each other.

>

Is the operator used to test the condition of one expression being greater than the other.

>=

Is the operator used to test the condition of one expression being greater than or equal to the other expression.

!>

Is the operator used to test the condition of one expression not being greater than the other expression.

<

Is the operator used to test the condition of one expression being less than the other.

<=

Is the operator used to test the condition of one expression being less than or equal to the other expression.

!<

Is the operator used to test the condition of one expression not being less than the other expression.

string_expression

Is a string of characters and wildcard characters.

[NOT] LIKE

Indicates that the subsequent character string is to be used with pattern matching

pattern

Is the specific string of characters to search for in *match_expression*, and can include the following valid wildcard characters. *pattern* can be a maximum of 8,000 bytes.

Wildcard character	**Description**	**Example**
%	Any string of zero or more characters.	WHERE title LIKE '%computer%' finds all book titles with the word 'computer' anywhere in the book title.
_ (underscore)	Any single character.	WHERE au_fname LIKE '_ean' finds all four-letter first names that end with ean (Dean, Sean, and so on).
[ ]	Any single character within the specified range ([a-f]) or set ([abcdef]).	WHERE au_lname LIKE '[C-P]arsen' finds author last names ending with arsen and starting with any single character between C and P, for example Carsen, Larsen, Karsen, and so on.
[^]	Any single character not within the specified range ([^a-f]) or set ([^abcdef]).	WHERE au_lname LIKE 'de[^l]%' all author last names starting with de and where the following letter is not l.

ESCAPE **'**escape_ character**'**

Allows for a wildcard character to be searched for in a character string instead of functioning as a wildcard. *escape_character* is the character that is put in front of the wildcard character to indicate this special use.

[NOT] BETWEEN

Specifies an inclusive range of values. Use AND to separate the starting and ending values.

IS [NOT] NULL

Specifies a search for null values, or for values that are not null, depending on the keywords used. An expression with a bitwise or arithmetic operator evaluates to NULL if any one of the operands is NULL.

CONTAINS

Searches columns that contain character-based data for precise or less precise (*fuzzy*) matches to single words and phrases, the proximity of words within a certain distance of one another, and weighted matches. This option can only be used with SELECT statements.

FREETEXT

Provides a simple form of natural language query by searching columns that contain character-based data for values that match the meaning instead of the exact words in the predicate. This option can only be used with SELECT statements.

[NOT] IN

Specifies the search for an expression, based on whether the expression is included in or excluded from a list. The search expression can be a constant or a column name, and the list can be a set of constants or, more typically, a subquery. Enclose the list of values in parentheses

subquery

Can be considered a restricted SELECT statement and is similar to <query_expresssion> in the SELECT statement. The ORDER BY clause, the COMPUTE clause, and the INTO keyword are not allowed

ALL

Used with a comparison operator and a subquery. Returns TRUE for <predicate> when all values retrieved for the subquery satisfy the comparison operation, or FALSE when not all values satisfy the comparison or when the subquery returns no rows to the outer statement

{ SOME | ANY }

Used with a comparison operator and a subquery. Returns TRUE for <predicate> when any value retrieved for the subquery satisfies the comparison operation, or FALSE when no values in the subquery satisfy the comparison or when the subquery returns no rows to the outer statement. Otherwise, the expression is UNKNOWN.

EXISTS

Used with a subquery to test for the existence of rows returned by the subquery.

Examples

Using WHERE with LIKE and ESCAPE syntax

The following example searches for the rows in which the LargePhotoFileName column has the characters green_, and uses the ESCAPE option because _ is a wildcard character. Without specifying the ESCAPE option, the query would search for any description values that contain the word green followed by any single character other than the _ character.

```
USE AdventureWorks ;
GO
SELECT *
FROM Production.ProductPhoto
```

```
WHERE LargePhotoFileName LIKE '%greena_%' ESCAPE 'a' ;
```

Using WHERE and LIKE syntax with Unicode data

The following example uses the WHERE clause to retrieve the mailing address for any company that is outside the United States (US) and in a city whose name starts with Mn.

```
USE AdventureWorks ;
GO
SELECT AddressLine1, AddressLine2, City, PostalCode, CountryRegionCode
FROM Person.Address AS a
JOIN Person.StateProvince AS s ON a.StateProvinceID = s.StateProvinceID
WHERE CountryRegionCode NOT IN ('US')
AND City LIKE N'Mn%' ;
```

GROUP BY clause

Specifies the groups into which output rows are to be placed. If aggregate functions are included in the SELECT clause <select list>, GROUP BY calculates a summary value for each group. When GROUP BY is specified, either each column in any nonaggregate expression in the select list should be included in the GROUP BY list, or the GROUP BY expression must exactly match the select list expression.

Syntax

```
[ GROUP BY [ ALL ] group_by_expression [ ,...n ]
    [ WITH { CUBE | ROLLUP } ]
]
```

Arguments

ALL

Includes all groups and result sets, even those that do not have any rows that meet the search condition specified in the WHERE clause. When ALL is specified, null values are returned for the summary columns of groups that do not meet the search condition. You cannot specify ALL with the CUBE or ROLLUP operators.

GROUP BY ALL is not supported in queries that access remote tables if there is also a WHERE clause in the query.

group_by_expression

Is an expression on which grouping is performed. *group_by_expression* is also known as a grouping column. *group_by expression* can be a column or a nonaggregate expression that references a column returned by the FROM clause. A column alias that is defined in the select list cannot be used to specify a grouping column.

Examples

Using a simple GROUP BY clause

The following example retrieves the total for each SalesOrderID from the SalesOrderDetail table.

```
USE AdventureWorks ;
GO
```

```
SELECT SalesOrderID, SUM(LineTotal) AS SubTotal
FROM Sales.SalesOrderDetail sod
GROUP BY SalesOrderID
ORDER BY SalesOrderID ;
```

Using a GROUP BY clause with multiple tables

The following example retrieves the number of employees for each City from the Address table joined with the Employee table.

```
USE AdventureWorks ;
GO
SELECT a.City, COUNT(e.AddressID) EmployeeCount
FROM HumanResources.Employee e
INNER JOIN Person.Address a
ON e.AddressID = a.AddressID
GROUP BY a.City
ORDER BY a.City ;
```

HAVING clause

Specifies a search condition for a group or an aggregate. HAVING can be used only with the SELECT statement. HAVING is typically used in a GROUP BY clause. When GROUP BY is not used, HAVING behaves like a WHERE clause.

Syntax

```
[ HAVING <search condition> ]
```

Arguments

<search_condition>

Specifies the search condition for the group or the aggregate to meet. When HAVING is used with GROUP BY ALL, the HAVING clause overrides ALL.

The **text**, **image**, and **ntext** data types cannot be used in a HAVING clause.

Examples

The following example that uses a simple HAVING clause retrieves the total for each SalesOrderID from the SalesOrderDetail table that exceeds $100000.00.

```
USE AdventureWorks ;
GO
SELECT SalesOrderID, SUM(LineTotal) AS SubTotal
FROM Sales.SalesOrderDetail sod
GROUP BY SalesOrderID
HAVING SUM(LineTotal) > 100000.00
ORDER BY SalesOrderID ;
```

ORDER BY clause

Specifies the sort order used on columns returned in a SELECT statement. The ORDER BY clause is not valid in views, inline functions, derived tables, and subqueries, unless TOP is also specified.

Syntax

```
[ ORDER BY
    {
    order_by_expression
  [ COLLATE collation_name ]
  [ ASC | DESC ]
    } [ ,...n ]
]
```

Arguments

order_by_expression

Specifies a column on which to sort. A sort column can be specified as a name or column alias, or a nonnegative integer representing the position of the name or alias in the select list. Column names and aliases can be qualified by the table or view name. In Microsoft SQL Server 2005, qualified column names and aliases are resolved to columns listed in the FROM clause. If *order_by_expression* is not qualified, it must be unique among all columns listed in the SELECT statement.

Multiple sort columns can be specified. The sequence of the sort columns in the ORDER BY clause defines the organization of the sorted result set.

The ORDER BY clause can include items not appearing in the select list. However, if SELECT DISTINCT is specified, or if the SELECT statement contains a UNION operator, the sort columns must appear in the select list.

Additionally, when the SELECT statement includes a UNION operator, the column names or column aliases must be those specified in the first select list.

COLLATE {collation_name}

Specifies that the ORDER BY operation should be performed according to the collation specified in *collation_name*, and not according to the collation of the column as defined in the table or view. *collation_name* can be either a Windows collation name or a SQL collation name. For more information, see Collation Settings in Setup and Using SQL Collations. COLLATE is applicable only for columns of the **char**, **varchar**, **nchar**, and **nvarchar** data types.

ASC

Specifies that the values in the specified column should be sorted in ascending order, from lowest value to highest value.

DESC

Specifies that the values in the specified column should be sorted in descending order, from highest value to lowest value.

Example:

The following query returns results ordered by ascending ProductID:

```
USE AdventureWorks;
GO
SELECT ProductID, ProductLine, ProductModelID
FROM Production.Product
ORDER BY ProductID
```

Subqueries

A subquery is a query that is nested inside a SELECT, INSERT, UPDATE, or DELETE statement, or inside another subquery. A subquery can be used anywhere an expression is allowed. In this example a subquery is used as a column expression named MaxUnitPrice in a SELECT statement.

```
SELECT Ord.SalesOrderID, Ord.OrderDate,
    (SELECT MAX(OrdDet.UnitPrice)
     FROM AdventureWorks.Sales.SalesOrderDetail AS OrdDet
     WHERE Ord.SalesOrderID = OrdDet.SalesOrderID) AS MaxUnitPrice
FROM AdventureWorks.Sales.SalesOrderHeader AS Ord
```

A subquery is also called an inner query or inner select, while the statement containing a subquery is also called an outer query or outer select.

Many Transact-SQL statements that include subqueries can be alternatively formulated as joins. Other questions can be posed only with subqueries. In Transact-SQL, there is usually no performance difference between a statement that includes a subquery and a semantically equivalent version that does not. However, in some cases where existence must be checked, a join yields better performance. Otherwise, the nested query must be processed for each result of the outer query to ensure elimination of duplicates. In such cases, a join approach would yield better results. The following is an example showing both a subquery SELECT and a join SELECT that return the same result set:

```
/* SELECT statement built using a subquery. */
SELECT Name
FROM AdventureWorks.Production.Product
WHERE ListPrice =
    (SELECT ListPrice
     FROM AdventureWorks.Production.Product
     WHERE Name = 'Chainring Bolts' )

/* SELECT statement built using a join that returns
   the same result set. */
SELECT Prd1. Name
FROM AdventureWorks.Production.Product AS Prd1
     JOIN AdventureWorks.Production.Product AS Prd2
       ON (Prd1.ListPrice = Prd2.ListPrice)
WHERE Prd2. Name = 'Chainring Bolts'
```

A subquery nested in the outer SELECT statement has the following components:

- A regular SELECT query including the regular select list components.
- A regular FROM clause including one or more table or view names.
- An optional WHERE clause.
- An optional GROUP BY clause.
- An optional HAVING clause.

The SELECT query of a subquery is always enclosed in parentheses. It cannot include a COMPUTE or FOR BROWSE clause, and may only include an ORDER BY clause when a TOP clause is also specified.

A subquery can be nested inside the WHERE or HAVING clause of an outer SELECT, INSERT, UPDATE, or DELETE statement, or inside another subquery. Up to 32 levels of nesting is possible, although the limit varies based on available memory and the complexity of other expressions in the query. Individual queries may not support nesting up to 32 levels. A subquery can appear anywhere an expression can be used, if it returns a single value.

If a table appears only in a subquery and not in the outer query, then columns from that table cannot be included in the output (the select list of the outer query).

Statements that include a subquery usually take one of these formats:

- WHERE *expression* [NOT] IN **(***subquery***)**
- WHERE *expression comparison_operator* [ANY | ALL] **(***subquery***)**
- WHERE [NOT] EXISTS **(***subquery***)**

In some Transact-SQL statements, the subquery can be evaluated as if it were an independent query. Conceptually, the subquery results are substituted into the outer query (although this is not necessarily how Microsoft SQL Server 2005 actually processes Transact-SQL statements with subqueries).

There are three basic types of subqueries. Those that:

- Operate on lists introduced with IN, or those that a comparison operator modified by ANY or ALL.
- Are introduced with an unmodified comparison operator and must return a single value.
- Are existence tests introduced with EXISTS.

INSERT Statement

Adds a new row to a table or a view.

Syntax

```
[ WITH <common_table_expression> [ ,...n ] ]
INSERT
    [ TOP ( expression ) [ PERCENT ] ]
    [ INTO]
    { <object> | rowset_function_limited
      [ WITH ( <Table_Hint_Limited> [ ...n ] ) ]
    }
{
    [ ( column_list ) ]
    [ <OUTPUT Clause> ]
    { VALUES ( { DEFAULT | NULL | expression } [ ,...n ] )
    | derived_table
    | execute_statement
    }
}
    | DEFAULT VALUES
[; ]

<object> ::=
{
    [ server_name . database_name . schema_name .
      | database_name .[ schema_name ] .
      | schema_name .
    ]
        table_or_view_name
}
```

Arguments

`WITH <common_table_expression>`:

- Specifies the temporary named result set, also known as common table expression, defined within the scope of the INSERT statement. The result set is derived from a SELECT statement.
- Common table expressions can also be used with the SELECT, DELETE, UPDATE, and CREATE VIEW statements..

`TOP` **(** *`expression`* **)** `[ PERCENT ]`:

- Specifies the number or percent of random rows that will be inserted. *expression* can be either a number or a percent of the rows. The rows referenced in the TOP expression that are used with INSERT, UPDATE, or DELETE are not arranged in any order.
- Parentheses delimiting *expression* in TOP are required in INSERT, UPDATE, and DELETE statements.

`INTO`: An optional keyword that can be used between INSERT and the target table.

server_name: The name of the server (using the OPENDATASOURCE function as the server name) on which the table or view is located. If *server_name* is specified, *database_name* and *schema_name* are required.

database_name: The name of the database.

schema_name : The name of the schema to which the table or view belongs.

table_or view_name : The name of the table or view that is to receive the data. A **table** variable, within its scope, can be used as a table source in an INSERT statement. The view referenced by *table_or_view_name* must be updatable and reference exactly one base table in the FROM clause of the view. For example, an INSERT into a multitable view must use a *column_list* that references only columns from one base table. For more information about updatable views,

rowset_function_limited: Either the OPENQUERY or OPENROWSET function.

WITH (<table_hint_limited> [... *n*]) :

- ❑ Specifies one or more table hints that are allowed for a target table. The WITH keyword and the parentheses are required.
- ❑ READPAST, NOLOCK, and READUNCOMMITTED are not allowed.
- ❑ Specifying the TABLOCK hint on a table that is the target of an INSERT statement has the same effect as specifying the TABLOCKX hint. An exclusive lock is taken on the table.

(column_list **)**:

- ❑ Is a list of one or more columns in which to insert data. *column_list* must be enclosed in parentheses and delimited by commas.
- ❑ If a column is not in *column_list*, the SQL Server 2005 Database Engine must be able to provide a value based on the definition of the column; otherwise, the row cannot be loaded. The Database Engine automatically provides a value for the column if the column:
    - o Has an IDENTITY property. The next incremental identity value is used.
    - o Has a default. The default value for the column is used.
    - o Has a **timestamp** data type. The current timestamp value is used.
    - o Is nullable. A null value is used.
    - o Is a computed column. The calculated value is used.

column_list and VALUES list must be used when explicit values are inserted into an identity column, and the SET IDENTITY_INSERT option must be ON for the table.

OUTPUT Clause: Returns inserted rows as part of the insert operation. The **OUTPUT clause i**s not supported in DML statements that reference local partitioned views, distributed partitioned views, or remote tables, or INSERT statements that contain an *execute_statement*.

VALUES: Introduces the list of data values to be inserted. There must be one data value for each column in *column_list*, if specified, or in the table. The values list must be enclosed in parentheses.

If the values in the VALUES list are not in the same order as the columns in the table or do not have a value for each column in the table, *column_list* must be used to explicitly specify the column that stores each incoming value.

DEFAULT : Forces the Database Engine to load the default value defined for a column. If a default does not exist for the column and the column allows null values, NULL is inserted. For a column defined with the **timestamp** data type, the next timestamp value is inserted. DEFAULT is not valid for an identity column.

expression: Is a constant, a variable, or an expression. The expression cannot contain a SELECT or EXECUTE statement.

derived_table: Is any valid SELECT statement that returns rows of data to be loaded into the table. The SELECT statement cannot contain a common table expression (CTE).

execute_statement:

- Is any valid EXECUTE statement that returns data with SELECT or READTEXT statements. The SELECT statement cannot contain a CTE.
- If *execute_statement* is used with INSERT, each result set must be compatible with the columns in the table or in *column_list*.
- *execute_statement* can be used to execute stored procedures on the same server or a remote server. The procedure in the remote server is executed, and the result sets are returned to the local server and loaded into the table in the local server.
- If *execute_statement* returns data with the READTEXT statement, each READTEXT statement can return a maximum of 1 MB (1024 KB) of data. *execute_statement* can also be used with extended procedures. *execute_statement* inserts the data returned by the main thread of the extended procedure; however, output from threads other than the main thread are not inserted.

DEFAULT VALUES forces the new row to contain the default values defined for each column.

Examples:

Using a simple INSERT statement:

The following example inserts one row in the Production.UnitMeasure table. Because values for all columns are supplied and are listed in the same order as the columns in the table, the column names do not have to be specified in *column_list.*

```
USE AdventureWorks;
GO
INSERT INTO Production.UnitMeasure
VALUES (N'F2', N'Square Feet', GETDATE());
GO
```

Inserting data that is not in the same order as the table columns

The following example uses *column_list* to explicitly specify the values that are inserted into each column. The column order in the UnitMeasure table is UnitMeasureCode, Name, ModifiedDate; however, the columns are not listed in that order in *column_list*.

```
USE AdventureWorks;
GO
INSERT INTO Production.UnitMeasure (Name, UnitMeasureCode,
    ModifiedDate)
VALUES (N'Square Yards', N'Y2', GETDATE());
GO
```

UPDATE statement

Changes existing data in a table or view.

Syntax

```
[ WITH <common_table_expression> [...n] ]
UPDATE
    [ TOP ( expression ) [ PERCENT ] ]
    { <object> | rowset_function_limited
     [ WITH ( <Table_Hint_Limited> [ ...n ] ) ]
    }
    SET
        { column_name = { expression | DEFAULT | NULL }
          | { udt_column_name.{ { property_name = expression
                                | field_name = expression }
                               | method_name ( argument [ ,...n ] )
                              }
            }
          | column_name { .WRITE ( expression , @Offset , @Length ) }
          | @variable = expression
          | @variable = column = expression [ ,...n ]
        } [ ,...n ]
    [ <OUTPUT Clause> ]
    [ FROM{ <table_source> } [ ,...n ] ]
    [ WHERE { <search_condition>
            | { [ CURRENT OF
                  { { [ GLOBAL ] cursor_name }
                      | cursor_variable_name
                  }
                ]
              }
            }
    ]
    [ OPTION ( <query_hint> [ ,...n ] ) ]
[ ; ]

<object> ::=
{
    [ server_name . database_name . schema_name .
    | database_name .[ schema_name ] .
    | schema_name .
    ]
        table_or_view_name}
```

Arguments:

WITH <common_table_expression>:

- Specifies the temporary named result set or view, also known as common table expression (CTE), defined within the scope of the UPDATE statement. The CTE result set is derived from a simple query and is referenced by UPDATE statement.
- Common table expressions can also be used with the SELECT, INSERT, DELETE, and CREATE VIEW statements.

TOP (expression) [PERCENT]:

- Specifies the number or percent of rows that will be updated. *expression* can be either a number or a percent of the rows.
- The rows referenced in the TOP expression used with INSERT, UPDATE, or DELETE are not arranged in any order.
- Parentheses delimiting *expression* in TOP are required in INSERT, UPDATE, and DELETE statements.

server_name: Is the name of the server (using a linked server name or the OPENDATASOURCE function as the server name) on which the table or view is located. If server_name is specified, database_name and schema_name are required.

database_name : Is the name of the database.

schema_name: Is the name of the schema to which the table or view belongs.

table_or view_name:

- Is the name of the table or view from which the rows are to be updated.
- A **table** variable, within its scope, can be used as a table source in an UPDATE statement.
- The view referenced by *table_or_view_name* must be updatable and reference exactly one base table in the FROM clause of the view.

rowset_function_limited : Is either the OPENQUERY or OPENROWSET function, subject to provider capabilities. For more information about capabilities needed by the provider, see UPDATE and DELETE Requirements for OLE DB Providers.

WITH (<Table_Hint_Limited>) : Specifies one or more table hints that are allowed for a target table. The WITH keyword and the parentheses are required. NOLOCK and READUNCOMMITTED are not allowed

SET: Specifies the list of column or variable names to be updated.

column_name: Is a column that contains the data to be changed. *column_name* must exist in *table_or view_name*. Identity columns cannot be updated.

expression: Is a variable, literal value, expression, or a subselect statement (enclosed with parentheses) that returns a single value. The value returned by *expression* replaces the existing value in *column_name* or *@variable*.

DEFAULT: Specifies that the default value defined for the column is to replace the existing value in the column. This can also be used to change the column to NULL if the column has no default and is defined to allow null values.

<OUTPUT_Clause> : Returns updated data or expressions based on it as part of the UPDATE operation. The OUTPUT clause is not supported in any DML statements that target remote tables or views.

FROM <table_source> :

- Specifies that a table, view, or derived table source is used to provide the criteria for the update operation.
- If the object being updated is the same as the object in the FROM clause and there is only one reference to the object in the FROM clause, an object alias may or may not be specified. If the object being updated appears more than one time in the FROM clause, one, and only one, reference to the object must not specify a table alias. All other references to the object in the FROM clause must include an object alias.
- A view with an INSTEAD OF UPDATE trigger cannot be a target of an UPDATE with a FROM clause.

WHERE: Specifies the conditions that limit the rows that are updated. There are two forms of update based on which form of the WHERE clause is used:

- Searched updates specify a search condition to qualify the rows to delete.
- Positioned updates use the CURRENT OF clause to specify a cursor. The update operation occurs at the current position of the cursor.

<search_condition> : Specifies the condition to be met for the rows to be updated. The search condition can also be the condition upon which a join is based. There is no limit to the number of predicates that can be included in a search condition.

Examples:

Using a simple UPDATE statement

The following examples show how all rows can be affected when a WHERE clause is not used to specify the row or rows to update.

This example updates the values in the Bonus, CommissionPct, and SalesQuota columns for all rows in the SalesPerson table.

```
USE AdventureWorks;
GO
UPDATE Sales.SalesPerson
SET Bonus = 6000, CommissionPct = .10, SalesQuota = NULL;
GO
```

You can also use computed values in an UPDATE statement. The following example doubles the value in the ListPrice column for all rows in the Product table.

```
USE AdventureWorks ;
GO
UPDATE Production.Product
SET ListPrice = ListPrice * 2;
GO
```

Using the UPDATE statement with a WHERE clause

The following example uses the WHERE clause to specify which rows to update. For example, Adventure Works Cycles sells their bicycle model Road-250 in two colors: red and black. The company has decided to change the color of red for this model to metallic red. The following statement updates the rows in the Production.Product table for all red Road-250 products.

```
USE AdventureWorks;
GO
UPDATE Production.Product
SET Color = N'Metallic Red'
WHERE Name LIKE N'Road-250%' AND Color = N'Red';
GO
```

Using the UPDATE statement with information from another table

The following example modifies the SalesYTD column in the SalesPerson table to reflect the most recent sales recorded in the SalesOrderHeader table.

```
USE AdventureWorks;
GO
UPDATE Sales.SalesPerson
SET SalesYTD = SalesYTD + SubTotal
FROM Sales.SalesPerson AS sp
JOIN Sales.SalesOrderHeader AS so
    ON sp.SalesPersonID = so.SalesPersonID
    AND so.OrderDate = (SELECT MAX(OrderDate)
                        FROM Sales.SalesOrderHeader
                        WHERE SalesPersonID =
                              sp.SalesPersonID);
GO
```

The previous example assumes that only one sale is recorded for a specified salesperson on a specific date and that updates are current. If more than one sale for a specified salesperson can be recorded on the same day, the example shown does not work correctly. The example runs without error, but each SalesYTD value is updated with only one sale, regardless of how many sales actually occurred on that day. This is because a single UPDATE statement never updates the same row two times.

In the situation in which more than one sale for a specified salesperson can occur on the same day, all the sales for each sales person must be aggregated together within the UPDATE statement, as shown in this example:

```
USE AdventureWorks;
GO
UPDATE Sales.SalesPerson
SET SalesYTD = SalesYTD +
    (SELECT SUM(so.SubTotal)
     FROM Sales.SalesOrderHeader AS so
     WHERE so.OrderDate = (SELECT MAX(OrderDate)
                           FROM Sales.SalesOrderHeader AS so2
                           WHERE so2.SalesPersonID =
                                 so.SalesPersonID)
     AND Sales.SalesPerson.SalesPersonID = so.SalesPersonID
     GROUP BY so.SalesPersonID);
GO
```

Using UPDATE with the TOP clause

The following example updates the VacationHours column by 25 percent for 10 random rows in the Employee table.

```
USE AdventureWorks;
GO
UPDATE TOP (10) HumanResources.Employee
SET VacationHours = VacationHours * 1.25 ;
GO
```

DELETE statement

Removes rows from a table or view.

Syntax

```
[ WITH <common_table_expression> [ ,...n ] ]
DELETE
    [ TOP ( expression ) [ PERCENT ] ]
    [ FROM ]
    { <object> | rowset_function_limited
      [ WITH ( <table_hint_limited> [ ...n ] ) ]
    }
    [ <OUTPUT Clause> ]
    [ FROM <table_source> [ ,...n ] ]
    [ WHERE { <search_condition>
            | { [ CURRENT OF
                   { { [ GLOBAL ] cursor_name }
                       | cursor_variable_name
                   }
                ]
              }
            }
    ]
    [ OPTION ( <Query Hint> [ ,...n ] ) ]
[; ]

<object> ::=
{
    [ server_name.database_name.schema_name.
      | database_name. [ schema_name ] .
      | schema_name.
    ]
        table_or_view_name
}
```

Arguments

`WITH <common_table_expression>` :

- Specifies the temporary named result set, also known as common table expression, defined within the scope of the DELETE statement. The result set is derived from a SELECT statement.
- Common table expressions can also be used with the SELECT, INSERT, UPDATE, and CREATE VIEW statements.

`TOP ( expression ) [ PERCENT ]`:

- Specifies the number or percent of random rows that will be deleted. *expression* can be either a number or a percent of the rows. The rows referenced in the TOP expression used with INSERT, UPDATE, or DELETE are not arranged in any order.
- Parentheses delimiting *expression* in TOP are required in INSERT, UPDATE, and DELETE statements.

FROM: Is an optional keyword that can be used between the DELETE keyword and the target *table_or_view_name*, or *rowset_function_limited*.

server_name : Is the name of the server (using a linked server name or the OPENDATASOURCE (Transact-SQL) function as the server name) on which the table or view is located. If *server_name* is specified, *database_name* and *schema_name* are required.

database_name: Is the name of the database.

schema_name: Is the name of the schema to which the table or view belongs

`table_or view_name`

- Is the name of the table or view from which the rows are to be removed.
- A **table** variable, within its scope, also can be used as a table source in a DELETE statement.
- The view referenced by *table_or_view_name* must be updatable and reference exactly one base table in the FROM clause of the view. For more information about updatable views.

rowset_function_limited: Is either the OPENQUERY (Transact-SQL) or OPENROWSET (Transact-SQL) function, subject to provider capabilities. For more information about the capabilities needed by the provider, see UPDATE and DELETE Requirements for OLE DB Providers.

WITH (<table_hint_limited> [... *n*]): Specifies one or more table hints that are allowed for a target table. The WITH keyword and the parentheses are required. NOLOCK and READUNCOMMITTED are not allowed.

<OUTPUT_Clause> : Returns deleted rows, or expressions based on them, as part of the DELETE operation. The OUTPUT clause is not supported in any DML statements targeting views or remote tables. For more information.

`FROM <table_source>:`

- Specifies an additional FROM clause. This Transact-SQL extension to DELETE allows specifying data from <table_source> and deleting the corresponding rows from the table in the first FROM clause.
- This extension, specifying a join, can be used instead of a subquery in the WHERE clause to identify rows to be removed.

`WHERE`:

- Specifies the conditions used to limit the number of rows that are deleted. If a WHERE clause is not supplied, DELETE removes all the rows from the table.
- There are two forms of delete operations based on what is specified in the WHERE clause:
    - Searched deletes specify a search condition to qualify the rows to delete. For example, WHERE *column_name* = *value*.
    - Positioned deletes use the CURRENT OF clause to specify a cursor. The delete operation occurs at the current position of the cursor. This can be more accurate than a searched DELETE statement that uses a WHERE *search_condition* clause to qualify the rows to be deleted. A searched DELETE statement deletes multiple rows if the search condition does not uniquely identify a single row.

`<search_condition>`: Specifies the restricting conditions for the rows to be deleted. There is no limit to the number of predicates that can be included in a search condition.

`CURRENT OF`: Specifies that the DELETE is performed at the current position of the specified cursor.

GLOBAL: Specifies that *cursor_name* refers to a global cursor.

`cursor_name` : Is the name of the open cursor from which the fetch is made. If both a global and a local cursor with the name *cursor_name* exist, this argument refers to the global cursor if GLOBAL is specified; otherwise, it refers to the local cursor. The cursor must allow updates.

`cursor_variable_name` : Is the name of a cursor variable. The cursor variable must reference a cursor that allows updates.

OPTION (<query_hint> [,... *n*]) : Are keywords that indicate that optimizer hints are used to customize the way the Database Engine processes the statement.

Examples

Using DELETE with no WHERE clause

The following example deletes all rows from the SalesPersonQuotaHistory table because a WHERE clause is not used to limit the number of rows deleted.

```
USE AdventureWorks;
GO
DELETE FROM Sales.SalesPersonQuotaHistory;
GO
```

Using DELETE on a set of rows

The following example deletes all rows from the ProductCostHistory table in which the value in the StandardCost column is more than 1000.00.

```
USE AdventureWorks;
GO
DELETE FROM Production.ProductCostHistory
WHERE StandardCost > 1000.00;
GO
```

Using DELETE on the current row of a cursor

The following example deletes a single row from the EmployeePayHistory table using a cursor named complex_cursor. The delete operation affects only the single row currently fetched from the cursor.

```
USE AdventureWorks;
GO
DECLARE complex_cursor CURSOR FOR
    SELECT a.EmployeeID
    FROM HumanResources.EmployeePayHistory AS a
    WHERE RateChangeDate <>
         (SELECT MAX(RateChangeDate)
          FROM HumanResources.EmployeePayHistory AS b
          WHERE a.EmployeeID = b.EmployeeID) ;
OPEN complex_cursor;
FETCH FROM complex_cursor;
DELETE FROM HumanResources.EmployeePayHistory
WHERE CURRENT OF complex_cursor;
CLOSE complex_cursor;
DEALLOCATE complex_cursor;
GO
```

Using DELETE based on a subquery and using the Transact-SQL extension

The following example shows the Transact-SQL extension used to delete records from a base table that is based on a join or correlated subquery. The first DELETE statement shows the SQL-2003-compatible subquery solution, and the second DELETE statement shows the Transact-SQL extension. Both queries remove rows from the SalesPersonQuotaHistory table based on the year-to-date sales stored in the SalesPerson table.

```
-- SQL-2003 Standard subquery

USE AdventureWorks;
GO
DELETE FROM Sales.SalesPersonQuotaHistory
WHERE SalesPersonID IN
    (SELECT SalesPersonID
     FROM Sales.SalesPerson
     WHERE SalesYTD > 2500000.00);
GO
```

```
-- Transact-SQL extension
USE AdventureWorks;
GO
DELETE FROM Sales.SalesPersonQuotaHistory
FROM Sales.SalesPersonQuotaHistory AS spqh
    INNER JOIN Sales.SalesPerson AS sp
    ON spqh.SalesPersonID = sp.SalesPersonID
WHERE sp.SalesYTD > 2500000.00;
GO
```

Using DELETE with the TOP clause

The following example deletes 2.5 percent of the rows (27 rows) in the ProductInventory table.

```
USE AdventureWorks;
GO
DELETE TOP (2.5) PERCENT
FROM Production.ProductInventory;
GO
```

TRUNCATE TABLE statement

Removes all rows from a table without logging the individual row deletions. TRUNCATE TABLE is functionally the same as the DELETE statement with no WHERE clause; however, TRUNCATE TABLE is faster and uses fewer system and transaction log resources.

Syntax

```
TRUNCATE TABLE
    [ { database_name.[ schema_name ]. | schema_name . } ]
    table_name
[ ; ]
```

Arguments

- **database_name:** Is the name of the database
- **schema_name:** Is the name of the schema to which the table belongs.
- **table_name:** Is the name of the table to truncate or from which all rows are removed.

Examples

The following example removes all data from the JobCandidate table. SELECT statements are included before and after the TRUNCATE TABLE statement to compare results.

```
USE AdventureWorks;
GO
SELECT COUNT(*) AS BeforeTruncateCount
```

```
FROM HumanResources.JobCandidate;
GO
TRUNCATE TABLE HumanResources.JobCandidate;
GO
SELECT COUNT(*) AS AfterTruncateCount
FROM HumanResources.JobCandidate;
GO
```

CREATE TABLE statement

Creates a new table.

Syntax

```
CREATE TABLE
    [ database_name . [ schema_name ] . | schema_name . ] table_name
        ( { <column_definition> | <computed_column_definition> }
        [ <table_constraint> ] [ ,...n ] )
    [ ON { partition_scheme_name ( partition_column_name ) | filegroup
        | "default" } ]
```

```
 [ PERSISTED [ NOT NULL ] ]
[
    [ CONSTRAINT constraint_name ]
    { PRIMARY KEY | UNIQUE }
        [ CLUSTERED | NONCLUSTERED ]

    | [ FOREIGN KEY ]
        REFERENCES referenced_table_name [ ( ref_column ) ]     | CHECK
[ NOT FOR REPLICATION ] ( logical_expression )
    [ ON { partition_scheme_name ( partition_column_name )
        | filegroup | "default" } ]
]

< table_constraint > ::=
[ CONSTRAINT constraint_name ]
{
    { PRIMARY KEY | UNIQUE }
        [ CLUSTERED | NONCLUSTERED ]
```

```
                (column [ ASC | DESC ] [ ,...n ] )
        [
            WITH FILLFACTOR = fillfactor
           |WITH ( <index_option> [ , ...n ] )
        ]
        [ ON { partition_scheme_name (partition_column_name)
```

```
            | filegroup | "default" } ]
    | FOREIGN KEY
               ( column [ ,...n ] )
        REFERENCES referenced_table_name [ ( ref_column [ ,...n ] ) ]
    | CHECK [ NOT FOR REPLICATION ] ( logical_expression )
}
```

Arguments

`database_name`: Is the name of the database in which the table is created. *database_name* must specify the name of an existing database. If not specified, *database_name* defaults to the current database . The login for the current connection must be associated with an existing user ID in the database specified by *database_name*, and that user ID must have CREATE TABLE permissions.

`schema_name`: Is the name of the schema to which the new table belongs.

`table_name`: Is the name of the new table. Table names must follow the rules for identifiers. *table_name* can be a maximum of 128 characters, except for local temporary table names (names prefixed with a single number sign (#)) that cannot exceed 116 characters.

`column_name`: Is the name of a column in the table. Column names must follow the rules for identifiers and must be unique in the table. *column_name* can be of 1 through 128 characters. *column_name* can be omitted for columns that are created with a **timestamp** data type. If *column_name* is not specified, the name of a **timestamp** column defaults to **timestamp**.

`computed_column_expression`:

- ❑ Is an expression that defines the value of a computed column. A computed column is a virtual column that is not physically stored in the table, unless the column is marked PERSISTED. The column is computed from an expression that uses other columns in the same table. For example, a computed column can have the definition: **cost** AS **price** * **qty**. The expression can be a noncomputed column name, constant, function, variable, and any combination of these connected by one or more operators. The expression cannot be a subquery or contain alias data types.
- ❑ Computed columns can be used in select lists, WHERE clauses, ORDER BY clauses, or any other locations in which regular expressions can be used, with the following exceptions:
    - o A computed column cannot be used as a DEFAULT or FOREIGN KEY constraint definition or with a NOT NULL constraint definition. However, a computed column can be used as a key column in an index or as part of any PRIMARY KEY or UNIQUE constraint, if the computed column value is defined by a deterministic expression and the data type of the result is allowed in index columns.
    - o A computed column cannot be the target of an INSERT or UPDATE statement.

`PERSISTED`: Specifies that the SQL Server Database Engine will physically store the computed values in the table, and update the values when any other columns on which the computed column depends are updated. Marking a computed column as PERSISTED lets you create an index on a computed column that is deterministic, but not precise. For more information, see Creating Indexes on Computed Columns. Any computed columns that are used as partitioning columns of a partitioned table must be explicitly marked PERSISTED. *computed_column_expression* must be deterministic when PERSISTED is specified.

`ON { <partition_scheme> | *filegroup* | "default" }:`

- Specifies the partition scheme or filegroup on which the table is stored. If <partition_scheme> is specified, the table is to be a partitioned table whose partitions are stored on a set of one or more filegroups specified in <partition_scheme>. If *filegroup* is specified, the table is stored in the named filegroup. The filegroup must exist within the database. If **"**default**"** is specified, or if ON is not specified at all, the table is stored on the default filegroup. The storage mechanism of a table as specified in CREATE TABLE cannot be subsequently altered.
- ON {<partition_scheme> **|** *filegroup* **|** **"**default**"**} can also be specified in a PRIMARY KEY or UNIQUE constraint. These constraints create indexes. If *filegroup* is specified, the index is stored in the named filegroup. If **"**default**"** is specified, or if ON is not specified at all, the index is stored in the same filegroup as the table. If the PRIMARY KEY or UNIQUE constraint creates a clustered index, the data pages for the table are stored in the same filegroup as the index. If CLUSTERED is specified or the constraint otherwise creates a clustered index, and a <partition_scheme> is specified that differs from the <partition_scheme> or *filegroup* of the table definition, or vice-versa, only the constraint definition will be honored, and the other will be ignored.

Example

Showing the complete table definition

The following example shows the complete table definitions with all constraint definitions for table PurchaseOrderDetail created in the **AdventureWorks** database.

```
CREATE TABLE [dbo].[PurchaseOrderDetail]
(
    [PurchaseOrderID] [int] NOT NULL
        REFERENCES Purchasing.PurchaseOrderHeader(PurchaseOrderID),
    [LineNumber] [smallint] NOT NULL,
    [ProductID] [int] NULL
        REFERENCES Production.Product(ProductID),
    [UnitPrice] [money] NULL,
    [OrderQty] [smallint] NULL,
    [ReceivedQty] [float] NULL,
    [RejectedQty] [float] NULL,
    [DueDate] [datetime] NULL,
    [rowguid] [uniqueidentifier] ROWGUIDCOL  NOT NULL
        CONSTRAINT [DF_PurchaseOrderDetail_rowguid] DEFAULT (newid()),
    [ModifiedDate] [datetime] NOT NULL
        CONSTRAINT [DF_PurchaseOrderDetail_ModifiedDate] DEFAULT
(getdate()),
    [LineTotal]  AS (([UnitPrice]*[OrderQty])),
    [StockedQty]  AS (([ReceivedQty]-[RejectedQty])),
CONSTRAINT [PK_PurchaseOrderDetail_PurchaseOrderID_LineNumber]
    PRIMARY KEY CLUSTERED ([PurchaseOrderID], [LineNumber])
    WITH (IGNORE_DUP_KEY = OFF)
)
ON [PRIMARY]
```

Functions

SQL Server 2005 provides built-in functions that can be used to perform certain operations. The following table lists the categories for the SQL Server functions.

Function category	Description
Aggregate Functions (Transact-SQL)	Perform operations that combine multiple values into one. Examples are COUNT, SUM, MIN, and MAX.
Configuration functions	Scalar functions that return information about configuration settings.
Cryptographic Functions (Transact-SQL)	Support encryption, decryption, digital signing, and the validation of digital signatures.
Cursor functions	Return information about the status of a cursor.
Date and Time functions	Change date and time values.
Mathematical functions	Perform trigonometric, geometric, and other numeric operations.
Metadata functions	Return information about the attributes of databases and database objects.
Ranking functions	Nondeterministic functions that return a ranking value for each row in a partition.
Rowset Functions (Transact-SQL)	Return the rowsets that can be used in the place of a table reference in a Transact-SQL statement.
Security functions	Return information about users and roles.
String functions	Change char, varchar, nchar, nvarchar, binary, and varbinary values.
System functions	Operate on or report on various system level options and objects.
System Statistical Functions (Transact-SQL)	Return information about the performance of SQL Server.
Text and image functions	Change text and image values.

Deterministic and Nondeterministic Functions

In SQL Server 2005, functions are classified as strictly deterministic, deterministic, or nondeterministic.

A function is strictly deterministic if, for a specific set of input values, the function always returns the same results.

For user-defined functions, a less rigid notion of determinism is applied. A user-defined function is deterministic if, for a specific set of input values and database state, the function always returns the same results. If the function is not strictly deterministic, it can be deterministic in this sense if it is data-accessing.

A nondeterministic function may return different results when it is called repeatedly with the same set of input values. For example, the function GETDATE() is nondeterministic. SQL Server puts restrictions on various classes of nondeterminism. Therefore, nondeterministic functions should be used carefully.

Aggregate Functions

Aggregate functions perform a calculation on a set of values and return a single value. Except for COUNT, aggregate functions ignore null values. Aggregate functions are frequently used with the GROUP BY clause of the SELECT statement.

All aggregate functions are deterministic. This means aggregate functions return the same value any time they are called by using a specific set of input values. For more information about function determinism, see Deterministic and Nondeterministic Functions.

Aggregate functions can be used as expressions only in the following:

- The select list of a SELECT statement (either a subquery or an outer query).
- A COMPUTE or COMPUTE BY clause.
- A HAVING clause.

Transact-SQL provides the following aggregate functions:

AVG	MIN
CHECKSUM	SUM
CHECKSUM_AGG	STDEV
COUNT	STDEVP
COUNT_BIG	VAR
GROUPING	VARP
MAX	

Example:

Using the SUM and AVG functions for calculations

The following example calculates the average vacation hours and the sum of sick leave hours that the vice presidents of Adventure Works Cycles have used. Each of these aggregate functions produces a single summary value for all the retrieved rows.

```
USE AdventureWorks;
GO
SELECT AVG(VacationHours)as 'Average vacation hours',
    SUM  (SickLeaveHours) as 'Total sick leave hours'
FROM HumanResources.Employee
WHERE Title LIKE 'Vice President%';
```

Date and Time Functions

The following scalar functions perform an operation on a date and time input value and return a string, numeric, or date and time value.

The Transact-SQL date and time functions and their determinism property are listed in the following table.

Function	Determinism
`DATEADD`	Deterministic
`DATEDIFF`	Deterministic
`DATENAME`	Nondeterministic
`DATEPART`	Deterministic except when used as DATEPART (dw, date). dw, the weekday datepart, depends on the value set by SET DATEFIRST, which sets the first day of the week.
`DAY`	Deterministic
`GETDATE`	Nondeterministic
`GETUTCDATE`	Nondeterministic
`MONTH`	Deterministic
`YEAR`	Deterministic

Syntax

DATEADD (*datepart* , number, *date*) : Returns a new **datetime** value based on adding an interval to the specified date.

DATEDIFF (datepart , startdate , enddate) : Returns the number of date and time boundaries crossed between two specified dates.

DATENAME (datepart ,date) : Returns a character string representing the specified datepart of the specified date.

DATEPART (datepart , date) : Returns an integer that represents the specified datepart of the specified date.

DAY (date) : Returns an integer representing the day

GETDATE ( ) : Returns the current system date and time

Arguments

datepart: Is the parameter that specifies on which part of the date to return a new value. The following table lists the dateparts and abbreviations recognized by Microsoft SQL Server 2005.

Datepart	Abbreviations
year	yy, yyyy
quarter	qq, q

Datepart	Abbreviations
month	mm, m
dayofyear	dy, y
day	dd, d
week	wk, ww
weekday	dw, w
hour	hh
minute	mi, n
second	ss, s
millisecond	ms

number: Is the value used to increment *datepart*. If you specify a value that is not an integer, the fractional part of the value is discarded. For example, if you specify **day** for *datepart* and**1.75** for *number*, *date* is incremented by 1.

date:

- Is an expression that returns a datetime or smalldatetime value, or a character string in a date format.
- If you specify only the last two digits of the year, values less than or equal to the last two digits of the value of the **two digit year cutoff** configuration option are in the same century as the cutoff year. Values greater than the last two digits of the value of this option are in the century that comes before the cutoff year. For example, if **two-digit year cutoff** is 2049 (default), 49 is interpreted as 2049 and 2050 is interpreted as 1950. To avoid ambiguity, use four-digit years.

Examples

The following example prints a listing of a time frame for orders in the AdventureWorks database. This time frame represents the existing order date plus 21 days.

```
USE AdventureWorks;
GO
SELECT DATEADD(day, 21, OrderDate)AS TimeFrame
FROM Sales.SalesOrderHeader;
GO
```

The following example determines the difference in days between the current date and the order date for products in the AdventureWorks database.

```
USE AdventureWorks;
GO
SELECT DATEDIFF(day, OrderDate, GETDATE()) AS NumberOfDays
FROM Sales.SalesOrderHeader;
```

```
GO
```

String Functions

The following scalar functions perform an operation on a string input value and return a string or numeric value:

ASCII	NCHAR	SOUNDEX
CHAR	PATINDEX	SPACE
CHARINDEX	QUOTENAME	STR
DIFFERENCE	REPLACE	STUFF
LEFT	REPLICATE	SUBSTRING
LEN	REVERSE	UNICODE
LOWER	RIGHT	UPPER
LTRIM	RTRIM	

All built-in string functions are deterministic. This means they return the same value any time they are called with a specific set of input values.

Examples

Using SUBSTRING with a character string

The following example shows how to return only a part of a character string. From the Contact table, this query returns the last name in one column with only the first initial in the second column.

```
USE AdventureWorks;
GO
SELECT LastName, SUBSTRING(FirstName, 1, 1) AS Initial
FROM Person.Contact
WHERE LastName like 'Barl%'
ORDER BY LastName
```

The following example replaces the string cde in abcdefghi with xxx.

```
SELECT REPLACE('abcdefghicde','cde','xxx');
GO
```

The following example uses LTRIM to remove leading spaces from a character variable.

```
DECLARE @string_to_trim varchar(60)
SET @string_to_trim = '     Five spaces are at the beginning of this
   string.'
SELECT 'Here is the string without the leading spaces: ' +
   LTRIM(@string_to_trim)
GO
```

Using LEFT with a column

The following example returns the five leftmost characters of each product name.

```
USE AdventureWorks;
GO
SELECT LEFT(Name, 5)
FROM Production.Product
ORDER BY ProductID;
GO
```

Ranking Functions

RANK() Function:

- Returns the rank of each row within the partition of a result set
- The rank of a row is one plus the number of ranks that come before the row in question

Example

The following simple example gives a row number to each row in the result set, ordering by SalesOrderID:

```
USE AdventureWorks
GO
SELECT SalesOrderID, CustomerID, RANK() OVER (ORDER BY CustomerID)
AS RunningCount FROM Sales.SalesOrderHeader
WHERE SalesOrderID > 10000
```

`ROW_NUMBER()` Function:

Returns the sequential number of a row within a partition of a result set, starting at one for the first row in each partition

Example

```
USE AdventureWorks
GO
SELECT SalesOrderID, CustomerID,
   ROW_NUMBER() OVER (ORDER BY SalesOrderID) AS RunningCount
FROM Sales.SalesOrderHeader
WHERE SalesOrderID>10000
```

Common Table Expressions

A Common Table Expression (CTE) closely resembles a nonpersistent view.

CTE provides the significant advantage of being able to reference itself, thereby creating a recursive CTE.

- A recursive CTE is one in which an initial CTE is repeatedly executed to return subsets of data until the complete result set is obtained.
- Simplify the code required to run a recursive query within a SELECT, INSERT, UPDATE, DELETE, or CREATE VIEW statement.

- CTE reduces the complexity of applying recursive query, which usually requires applying temporary tables, cursors, and logic to control the flow of the recursive steps in earlier version of SQL Server.

```
/*Example shows the number of employees reporting directly to each
manager at Adventure Works Cycles. */
USE AdventureWorks;
GO
WITH DirReps(ManagerID, DirectReports) AS
(
    SELECT ManagerID, COUNT(*)
    FROM HumanResources.Employee AS e
    WHERE ManagerID IS NOT NULL
    GROUP BY ManagerID
)
SELECT ManagerID, DirectReports
FROM DirReps
ORDER BY ManagerID;
GO
```

Summary

- Transact SQL (T-SQL) is a powerful and unique superset of the SQL standard.
- GROUP BY calculates a summary value for each group
- ORDER BY specifies the sort order used on columns returned in a SELECT statement
- A subquery is a query that is nested inside a SELECT, INSERT, UPDATE, or DELETE statement, or inside another subquery.
- TRUNCATE TABLE statement removes all rows from a table without logging the individual row deletions
- A Common Table Expression (CTE) closely resembles a nonpersistent view.

Session 15: Database and Data files

Learning Objectives

After completing this session, you will be able to:

- Identify the types of Database Files
- Explain the Database Filegroups
- Describe different ways of Creating Databases

Overview

SQL Server 2005 maps a database over a set of operating-system files. Data and log information are never mixed in the same file, and individual files are used only by one database. Filegroups are named collections of files and are used to help with data placement and administrative tasks such as backup and restore operations.

Database Files

SQL Server 2005 databases have three types of files:

- Primary data files: The primary data file is the starting point of the database and points to the other files in the database. Every database has one primary data file. The recommended file name extension for primary data files is .mdf.
- Secondary data files: Secondary data files make up all the data files, other than the primary data file. Some databases may not have any secondary data files, while others have several secondary data files. The recommended file name extension for secondary data files is .ndf.
- Log files: Log files hold all the log information that is used to recover the database. There must be at least one log file for each database, although there can be more than one. The recommended file name extension for log files is .ldf.

SQL Server 2005 does not enforce the .mdf, .ndf, and .ldf file name extensions, but these extensions help you identify the different kinds of files and their use.

In SQL Server 2005, the locations of all the files in a database are recorded in the primary file of the database and in the **master** database. The Database Engine uses the file location information from the **master** database most of the time. However, the database engine uses the file location information from the primary file to initialize the file location entries in the **master** database in the following situations:

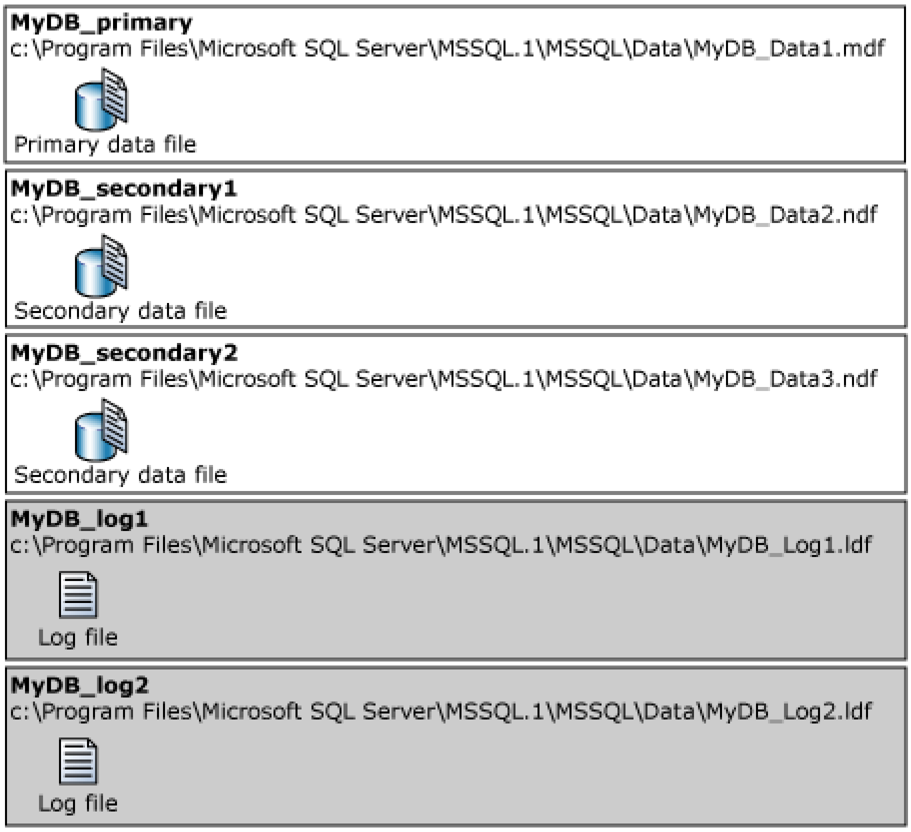
- When attaching a database using the CREATE DATABASE statement with either the FOR ATTACH or FOR ATTACH_REBUILD_LOG options.
- When upgrading from SQL Server version 2000 or version 7.0 to SQL Server 2005.

When restoring the **master** database.

Logical and Physical File Names

SQL Server 2005 files have two names:

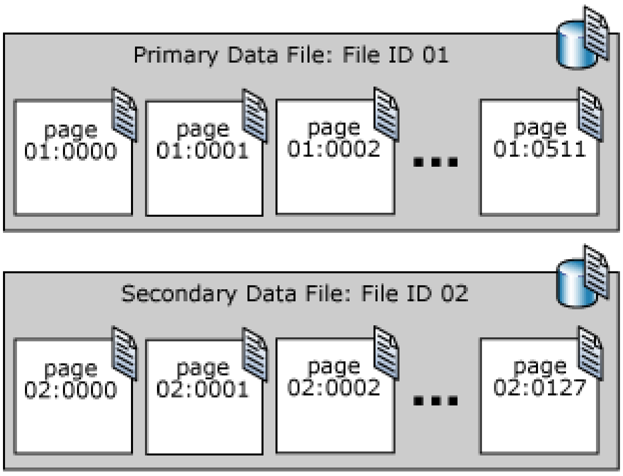
- **logical_file_name:** The *logical_file_name* is the name used to refer to the physical file in all Transact-SQL statements. The logical file name must comply with the rules for SQL Server identifiers and must be unique among logical file names in the database.
- **os_file_name:**
  - The *os_file_name* is the name of the physical file including the directory path. It must follow the rules for the operating system file names.
  - The following illustration shows examples of the logical file names and the physical file names of a database created on a default instance of SQL Server 2005:


MyDB_primary
c:\Program Files\Microsoft SQL Server\MSSQL.1\MSSQL\Data\MyDB_Data1.mdf
Primary data file
MyDB_secondary1
c:\Program Files\Microsoft SQL Server\MSSQL.1\MSSQL\Data\MyDB_Data2.ndf
Secondary data file
MyDB_secondary2
c:\Program Files\Microsoft SQL Server\MSSQL.1\MSSQL\Data\MyDB_Data3.ndf
Secondary data file
MyDB_log1
c:\Program Files\Microsoft SQL Server\MSSQL.1\MSSQL\Data\MyDB_Log1.ldf
Log file
MyDB_log2
c:\Program Files\Microsoft SQL Server\MSSQL.1\MSSQL\Data\MyDB_Log2.ldf
Log file

SQL Server data and log files can be put on either FAT or NTFS file systems. NTFS is recommended for the security aspects of NTFS. Read/write data filegroups and log files cannot be placed on an NTFS compressed file system. Only read-only databases and read-only secondary filegroups can be put on an NTFS compressed file system.

When multiple instances of SQL Server are run on a single computer, each instance receives a different default directory to hold the files for the databases created in the instance. For more information, see File Locations for Default and Named Instances of SQL Server 2005.

Data File Pages: Pages in a SQL Server 2005 data file are numbered sequentially, starting with zero (0) for the first page in the file. Each file in a database has a unique file ID number. To uniquely identify a page in a database, both the file ID and the page number are required. The following example shows the page numbers in a database that has a 4-MB primary data file and a 1-MB secondary data file.



The first page in each file is a file header page that contains information about the attributes of the file. Several of the other pages at the start of the file also contain system information, such as allocation maps. One of the system pages stored in both the primary data file and the first log file is a database boot page that contains information about the attributes of the database

File Size:

SQL Server 2005 files can grow automatically from their originally specified size. When you define a file, you can specify a specific growth increment. Every time the file is filled, it increases its size by the growth increment. If there are multiple files in a filegroup, they will not autogrow until all the files are full. Growth then occurs in a round-robin fashion.

Each file can also have a maximum size specified. If a maximum size is not specified, the file can continue to grow until it has used all available space on the disk. This feature is especially useful when SQL Server is used as a database embedded in an application where the user does not have convenient access to a system administrator. The user can let the files autogrow as required to reduce the administrative burden of monitoring free space in the database and manually allocating additional space.

Database Filegroups

Database objects and files can be grouped together in filegroups for allocation and administration purposes. There are two types of filegroups:

- **Primary:** The primary filegroup contains the primary data file and any other files not specifically assigned to another filegroup. All pages for the system tables are allocated in the primary filegroup.
- **User-defined:** User-defined filegroups are any filegroups that are specified by using the FILEGROUP keyword in a CREATE DATABASE or ALTER DATABASE statement.

Log files are never part of a filegroup. Log space is managed separately from data space.

No file can be a member of more than one filegroup. Tables, indexes, and large object data can be associated with a specified filegroup. In this case, all their pages will be allocated in that filegroup, or the tables and indexes can be partitioned. The data of partitioned tables and indexes is divided into units each of which can be placed in a separate filegroup in a database.

One filegroup in each database is designated the default filegroup. When a table or index is created without specifying a filegroup, it is assumed all pages will be allocated from the default filegroup. Only one filegroup at a time can be the default filegroup. Members of the **db_owner** fixed database role can switch the default filegroup from one filegroup to another. If no default filegroup is specified, the primary filegroup is the default filegroup.

File and Filegroup Example

The following example creates a database on an instance of SQL Server. The database has a primary data file, a user-defined filegroup, and a log file. The primary data file is in the primary filegroup and the user-defined filegroup has two secondary data files. An ALTER DATABASE statement makes the user-defined filegroup the default. A table is then created specifying the user-defined filegroup.

```
USE master;
GO
-- Create the database with the default data
-- filegroup and a log file. Specify the
-- growth increment and the max size for the
-- primary data file.
CREATE DATABASE MyDB
ON PRIMARY
  ( NAME='MyDB_Primary',
    FILENAME=
       'c:\Program Files\Microsoft SQL
Server\MSSQL.1\MSSQL\data\MyDB_Prm.mdf',
    SIZE=4MB,
    MAXSIZE=10MB,
    FILEGROWTH=1MB),
FILEGROUP MyDB_FG1
  ( NAME = 'MyDB_FG1_Dat1',
    FILENAME =
       'c:\Program Files\Microsoft SQL
Server\MSSQL.1\MSSQL\data\MyDB_FG1_1.ndf',
    SIZE = 1MB,
    MAXSIZE=10MB,
    FILEGROWTH=1MB),
  ( NAME = 'MyDB_FG1_Dat2',
    FILENAME =
       'c:\Program Files\Microsoft SQL
Server\MSSQL.1\MSSQL\data\MyDB_FG1_2.ndf',
    SIZE = 1MB,
    MAXSIZE=10MB,
    FILEGROWTH=1MB)
LOG ON
  ( NAME='MyDB_log',
    FILENAME =
       'c:\Program Files\Microsoft SQL
Server\MSSQL.1\MSSQL\data\MyDB.ldf',
    SIZE=1MB,
    MAXSIZE=10MB,
    FILEGROWTH=1MB);
GO
ALTER DATABASE MyDB
  MODIFY FILEGROUP MyDB_FG1 DEFAULT;
GO

-- Create a table in the user-defined filegroup.
USE MyDB;
CREATE TABLE MyTable
  ( cola int PRIMARY KEY,
    colb char(8) )
ON MyDB_FG1;
GO
```

The following illustration summarizes the results of the previous example.

CREATE DATABASE (T-SQL Statement)

Creates a new database and the files used to store the database, creates a database snapshot, or attaches a database from the detached files of a previously created database.

Syntax

```
CREATE DATABASE database_name
    [ ON
        [ PRIMARY ] [ <filespec> [ ,...n ]
        [ , <filegroup> [ ,...n ] ]
    [ LOG ON { <filespec> [ ,...n ] } ]
    ]
    [ COLLATE collation_name ]
    [ WITH <external_access_option> ]
]
[;]

To attach a database
CREATE DATABASE database_name
    ON <filespec> [ ,...n ]
    FOR { ATTACH [ WITH <service_broker_option> ]
        | ATTACH_REBUILD_LOG }
[;]

<filespec> ::=
{
```

```
(
    NAME = logical_file_name ,
    FILENAME = 'os_file_name'
        [ , SIZE = size [ KB | MB | GB | TB ] ]
        [ , MAXSIZE = { max_size [ KB | MB | GB | TB ] | UNLIMITED } ]
        [ , FILEGROWTH = growth_increment [ KB | MB | GB | TB | % ] ]
) [ ,...n ]
}

<filegroup> ::=
{
FILEGROUP filegroup_name [ DEFAULT ]
    <filespec> [ ,...n ]
}

<external_access_option> ::=
{
    DB_CHAINING { ON | OFF }
  | TRUSTWORTHY { ON | OFF }
}
<service_broker_option> ::=
{
    ENABLE_BROKER
  | NEW_BROKER
  | ERROR_BROKER_CONVERSATIONS
}

Create a database snapshot
CREATE DATABASE database_snapshot_name
    ON
        (
        NAME = logical_file_name,
        FILENAME = 'os_file_name'
        ) [ ,...n ]
    AS SNAPSHOT OF source_database_name
[;]
```

Arguments

`database_name`

- Is the name of the new database. Database names must be unique within an instance of SQL Server and comply with the rules for identifiers.
- *database_name* can be a maximum of 128 characters, unless a logical name is not specified for the log file. If a logical log file name is not specified, SQL Server generates the *logical_file_name* and the *os_file_name* for the log by appending a

suffix to *database_name*. This limits *database_name* to 123 characters so that the generated logical file name is no more than 128 characters.

- If data file name is not specified, SQL Server uses *database_name* as both the *logical_file_name* and as the *os_file_name*.

ON: Specifies that the disk files used to store the data sections of the database, data files, are explicitly defined. ON is required when followed by a comma-separated list of <filespec> items that define the data files for the primary filegroup. The list of files in the primary filegroup can be followed by an optional, comma-separated list of <filegroup> items that define user filegroups and their files.

PRIMARY:

- Specifies that the associated <filespec> list defines the primary file. The first file specified in the <filespec> entry in the primary filegroup becomes the primary file. A database can have only one primary file. For more information, see Physical Database Files and Filegroups.
- If PRIMARY is not specified, the first file listed in the CREATE DATABASE statement becomes the primary file.

LOG ON: Specifies that the disk files used to store the database log, log files, are explicitly defined. LOG ON is followed by a comma-separated list of <filespec> items that define the log files. If LOG ON is not specified, one log file is automatically created that has a size that is 25 percent of the sum of the sizes of all the data files for the database or 512 KB, whichever is larger. LOG ON cannot be specified on a database snapshot.

COLLATE collation_name:

- Specifies the default collation for the database. Collation name can be either a Windows collation name or a SQL collation name. If not specified, the database is assigned the default collation of the instance of SQL Server. A collation name cannot be specified on a database snapshot.
- A collation name cannot be specified with the FOR ATTACH or FOR ATTACH_REBUILD_LOG clauses..

FOR ATTACH: Specifies that the database is created by attaching an existing set of operating system files. There must be a <filespec> entry that specifies the primary file. The only other <filespec> entries required are those for any files that have a different path from when the database was first created or last attached. A <filespec> entry must be specified for these files. FOR ATTACH requires the following:

- All data files (MDF and NDF) must be available.
- If multiple log files exist, they must all be available.

If a read/write database has a single log file that is currently unavailable, and if the database was shut down with no users or open transactions before the attach operation, FOR ATTACH automatically rebuilds the log file and updates the primary file. In contrast, for a read-only database, the log cannot be rebuilt because the primary file cannot be updated. Therefore, when you attach a read-only database whose log is unavailable, you must provide the log files or files in the FOR ATTACH clause.

In SQL Server 2005, any full-text files that are part of the database that is being attached will be attached with the database. To specify a new path of the full-text catalog, specify the new location without the full-text operating system file name. For more information, see the Examples section. FOR ATTACH cannot be specified on a database snapshot.

`<filespec>`: Controls the file properties

`NAME logical_file_name:` Specifies the logical name for the file. NAME is required when FILENAME is specified, except when specifying one of the FOR ATTACH clauses.

`logical_file_name:` Is the logical name used in SQL Server when referencing the file. *logical_file_name* must be unique in the database and comply with the rules for identifiers. The name can be a character or Unicode constant, or a regular or delimited identifier.

`FILENAME '` ***`os_file_name`*** **`':`** Specifies the operating system (physical) file name.

`' os_file_name ':`

- Is the path and file name used by the operating system when you create the file. The file must reside on one of the following devices: the local server on which SQL Server is installed, a Storage Area Network [SAN], or an iSCSI-based network. The specified path must exist before executing the CREATE DATABASE statement. For more information, see "Database Files and Filegroups" in the Remarks section.
- SIZE, MAXSIZE, and FILEGROWTH parameters cannot be set when a UNC path is specified for the file.
- If the file is on a raw partition, *os_file_name* must specify only the drive letter of an existing raw partition. Only one data file can be created on each raw partition.
- Data files should not be put on compressed file systems unless the files are read-only secondary files, or the database is read-only. Log files should never be put on compressed file systems. For more information, see Read-Only Filegroups.

`SIZE size:`

- Specifies the size of the file.
- SIZE cannot be specified when the *os_file_name* is specified as a UNC path.

`size:`

- Is the initial size of the file.
- When *size* is not supplied for the primary file, the Database Engine uses the size of the primary file in the **model** database. When a secondary data file or log file is specified but *size* is not specified for the file, the Database Engine makes the file 1 MB. The size specified for the primary file must be at least as large as the primary file of the **model** database.
- The kilobyte (KB), megabyte (MB), gigabyte (GB), or terabyte (TB) suffixes can be used. The default is MB. Specify a whole number; do not include a decimal.

`MAXSIZE max_size:` Specifies the maximum size to which the file can grow. MAXSIZE cannot be specified when the *os_file_name* is specified as a UNC path.

`max_size:` Is the maximum file size. The KB, MB, GB, and TB suffixes can be used. The default is MB. Specify a whole number; do not include a decimal. If *max_size* is not specified, the file grows until the disk is full.

`UNLIMITED`: Specifies that the file grows until the disk is full. In SQL Server 2005, a log file specified with unlimited growth has a maximum size of 2 TB, and a data file has a maximum size of 16 TB.

`FILEGROWTH growth_increment:` Specifies the automatic growth increment of the file. The FILEGROWTH setting for a file cannot exceed the MAXSIZE setting. FILEGROWTH cannot be specified when the *os_file_name* is specified as a UNC path.

`growth_increment`:

- Is the amount of space added to the file every time new space is required.
- The value can be specified in MB, KB, GB, TB, or percent (%). If a number is specified without an MB, KB, or % suffix, the default is MB. When % is specified, the growth increment size is the specified percentage of the size of the file at the time the increment occurs. The size specified is rounded to the nearest 64 KB.
- A value of 0 indicates that automatic growth is off and no additional space is allowed.

If FILEGROWTH is not specified, the default value is 1 MB for data files and 10% for log files, and the minimum value is 64 KB.

`<filegroup>` : Controls the filegroup properties. Filegroup cannot be specified on a database snapshot.

`FILEGROUP filegroup_name:` Is the logical name of the filegroup

`filegroup_name`: *filegroup_name* must be unique in the database and cannot be the system-provided names PRIMARY and PRIMARY_LOG. The name can be a character or Unicode constant, or a regular or delimited identifier. The name must comply with the rules for identifiers.

`DEFAULT`: Specifies the named filegroup is the default filegroup in the database.

`<external_access_option>:` Controls external access to and from the database.

`DB_CHAINING { ON | OFF }:`

- When ON is specified, the database can be the source or target of a cross-database ownership chain.
- When OFF, the database cannot participate in cross-database ownership chaining. The default is OFF.
- To set this option, requires membership in the **sysadmin** fixed server role. The DB_CHAINING option cannot be set on these system databases: **master**, **model**, **tempdb**.

`TRUSTWORTHY { ON | OFF }:`

- When ON is specified, database modules (for example, views, user-defined functions, or stored procedures) that use an impersonation context can access resources outside the database.
- When OFF, database modules in an impersonation context cannot access resources outside the database. The default is OFF.
- TRUSTWORTHY is set to OFF whenever the database is attached.
- By default, the **master** database always has TRUSTWORTHY set to ON. The **model**, **tempdb** databases always have TRUSTWORTHY set to OFF and the value cannot be changed for these databases.
- To set this option, requires membership in the **sysadmin** fixed server role .

`<service_broker_option>`:

- Controls Service Broker options on the database.
- Service Broker options can only be specified when the FOR ATTACH clause is used.

ENABLE_BROKER: Specifies that Service Broker is enabled for the specified database. That is, **is_broker_enabled** is set to true in the **sys.databases** catalog view and message delivery is started.

NEW_BROKER: Creates a new **service_broker_guid** in **sys.databases** and the restored database.

ERROR_BROKER_CONVERSATIONS: Ends all conversations with an error that indicates a copy of the broker has been created.

database_snapshot_name: Is the name of the new database snapshot. Database snapshot names must be unique within an instance of SQL Server and comply with the rules for identifiers. *database_snapshot_name* can be a maximum of 128 characters.

ON **(** NAME **=** *logical_file_name*, FILENAME **=** '*os_file_name*' **)** [**,...** *n*]:

- For creating a database snapshot, specifies a list of files in the source database. For the snapshot to work, all the data files must be specified individually. However, log files are not allowed for database snapshots.
- For descriptions of NAME and FILENAME and their values see the descriptions of the equivalent <filespec> values.

AS SNAPSHOT OF source_database_name: Specifies that the database being created is a database snapshot of the source database specified by *source_database_name*. The snapshot and source database must be on the same instance.

Examples

Creating a database without specifying files

The following example creates the database mytest and creates a corresponding primary and transaction log file. Because the statement has no <filespec> items, the primary database file is the size of the **model** database primary file. The transaction log is set to the larger of these values: 512KB or 25% the size of the primary data file. Because MAXSIZE is not specified, the files can grow to fill all available disk space.

```
USE master;
GO
IF DB_ID (N'mytest') IS NOT NULL
DROP DATABASE mytest;
GO
CREATE DATABASE mytest;
GO
-- Verify the database files and sizes
SELECT name, size, size*1.0/128 AS [Size in MBs]
FROM sys.master_files
WHERE name = N'mytest';
GO
```

Creating a database that specifies the data and transaction log files

The following example creates the database Sales. Because the keyword PRIMARY is not used, the first file (Sales_dat) becomes the primary file. Because neither MB nor KB is specified in the SIZE parameter for the Sales_dat file, it uses MB and is allocated in megabytes. The Sales_log file is allocated in megabytes because the MB suffix is explicitly stated in the SIZE parameter.

```
USE master;
GO
IF DB_ID (N'Sales') IS NOT NULL
DROP DATABASE Sales;
GO
-- Get the SQL Server data path
DECLARE @data_path nvarchar(256);
SET @data_path = (SELECT SUBSTRING(physical_name, 1,
CHARINDEX(N'master.mdf', LOWER(physical_name)) - 1)
                  FROM master.sys.master_files
                  WHERE database_id = 1 AND file_id = 1);

-- execute the CREATE DATABASE statement
EXECUTE ('CREATE DATABASE Sales
ON
( NAME = Sales_dat,
    FILENAME = '''+ @data_path + 'saledat.mdf'',
    SIZE = 10,
    MAXSIZE = 50,
    FILEGROWTH = 5 )
LOG ON
( NAME = Sales_log,
    FILENAME = '''+ @data_path + 'salelog.ldf'',
    SIZE = 5MB,
    MAXSIZE = 25MB,
    FILEGROWTH = 5MB )'
);
GO

GO
```

Creating a database by specifying multiple data and transaction log files

The following example creates the database Archive that has three 100-MB data files and two 100-MB transaction log files. The primary file is the first file in the list and is explicitly specified with the PRIMARY keyword. The transaction log files are specified following the LOG ON keywords. Note the extensions used for the files in the FILENAME option: .mdf is used for primary data files, .ndf is used for the secondary data files, and .ldf is used for transaction log files.

```
USE master;
GO
IF DB_ID (N'Archive') IS NOT NULL
DROP DATABASE Archive;
GO
-- Get the SQL Server data path
DECLARE @data_path nvarchar(256);
SET @data_path = (SELECT SUBSTRING(physical_name, 1,
CHARINDEX(N'master.mdf', LOWER(physical_name)) - 1)
                  FROM master.sys.master_files
                  WHERE database_id = 1 AND file_id = 1);

-- execute the CREATE DATABASE statement
EXECUTE ('CREATE DATABASE Archive
ON
PRIMARY
    (NAME = Arch1,
    FILENAME = '''+ @data_path + 'archdat1.mdf'',
    SIZE = 100MB,
    MAXSIZE = 200,
    FILEGROWTH = 20),
    ( NAME = Arch2,
    FILENAME = '''+ @data_path + 'archdat2.ndf'',
    SIZE = 100MB,
    MAXSIZE = 200,
    FILEGROWTH = 20),
    ( NAME = Arch3,
    FILENAME = '''+ @data_path + 'archdat3.ndf'',
    SIZE = 100MB,
    MAXSIZE = 200,
    FILEGROWTH = 20)
LOG ON
   (NAME = Archlog1,
    FILENAME = '''+ @data_path + 'archlog1.ldf'',
    SIZE = 100MB,
    MAXSIZE = 200,
    FILEGROWTH = 20),
   (NAME = Archlog2,
    FILENAME = '''+ @data_path + 'ldf'',
    SIZE = 100MB,
    MAXSIZE = 200,
    FILEGROWTH = 20)'
);
GO
```

Creating a database that has filegroups

The following example creates the database Sales that has the following filegroups:

- The primary filegroup with the files Spri1_dat and Spri2_dat. The FILEGROWTH increments for these files is specified as 15%.
- A filegroup named SalesGroup1 with the files SGrp1Fi1 and SGrp1Fi2.
- A filegroup named SalesGroup2 with the files SGrp2Fi1 and SGrp2Fi2.

```
USE master;
GO
IF DB_ID (N'Sales') IS NOT NULL
DROP DATABASE Sales;
GO
-- Get the SQL Server data path
DECLARE @data_path nvarchar(256);
SET @data_path = (SELECT SUBSTRING(physical_name, 1,
CHARINDEX(N'master.mdf', LOWER(physical_name)) - 1)
                  FROM master.sys.master_files
                  WHERE database_id = 1 AND file_id = 1);

-- execute the CREATE DATABASE statement
EXECUTE ('CREATE DATABASE  Sales
ON PRIMARY
( NAME = SPri1_dat,
    FILENAME = '''+ @data_path + 'SPri1dat.mdf'',
    SIZE = 10,
    MAXSIZE = 50,
    FILEGROWTH = 15% ),
( NAME = SPri2_dat,
    FILENAME = '''+ @data_path + 'SPri2dt.ndf'',
    SIZE = 10,
    MAXSIZE = 50,
    FILEGROWTH = 15% ),
FILEGROUP SalesGroup1
( NAME = SGrp1Fi1_dat,
    FILENAME = '''+ @data_path + 'SG1Fi1dt.ndf'',
    SIZE = 10,
    MAXSIZE = 50,
    FILEGROWTH = 5 ),
( NAME = SGrp1Fi2_dat,
    FILENAME = '''+ @data_path + 'SG1Fi2dt.ndf'',
    SIZE = 10,
    MAXSIZE = 50,
    FILEGROWTH = 5 ),
FILEGROUP SalesGroup2
( NAME = SGrp2Fi1_dat,
    FILENAME = '''+ @data_path + 'SG2Fi1dt.ndf'',
    SIZE = 10,
    MAXSIZE = 50,
    FILEGROWTH = 5 ),
( NAME = SGrp2Fi2_dat,
    FILENAME = '''+ @data_path + 'SG2Fi2dt.ndf'',
    SIZE = 10,
    MAXSIZE = 50,
    FILEGROWTH = 5 )
LOG ON
( NAME = Sales_log,
    FILENAME = '''+ @data_path + 'salelog.ldf'',
    SIZE = 5MB,
    MAXSIZE = 25MB,
    FILEGROWTH = 5MB )'
);
GO
```

Attaching a database: The following example detaches the database Archive created in example D, and then attaches it by using the FOR ATTACH clause. Archive was defined to have multiple data and log files. However, because the location of the files has not changed since they were created, only the primary file has to be specified in the FOR ATTACH clause. In SQL Server 2005, any full-text files that are part of the database that is being attached will be attached with the database.

```
USE master;
GO
sp_detach_db Archive;
GO
-- Get the SQL Server data path
DECLARE @data_path nvarchar(256);
```

```
SET @data_path = (SELECT SUBSTRING(physical_name, 1,
CHARINDEX(N'master.mdf', LOWER(physical_name)) - 1)
                  FROM master.sys.master_files
                  WHERE database_id = 1 AND file_id = 1);
-- Execute CREATE DATABASE FOR ATTACH statement
EXEC ('CREATE DATABASE Archive
      ON (FILENAME = '''+ @data_path + 'archdat1.mdf'')
      FOR ATTACH');
GO
```

Creating Database (Using using SQL Server Management Studio)

Steps to Create Database:

1. In Object Explorer, connect to an instance of the SQL Server 2005 Database Engine and then expand that instance.
2. Right-click Databases and then click **New Database**.
3. In **New Database**, enter a database name.
4. To create the database by accepting all default values, click **OK**, otherwise, continue with the following optional steps.
5. To change the default values of the primary data and transaction log files, in the **Database files** grid, click the appropriate cell and enter the new value
6. To add a new filegroup, click the **Filegroups** page. Click **Add** and then enter the values for the filegroup.
7. To create the database, click **OK**.

Summary

- SQL Server 2005 databases have three types of files
  - Primary data files
  - Secondary data files
  - Log files
- Filegroup is a logical collection of data files that enables administrators to manage all files within the filegroup as single item.

Session 17: Views and Partition Tables

Learning Objectives

After completing this session, you will be able to:

- Execute Database Console Commands
- Identify the types of views and their advantages
- Explain Partition Tables

Database Console Commands

T-SQL provides DBCC statements that act as Database Console Commands for SQL Server 2005.They are grouped into the following categories:

- Maintenance: Maintenance tasks on a database, index or filegroup.
- Miscellaneous: Miscellaneous tasks such as enabling trace flags or removing a DLL from memory.
- Informational: Tasks that gather and display various types of information.
- Validation: Validation operations on a database, table, index, catalog, filegroup or allocation.

Some useful DBCC commands are as follows:

- **DBCC SHRINKDATABASE:** Shrinks the size of the data files in the specified database.
- **DBCC SHRINKFILE:** Shrinks the size of the specified data file or log file for the related database.
- **DBCC INDEXDEFRAG:** Defragments indexes of the specified table or view.
- **DBCC DBREINDEX:** Rebuilds one or more indexes for a table in the specified database.
- **DBCC CLEANTABLE:** Reclaims space for dropped variable length columns and text columns.
- **DBCC CHECKDB:** Checks the allocation, structural, and logical integrity of all the objects in the specified database.
- **DBCC SHOW_STATISTICS:** Displays the current distribution statistics for the specified target on the specified table.

Views

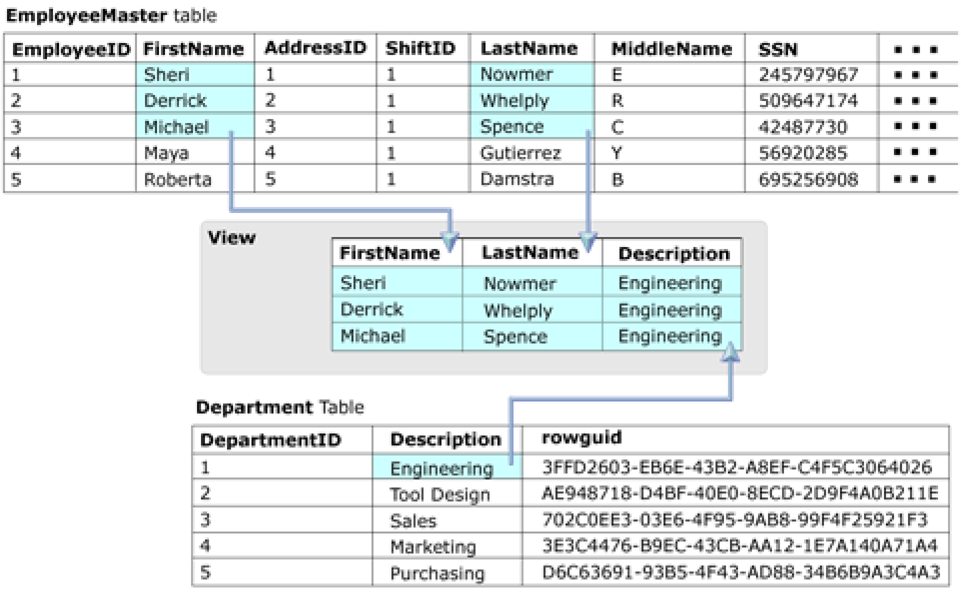
A view is a virtual table whose contents are defined by a query. Like a real table, a view consists of a set of named columns and rows of data. Unless indexed, a view does not exist as a stored set of data values in a database. The rows and columns of data come from tables referenced in the query defining the view and are produced dynamically when the view is referenced.

A view acts as a filter on the underlying tables referenced in the view. The query that defines the view can be from one or more tables or from other views in the current or other databases.

Distributed queries can also be used to define views that use data from multiple heterogeneous sources. This is useful, for example, if you want to combine similarly structured data from different servers, each of which stores data for a different region of your organization.

There are no restrictions on querying through views and few restrictions on modifying data through them.

This illustration shows a view based on two tables.



EmployeeMaster table

EmployeeID	FirstName	AddressID	ShiftID	LastName	MiddleName	SSN	▪ ▪ ▪
1	Sheri	1	1	Nowmer	E	245797967	▪ ▪ ▪
2	Derrick	2	1	Whelply	R	509647174	▪ ▪ ▪
3	Michael	3	1	Spence	C	42487730	▪ ▪ ▪
4	Maya	4	1	Gutierrez	Y	56920285	▪ ▪ ▪
5	Roberta	5	1	Damstra	B	695256908	▪ ▪ ▪

View

FirstName	LastName	Description
Sheri	Nowmer	Engineering
Derrick	Whelply	Engineering
Michael	Spence	Engineering

Department Table

DepartmentID	Description	rowguid
1	Engineering	3FFD2603-EB6E-43B2-A8EF-C4F5C3064026
2	Tool Design	AE948718-D4BF-40E0-8ECD-2D9F4A0B211E
3	Sales	702C0EE3-03E6-4F95-9AB8-99F4F25921F3
4	Marketing	3E3C4476-B9EC-43CB-AA12-1E7A140A71A4
5	Purchasing	D6C63691-93B5-4F43-AD88-34B6B9A3C4A3

Types of Views

In SQL Server 2005, you can create standard views, indexed views, and partitioned views:

- **Standard Views:** Combining data from one or more tables through a standard view lets you satisfy most of the benefits of using views. These include focusing on specific data and simplifying data manipulation.
- **Indexed Views:** An indexed view is a view that has been materialized. This means it has been computed and stored. You index a view by creating a unique clustered index on it. Indexed views dramatically improve the performance of some types of queries. Indexed views work best for queries that aggregate many rows. They are not well-suited for underlying data sets that are frequently updated.
- **Partitioned Views:** A partitioned view joins horizontally partitioned data from a set of member tables across one or more servers. This makes the data appear as if from one table. A view that joins member tables on the same instance of SQL Server is a local partitioned view. When a view joins data from tables across servers, it is a distributed partitioned view. Distributed partitioned views are used to implement a federation of database servers. A federation is a group of servers administered independently, but which cooperate to share the processing load of a system. Forming a federation of database servers by partitioning data is the mechanism that lets you scale out a set of servers to support the processing requirements of large, multitiered Web sites.

Advantages of Views

Views are commonly applied to:

- **To Focus on Specific Data:** Views let users focus on specific data that interests them and on the specific tasks for which they are responsible. Unnecessary or sensitive data can be left out of the view.
- **To simplify the Data Manipulation:** Simplify the users work with data. You can define frequently used joins, projections, UNION queries, and SELECT queries as views so that users do not have to specify all the conditions and qualifications every time an additional operation is performed on that data.
- **To Customize Data:** Views let different users to refer the data in different ways, even when they are using the same data at the same time.
- **To Export and Import Data:** Views can be used to export data to other applications.
- **To Combine Partitioned Data Across Servers:** The Transact-SQL UNION set operator can be used within a view to combine the results of two or more queries from separate tables into a single result set. This appears to the user as a single table that is called a partitioned view.

CREATE VIEW

Creates a virtual table that represents the data in one or more tables in an alternative way. CREATE VIEW must be the first statement in a query batch.

Syntax

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

<view_attribute> ::=
{
    [ ENCRYPTION ]
    [ SCHEMABINDING ]
    [ VIEW_METADATA ]     }
```

Arguments

schema_name: Is the name of the schema to which the view belongs.

view_name: Is the name of the view. View names must follow the rules for identifiers. Specifying the view owner name is optional.

column: Is the name to be used for a column in a view. A column name is required only when a column is derived from an arithmetic expression, a function, or a constant; when two or more columns may otherwise have the same name, typically because of a join; or when a column in a view is specified a name different from that of the column from which it is derived. Column names can also be assigned in the SELECT statement. If *column* is not specified, the view columns acquire the same names as the columns in the SELECT statement.

AS: Specifies the actions the view is to perform.

`select_statement:`

- Is the SELECT statement that defines the view. The statement can use more than one table and other views. Appropriate permissions are required to select from the objects referenced in the SELECT clause of the view that is created.
- A view does not have to be a simple subset of the rows and columns of one particular table. A view can be created that uses more than one table or other views with a SELECT clause of any complexity.
- In an indexed view definition, the SELECT statement must be a single table statement or a multitable JOIN with optional aggregation.
- The SELECT clauses in a view definition cannot include the following:
  - COMPUTE or COMPUTE BY clauses
  - An ORDER BY clause, unless there is also a TOP clause in the select list of the SELECT statement
  - The INTO keyword
  - The OPTION clause
  - A reference to a temporary table or a table variable.
- Because *select_statement* uses the SELECT statement, it is valid to use <join_hint> and <table_hint> hints as specified in the FROM clause
- Functions and multiple SELECT statements separated by UNION or UNION ALL can be used in *select_statement*.

`CHECK OPTION`: Forces all data modification statements executed against the view to follow the criteria set within *select_statement*. When a row is modified through a view, the WITH CHECK OPTION makes sure the data remains visible through the view after the modification is committed. CHECK OPTION cannot be specified if TOP is used anywhere in *select_statement*.

`ENCRYPTION:` Encrypts the entries in sys.syscomments that contain the text of the CREATE VIEW statement. Using WITH ENCRYPTION prevents the view from being published as part of SQL Server replication.

`SCHEMABINDING:`

- Binds the view to the schema of the underlying table or tables. When SCHEMABINDING is specified, the base table or tables cannot be modified in a way that would affect the view definition. The view definition itself must first be modified or dropped to remove dependencies on the table that is to be modified. When you use SCHEMABINDING, the *select_statement* must include the two-part names (*schema***.***object*) of tables, views, or user-defined functions that are referenced. All referenced objects must be in the same database.
- Views or tables that participate in a view created with the SCHEMABINDING clause cannot be dropped unless that view is dropped or changed so that it no longer has schema binding. Otherwise, the Microsoft SQL Server 2005 Database Engine raises an error. Also, executing ALTER TABLE statements on tables that participate in views that have schema binding fail when these statements affect the view definition.
- SCHEMABINDING cannot be specified if the view contains alias data type columns.

VIEW_METADATA:

- Specifies that the instance of SQL Server will return to the DB-Library, ODBC, and OLE DB APIs the metadata information about the view, instead of the base table or tables, when browse-mode metadata is being requested for a query that references the view. Browse-mode metadata is additional metadata that the instance of SQL Server returns to these client-side APIs. This metadata enables the client-side APIs to implement updatable client-side cursors. Browse-mode metadata includes information about the base table that the columns in the result set belong to.
- For views created with VIEW_METADATA, the browse-mode metadata returns the view name and not the base table names when it describes columns from the view in the result set.
- When a view is created by using WITH VIEW_METADATA, all its columns, except a **timestamp** column, are updatable if the view has INSTEAD OF INSERT or INSTEAD OF UPDATE triggers. For more information about updatable views, see Remarks.

Examples

Using a simple CREATE VIEW

The following example creates a view by using a simple SELECT statement. A simple view is helpful when a combination of columns is queried frequently. The data from this view comes from the HumanResources.Employee and Person.Contact tables of the AdventureWorks database. The data provides name and hire date information for the employees of Adventure Works Cycles. The view could be created for the person in charge of tracking work anniversaries but without giving this person access to all the data in these tables.

```
USE AdventureWorks ;
GO
IF OBJECT_ID ('hiredate_view', 'view') IS NOT NULL
DROP VIEW hiredate_view ;
GO
CREATE VIEW hiredate_view
AS
SELECT c.FirstName, c.LastName, e.EmployeeID, e.HireDate
FROM HumanResources.Employee e JOIN Person.Contact c on e.ContactID =
c.ContactID ;
GO
```

Using WITH ENCRYPTION

The following example uses the WITH ENCRYPTION option and shows computed columns, renamed columns, and multiple columns.

```
USE AdventureWorks ;
GO
IF OBJECT_ID ('PurchaseOrderReject', 'V') IS NOT NULL
DROP VIEW PurchaseOrderReject ;
GO
CREATE VIEW PurchaseOrderReject
WITH ENCRYPTION
```

```
AS
SELECT PurchaseOrderID, ReceivedQty, RejectedQty, RejectedQty /
ReceivedQty AS RejectRatio
FROM Purchasing.PurchaseOrderDetail
WHERE RejectedQty / ReceivedQty > 0
AND DueDate > '06/30/2001' ;
GO
```

Using WITH CHECK OPTION

The following example shows a view named SeattleOnly that references five tables and allows for data modifications to apply only to employees who live in Seattle.

```
USE AdventureWorks ;
GO
IF OBJECT_ID ('SeattleOnly', 'V') IS NOT NULL
DROP VIEW SeattleOnly ;
GO
CREATE VIEW SeattleOnly
AS
SELECT c.LastName, c.FirstName, a.City, s.StateProvinceCode
FROM Person.Contact c JOIN HumanResources.Employee e ON c.ContactID =
e.ContactID
JOIN HumanResources.EmployeeAddress ea ON e.EmployeeID = ea.EmployeeID
JOIN Person.Address a ON ea.AddressID = a.AddressID
JOIN Person.StateProvince s ON a.StateProvinceID = s.StateProvinceID
WHERE a.City = 'Seattle'
WITH CHECK OPTION ;
GO
```

Using built-in functions within a view

The following example shows a view definition that includes a built-in function. When you use functions, you must specify a column name for the derived column.

```
USE AdventureWorks ;
GO
IF OBJECT_ID ('SalesPersonPerform', 'view') IS NOT NULL
DROP VIEW SalesPersonPerform ;
GO
CREATE VIEW SalesPersonPerform
AS
SELECT TOP 100 SalesPersonID, SUM(TotalDue) AS TotalSales
FROM Sales.SalesOrderHeader
WHERE OrderDate > '12/31/2000'
GROUP BY SalesPersonID
ORDER BY TotalSales DESC ;
GO
```

Using partitioned data

The following example uses tables named SUPPLY1, SUPPLY2, SUPPLY3, and SUPPLY4. These tables correspond to the supplier tables from four offices, located in different countries/regions.

```
--Create the tables and insert the values.
CREATE TABLE SUPPLY1 (
supplyID INT PRIMARY KEY CHECK (supplyID BETWEEN 1 and 150),
supplier CHAR(50)
);
CREATE TABLE SUPPLY2 (
supplyID INT PRIMARY KEY CHECK (supplyID BETWEEN 151 and 300),
supplier CHAR(50)
);
CREATE TABLE SUPPLY3 (
supplyID INT PRIMARY KEY CHECK (supplyID BETWEEN 301 and 450),
supplier CHAR(50)
);
CREATE TABLE SUPPLY4 (
supplyID INT PRIMARY KEY CHECK (supplyID BETWEEN 451 and 600),
supplier CHAR(50)
);
GO
INSERT SUPPLY1 VALUES ('1', 'CaliforniaCorp');
INSERT SUPPLY1 VALUES ('5', 'BraziliaLtd');
INSERT SUPPLY2 VALUES ('231', 'FarEast');
INSERT SUPPLY2 VALUES ('280', 'NZ');
INSERT SUPPLY3 VALUES ('321', 'EuroGroup');
INSERT SUPPLY3 VALUES ('442', 'UKArchip');
INSERT SUPPLY4 VALUES ('475', 'India');
INSERT SUPPLY4 VALUES ('521', 'Afrique');
GO
--Create the view that combines all supplier tables.
CREATE VIEW all_supplier_view
AS
SELECT *
FROM SUPPLY1
UNION ALL
SELECT *
FROM SUPPLY2
UNION ALL
SELECT *
FROM SUPPLY3
UNION ALL
SELECT *
FROM SUPPLY4;
```

Derived Tables

Derived tables are the result set of a SELECT query in the FROM clause.A completely dynamic object.Present only for the duration of query execution

Example:

```
USE AdventureWorks ;
GO
SELECT RTRIM(c.FirstName) + ' ' + LTRIM(c.LastName) AS Name,
d.City
FROM Person.Contact c
INNER JOIN HumanResources.Employee e ON c.ContactID = e.ContactID
INNER JOIN (SELECT AddressID, City FROM Person.Address) AS d --Derived
table
ON e.AddressID = d.AddressID
ORDER BY c.LastName, c.FirstName ;
```

Partitioned Tables

- Partitioned tables are tables whose data is horizontally divided into units, which may be spread across more than one filegroup in a database.
- Partitioning makes large tables or indexes more manageable by letting you to access or manage subsets of data quickly and efficiently.
- Performance benefits partitioned table includes the following:
  - Faster index searches
  - Enhanced JOIN performance
  - Reduced locking
- Manageability benefits of partitioned tables includes the following:
  - The ability to implement separate backup strategies: Different sets of data might have different backup requirements. For example: Recent orders data might be updated frequently and require regular backups, while older orders might change rarely and require only infrequent backups
  - Control over storage media: Partitioning a table lets you to choose appropriate storage for date based on its access requirements.
  - Index management benefits: In addition to partitioning a table you can partition its indexes. This allows you to reorganize, optimize and rebuild indexes by partition, which is faster and less intrusive than managing an entire index.
- Steps to create Partitioned Tables:
  - First create Partition Function: (CREATE PARTITION FUNCTION)

    Partition Functions define partition boundaries

    Boundary values can be assigned either to LEFT to Right Function
  - Second create Partition Scheme: **(**CREATE PARTITION SCHEME **)**

    A Partition scheme assign partitions to filegroups

    The “next” filegroup can also be defined
  - Third create Partition table: (CREATE TABLE)

Example:

```
CREATE PARTITION FUNCTION myRangePF1 (int)
AS RANGE LEFT FOR VALUES (1, 100, 1000) ;
GO
CREATE PARTITION SCHEME myRangePS1
AS PARTITION myRangePF1 TO (test1fg, test2fg, test3fg, test4fg) ;
GO
CREATE TABLE PartitionTable (col1 int, col2 char(10))
ON myRangePS1 (col1) ;
GO
```

Summary

- Views are applied to give a different perspective of data
- Views are not stored physically and only consists of a stored query
- You can modify base tables through a view
- Indexed views are stored physically
- Partitioned views are applied to increase scalability and performance

Session 19: Control Flow Constructs

Learning Objectives

After completing this session, you will be able to:

- ❑ Execute Control flow statements

Overview

Transact-SQL provides special words called control-of-flow language that control the flow of execution of Transact-SQL statements, statement blocks, and stored procedures. These words can be used in ad hoc Transact-SQL statements, in batches, and in stored procedures.

Without control-of-flow language, separate Transact-SQL statements are performed sequentially, as they occur. Control-of-flow language permits statements to be connected, related to each other, and made interdependent using programming-like constructs.

These control-of-flow words are useful when you need to direct Transact-SQL to take some kind of action. For example, use a BEGIN...END pair of statements when including more than one Transact-SQL statement in a logical block. Use an IF...ELSE pair of statements when a certain statement or block of statements needs to be executed IF some condition is met, and another statement or block of statements should be executed if that condition is not met (the ELSE condition).

The control-of-flow statements cannot span multiple batches or stored procedures.

The Transact-SQL control-of-flow language keywords are:

BEGIN...END	RETURN
BREAK	TRY...CATCH
CONTINUE	WAITFOR
GOTO label	WHILE
IF...ELSE	

END (BEGIN...END)

Encloses a series of Transact-SQL statements that will execute as a group. BEGIN...END blocks can be nested.

Syntax

```
BEGIN
     { sql_statement | statement_block }
END
```

Arguments

{sql_statement | statement_block}: Is any valid Transact-SQL statement or statement grouping as defined with a statement block. To define a statement block (batch), use the control-of-flow language keywords BEGIN and END. Although all Transact-SQL statements are valid within a BEGIN...END block, certain Transact-SQL statements should not be grouped together within the same batch (statement block).

IF...ELSE

Imposes conditions on the execution of a Transact-SQL statement. The Transact-SQL statement that follows an IF keyword and its condition is executed if the condition is satisfied: the Boolean expression returns TRUE. The optional ELSE keyword introduces another Transact-SQL statement that is executed when the IF condition is not satisfied: the Boolean expression returns FALSE.

Syntax

```
IF Boolean_expression
     { sql_statement | statement_block }
[ ELSE
     { sql_statement | statement_block } ]
```

Arguments

Boolean_expression: Is an expression that returns TRUE or FALSE. If the Boolean expression contains a SELECT statement, the SELECT statement must be enclosed in parentheses.

{ sql_statement | statement_block }:

- Is any Transact-SQL statement or statement grouping as defined by using a statement block. Unless a statement block is used, the IF or ELSE condition can affect the performance of only one Transact-SQL statement.
- To define a statement block, use the control-of-flow keywords BEGIN and END. CREATE TABLE or SELECT INTO statements must refer to the same table name when the CREATE TABLE or SELECT INTO statements are used in both the IF and ELSE areas of the IF...ELSE block.

Examples:

The following example uses IF…ELSE with output from the usp_GetList stored procedure. This stored procedure is defined in Creating Stored Procedures. In this example, the stored procedure returns a list of bikes with a list price less than $700. This causes the first PRINT statement to execute.

```
DECLARE @compareprice money, @cost money
EXECUTE usp_GetList '%Bikes%', 700,
    @compareprice OUT,
    @cost OUTPUT
IF @cost <= @compareprice
BEGIN
    PRINT 'These products can be purchased for less than
    $'+RTRIM(CAST(@compareprice AS varchar(20)))+'.'
END
```

```
ELSE
    PRINT 'The prices for all products in this category exceed
    $'+ RTRIM(CAST(@compareprice AS varchar(20)))+'.'
```

CASE

Both formats support an optional ELSE argument.

Syntax

```
Simple CASE function:
CASE input_expression
     WHEN when_expression THEN result_expression
    [ ...n ]
     [
    ELSE else_result_expression
     ]
END
Searched CASE function:
CASE
     WHEN Boolean_expression THEN result_expression
    [ ...n ]
     [
    ELSE else_result_expression
     ]
END
```

Arguments

input_expression: Is the expression evaluated when the simple CASE format is used. *input_expression* is any valid expression.

WHEN when_expression: Is a simple expression to which *input_expression* is compared when the simple CASE format is used. *when_expression* is any valid expression. The data types of *input_expression* and each *when_expression* must be the same or must be an implicit conversion.

n: Is a placeholder that indicates that multiple WHEN *when_expression* THEN *result_expression* clauses, or multiple WHEN *Boolean_expression* THEN *result_expression* clauses can be used.

THEN result_expression: Is the expression returned when *input_expression* equals *when_expression* evaluates to TRUE, or *Boolean_expression* evaluates to TRUE. *result expression* is any valid expression.

`ELSE 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, CASE returns NULL. *else_result_expression* is any valid expression. The data types of *else_result_expression* and any *result_expression* must be the same or must be an implicit conversion.

`WHEN Boolean_expression:` Is the Boolean expression evaluated when using the searched CASE format. *Boolean_expression* is any valid Boolean expression.

Result Types: Returns the highest precedence type from the set of types in *result_expressions* and the optional *else_result_expression*. For more information, see Data Type Precedence (Transact-SQL).

Examples:

Using a SELECT statement with a simple CASE function

Within a SELECT statement, a simple CASE function allows for only an equality check; no other comparisons are made. The following example uses the CASE function to change the display of product line categories to make them more understandable.

```
USE AdventureWorks;
GO
SELECT   ProductNumber, Category =
      CASE ProductLine
         WHEN 'R' THEN 'Road'
         WHEN 'M' THEN 'Mountain'
         WHEN 'T' THEN 'Touring'
         WHEN 'S' THEN 'Other sale items'
         ELSE 'Not for sale'
      END,
   Name
FROM Production.Product
ORDER BY ProductNumber;
GO
```

Using a SELECT statement with simple and searched CASE function

Within a SELECT statement, the searched CASE function allows for values to be replaced in the result set based on comparison values. The following example displays the list price as a text comment based on the price range for a product.

```
USE AdventureWorks;
GO
SELECT   ProductNumber, Name, 'Price Range' =
      CASE
         WHEN ListPrice =  0 THEN 'Mfg item - not for resale'
         WHEN ListPrice < 50 THEN 'Under $50'
         WHEN ListPrice >= 50 and ListPrice < 250 THEN 'Under $250'
         WHEN ListPrice >= 250 and ListPrice < 1000 THEN 'Under $1000'
         ELSE 'Over $1000'
      END
FROM Production.Product
ORDER BY ProductNumber ;
GO
```

Result Values:

Simple CASE function:

- Evaluates *input_expression*, and then in the order specified, evaluates *input_expression* = *when_expression* for each WHEN clause.
- Returns the *result_expression* of the first *input_expression* = *when_expression* that evaluates to TRUE.
- If no *input_expression* = *when_expression* evaluates to TRUE, the SQL Server 2005 Database Engine returns the *else_result_expression* if an ELSE clause is specified, or a NULL value if no ELSE clause is specified.

Searched CASE function:

- Evaluates, in the order specified, *Boolean_expression* for each WHEN clause.
- Returns *result_expression* of the first *Boolean_expression* that evaluates to TRUE.
- If no *Boolean_expression* evaluates to TRUE, the Database Engine returns the *else_result_expression* if an ELSE clause is specified, or a NULL value if no ELSE clause is specified.

GOTO

Alters the flow of execution to a label. The Transact-SQL statement or statements that follow GOTO are skipped and processing continues at the label. GOTO statements and labels can be used anywhere within a procedure, batch, or statement block. GOTO statements can be nested.

Syntax

```
Define the label:
  label : Alter the execution:
     GOTO label
```

Arguments

label: Is the point after which processing starts if a GOTO is targeted to that label. Labels must follow the rules for identifiers. A label can be used as a commenting method whether GOTO is used.

Examples:

The following example shows GOTO looping as an alternative to using WHILE.

```
USE AdventureWorks;
GO
DECLARE @tablename sysname
```

```
SET @tablename = N'Person.AddressType'
table_loop:
   IF (@@FETCH_STATUS <> -2)
   BEGIN
      SELECT @tablename = RTRIM(UPPER(@tablename))
      EXEC ('SELECT ''' + @tablename + ''' = COUNT(*) FROM '
            + @tablename )
      PRINT '  '
   END
   FETCH NEXT FROM tnames_cursor INTO @tablename
IF (@@FETCH_STATUS <> -1) GOTO table_loop
GO
```

BREAK

Exits the innermost loop in a WHILE or IF…ELSE statement. Any statements appearing after the END keyword, marking the end of the loop, are executed. BREAK is frequently, but not always, started by an IF test.

CONTINUE

Restarts a WHILE loop. Any statements after the CONTINUE keyword are ignored. CONTINUE is frequently, but not always, opened by an IF test.

WHILE

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

Syntax

```
WHILE Boolean_expression
     { sql_statement | statement_block }
     [ BREAK ]
     { sql_statement | statement_block }
     [ CONTINUE ]
     { sql_statement | statement_block }
```

Arguments:

Boolean_expression: Is an expression that returns **TRUE** or **FALSE**. If the Boolean expression contains a SELECT statement, the SELECT statement must be enclosed in parentheses.

{sql_statement | statement_block}: Is any Transact-SQL statement or statement grouping as defined with a statement block. To define a statement block, use the control-of-flow keywords BEGIN and END.

BREAK: Causes an exit from the innermost WHILE loop. Any statements that appear after the END keyword, marking the end of the loop, are executed.

CONTINUE: Causes the WHILE loop to restart, ignoring any statements after the CONTINUE keyword.

Remarks: If two or more WHILE loops are nested, the inner BREAK exits to the next outermost loop.All the statements after the end of the inner loop run first, and then the next outermost loop restarts.

Examples:

Using BREAK and CONTINUE with nested IF...ELSE and WHILE

In the following example, if the average list price of a product is less than $300, the WHILE loop doubles the prices and then selects the maximum price. If the maximum price is less than or equal to $500, the WHILE loop restarts and doubles the prices again. This loop continues doubling the prices until the maximum price is greater than $500, and then exits the WHILE loop and prints a message.

```
USE AdventureWorks;
GO
WHILE (SELECT AVG(ListPrice) FROM Production.Product) < $300
BEGIN
   UPDATE Production.Product
      SET ListPrice = ListPrice * 2
   SELECT MAX(ListPrice) FROM Production.Product
   IF (SELECT MAX(ListPrice) FROM Production.Product) > $500
      BREAK
   ELSE
      CONTINUE
END
PRINT 'Too much for the market to bear';
```

WAITFOR

Blocks the execution of a batch, stored procedure, or transaction until a specified time or time interval is reached, or a specified statement modifies or returns at least one row.

Syntax

```
WAITFOR
{
    DELAY 'time_to_pass'
  | TIME 'time_to_execute'
  | ( receive_statement ) [ , TIMEOUT timeout ]
}
```

Arguments:

DELAY: Is the specified period of time that must pass, up to a maximum of 24 hours, before execution of a batch, stored procedure, or transaction proceeds.

'time_to_pass': Is the period of time to wait. *time_to_pass* can be specified in one of the acceptable formats for **datetime** data, or it can be specified as a local variable. Dates cannot be specified; therefore, the date part of the **datetime** value is not allowed.

TIME: Is the specified time when the batch, stored procedure, or transaction runs.

'time_to_execute': Is the time at which the WAITFOR statement finishes. *time_to_execute* can be specified in one of the acceptable formats for **datetime** data, or it can be specified as a local variable. Dates cannot be specified; therefore, the date part of the **datetime** value is not allowed.

receive_statement: Is a valid RECEIVE statement.

TIMEOUT timeout: Specifies the period of time, in milliseconds, to wait for a message to arrive on the queue.

Examples:

Using WAITFOR TIME:

The following example executes the stored procedure sp_update_job at 10:20 P.M. (22:20).

```
USE msdb;
EXECUTE sp_add_job @job_name = 'TestJob';
BEGIN
    WAITFOR TIME '22:20';
    EXECUTE sp_update_job @job_name = 'TestJob',
        @new_name = 'UpdatedJob';
END;
GO
```

Using WAITFOR DELAY:

The following example executes the stored procedure after a 2-hour delay.

```
BEGIN
    WAITFOR DELAY '02:00';
    EXECUTE sp_helpdb;
END;
GO
```

TRY...CATCH

Implements error handling for Transact-SQL that is similar to the exception handling in the C# and C++ languages. A group of Transact-SQL statements can be enclosed in a TRY block. If an error occurs within the TRY block, control is passed to another group of statements enclosed in a CATCH block.

Syntax:

```
BEGIN TRY
     { sql_statement | statement_block }
END TRY
BEGIN CATCH
     { sql_statement | statement_block }
END CATCH
[ ; ]
```

Arguments:

- **sql_statement:** Is any Transact-SQL statement.
- **statement_block:** Any group of Transact-SQL statements in a batch or enclosed in a BEGIN…END block.

Remarks:

A TRY…CATCH construct catches all execution errors with severity greater than 10 that do not terminate the database connection.

A TRY block must be followed immediately by an associated CATCH block. Placing any other statements between the END TRY and BEGIN CATCH statements generates a syntax error.

A TRY…CATCH construct cannot span multiple batches. A TRY…CATCH construct cannot span multiple blocks of Transact-SQL statements. For example, a TRY…CATCH construct cannot span two BEGIN…END blocks of Transact-SQL statements and cannot span an IF…ELSE construct.

If there are no errors in the code enclosed in a TRY block, when the last statement in the TRY block completes, control passes to the statement immediately after the associated END CATCH statement. If there is an error in the code enclosed in a TRY block, control passes to the first statement in the associated CATCH block. If the END CATCH statement is the last statement in a stored procedure or trigger, control is passed back to the statement that invoked the stored procedure or trigger.

When the code in the CATCH block completes, control passes to the statement immediately after the END CATCH statement. Errors trapped by a CATCH block are not returned to the calling application. If any of the error information must be returned to the application, the code in the CATCH block must do so using mechanisms, such as SELECT result sets or the RAISERROR and PRINT statements. For more information about using RAISERROR in conjunction with TRY…CATCH, see Using TRY...CATCH in Transact-SQL.

TRY…CATCH constructs can be nested. Either a TRY block or a CATCH block can contain nested TRY…CATCH constructs. For example, a CATCH block can contain an embedded TRY…CATCH construct to handle errors encountered by the CATCH code.

Errors encountered in a CATCH block are treated like errors generated anywhere else. If the CATCH block contains a nested TRY…CATCH construct, any error in the nested TRY block will pass control to the nested CATCH block. If there is no nested TRY…CATCH construct, the error is passed back to the caller.

TRY…CATCH constructs catch unhandled errors from stored procedures or triggers executed by the code in the TRY block. Alternatively, the stored procedures or triggers can contain their own TRY…CATCH constructs to handle errors generated by their code. For example, when a TRY block executes a stored procedure and an error occurs in the stored procedure, the error can be handled in the following ways:

- If the stored procedure does not contain its own TRY…CATCH construct, the error returns control to the CATCH block associated with the TRY block containing the EXECUTE statement.
- If the stored procedure contains a TRY…CATCH construct, the error transfers control to the CATCH block in the stored procedure. When the CATCH block code completes, control is passed back to the statement immediately after the EXECUTE statement that called the stored procedure.

GOTO statements cannot be used to enter a TRY or CATCH block. GOTO statements can be used to jump to a label within the same TRY or CATCH block or to leave a TRY or CATCH block.

The TRY…CATCH construct cannot be used within a user-defined function.

Retrieving Error Information

Within the scope of a CATCH block, the following system functions can be used to obtain information about the error that caused the CATCH block to be executed:

- ERROR_NUMBER() returns the number of the error.
- ERROR_SEVERITY() returns the severity.
- ERROR_STATE() returns the error state number.
- ERROR_PROCEDURE() returns the name of the stored procedure or trigger where the error occurred.
- ERROR_LINE() returns the line number inside the routine that caused the error.
- ERROR_MESSAGE() returns the complete text of the error message. The text includes the values supplied for any substitutable parameters, such as lengths, object names, or times.

These functions return NULL if called outside the scope of the CATCH block. Error information may be retrieved using these functions from anywhere within the scope of the CATCH block. For example, the script below demonstrates a stored procedure that contains error-handling functions. In the CATCH block of a TRY…CATCH construct, the stored procedure is called and information about the error is returned.

```
USE AdventureWorks;
GO
-- Verify that the stored procedure does not already exist.
IF OBJECT_ID ( 'usp_GetErrorInfo', 'P' ) IS NOT NULL
    DROP PROCEDURE usp_GetErrorInfo;
GO
```

```
-- Create procedure to retrieve error information.
CREATE PROCEDURE usp_GetErrorInfo
AS
    SELECT
        ERROR_NUMBER() AS ErrorNumber,
        ERROR_SEVERITY() AS ErrorSeverity,
        ERROR_STATE() AS ErrorState,
        ERROR_PROCEDURE() AS ErrorProcedure,
        ERROR_LINE() AS ErrorLine,
        ERROR_MESSAGE() AS ErrorMessage;
GO

BEGIN TRY
    -- Generate divide-by-zero error.
    SELECT 1/0;
END TRY
BEGIN CATCH
    -- Execute error retrieval routine.
    EXECUTE usp_GetErrorInfo;
END CATCH;
```

DECLARE @local_variable

Variables are declared in the body of a batch or procedure with the DECLARE statement and are assigned values with either a SET or SELECT statement. Cursor variables can be declared with this statement and used with other cursor-related statements. After declaration, all variables are initialized as NULL.

Syntax:

```
DECLARE
     {{ @local_variable [AS] data_type }
    | { @cursor_variable_name CURSOR }
    | { @table_variable_name < table_type_definition > }
     } [ ,...n]

< table_type_definition > ::=
     TABLE ( { < column_definition > | < table_constraint > } [ ,... ]
      )

< column_definition > ::=
          column_name { scalar_data_type | AS computed_column_expression
}
     [ COLLATE collation_name ]
     [ [ DEFAULT constant_expression ] | IDENTITY [ ( seed,increment ) ]
]
```

```
     [ ROWGUIDCOL ]
     [ < column_constraint > ]

< column_constraint > ::=
     { [ NULL | NOT NULL ]
     | [ PRIMARY KEY | UNIQUE ]
     | CHECK ( logical_expression )
     }

< table_constraint > ::=
     { { PRIMARY KEY | UNIQUE } ( column_name [ ,... ] )
     | CHECK ( search_condition )
     }
```

Arguments:

@local_variable: Is the name of a variable. Variable names must begin with an at (@) sign. Local variable names must conform to the rules for identifiers.

data_type: Is any system-supplied, CLR user-defined type, or alias data type. A variable cannot be of **text**, **ntext**, or **image** data type.

@cursor_variable_name: Is the name of a cursor variable. Cursor variable names must begin with an at (@) sign and conform to the rules for identifiers.

CURSOR: Specifies that the variable is a local cursor variable.

@table_variable_name: Is the name of a variable of type **table**. Variable names must begin with an at (@) sign and conform to the rules for identifiers.

table_type_definition:

- ❑ Defines the **table** data type. The table declaration includes column definitions, names, data types, and constraints. The only constraint types allowed are PRIMARY KEY, UNIQUE, NULL, and CHECK. An alias data type cannot be used as a column scalar data type if a rule or default definition is bound to the type.
- ❑ *table_type_definition* is a subset of information used to define a table in CREATE TABLE. Elements and essential definitions are included here.

n: Is a placeholder indicating that multiple variables can be specified and assigned values. When declaring **table** variables, the **table** variable must be the only variable being declared in the DECLARE statement.

column_name: Is the name of the column in the table.

scalar_data_type : Specifies that the column is a scalar data type.

computed_column_expression: Is an expression defining the value of a computed column. It is computed from an expression using other columns in the same table. For example, a computed column can have the definition **cost** AS **price * qty**. The expression can be a noncomputed column name, constant, built-in function, variable, or any combination of these connected by one or more operators. The expression cannot be a subquery or a user-defined function.

`[ COLLATE collation_name ]:` Specifies the collation for the column. *collation_name* can be either a Windows collation name or an SQL collation name, and is applicable only for columns of the **char**, **varchar**, **text**, **nchar**, **nvarchar**, and **ntext** data types. If not specified, the column is assigned either the collation of the user-defined data type (if the column is of a user-defined data type) or the default collation of **tempdb**. To create a **table** variable with columns that inherit the collation of the current database, add COLLATE database_default to the definitions of **char**, **varchar**, **text**, **nchar**, **nvarchar**, and **ntext** columns.

`DEFAULT:` Specifies the value provided for the column when a value is not explicitly supplied during an insert. DEFAULT definitions can be applied to any columns except those defined as **timestamp** or those with the IDENTITY property. DEFAULT definitions are removed when the table is dropped. Only a constant value, such as a character string; a system function, such as a SYSTEM_USER(); or NULL can be used as a default. To maintain compatibility with earlier versions of SQL Server, a constraint name can be assigned to a DEFAULT.

`constant_expression`: Is a constant, NULL, or a system function used as the default value for the column.

`IDENTITY:` Indicates that the new column is an identity column. When a new row is added to the table, SQL Server provides a unique incremental value for the column. Identity columns are commonly used in conjunction with PRIMARY KEY constraints to serve as the unique row identifier for the table. The IDENTITY property can be assigned to **tinyint**, **smallint**, **int**, **decimal(p,0)**, or **numeric(p,0)** columns. Only one identity column can be created per table. Bound defaults and DEFAULT constraints cannot be used with an identity column. You must specify both the seed and increment, or neither. If neither is specified, the default is (1,1).

`seed:` Is the value used for the very first row loaded into the table.

`increment:` Is the incremental value added to the identity value of the previous row that was loaded.

`ROWGUIDCOL:` Indicates that the new column is a row global unique identifier column. Only one **uniqueidentifier** column per table can be designated as the ROWGUIDCOL column. The ROWGUIDCOL property can be assigned only to a **uniqueidentifier** column.

`NULL | NOT NULL:` Are keywords that determine whether or not null values are allowed in the column.

`PRIMARY KEY:` Is a constraint that enforces entity integrity for a given column or columns through a unique index. Only one PRIMARY KEY constraint can be created per table.
`UNIQUE:` Is a constraint that provides entity integrity for a given column or columns through a unique index. A table can have multiple UNIQUE constraints.

`CHECK:` Is a constraint that enforces domain integrity by limiting the possible values that can be entered into a column or columns.

`logical_expression`: Is a logical expression that returns TRUE or FALSE.

Remarks:

Variables are often used in a batch or procedure as counters for WHILE, LOOP, or for an IF...ELSE block.

Variables can be used only in expressions, not in place of object names or keywords. To construct dynamic SQL statements, use EXECUTE.

The scope of a local variable is the batch in which it is declared.

A cursor variable that currently has a cursor assigned to it can be referenced as a source in a:

- CLOSE statement.
- DEALLOCATE statement.
- FETCH statement.
- OPEN statement.
- Positioned DELETE or UPDATE statement.
- SET CURSOR variable statement (on the right side).

In all these statements, Microsoft SQL Server raises an error if a referenced cursor variable exists but does not have a cursor currently allocated to it. If a referenced cursor variable does not exist, SQL Server raises the same error raised for an undeclared variable of another type.

Cursor variable values do not change after a cursor is declared. In SQL Server version 6.5 and earlier, variable values are refreshed every time a cursor is reopened.

A cursor variable:

- Can be the target of either a cursor type or another cursor variableCan be referenced as the target of an output cursor parameter in an EXECUTE statement if the cursor variable does not have a cursor currently assigned to it.
- Should be regarded as a pointer to the cursor.

Examples

Using DECLARE

The following example uses a local variable named **@find** to retrieve contact information for all last names beginning with 'Man'.

```
USE AdventureWorks;
GO
DECLARE @find varchar(30);
SET @find = 'Man%';
SELECT LastName, FirstName, Phone
FROM Person.Contact
WHERE LastName LIKE @find;
```

Use DECLARE with two variables

The following example retrieves the names of Adventure Works Cycles sales representatives who are located in the North American sales territory and have at least $2,000,000 in sales for the year.

```
USE AdventureWorks;
GO
SET NOCOUNT ON;
GO
DECLARE @Group nvarchar(50), @Sales money;
SET @Group = N'North America';
SET @Sales = 2000000;
SET NOCOUNT OFF;
SELECT FirstName, LastName, SalesYTD
FROM Sales.vSalesPerson
WHERE TerritoryGroup = @Group and SalesYTD >= @Sales;
```

DECLARE a variable of type table

The following example creates a **table** variable that stores the values specified in the OUTPUT clause of the UPDATE statement. Two SELECT statements follow that return the values in @MyTableVar and the results of the update operation in the Employee table. Note the results in the INSERTED.ModifiedDate column are different from the values in the ModifiedDate column in the Employee table. This is because the AFTER UPDATE trigger, which updates the value of ModifiedDate to the current date, is defined on the Employee table. However, the columns returned from OUTPUT reflect the data before triggers are fired

```
USE AdventureWorks;
GO
DECLARE @MyTableVar table(
    EmpID int NOT NULL,
    OldVacationHours int,
    NewVacationHours int,
    ModifiedDate datetime);
UPDATE TOP (10) HumanResources.Employee
SET VacationHours = VacationHours * 1.25
OUTPUT INSERTED.EmployeeID,
       DELETED.VacationHours,
       INSERTED.VacationHours,
       INSERTED.ModifiedDate
INTO @MyTableVar;
--Display the result set of the table variable.
SELECT EmpID, OldVacationHours, NewVacationHours, ModifiedDate
FROM @MyTableVar;
GO
--Display the result set of the table.
--Note that ModifiedDate reflects the value generated by an
--AFTER UPDATE trigger.
```

```
SELECT TOP (10) EmployeeID, VacationHours, ModifiedDate
FROM HumanResources.Employee;
GO
```

Summary

- Transact-SQL provides special words called control-of-flow language that control the flow of execution of Transact-SQL statements,
- CASE has two formats Both formats support an optional ELSE argument:
  - The simple CASE function compares an expression to a set of simple expressions to determine the result.
  - The searched CASE function evaluates a set of Boolean expressions to determine the result.
- WHILE construct sets condition for the repeated execution of an SQL statement or statement block.
- WAITFOR blocks the execution of a batch, stored procedure or transaction until a specified time or time interval is reached.

Session 21: Cursors

Learning Objectives

After completing this session, you will be able to:

- Identify cursors and cursor types
- Apply cursors

Cursors

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

Cursors extend result processing by:

- Allowing positioning at specific rows of the result set.
- Retrieving one row or block of rows from the current position in the result set.
- Supporting data modifications to the rows at the current position in the result set.
- Supporting different levels of visibility to changes made by other users to the database data that is presented in the result set.
- Providing Transact-SQL statements in scripts, stored procedures, and triggers access to the data in a result set.

Requesting a Cursor

Microsoft SQL Server 2005 supports two methods for requesting a cursor:

- Transact-SQL : The Transact-SQL language supports a syntax for using cursors modeled after the SQL-92 cursor syntax.

Database application programming interface (API) cursor functions SQL Server supports the cursor functionality of these database APIs:

- ADO (Microsoft ActiveX Data Object)
- OLE DB
- ODBC (Open Database Connectivity)

An application should never mix these two methods of requesting a cursor. An application that has used the API to specify cursor behaviors should not then execute a Transact-SQL DECLARE CURSOR statement to also request a Transact-SQL cursor. An application should only execute DECLARE CURSOR if it has set all the API cursor attributes back to their defaults.

If neither a Transact-SQL nor API cursor has been requested, SQL Server defaults to returning a complete result set, known as a default result set, to the application.

Server Cursor Advantage

Server Cursors versus Client Cursors: There are several advantages to using server cursors instead of client cursors:

- **Performance:** If you are going to access a fraction of the data in the cursor (typical of many browsing applications), using server cursors provides optimal performance because only fetched data is sent over the network. Client cursors cache the entire result set on the client.
- **Additional cursor types:** If the SQL Native Client ODBC driver used only client cursors, it could support only forward-only and static cursors. By using API server cursors the driver can also support keyset-driven and dynamic cursors. SQL Server also supports the full range of cursor concurrency attributes only through server cursors. Client cursors are limited in the functionality they support.
- **More accurate positioned updates:** Server cursors directly support positioned operations, such as the ODBC **SQLSetPos** function or UPDATE and DELETE statements with the WHERE CURRENT OF clause. Client cursors, on the other hand, simulate positioned cursor updates by generating a Transact-SQL searched UPDATE statement, which leads to unintended updates if more than one row matches the WHERE clause conditions of the UPDATE statement.
- **Memory usage:** When using server cursors, the client does not need to cache large amounts of data or maintain information about the cursor position because the server does that.
- **Multiple active statements:** When using server cursors, no results are left outstanding on the connection between cursor operations. This allows you to have multiple cursor-based statements active at the same time.

The operation of all server cursors, except static or insensitive cursors, depends on the schema of the underlying tables. Any schema changes to those tables after a cursor has been declared results in an error on any subsequent operation on that cursor.

Cursor Process

Transact-SQL cursors and API cursors have different syntax, but the following general process is used with all SQL Server cursors:

1. Associate a cursor with the result set of a Transact-SQL statement, and define characteristics of the cursor, such as whether the rows in the cursor can be updated.
2. Execute the Transact-SQL statement to populate the cursor.
3. Retrieve the rows in the cursor you want to see. The operation to retrieve one row or one block of rows from a cursor is called a fetch. Performing a series of fetches to retrieve rows in either a forward or backward direction is called scrolling.
4. Optionally, perform modification operations (update or delete) on the row at the current position in the cursor.
5. Close the cursor.

TSQL Cursors

The four cursor types for Transact-SQL cursors are:

- ❑ Static cursors
- ❑ Dynamic cursors
- ❑ Forward-only cursors
- ❑ Keyset-driven cursors

These cursors differ primarily in how changes to underlying data are handled. These cursors also vary in their ability to detect changes to the result set and in the resources, such as memory and space in tempdb, they consume. A cursor can detect changes to rows only when it attempts to fetch those rows a second time. There is no way for the data source to notify the cursor of changes to the currently fetched rows. The ability of a cursor to detect changes is also influenced by the transaction isolation level.

Static cursors detect few or no changes but consume relatively few resources while scrolling, although they store the entire cursor in tempdb. Dynamic cursors detect all changes but consume more resources while scrolling, although they make the lightest use of tempdb. Keyset-driven cursors lie in between, detecting most changes but at less expense than dynamic cursors.

These cursors also differ on the types of locks used:

- ❑ Static cursors retrieve the entire result set at the time the cursor is opened. This locks each row of the result set at open time.
- ❑ Keyset-driven cursors retrieve the keys of each row of the result set at the time the cursor is opened. This locks each row of the result set at open time.
- ❑ Dynamic cursors (including regular forward-only cursors) do not retrieve rows until they are fetched. Locks are not acquired on the rows until they have been fetched.
- ❑ Fast forward-only cursors vary in when they acquire their locks depending on the execution plan chosen by the query optimizer. If a dynamic plan is chosen, no locks are taken until the rows are fetched. If worktables are generated, then the rows are read into the worktable and locked at open time.

SQL Cursor Variables

SQL Server 2005 allows declaring variables with a type of cursor. To assign a cursor to a cursor variable, the SET statement is used. Cursors variables cannot be assigned a value with an assignment SELECT. Cursor variables can be declared with the DECLARE statement and used with other cursor-related statements. Cursor variables store a reference to a cursor definition. Cursor variables can be used to declare any type of cursor.

A cursor variable that currently has a cursor assigned to it can be referenced as a source in a:

- ❑ CLOSE statement.
- ❑ DEALLOCATE statement.
- ❑ FETCH statement.
- ❑ OPEN statement.
- ❑ Positioned DELETE or UPDATE statement.
- ❑ SET CURSOR variable statement (on the right side).

In all these statements, Microsoft SQL Server raises an error if a referenced cursor variable exists but does not have a cursor currently allocated to it. If a referenced cursor variable does not exist, SQL Server raises the same error raised for an undeclared variable of another type.

A cursor variable:

- Can be the target of either a cursor type or another cursor variable.
- Can be referenced as the target of an output cursor parameter in an EXECUTE statement if the cursor variable does not have a cursor currently assigned to it.
- Should be regarded as a pointer to the cursor.

Examples:

Using DECLARE

The following example uses a local variable named **@find** to retrieve contact information for all last names beginning with 'Man'.

```
USE AdventureWorks;
GO
DECLARE @find varchar(30);
SET @find = 'Man%';
SELECT LastName, FirstName, Phone
FROM Person.Contact
WHERE LastName LIKE @find;
```

Use DECLARE with two variables

The following example retrieves the names of Adventure Works Cycles sales representatives who are located in the North American sales territory and have at least $2,000,000 in sales for the year.

```
USE AdventureWorks;
GO
SET NOCOUNT ON;
GO
DECLARE @Group nvarchar(50), @Sales money;
SET @Group = N'North America';
SET @Sales = 2000000;
SET NOCOUNT OFF;
SELECT FirstName, LastName, SalesYTD
FROM Sales.vSalesPerson
WHERE TerritoryGroup = @Group and SalesYTD >= @Sales;
```

DECLARE CURSOR

Defines the attributes of a Transact-SQL server cursor, such as its scrolling behavior and the query used to build the result set on which the cursor operates. DECLARE CURSOR accepts both a syntax based on the SQL-92 standard and a syntax using a set of Transact-SQL extensions.

Syntax:

```
SQL 92 Syntax
DECLARE cursor_name [ INSENSITIVE ] [ SCROLL ] CURSOR FOR
select_statement [ FOR { READ ONLY | UPDATE [ OF column_name [ ,...n ] ]
} ]
[;]
Transact-SQL Extended Syntax
DECLARE cursor_name CURSOR[ LOCAL | GLOBAL ][ FORWARD_ONLY | SCROLL ][
STATIC | KEYSET | DYNAMIC | FAST_FORWARD ][ READ_ONLY | SCROLL_LOCKS |
OPTIMISTIC ][ TYPE_WARNING ]FOR select_statement[ FOR UPDATE [ OF
column_name [ ,...n ] ] ]
[;]
```

Arguments:

cursor_name: Is the name of the Transact-SQL server cursor defined. *cursor_name* must conform to the rules for identifiers. For more information about rules for identifiers, see Using Identifiers as Object Names.

INSENSITIVE: Defines a cursor that makes a temporary copy of the data to be used by the cursor. All requests to the cursor are answered from this temporary table in **tempdb**; therefore, modifications made to base tables are not reflected in the data returned by fetches made to this cursor, and this cursor does not allow modifications. When SQL-92 syntax is used, if INSENSITIVE is omitted, committed deletes and updates made to the underlying tables (by any user) are reflected in subsequent fetches.

SCROLL: Specifies that all fetch options (FIRST, LAST, PRIOR, NEXT, RELATIVE, ABSOLUTE) are available. If SCROLL is not specified in an SQL-92 DECLARE CURSOR, NEXT is the only fetch option supported. SCROLL cannot be specified if FAST_FORWARD is also specified.

select_statement: Is a standard SELECT statement that defines the result set of the cursor. The keywords COMPUTE, COMPUTE BY, FOR BROWSE, and INTO are not allowed within *select_statement* of a cursor declaration.

Microsoft SQL Server implicitly converts the cursor to another type if clauses in *select_statement* conflict with the functionality of the requested cursor type. For more information, see Using Implicit Cursor Conversions.

READ ONLY: Prevents updates made through this cursor. The cursor cannot be referenced in a WHERE CURRENT OF clause in an UPDATE or DELETE statement. This option overrides the default capability of a cursor to be updated.

UPDATE [OF *column_name* [,...*n*]]: Defines updatable columns within the cursor. If OF *column_name* [,...*n*] is specified, only the columns listed allow modifications. If UPDATE is specified without a column list, all columns can be updated.

cursor_name: Is the name of the Transact-SQL server cursor defined. *cursor_name* must conform to the rules for identifiers. For more information about rules for identifiers, see Using Identifiers as Object Names.

LOCAL: Specifies that the scope of the cursor is local to the batch, stored procedure, or trigger in which the cursor was created. The cursor name is only valid within this scope. The cursor can be referenced by local cursor variables in the batch, stored procedure, or trigger, or a stored procedure OUTPUT parameter. An OUTPUT parameter is used to pass the local cursor back to the calling batch, stored procedure, or trigger, which can assign the parameter to a cursor variable to reference the cursor after the stored procedure terminates. The cursor is implicitly deallocated when the batch, stored procedure, or trigger terminates, unless the cursor was passed back in an OUTPUT parameter. If it is passed back in an OUTPUT parameter, the cursor is deallocated when the last variable referencing it is deallocated or goes out of scope.

GLOBAL: Specifies that the scope of the cursor is global to the connection. The cursor name can be referenced in any stored procedure or batch executed by the connection. The cursor is only implicitly deallocated at disconnect.

FORWARD_ONLY: Specifies that the cursor can only be scrolled from the first to the last row. FETCH NEXT is the only supported fetch option. If FORWARD_ONLY is specified without the STATIC, KEYSET, or DYNAMIC keywords, the cursor operates as a DYNAMIC cursor. When neither FORWARD_ONLY nor SCROLL is specified, FORWARD_ONLY is the default, unless the keywords STATIC, KEYSET, or DYNAMIC are specified. STATIC, KEYSET, and DYNAMIC cursors default to SCROLL. Unlike database APIs such as ODBC and ADO, FORWARD_ONLY is supported with STATIC, KEYSET, and DYNAMIC Transact-SQL cursors.

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

KEYSET:

- Specifies that the membership and order of rows in the cursor are fixed when the cursor is opened. The set of keys that uniquely identify the rows is built into a table in **tempdb** known as the **keyset**.
- Changes to nonkey values in the base tables, either made by the cursor owner or committed by other users, are visible as the owner scrolls around the cursor. Inserts made by other users are not visible (inserts cannot be made through a Transact-SQL server cursor). If a row is deleted, an attempt to fetch the row returns an @@FETCH_STATUS of -2. Updates of key values from outside the cursor resemble a delete of the old row followed by an insert of the new row. The row with the new values is not visible, and attempts to fetch the row with the old values return an @@FETCH_STATUS of -2. The new values are visible if the update is done through the cursor by specifying the WHERE CURRENT OF clause.

DYNAMIC: Defines a cursor that reflects all data changes made to the rows in its result set as you scroll around the cursor. The data values, order, and membership of the rows can change on each fetch. The ABSOLUTE fetch option is not supported with dynamic cursors.

FAST_FORWARD: Specifies a FORWARD_ONLY, READ_ONLY cursor with performance optimizations enabled. FAST_FORWARD cannot be specified if SCROLL or FOR_UPDATE is also specified.

READ_ONLY: Prevents updates made through this cursor. The cursor cannot be referenced in a WHERE CURRENT OF clause in an UPDATE or DELETE statement. This option overrides the default capability of a cursor to be updated.

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

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

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

select_statement:

- Is a standard SELECT statement that defines the result set of the cursor. The keywords COMPUTE, COMPUTE BY, FOR BROWSE, and INTO are not allowed within *select_statement* of a cursor declaration.
- SQL Server implicitly converts the cursor to another type if clauses in *select_statement* conflict with the functionality of the requested cursor type. For more information, see Implicit Cursor Conversions.

FOR UPDATE [OF *column_name* [,...*n*]]: Defines updatable columns within the cursor. If OF *column_name* [**,**...*n*] is supplied, only the columns listed allow modifications. If UPDATE is specified without a column list, all columns can be updated, unless the READ_ONLY concurrency option was specified.

Examples:

Use simple cursor and syntax:

The result set generated at the opening of this cursor includes all rows and all columns in the table. This cursor can be updated, and all updates and deletes are represented in fetches made against this cursor. FETCH NEXT is the only fetch available because the SCROLL option has not been specified.

```
DECLARE vend_cursor CURSOR
FOR SELECT * FROM Purchasing.Vendor
OPEN vend_cursor
FETCH NEXT FROM vend_cursor
```

Use nested cursors to produce report output

This example shows how cursors can be nested to produce complex reports. The inner cursor is declared for each vendor.

```
SET NOCOUNT ON

DECLARE @vendor_id int, @vendor_name nvarchar(50),
@message varchar(80), @product nvarchar(50)

PRINT '-------- Vendor Products Report --------'

DECLARE vendor_cursor CURSOR FOR
SELECT VendorID, Name
FROM Purchasing.Vendor
WHERE PreferredVendorStatus = 1
ORDER BY VendorID

OPEN vendor_cursor

FETCH NEXT FROM vendor_cursor
INTO @vendor_id, @vendor_name

WHILE @@FETCH_STATUS = 0
BEGIN
PRINT ' '
SELECT @message = '----- Products From Vendor: ' +
@vendor_name

PRINT @message

-- Declare an inner cursor based
-- on vendor_id from the outer cursor.

DECLARE product_cursor CURSOR FOR
SELECT v.Name
FROM Purchasing.ProductVendor pv, Production.Product v
WHERE pv.ProductID = v.ProductID AND
pv.VendorID = @vendor_id-- Variable value from the outer cursor

OPEN product_cursor
FETCH NEXT FROM product_cursor INTO @product

IF @@FETCH_STATUS <> 0
PRINT '         <<None>>'
```

```
WHILE @@FETCH_STATUS = 0
BEGIN

SELECT @message = '         ' + @product
PRINT @message
FETCH NEXT FROM product_cursor INTO @product

END

CLOSE product_cursor
DEALLOCATE product_cursor

-- Get the next vendor.
FETCH NEXT FROM vendor_cursor
INTO @vendor_id, @vendor_name
END
CLOSE vendor_cursor
DEALLOCATE vendor_cursor
```

Opening cursor

OPEN opens a Transact-SQL server cursor and populates the cursor by executing the Transact-SQL statement specified on the DECLARE CURSOR or SET *cursor_variable* statement.

Syntax:

```
OPEN { { [ GLOBAL ] cursor_name } | cursor_variable_name }
```

Arguments:

GLOBAL: Specifies that *cursor_name* refers to a global cursor.

cursor_name: Is the name of a declared cursor. If both a global and a local cursor exist with *cursor_name* as their name, *cursor_name* refers to the global cursor if GLOBAL is specified; otherwise, *cursor_name* refers to the local cursor.

cursor_variable_name: Is the name of a cursor variable that references a cursor.

Examples:

The following example opens a cursor and fetches all the rows.

```
DECLARE Employee_Cursor CURSOR FOR
SELECT LastName, FirstName
FROM AdventureWorks.HumanResources.vEmployee
WHERE LastName like 'B%';

OPEN Employee_Cursor;
```

```
FETCH NEXT FROM Employee_Cursor;
WHILE @@FETCH_STATUS = 0
BEGIN
    FETCH NEXT FROM Employee_Cursor
END;

CLOSE Employee_Cursor;
DEALLOCATE Employee_Cursor;
```

Fetching cursor

The operation to retrieve a row from a cursor is called a fetch. The FETCH keyword is used to retrieve specific rows from a cursor.

Syntax:

```
FETCH
          [ [ NEXT | PRIOR | FIRST | LAST
                    | ABSOLUTE { n | @nvar }
                    | RELATIVE { n | @nvar }
               ]
               FROM
          ]
{ { [ GLOBAL ] cursor_name } | @cursor_variable_name }
[ INTO @variable_name [ ,...n ] ]
```

Arguments:

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

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

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

ABSOLUTE { *n* | *@nvar*}: If *n* or *@nvar* is positive, returns the row *n* rows from the front of the cursor and makes the returned row the new current row. If *n* or *@nvar* is negative, returns the row *n* rows before the end of the cursor and makes the returned row the new current row. 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*}: If *n* or @*nvar* is positive, returns the row *n* rows beyond the current row and makes the returned row the new current row. If *n* or @*nvar* is negative, returns the row *n* rows prior to the current row and makes the returned row the new current row. If *n* or @*nvar* is 0, returns the current row. If FETCH RELATIVE is specified with *n* or @*nvar* set to negative numbers or 0 on the first fetch done against a cursor, no rows are returned. *n* must be an integer constant and @*nvar* must be **smallint**, **tinyint**, or **int**.

GLOBAL: Specifies that *cursor_name* refers to a global cursor.

cursor_name: Is the name of the open cursor from which the fetch should be made. If both a global and a local cursor exist with *cursor_name* as their name, *cursor_name* to the global cursor if GLOBAL is specified and to the local cursor if GLOBAL is not specified.

@cursor_variable_name: Is the name of a cursor variable referencing the open cursor from which the fetch should be made.

INTO @variable_name[,...n]: Allows data from the columns of a fetch to be placed into local variables. Each variable in the list, from left to right, is associated with the corresponding column in the cursor result set. The data type of each variable must either match or be a supported implicit conversion of the data type of the corresponding result set column. The number of variables must match the number of columns in the cursor select list.

Examples:

Use FETCH in a simple cursor:

This example declares a simple cursor for the rows in the **Person.Contact** table with a last name beginning with B, and uses FETCH NEXT to step through the rows. The FETCH statements return the value for the column specified in the DECLARE CURSOR as a single-row result set.

```
USE AdventureWorks
GO
DECLARE contact_cursor CURSOR FOR
SELECT LastName FROM Person.Contact
WHERE LastName LIKE 'B%'
ORDER BY LastName

OPEN contact_cursor

-- Perform the first fetch.
FETCH NEXT FROM contact_cursor

-- Check @@FETCH_STATUS to see if there are any more rows to fetch.
WHILE @@FETCH_STATUS = 0
BEGIN
   -- This is executed as long as the previous fetch succeeds.
   FETCH NEXT FROM contact_cursor
END
```

```
CLOSE contact_cursor
DEALLOCATE contact_cursor
GO
```

Use FETCH to store values in variables :

This example is similar to the last example, except the output of the FETCH statements is stored in local variables rather than being returned directly to the client. The PRINT statement combines the variables into a single string and returns them to the client.

```
USE AdventureWorks
GO
-- Declare the variables to store the values returned by FETCH.
DECLARE @LastName varchar(50), @FirstName varchar(50)

DECLARE contact_cursor CURSOR FOR
SELECT LastName, FirstName FROM Person.Contact
WHERE LastName LIKE 'B%'
ORDER BY LastName, FirstName

OPEN contact_cursor

-- Perform the first fetch and store the values in variables.
-- Note: The variables are in the same order as the columns
-- in the SELECT statement.

FETCH NEXT FROM contact_cursor
INTO @LastName, @FirstName

-- Check @@FETCH_STATUS to see if there are any more rows to fetch.
WHILE @@FETCH_STATUS = 0
BEGIN

   -- Concatenate and display the current values in the variables.
   PRINT 'Contact Name: ' + @FirstName + ' ' +  @LastName

   -- This is executed as long as the previous fetch succeeds.
   FETCH NEXT FROM contact_cursor
   INTO @LastName, @FirstName
END

CLOSE contact_cursor
DEALLOCATE contact_cursor
GO
```

Declare a SCROLL cursor and use the other FETCH options :

This example creates a SCROLL cursor to allow full scrolling capabilities through the LAST, PRIOR, RELATIVE, and ABSOLUTE options.

```
USE AdventureWorks
GO
-- Execute the SELECT statement alone to show the
-- full result set that is used by the cursor.
SELECT LastName, FirstName FROM Person.Contact
ORDER BY LastName, FirstName

-- Declare the cursor.
DECLARE contact_cursor SCROLL CURSOR FOR
SELECT LastName, FirstName FROM Person.Contact
ORDER BY LastName, FirstName

OPEN contact_cursor

-- Fetch the last row in the cursor.
FETCH LAST FROM contact_cursor

-- Fetch the row immediately prior to the current row in the cursor.
FETCH PRIOR FROM contact_cursor

-- Fetch the second row in the cursor.
FETCH ABSOLUTE 2 FROM contact_cursor

-- Fetch the row that is three rows after the current row.
FETCH RELATIVE 3 FROM contact_cursor

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

CLOSE contact_cursor
DEALLOCATE contact_cursor
GO
```

Closing Cursor

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

Syntax:

```
CLOSE { { [ GLOBAL ] cursor_name } | cursor_variable_name }
```

Arguments:

- **`GLOBAL`** **:** Specifies that *cursor_name* refers to a global cursor.
- **`cursor_name:`** Is the name of an open cursor. If both a global and a local cursor exist with *cursor_name* as their name, *cursor_name* refers to the global cursor when GLOBAL is specified; otherwise, *cursor_name* refers to the local cursor.
- **`cursor_variable_name:`** Is the name of a cursor variable associated with an open cursor.

Deallocating Cursors

DEALLOCATE removes a cursor reference. When the last cursor reference is deallocated, the data structures comprising the cursor are released by Microsoft SQL Server.

Syntax:

```
DEALLOCATE { { [ GLOBAL ] cursor_name } | @cursor_variable_name }
```

Arguments:

`cursor_name`: Is the name of an already declared cursor. If both a global and a local cursor exist with *cursor_name* as their name, *cursor_name* refers to the global cursor if GLOBAL is specified and to the local cursor if GLOBAL is not specified.

`@cursor_variable_name:` Is the name of a **cursor** variable. *@cursor_variable_name* must be of type **cursor**.

Cursor: Disadvantages

Disadvantages of cursor are stated as follows:

- Individually row processing is very inefficient
- Default result sets are preferable most of the time
- Static, keyset cursors are worse for performance
  - Slower as the entire result set needs to be rebuilt
  - Take up TempDB space
- SQL Server is inherently a set-based database
  - Always more efficient to manipulate data as a set

Summary

- ❑ Cursors are used for row by row processing operations
- ❑ SQL Server 2005 has a rich set of cursor types
- ❑ You can define static, dynamic, read-only, and forward-only cursors
- ❑ Cursors are much slower than set based statements
- ❑ In most situations, a cursor usage can be replaced by an equivalent set based statement

Session 23: Stored Procedures

Learning Objectives

After completing this session, you will be able to:

- Define Stored Procedure
- Identify the types of Stored Procedures
- Create Stored Procedures
- Create Stored Procedures using WITH RECOMPILE option
- Creating Stored Procedures using WITH ENCRYPTION option
- Describe System Stored Procedures
- Explain Error Handling
- Describe CLR Integration
- Explain CLR Procedures
- Write EXECUTE AS clause

Stored Procedure

Stored procedures in Microsoft SQL Server are similar to procedures in other programming languages in that they can:

- Accept input parameters and return multiple values in the form of output parameters to the calling procedure or batch.
- Contain programming statements that perform operations in the database, including calling other procedures.
- Return a status value to a calling procedure or batch to indicate success or failure (and the reason for failure).

You can use the Transact-SQL EXECUTE statement to run a stored procedure. Stored procedures are different from functions in that they do not return values in place of their names and they cannot be used directly in an expression.

The benefits of using stored procedures in SQL Server rather than Transact-SQL programs stored locally on client computers are:

- They are registered at the server.
- They can have security attributes (such as permissions) and ownership chaining, and certificates can be attached to them. Users can be granted permission to execute a stored procedure without having to have direct permissions on the objects referenced in the procedure.
- They can enhance the security of your application. Parameterized stored procedures can help protect your application from SQL Injection attacks. For more information see SQL Injection.
- They allow modular programming. You can create the procedure once, and call it any number of times in your program. This can improve the maintainability of your application and allow applications to access the database in a uniform manner.

- They are named code allowing for delayed binding. This provides a level of indirection for easy code evolution.
- They can reduce network traffic. An operation requiring hundreds of lines of Transact-SQL code can be performed through a single statement that executes the code in a procedure, rather than by sending hundreds of lines of code over the network.

Types of Stored Procedures

There are many types of stored procedures available in Microsoft SQL Server 2005. This topic briefly describes each stored procedure type and includes an example of each.

- **User-defined Stored Procedures:** Stored procedures are modules or routines that encapsulate code for reuse. A stored procedure can take input parameters, return tabular or scalar results and messages to the client, invoke data definition language (DDL) and data manipulation language (DML) statements, and return output parameters. In SQL Server 2005, a stored procedure can be of two types: Transact-SQL or CLR.
- **Transact-SQL:** A Transact-SQL stored procedure is a saved collection of Transact-SQL statements that can take and return user-supplied parameters. For example, a stored procedure might contain the statements needed to insert a new row into one or more tables based on information supplied by the client application. Or, the stored procedure might return data from the database to the client application. For example, an e-commerce Web application might use a stored procedure to return information about specific products based on search criteria specified by the online user.
- **CLR:** A CLR stored procedure is a reference to a Microsoft .NET Framework common language runtime (CLR) method that can take and return user-supplied parameters. They are implemented as public, static methods on a class in a .NET Framework assembly. For more information, see CLR Stored Procedures.
- **Extended Stored Procedures:** Extended stored procedures let you create your own external routines in a programming language such as C. Extended stored procedures are DLLs that an instance of Microsoft SQL Server can dynamically load and run. Extended stored procedures run directly in the address space of an instance of SQL Server and are programmed by using the SQL Server Extended Stored Procedure API.
- System Stored Procedures:
  - Many administrative activities in SQL Server 2005 are performed through a special kind of procedure known as a system stored procedure. For example, **sys.sp_changedbowner** is a system stored procedure. System stored procedures are physically stored in the Resource database and have the **sp_** prefix. System stored procedures logically appear in the **sys** schema of every system- and user-defined database. In SQL Server 2005, GRANT, DENY, and REVOKE permissions can be applied to system stored procedures. For a complete list of system stored procedures, see System Stored Procedures (Transact-SQL).
  - SQL Server supports the system stored procedures that provide an interface from SQL Server to external programs for various maintenance activities. These extended stored procedures use the xp_ prefix. For a complete list of extended stored procedures, see General Extended Stored Procedures (Transact-SQL).

- **Temporary Stored Procedures:** Private and global temporary stored procedures, analogous to temporary tables, can be created with the # and ## prefixes added to the procedure name. # denotes a local temporary stored procedure; ## denotes a global temporary stored procedure. These procedures do not exist after SQL Server is shut down.

Creating Stored Procedures

Use CREATE PROCEDURE statement to create procedures

Syntax:

```
CREATE { PROC | PROCEDURE } [schema_name.] procedure_name [ ; number ]
    [ { @parameter [ type_schema_name. ] data_type }
        [ VARYING ] [ = default ] [ [ OUT [ PUT ]
    ] [ ,...n ]
[ WITH <procedure_option> [ ,...n ]
[ FOR REPLICATION ]
AS { <sql_statement> [;][ ...n ] | <method_specifier> }
[;]
<procedure_option> ::=
    [ ENCRYPTION ]
    [ RECOMPILE ]
    [ EXECUTE_AS_Clause ]
```

Arguments:

- VARYING specifies the result set supported as an output parameter. This parameter is dynamically constructed by the stored procedure and its contents may vary and applies only to cursor parameters.
- OUTPUT indicates that the parameter is an output parameter.
- RECOMPILE indicates that the Database Engine does not cache a plan for this procedure and the procedure is compiled at run time.
- ENCRYPTION indicates that SQL Server will convert the original text of the CREATE PROCEDURE statement to an obfuscated format. The output of the obfuscation is not directly visible in any of the catalog views in SQL Server 2005.
- EXECUTE AS specifies the security context under which to execute the stored procedure.
- FOR REPLICATION specifies that stored procedures, which are created for replication cannot be executed on the Subscriber.

Examples:

Using a simple procedure with a complex SELECT:

The following stored procedure returns all employees (first and last names supplied), their titles, and their department names from a view. This stored procedure does not use any parameters.

```
USE AdventureWorks;
GO
IF OBJECT_ID ( 'HumanResources.usp_GetAllEmployees', 'P' ) IS NOT NULL
    DROP PROCEDURE HumanResources.usp_GetAllEmployees;
GO
CREATE PROCEDURE HumanResources.usp_GetAllEmployees
AS
    SELECT LastName, FirstName, JobTitle, Department
    FROM HumanResources.vEmployeeDepartment;
GO
```

The usp_GetEmployees stored procedure can be executed in these ways:

```
EXECUTE HumanResources.usp_GetAllEmployees;
GO
-- Or
EXEC HumanResources.usp_GetAllEmployees;
GO
-- Or, if this procedure is the first statement within a batch:
HumanResources.usp_GetAllEmployees;
```

Using a simple procedure with parameters:

The following stored procedure returns only the specified employee (first and last name supplied), her title, and her department name from a view. This stored procedure accepts exact matches for the parameters passed.

```
USE AdventureWorks;
GO
IF OBJECT_ID ( 'HumanResources.usp_GetEmployees', 'P' ) IS NOT NULL
    DROP PROCEDURE HumanResources.usp_GetEmployees;
GO
CREATE PROCEDURE HumanResources.usp_GetEmployees
    @lastname varchar(40),
    @firstname varchar(20)
AS
    SELECT LastName, FirstName, JobTitle, Department
    FROM HumanResources.vEmployeeDepartment
    WHERE FirstName = @firstname AND LastName = @lastname;
GO
```

The usp_GetEmployees stored procedure can be executed in the following ways:

```
EXECUTE HumanResources.usp_GetEmployees 'Ackerman', 'Pilar';
-- Or
EXEC HumanResources.usp_GetEmployees @lastname = 'Ackerman', @firstname
= 'Pilar';
```

```
GO
-- Or
EXECUTE HumanResources.usp_GetEmployees @firstname = 'Pilar', @lastname
= 'Ackerman';
GO
-- Or, if this procedure is the first statement within a batch:
HumanResources.usp_GetEmployees 'Ackerman', 'Pilar';
```

Using a simple procedure with wildcard parameters:

The following stored procedure returns only the specified employees (first and last names supplied), their titles, and their departments from a view. This stored procedure pattern matches the parameters passed or, if not supplied, uses the preset default (last names that start with the letter D).

```
USE AdventureWorks;
GO
IF OBJECT_ID ( 'HumanResources.usp_GetEmployees2', 'P' ) IS NOT NULL
    DROP PROCEDURE HumanResources.usp_GetEmployees2;
GO
CREATE PROCEDURE HumanResources.usp_GetEmployees2
    @lastname varchar(40) = 'D%',
    @firstname varchar(20) = '%'
AS
    SELECT LastName, FirstName, JobTitle, Department
    FROM HumanResources.vEmployeeDepartment
    WHERE FirstName LIKE @firstname
        AND LastName LIKE @lastname;
GO
```

The usp_GetEmployees2 stored procedure can be executed in many combinations. Only a few combinations are shown here:

```
EXECUTE HumanResources.usp_GetEmployees2;
-- Or
EXECUTE HumanResources.usp_GetEmployees2 'Wi%';
-- Or
EXECUTE HumanResources.usp_GetEmployees2 @firstname = '%';
-- Or
EXECUTE HumanResources.usp_GetEmployees2 '[CK]ars[OE]n';
-- Or
EXECUTE HumanResources.usp_GetEmployees2 'Hesse', 'Stefen';
-- Or
EXECUTE HumanResources.usp_GetEmployees2 'H%', 'S%';
```

Using OUTPUT parameters:

The following example creates the usp_GetList stored procedure, which returns a list of products that have prices that do not exceed a specified amount. The example shows using multiple SELECT statements and multiple OUTPUT parameters. OUTPUT parameters allow an external procedure, a batch, or more than one Transact-SQL statement to access a value set during the procedure execution.

```
USE AdventureWorks;
GO
IF OBJECT_ID ( 'Production.usp_GetList', 'P' ) IS NOT NULL
    DROP PROCEDURE Production.usp_GetList;
GO
CREATE PROCEDURE Production.usp_GetList @product varchar(40)
    , @maxprice money
    , @compareprice money OUTPUT
    , @listprice money OUT
AS
    SELECT p.name AS Product, p.ListPrice AS 'List Price'
    FROM Production.Product p
    JOIN Production.ProductSubcategory s
      ON p.ProductSubcategoryID = s.ProductSubcategoryID
    WHERE s.name LIKE @product AND p.ListPrice < @maxprice;
-- Populate the output variable @listprice.
SET @listprice = (SELECT MAX(p.ListPrice)
        FROM Production.Product p
        JOIN  Production.ProductSubcategory s
          ON p.ProductSubcategoryID = s.ProductSubcategoryID
        WHERE s.name LIKE @product AND p.ListPrice < @maxprice);
-- Populate the output variable @compareprice.
SET @compareprice = @maxprice;
GO
```

Execute usp_GetList to return a list of Adventure Works products (Bikes) that cost less than $700. The OUTPUT parameters **@cost** and **@compareprices** are used with control-of-flow language to return a message in the **Messages** window.

```
DECLARE @compareprice money, @cost money
EXECUTE Production.usp_GetList '%Bikes%', 700,
    @compareprice OUT,
    @cost OUTPUT
IF @cost <= @compareprice
BEGIN
    PRINT 'These products can be purchased for less than
    $'+RTRIM(CAST(@compareprice AS varchar(20)))+'.'
END
ELSE
```

```
    PRINT 'The prices for all products in this category exceed
    $'+ RTRIM(CAST(@compareprice AS varchar(20)))+'.'
```

Creating Stored Procedures WITH RECOMPILE option

If database is changed by actions such as adding indexes or changing data in indexed columns, then the original query plans applied to access its tables should be optimized by recompiling them.

Creating a stored procedure that specifies the WITH RECOMPILE option in its definition indicates that SQL Server does not cache a plan for stored procedure. The stored procedure is recompiled each time it is executed.

Example:

```
CREATE PROCEDURE dbo.usp_product_by_vendor @name varchar(30) = '%'
WITH RECOMPILE
AS
    SELECT v.Name AS 'Vendor name', p.Name AS 'Product name'
    FROM Purchasing.Vendor v
    JOIN Purchasing.ProductVendor pv
      ON v.VendorID = pv.VendorID
    JOIN Production.Product p
      ON pv.ProductID = p.ProductID
    WHERE v.Name LIKE @name;
GO
```

Creating Stored Procedures WITH ENCRYPTION option

WITH ENCRYPTION indicates that SQL Server will convert the original text of the CREATE PROCEDURE statement to an obfuscated format. The output of the obfuscation is not directly visible in any of the catalog views in SQL Server 2005.

Example:

The following example creates the HumanResources.usp_encrypt_this stored procedure.

```
USE AdventureWorks;
GO
IF OBJECT_ID ( 'HumanResources.usp_encrypt_this', 'P' ) IS NOT NULL
    DROP PROCEDURE HumanResources.usp_encrypt_this;
GO
CREATE PROCEDURE HumanResources.usp_encrypt_this
WITH ENCRYPTION
AS
    SELECT EmployeeID, Title, NationalIDNumber, VacationHours,
SickLeaveHours
    FROM HumanResources.Employee;
```

```
GO
```

The WITH ENCRYPTION option prevents the definition of the stored procedure from being returned, as shown by the following examples.

Run sp_helptext:

```
EXEC sp_helptext 'HumanResources.usp_encrypt_this';
```

Here is the result set.

```
The text for object 'HumanResources.usp_encrypt_this' is encrypted.
```

Directly query the sys.sql_modules catalog view:

```
USE AdventureWorks;
GO
SELECT definition FROM sys.sql_modules
WHERE object_id = OBJECT_ID('HumanResources.usp_encrypt_this');
```

Here is the result set.

```
definition
----------------------
NULL

(1 row(s) affected)
```

System Stored Procedures

System stored procedures installed by SQL Server for administrative and informative purposes:

- Reside on the master database
- **Executable from any database:** Takes execution context of current database
- Start with ‘sp_’ prefix:
  - **Example: sp_help,** a system procedure that gives information on database objects
  - **Best practice:** User procedures should not have this prefix
- **Catalog procedures:** Access and modify system tables
  - Examples are sp_databases, sp_tables, sp_stored_procedures

Error Handling

In SQL Server 2000 and earlier versions @@ERROR function is the primary means of detecting errors in T-SQL statements.

The SQL Server 2005 Database Engine introduces the TRY…CATCH blocks to implement structured exception handling, which provides improved functionality.

Try.. catch block provides the following structure:

- Try block contains protected transactions
- Catch block handles errors

Syntax:

```
BEGIN TRY
{  sql_statement | statement_block }
END TRY
BEGIN CATCH
{  sql_statement | statement_block }
END CATCH
```

Example:

```
CREATE TABLE tbl_TestData( ColA int primary key, ColB int)

CREATE PROCEDURE dbo. usp_AddData @a int ,@b int
AS
BEGIN TRY
      INSERT INTO tbl_TestData VALUES (@a, @b)
END TRY
BEGIN CATCH
      SELECT ERROR_NUMBER() ErrorNumber,  ERROR_MESSAGE() [Message]
END CATCH
EXEC dbo. usp_AddData 1 , 1
EXEC dbo. usp_AddData 2 , 2
EXEC dbo. usp_AddData 2 , 3 --violates the primary key
```

Try…catch uses error functions to capture error information:

- ERROR_NUMBER() returns the error number.
- ERROR_MESSAGE() returns the complete text of the error message. The text includes the values supplied for any substitutable parameters such as lengths, object names, or times.
- ERROR_SEVERITY() returns the error severity.
- ERROR_STATE() returns the error state number.
- ERROR_LINE() returns the line number inside the routine that caused the error.
- ERROR_PROCEDURE() returns the name of the stored procedure or trigger where the error has occurred.

Error functions can only be applied inside the catch block.

Database Engine Error Severities:

- When an error is raised by the SQL Server Database Engine, the severity of the error indicates the type of problem encountered by SQL Server.
- A Try…catch construct catches all execution errors with severity greater than 10 that do not terminate the database connection.
- Errors with severity from zero through 10 are informational messages and do not cause execution to jump from the catch block of a Try…catch construct.

- Errors that terminate the database connection, usually with severity from 20 through 25, are not handled by the catch block because execution is aborted when the connection terminates.

CLR Integration

- CLR Integration in SQL Server 2005 allows you to create managed objects. Including stored procedure, triggers, user-defined types, and aggregates in any .NET language.
- Benefits of Integration:
  - Enhanced Programming mode: The .NET languages offer richer development capabilities than T-SQL does.
  - Enhanced safety and security: Managed code runs in CLR environment hosted by the SQL Server Database engine. This provides a safer and more secure system than extended stored procedures do.
  - Performance and scalability: Managed code and managed execution can deliver improved performance over T-SQL in many situations.

CLR Procedures

In SQL Server 2005, you can create a database object inside an instance of SQL Server that is programmed in an assembly created in the Microsoft .NET Framework common language runtime (CLR). Database objects that can leverage the rich programming model provided by the CLR include triggers, stored procedures, functions, aggregate functions, and types.

Creating a CLR stored procedure in SQL Server involves the following steps:

- Define the stored procedure as a static method of a class in a language supported by the .NET Framework. For more information about how to program CLR stored procedures, see CLR Stored Procedures. Then, compile the class to build an assembly in the .NET Framework by using the appropriate language compiler.
- Register the assembly in SQL Server by using the CREATE ASSEMBLY statement.
- Create the stored procedure that references the registered assembly by using the CREATE PROCEDURE statement.

Example:

The following example creates the GetPhotoFromDB stored procedure that references the GetPhotoFromDB method of the LargeObjectBinary class in the HandlingLOBUsingCLR assembly. Before the stored procedure is created, the HandlingLOBUsingCLR assembly is registered in the local database.

```
CREATE ASSEMBLY HandlingLOBUsingCLR
FROM
'\\MachineName\HandlingLOBUsingCLR\bin\Debug\HandlingLOBUsingCLR.dll'';
GO
CREATE PROCEDURE dbo.GetPhotoFromDB
(
    @ProductPhotoID int,
    @CurrentDirectory nvarchar(1024),
    @FileName nvarchar(1024)
)
```

```
AS EXTERNAL NAME HandlingLOBUsingCLR.LargeObjectBinary.GetPhotoFromDB;
GO
```

EXECUTE AS clause

In SQL Server 2005 you can implicitly define the execution context of the following user-defined modules: functions (except inline table-valued functions), procedures, queues, and triggers.

By specifying the context in which the module is executed, you can control which user account the SQL Server 2005 Database Engine uses to validate permissions on objects that are referenced by the module. This provides additional flexibility and control in managing permissions across the object chain that exists between user-defined modules and the objects referenced by those modules. Permissions must be granted to users only on the module itself, without having to grant them explicit permissions on the referenced objects. Only the user that the module is running as must have permissions on the objects accessed by the module.

Syntax:

```
Functions (except inline table-valued functions), Stored Procedures, and
DML Triggers
{ EXEC | EXECUTE } AS { CALLER | SELF | OWNER | 'user_name' }

DDL Triggers with Database Scope
{ EXEC | EXECUTE } AS { CALLER | SELF | 'user_name' }

DDL Triggers with Server Scope
{ EXEC | EXECUTE } AS { CALLER | SELF | 'login_name' }

Queues
{ EXEC | EXECUTE } AS { SELF | OWNER | 'user_name' }
```

Arguments:

`CALLER:`

- Specifies the statements inside the module are executed in the context of the caller of the module. The user executing the module must have appropriate permissions not only on the module itself, but also on any database objects that are referenced by the module.
- CALLER is the default for all modules except queues, and is the same as SQL Server 2000 behavior.
- CALLER cannot be specified in a CREATE QUEUE or ALTER QUEUE statement.

`SELF:`

- EXECUTE AS SELF is equivalent to EXECUTE AS *user_name*, where the specified user is the person creating or altering the module. The actual user ID of the person creating or modifying the modules is stored in the **execute_as_principal_id** column in the **sys.sql_modules** or **sys.service_queues** catalog view.
- SELF is the default for queues.

OWNER: Specifies the statements inside the module executes in the context of the current owner of the module. If the module does not have a specified owner, the owner of the schema of the module is used. OWNER cannot be specified for DDL triggers.

`'user_name'`:

- Specifies the statements inside the module execute in the context of the user specified in *user_name*. Permissions for any objects within the module are verified against *user_name*. *user_name* cannot be specified for DDL triggers with server scope. Use *login_name* instead.
- *user_name* must exist in the current database and must be a singleton account. *user_name* cannot be a group, role, certificate, key, or built-in account, such as NT AUTHORITY\LocalService, NT AUTHORITY\NetworkService, or NT AUTHORITY\LocalSystem.
- The user ID of the execution context is stored in metadata and can be viewed in the **execute_as_principal_id** column in the **sys.sql_modules** or **sys.assembly_modules** catalog view.

`'login_name':`

- Specifies the statements inside the module execute in the context of the SQL Server login specified in *login_name*. Permissions for any objects within the module are verified against *login_name*. *login_name* can be specified only for DDL triggers with server scope.
- *login_name* cannot be a group, role, certificate, key, or built-in account, such as NT AUTHORITY\LocalService, NT AUTHORITY\NetworkService, or NT AUTHORITY\LocalSystem.

Example:

Assume the following conditions:

- **CompanyDomain\SQLUsers** group has access to the **Sales** database.
- **CompanyDomain\SqlUser1** is a member of **SQLUsers** and, therefore, has access to the **Sales** database.
- The user that is creating or altering the module has permissions to create principals.

When the following CREATE PROCEDURE statement is run, the CompanyDomain\SqlUser1 is implicitly created as a database principal in the Sales database.

```
USE Sales;
GO
CREATE PROCEDURE dbo.usp_Demo
WITH EXECUTE AS 'CompanyDomain\SqlUser1'
AS
SELECT user_name();
GO
```

Using EXECUTE AS CALLER Stand-alone Statement:

Use the EXECUTE AS CALLER stand-alone statement inside a module to set the execution context to the caller of the module.

Assume the following stored procedure is called by SqlUser2.

```
CREATE PROCEDURE dbo.usp_Demo
WITH EXECUTE AS 'SqlUser1'
AS
SELECT user_name(); -- Shows execution context is set to SqlUser1.
EXECUTE AS CALLER;
SELECT user_name(); -- Shows execution context is set to SqlUser2, the
caller of the module.
REVERT;
SELECT user_name(); -- Shows execution context is set to SqlUser1.
GO
```

ALTER PROCEDURE

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

Example:

The following example alters the usp_vendor_info_all stored procedure (without encryption) to return only those vendors that supply paint, have an excellent credit rating, and who currently are available. The LEFT and CASE functions and the ORDER BY clause customize the appearance of the result set.

```
ALTER PROCEDURE dbo.usp_vendor_info_all
    @product varchar(25)
AS
    SELECT LEFT(v.Name, 25) AS Vendor, LEFT(p.Name, 25) AS 'Product
name',
    'Credit rating' = CASE v.CreditRating
        WHEN 1 THEN 'Superior'
        WHEN 2 THEN 'Excellent'
        WHEN 3 THEN 'Above average'
        WHEN 4 THEN 'Average'
        WHEN 5 THEN 'Below average'
        ELSE 'No rating'
        END
    , Availability = CASE v.ActiveFlag
        WHEN 1 THEN 'Yes'
        ELSE 'No'
        END
    FROM Purchasing.Vendor v
    INNER JOIN Purchasing.ProductVendor pv
      ON v.VendorID = pv.VendorID
    INNER JOIN Production.Product p
      ON pv.ProductID = p.ProductID
    WHERE p.Name LIKE @product
    ORDER BY v.Name ASC;
GO
```

Summary

- A stored procedure is a piece of TSQL code stored in the database.
- Stored procedures provide the best possible performance.
- Stored procedures increase the security of the data access code.
- Temporary stored procedures are stored in TempDB.
- Extended stored procedures are replaced with CLR Store procedures in SQL Server 2005.

Session 26: Functions

Learning Objectives

After completing this session, you will be able to:

- ❑ Apply user-defined functions
- ❑ Create user-defined functions
- ❑ List the types of user-defined functions

User-defined Functions

Like functions in programming languages, Microsoft SQL Server 2005 user-defined functions are routines that accept parameters, perform an action, such as a complex calculation, and return the result of that action as a value. The return value can either be a single scalar value or a result set.

User-defined Function Benefits: The benefits of using user-defined functions in SQL Server are:

- ❑ **They allow modular programming.:** You can create the function once, store it in the database, and call it any number of times in your program. User-defined functions can be modified independently of the program source code.
- ❑ **They allow faster execution:** Similar to stored procedures, Transact-SQL user-defined functions reduce the compilation cost of Transact-SQL code by caching the plans and reusing them for repeated executions. This means the user-defined function does not need to be reparsed and reoptimized with each use resulting in much faster execution times. CLR functions offer significant performance advantage over Transact-SQL functions for computational tasks, string manipulation, and business logic. Transact-SQL functions are better suited for data-access intensive logic.
- ❑ **They can reduce network traffic:** An operation that filters data based on some complex constraint that cannot be expressed in a single scalar expression can be expressed as a function. The function can then invoked in the WHERE clause to reduce the number or rows sent to the client.

SQL Server 2005 supports three types of user-defined functions:

- ❑ Scalar functions
- ❑ Inline table-valued functions
- ❑ Multi statement table-valued functions

CREATE FUNCTION

Creates a user-defined function. This is a saved Transact-SQL or common language runtime (CLR) routine that returns a value. User-defined functions cannot be used to perform actions that modify the database state. User-defined functions, like system functions, can be invoked from a query. Scalar functions can be executed by using an EXECUTE statement like stored procedures.

User-defined functions are modified by using ALTER FUNCTION and dropped by using DROP FUNCTION.

Syntax:

```
Scalar Functions
CREATE FUNCTION [ schema_name. ] function_name
( [ { @parameter_name [ AS ][ type_schema_name. ] parameter_data_type
    [ = default ] }
    [ ,...n ]
  ]
)
RETURNS return_data_type
    [ WITH <function_option> [ ,...n ] ]
    [ AS ]
    BEGIN
                function_body
        RETURN scalar_expression
    END
[ ; ]

Inline Table-valued Functions
CREATE FUNCTION [ schema_name. ] function_name
( [ { @parameter_name [ AS ] [ type_schema_name. ] parameter_data_type
    [ = default ] }
    [ ,...n ]
  ]
)
RETURNS TABLE
    [ WITH <function_option> [ ,...n ] ]
    [ AS ]
    RETURN [ ( ] select_stmt [ ) ]
[ ; ]

Multistatement Table-valued Functions
CREATE FUNCTION [ schema_name. ] function_name
( [ { @parameter_name [ AS ] [ type_schema_name. ] parameter_data_type
    [ = default ] }
    [ ,...n ]
  ]
)
RETURNS @return_variable TABLE < table_type_definition >
    [ WITH <function_option> [ ,...n ] ]
    [ AS ]
    BEGIN
                function_body
        RETURN
    END
[ ; ]

CLR Functions
CREATE FUNCTION [ schema_name. ] function_name
( { @parameter_name [AS] [ type_schema_name. ] parameter_data_type
        [ = default ] }
    [ ,...n ]
)
RETURNS { return_data_type | TABLE <clr_table_type_definition> }
    [ WITH <clr_function_option> [ ,...n ] ]
    [ AS ] EXTERNAL NAME <method_specifier>
[ ; ]

Method Specifier
<method_specifier>::=
    assembly_name.class_name.method_name

Function Options
<function_option>::=
{
    [ ENCRYPTION ]
  | [ SCHEMABINDING ]
  | [ RETURNS NULL ON NULL INPUT | CALLED ON NULL INPUT ]
  | [ EXECUTE_AS_Clause ]
}
```

Arguments:

schema_name: Is the name of the schema to which the user-defined function belongs.

function_name: Is the name of the user-defined function. 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 user-defined function. One or more parameters can be declared.
- A function can have a maximum of 1,024 parameters. The value of each declared parameter must be supplied by the user when the function is executed, unless a default for the parameter is defined.
- Specify a parameter name by using an at sign (@) as the first character. The parameter name must comply with the rules for identifiers. Parameters are local to the function; the same parameter names can be used in other functions. Parameters can take the place only of constants; they cannot be used instead of table names, column names, or the names of other database objects.

`[ type_schema_name. ] parameter_data_type:`

- Is the parameter data type, and optionally the schema to which it belongs. For Transact-SQL functions, all data types, including CLR user-defined types, are allowed except the **timestamp** data type. For CLR functions, all data types, including CLR user-defined types, are allowed except **text**, **ntext**, **image**, and **timestamp** data types. The nonscalar types **cursor** and **table** cannot be specified as a parameter data type in either Transact-SQL or CLR functions.
- If *type_schema_name* is not specified, the SQL Server 2005 Database Engine looks for the *scalar_parameter_data_type* in the following order:
  - The schema that contains the names of SQL Server system data types.
  - The default schema of the current user in the current database.
  - The **dbo** schema in the current database.

`[ = default ]:`

- Is a default value for the parameter. If a *default* value is defined, the function can be executed without specifying a value for that parameter.
- When a parameter of the function has a default value, the keyword DEFAULT must be specified when the function to retrieve the default value. This behavior is different from using parameters with default values in stored procedures in which omitting the parameter also implies the default value.

`return_data_type:` Is the return value of a scalar user-defined function. For Transact-SQL functions, all data types, including CLR user-defined types, are allowed except the **timestamp** data type. For CLR functions, all data types, including CLR user-defined types, are allowed except **text**, **ntext**, **image**, and **timestamp** data types. The nonscalar types **cursor** and **table** cannot be specified as a return data type in either Transact-SQL or CLR functions.

`function_body:`

- Specifies that a series of Transact-SQL statements, which together do not produce a side effect such as modifying a table, define the value of the function. *function_body* is used only in scalar functions and multistatement table-valued functions.
- In scalar functions, *function_body* is a series of Transact-SQL statements that together evaluate to a scalar value.
- In multistatement table-valued functions, *function_body* is a series of Transact-SQL statements that populate a TABLE return variable.

`scalar_expression:` Specifies the scalar value that the scalar function returns.

TABLE:

- Specifies that the return value of the table-valued function is a table. Only constants and *@local_variables* can be passed to table-valued functions.
- In inline table-valued functions, the TABLE return value is defined through a single SELECT statement. Inline functions do not have associated return variables.
- In multistatement table-valued functions, *@return_variable* is a TABLE variable, used to store and accumulate the rows that should be returned as the value of the function. *@return_variable* can be specified only for Transact-SQL functions and not for CLR functions.

`select_stmt:` Is the single SELECT statement that defines the return value of an inline table-valued function.

`EXTERNAL NAME <method_specifier>, assembly_name.class_name.method_name:` Specifies the method of an assembly to bind with the function. *assembly_name* must match an existing assembly in SQL Server in the current database with visibility on. *class_name* must be a valid SQL Server identifier and must exist as a class in the assembly. If the class has a namespace-qualified name that uses a period (**.**) to separate namespace parts, the class name must be delimited by using brackets (**[ ]**) or quotation marks (**" "**). *method_name* must be a valid SQL Server identifier and must exist as a static method in the specified class.

`ENCRYPTION:` Indicates that the Database Engine encrypts the catalog view columns that contain the text of the CREATE FUNCTION statement. Using ENCRYPTION prevents the function from being published as part of SQL Server replication. ENCRYPTION cannot be specified for CLR functions.

`SCHEMABINDING:`

- ❑ Specifies that the function is bound to the database objects that it references. This condition will prevent changes to the function if other schema-bound objects are referencing it.
- ❑ The binding of the function to the objects it references is removed only when one of the following actions occurs:
  - o The function is dropped.
  - o The function is modified by using the ALTER statement with the SCHEMABINDING option not specified.

A function can be schema bound only if the following conditions are true:

- ❑ The function is a Transact-SQL function.
- ❑ The user-defined functions and views referenced by the function are also schema-bound.
- ❑ The objects referenced by the function are referenced using a two-part name.
- ❑ The function and the objects it references belong to the same database.
- ❑ The user who executed the CREATE FUNCTION statement has REFERENCES permission on the database objects that the function references.

SCHEMABINDING cannot be specified for CLR functions or functions that reference alias data types.

`RETURNS NULL ON NULL INPUT | CALLED ON NULL INPUT:` Specifies the **OnNULLCall** attribute of a scalar-valued function. If not specified, CALLED ON NULL INPUT is implied by default. This means that the function body executes even if NULL is passed as an argument.

If RETURNS NULL ON NULL INPUT is specified in a CLR function, it indicates that SQL Server can return NULL when any of the arguments it receives is NULL, without actually invoking the body of the function. If the method of a CLR function specified in <method_specifier> already has a custom attribute that indicates RETURNS NULL ON NULL INPUT, but the CREATE FUNCTION statement indicates CALLED ON NULL INPUT, the CREATE FUNCTION statement takes precedence. The **OnNULLCall** attribute cannot be specified for CLR table-valued functions.

EXECUTE AS Clause: Specifies the security context under which the user-defined function is executed. Therefore, you can control which user account SQL Server uses to validate permissions on any database objects that are referenced by the function.

Scalar functions

User-defined scalar functions return a single data value of the type defined in the RETURNS clause. For an inline scalar function, there is no function body; the scalar value is the result of a single statement. For a multistatement scalar function, the function body, defined in a BEGIN...END block, contains a series of Transact-SQL statements that return the single value. The return type can be any data type except **text**, **ntext**, **image**, **cursor**, and **timestamp**.

Example:

The following examples creates a multistatement scalar function. The function takes one input value, a ProductID, and returns a single data value, the aggregated quantity of the specified product in inventory.

```
CREATE FUNCTION dbo.ufnGetStock(@ProductID int)
RETURNS int
AS
-- Returns the stock level for the product.
BEGIN
    DECLARE @ret int;
    SELECT @ret = SUM(p.Quantity)
    FROM Production.ProductInventory p
    WHERE p.ProductID = @ProductID
        AND p.LocationID = '6';
     IF (@ret IS NULL)
        SET @ret = 0
    RETURN @ret
END;
GO
```

The following example uses the ufnGetStock function to return the current inventory quantity for products that have an ProductModelID between 75 and 80.

```
USE AdventureWorks;
GO
SELECT ProductModelID, Name, dbo.ufnGetStock(ProductID)AS CurrentSupply
FROM Production.Product
WHERE ProductModelID BETWEEN 75 and 80;
GO
```

Inline Table-Valued Functions

- Inline functions should only have TABLE as the return type.
- A single SELECT statement is present in the body of the function.
- Function body is not delimited by BEGIN and END.

Example:

The following example creates an inline table-valued function. The function takes one input parameter, a customer (store) ID, and returns the columns ProductID, Name, and the aggregate of year-to-date sales as YTD Total for each product sold to the store.

```
USE AdventureWorks;
GO
CREATE FUNCTION Sales.fn_SalesByStore (@storeid int)
RETURNS TABLE
AS
RETURN
(
    SELECT P.ProductID, P.Name, SUM(SD.LineTotal) AS 'YTD Total'
    FROM Production.Product AS P
      JOIN Sales.SalesOrderDetail AS SD ON SD.ProductID = P.ProductID
      JOIN Sales.SalesOrderHeader AS SH ON SH.SalesOrderID =
SD.SalesOrderID
    WHERE SH.CustomerID = @storeid
    GROUP BY P.ProductID, P.Name
);
GO
```

The following example invokes the function and specifies customer ID 602.

```
SELECT * FROM Sales.fn_SalesByStore (602);
```

Multi-Valued Table Functions

- RETURNS clause always specifies a Table variable
- Lists the column names and their data types
- The only statements allowed in the function body are:
    - Assignment statements
    - Control-of-flow
    - DECLARE statements
    - SELECT that assign values to local variables
    - Cursors operations on local cursors
    - INSERT,UPDATE, and DELETE that modify local table variables
- Invocation similar to inline functions

Example:

The following example creates a table-valued function. The function takes a single input parameter, an EmployeeID and returns a list of all the employees who report to the specified employee directly or indirectly.

```
USE AdventureWorks;
GO
CREATE FUNCTION dbo.fn_FindReports (@InEmpID INTEGER)
RETURNS @retFindReports TABLE
(
    EmployeeID int primary key NOT NULL,
    Name nvarchar(255) NOT NULL,
    Title nvarchar(50) NOT NULL,
    EmployeeLevel int NOT NULL,
    Sort nvarchar (255) NOT NULL
)
--Returns a result set that lists all the employees who report to the
--specific employee directly or indirectly.*/
AS
BEGIN
   WITH DirectReports(Name, Title, EmployeeID, EmployeeLevel, Sort) AS
    (SELECT CONVERT(Varchar(255), c.FirstName + ' ' + c.LastName),
        e.Title,
        e.EmployeeID,
        1,
        CONVERT(Varchar(255), c.FirstName + ' ' + c.LastName)
     FROM HumanResources.Employee AS e
          JOIN Person.Contact AS c ON e.ContactID = c.ContactID
     WHERE e.EmployeeID = @InEmpID
   UNION ALL
     SELECT CONVERT(Varchar(255), REPLICATE ('| ' , EmployeeLevel) +
        c.FirstName + ' ' + c.LastName),
        e.Title,
        e.EmployeeID,
        EmployeeLevel + 1,
        CONVERT (Varchar(255), RTRIM(Sort) + '| ' + FirstName + ' ' +
                 LastName)
     FROM HumanResources.Employee as e
          JOIN Person.Contact AS c ON e.ContactID = c.ContactID
          JOIN DirectReports AS d ON e.ManagerID = d.EmployeeID
    )
-- copy the required columns to the result of the function
   INSERT @retFindReports
   SELECT EmployeeID, Name, Title, EmployeeLevel, Sort
   FROM DirectReports
   RETURN
END;
GO
```

In the following example, the function is invoked.

```
-- Example invocation
SELECT EmployeeID, Name, Title, EmployeeLevel
FROM dbo.fn_FindReports(109)
ORDER BY Sort;
```

Summary

- Inline table functions have only a simple SELECT statement.
- Multi-valued table functions can have multiple statements.

Session 29: Triggers

Learning Objectives

After completing this session, you will be able to:

- ❑ Define trigger
- ❑ Create trigger
- ❑ Identify the types of trigger

Triggers

Microsoft SQL Server 2005 provides two primary mechanisms for enforcing business rules and data integrity: constraints and triggers. A trigger is a special type of stored procedure that automatically takes effect when a language event executes.

SQL Server includes two general types of triggers:

- ❑ DML triggers
- ❑ DDL triggers.

DDL triggers are new to SQL Server 2005. These triggers are invoked when a data definition language (DDL) event takes place in the server or database.

DML triggers are invoked when a data manipulation language (DML) event takes place in the database. DML events include INSERT, UPDATE, or DELETE statements that modify data in a specified table or view. A DML trigger can query other tables and can include complex Transact-SQL statements. The trigger and the statement that fires it are treated as a single transaction, which can be rolled back from within the trigger. If a severe error is detected (for example, insufficient disk space), the entire transaction automatically rolls back.

DML triggers are useful in these ways:

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

Types of DML Triggers

- **AFTER Triggers:** AFTER triggers are executed after the action of the INSERT, UPDATE, or DELETE statement is performed. Specifying AFTER is the same as specifying FOR, which is the only option available in earlier versions of Microsoft SQL Server. AFTER triggers can be specified only on tables.
- **INSTEAD OF Triggers:** INSTEAD OF triggers are executed in place of the usual triggering action. INSTEAD OF triggers can also be defined on views with one or more base tables, where they can extend the types of updates a view can support.
- **CLR Triggers:** A CLR Trigger can be either an AFTER or INSTEAD OF trigger. A CLR trigger can also be a DDL trigger. Instead of executing a Transact-SQL stored procedure, a CLR trigger executes one or more methods written in managed code that are members of an assembly created in the .NET Framework and uploaded in SQL Server

CREATE TRIGGER

Syntax:

```
Trigger on an INSERT, UPDATE, or DELETE statement to a table or view
(DML Trigger)
CREATE TRIGGER [ schema_name . ]trigger_name
ON { table | view }
[ WITH <dml_trigger_option> [ ,...n ] ]
{ FOR | AFTER | INSTEAD OF }
{ [ INSERT ] [ , ] [ UPDATE ] [ , ] [ DELETE ] }
[ WITH APPEND ]
[ NOT FOR REPLICATION ]
AS { sql_statement  [ ; ] [ ...n ] | EXTERNAL NAME <method specifier [ ;
] > }

<dml_trigger_option> ::=
    [ ENCRYPTION ]
    [ EXECUTE AS Clause ]

<method_specifier> ::=
    assembly_name.class_name.method_name

Trigger on a CREATE, ALTER, DROP, GRANT, DENY, REVOKE, or UPDATE
STATISTICS statement (DDL Trigger)
CREATE TRIGGER trigger_name
ON { ALL SERVER | DATABASE }
[ WITH <ddl_trigger_option> [ ,...n ] ]
{ FOR | AFTER } { event_type | event_group } [ ,...n ]
AS { sql_statement  [ ; ] [ ...n ] | EXTERNAL NAME < method specifier >
[ ; ] }

<ddl_trigger_option> ::=
    [ ENCRYPTION ]
    [ EXECUTE AS Clause ]

<method_specifier> ::=
    assembly_name.class_name.method_name
```

Arguments:

schema_name: Is the name of the schema to which a DML trigger belongs. DML triggers are scoped to the schema of the table or view on which they are created. *schema_name* cannot be specified for DDL triggers.

trigger_name: Is the name of the trigger. A *trigger_name* must comply with the rules for identifiers, with the exception that *trigger_name* cannot start with # or ##.

table | view: Is the table or view on which the DML trigger is executed and is sometimes referred to as the trigger table or trigger view. Specifying the fully qualified name of the table or view is optional. A view can be referenced only by an INSTEAD OF trigger.

DATABASE: Applies the scope of a DDL trigger to the current database. If specified, the trigger fires whenever *event_type* or *event_group* occurs in the current database.

ALL SERVER: Applies the scope of a DDL trigger to the current server. If specified, the trigger fires whenever *event_type* or *event_group* occurs anywhere in the current server.

WITH ENCRYPTION: Encrypts the text of the CREATE TRIGGER statement. Using WITH ENCRYPTION prevents the trigger from being published as part of SQL Server replication. WITH ENCRYPTION cannot be specified for CLR triggers.

EXECUTE AS: Specifies the security context under which the trigger is executed. Enables you to control which user account the instance of SQL Server uses to validate permissions on any database objects that are referenced by the trigger.

AFTER:

- Specifies that the DML trigger is fired only when all operations specified in the triggering SQL statement have executed successfully. All referential cascade actions and constraint checks also must succeed before this trigger fires.
- AFTER is the default when FOR is the only keyword specified.
- AFTER triggers cannot be defined on views.

INSTEAD OF:

- Specifies that the DML trigger is executed *instead of* the triggering SQL statement, therefore, overriding the actions of the triggering statements. INSTEAD OF cannot be specified for DDL triggers.
- At most, one INSTEAD OF trigger per INSERT, UPDATE, or DELETE statement can be defined on a table or view. However, you can define views on views where each view has its own INSTEAD OF trigger.

- INSTEAD OF triggers are not allowed on updatable views that use WITH CHECK OPTION. SQL Server raises an error when an INSTEAD OF trigger is added to an updatable view WITH CHECK OPTION specified. The user must remove that option by using ALTER VIEW before defining the INSTEAD OF trigger.

{ [DELETE] [,] [INSERT] [,] [UPDATE] }:

- Specifies the data modification statements that activate the DML trigger when it is tried against this table or view. At least one option must be specified. Any combination of these options in any order is allowed in the trigger definition.
- For INSTEAD OF triggers, the DELETE option is not allowed on tables that have a referential relationship specifying a cascade action ON DELETE. Similarly, the UPDATE option is not allowed on tables that have a referential relationship specifying a cascade action ON UPDATE.

event_type: Is the name of a Transact-SQL language event that, after execution, causes a DDL trigger to fire. Events that are valid for use in DDL triggers are listed in DDL Events for Use with DDL Triggers.

event_group:

- Is the name of a predefined grouping of Transact-SQL language events. The DDL trigger fires after execution of any Transact-SQL language event that belongs to *event_group*. Event groups that are valid for use in DDL triggers are listed in Event Groups for Use with DDL Triggers.
- After the CREATE TRIGGER has finished running, *event_group* also acts as a macro by adding the event types it covers to the **sys.trigger_events** catalog view.

WITH APPEND:

- Specifies that an additional trigger of an existing type should be added. Use of this optional clause is required only when the compatibility level is 65 or lower. If the compatibility level is 70 or higher, the WITH APPEND clause is not required to add an additional trigger of an existing type. This is the default behavior of CREATE TRIGGER with the compatibility level setting of 70 or higher.
- WITH APPEND cannot be used with INSTEAD OF triggers or if AFTER trigger is explicitly stated. WITH APPEND can be used only when FOR is specified, without INSTEAD OF or AFTER, for backward compatibility reasons. WITH APPEND cannot be specified if EXTERNAL NAME is specified (that is, if the trigger is a CLR trigger).

NOT FOR REPLICATION: Indicates that the trigger should not be executed when a replication agent modifies the table that is involved in the trigger. For more information, see Controlling Constraints, Identities, and Triggers with NOT FOR REPLICATION.

sql_statement:

- Is the trigger conditions and actions. Trigger conditions specify additional criteria that determine whether the tried DML or DDL statements cause the trigger actions to be performed.
- The trigger actions specified in the Transact-SQL statements go into effect when the DML or DDL operation is tried.

- Triggers can include any number and kind of Transact-SQL statements, with exceptions. For more information, see Remarks. A trigger is designed to check or change data based on a data modification or definition statement; it should not return data to the user. The Transact-SQL statements in a trigger frequently include control-of-flow language.
- DML triggers use the **deleted** and **inserted** logical (conceptual) tables. They are structurally similar to the table on which the trigger is defined, that is, the table on which the user action is tried. The **deleted** and **inserted** tables hold the old values or new values of the rows that may be changed by the user action. For example, to retrieve all values in the deleted table, use:

```
SELECT *
FROM deleted
```

- DDL triggers capture information about the triggering event by using the EVENTDATA (Transact-SQL) function. For more information, see Using the EVENTDATA Function.
- In a DELETE, INSERT, or UPDATE trigger, SQL Server does not allow **text**, **ntext**, or **image** column references in the **inserted** and **deleted** tables if the compatibility level is set to 70. The **text**, **ntext**, and **image** values in the **inserted** and **deleted** tables cannot be accessed. To retrieve the new value in either an INSERT or UPDATE trigger, join the **inserted** table with the original update table. When the compatibility level is 65 or lower, null values are returned for **inserted** or **deleted text**, **ntext**, or **image** columns that allow null values; zero-length strings are returned if the columns are not nullable.
- If the compatibility level is 80 or higher, SQL Server allows for the update of **text**, **ntext**, or **image** columns through the INSTEAD OF trigger on tables or views.

`< method_specifier >`: For a CLR trigger, specifies the method of an assembly to bind with the trigger. The method must take no arguments and return void. *class_name* must be a valid SQL Server identifier and must exist as a class in the assembly with assembly visibility. If the class has a namespace-qualified name that uses '.' to separate namespace parts, the class name must be delimited by using [ ] or " " delimiters. The class cannot be a nested class.

Examples:

Using a DML trigger with a reminder message

The following DML trigger prints a message to the client when anyone tries to add or change data in the Customer table.

```
USE AdventureWorks
IF OBJECT_ID ('Sales.reminder1', 'TR') IS NOT NULL
   DROP TRIGGER Sales.reminder1
GO
CREATE TRIGGER reminder1
ON Sales.Customer
AFTER INSERT, UPDATE
AS RAISERROR ('Notify Customer Relations', 16, 10)
GO
```

AFTER Triggers

- The trigger executes after the statement that triggered it completes. If the statement fails with an error, such as a constraint violation or syntax error, then the trigger is not executed.
- AFTER triggers are executed after the action of the INSERT, UPDATE or DELETE statement is performed.
- AFTER triggers can be specified only on tables, not on views.
- A table can have several AFTER triggers for each triggering action.

Example:

```
CREATE TRIGGER LowCredit ON Purchasing.PurchaseOrderHeader
AFTER INSERT
AS
DECLARE @creditrating tinyint,
   @vendorid int
SELECT @creditrating = v.CreditRating, @vendorid = p.VendorID
FROM Purchasing.PurchaseOrderHeader p INNER JOIN inserted i ON
 p.PurchaseOrderID =   i.PurchaseOrderID JOIN
 Purchasing.Vendor v on v.VendorID = i.VendorID
IF @creditrating = 5
BEGIN
   RAISERROR ('This vendor''s credit rating is too low to accept new
      purchase orders.', 16, 1)
ROLLBACK TRANSACTION
END
```

INSTEAD OF Triggers

INSTEAD OF triggers can be used to perform to enhance integrity checks on the data values supplied in INSERT and UPDATE statements. INSTEAD OF triggers can be used to perform enhance integrity checks on the data values supplied in INSERT and UPDATE statements.

- INSTEAD OF triggers are executed in place of the usual triggering action.
- INSTEAD OF triggers can also be defined on views with one or more base tables, where they can extend the types of updates a view can support.
- One INSTEAD OF trigger per INSERT, UPDATE or DELETE statement can be defined on a table or view.

Examples:

```
CREATE TRIGGER [delEmployee] ON [HumanResources].[Employee]
INSTEAD OF DELETE NOT FOR REPLICATION AS
BEGIN
      SET NOCOUNT ON;
      DECLARE @DeleteCount int;
      Select @DeleteCount =COUNT(*) FROM deleted;
      IF @DeleteCount > 0
      BEGIN
            RAISERROR
                  (N'Employees cannot be deleted. They can only be
 marked as not  current.',--Message
                  10,--Severity.
                  1);--State.
            --Rollback any active or uncommitable transactions
            IF @@TRANCOUNT > 0
            BEGIN
                  ROLLBACK TRANSACTION;
            END
      END;
END;
```

Creating CLR Trigger

Microsoft SQL Server 2005 provides the ability to create a database object inside SQL Server that is programmed in an assembly created in the .NET Framework common language runtime (CLR). Database objects that can leverage the rich programming model provided by the CLR include DML triggers, DDL triggers, stored procedures, functions, aggregate functions, and types.

Creating a CLR trigger (DML or DDL) in SQL Server involves the following steps:

- Define the trigger as a class in a .NET language. For more information about how to program triggers in the CLR, see CLR Triggers. Then, compile the class to build an assembly in the .NET Framework using the appropriate language compiler.
- Register the assembly in SQL Server using the CREATE ASSEMBLY statement.
- Create the trigger that references the registered assembly.

Triggers written in a CLR language differ from other CLR integration objects in several significant ways. CLR triggers can:

- Reference data in the **INSERTED** and **DELETED** tables
- Determine which columns have been modified as a result of an **UPDATE** operation
- Access information about database objects affected by the execution of DDL statements.

Sample CLR Trigger:

In this example, consider the scenario in which you let the user choose any ID they want, but you want to know the users that specifically entered an e-mail address as an ID. The following trigger would detect that information and log it to an audit table.

```
using System;
using System.Data;
using System.Data.Sql;
using Microsoft.SqlServer.Server;
using System.Data.SqlClient;
using System.Data.SqlTypes;
using System.Xml;
using System.Text.RegularExpressions;

public class CLRTriggers
{
   [SqlTrigger(Name = @"EmailAudit", Target = "[dbo].[Users]", Event =
"FOR INSERT, UPDATE, DELETE")]
   public static void EmailAudit()
   {
      string userName;
      string realName;
      SqlCommand command;
      SqlTriggerContext triggContext = SqlContext.TriggerContext;
      SqlPipe pipe = SqlContext.Pipe;
      SqlDataReader reader;

      switch (triggContext.TriggerAction)
      {
         case TriggerAction.Insert:
         // Retrieve the connection that the trigger is using
         using (SqlConnection connection
            = new SqlConnection(@"context connection=true"))
         {
            connection.Open();
            command = new SqlCommand(@"SELECT * FROM INSERTED;",
               connection);
            reader = command.ExecuteReader();
            reader.Read();
            userName = (string)reader[0];
            realName = (string)reader[1];
            reader.Close();

            if (IsValidEMailAddress(userName))
            {
               command = new SqlCommand(
                  @"INSERT [dbo].[UserNameAudit] VALUES ('"
                  + userName + @"', '" + realName + @"');",
                  connection);
               pipe.Send(command.CommandText);
               command.ExecuteNonQuery();
               pipe.Send("You inserted: " + userName);
            }
         }
         break;

         case TriggerAction.Update:
         // Retrieve the connection that the trigger is using
         using (SqlConnection connection
            = new SqlConnection(@"context connection=true"))
         {
            connection.Open();
            command = new SqlCommand(@"SELECT * FROM INSERTED;",
               connection);
            reader = command.ExecuteReader();
            reader.Read();

            userName = (string)reader[0];
            realName = (string)reader[1];

            pipe.Send(@"You updated: '" + userName + @"' - '"
               + realName + @"'");

            for (int columnNumber = 0; columnNumber <
triggContext.ColumnCount; columnNumber++)
            {
               pipe.Send("Updated column "
                  + reader.GetName(columnNumber) + "? "
                  +
triggContext.IsUpdatedColumn(columnNumber).ToString());
            }

            reader.Close();
         }

         break;

         case TriggerAction.Delete:
            using (SqlConnection connection
               = new SqlConnection(@"context connection=true"))
               {
                  connection.Open();
                  command = new SqlCommand(@"SELECT * FROM DELETED;",
                     connection);
                  reader = command.ExecuteReader();

                  if (reader.HasRows)
                  {
                     pipe.Send(@"You deleted the following rows:");
                     while (reader.Read())
                     {
                        pipe.Send(@"'" + reader.GetString(0)
                        + @"', '" + reader.GetString(1) + @"'");
                     }

                     reader.Close();

                    //alternately, to just send a tabular resultset back:
                     //pipe.ExecuteAndSend(command);
                  }
                  else
                  {
                     pipe.Send("No rows affected.");
                  }
               }

               break;
            }
        }

     public static bool IsValidEMailAddress(string email)
     {
         return Regex.IsMatch(email, @"^([\w-]+\.)*?[\w-]+@[\w-]+\.([\w-
]+\.)*?[\w]+$");
     }
}
```

Visual Basic:

```
Imports System
Imports System.Data
Imports System.Data.Sql
Imports System.Data.SqlTypes
Imports Microsoft.SqlServer.Server
Imports System.Data.SqlClient
Imports System.Text.RegularExpressions

'The Partial modifier is only required on one class definition per
project.
Partial Public Class CLRTriggers

    <SqlTrigger(Name:="EmailAudit", Target:="[dbo].[Users]", Event:="FOR
INSERT, UPDATE, DELETE")> _
    Public Shared Sub EmailAudit()
```

```
        Dim userName As String
        Dim realName As String
        Dim command As SqlCommand
        Dim triggContext As SqlTriggerContext
        Dim pipe As SqlPipe
        Dim reader As SqlDataReader

        triggContext = SqlContext.TriggerContext
        pipe = SqlContext.Pipe

        Select Case triggContext.TriggerAction
           Case TriggerAction.Insert
              Using connection As New SqlConnection("context 
connection=true")
                 connection.Open()
                 command = new SqlCommand("SELECT * FROM INSERTED;", 
connection)

                 reader = command.ExecuteReader()
                 reader.Read()

                 userName = CType(reader(0), String)
                 realName = CType(reader(1), String)

                 reader.Close()

                 If IsValidEmailAddress(userName) Then
                     command = New SqlCommand("INSERT 
[dbo].[UserNameAudit] VALUES ('" & _
                       userName & "', '" & realName & "');", connection)

                    pipe.Send(command.CommandText)
                    command.ExecuteNonQuery()
                    pipe.Send("You inserted: " & userName)

                 End If
              End Using

           Case TriggerAction.Update
              Using connection As New SqlConnection("context 
connection=true")
                 connection.Open()
                 command = new SqlCommand("SELECT * FROM INSERTED;", 
connection)
```

```
                 reader = command.ExecuteReader()
                 reader.Read()

                 userName = CType(reader(0), String)
                 realName = CType(reader(1), String)

                 pipe.Send("You updated: " & userName & " - " &
realName)

                 Dim columnNumber As Integer

                 For columnNumber=0 To triggContext.ColumnCount-1

                    pipe.Send("Updated column " &
reader.GetName(columnNumber) & _
                      "? " &
triggContext.IsUpdatedColumn(columnNumber).ToString() )

                 Next

                 reader.Close()
              End Using

           Case TriggerAction.Delete
              Using connection As New SqlConnection("context
connection=true")
                 connection.Open()
                 command = new SqlCommand("SELECT * FROM DELETED;",
connection)

                 reader = command.ExecuteReader()

                 If reader.HasRows Then
                    pipe.Send("You deleted the following rows:")

                    While reader.Read()

                       pipe.Send( reader.GetString(0) & ", " &
reader.GetString(1) )

                    End While

                    reader.Close()

                    ' Alternately, just send a tabular resultset back:
```

```
                    ' pipe.ExecuteAndSend(command)

                 Else
                   pipe.Send("No rows affected.")
                 End If

              End Using
        End Select
    End Sub

    Public Shared Function IsValidEMailAddress(emailAddress As String)
As Boolean

       return Regex.IsMatch(emailAddress, "^([\w-]+\.)*?[\w-]+@[\w-
]+\.([\w-]+\.)*?[\w]+$")
    End Function
End Class
```

Assuming two tables exist with the following definitions:

```
CREATE TABLE Users
(
    UserName nvarchar(200) NOT NULL,
    RealName nvarchar(200) NOT NULL
);
GO CREATE TABLE UserNameAudit
(
    UserName nvarchar(200) NOT NULL,
    RealName nvarchar(200) NOT NULL
)
```

The Transact-SQL statement that creates the trigger in SQL Server is as follows, and assumes assembly **SQLCLRTest** is already registered in the current SQL Server database.

```
CREATE TRIGGER EmailAudit
ON Users
FOR INSERT
AS
EXTERNAL NAME SQLCLRTest.CLRTriggers.EmailAudit
```

DDL Triggers

DDL triggers are used when:

- They can prevent certain changes to your database schema.
- They can perform administrative tasks and enforce business rules that affect databases.
- They can record changes or events in the database schema.

Example:

Using a database-scoped DDL trigger

The following example uses a DDL trigger to prevent any synonym in a database from being or dropped.

```
USE AdventureWorks
IF EXISTS (SELECT * FROM sys.triggers
    WHERE parent_class = 0 AND name = 'safety')
DROP TRIGGER safety
ON DATABASE
GO
CREATE TRIGGER safety
ON DATABASE
FOR DROP_SYNONYM
AS
   PRINT 'You must disable Trigger "safety" to drop synonyms!'
   ROLLBACK
GO
DROP TRIGGER safety
ON DATABASE
GO
```

OUTPUT Clause

Returns information from, or expressions based on, each row affected by an INSERT, UPDATE, or DELETE statement. These results can be returned to the processing application for use in such things as confirmation messages, archiving, and other such application requirements. Alternatively, results can be inserted into a table or table variable.

Specify the OUTPUT Clause (Transact-SQL) with an INSERT, UPDATE or DELETE statement inside the body of a DML trigger or DML statements to return rows affected a modification.

Example:

You can use OUTPUT in applications that use tables as queues, or to hold intermediate result sets. That is, the application is constantly adding or removing rows from the table. The following example uses the OUTPUT clause in a DELETE statement to return the deleted row to the calling application.

```
USE AdventureWorks;
GO
DELETE TOP(1) dbo.DatabaseLog WITH (READPAST)
OUTPUT deleted.*
WHERE DatabaseLogID = 7;
GO
```

Summary

- ❑ A trigger is a special type of stored procedure that automatically takes effect when a language event executes.
- ❑ SQL Server uses DDL and DML Triggers.

Session 31: Indexes

Learning Objectives

After completing this session, you will be able to:

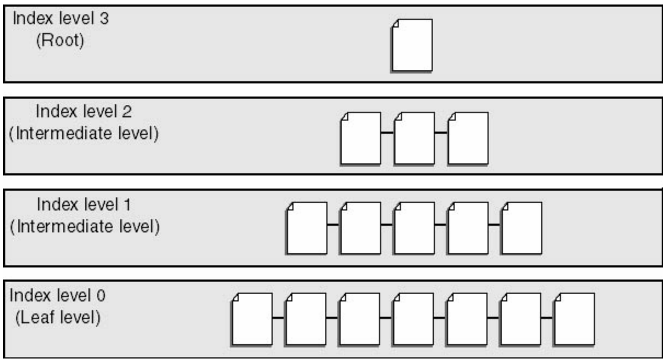
- ❑ Explain indexes
- ❑ Define clustered and non-clustered indexes
- ❑ Implement Database Engine Tuning Advisor

Index

Indexes in databases are similar to indexes in books. In a book, an index allows to find information quickly without reading the entire book. An index in a database allows the database program to find data in a table without scanning the entire table. An index is a structure that orders the values of one or more columns in a database table. The index provides pointers to the data values stored in specified columns of the table, and then orders those pointers according to the sort order specified. It is a list of values in a table with the storage locations of rows in the table that contain each value.

Indexes can be created on either a single column or a combination of columns in a table and are implemented in the form of B-trees. An index contains an entry with one or more columns (the search key) from each row in a table. A B-tree is sorted on the search key, and can be searched efficiently on any leading subset of the search key.

A B-tree provides fast access to data by searching on a key value of the index. B-trees cluster records with similar keys. The B stands for balanced, and balancing the tree is a core feature of a B-tree's usefulness. The trees are managed, and branches are grafted as necessary, so that navigating down the tree to find a value and locate a specific record takes only a few page accesses. Because the trees are balanced, finding any record requires about the same amount of resources, and retrieval speed is consistent because the index has the same depth throughout. An index consists of a tree with a root from which the navigation begins possible intermediate index levels, and bottom-level leaf pages.



A B-tree for a SQL Server index

Each page in an index holds a page header followed by index rows. Each index row contains a key value and a pointer to either a lower-level page or a data row. Each page in an index is called an index node. The top node of the B-tree is called the root node. The bottom layer of nodes in the index are called the leaf nodes. The pages in each level of the index are linked together in a doubly-linked list.

For any query, the Database searches the index to find a particular value and then follows the pointer to the row containing that value. Multiple-column indexes enable to distinguish between rows in which one column may have the same value. Indexes are also helpful to search or sort by two or more columns at a time.

Indexes allow data to be organized in a way that allows optimum performance when it is accessed or modified. Indexes can be used to quickly find data rows that satisfy conditions in WHERE clauses, to find matching rows in JOIN clauses, or to efficiently maintain uniqueness of key columns during INSERT and UPDATE operations. In some cases, indexes can be used to help SQL Server sort, aggregate, or group data or to find the first few rows as indicated in a TOP clause.

The two types of SQL Server indexes are:

- ❑ Clustered
- ❑ Nonclustered

Clustered Indexes

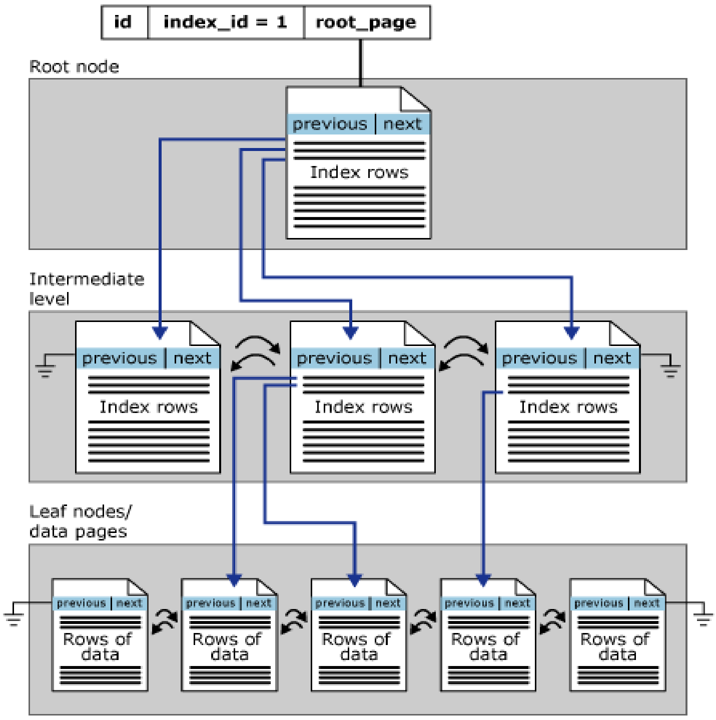
A clustered index determines the physical order of data in a table. A clustered index is analogous to a telephone directory, which arranges data by last name. Because the clustered index dictates the physical storage order of the data in the table, a table can contain only one clustered index. However, the index can comprise multiple columns (a composite index).

The indexes are organized as B-trees. In a clustered index, the data pages make up the leaf nodes. That is, data itself is part of the index. Any index levels between the root and the leaves are collectively known as intermediate levels. When the index is traversed to the leaf level, the data itself has been retrieved, not simply pointed to.

The only time the data rows in a table are stored in sorted order is when the table contains a clustered index. If a table has no clustered index, its data rows are stored in a heap.

PRIMARY KEY constraints create clustered indexes automatically if no clustered index already exists on the table and a nonclustered index is not specified when creating the PRIMARY KEY constraint.

The clustered index structure is shown in the following graphic:



Clustered Indexes - Usage

A clustered index is efficient on columns that are often searched for ranges of values. After the row with the first value is found using the clustered index, rows with subsequent indexed values are guaranteed to be physically adjacent. Clustered indexes are also efficient for finding a specific row when the indexed value is unique.

A clustered index can be used for:

- Columns that contain a large number of distinct values.
- Queries that return a range of values using operators such as BETWEEN, >, >=, <, and <=.
- Columns that are accessed sequentially.
- Queries that return large result sets.
- Columns that are frequently accessed by queries involving join or GROUP BY clauses; typically these are foreign key columns. An index on the column(s) specified in the ORDER BY or GROUP BY clause eliminates the need for SQL Server to sort the data because the rows are already sorted. This improves query performance.
- OLTP-type applications where very fast single row lookup is required, typically by means of the primary key. Create a clustered index on the primary key.

It is important to define the clustered index key with as few columns as possible. If a large clustered index key is defined, any nonclustered indexes that are defined on the same table will be significantly larger because the nonclustered index entries contain the clustering key. Clustered indexes can be defined on columns with high density, that is, uniqueness.

Clustered indexes are not a good choice for:

- **Columns that undergo frequent changes:** This results in the entire row moving (because SQL Server must keep the data values of a row in physical order). This is an important consideration in high-volume transaction processing systems where data tends to be volatile.
- **Wide keys:** The key values from the clustered index are used by all nonclustered indexes as lookup keys and therefore are stored in each nonclustered index leaf entry

Non Clustered Indexes

Nonclustered indexes have the same B-tree structure as clustered indexes, with two significant differences:

- The data rows are not sorted and stored in order based on their nonclustered keys.
- The leaf layer of a nonclustered index does not consist of the data pages.

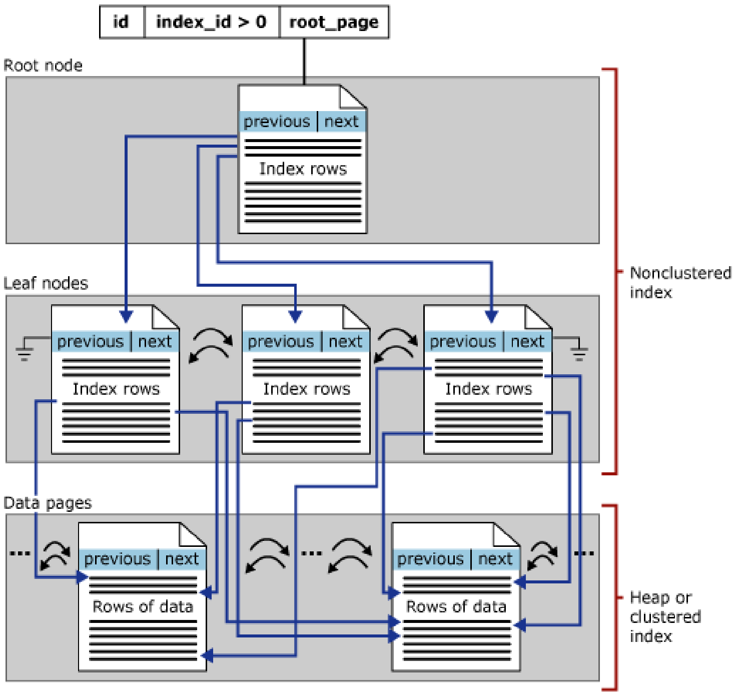
Instead, the leaf nodes contain index rows. Each index row contains the nonclustered key value and one or more row locators that point to the data row (or rows if the index is not unique) having the key value.

The items in the index are stored in the order of the index key values, but the information in the table is stored in a different order (which can be dictated by a clustered index). If no clustered index is created on the table, the rows are not guaranteed to be in any particular order.

The presence or absence of a nonclustered index doesn't affect how the data pages are organized, so there is no restriction to have only one nonclustered index per table, as is the case with clustered indexes. Each table can include as many as 249 nonclustered indexes.

The pointer from an index row in a nonclustered index to a data row is called a row locator. The structure of the row locator depends on whether the data pages are stored in a heap or are clustered. For a heap, a row locator is a pointer to the row. For a table with a clustered index, the row locator is the clustered index key.

The Nonclustered index structure is shown in the following graphic:



Nonclustered indexes can be used for:

- Columns that contain a large number of distinct values, such as a combination of last name and first name (if a clustered index is used for other columns). If there are very few distinct values, such as only 1 and 0, most queries will not use the index because a table scan is usually more efficient.
- Queries that do not return large result sets.
- Columns frequently involved in search conditions of a query (WHERE clause) that return exact matches.
- Decision-support-system applications for which joins and grouping are frequently required. Create multiple nonclustered indexes on columns involved in join and grouping operations, and a clustered index on any foreign key columns.
- Covering all columns from one table in a given query. This eliminates accessing the table or clustered index altogether.

CREATE INDEX

Creates a relational index on a specified table or view, or an XML index on a specified table. An index can be created before there is data in the table. Indexes can be created on tables or views in another database by specifying a qualified database name.

Syntax:

```
Create Relational Index
CREATE [ UNIQUE ] [ CLUSTERED | NONCLUSTERED ] INDEX index_name
    ON <object> ( column [ ASC | DESC ] [ ,...n ] )
    [ INCLUDE ( column_name [ ,...n ] ) ]
    [ WITH ( <relational_index_option> [ ,...n ] ) ]
    [ ON { partition_scheme_name ( column_name )
         | filegroup_name
         | default
         }
    ]
[ ; ]

<object> ::=
{
    [ database_name. [ schema_name ] . | schema_name. ]
        table_or_view_name
}

<relational_index_option> ::=
{
    PAD_INDEX  = { ON | OFF }
  | FILLFACTOR = fillfactor
  | SORT_IN_TEMPDB = { ON | OFF }
  | IGNORE_DUP_KEY = { ON | OFF }
  | STATISTICS_NORECOMPUTE = { ON | OFF }
  | DROP_EXISTING = { ON | OFF }
  | ONLINE = { ON | OFF }
  | ALLOW_ROW_LOCKS = { ON | OFF }
  | ALLOW_PAGE_LOCKS = { ON | OFF }
  | MAXDOP = max_degree_of_parallelism
}

Create XML Index
CREATE [ PRIMARY ] XML INDEX index_name
    ON <object> ( xml_column_name )
    [ USING XML INDEX xml_index_name
        [ FOR { VALUE | PATH | PROPERTY } ]
    [ WITH ( <xml_index_option> [ ,...n ] ) ]
[ ; ]

<object> ::=
{
    [ database_name. [ schema_name ] . | schema_name. ]
        table_name
}

<xml_index_option> ::=
{
    PAD_INDEX  = { ON | OFF }
  | FILLFACTOR = fillfactor
  | SORT_IN_TEMPDB = { ON | OFF }
  | STATISTICS_NORECOMPUTE = { ON | OFF }
  | DROP_EXISTING = { ON | OFF }
  | ALLOW_ROW_LOCKS = { ON | OFF }
  | ALLOW_PAGE_LOCKS = { ON | OFF }
  | MAXDOP = max_degree_of_parallelism
}

Backward Compatible Relational Index
Important   The backward compatible relational index syntax structure
will be removed in a future version of SQL Server. Avoid using this
syntax structure in new development work, and plan to modify
applications that currently use the feature. Use the syntax structure
specified in <relational_index_option> instead.

CREATE [ UNIQUE ] [ CLUSTERED | NONCLUSTERED ] INDEX index_name
    ON <object> ( column_name [ ASC | DESC ] [ ,...n ] )
    [ WITH <backward_compatible_index_option> [ ,...n ] ]
    [ ON { filegroup_name | "default" } ]

<object> ::=
{
    [ database_name. [ owner_name ] . | owner_name. ]
        table_or_view_name
}

<backward_compatible_index_option> ::=
{
    PAD_INDEX
  | FILLFACTOR = fillfactor
  | SORT_IN_TEMPDB
  | IGNORE_DUP_KEY
  | STATISTICS_NORECOMPUTE
  | DROP_EXISTING
}
```

Arguments:

UNIQUE:

- Creates a unique index on a table or view. A unique index is one in which no two rows are permitted to have the same index key value. A clustered index on a view must be unique.
- The SQL Server 2005 Database Engine does not allow creating a unique index on columns that already include duplicate values, whether or not IGNORE_DUP_KEY is set to ON. If this is tried, the Database Engine displays an error message. Duplicate

values must be removed before a unique index can be created on the column or columns. Columns that are used in a unique index should be set to NOT NULL, because multiple null values are considered duplicates when a unique index is created.

CLUSTERED:

- Creates an index in which the logical order of the key values determines the physical order of the corresponding rows in a table. The bottom, or leaf, level of the clustered index contains the actual data rows of the table. A table or view is allowed one clustered index at a time. For more information, see Clustered Index Structures.
- A view with a unique clustered index is called an indexed view. Creating a unique clustered index on a view physically materializes the view. A unique clustered index must be created on a view before any other indexes can be defined on the same view. For more information, see Designing Indexed Views.
- Create the clustered index before creating any nonclustered indexes. Existing nonclustered indexes on tables are rebuilt when a clustered index is created.
- If CLUSTERED is not specified, a nonclustered index is created. <u>NONCLUSTERED</u>
- Creates an index that specifies the logical ordering of a table. With a nonclustered index, the physical order of the data rows is independent of their indexed order. For more information, see Nonclustered Index Structures.
- Each table can have up to 249 nonclustered indexes, regardless of how the indexes are created: either implicitly with PRIMARY KEY and UNIQUE constraints, or explicitly with CREATE INDEX.
- For indexed views, nonclustered indexes can be created only on a view that has a unique clustered index already defined.
- The default is NONCLUSTERED.

index_name:

- Is the name of the index. Index names must be unique within a table or view but do not have to be unique within a database. Index names must follow the rules of identifiers.
- Primary XML index names cannot start with the following characters: #, ##, @, or @@.

column:

- Is the column or columns on which the index is based. Specify two or more column names to create a composite index on the combined values in the specified columns. List the columns to be included in the composite index, in sort-priority order, inside the parentheses after *table_or_view_name*.
- Up to 16 columns can be combined into a single composite index key. All the columns in a composite index key must be in the same table or view. The maximum allowable size of the combined index values is 900 bytes. For more information about variable type columns in composite indexes, see the Remarks section.
- Columns that are of the large object (LOB) data types **ntext**, **text**, **varchar(max)**, **nvarchar(max)**, **varbinary(max)**, **xml**, or **image** cannot be specified as key columns for an index. Also, a view definition cannot include **ntext**, **text**, or **image** columns, even if they are not referenced in the CREATE INDEX statement.

- You can create indexes on CLR user-defined type columns if the type supports binary ordering. You can also create indexes on computed columns that are defined as method invocations off a user-defined type column, as long as the methods are marked deterministic and do not perform data access operations. For more information about indexing CLR user-defined type columns, see CLR User-defined Types.

[ASC | DESC]: Determines the ascending or descending sort direction for the particular index column. The default is ASC.

INCLUDE (column [,... n]):

- Specifies the nonkey columns to be added to the leaf level of the nonclustered index. The nonclustered index can be unique or nonunique.
- The maximum number of included nonkey columns is 1,023 columns; the minimum number is 1 column.
- Column names cannot be repeated in the INCLUDE list and cannot be used simultaneously as both key and nonkey columns. For more information, see Index with Included Columns.
- All data types are allowed except **text**, **ntext**, and **image**. The index must be created or rebuilt offline (ONLINE = OFF) if any one of the specified nonkey columns are **varchar(max)**, **nvarchar(max)**, or **varbinary(max)** data types.
- Computed columns that are deterministic and either precise or imprecise can be included columns. Computed columns derived from **image**, **ntext**, **text**, **varchar(max)**, **nvarchar(max)**, **varbinary(max)**, and **xml** data types can be included nonkey columns as long as the computed column data types is allowable as an included column. For more information, see Creating Indexes on Computed Columns.

ON partition_scheme_name (column_name):

- Specifies the partition scheme that defines the filegroups onto which the partitions of a partitioned index will be mapped. The partition scheme must exist within the database by executing either **CREATE PARTITION SCHEME** or **ALTER PARTITION SCHEME.** column_name specifies the column against which a partitioned index will be partitioned. This column must match the data type, length, and precision of the argument of the partition function that *partition_scheme_name* is using. *column_name* is not restricted to the columns in the index definition. Any column in the base table can be specified, except when partitioning a UNIQUE index, *column_name* must be chosen from among those used as the unique key. This restriction allows the Database Engine to verify uniqueness of key values within a single partition only.
- If *partition_scheme_name* or *filegroup* is not specified and the table is partitioned, the index is placed in the same partition scheme, using the same partitioning column, as the underlying table.
- You cannot specify a partitioning scheme on an XML index. If the base table is partitioned, the XML index uses the same partition scheme as the table.
- For more information about partitioning indexes, see Special Guidelines for Partitioned Indexes.

ON filegroup_name: Creates the specified index on the specified filegroup. If no location is specified and the table or view is not partitioned, the index uses the same filegroup as the underlying table or view. The filegroup must already exist. XML indexes use the same filegroup as the table.

ON "default":

- Creates the specified index on the default filegroup.
- The term default, in this context, is not a keyword. It is an identifier for the default filegroup and must be delimited, as in ON **"**default**"** or ON **[**default**]**. If "default" is specified, the QUOTED_IDENTIFIER option must be ON for the current session. This is the default setting.

[PRIMARY] XML: Creates an XML index on the specified **xml** column. When PRIMARY is specified, a clustered index is created with the clustered key formed from the clustering key of the user table and an XML node identifier. Each table can have up to 249 XML indexes. Note the following when you create an XML index:

- A clustered index must exist on the primary key of the user table.
- The clustering key of the user table is limited to 15 columns.
- Each **xml** column in a table can have one primary XML index and multiple secondary XML indexes.
- A primary XML index on an **xml** column must exist before a secondary XML index can be created on the column.
- An XML index can only be created on a single **xml** column. You cannot create an XML index on a non-**xml** column, nor can you create a relational index on an **xml** column.
- You cannot create an XML index, either primary or secondary, on an **xml** column in a view, on a table-valued variable with **xml** columns, or **xml** type variables.
- You cannot create a primary XML index on a computed **xml** column.
- The SET option settings must be the same as those required for indexed views and computed column indexes. Specifically, the option ARITHABORT must be set to ON when an XML index is created and when inserting, deleting, or updating values in the **xml** column. For more information, see SET Options That Affect Results.

xml_column_name: Is the **xml** column on which the index is based. Only one **xml** column can be specified in a single XML index definition; however, multiple secondary XML indexes can be created on an **xml** column.

USING XML INDEX *xml_index_name*: Specifies the primary XML index to use in creating a secondary XML index.

| FOR { VALUE | PATH PROPERTY }: Specifies the type of secondary XML index.

VALUE: Creates a secondary XML index on columns where key columns are (node value and path) of the primary XML index.

PATH : Creates a secondary XML index on columns built on path values and node values in the primary XML index. In the PATH secondary index, the path and node values are key columns that allow efficient seeks when searching for paths.

PROPERTY: Creates a secondary XML index on columns (PK, path and node value) of the primary XML index where PK is the primary key of the base table.

<object>::=: Is the fully qualified or nonfully qualified object to be indexed.

database_name: Is the name of the database.

schema_name: Is the name of the schema to which the table or view belongs.

table_or_view_name:

- Is the name of the table or view to be indexed.
- The view must be defined with SCHEMABINDING to create an index on it. A unique clustered index must be created on a view before any nonclustered index is created. For more information about indexed views, see the Remarks section.

<relational_index_option>:: = Specifies the options to use when you create the index

PAD_INDEX = { ON | OFF }: Specifies index padding. The default is OFF.

ON:

- The percentage of free space that is specified by *fillfactor* is applied to the intermediate-level pages of the index.
- OFF or *fillfactor* is not specified
- The intermediate-level pages are filled to near capacity, leaving sufficient space for at least one row of the maximum size the index can have, considering the set of keys on the intermediate pages.
- The PAD_INDEX option is useful only when FILLFACTOR is specified, because PAD_INDEX uses the percentage specified by FILLFACTOR. If the percentage specified for FILLFACTOR is not large enough to allow for one row, the Database Engine internally overrides the percentage to allow for the minimum. The number of rows on an intermediate index page is never less than two, regardless of how low the value of *fillfactor*.
- In backward compatible syntax, WITH PAD_INDEX is equivalent to WITH PAD_INDEX = ON.

FILLFACTOR = fillfactor*:* Specifies a percentage that indicates how full the Database Engine should make the leaf level of each index page during index creation or rebuild. *fillfactor* must be an integer value from 1 to 100. The default is 0. If *fillfactor* is 100 or 0, the Database Engine creates indexes with leaf pages filled to capacity, but some space remains within the upper level of the index tree to allow for at least one additional index row.

The FILLFACTOR setting applies only when the index is created or rebuilt. The Database Engine does not dynamically keep the specified percentage of empty space in the pages.

SORT_IN_TEMPDB = { ON | OFF }: Specifies whether to store temporary sort results in **tempdb**. The default is OFF.

ON: The intermediate sort results that are used to build the index are stored in **tempdb**. This may reduce the time required to create an index if **tempdb** is on a different set of disks than the user database. However, this increases the amount of disk space that is used during the index build.

OFF:

- The intermediate sort results are stored in the same database as the index.
- In addition to the space required in the user database to create the index, **tempdb** must have about the same amount of additional space to hold the intermediate sort results. For more information, see tempdb and Index Creation.
- In backward compatible syntax, WITH SORT_IN_TEMPDB is equivalent to WITH SORT_IN_TEMPDB = ON.

IGNORE_DUP_KEY = { ON | OFF }: Specifies the error response to duplicate key values in a multiple-row insert operation on a unique clustered or unique nonclustered index. The default is OFF.

ON: A warning message is issued and only the rows violating the unique index fail.

OFF: An error message is issued and the entire INSERT transaction is rolled back.

The IGNORE_DUP_KEY setting applies only to insert operations that occur after the index is created or rebuilt. The setting has no affect during the index operation.

IGNORE_DUP_KEY cannot be set to ON for XML indexes and indexes created on a view.

In backward compatible syntax, WITH IGNORE_DUP_KEY is equivalent to WITH IGNORE_DUP_KEY = ON.

STATISTICS_NORECOMPUTE = { ON | OFF } : Specifies whether distribution statistics are recomputed. The default is OFF.

ON: Out-of-date statistics are not automatically recomputed.

OFF: Automatic statistics updating are enabled.

To restore automatic statistics updating, set the STATISTICS_NORECOMPUTE to OFF, or execute UPDATE STATISTICS without the NORECOMPUTE clause.

In backward compatible syntax, WITH STATISTICS_NORECOMPUTE is equivalent to WITH STATISTICS_NORECOMPUTE = ON.

DROP_EXISTING = { ON | OFF }: Specifies that the named, preexisting clustered, nonclustered, or XML index is dropped and rebuilt. The default is OFF.

ON: The existing index is dropped and rebuilt. The index name specified must be the same as a currently existing index; however, the index definition can be modified. For example, you can specify different columns, sort order, partition scheme, or index options.

OFF: An error is displayed if the specified index name already exists.
The index type, relational or XML, cannot be changed by using DROP_EXISTING. Also, a primary XML index cannot be redefined as a secondary XML index, or vice versa.

In backward compatible syntax, WITH DROP_EXISTING is equivalent to WITH DROP_EXISTING = ON.

ONLINE = { ON | OFF }: Specifies whether underlying tables and associated indexes are available for queries and data modification during the index operation. The default is OFF.

ON: Long-term table locks are not held for the duration of the index operation. During the main phase of the index operation, only an Intent Share (IS) lock is held on the source table. This enables queries or updates to the underlying table and indexes to proceed. At the start of the operation, a Shared (S) lock is held on the source object for a very short period of time. At the end of the operation, for a short period of time, an S (Shared) lock is acquired on the source if a nonclustered index is being created; or an SCH-M (Schema Modification) lock is acquired when a clustered index is created or dropped online and when a clustered or nonclustered index is being rebuilt. ONLINE cannot be set to ON when an index is being created on a local temporary table.

OFF:

- Table locks are applied for the duration of the index operation. An offline index operation that creates, rebuilds, or drops a clustered index, or rebuilds or drops a nonclustered index, acquires a Schema modification (Sch-M) lock on the table. This prevents all user access to the underlying table for the duration of the operation. An offline index operation that creates a nonclustered index acquires a Shared (S) lock on the table. This prevents updates to the underlying table but allows read operations, such as SELECT statements.
- For more information, see How Online Index Operations Work. For more information about locks, see Lock Modes.

Indexes, including indexes on global temp tables, can be created online with the following exceptions:

- XML index
- Index on a local temp table
- Clustered index if the underlying table contains LOB data types: **image**, **ntext**, **text**, **varchar(max)**, **nvarchar(max)**, **varbinary(max)**, and **xml**.
- Nonclustered index defined with LOB data type columns.
- Initial unique clustered index on a view.

A nonclustered index can be created online if the table contains LOB data types but none of these columns are used in the index definition as either key or nonkey (included) columns.

For more information, see Performing Index Operations Online.

ALLOW_ROW_LOCKS = { ON | OFF }: Specifies whether row locks are allowed. The default is ON.

ON: Row locks are allowed when accessing the index. The Database Engine determines when row locks are used.

OFF: Row locks are not used.

ALLOW_PAGE_LOCKS = { ON | OFF }: Specifies whether page locks are allowed. The default is ON.

ON: Page locks are allowed when accessing the index. The Database Engine determines when page locks are used.

OFF: Page locks are not used.

`MAXDOP = max_degree_of_parallelism` overrides the max degree of parallelism configuration option for the duration of the index operation. Use MAXDOP to limit the number of processors used in a parallel plan execution. The maximum is 64 processors.

max_degree_of_parallelism can be 1

Suppresses parallel plan generation.

>1

Restricts the maximum number of processors used in a parallel index operation to the specified number or fewer based on the current system workload.

0 (default)

Uses the actual number of processors or fewer based on the current system workload.

For more information, see Configuring Parallel Index Operations.

Examples:

Creating a simple nonclustered index:

The following example creates a nonclustered index on the VendorID column of the Purchasing.ProductVendor table.

```
USE AdventureWorks;
GO
IF EXISTS (SELECT name FROM sys.indexes
            WHERE name = N'IX_ProductVendor_VendorID')
    DROP INDEX IX_ProductVendor_VendorID ON Purchasing.ProductVendor;
GO
CREATE INDEX IX_ProductVendor_VendorID
    ON Purchasing.ProductVendor (VendorID);
GO
```

Creating a simple nonclustered composite index:

The following example creates a nonclustered composite index on the SalesQuota and SalesYTD columns of the Sales.SalesPerson table.

```
USE AdventureWorks
GO
IF EXISTS (SELECT name FROM sys.indexes
            WHERE name = N'IX_SalesPerson_SalesQuota_SalesYTD')
    DROP INDEX IX_SalesPerson_SalesQuota_SalesYTD ON Sales.SalesPerson ;
GO
CREATE NONCLUSTERED INDEX IX_SalesPerson_SalesQuota_SalesYTD
    ON Sales.SalesPerson (SalesQuota, SalesYTD);
GO
```

Creating a unique nonclustered index:

The following example creates a unique nonclustered index on the Name column of the Production.UnitMeasure table. The index will enforce uniqueness on the data inserted into the Name column.

```
USE AdventureWorks;
GO
IF EXISTS (SELECT name from sys.indexes
             WHERE name = N'AK_UnitMeasure_Name')
    DROP INDEX AK_UnitMeasure_Name ON Production.UnitMeasure;
GO
CREATE UNIQUE INDEX AK_UnitMeasure_Name
    ON Production.UnitMeasure(Name);
GO
```

The following query tests the uniqueness constraint by attempting to insert a row with the same value as that in an existing row.

```
--Verify the existing value.
SELECT Name FROM Production.UnitMeasure WHERE Name = N'Ounces';
GO
INSERT INTO Production.UnitMeasure (UnitMeasureCode, Name, ModifiedDate)
    VALUES ('OC', 'Ounces', GetDate());
```

The resulting error message is:

```
Server: Msg 2601, Level 14, State 1, Line 1
Cannot insert duplicate key row in object 'UnitMeasure' with unique
index 'AK_UnitMeasure_Name'. The statement has been terminated.
```

Using the IGNORE_DUP_KEY option

The following example demonstrates the affect of the IGNORE_DUP_KEY option by inserting multiple rows into a temporary table first with the option set to ON and again with the option set to

OFF. A single row is inserted into the #Test table that will intentionally cause a duplicate value when the second multiple-row INSERT statement is executed. A count of rows in the table returns the number of rows inserted.

```
USE AdventureWorks;
GO
CREATE TABLE #Test (C1 nvarchar(10), C2 nvarchar(50), C3 datetime);
GO
CREATE UNIQUE INDEX AK_Index ON #Test (C2)
    WITH (IGNORE_DUP_KEY = ON);
GO
INSERT INTO #Test VALUES (N'OC', N'Ounces', GETDATE());
INSERT INTO #Test SELECT * FROM Production.UnitMeasure;
GO
SELECT COUNT(*)AS [Number of rows] FROM #Test;
GO
DROP TABLE #Test;
GO
```

Here are the results of the second INSERT statement.

```
Server: Msg 3604, Level 16, State 1, Line 5 Duplicate key was ignored.

Number of rows
--------------
38
```

Notice that the rows inserted from the Production.UnitMeasure table that did not violate the uniqueness constraint were successfully inserted. A warning was issued and the duplicate row ignored, but the entire transaction was not rolled back.

The same statements are executed again, but with IGNORE_DUP_KEY set to OFF.

```
USE AdventureWorks;
GO
CREATE TABLE #Test (C1 nvarchar(10), C2 nvarchar(50), C3 datetime);
GO
CREATE UNIQUE INDEX AK_Index ON #Test (C2)
    WITH (IGNORE_DUP_KEY = OFF);
GO
INSERT INTO #Test VALUES (N'OC', N'Ounces', GETDATE());
INSERT INTO #Test SELECT * FROM Production.UnitMeasure;
GO
SELECT COUNT(*)AS [Number of rows] FROM #Test;
GO
DROP TABLE #Test;
GO
```

Here are the results of the second INSERT statement.

```
Server: Msg 2601, Level 14, State 1, Line 5
Cannot insert duplicate key row in object '#Test' with unique index
'AK_Index'. The statement has been terminated.

Number of rows
--------------
1
```

Notice that none of the rows from the Production.UnitMeasure table were inserted into the table even though only one row in the table violated the UNIQUE index constraint.

Using DROP_EXISTING to drop and re-create an index

The following example drops and re-creates an existing index on the ProductID column of the Production.WorkOrder table by using the DROP_EXISTING option. The options FILLFACTOR and PAD_INDEX are also set.

```
USE AdventureWorks;
GO
CREATE NONCLUSTERED INDEX IX_WorkOrder_ProductID
    ON Production.WorkOrder(ProductID)
    WITH (FILLFACTOR = 80,
        PAD_INDEX = ON,
        DROP_EXISTING = ON);
GO
```

Creating an index on a view

The following example creates a view and an index on that view. Two queries are included that use the indexed view.

```
USE AdventureWorks;
GO
--Set the options to support indexed views.
SET NUMERIC_ROUNDABORT OFF;
SET ANSI_PADDING, ANSI_WARNINGS, CONCAT_NULL_YIELDS_NULL, ARITHABORT,
    QUOTED_IDENTIFIER, ANSI_NULLS ON;
GO
--Create view with schemabinding.
IF OBJECT_ID ('Sales.vOrders', 'view') IS NOT NULL
DROP VIEW Sales.vOrders ;
GO
CREATE VIEW Sales.vOrders
WITH SCHEMABINDING
AS
    SELECT SUM(UnitPrice*OrderQty*(1.00-UnitPriceDiscount)) AS Revenue,
        OrderDate, ProductID, COUNT_BIG(*) AS COUNT
    FROM Sales.SalesOrderDetail AS od, Sales.SalesOrderHeader AS o
    WHERE od.SalesOrderID = o.SalesOrderID
    GROUP BY OrderDate, ProductID;
GO
--Create an index on the view.
CREATE UNIQUE CLUSTERED INDEX IDX_V1
    ON Sales.vOrders (OrderDate, ProductID);
GO
--This query can use the indexed view even though the view is
--not specified in the FROM clause.
SELECT SUM(UnitPrice*OrderQty*(1.00-UnitPriceDiscount)) AS Rev,
    OrderDate, ProductID
FROM Sales.SalesOrderDetail AS od
    JOIN Sales.SalesOrderHeader AS o ON od.SalesOrderID=o.SalesOrderID
        AND ProductID BETWEEN 700 and 800
        AND OrderDate >= CONVERT(datetime,'05/01/2002',101)
GROUP BY OrderDate, ProductID
ORDER BY Rev DESC;
GO
--This query can use the above indexed view.
SELECT  OrderDate, SUM(UnitPrice*OrderQty*(1.00-UnitPriceDiscount)) AS
Rev
FROM Sales.SalesOrderDetail AS od
    JOIN Sales.SalesOrderHeader AS o ON od.SalesOrderID=o.SalesOrderID
        AND DATEPART(mm,OrderDate)= 3
        AND DATEPART(yy,OrderDate) = 2002
GROUP BY OrderDate
ORDER BY OrderDate ASC;
GO
```

Creating an index with included (nonkey) columns:

The following example creates a nonclustered index with one key column (PostalCode) and four nonkey columns (AddressLine1, AddressLine2, City, StateProvinceID). A query that is covered by the index follows. To display the index that is selected by the query optimizer, on the **Query** menu in SQL Server Management Studio, select **Display Actual Execution Plan** before executing the query.

```
USE AdventureWorks;
GO
IF EXISTS (SELECT name FROM sys.indexes
            WHERE name = N'IX_Address_PostalCode')
    DROP INDEX IX_Address_PostalCode ON Person.Address;
GO
CREATE NONCLUSTERED INDEX IX_Address_PostalCode
    ON Person.Address (PostalCode)
    INCLUDE (AddressLine1, AddressLine2, City, StateProvinceID);
```

```
GO
SELECT AddressLine1, AddressLine2, City, StateProvinceID, PostalCode
FROM Person.Address
WHERE PostalCode BETWEEN N'98000' and N'99999';
GO
```

Creating a primary XML index:

The following example creates a primary XML index on the CatalogDescription column in the Production.ProductModel table.

```
USE AdventureWorks;
GO
IF EXISTS (SELECT * FROM sys.indexes
            WHERE name = N'PXML_ProductModel_CatalogDescription')
    DROP INDEX PXML_ProductModel_CatalogDescription
        ON Production.ProductModel;
GO
CREATE PRIMARY XML INDEX PXML_ProductModel_CatalogDescription
    ON Production.ProductModel (CatalogDescription);
GO
```

Creating a secondary XML index:

The following example creates a secondary XML index on the CatalogDescription column in the Production.ProductModel table.

```
USE AdventureWorks;
GO
IF EXISTS (SELECT name FROM sys.indexes
            WHERE name = N'IXML_ProductModel_CatalogDescription_Path')
    DROP INDEX IXML_ProductModel_CatalogDescription_Path
        ON Production.ProductModel;
GO
CREATE XML INDEX IXML_ProductModel_CatalogDescription_Path
    ON Production.ProductModel (CatalogDescription)
    USING XML INDEX PXML_ProductModel_CatalogDescription FOR PATH ;
GO
```

Creating a partitioned index:

The following example creates a nonclustered partitioned index on TransactionsPS1, an existing partition scheme. This example assumes the partitioned index sample has been installed. For installation information, see Readme_PartitioningScript.

```
USE AdventureWorks;
GO
IF EXISTS (SELECT name FROM sys.indexes
            WHERE name = N'IX_TransactionHistory_ReferenceOrderID')
    DROP INDEX IX_TransactionHistory_ReferenceOrderID
        ON Production.TransactionHistory;
GO
CREATE NONCLUSTERED INDEX IX_TransactionHistory_ReferenceOrderID
ON Production.TransactionHistory (ReferenceOrderID)
ON TransactionsPS1 (TransactionDate);
GO
```

Database Engine Tuning Advisor

- The Database Engine Tuning Advisor is a new tool in SQL Server 2005 that analyses the performance effects of workloads run against one or more databases. (Workload is a set of T-SQL statements that executes against databases)
- Database Engine Tuning Advisor provides higher quality recommendations to improve the performance.
- Database Engine Tuning Advisor replaces the most of the features of the Index Tuning Wizard in earlier versions of SQL Server.
- This tool also provides graphical and command-line interfaces.

Summary

- Index is an on-disk structure associated with a table or view that speeds retrieval of rows from the table or view.
- Database Engine Tuning Advisor provides higher quality recommendations to improve the performance.

Session 33: Transactions and Locking

Learning Objectives

After completing this session, you will be able to:

- Identify transactions in SQL Server
- Define locks
- List the Isolation Levels
- Describe Query Processing
- Explain the overview of using Multiple Servers:
  - Distributed Queries
  - Linked Servers

Transactions

A transaction is a sequence of operations performed as a single logical unit of work. A logical unit of work must exhibit four properties, called the atomicity, consistency, isolation, and durability (ACID) properties, to qualify as a transaction.

Atomicity: A transaction must be an atomic unit of work; either all of its data modifications are performed, or none of them is performed.

Consistency: When completed, a transaction must leave all data in a consistent state. In a relational database, all rules must be applied to the transaction's modifications to maintain all data integrity. All internal data structures, such as B-tree indexes or doubly-linked lists, must be correct at the end of the transaction.

Isolation: Modifications made by concurrent transactions must be isolated from the modifications made by any other concurrent transactions. A transaction either recognizes data in the state it was in before another concurrent transaction modified it, or it recognizes the data after the second transaction has completed, but it does not recognize an intermediate state. This is referred to as serializability because it results in the ability to reload the starting data and replay a series of transactions to end up with the data in the same state it was in after the original transactions were performed.

Durability: After a transaction has completed, its effects are permanently in place in the system. The modifications persist even in the event of a system failure.

SQL Server operates in the following transaction modes:

- Autocommit transactions: Each individual statement is a transaction.
- Explicit transactions: Each transaction is explicitly started with the BEGIN TRANSACTION statement and explicitly ended with a COMMIT or ROLLBACK statement.

- ❑ Implicit transactions: A new transaction is implicitly started when the prior transaction completes, but each transaction is explicitly completed with a COMMIT or ROLLBACK statement.
- ❑ Batch-scoped transactions: Applicable only to multiple active result sets (MARS), a Transact-SQL explicit or implicit transaction that starts under a MARS session becomes a batch-scoped transaction. A batch-scoped transaction that is not committed or rolled back when a batch completes is automatically rolled back by SQL Server.

Explicit Transactions

An explicit transaction is one in which you explicitly define both the start and end of the transaction. Explicit transactions were also called user-defined or user-specified transactions in SQL Server 7.0 or earlier.

DB-Library applications and Transact-SQL scripts use the BEGIN TRANSACTION, COMMIT TRANSACTION, COMMIT WORK, ROLLBACK TRANSACTION, or ROLLBACK WORK Transact-SQL statements to define explicit transactions.

`BEGIN TRANSACTION`: Marks the starting point of an explicit transaction for a connection.

`COMMIT TRANSACTION` or `COMMIT WORK`: Used to end a transaction successfully if no errors were encountered. All data modifications made in the transaction become a permanent part of the database. Resources held by the transaction are freed.

`ROLLBACK TRANSACTION` or `ROLLBACK WORK`: Used to erase a transaction in which errors are encountered. All data modified by the transaction is returned to the state it was in at the start of the transaction. Resources held by the transaction are freed.

Examples:

```
DECLARE @TranName VARCHAR(20);
SELECT @TranName = 'MyTransaction';

BEGIN TRANSACTION @TranName;
GO
USE AdventureWorks;
GO
DELETE FROM AdventureWorks.HumanResources.JobCandidate
    WHERE JobCandidateID = 13;
GO

COMMIT TRANSACTION MyTransaction;
GO
```

```
BEGIN TRANSACTION CandidateDelete
    WITH MARK N'Deleting a Job Candidate';
GO
USE AdventureWorks;
GO
DELETE FROM AdventureWorks.HumanResources.JobCandidate
    WHERE JobCandidateID = 13;
GO
COMMIT TRANSACTION CandidateDelete;
GO
```

Nesting Transactions

Explicit transactions can be nested. This is primarily intended to support transactions in stored procedures that can be called either from a process already in a transaction or from processes that have no active transaction.

The transaction is either committed or rolled back based on the action taken at the end of the outermost transaction. If the outer transaction is committed, then the inner nested transactions are also committed. If the outer transaction is rolled back, then all inner transactions are also rolled back, regardless of whether or not the inner transactions were individually committed.

```
USE AdventureWorks;
GO
CREATE TABLE TestTrans(Cola INT PRIMARY KEY,
               Colb CHAR(3) NOT NULL);
GO
CREATE PROCEDURE TransProc @PriKey INT, @CharCol CHAR(3) AS
BEGIN TRANSACTION InProc
INSERT INTO TestTrans VALUES (@PriKey, @CharCol)
INSERT INTO TestTrans VALUES (@PriKey + 1, @CharCol)
COMMIT TRANSACTION InProc;
GO
/* Start a transaction and execute TransProc. */
BEGIN TRANSACTION OutOfProc;
GO
EXEC TransProc 1, 'aaa';
GO
/* Roll back the outer transaction, this will
   roll back TransProc's nested transaction. */
ROLLBACK TRANSACTION OutOfProc;
GO
EXECUTE TransProc 3,'bbb';
GO
/* The following SELECT statement shows only rows 3 and 4 are
   still in the table. This indicates that the commit
   of the inner transaction from the first EXECUTE statement of
   TransProc was overridden by the subsequent rollback. */
SELECT * FROM TestTrans;
GO
```

Distributed Transactions

A transaction within a single instance of the Database Engine that spans two or more databases is actually a distributed transaction.

Distributed transactions span two or more servers known as resource managers. The management of the transaction must be coordinated between the resource managers by a server component called a transaction manager.

Instance of the SQL Server Database Engine can operate as a resource manager in distributed transactions coordinated by transaction managers, such as Microsoft Distributed Transaction Coordinator (MS DTC).

Achieved by managing the commit process in two phases known as a two-phase commit (2PC).:

- Prepare phase
- Commit phase

Locks

Locking is a mechanism used by the Microsoft SQL Server Database Engine to synchronize access by multiple users to the same piece of data at the same time.

Ensures transactional integrity:

- Prevent “reads” on data being modified by other users
- Prevent multiple users from changing same data at simultaneously

Locks are managed internally by a part of the Database Engine called the lock manager.
SQL server can lock the following resources:

- **RID:** Row identifier used to lock a single row within a table
- **Key:** Row lock within an index used to protect key changes in serializable transactions
- **Page:** Data page or index page (8 KB)
- **Extent:** Contiguous group of eight data pages
- **Table:** Entire table including all table and indexes
- **DB:** Database
- **FILE:** A database file.

The Microsoft SQL Server Database Engine locks resources using different lock modes that determine how the resources can be accessed by concurrent transactions
Lock modes that uses Database Engine :

- **Shared (S):** Used for read operations that do not change or update data, such as a SELECT statement.

- **Exclusive (X):** Ensures that multiple updates cannot be made to the same resource at the same time.
- Update (U): Used on resources that can be updated. Prevents a common form of deadlock that occurs when multiple sessions are reading, locking, and potentially updating resources.
- Other lock modes like Intent , Schema , Bulk Update (BU) and Key-range

Isolation Levels

Isolation levels allow you to control the consistency level while manipulating data when multiple processes might be running concurrently.

The isolation behavior of a transaction can be tweaked to your application's needs. This is generally done by setting the isolation level of a transaction. Put simply, isolation levels determine how concurrent transactions behave. Do they block each other? Do they let each other step over themselves? Or do they present a snapshot of a previous stable state of data in the event of an overstepping?

You might find a slight mismatch between the isolation levels defined in ADO.NET compared with the isolation levels in SQL Server 2005 because ADO.NET was not written exclusively for SQL Server. We'll begin by looking at the isolation-level support in SQL Server 2005.

Isolation Levels in SQL Server 2005:

You can set isolation levels in SQL Server 2005 by using the SET TRANSACTION ISOLATION LEVEL T-SQL command, which uses the following syntax:

```
SET TRANSACTION ISOLATION LEVEL
    { READ UNCOMMITTED
    | READ COMMITTED
    | REPEATABLE READ
    | SNAPSHOT
    | SERIALIZABLE
    }
```

A sample usage of this command with the BEGIN TRANSACTION command is shown here:

```
SET TRANSACTION ISOLATION LEVEL READ UNCOMMITTED
BEGIN TRANSACTION
 SELECT TestColumn FROM TestTable
COMMIT
```

As you can see from the syntax, SQL Server 2005 supports five isolation levels:

- Read uncommitted
- Read committed
- Repeatable read
- Snapshot
- Serializable

Read Uncommitted Isolation Level:

By specifying the read uncommitted isolation level, you essentially tell the database to violate all locks and read the current immediate state of data. But by doing so, you might end up with a dirty readreading data that is not yet committed. You should therefore avoid this isolation level because it can return logically corrupt data.

Let's explore this isolation level by using an example.

1. Create a table called Test with the following data in it.

   TestID   TestColumn
2. -------------------

   100
3. Open two instances of SQL Server Management Studio. In each instance, use the database that contains the Test table. These two instances will be used to simulate two users running two concurrent transactions.
4. In instance 1, execute an UPDATE on the row of data by executing the following code block:

   BEGIN TRANSACTION

   UPDATE Test SET TestColumn = 200 WHERE TestId = 1

   In instance 2 of Management Studio, execute the following query:

   SELECT TestColumn FROM Test WHERE TestId = 1
5. You will notice that your SELECT query is blocked. This makes sense because you are trying to read the same data that instance 1 is busy modifying. Unless instance 1 issues a COMMIT or a ROLLBACK, your query will remain blocked or will simply time out.
6. Cancel your blocked SELECT query by pressing Alt+Break or clicking the Cancel button on the toolbar. Execute the following T-SQL command to set the isolation level of your SELECT query to read uncommitted on the connection held by instance 2.

   SET TRANSACTION ISOLATION LEVEL READ UNCOMMITTED
7. Execute the SELECT query again.

   SELECT TestColumn FROM Test WHERE TestId = 1
8. You will find that the query isn't blocked; it produces 200 as a result.
9. Go back to instance 1 and issue a ROLLBACK.
10. Back in instance 2, execute the same SELECT query again. You should get 100 as the result.

As you might have noticed, instance 2 returned different results for the same query at different times. As a matter of fact, the value 200 was never committed to the database, but because you explicitly requested a dirty read by specifying the READ UNCOMMITTED isolation level, you ended up reading data that was never meant to be final. So the downside is that you ended up reading logically incorrect data. On the upside, however, your query did not get blocked.

Read Committed Isolation Level

Read committed is the default isolation level in both SQL Server and Oracle databases. As you will see shortly, this isolation level is the default because it represents the best compromise between data integrity and performance. This isolation level respects locks and prevents dirty reads from occurring. In the example you saw earlier, until we explicitly requested that the isolation level be changed to read uncommitted, the connection worked at the read committed isolation level. Therefore, our second transaction (the autocommit mode transaction in the SELECT query) was blocked by the transaction executing the UPDATE query.

Read committed prevents dirty reads, but phantom reads and nonrepeatable reads are still possible when using this isolation level. This is because the read committed isolation level does not prevent one transaction from changing the same data at the same time as another transaction is reading from it.

A phantom read can occur in the following type of situation:

- ❑ Transaction 1 begins.
- ❑ Transaction 1 reads a row.
- ❑ Transaction 2 begins.
- ❑ Transaction 2 deletes the row that was read by transaction 1.
- ❑ Transaction 2 commits. Transaction 1 can no longer repeat its initial read because the row no longer exists, resulting in a phantom row.

A nonrepeatable read can occur in the following type of situation:

- ❑ Transaction 1 begins.
- ❑ Transaction 1 reads a row.
- ❑ Transaction 2 begins.
- ❑ Transaction 2 changes the value of the same row read by transaction 1.
- ❑ Transaction 2 commits.
- ❑ Transaction 1 reads the row again. Transaction 1 has inconsistent data because the row now contains different values from the previous read, all within the scope of transaction 1.

Repeatable Read Isolation Level

As the name suggests, the repeatable read isolation level prevents nonrepeatable reads. It does so by placing locks on the data that was used in a query within a transaction. As you might expect, you pay a higher price in terms of concurrent transactions blocking each other, so you should use this isolation level only when necessary. The good news, however, is that a concurrent transaction can add new data that matches the WHERE clause of the original transaction. This is because the first transaction will place a lock only on the rows it originally read into its result set. In other words, a transaction using this isolation level acquires read locks on all retrieved data but does not acquire range locks.

If you examine this pattern closely, you'll see that although nonrepeatable reads are avoided when using this isolation level, phantom reads can still occur. They can occur under the following circumstances:

- ❑ Transaction 1 begins.
- ❑ Transaction 1 reads all rows with, say, TestColumn = 100.
- ❑ Transaction 2 begins.
- ❑ Transaction 2 inserts a new row with TestID = 2, TestColumn = 100.
- ❑ Transaction 2 commits.
- ❑ Transaction 1 runs an UPDATE query and modifies TestColumn for the rows with TestColumn = 100. This also ends up updating the row that transaction 2 inserted.
- ❑ Transaction 1 commits.

Because shared locks are not released until the end of the transaction, concurrency is lower than when using the read committed isolation level, so care must be taken to avoid unexpected results.

Serializable Isolation Level:

A transaction running at the serializable isolation level will not permit dirty reads, phantom reads, or nonrepeatable reads. This isolation level places the most restrictive locks on the data being read or modified, keeping your data perfectly clean. This might sound like an isolation level that gives you perfect isolation behavior, but there is a good reason why you should seldom use this isolation level. This places the most restrictive locks on the data being read, or modified. In a sense, this is the perfect transaction, but transactions will block other running transactions, thereby affecting concurrent performance or even creating deadlocks. Thus even if this transaction will keep your data perfectly clean, it will severely affect system performance. In most practical situations, you can get away with a lower isolation level.

Snapshot Isolation Level:

In all isolation levels previously presented, we seem to be trading concurrent performance for logical sanctity of data. Because a transaction locks the data it is working on, other transactions that attempt to work with the same data are blocked until the first transaction commits or rolls back. Of course, the traditional way of getting around this problem is to allow dirty reads (and hence corrupt data) or simply reduce the duration of transactions. But neither of these solutions allows you to read logically consistent data while offering nonblocking concurrent behavior.

Application architectures frequently present circumstances in which even the smallest transactions become a problem or transactions end up modifying so much data that their duration cannot be kept small. To get around this issue, SQL Server 2005 introduces a new isolation level: the snapshot isolation level. This isolation level gives you consistent reads without blocking.

Transactions running under the snapshot isolation level do not create shared locks on the rows being read. In addition, repeated requests for the same data within a snapshot transaction guarantee the same results, thus ensuring repeatable reads without any blocking. This sounds like the best of both worldsthe responsiveness of read uncommitted combined with the consistency of repeatable read. However, you pay a price.

This nonblocking, repeatable read behavior is made possible by storing previously committed versions of rows in the tempdb database. As a result, other transactions that were started before the write in the current transaction and that have already read the previous version will continue to read that version. Because the previous version is being read from tempdb, the write can occur in a nonblocking fashion and other transactions will see the new version. The obvious problem, of course, is the increased overhead on the tempdb database. For this reason, SQL Server requires you to enable the snapshot isolation level before you can use it. You shouldn't arbitrarily enable snapshot isolation on databases. But after testing, if you decide that your database needs this isolation level, you can enable it by using the following T-SQL commands:

```
ALTER DATABASE Test
SET ALLOW_SNAPSHOT_ISOLATION ON
```

As with all isolation levels, once you enable snapshot isolation for a database, you can use it on individual connections by using the following T-SQL statement:

```
SET TRANSACTION ISOLATION LEVEL SNAPSHOT
```

Read Committed Snapshot Isolation Level:

Snapshot isolation prevents readers from being blocked by writers by providing readers with data from a previously committed version. Over the duration of the transaction, you are thus assured of repeatable reads. However, this method of ensuring a repeatable read incurs additional overhead and bookkeeping for the database engine that might not be necessary in all situations. Thus, SQL Server 2005 offers a slight modification to the read committed isolation level that provides nonrepeatable reads over the duration of the transaction that are not blocked by transactions writers. It is called the read committed snapshot isolation level. This isolation level guarantees consistency of the data over the duration of a read query within a transaction but not over the entire transaction that holds the reader. The obvious advantage over read committed snapshot as compared to read committed is that your readers do not get blocked. When they request data, they are offered either a previous state of data (before any write operations) or the new state of data (after write operations), depending on the state of other concurrently running transactions, but they are never required to wait until other concurrent transactions release their locks on the data being requested.

To use the read committed snapshot isolation level, you must first enable it at the database level by using the following T-SQL command:

```
ALTER DATABASE Test
SET READ_COMMITTED_SNAPSHOT ON
```

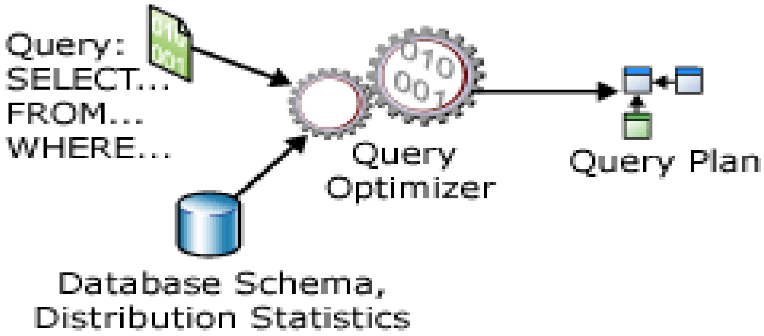
Once you have enabled the read committed snapshot isolation level on a database, all queries using the read committed isolation level will exhibit snapshot-like behavior. Although this isolation level will give you snapshot-like behavior, you will not be able to do repeatable reads over the duration of a transaction.

Query Processing

Steps that SQL Server applies to process a SELECT statement:

1. The parser scans the SELECT statement and breaks it into logical units such as keywords, expressions, operators, and identifiers.
2. A query tree, sometimes referred to as a sequence tree, is built describing the logical steps needed to transform the source data into the format required by the result set.
3. The query optimizer analyzes different ways the source tables can be accessed. It then selects the series of steps that returns the results fastest while using fewer resources. The query tree is updated to record this exact series of steps. The final, optimized version of the query tree is called the execution plan.
4. The relational engine starts executing the execution plan. As the steps that require data from the base tables are processed, the relational engine requests that the storage engine pass up data from the rowsets requested from the relational engine.
5. The relational engine processes the data returned from the storage engine into the format defined for the result set and returns the result set to the client.

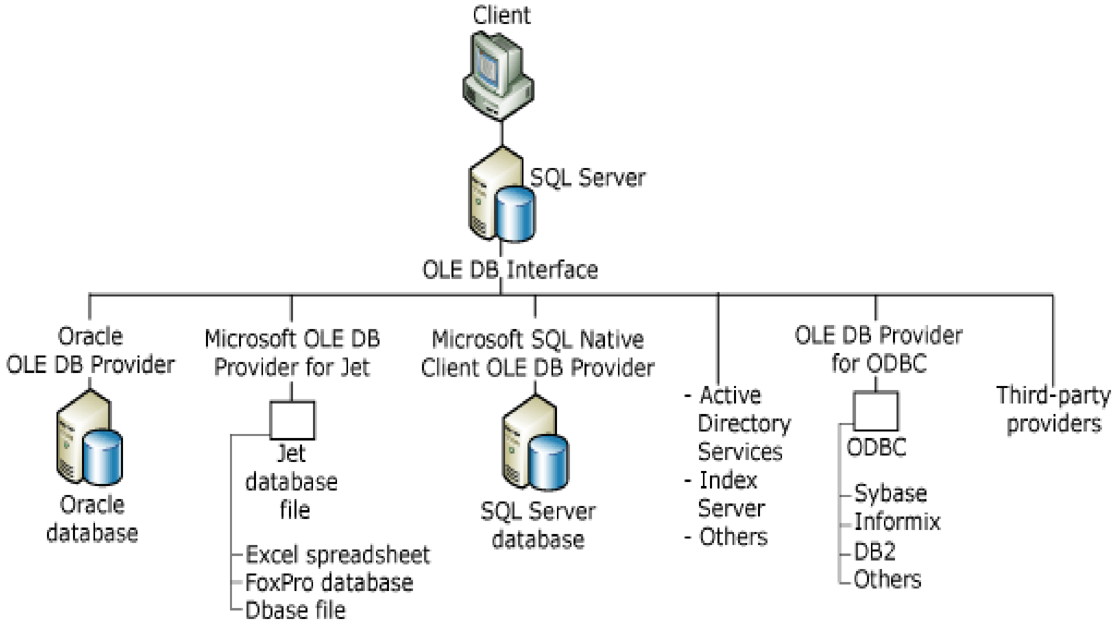
Query Optimization of a single SELECT statement



Distributed Queries

- ❑ Access data stored in multiple SQL servers
- ❑ Access data from heterogeneous data sources
- ❑ Implemented using OLE DB
  - o OLE DB to communicate between Relational and storage engines
  - o OLE DB exposes data in tabular objects (rowsets)
  - o Allows rowsets to be accessed in T-SQL statements
  - o Any data source with OLE DB provider allows heterogeneous access
- ❑ Two types of access are as follows:
  - o AD HOC queries
  - o Using linked servers

Architecture shows the connections between a client computer, an instance of SQL Server, and an OLE DB provider



OPENROWSET function:

- A one-time, ad hoc method of connecting and accessing remote data.
- Specify connection information and a Table, view or Select query.

```
SELECT a.*
FROM OPENROWSET('SQLNCLI', 'Server=Seattle1;
Trusted_Connection=yes;','SELECT GroupName, Name,
 DepartmentID FROM AdventureWorks.HumanResources.Department
      ORDER BY GroupName, Name') AS a;
```

OPENDATASOURCE function: Provides ad hoc connection information as part of a four-part object name.

Syntax :

```
OPENDATASOURCE ( provider_name, init_string )
```

- AD HOC naming only for rarely used quires
- Frequent usage: Apply Linked servers

Linked Servers:

- A linked server configuration enables SQL Server to execute commands against OLE DB data sources on remote servers.
- Linked servers are applied to handle distributed queries.
- Linked servers offer the following advantages:
  - Remote server access.
  - The ability to issue distributed queries, updates, commands, and transactions on heterogeneous data sources across the enterprise.
  - The ability to address diverse data sources similarly.
- Apply stored procedures and catalog views to manage linked server definitions:
  - Create a linked server definition by running **sp_addlinkedserver** procedure.
  - View information about the linked servers defined in a specific instance of SQL Server by running a query against the **sys.servers** system catalog views.
  - Delete a linked server definition by running **sp_dropserver** procedure.

Syntax:

```
sp_addlinkedserver [ @server= ] 'server' [ , [ @srvproduct= ]
'product_name' ]
     [ , [ @provider= ] 'provider_name' ]
     [ , [ @datasrc= ] 'data_source' ]
     [ , [ @location= ] 'location' ]
     [ , [ @provstr= ] 'provider_string' ]
     [ , [ @catalog= ] 'catalog' ]
```

Example:

```
sp_addlinkedserver @server = N'LinkServer',
    @srvproduct = N' ',
    @provider = N'SQLNCLI',
    @datasrc = N'ServerNetName',
    @catalog = N'AdventureWorks'
GO
SELECT *
FROM LinkServer.AdventureWorks.dbo.Vendor
```

Summary

- A transaction is a single unit of work.
- Locking is a mechanism used by the Microsoft SQL Server Database Engine to synchronize access by multiple users to the same piece of data at the same time.
- Distributed queries access data from multiple heterogeneous data sources.
- A linked server configuration enables SQL Server to execute commands against OLE DB data sources on remote servers

Session 36: XML Support in SQL Server

Learning Objectives

After completing this session, you will be able to:

- ❑ Apply XML in SQL Server
- ❑ Identify XML data type
- ❑ Write FOR XML commands
- ❑ Execute XQuery

XML in SQL Server

Working with XML is a common requirement of many of applications today. There are many cases when an application developer must transform data between XML and relational formats, and even store XML and manipulate data natively in a relational database.

- ❑ XML supports in SQL Server are as follows:
  - o Support for URL access
  - o Support for XML data schema
  - o Support to parse an XML document
  - o Ability to specify XPath queries
  - o Ability to read data as XML:

    Using SELECT …. FOR XML

    Ability to write XML to the database:

    Using OPENXML
  - o Ability to specify an XQuery query against XML data stored in columns and variables of the xml type.

The XML Data Type

SQL Server 2005 introduces a new data type for working with XML data: the XML data type. Using this new data type, you can store XML in its native format, query the data within the XML, efficiently and easily modify data within the XML without having to replace the entire contents, and index the data in the XML. You can use it as any of the following:

- ❑ A variable
- ❑ A parameter in a stored procedure or a user-defined function (UDF)
- ❑ A return value from a UDF
- ❑ A column in a table

There are some limitations of the XML data type that you should be aware of. Although this data type can contain and be checked for null values, unlike other native types, you cannot directly compare an instance of an XML data type to another instance of an XML data type. Any such equality comparisons require first casting the XML type to a character type. This limitation also means that you cannot use ORDER BY or GROUP BY with an XML data type. There are several other restrictions, which we will discuss in more detail later.

These might seem like pretty severe restrictions, but they don't really affect the XML data type when it is used appropriately. The XML data type also has a rich feature set that more than compensates for these limitations.

Working with the XML Data Type as a Variable:

Let's start by writing some code that uses the new XML data type as a variable. As with any other T-SQL variable, you simply declare it and assign data to it, as in the following example, which uses a generic piece of XML to represent a sales representative's data as XML:

```
DECLARE @xmlData AS XML
Set @xmlData='
<Customers>
  <CustomerID>CORZN</CustomerID>
  <CompanyName>Corzen, Inc</CompanyName>
  <ContactName>Stephen Forte</ContactName>
  <ContactTitle>Sales Representative</ContactTitle>
  <Address>5-9 Union Square West</Address>
  <City>New York</City>
  <PostalCode>10028</PostalCode>
  <Country>USA</Country>
  <Phone>030-0074321</Phone>
  <Fax>030-0076545</Fax>
  </Customers>
'
Select @xmlData
```

This basic example shows an XML variable being declared like any other native SQL Server data type by using the DECLARE statement. Then the variable is assigned a value. Oddly enough, a string of XML data is assigned to the XML data type and the type parses it into XML. (Coincidentally, CLR-based user-defined types also support and require this same parsing functionality.) The example also checks that the XML is well formed, such as by validating that an element's start and end tags match not only in name but also in case.

Working with XML in Tables:

As we stated earlier, you can also define a column in a table as XML, as shown in the following example:

```
USE AdventureWorks
GO
--create the table with the XML Datatype
CREATE TABLE OrdersXML
  (OrderDocID INT PRIMARY KEY,
  xOrders XML NOT NULL)
```

As we also stated earlier, the XML data type has a few other restrictions, in this case when used as a column in a table:

- It cannot be used as a primary key.
- It cannot be used as a foreign key.
- It cannot be declared with a UNIQUE constraint.
- It cannot be declared with the COLLATE keyword.

We stated earlier that you can't compare two instances of the XML data type. Primary keys, foreign keys, and unique constraints all require that you must be able to compare any included data types; therefore, XML cannot be used in any of those situations. The SQL Server COLLATE statement is meaningless with the XML data type because SQL Server does not store the XML as text; rather, it uses a distinct type of encoding particular to XML.

Now let's get some data into the column. This example takes some simple static XML and inserts it into the OrdersXML table we just created, using the XML data type as a variable.

```
USE AdventureWorks
GO
--Insert Static XML via a variable
DECLARE @xmlData AS XML
SET @xmlData = '
<Orders>
   <Order>
   <OrderID>5</OrderID>
   <CustomerID>65</CustomerID>
   <OrderAmount>25</OrderAmount>
   </Order>
</Orders>'
--insert into the table
INSERT INTO OrdersXML (OrderDocID, xOrders) Values (1, @xmlData)
```

You can insert XML into these columns in a variety of other ways: XML bulk load (which we will discuss later in the chapter), loading from an XML variable (as shown here), or loading from a SELECT statement using the FOR XML TYPE feature, which we will discuss shortly. Only well-formed XML (including fragments) can be inserted; any attempt to insert malformed XML will result in an exception, as shown in this fragment:

```
--Fails because of the malformed XML
INSERT INTO OrdersXML (OrderDocID, xOrders) VALUES (3, '<nm>steve</NM>')
```

The results produce the following error from SQL Server:

```
Msg 9436, Level 16, State 1, Line 1
XML parsing: line 1, character 14, end tag does not match start tag
```

Defaults and Constraints

The XML data type can, like other data types, conform to nullability, defaults, and constraints. If you want to make a field required (NOT NULL) and provide a default value on your XML column, just specify this as you would for any other column.

```
USE AdventureWorks
GO
CREATE TABLE OrdersXML
   (OrderDocID INT PRIMARY KEY,
   xOrders XML NOT NULL DEFAULT '<Orders/>'
```

The following insert works because it relies on the default:

```
INSERT INTO OrdersXML (OrderDocID, xOrders) VALUES (2, DEFAULT)
```

Adding a default does not enforce just <Orders> from being added. For example, we have specified an Orders node as the default but have not yet indicated any way to enforce that. The following insert works even if we only want <Orders> in our table because we have not declared a constraint on the column.

```
Insert Into OrdersXML (OrderDocID, xOrders) Values (3,
'<blah>steve</blah>')
```

FOR XML Commands

SQL Server 2000 introduced an enhancement to the T-SQL syntax that enables normal relational queries to output their result set as XML, using any of these three approaches:

- ❑ FOR XML RAW
- ❑ FOR XML AUTO
- ❑ FOR XML EXPLICIT

As you probably expected, these three features are also in SQL Server 2005. We'll first discuss the features common to both versions and then look closely at the new and enhanced features available in SQL Server 2005.

FOR XML RAW:

FOR XML RAW produces what we call attribute-based XML. FOR XML RAW essentially creates a flat representation of the data where each row returned becomes an element and the returned columns become the attributes of each element. FOR XML RAW also doesn't interpret joins in any special way. (Joins become relevant in FOR XML AUTO.) example of a simple query that retrieves customer and order header data.

Customer and Order Header Data with FOR XML RAW

```
USE AdventureWorks
GO

SELECT TOP 10 -- limits the result rows for demo purposes
   Customer.CustomerID, OrderHeader.SalesOrderID, OrderHeader.OrderDate
FROM Sales.Customer Customer
```

```
INNER JOIN Sales.SalesOrderHeader OrderHeader
   ON OrderHeader.CustomerID = Customer.CustomerID
ORDER BY Customer.CustomerID
FOR XML RAW
```

If you are using SQL Server 2000, this will be output as a stream of text to Query Analyzer. However, we will assume that you are working with SQL Server 2005, so you can click on the XML hyperlink in the returned results to see the results shown here:

```
<row CustomerID="1" SalesOrderID="43860" OrderDate="2001-08-01T00:00:00"
/>
<row CustomerID="1" SalesOrderID="44501" OrderDate="2001-11-01T00:00:00"
/>
<row CustomerID="1" SalesOrderID="45283" OrderDate="2002-02-01T00:00:00"
/>
<row CustomerID="1" SalesOrderID="46042" OrderDate="2002-05-01T00:00:00"
/>
<row CustomerID="2" SalesOrderID="46976" OrderDate="2002-08-01T00:00:00"
/>
<row CustomerID="2" SalesOrderID="47997" OrderDate="2002-11-01T00:00:00"
/>
<row CustomerID="2" SalesOrderID="49054" OrderDate="2003-02-01T00:00:00"
/>
<row CustomerID="2" SalesOrderID="50216" OrderDate="2003-05-01T00:00:00"
/>
<row CustomerID="2" SalesOrderID="51728" OrderDate="2003-08-01T00:00:00"
/>
<row CustomerID="2" SalesOrderID="57044" OrderDate="2003-11-01T00:00:00"
/>
```

As promised, you get flat results where each row returned from the query becomes a single element named row and where all columns are output as attributes of that element. Odds are, however, that you will want more structured XML output, which leads us to the next topic, FOR XML AUTO.

FOR XML AUTO:

FOR XML AUTO produces attribute-based XML by default and can create nested results based on the tables in the query's join clause. For example, using the same query just demonstrated, you can simply change the FOR XML clause to FOR XML AUTO, as shown here.

Customer and Order Header Data with FOR XML AUTO

```
USE AdventureWorks
GO

SELECT TOP 10 -- limits the result rows for demo purposes
   Customer.CustomerID, OrderHeader.SalesOrderID, OrderHeader.OrderDate
FROM Sales.Customer Customer
INNER JOIN Sales.SalesOrderHeader OrderHeader
```

```
   ON OrderHeader.CustomerID = Customer.CustomerID
ORDER BY Customer.CustomerID
FOR XML AUTO
```

Execute and click the XML hyperlink in the results, and you will see the following output:

```
<Customer CustomerID="1">
  <OrderHeader SalesOrderID="43860" OrderDate="2001-08-01T00:00:00" />
  <OrderHeader SalesOrderID="44501" OrderDate="2001-11-01T00:00:00" />
  <OrderHeader SalesOrderID="45283" OrderDate="2002-02-01T00:00:00" />
  <OrderHeader SalesOrderID="46042" OrderDate="2002-05-01T00:00:00" />
</Customer>
<Customer CustomerID="2">
  <OrderHeader SalesOrderID="46976" OrderDate="2002-08-01T00:00:00" />
  <OrderHeader SalesOrderID="47997" OrderDate="2002-11-01T00:00:00" />
  <OrderHeader SalesOrderID="49054" OrderDate="2003-02-01T00:00:00" />
  <OrderHeader SalesOrderID="50216" OrderDate="2003-05-01T00:00:00" />
  <OrderHeader SalesOrderID="51728" OrderDate="2003-08-01T00:00:00" />
  <OrderHeader SalesOrderID="57044" OrderDate="2003-11-01T00:00:00" />
</Customer>
```

As you can see, the XML data has main elements named Customer (based on the alias assigned in the query) and subelements named OrderHeader (again from the alias). Note that FOR XML AUTO determines the element nesting order based on the order of the columns in the SELECT clause. You can rewrite the SELECT clause so an OrderHeader column comes before a Customer Column, as shown here:

```
SELECT TOP 10 -- limits the result rows for demo purposes
   OrderHeader.SalesOrderID, OrderHeader.OrderDate, Customer.CustomerID
```

The output (as viewed in the XML viewer) now looks like this:

```
<OrderHeader SalesOrderID="43860" OrderDate="2001-08-01T00:00:00">
  <Customer CustomerID="1" />
</OrderHeader>
<OrderHeader SalesOrderID="44501" OrderDate="2001-11-01T00:00:00">
  <Customer CustomerID="1" />
</OrderHeader>
<OrderHeader SalesOrderID="45283" OrderDate="2002-02-01T00:00:00">
  <Customer CustomerID="1" />
</OrderHeader>
<OrderHeader SalesOrderID="46042" OrderDate="2002-05-01T00:00:00">
  <Customer CustomerID="1" />
</OrderHeader>
<OrderHeader SalesOrderID="46976" OrderDate="2002-08-01T00:00:00">
  <Customer CustomerID="2" />
</OrderHeader>
<OrderHeader SalesOrderID="47997" OrderDate="2002-11-01T00:00:00">
```

```
  <Customer CustomerID="2" />
</OrderHeader>
<OrderHeader SalesOrderID="49054" OrderDate="2003-02-01T00:00:00">
  <Customer CustomerID="2" />
</OrderHeader>
<OrderHeader SalesOrderID="50216" OrderDate="2003-05-01T00:00:00">
  <Customer CustomerID="2" />
</OrderHeader>
<OrderHeader SalesOrderID="51728" OrderDate="2003-08-01T00:00:00">
  <Customer CustomerID="2" />
</OrderHeader>
<OrderHeader SalesOrderID="57044" OrderDate="2003-11-01T00:00:00">
  <Customer CustomerID="2" />
</OrderHeader>
```

These results are probably not the results you wanted. To keep the XML hierarchy matching the table hierarchy, you must list at least one column from the parent table before any column from a child table. If there are three levels of tables, at least one other column from the child table must come before any from the grandchild table, and so on.

FOR XML EXPLICIT:

FOR XML EXPLICIT is the most complex but also the most useful and flexible of the three options. It produces XML by constructing a UNION query of the various levels of output elements. So, if again you have the Customer and SalesOrderHeader tables and you want to produce XML output, you must have two SELECT statements with a UNION. If you add the SalesOrderDetail table, you must add another UNION statement and SELECT statement.

As we said, FOR XML EXPLICIT is more complex than its predecessors. For starters, you are responsible for defining two additional columns that establish the hierarchical relationship of the XML: a Tag column that acts as a row's identifier and a Parent column that links child records to the parent record's Tag value (similar to EmployeeID and ManagerID). You must also alias all columns to indicate the element, Tag, and display name for the XML output, as shown here. Keep in mind that only the first SELECT statement must enforce these rules because aliases in subsequent SELECT statements in a UNION query are ignored.

Customer and Order Header Data with FOR XML EXPLICIT

```
USE AdventureWorks
GO

SELECT TOP 2 -- limits the result rows for demo purposes
   1 AS Tag,
   NULL AS Parent,
   CustomerID AS [Customer!1!CustomerID],
   NULL AS [SalesOrder!2!SalesOrderID],
   NULL AS [SalesOrder!2!OrderDate]
FROM Sales.Customer AS Customer
```

```
UNION ALL

SELECT TOP 10 -- limits the result rows for demo purposes
   2,
   1,
   Customer.CustomerID,
   OrderHeader.SalesOrderID,
   OrderHeader.OrderDate
FROM Sales.Customer AS Customer
INNER JOIN Sales.SalesOrderHeader AS OrderHeader
   ON OrderHeader.CustomerID = Customer.CustomerID

ORDER BY [Customer!1!CustomerID], [SalesOrder!2!SalesOrderID]
FOR XML EXPLICIT
```

Execute and click the XML hyperlink to see the following results:

```
<Customer CustomerID="1">
  <SalesOrder SalesOrderID="43860" OrderDate="2001-08-01T00:00:00" />
  <SalesOrder SalesOrderID="44501" OrderDate="2001-11-01T00:00:00" />
  <SalesOrder SalesOrderID="45283" OrderDate="2002-02-01T00:00:00" />
  <SalesOrder SalesOrderID="46042" OrderDate="2002-05-01T00:00:00" />
</Customer>
<Customer CustomerID="2">
  <SalesOrder SalesOrderID="46976" OrderDate="2002-08-01T00:00:00" />
  <SalesOrder SalesOrderID="47997" OrderDate="2002-11-01T00:00:00" />
  <SalesOrder SalesOrderID="49054" OrderDate="2003-02-01T00:00:00" />
  <SalesOrder SalesOrderID="50216" OrderDate="2003-05-01T00:00:00" />
  <SalesOrder SalesOrderID="51728" OrderDate="2003-08-01T00:00:00" />
  <SalesOrder SalesOrderID="57044" OrderDate="2003-11-01T00:00:00" />
</Customer>
```

FOR XML EXPLICIT allows for some alternative outputs that are not achievable using RAW. For example, you can specify that certain values be composed as elements instead of attributes by including !ELEMENT on the end of the aliased column.

Using FOR XML EXPLICIT

```
USE AdventureWorks
GO
--XML EXPLICIT
SELECT TOP 2 -- limits the result rows for demo purposes
   1 AS Tag,
   NULL AS Parent,
   CustomerID AS [Customer!1!CustomerID],
   NULL AS [SalesOrder!2!SalesOrderID],
   NULL AS [SalesOrder!2!OrderDate!ELEMENT] --Render as an element
FROM Sales.Customer AS Customer
UNION ALL

SELECT TOP 10 -- limits the result rows for demo purposes
   2,
   1,
   Customer.CustomerID,
   OrderHeader.SalesOrderID,
   OrderHeader.OrderDate
FROM Sales.Customer AS Customer
INNER JOIN Sales.SalesOrderHeader AS OrderHeader
   ON OrderHeader.CustomerID = Customer.CustomerID

ORDER BY [Customer!1!CustomerID], [SalesOrder!2!SalesOrderID]
FOR XML EXPLICIT
```

Only one minor change was made (as indicated by the comment). However, this change has a major effect on the final output, as shown here:

```
<Customer CustomerID="1">
  <SalesOrder SalesOrderID="43860">
    <OrderDate>2001-08-01T00:00:00</OrderDate>
  </SalesOrder>
  <SalesOrder SalesOrderID="44501">
    <OrderDate>2001-11-01T00:00:00</OrderDate>
  </SalesOrder>
  <SalesOrder SalesOrderID="45283">
    <OrderDate>2002-02-01T00:00:00</OrderDate>
  </SalesOrder>
  <SalesOrder SalesOrderID="46042">
    <OrderDate>2002-05-01T00:00:00</OrderDate>
  </SalesOrder>
</Customer>
<Customer CustomerID="2">
  <SalesOrder SalesOrderID="46976">
    <OrderDate>2002-08-01T00:00:00</OrderDate>
  </SalesOrder>
  <SalesOrder SalesOrderID="47997">
    <OrderDate>2002-11-01T00:00:00</OrderDate>
  </SalesOrder>
  <SalesOrder SalesOrderID="49054">
    <OrderDate>2003-02-01T00:00:00</OrderDate>
  </SalesOrder>
  <SalesOrder SalesOrderID="50216">
    <OrderDate>2003-05-01T00:00:00</OrderDate>
  </SalesOrder>
  <SalesOrder SalesOrderID="51728">
    <OrderDate>2003-08-01T00:00:00</OrderDate>
  </SalesOrder>
  <SalesOrder SalesOrderID="57044">
    <OrderDate>2003-11-01T00:00:00</OrderDate>
  </SalesOrder>
</Customer>
```

Notice that the OrderDate is rendered as a child element of the SalesOrder element. Although EXPLICIT mode could create robust results, it also requires creating even more complex queries to get such results. For example, to add a few more fields from OrderHeader and to add some additional fields from OrderDetail, you would have to write the query using FOR XML EXPLICIT.

Using FOR XML EXPLICIT with added fields:

```
USE AdventureWorks
GO
--XML EXPLICIT
SELECT --TOP 2 -- limits the result rows for demo purposes
   1 AS Tag,
   NULL AS Parent,
   CustomerID AS [Customer!1!CustomerID],
   NULL AS [SalesOrder!2!SalesOrderID],
   NULL AS [SalesOrder!2!TotalDue],
   NULL AS [SalesOrder!2!OrderDate!ELEMENT],
   NULL AS [SalesOrder!2!ShipDate!ELEMENT],
   NULL AS [SalesDetail!3!ProductID],
   NULL AS [SalesDetail!3!OrderQty],
   NULL AS [SalesDetail!3!LineTotal]
FROM Sales.Customer AS Customer
WHERE Customer.CustomerID IN (1, 2)

UNION ALL

SELECT
   2,
   1,
   Customer.CustomerID,
   OrderHeader.SalesOrderID,
   OrderHeader.TotalDue,
   OrderHeader.OrderDate,
   OrderHeader.ShipDate,
   NULL,
   NULL,
   NULL
FROM Sales.Customer AS Customer
INNER JOIN Sales.SalesOrderHeader AS OrderHeader
   ON OrderHeader.CustomerID = Customer.CustomerID
WHERE Customer.CustomerID IN (1, 2)

UNION ALL
SELECT
   3,
   2,
   Customer.CustomerID,
   OrderHeader.SalesOrderID,
   OrderHeader.TotalDue,
   OrderHeader.OrderDate,
   OrderHeader.ShipDate,
   OrderDetail.ProductID,
   OrderDetail.OrderQty,
   OrderDetail.LineTotal
FROM Sales.Customer AS Customer
INNER JOIN Sales.SalesOrderHeader AS OrderHeader
   ON OrderHeader.CustomerID = Customer.CustomerID
INNER JOIN Sales.SalesOrderDetail AS OrderDetail
   ON OrderDetail.SalesOrderID = OrderHeader.SalesOrderID
WHERE Customer.CustomerID IN (1, 2)

ORDER BY [Customer!1!CustomerID], [SalesOrder!2!SalesOrderID]
FOR XML EXPLICIT
```

As you can see, the code has become quite complex and will become increasingly complex as you add additional data to the output. Although this query is perfectly valid in SQL Server 2005, this solution is unacceptable, which leads us to the new and improved FOR XML statement.

FOR XML Enhancements in SQL Server 2005

As you can see, SQL Server 2000 has a lot of XML features, even some not mentioned here (such as viewing the results over HTTP). Just about all of the SQL Server 2000 XML features revolve around FOR XML, a feature that appears to be much underused by developers. Following are a few ways SQL Server 2005 enhances the FOR XML feature:

- Using the TYPE option, FOR XML can output XML (as opposed to streamed results) from a SELECT statement using FOR XML, which in turn allows you to nest the results of SELECT...FOR XML into another SELECT statement.
- The new option, FOR XML PATH, allows you to more easily shape data and produce element-based XML than with the EXPLICIT option.
- You can explicitly specify a ROOT element for your output.
- You can produce element-based XML with XML AUTO.

- FOR XML can produce XML using an XSD Schema that uses a namespace you specify.

Nesting, whitespace, and null handling have been improved.

FOR XML's TYPE Option:

XML is an intrinsic data type of SQL Server 2005, so we can now automatically cast the XML output from the FOR XML query into an XML data type instance, as opposed to streamed results. You accomplish this by using the TYPE keyword after your FOR XML statement, like this:

```
USE AdventureWorks
DECLARE @xmlData AS XML

SET @xmlData =
(Select Customer.CustomerID, OrderDetail.SalesOrderID,
   OrderDetail.OrderDate
From Sales.Customer Customer
   inner join Sales.SalesOrderHeader OrderDetail
   on OrderDetail.customerid=Customer.customerid
WHERE Customer.CustomerID<3
ORDER BY Customer.CustomerID
For XML AUTO, TYPE)--Casts to XML type
SELECT @xmlData
```

This example declares a variable of XML and then sets that variable to a casted result of a FOR XML query using the TYPE statement. The results of this query are shown here for demonstration purposes, but you can use this new XML variable as part of an INSERT statement (to an XML column) or pass it to a stored procedure, as a couple of examples.

```
<Customer CustomerID="1">
  <OrderDetail SalesOrderID="43860" OrderDate="2001-08-01T00:00:00" />
  <OrderDetail SalesOrderID="44501" OrderDate="2001-11-01T00:00:00" />
  <OrderDetail SalesOrderID="45283" OrderDate="2002-02-01T00:00:00" />
  <OrderDetail SalesOrderID="46042" OrderDate="2002-05-01T00:00:00" />
</Customer>
<Customer CustomerID="2">
  <OrderDetail SalesOrderID="46976" OrderDate="2002-08-01T00:00:00" />
  <OrderDetail SalesOrderID="47997" OrderDate="2002-11-01T00:00:00" />
  <OrderDetail SalesOrderID="49054" OrderDate="2003-02-01T00:00:00" />
  <OrderDetail SalesOrderID="50216" OrderDate="2003-05-01T00:00:00" />
  <OrderDetail SalesOrderID="51728" OrderDate="2003-08-01T00:00:00" />
  <OrderDetail SalesOrderID="57044" OrderDate="2003-11-01T00:00:00" />
  <OrderDetail SalesOrderID="63198" OrderDate="2004-02-01T00:00:00" />
  <OrderDetail SalesOrderID="69488" OrderDate="2004-05-01T00:00:00" />
</Customer>
```

You can use FOR XML, TYPE in any valid SQL expression. The next example uses the FOR XML, TYPE syntax as an expression in the SELECT statement.

```
USE AdventureWorks
GO

SELECT
   CustomerID,
   (SELECT SalesOrderID,
                 TotalDue,
                 OrderDate,
                 ShipDate
   FROM Sales.SalesOrderHeader AS OrderHeader
   WHERE CustomerID = Customer.CustomerID
   FOR XML AUTO, TYPE) AS OrderHeaders
FROM Sales.Customer AS Customer
WHERE CustomerID IN (1, 2)
FOR XML AUTO

The results are shown here:
1 <OrderHeader SalesOrderID="43860" TotalDue="14603.7393"
OrderDate="2001-08-
01T00:00:00" ShipDate="2001-08-08T00:00:00" />
<OrderHeader SalesOrderID="44501" TotalDue="26128.8674" OrderDate="2001-
11-
01T00:00:00" ShipDate="2001-11-08T00:00:00" />
<OrderHeader SalesOrderID="45283" TotalDue="37643.1378" OrderDate="2002-
02-
01T00:00:00" ShipDate="2002-02-08T00:00:00" />
<OrderHeader SalesOrderID="46042" TotalDue="34722.9906" OrderDate="2002-
05-
01T00:00:00" ShipDate="2002-05-08T00:00:00" />
2 <OrderHeader SalesOrderID="46976" TotalDue="10184.0774"
OrderDate="2002-08-
01T00:00:00" ShipDate="2002-08-08T00:00:00" />
<OrderHeader SalesOrderID="47997" TotalDue="5469.5941" OrderDate="2002-
11-
01T00:00:00" ShipDate="2002-11-08T00:00:00" />
<OrderHeader SalesOrderID="49054" TotalDue="1739.4078" OrderDate="2003-
02-
01T00:00:00" ShipDate="2003-02-08T00:00:00" />
<OrderHeader SalesOrderID="50216" TotalDue="1935.5166" OrderDate="2003-
05-
01T00:00:00" ShipDate="2003-05-08T00:00:00" />
<OrderHeader SalesOrderID="51728" TotalDue="3905.2547" OrderDate="2003-
08-
01T00:00:00" ShipDate="2003-08-08T00:00:00" />
```

```
<OrderHeader SalesOrderID="57044" TotalDue="4537.8484" OrderDate="2003-
11-
01T00:00:00" ShipDate="2003-11-08T00:00:00" />
<OrderHeader SalesOrderID="63198" TotalDue="4053.9506" OrderDate="2004-
02-
01T00:00:00" ShipDate="2004-02-08T00:00:00" />
<OrderHeader SalesOrderID="69488" TotalDue="908.3199" OrderDate="2004-
05-
01T00:00:00" ShipDate="2004-05-08T00:00:00" />
```

FOR XML PATH:

If you want to create element-based XML, you can use FOR XML PATH to specify column aliases that contain valid XPath expressions that will shape your XML output.

```
USE AdventureWorks
GO
--XML FOR PATH
SELECT TOP 2 --limits result rows for demo purposes
ContactID AS [@Contact_ID],
FirstName AS [ContactName/First],
LastName AS [ContactName/Last],
Phone AS [ContactPhone/Phone1]
FROM Person.Contact FOR XML PATH
```

The output looks like this:

```
<row Contact_ID="1">
 <ContactName>
   <First>Gustavo</First>
   <Last>Achong</Last>
 </ContactName>
 <ContactPhone>
   <Phone1>398-555-0132</Phone1>
 </ContactPhone>
</row>
<row Contact_ID="2">
  <ContactName>
    <First>Catherine</First>
    <Last>Abel</Last>
  </ContactName>
  <ContactPhone>
    <Phone1>747-555-0171</Phone1>
  </ContactPhone>
</row>
```

FOR XML EXPLICIT. Using the TYPE option in conjunction with FOR XML PATH, you can reproduce that awful and complex query with a much simpler version, as shown here:

```
USE AdventureWorks
GO

SELECT
   CustomerID AS [@CustomerID],
   (SELECT SalesOrderID AS [@SalesOrderID],
            TotalDue AS [@TotalDue],
            OrderDate,
            ShipDate,
            (SELECT ProductID AS [@ProductID],
                     OrderQty AS [@OrderQty],
                     LineTotal AS [@LineTotal]
                     FROM Sales.SalesOrderDetail
                     WHERE SalesOrderID = OrderHeader.SalesOrderID
                     FOR XML PATH('OrderDetail'), TYPE)
   FROM Sales.SalesOrderHeader AS OrderHeader
   WHERE CustomerID = Customer.CustomerID
   FOR XML PATH('OrderHeader'), TYPE)
FROM Sales.Customer AS Customer
WHERE CustomerID IN (1, 2)
FOR XML PATH ('Customer')
```

Isn't that much better? This query uses a subselect using the XML PATH statement in conjunction with TYPE to produce element-based XML nested inside a much larger FOR XML PATH statement. This returns each separate Order for the customer as a new child node of the Customer ID node; you can see this in the results of the following query:

```
<Customer CustomerID="1">
  <OrderHeader SalesOrderID="43860" TotalDue="14603.7393">
    <OrderDate>2001-08-01T00:00:00</OrderDate>
    <ShipDate>2001-08-08T00:00:00</ShipDate>
    <OrderDetail ProductID="761" OrderQty="2" LineTotal="838.917800" />
    <OrderDetail ProductID="770" OrderQty="1" LineTotal="419.458900" />
More...
  </OrderHeader>
  <OrderHeader SalesOrderID="44501" TotalDue="26128.8674">
    <OrderDate>2001-11-01T00:00:00</OrderDate>
    <ShipDate>2001-11-08T00:00:00</ShipDate>
    <OrderDetail ProductID="761" OrderQty="1" LineTotal="419.458900" />
    <OrderDetail ProductID="768" OrderQty="3" LineTotal="1258.376700" />
More...
  </OrderHeader>
  <OrderHeader SalesOrderID="45283" TotalDue="37643.1378">
    <OrderDate>2002-02-01T00:00:00</OrderDate>
    <ShipDate>2002-02-08T00:00:00</ShipDate>
    <OrderDetail ProductID="759" OrderQty="1" LineTotal="419.458900" />
    <OrderDetail ProductID="758" OrderQty="3" LineTotal="2624.382000" />
    <OrderDetail ProductID="750" OrderQty="2" LineTotal="4293.924000" />
More...
  </OrderHeader>
  <OrderHeader SalesOrderID="46042" TotalDue="34722.9906">
    <OrderDate>2002-05-01T00:00:00</OrderDate>
    <ShipDate>2002-05-08T00:00:00</ShipDate>
    <OrderDetail ProductID="763" OrderQty="2" LineTotal="838.917800" />
    <OrderDetail ProductID="757" OrderQty="4" LineTotal="3499.176000" />
More...
  </OrderHeader>
</Customer>
More...
```

If you are familiar and comfortable with XPath, you might like some additional XML PATH features. You can use the following XPath node test functions to further control the shape of your XML output:

- data
- comment
- node
- text
- processing-instruction

The following example uses the data and comment methods of XPath. The data method takes the results of the underlying query and places them all inside one element. The comment method takes data and transforms it into an XML comment, as shown in this example:

```
SELECT
   Customer.CustomerID AS [@CustomerID],
   Contact.FirstName + ' ' + Contact.LastName AS [comment()],
   (SELECT SalesOrderID AS [@SalesOrderID],
         TotalDue AS [@TotalDue],
         OrderDate,
         ShipDate,
         (SELECT ProductID AS [data()]
         FROM Sales.SalesOrderDetail
         WHERE SalesOrderID = OrderHeader.SalesOrderID
         FOR XML PATH('')) AS [ProductIDs]
   FROM Sales.SalesOrderHeader AS OrderHeader
   WHERE CustomerID = Customer.CustomerID
   FOR XML PATH('OrderHeader'), TYPE)
FROM Sales.Customer AS Customer
INNER JOIN Sales.Individual AS Individual
   ON Customer.CustomerID = Individual.CustomerID
```

```
INNER JOIN Person.Contact AS Contact
   ON Contact.ContactID = Individual.ContactID
WHERE Customer.CustomerID IN (11000, 11001)
FOR XML PATH ('Customer')
```

The results are as follows; as you can see, the concatenated contact name becomes an XML comment, and the subquery of Product IDs is transformed into one element:

```
<Customer CustomerID="11000">
  <!--Jon Yang-->
  <OrderHeader SalesOrderID="43793" TotalDue="3756.9890">
    <OrderDate>2001-07-22T00:00:00</OrderDate>
    <ShipDate>2001-07-29T00:00:00</ShipDate>
    <ProductIDs> 966 934 923 707 881</ProductIDs>
  </OrderHeader>
More...
</Customer>
<Customer CustomerID="11001">
  <!--Eugene Huang-->
  <OrderHeader SalesOrderID="43767" TotalDue="3729.3640">
    <OrderDate>2001-07-18T00:00:00</OrderDate>
    <ShipDate>2001-07-25T00:00:00</ShipDate>
    <ProductIDs> 779 878 870 871 884 712</ProductIDs>
  </OrderHeader>
More...
</Customer>
```

Specifying a ROOT Element:

The ROOT option allows you to add a main, or root, element to your FOR XML output. You can combine this with other FOR XML keywords, as shown here:

```
USE AdventureWorks
GO
--Root
SELECT Customer.CustomerID,
   OrderDetail.SalesOrderID, OrderDetail.OrderDate
FROM Sales.Customer AS Customer
INNER JOIN Sales.SalesOrderHeader OrderDetail
      ON OrderDetail.customerid=Customer.customerid
WHERE Customer.CustomerID<20
ORDER BY Customer.CustomerID
```

FOR XML AUTO, ROOT ('Orders')
The output looks like this:

```
<Orders>
  <Customer CustomerID="1">
    <OrderDetail SalesOrderID="43860" OrderDate="2001-08-01T00:00:00" />
    <OrderDetail SalesOrderID="44501" OrderDate="2001-11-01T00:00:00" />
    <OrderDetail SalesOrderID="45283" OrderDate="2002-02-01T00:00:00" />
    <OrderDetail SalesOrderID="46042" OrderDate="2002-05-01T00:00:00" />
  </Customer>
...more...
</Orders>
```

The code output here is the same as any FOR XML AUTO output for this query, except that the XML ROOT we specified with the ROOT keyword now surrounds the data. In this example, we used ROOT ('Orders'), so our output is surrounded with an <Orders> XML element.

OPENXML Enhancements in SQL Server 2005

Up until now, we have been composing XML from rows of data, but what if we already have XML data and we want to shred it back into relational data? Well, SQL Server 2000 introduced a feature for this purpose called OPENXML. OPENXML is a system function that allows an XML document to be shredded into T-SQL rows. SQL Server 2005 also has the OPENXML functionwith some enhancements, of course.

To shred data into relational rows using OPENXML, you must first create an XML document handle using the system stored procedure sp_xml_preparedocument. This system stored procedure takes an XML document and creates a representation that is referenced via a handle, which it returns via an OUTPUT parameter. OPENXML uses this handle along with a specified path and behaves like a database view to the XML data, so you simply choose SELECT from the OPENXML function just as you would SELECT from a table or a view. The following code shows an example of OPENXML in action.

```
USE AdventureWorks

DECLARE @int int
DECLARE @xmlORDER varchar(1000)
SET @xmlORDER ='
<ROOT>
<Customer CustomerID="BRU" ContactName="Andrew Brust">
   <Order CustomerID="BRU" EmployeeID="5" OrderDate="2005-11-04">
      <OrderDetail OrderID="10248" ProductID="16" Quantity="12"/>
      <OrderDetail OrderID="10248" ProductID="32" Quantity="10"/>
   </Order>
</Customer>
<Customer CustomerID="ZAC" ContactName="Bill Zack">
   <Order CustomerID="ZAc" EmployeeID="3" OrderDate="2005-11-16">
      <OrderDetail OrderID="10283" ProductID="99" Quantity="3"/>
   </Order>
```

```
</Customer>
</ROOT>'
--Create an internal representation of the XML doc
EXEC sp_xml_preparedocument @int OUTPUT, @xmlORDER
-- OPENXML rowset provider.
SELECT    *
FROM OPENXML (@int, '/ROOT/Customer',1)
WITH (CustomerID  varchar(10),ContactName varchar(20))
```

The code here takes the XML text and allows you to query and work with it as if it were relational data. The output looks like this:

```
CustomerID     ContactName
---------------------------
BRU            Andrew Brust
ZAC            Bill Zack
(2 row(s) affected)
```

You can optionally specify if you want OPENXML to use element-based or attribute-based XML relational mapping between the rowset columns and the XML nodes. There are two ways to control the mapping. The first is to use the flags parameter, which assumes that the XML nodes will map to corresponding rowset columns with exactly the same name. You can also use the ColPattern parameter, an XPath expression that allows you to use a schema to perform the mapping as part of SchemaDeclaration in the WITH clause. The mapping specified in ColPattern overwrites the mapping specified by the flags parameter.

SQL Server 2005 introduces two enhancements to OPENXML, both involving the new XML data type. First, the XML data type is supported as an output column or an overflow column with the OPENXML statement. Second, you can pass an XML data type variable directly into sp_xml_preparedocument. Both of these enhancements enable you to more easily work with existing XML data in an XML column or created data using FOR XML TYPE.

Selection Logic: FLWOR Expressions

Just as SELECT, FROM, WHERE, GROUP BY, and ORDER BY form the basis of SQL's selection logic, the **for, let, where, order by, and return** (FLWOR) keywords form the basis of every XQuery query you write. You use the for and let keywords to assign variables and iterate through the data within the context of the XQuery. (The let keyword is not supported in the SQL Server 2005 implementation of XQuery.) The where keyword works as a restriction and outputs the value of the variable. For example, the following basic XQuery uses the XPath expression /catalog/book to obtain a reference to all the <book> nodes, and the for keyword initiates a loop, but only of elements where the category attribute is equal to "ITPro". This simple code snippet iterates through each /catalog/book node using the $b variable with the for statement only where the category attribute is "ITPro" and returns as output the resulting information ordered by the author's name using the order keyword.

```
for $b in /catalog/book
where $b/@category="ITPro"
   order by $b/author[1] descending
   return ($b)
```

Armed with this basic knowledge of XQuery expressions, you are ready to see it in action. Here is a simple example that uses this XQuery expression on an XML data type variable. We assign the XML to the variable and then use the preceding XQuery expression in the query() method (explained in the next section) of the XML data type.

```
DECLARE @XML xml
Set @XML='<catalog>
  <book category="ITPro">
    <title>Windows Step By Step</title>
    <author>Bill Zack</author>
    <price>49.99</price>
  </book>
  <book category="Developer">
    <title>Developing ADO .NET</title>
    <author>Andrew Brust</author>
    <price>39.93</price>
  </book>
  <book category="ITPro">
    <title>Windows Cluster Server</title>
    <author>Stephen Forte</author>
    <price>59.99</price>
  </book>
</catalog>
'
Select @XML.query('for $b in /catalog/book
   where $b/@category="ITPro"
   order by $b/author[1] descending
   return ($b)')
```

The results are as follows. Notice that Stephen's record is first because our order is descending by the author element. Andrew's record is not in the output because we are restricting only for "ITPro" in the category element.

```
<book category="ITPro">
  <title>Windows Cluster Server</title>
  <author>Stephen Forte</author>
  <price>59.99</price>
</book>
<book category="ITPro">
  <title>Windows Step By Step</title>
  <author>Bill Zack</author>
  <price>49.99</price>
</book>
```

XQuery

SQL Server 2005 has a standards-based implementation of XQuery that directly supports XQuery functions on the XML data type. It supports XQuery by using five methods of the XML data type:

- xml.exist() uses XQuery input to return 0, 1, or NULL, depending on the result of the query. It returns 0 if no elements match, 1 if there is a match, and NULL if there is no XML data on which to query. This method is often used for query predicates.
- xml.value() accepts an XQuery as input and returns an SQL Server scalar type.
- xml.query() accepts an XQuery as input and returns an XML data type stream as output.
- xml.nodes() accepts an XQuery as input and returns a single-column rowset from the XML document. In essence, it shreds XML into multiple smaller XML results.
- xml.modify() allows you to insert, delete, or modify nodes or sequences of nodes in an XML data type instance using XQuery Data Manipulation Language.

We will discuss the methods of the XML data type shortly. But first we must create some sample data. We will create a simple table that contains speakers at a software developers conference and the corresponding classes they will teach. Usually you normalize the data and have a one-to-many relationship between a speakers table and the classes table. Instead of using an additional normalized table, we will model this as one table with the speaker's information and one XML column with the speaker's classes. In the real world, you might encounter this scenario when in a back-office database you have the speaker and his classes represented in a series of one-to-many tables. Then for the Web database, you might "publish" a database on a frequent time interval (like a reporting database) or transform normalized data and use the XML column for easy HTML display (or XSLT transformations).

We first create an XSD schema (for reasons that will soon become clear) for our XML column. This schema will define the type of XML allowed in the column, including the XML data types and required properties for particular XML elements:

```
use AdventureWorks
go

CREATE xml schema collection dbo.classes_xsd
As
 '<?xml version="1.0" encoding="UTF-8" ?>
  <xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:element name="class">
  <xs:complexType>
  <xs:attribute name="name" type="xs:string" use="required" />
  </xs:complexType>
  </xs:element>
  <xs:element name="classes">
  <xs:complexType>
  <xs:sequence>
  <xs:element ref="class" maxOccurs="unbounded" />
  </xs:sequence>
  </xs:complexType>
  </xs:element>
  </xs:schema>'
```

Next we create our table, tblSpeakers. Notice that the XML column, Speaker_XML, uses the classes_xsd XSD schema described earlier.

```
Create Table tblSpeakers
(
Speaker_ID Integer Primary Key Identity,
Speaker_NM nVarChar(50),
Speaker_Country nVarChar(25),
Speaker_XML XML (classes_xsd) Not Null)
```

XQuery runs more efficiently when there is an XML index on the XML column. As you learned earlier, an XML index works only if there is a primary key constraint on the table (which we have). The code here creates a primary and then a structural (PATH) index because our examples do a lot of where restrictions on the values of particular elements.

```
--XML Index: Primary
CREATE Primary XML INDEX idx_1
   ON tblSpeakers (Speaker_XML)
--PATH
CREATE XML INDEX idx_a
   ON tblSpeakers (Speaker_XML)
USING XML INDEX idx_1 FOR PATH
```

Now that we have our index, remember that XQuery works more efficiently if it is strongly typed, so you should always use an XSD schema on your XML column for the best performance. Without an XSD schema, the SQL Server 2005 XQuery engine assumes that everything is untyped and treats it as string data.

Lastly, we need to get some data into the table by using some T-SQL INSERT statements. The last INSERT, 'Bad Speaker', will fail because it violates the classes_xsd schema and does not contain a <classes> element:

```
Insert into tblSpeakers Values('Stephen Forte', 'USA',
'
<classes>
   <class name="Writing Secure Code for ASP .NET "/>
   <class name="Using XQuery to Query and Manipulate XML Data in SQL
Server 2005"/>
   <class name="SQL Server and Oracle Working Together"/>
   <class name="Protecting against SQL Injection Attacks "/>
   </classes>
'
)

Insert into tblSpeakers Values('Richard Campbell', 'Canada',
'
<classes>
   <class name="SQL Server Profiler"/>
   <class name="Advanced SQL Querying Techniques"/>
   <class name="SQL Server and Oracle Working Together"/>
   <class name="T-SQL Error Handling in Yukon"/>
</classes>
'
)

Insert into tblSpeakers Values('Tim Huckaby', 'USA',
'
<classes>
   <class name="Smart Client Stuff"/>
   <class name="More Smart Client Stuff"/>
</classes>
'
)

Insert into tblSpeakers Values('Malek Kemmou', 'Morocco',
'
<classes>
   <class name="SmartPhone 2005"/>
   <class name="Office System 2003"/>
</classes>
'
)

Insert into tblSpeakers Values('Goksin Bakir', 'Turkey',
'
<classes>
   <class name="SmartPhone 2005"/>
   <class name="Office System 2003"/>
</classes>
'
)

Insert into tblSpeakers Values('Clemens F. Vasters', 'Germany',
'
<classes>
   <class name="SOA"/>
   <class name="FABRIQ"/>
</classes>
'
)
```

```
Insert into tblSpeakers Values('Kimberly L. Tripp', 'USA',
'
<classes>
   <class name="SQL Server Index"/>
   <class name="SQL Precon"/>
</classes>
'
)

Insert into tblSpeakers Values('Bad Speaker', 'France',
'
<CLASSES>
   <class name="SQL Server Index"/>
   <class name="SQL Precon"/>
</CLASSES>
'
)
```

Now that we have our data, it is time to start writing some XQuery expressions in SQL Server 2005. To do this, we will use the aforementioned methods of the XML data type inside a regular T-SQL query.

xml.exist(): Having XML in the database is almost useless unless you can query the elements and attributes of the XML data natively. XQuery becomes very useful when you can use it to search based on the values of a particular element or attribute. The xml.exist() function accepts an XQuery as input and returns 0, 1, or NULL, depending on the result of the query; 0 is returned if no elements match, 1 is returned if there is a match, and NULL is returned if there is no data to query on. For example, we will see if a node exists in this particular XML string of classes.

```
DECLARE @XML xml
Set @XML='
<classes>
   <class name="SQL Server Index"/>
   <class name="SQL Precon"/>
</classes>
'
Select @XML.exist('/classes')
```

The code returns 1 because the "classes" element exists in the XML variable. If you change the XQuery expression to search for an XML node that does not exist (Select @XML.exist ('/dogs), for example), it will return 0. You can see this in action as part of a CHECK CONSTRAINT. SQL Server does not allow you to use an xml.exist as part of a CHECK CONSTRAINT. You have to first create a user-defined function (UDF) to perform the action. This UDF accepts an XML field and returns the value of an xml.exist() method looking for an instance of <Orders>:

```
USE AdventureWorks
GO
```

```
CREATE FUNCTION dbo.DoesOrderXMLDataExist
(@XML XML)
RETURNS bit
AS
BEGIN
RETURN @XML.exist('/Orders')
END;
GO
```

To use this UDF as a CHECK CONSTRAINT, just create a table and pass to the UDF you just created the column you want to apply the constraint to.

```
--create the table using the function
CREATE TABLE OrdersXMLCheck
   (OrderDocID INT PRIMARY KEY,
   xOrders XML NOT NULL Default '<Orders/>'
 CONSTRAINT xml_orderconstraint
  CHECK(dbo.DoesOrderXMLDataExist(xOrders)=1))
```

You will most likely use the return value of xml.exist() (0, 1, or NULL) as part of a T-SQL WHERE clause. Think about it: You can run a T-SQL query and restrict the query on a value of a particular XML element! Going back to our main example, let's look for the value of 'SQL Server and Oracle Working Together' in the <class> element. Here is the XQuery expression to do this:

```
/classes/class[@name="SQL Server and Oracle Working Together"]
```

This is how you put it to work:

```
Select * From tblSpeakers
Where Speaker_XML.exist('/classes/
class[@name="SQL Server and Oracle Working Together"]')=1
```

The results look like this:

```
1  Stephen Forte    USA     < classes>data</classes>
2  Richard Campbell Canada  <classes/>data</classes>
```

The XML returned in these results look like this for Stephen:

```
<classes>
  <class name="Writing Secure Code for ASP .NET " />
  <class name="Using XQuery to Query and Manipulate XML Data in SQL
Server 2005" />
  <class name="SQL Server and Oracle Working Together" />
  <class name="Protecting against SQL Injection Attacks " />
</classes>
```

xml.value(): The xml.value() function takes a valid XQuery expression and returns an SQL Server scalar value that you specify. For example, let's say you have an XML column and inside of a T-SQL query you want to return some data from the XML as an intrinsic SQL data type. You call the xml.value() function on that column by passing in the XQuery expression and the data type you want to convert the output to. This requires you to know and understand the data in your XML column. Here's an example of the syntax against our current XML document in the database:

```
xml.value('/classes[1]/class[1]/@name', 'varchar(40)')
```

This XQuery contains an XPath expression that navigates the first class's name attribute and a cast to varchar(40).

You must perform an XQuery expression on an XML column as part of a regular T-SQL query, as shown here. What is cool is that SQL Server combines both relational queries and XQuery in one query because in this example we use a traditional T-SQL WHERE clause to show only speakers from the USA.

```
USE AdventureWorks
select Speaker_ID, Speaker_NM,
Speaker_Country,
   Speaker_XML.value('/classes[1]/class[1]/@name', 'varchar(40)') as
Sessions
From tblSpeakers
where speaker_country ='USA'
```

The results are shown here:

```
Speaker_ID     Speaker_NM       Speaker_Country      Session
--------------------------------------------------------------------------
------------
1              Stephen Forte    USA                  Writing Secure Code
for ASP .NET
3              Tim Davis        USA                  Smart Client Stuff
7              Kimberly Smith   USA                  SQL Server Index

(3 row(s) affected)
```

Let's dissect the XQuery expression. As you'll recall from our earlier listing, the XML for Stephen looks like this:

```
<classes>
   <class name="Writing Secure Code for ASP .NET "/>
   <class name="Using XQuery to Query and Manipulate XML Data in SQL
Server 2005"/>
   <class name="SQL Server and Oracle Working Together"/>
   <class name="Protecting against SQL Injection Attacks "/>
</classes>
```

The following XQuery path expression returns the value of the first class's name attribute (Writing Secure Code for ASP.NET) as a varchar(40). So in the preceding query, the XQuery expression is placed in the value() method of the XML column (Speaker_XML in our T-SQL query). The following XQuery expression does all the work of getting the first class's name:

/classes[1]/class[1]/@name

This approach is useful when you want to pull standard data out of the XML column and display it as regular scalar SQL Server data.

xml.query():

The xml.query() function works much like the xml.value() function, except it returns an XML data type value, so you have a lot more flexibility. It is useful only if you want the end result of the column in the query to be XML; if you want scalar data, use xml.value().

If you want to return the same data as in the previous example, but you want to present the summary column in XML format, run the exact same query except with the xml.query():

```
--xml.query
--returns XML data type
--same as previous example but returns XML
select Speaker_ID, Speaker_NM,
Speaker_Country,
   Speaker_XML.query('/classes[1]/class[1]') as Sessions
From tblSpeakers
where speaker_country ='USA'
```

XML.query() works by passing in an XQuery expression that will result in XML output. The XQuery expression can return a single element, all the elements, or use a RETURN (the R in FLOWR) expression to completely transform the results. In this example, the first instance of the class element is returned. The results of the "sessions" column for Stephen's records are the same as in the previous example except that the Sessions column is now formatted in XML:

```
<class name="Writing Secure Code for ASP .NET " />
```

Instead of using xml.value(), you can return a larger XML result of many nodes by leaving out the [1] ordinal position indicators in your path expression.

```
--same as previous but returns all
select Speaker_ID, Speaker_NM,
Speaker_Country,
   Speaker_XML.query('/classes/class') as Sessions
From tblSpeakers
where speaker_country ='USA'
```

The results are the same except for Stephen's classes; we get all of the XML results as an XML column:

```
<class name="Writing Secure Code for ASP .NET " />
<class name="Using XQuery to Query and Manipulate XML Data in SQL Server
2005" />
<class name="SQL Server and Oracle Working Together" />
<class name="Protecting Against SQL Injection Attacks" />
```

You can gain further control over your XQuery path expression by using the FLOWR expressions. For example, you can write an expression like this:

```
for $b in /classes/class
return ($b)
```

The expression uses the for and return keywords to loop through all of the class elements and return the values, and it yields the same results as the preceding example, which has '/classes/class' as its XQuery expression. (Of course, you can come up with much more interesting examples.) You can incorporate this expression into your T-SQL query:

```
Select Speaker_ID, Speaker_NM, Speaker_Country, Speaker_XML.query('
   for $b in /classes/class
   return ($b)
   ') As Sessions
From tblSpeakers
```

Let's say you want to have more control over the XML output using xml.query() or xml.value() as well as combining the XQuery and a traditional T-SQL WHERE clause to show only the speakers in the United States. You can use an XML method in both the SELECT and WHERE clauses of a single query:

```
Select Speaker_ID, Speaker_NM, Speaker_Country,
Speaker_XML.query('/classes/
class') As Sessions
From tblSpeakers
Where Speaker_Country='USA'
and Speaker_XML.exist('/classes/
class[@name="SQL Server and Oracle Working Together"]')=1
```

The T-SQL WHERE clause restricts on the Speaker_Country column, and our XQuery expression filters only for the <class> element we are interested in.

```
<class name="Writing Secure Code for ASP .NET " />
<class name="Using XQuery to Query and Manipulate XML Data in SQL Server
2005" />
<class name="SQL Server and Oracle Working Together" />
<class name="Protecting against SQL Injection Attacks " />
```

xml.nodes(): The xml.nodes() method takes an XQuery expression just like exist(), value(), and query(). xml.nodes() then returns instances of a special XML data type, each of which has its context set to a different node that the XQuery expression you supplied evaluates to. The special XML data type is used for subqueries of XML data in your result set.

Summary

- Reading mode used In FOR XML clause
    - RAW
    - AUTO
    - EXPLICIT
    - PATH
- XQuery is a language that can query structured or semi-structured XML data

- Methods used to query or modify XML data.
  - `query()`
  - `value()`
  - `exists()`
  - `modify()`
  - `nodes()`

References

Websites

- http://msdn.microsoft.com/SQL/

Books

- MSDN online Help
- Implementing a Microsoft SQL Server 2005 Database (2779A) Microsoft Offical Curriculum

STUDENT NOTES: